TP 1 d'Interprétation et compilation

Christian Rinderknecht

23 et 24 septembre 2003

Le but de ce TP est de réaliser la calculette fonctionnelle présentée lors du premier cours et de l'étendre. Un certain nombre d'éléments vous sont fournis sous la forme de fichiers (analyseur lexical et syntaxique, un début de programme principal et un `Makefile`). À vous de construire l'évaluateur.

Ouvrez l'archive `tp1.tgz`, et construisez la calculette initiale avec `make calc`. L'exécutable produit s'appelle `calc`, et les fichiers à modifier et compléter sont `eval.ml` et `main.ml`. Les autres fichiers fournis sont les suivants (vous n'avez pas besoin de les modifier) :

- `ast.ml` : le type des arbres de syntaxe abstraite ;
- `lexer.mll` : la spécification de l'analyseur lexical. Sa compilation par ocamllex produit `lexer.ml` (l'analyseur lexical).
- `parser.mly` : le spécification de la syntaxe concrète (la grammaire). Sa compilation par ocamlyacc produit `parser.mli` (interface spécifiant le type des lexèmes et le type de l'analyseur syntaxique) et `parser.ml`, contenant l'analyseur syntaxique proprement dit.

Le fonctionnement global de la calculette est le suivant :

1. L'analyseur lexical Lexer exporte une fonction token qui renvoie le premier lexème reconnaissable à partir de l'entrée standard (par défaut il s'agit du clavier) ;
2. l'analyseur syntaxique Parser exporte une fonction expression qui prend en argument Lexer.token et l'entrée standard, et renvoie l'arbre de syntaxe abstraite qui doit être évalué ;
3. l'évaluateur Eval exporte une fonction eval qui prend en argument un environnement et l'arbre de syntaxe abstraite correspondant à l'expression, et renvoie la valeur associée ou déclenche une exception en cas d'erreur ;
4. le pilote Main invite l'usager à saisir une expression, lance l'analyse lexico-syntaxique puis l'évaluation (dans un environnement vide) ; il affiche ensuite le résultat. En cas d'erreur, il affiche l'erreur dont le message est associé aux exceptions provenant de l'évaluateur.

Pour l'instant, l'évaluateur ne fait que renvoyer la chaîne `"ok"` quelque soit l'expression (attention, le rôle de l'évaluateur n'est pas de faire des affichages : c'est un des rôles du pilote), et le pilote passe effectivement l'AST à l'évaluateur, mais il ne gère aucune erreur provenant de celui-ci.
Le travail à faire est le suivant :

1. Implantez la calculette présentée dans le cours sans tenir compte des cas d'erreurs, puis traitez ceux-ci à l'aide des exceptions O'Caml.
2. Ajoutez maintenant la liaison locale et les variables. Identifiez les erreurs possibles et ajoutez leur gestion. Testez des cas d'évaluation correcte et des cas provoquant ces erreurs. (*Gardez une trace de vos cas de test.*) Par exemple :

   ```
   1+2*(ifz 0 then let x = 0 in x+3 else 4)
   ```

3. On souhaite se doter d'une conditionnelle *sans ajout des booléens*, dans le style du langage C :
   - **Syntaxe concrète**
     ```
     Expression ::=  ...
       | "ifz" Expression "then" Expression "else" Expression
     ```
   - **Exemple** `1+2*(ifz 0 then 3 else 4)`
   - **Syntaxe abstraite**
     type *expr* = ... | Ifz **of** *expr* × *expr* × *expr*;;

   Donnez une sémantique opérationnelle pour cette conditionnelle et complétez votre évaluateur par son implantation.
4. Enrichissez la calculette par les booléens, les opérateurs logiques de conjonction et disjonction, ainsi que d'une conditionnelle dont la condition est booléenne. Un lexique et une grammaire sont respectivement proposés dans les fichiers lexer.mll et parser.mly. Proposez une sémantique opérationnelle et une implantation en O'Caml.
5. Le codage fonctionnel pour les environnements est très inefficace car, dans le pire des cas, pour obtenir la valeur d'une variable il faut évaluer autant d'appels de fonction qu'il y a de variables dans l'environnement, et un appel de fonction est très coûteux, en général. Il est plus judicieux, dans un premier temps, de remplacer les appels de fonction par des accès dans une liste. Ainsi un environnement est une liste qui associe les variables aux valeurs. La recherche d'une valeur se fait alors à l'aide de la fonction List.assoc : $\forall\alpha,\beta.\alpha \to (\alpha\times\beta)\,\mathsf{list} \to \beta$. Conservez un exemplaire de `eval.ml` et réécrivez-le en implantant ce schéma et testez à nouveau. (*Gardez une trace de vos cas de test.*)