TP n°8

Menhir

Grammaire pour des expressions booléennes

Le but de ce TP est de comprendre le fonctionnement d'un analyseur syntaxique pour une grammaire simple et de la modifier ensuite.
Récupérer les fichiers fournis sur Moodle. Voici une définition de certaines expressions booléennes.

$$
\begin{array}{rcl}
e & ::= & \texttt{id} \\
& | & \texttt{true} \\
& | & \texttt{false} \\
& | & e \vee e \\
& | & \texttt{(}e\texttt{)}
\end{array}
$$

où e est un non-terminal et `id`, `true`, `false`, $\vee$, `(` et `)` sont des terminaux.

L'analyseur fourni produit un arbre de syntaxe abstrait (dont le type est défini dans `ast.ml`) en lisant une entrée sur l'entrée standard. L'analyseur lexicale (fourni dans `lexeur.mll`) produit un flux de tokens. Dans `parser.mly` il y a la définition de la grammaire au format menhir et dans `main.ml` un programme principal qui initialise le lexeur et invoque le parseur généré par menhir sur l'entrée standard en passant par le lexeur. Le résultat est un arbre de syntaxe abstrait qui est imprimé sur l'écran (avec des parenthèses pour refléter sa structure). Un fichier `dune` est fourni qui permet de compiler les fichiers avec `dune build` pour obtenir un `main.exe` exécutable dans le répertoire `_build/default`. On peut tester `main.exe` en l'exécutant et en tapant sur l'entrée standard une expression qu'on termine avec `Ctrl-D`. On peut également tester `main.exe` en utilisant un fichier contenant une expression (`exemple.exp` est fourni) et en invoquant `_build/default/main.exe < exemple.exp`

Quand on compile pour la première fois menhir signale un conflit. Attention : menhir donne un `warning` et produit quand même un parseur qui ne fait pas forcément ce qu'on attend. La première fois un fichier expliquant les conflits est créé. **Ce fichier disparaît à la deuxième compilation.** On peut toujours recommencer en faisant `dune clean` et `dune build`.

Exercice 1 *Regarder le fichier* `_build/default/parser.conflicts` *et donner deux dérivations droites différentes du mot* **true** $\vee$ **true** $\vee$ **true** *qui illustrent le problème.*

Ici, le conflit est entre shifter `OR` ou réduire par une règle qui a `OR` comme dernier terminal. Dans ce cas, on peut résoudre le conflit en indiquant soit que le `OR` est associatif à gauche (avec la directive `%left OR`) soit qu'il est associatif à droite (avec la directive `%right OR`).

Exercice 2 *Ajouter la directive* `%right OR` *et compiler. Parser "*`true \/ x \/ y`*". Changer ensuite en* `%left OR`*, compiler et parser. Observer la différence. Quelle version est préférable ?*

On souhaite ajouter la conjonction aux expressions en ajoutant à la grammaire la règle $e ::= e \wedge e$.

Exercice 3 *Ajouter ce qu'il faut pour traiter la conjonction. Indication : il faut modifier* `ast.ml`, `lexer.mll` *et* `parser.mly`.

Exercice 4 *Résoudre les nouveaux conflits et vérifier que* `x \/ y /\ z` *est correctement traité.*

On souhaite maintenant ajouter aux expressions des définitions très simples de la forme

```
let x = e in e
```

comme par exemple `let x = true \/ false in x /\ true`

Exercice 5 *Modifier la grammaire pour cela et modifier les fichiers en conséquence. Attention aux éventuels conflits.*

On souhaite ajouter des définitions simultanées de la forme

```
let x = e and .... and y = e in e
```

comme par exemple `let x = true /\ true and y = false in x \/ y`

Exercice 6 *Modifier la grammaire pour cela et modifier les fichiers en conséquence. Indication : regarder* `separated_nonempty_list` *dans le manuel de menhir.*

Exercice 7 *Compléter la fonction* `eval` *de* `main.ml` *et la tester. Ici, la valeur d'une variable est arbitrairement fixée à* `false`.