TP analyse syntaxique avec `ocamllex` et `menhir`

Exercice 1 (Branchements conditionnels)
On considère la grammaire `menhir` ci-contre, qui représente des séquences d'expressions pouvant faire intervenir des branchements conditionnels. Les non terminaux sont le symbole de départ `prog` (un programme complet), `seq` (une séquence d'expressions séparées par des `;`) et `expr` (une expression). Les terminaux sont `EOF` (la fin de fichier), `SEMI` (le séparateur `;`), `IF` et `ELSE` (pour les branchements conditionnels), ainsi que `ATOM` et `COND` (pour représenter des expressions atomiques et des conditions qui ne seront pas détaillées ici).

```
%token ATOM COND IF ELSE SEMI EOF
%start <unit> prog
%%

prog:
| seq EOF {}
;

seq:
| expr {}
| seq SEMI expr {}
;

expr:
| ATOM {}
| IF COND seq {}
| IF COND seq ELSE seq {}
;
```

0. Vous devez au préalable récupérer cette grammaire sur la page du cours (fichier `if.mly`) et la compiler avec l'option bavarde (`menhir -v --infer`).
1. Dessiner l'automate décrit dans le fichier `if.automaton`.
2. Pour chacun des trois conflits de cette grammaire, donner une entrée aboutissant à ce conflit, et des arbres de dérivation justifiant les différentes possibilités. Vous pouvez vous aider du fichier `if.conflicts`.
3. Affecter des priorités à certaines règles pour obtenir les comportements suivants :
   — Un `ELSE` est toujours associé au dernier `IF` rencontré.
   — Un `SEMI` clos toutes les expressions commencées.

□

Exercice 2 (Appels de fonctions)
On veut définir un langage qui permet d'écrire des appels de fonction à la fois en style Caml :

```
f e1 e2 ... eN
```

et en style C/Java :

```
f(e1, e2, ..., eN)
```

On se donne pour cela la grammaire `menhir` ci-contre. L'appel de fonction curryfié (à la Caml) est représenté par la production `expr -> ID nonempty_list(simple_expr)` où `nonempty_list(simple_expr)` désigne une suite non vide d'expressions simples, et l'appel de fonction décurryfié (à la C/Java) est représenté par la production expr -> ID tuple où le non terminal `tuple` désigne un n-uplet, formé par une suite éventuellement vide et délimitée par des parenthèses d'expressions séparées par des virgules (`COMMA`). La notion d'expression simple (non terminal `simple_expr`) isole les expressions clairement délimitées : les identifiants `ID`, les expressions entre parenthèses, et les n-uplets.
Les constructions `nonempty_list` et `separated_list` sont fournies par la bibliothèque standard de `menhir` et permettent d'écrire de manière compacte des suites d'éléments (alternativement, on aurait pu utiliser la grammaire donnée à la fin de cette section pour rester compatible avec des outils moins riches tels `ocamlyacc`).

```
%token ID LPAR RPAR COMMA EOF
%start <unit> prog
%%

prog:
| expr EOF {}
;

expr:
| simple_expr {}
| ID nonempty_list(simple_expr) {}
| ID tuple {}
;

simple_expr:
| ID {}
| LPAR expr RPAR {}
| tuple {}
;

tuple:
| LPAR separated_list(COMMA, expr) RPAR {}
;
```

0. Vous devez au préalable récupérer cette grammaire sur la page du cours (fichier `fcall.mly`) et la compiler avec l'option bavarde (`menhir -v --infer`).
1. Pour chacun des deux états présentant un conflit dans cette grammaire, donner une entrée aboutissant à ce conflit, et des arbres de dérivation justifiant les différentes possibilités. Vous pouvez vous aider des fichiers `fcall.conflicts` et `fcall.automaton`
2. Modifier la grammaire pour résoudre ces conflits.

□

Écriture alternative de la grammaire. Sur la page du cours, le fichier `fcall2.mly` donne une écriture alternative de la grammaire qui explicite le contenu de `nonempty_list` et `separated_list`. On y utilise deux non-terminaux additionnels `simple_expr_list` pour les suites d'expressions simples données en argument lors d'un appel de fonction curryfié, et `expr_comma_separated_list` pour les suites d'expressions séparées par des virgules (`COMMA`) utilisée dans la définition des n-uplets. Par rapport à la grammaire précédente, on a aussi prévu une deuxième production pour `tuple` pour traiter le cas vide.

Exercice 3 (Analyse syntaxique pour IMP++) Cet exercice vise à construire un analyseur syntaxique complet pour une version étendue de notre noyau de langage impératif IMP. Un programme IMP++ est formé par une série de déclarations de variables globales suivie d'une série de définitions de fonctions, où l'on suppose que l'une des fonctions est nommée `main`.

— Une déclaration de variable globale est introduite par le mot clé `var`, et fournit optionnellement une valeur initiale.

```
var taille = 64;
```

— Une définition de fonction est introduite par le mot clé `function` et précise les noms des paramètres.

```
function pgcd(a, b) { ... }
```

— Le corps d'une fonction est composé d'une série de déclarations de variables locales suivie d'une séquence d'instructions. Les déclarations de variables locales obéissent à la même syntaxe que les déclarations de variables globales.

— Les instructions comportent :
  — l'affichage d'un caractère donné par son code ASCII : `print(e);`
  — l'affectation d'une nouvelle valeur à une variable : `x = e;`
  — le branchement conditionnel : `if (c) { s1 } else { s2 }`
  — la boucle : `while (c) { s }`
  — le retour d'une fonction : `return e;`

  en outre, toute expression peut être utilisée comme une instruction (en particulier les appels de fonctions).

— Les expressions comportent :
  — les constantes entières et booléennes : `42`, `true`, ...
  — des opérations arithmétiques et logiques : `+`, `*`, `-`, `/`, `%`, `<`, `<=`, `>`, `>=`, `==`, `!=`, `||`, `&&`, `!`
  — l'accès à une variable : `x`
  — l'appel d'une fonction : `f(e1, ..., eN)`
  — la création d'un nouveau tableau : `{e1; ... eN;}`
  — l'accès à un élément d'un tableau : `t[e]`
  — l'affectation à une case d'un tableau : `t[e1] = e2`

Vous trouverez sur la page du cours une archive `impcat.zip` contenant les éléments suivants :

Fichier	**Contenu**	**Langage**	**Commentaire**
`imp.ml`	définition de la syntaxe abstraite	`ocaml`	
`implexer.mll`	analyse lexicale	`ocamllex`	*à compléter*
`impparser.mly`	analyse syntaxique	`menhir`	*à compléter*
`imppp.ml`	afficheur	`ocaml`	
`impcat.ml`	programme principal	`ocaml`	
`dune`, `dune-project`	configuration		
`tests` (dossier)	échantillons de programmes à analyser	`IMP`	*à compléter*

Votre objectif principal est de compléter les fichiers d'analyse lexicale `implexer.mll` et d'analyse syntaxique `impparser.mly`. La définition de la syntaxe abstraite est fournie, assortie d'un *pretty-printer* (une fonction qui affiche « joliment » le programme représenté par un AST) et d'une fonction principale qui analyse un fichier source `file.imp` et écrit le résultat de son analyse dans un fichier `file.cat.imp`. Avec les fichiers de configuration fournis, vous pouvez compiler l'ensemble du programme avec la commande

```
dune build
```

puis exécuter le programme `impcat` sur l'un des échantillons de programmes IMP fournis avec

```
./impcat.exe tests/min.imp
```

Les fichiers fournis, dans leur état actuel, permettent d'analyser le programme minimal `tests/min.imp`. Nous vous recommandons de progresser incrémentalement, selon le cycle suivant :

1. choisir un élément de la syntaxe pas encore traité,
2. ajouter dans `imparser.mly` les nouvelles déclarations de lexèmes nécessaires,
3. étendre implexer.mll avec la reconnaissance de ces nouveaux lexèmes,
4. étendre les règles de `impparser.mly` pour reconnaître le nouvel élément,
5. observer les éventuels avertissements de `menhir` et régler les conflits par l'ajout de priorités le cas échéant,
6. tester l'analyseur sur un programme IMP++ utilisant les éléments traités (fourni dans `tests` ou ajouté par vos soins),
7. recommencer.

Résumé, en vrac, des éléments qui restent à traiter : constantes booléennes, accès à une variable, déclaration d'une variable globale ou d'une variable locale, valeur initiale optionnelle pour une variable, opérations arithmétiques et logiques autres que `+` et `*`, instructions d'affectation et de retour, instruction de branchement, instruction de boucle, paramètres des fonctions, expression d'appel de fonction, expression utilisée comme une instruction, expression d'accès à un tableau, instruction d'affectation à une case d'un tableau, expression de définition d'un nouveau tableau.

□