Langages, interprétation, compilation
TP4 - Analyse syntaxique et conflits

1. Prélude : construction d'un automate LR(0)

On considère la grammaire naïve suivante pour un fragment de l'ensemble des expressions arithmétiques.

```
S -> E #

E -> n
  |  E + E
  |  ( E )
```

Questions

1.1. Construction de l'automate

Construire l'automate LR(0) associé à cette grammaire.

1.2. Analyse de conflit

Décrire le conflit obtenu. Est-il lié à une ambiguïté de la grammaire ? Pour chacun des deux choix possibles d'action au niveau du conflit, détailler les étapes de l'analyse ascendante de l'entrée `1 + 2 + 3 + 4` et donner les arbres de dérivation correspondants.

2. Branchements conditionnels

On considère la grammaire Menhir suivante, qui représente des séquences d'expressions pouvant faire intervenir des branchements conditionnels :

```
%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 {}
;
```

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).

Questions

Vous devez au préalable copier la grammaire dans un fichier `if.mly` et la compiler avec l'option bavarde (`menhir -v`).

2.1. Automate

Dessiner l'automate décrit dans le fichier `if.automaton`.

2.2. Conflits

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`.

2.3. Correction

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.

3. 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 suivante :

```
%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 {}
;
```

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`).

Questions

Vous devez au préalable copier la grammaire dans un fichier `fcall.mly` et la compiler avec l'option bavarde (`menhir -v --infer`).

3.1. Conflits

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`.

3.2. Correction

Modifier la grammaire pour résoudre ces conflits.

Écriture alternative de la grammaire

On 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.

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

prog:
| expr EOF {}
;
```

```
expr:
| simple_expr {}
| ID simple_expr_list {}
| ID tuple {}
;

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

tuple:
| LPAR RPAR {}
| LPAR expr_comma_separated_list RPAR {}
;

simple_expr_list:
| simple_expr {}
| simple_expr_list simple_expr {}
;

expr_comma_separated_list:
| expr {}
| expr_comma_separated_list COMMA expr {}
;
```

4. Analyse syntaxique pour Imp

Cette dernière partie consiste à réaliser une chaîne complète d'analyse syntaxique pour un noyau de langage impératif :

- analyse lexicale
- analyse syntaxique
- production d'arbre de syntaxe

Présentation de la syntaxe concrète

Le langage `Imp` comporte des séquences d'instructions qui peuvent être :

- d'affichage : `putchar(e);`, qui affiche un caractère donné par son code ASCII
- d'affectation : `x = e;`, qui affecte une nouvelle valeur à une variable globale

ainsi que des structures de contrôle usuelles suivantes :

- branchement conditionnel : `if(e) { s1 } else { s2 }`
- boucle conditionnelle : `while(e) { s }`

Les expressions peuvent faire intervenir des constantes entières ou booléennes, des variables (dont une variable spéciale `arg`), et toute combinaison des opérateurs

- arithmétiques : `+`, `-`, `*`, `/`, `%`
- de comparaison : `==`, `!=`, `<`, `<=`, `>`, `>=`
- booléens : `!`, `$$`, `||`

Un programme complet est formé par une séquence de déclaration de variables globales de la forme `var x;` et d'une séquence d'instructions principale regroupée dans un bloc `main`.

Voici un exemple de programme `Imp`

```
var i;
var j;
var continue;

main {
  continue = true;
  i = 0;

  while (continue) {
    continue = false;
    j = 0;
    while (j < arg+1) {
      if (i*i + j*j < arg*arg) {
        putchar(46);
        continue = true;
      } else {
        putchar(35);
      }
      putchar(32);
      j = j+1;
    }
    putchar(10);
    i = i+1;
  }
}
```

Questions

4.1. Définition de la syntaxe abstraite

Dans un fichier `imp.ml`, définir des types `expr` et `instr` représentant respectivement les expressions et les instructions du langage Imp, ainsi qu'un type `prog` représentant un programme complet.

4.2. Analyse lexicale

Définir dans un fichier `impparser.mly` les lexèmes du langage Imp, et créer un analyseur lexical associé dans un fichier `implexer.mll`.

4.3. Analyse syntaxique

Compléter le fichier `impparser.mly` pour réaliser une fonction `Impparser.prog` qui fait l'analyse syntaxique d'un programme Imp et renvoie son arbre de syntaxe.

4.4. Bilan

Créer un fichier principal regroupant toute la chaîne d'analyse syntaxique, et ajouter des fonctionnalités permettant de tester votre travail (au choix : affichage, évaluation, etc).

4.5. Bonus

Étendre l'ensemble pour ajouter au langage la possibilité de définir et appeler des fonctions.