# Examen du cours "Interprétation des programmes"
# Durée: 3 heures

Une feuille manuscrite A4 autorisée.

**Exercice 1** *(Analyse syntaxique LL(1))*

*Soit la grammaire $G$ suivante :*

$$\begin{array}{rcl} j & ::= & \texttt{'\{'}\ b\ \texttt{'\}'} \\ b & ::= & \epsilon \mid b\ f \\ f & ::= & \texttt{'l'}\ \texttt{':'}\ j \end{array}$$

*où $j$, $b$ et $f$ sont des symboles non-terminaux, $j$ est le symbole d'entrée de la grammaire et* `'{'`, `'}'`, `'l'` *et* `':'` *sont des symboles terminaux.*

1. *Pourquoi la grammaire $G$ n'est-elle pas adaptée à l'analyse LL(1) ? Justifiez votre réponse.*
2. *Donnez une grammaire $G'$ équivalente à $G$ et adaptée à l'analyse LL(1). (On ne demande pas ici de construire la table LL mais seulement de s'assurer que la grammaire "a une chance" d'être LL(1) s'il n'y a pas de conflits LL.)*
3. *Quels sont les non-terminaux de la grammaire $G'$ qui sont annulables (c'est-à-dire à partir desquels on peut dériver le mot vide) ?*
4. *Calculer la fonction $FIRST$ pour les non terminaux de la grammaire $G'$.*
5. *Calculer la fonction $FOLLOW$ pour les non terminaux de la grammaire $G'$.*
6. *Construire la table LL(1) de la grammaire $G'$.*
7. *Décrire les étapes de la reconnaissance du mot « `{ l : { } l : { } }` ».*
8. *Décrire les étapes de l'analyse du mot « `{ l : { }` » en expliquant comment l'algorithme LL détermine que ce mot n'est pas dans la grammaire $G'$.*

□

**Exercice 2** *(Erreur dans* HOPIX*)*

*Dans cet exercice, nous allons rajouter deux nouvelles constructions dans* HOPIX *pour l'étendre avec un mécanisme de signalement d'erreurs. Ce nouveau langage est appelé* $\text{HOPIX}_{\perp}$.

*La première construction est le signalement d'erreur à proprement parlé. Une expression de la forme « **error** » provoque le déclenchement d'un mode d'évaluation dit "exceptionnel" : tous les calculs prévus sont annulés petit à petit tant que l'erreur n'est pas prise en charge par un gestionnaire d'erreur.*

*La seconde construction est la définition d'un gestionnaire d'erreur. On introduit une expression de la forme « **try** $e$ **orelse** $e'$ » qui installe un gestionnaire d'erreur pour l'évaluation de l'expression $e$. Ainsi, l'évaluation d'une telle expression débute par l'évaluation de $e$ puis, si l'évaluation de $e$ produit une erreur alors elle se poursuit par l'évaluation de $e'$. Si l'expression $e$ s'est évaluée normalement en une valeur $v$ alors c'est aussi le résultat de l'évaluation de « **try** $e$ **orelse** $e'$ » (i.e. $e'$ n'est pas évaluée).*

*On peut bien sûr imbriquer les gestionnaires d'erreur. Dans ce cas, c'est le dernier gestionnaire d'erreur installé qui a la priorité. Ainsi, l'expression* HOPIX *suivante s'évalue en 3.*

```
try do
  val y := 1;
  try
    if y = 1 then error else 2 fi
  orelse 3;
  if y = 0 then error else 5 fi
done orelse 4
```

*Si on change* `val y := 1;` *en* `val y := 0` *alors cette expression s'évalue en* 4.
*Enfin, si une erreur n'est jamais prise en charge par un gestionnaire d'erreur le programme est interrompu.*

1. *En quoi s'évalue l'expression suivante ?*

   ```
   (try 1 orelse 2) + (try error orelse 3)
   ```

2. *En quoi s'évalue l'expression suivante ?*

   ```
   6 * error
   ```

3. *En quoi s'évalue l'expression suivante ?*

   ```
   try try error orelse 42 orelse 0
   ```

4. *Pour spécifier la sémantique opérationnelle de* HOPIX$_\perp$, *il est nécessaire de modifier le jugement que vous avez utilisé dans le jalon 2 du projet. On utilise maintenant un jugement de la forme :*

   $$E, M \vdash e \Downarrow r, M'$$

   *où* $r$ *est soit une valeur* $v$, *soit une erreur notée* $\perp$.
   *Ainsi, un jugement de la forme*

   $$E, M \vdash e \Downarrow v, M'$$

   *se lit comme d'habitude :*
   *« Dans l'environnement d'évaluation E, l'expression e s'évalue en v et change la mémoire M en M'. »*
   *Par contre, un jugement de la forme*

   $$E, M \vdash e \Downarrow \perp, M'$$

   *se lit :*

   *« Dans l'environnement d'évaluation E, l'évaluation de e échoue et change la mémoire M en M'. »*

5. *Rappelez la définition et le rôle de l'environnement d'évaluation E en* HOPIX.
6. *Rappelez la définition et le rôle de la mémoire M en* HOPIX.
7. *Rappelez la règle d'évaluation pour la construction « `if ... then ... else ... fi` » de* HOPIX.
8. *Proposez une (ou plusieurs) règle(s) d'évaluation pour la construction « `if ... then ... else ... fi` » de* HOPIX$_\perp$.
9. *En vous inspirant de votre réponse à la question précédente, proposez une façon de mettre à jour l'ensemble des règles d'évaluation de* HOPIX *pour obtenir celles de* HOPIX$_\perp$.
10. *Proposez une (ou plusieurs) règle(s) d'évaluation pour la construction* **error** *de* HOPIX$_\perp$.
11. *Proposez une (ou plusieurs) règle(s) d'évaluation pour la construction* **try ... orelse ...** *de* HOPIX$_\perp$.
12. *Comment mettre à jour le type* OCAML *de l'arbre de syntaxe abstraite de* HOPIX *pour définir la syntaxe de* HOPIX$_\perp$ *?*
13. *Rappelez le type* OCAML *de la fonction d'évaluation des expressions de* HOPIX.
14. *Proposez un type pour la fonction d'évaluation des expressions de* HOPIX$_\perp$. *Vous indiquerez les éventuelles modifications à effectuer sur les déclarations de type de l'interpréteur.*
15. *Donnez le code* OCAML *correspondant à la règle d'évaluation de la construction « `if ... then ... else ... fi` » de* HOPIX$_\perp$.
16. *Donnez le code* OCAML *correspondant à la règle d'évaluation de la construction « **error** » de* HOPIX$_\perp$.
17. *Donnez le code* OCAML *correspondant à la règle d'évaluation de la construction « **try ... orelse ...** » de* HOPIX$_\perp$.
18. *Comment étendre ce mécanisme de signalement d'erreur pour obtenir le même comportement que les constructions « **raise** » et « **try ... with ...** » d'*OCAML *? Vous détaillerez les modifications à apporter à la forme du jugement d'évaluation et aux règles de la sémantique opérationnelle ainsi que les modifications à apporter à votre code* OCAML.

□