Examen du cours "Interprétation des programmes"
Première session
Durée: 3h00

Tout document autorisé. Les appareils électroniques sont interdits.

Exercice 1 *(Analyse syntaxique LL(1))*
Soit la grammaire G suivante :

$$\begin{array}{lcl} p & ::= & a\ \# \\ a & ::= & o\ b\ \text{'!'} \mid \text{'n'} \\ o & ::= & \text{'+'} \mid \text{'*'} \\ b & ::= & \epsilon \mid b\ a \end{array}$$

où p, a, o et b sont des symboles non-terminaux, p est le symbole d'entrée de la grammaire et '!', 'n', '+' et '' sont des symboles terminaux. ϵ représente le mot vide.*

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 sans récursion à gauche.*
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 « + n * ! ! # » par votre table LL(1).*
8. *Décrire les étapes de l'analyse du mot « * + n ! # » en expliquant comment l'algorithme LL détermine que ce mot n'est pas dans la grammaire G'.*

□

Exercice 2 *(Tableaux dans HOPIX)*
Dans cet exercice, on étudie l'extension de HOPIX avec des tableaux homogènes (dont tous les éléments sont du même type). Dans cette extension, on souhaite autoriser le programmeur à allouer un tableau d'une taille donnée et dont les cases sont initialisées avec une valeur donnée ; à écrire une valeur dans une case particulière d'un tableau et enfin à lire la valeur d'une case particulière d'un tableau. Comme les références, les tableaux doivent survivre aux appels de fonctions : ils doivent donc être alloués dans le tas. Enfin, on supposera donnée une primitive renvoyant la taille du tableau qu'on lui passe en argument.

1. *Proposez une syntaxe concrète pour les trois constructions citées dans le paragraphe précédent. Pour illustrer cette syntaxe, vous écrirez un programme de votre choix et utilisant ces constructions de l'extension d'HOPIX obtenue.*
2. *Comment étendre la syntaxe abstraite d'HOPIX pour prendre en compte ces constructions ?*
3. *On rappelle que dans la spécification du jalon 2, une mémoire M associe une valeur à chaque adresse mémoire. D'après vous, est-ce que ce modèle de la mémoire est suffisant pour prendre en charge les tableaux ? Justifiez votre réponse. Dans le cas d'une réponse négative, proposez une nouvelle définition adaptée aux tableaux.*

4. *Proposez trois règles de sémantique opérationnelle à grands pas pour interpréter ces trois constructions. On rappelle que le jugement d'évaluation des expressions d'*HOPIX *s'écrit :*

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

et se lit "Dans l'environnement d'évaluation E, l'expression e s'évalue en la valeur v et transforme la mémoire M en la mémoire M'."

5. *Écrire le code* OCAML *des nouveaux cas à rajouter dans l'interpréteur de* HOPIX *pour prendre en charge cette extension.*
6. *Proposez des règles de typage pour ces trois constructions.*

□

Exercice 3 *(Clause* ***when*** *dans l'analyse de motifs)*
La construction ***when*** *d'*OCAML *n'a pas été introduite dans l'analyse par motifs d'*HOPIX*. Dans cet exercice, on étudie ce mécanisme. Pour rappel, en* OCAML*, on peut écrire :*

```
1 let rec filter pred = function
2 | [] -> []
3 | x :: xs when pred x -> x :: filter pred xs
4 | _ :: xs -> filter pred xs
```

Les motifs des lignes 2 et 3 capturent toutes les deux les valeurs qui sont des listes non vides. Cependant, le premier motif est décoré d'une clause ***when*** *qui restreint son domaine d'application aux listes non vides dont le premier élément vérifie le prédicat* pred.

1. *Comment étendre l'arbre de syntaxe abstraite pour prendre en compte cette extension ?*
2. *Proposez une nouvelle règle d'évaluation pour l'analyse de motif de façon à prendre en compte les clauses* ***when****.*
3. *Décrivez informellement les nouvelles contraintes de typage imposées par la construction* ***when****.*
4. *D'après vous, quelles sont les conséquences de l'introduction de* ***when*** *sur l'analyse statique qui vérifie l'exhaustivité des analyses par motifs ?*

□

Exercice 4 *(Applications plus souples en* HOPIX*)*

Pour rappel, l'évaluation des applications en HOPIX *s'appuie sur la règle suivante :*

$$\frac{E, M \vdash e_f \Downarrow (m_1 \cdots m_n \texttt{ => } e)[E'], M' \qquad M_0 = M' \qquad \forall i \in [1 \ldots n], E, M_{i-1} \vdash e_i \Downarrow v_i, M_i \qquad E_0 = E' \qquad \forall i \in [1 \ldots n], E_{i-1}, M_n \vdash m_i \sim v_i \Uparrow E_i \qquad E_n, M_n \vdash e \Downarrow v, M'}{E, M \vdash e_f\,(e_1, \cdots, e_n) \Downarrow v, M'}$$

Dans cette règle, le nombre d'arguments effectifs passés à la fermeture résultante de l'évaluation de e_f doit être exactement n, c'est-à-dire le nombre de ses arguments formels. Dans les langages fonctionnels, comme OCAML *ou* HASKELL*, l'évaluation des applications est plus souple : le nombre d'arguments effectifs peut être supérieur à n et on parle alors de* ***sur-application*** *ou bien inférieur à n et on parle alors d'****application partielle****. Voici des exemples de telles applications en* OCAML *:*

```
let f x y = (* f attend deux arguments... *)
  let z = x + y in
  fun k -> 2 * z + k (* ... et produit une fonction. *)
let g = f 1 2    (* Une application totale. *)
let n = f 1 2 3 (* Une sur-application. *)
let h = f 1      (* Une application partielle. *)
```

1. *Qu'est-ce qu'une fermeture ?*
2. *Proposez une nouvelle règle d'évaluation pour traiter le cas des applications partielles. Donnez un exemple d'arbre de dérivation illustrant un usage de cette règle.*
3. *Proposez une nouvelle règle d'évaluation pour traiter le cas des sur-applications. Donnez un exemple d'arbre de dérivation illustrant un usage de cette règle.*
4. *Ajustez la règle de typage des applications pour que les sur-applications et les applications partielles soient autorisées dans les programmes bien typés.*

□