Sémantique et Traduction des Langages
Epreuve de contrôle – Session 3
Avec documents de cours – Sans questions
Lundi 13 Juin
Durée : 1h30
Sujet à rendre avec votre copie
Nom :
Prénom :

Les calculatrices, ordinateurs et téléphones portables sont interdits.

N.B. : Vous attacherez la plus grande importance à la clarté, à la précision et à la concision de la rédaction.
Toute réponse devra être justifiée.
Si vous repérez ce qui vous semble être une erreur d'énoncé, signalez celle-ci sur votre copie et poursuivez la composition en expliquant les hypothèses que vous avez faites.
Il est conseillé de lire tout le texte avant de répondre, pour avoir une vision globale de l'épreuve et traiter les questions dans l'ordre correspondant le mieux à vos connaissances.

1 Questions de cours

Vous donnerez des réponses précises et concises en prenant des exemples concrets si cela est nécessaire pour illustrer votre propos.

a/Que propose le formalisme des grammaires attribuées pour décrire la sémantique des langages ?

b/ Quelle est la méthode pour concevoir une grammaire attribuée à partir de la grammaire du langage, des informations disponibles dans le contexte (axiome de la grammaire) d'un programme et des informations qui doivent être produite par la sémantique (au niveau de l'axiome) ?

c/ En utilisant les règles de sémantique opérationnelle de miniML étudiée en cours, TD et TP, construire l'arbre correspondant au typage de :

$$\{\texttt{i} : \mathit{Entier}\} \vdash \texttt{if (i = 0) then i + 1 else i - 1} : \mathit{Entier}$$

2 Sémantique attribuée

L'objectif de cet exercice est de définir une sémantique attribuée pour calculer la profondeur maximum d'une expression parenthésée défini par la grammaire (A, V, S, P) avec l'alphabet $A = \{\texttt{po}, \texttt{pf}, \texttt{id}\}$ (`po` désigne une parenthèse ouvrante, `pf` désigne une parenthèse fermante, `id` désigne un identificateur), les non-terminaux $V = \{S, L\}$, l'axiome S et les règles de production :

$$P = \left\{ \begin{array}{l} S \rightarrow \texttt{po}\, L\, \texttt{pf} \\ L \rightarrow \texttt{id}\, L \\ L \rightarrow \Lambda \\ L \rightarrow S\, L \end{array} \right\}$$

a/ Construire l'arbre syntaxique (arbre de dérivation) pour l'exemple `(() a (b c) (()))`.

b/ Proposer des attributs sémantiques pour les non terminaux de V et décorer l'arbre syntaxique donné à la question précédente (vous ne recopierez pas l'arbre) pour calculer la valeur 3 qui correspond à la profondeur d'imbrication maximale des parenthèses.

c/ Définir les actions sémantiques pour les règles de production P qui calculent la profondeur d'imbrication maximale des parenthèses.

3 Répétition en miniML

L'objectif est de compléter le langage miniML étudié en cours, TD et TP par l'expression `for`. Celle-ci exécute une expression $m - n + 1$ fois avec $m \geq n$ et associe à une variable les valeurs comprises entre n et m à l'intérieur de cette expression. Cette variable n'est pas visible à l'extérieur de cette expression. Cette expression renvoie la valeur `()` comme toutes les expressions avec effet de bord de miniML. Si $m < n$, l'expression renvoie la valeur `()` directement (sans étape de calcul intermédiaire).

Le programme suivant renvoie la valeur 15.

```
let r = ref 0 in
  (for i = 1 to 5 do
    r := i + (!r)
  done;
  (! r))
```

La syntaxe de miniML est étendue de la manière suivante :

$$\begin{array}{rcl} \mathit{Expr} & \rightarrow & \cdots \\ & | & \texttt{for}\ \mathit{Ident} = \mathit{Expr}\ \texttt{to}\ \mathit{Expr}\ \texttt{do}\ \mathit{Expr}\ \texttt{done} \end{array}$$

Une approche possible pour les questions suivantes serait de traduire la répétition de type *Pour* en une répétition de type *Tant que* et de vous appuyer sur les règles définies en cours, TD et TP. Ce n'est pas ce qui est attendu dans les réponses. Il s'agit de définir des règles spécifiques aux répétitions de type *Pour*. Vous pouvez bien sûr vous inspirer des règles définies pour les répétitions de type *Tant que*.

a/ Ecrire les règles d'évaluation (sémantique opérationnelle) avec effets de bord pour cette nouvelle expression pour les cas sans erreurs. Les jugements seront de la forme : $\gamma \vdash [\texttt{for}\ i = e_1\ \texttt{to}\ e_2\ \texttt{do}\ e_3\ \texttt{done}, m_1] \Downarrow [\ldots, m_2]$

b/ Ecrire les règles d'évaluation (sémantique opérationnelle) avec effets de bord pour les cas d'erreur correspondant à cette nouvelle expression. Les jugements seront de la forme : $\gamma \vdash [\texttt{for}\ i = e_1\ \texttt{to}\ e_2\ \texttt{do}\ e_3\ \texttt{done}, m_1] \Downarrow [\bot, m_2]$

c/ Ecrire les règles de typage pour cette nouvelle expression. Les jugements seront de la forme : $\sigma \vdash \texttt{for}\ i = e_1\ \texttt{to}\ e_2\ \texttt{do}\ e_3\ \texttt{done} : \ldots$

Le type CaML `ast` est étendu pour prendre en compte cette nouvelle expression :

```
type ast =
| ForNode of string * ast * ast * ast
| ...
```

La fonction de typage `type_of_expr` est étendue par le cas :

```
let rec type_of_expr expr env =
  match expr with
    | (ForNode name minExpr maxExpr bodyExpr) -> ruleFor name minExpr maxExpre bodyExpr env
    | ...
```

d/ Proposer une définition en CaML pour la fonction `ruleFor`.

La fonction d'évaluation `value_of_expr` est étendue par le cas :

```
let rec value_of_expr (expr,mem) env =
  match expr with
    | (ForNode name minExpr maxExpr bodyExpr) -> ruleFor name minExpr maxExpre bodyExpr mem env
    | ...
```

e/ Proposer une définition en CaML pour la fonction `ruleFor`.

4 Répétition en Bloc

L'objectif est de compléter le langage Bloc par une instruction de répétition. Nous ajoutons pour cela une instruction `for` inspirée des langages de la famille C (C, C++, C#, Java, etc) selon la syntaxe CUP suivante :

```
Instruction ::= ...
   | UL_Pour UL_Parenthese_Ouvrante
        UL_Type_Entier:type UL_Identificateur:name UL_Egal Expression:min UL_Point_Virgule
        UL_Identificateur UL_Inferieur Expression:max UL_Point_Virgule
        UL_Identificateur UL_Double_Plus
     UL_Parenthese_Fermante Bloc:corps
    {:




    :}
```

L'instruction `for` exécute un bloc $m - n + 1$ fois si $m \geq n$ et associe à une variable les valeurs comprises entre n et m à l'intérieur de ce bloc. Cette variable n'est pas visible à l'extérieur de ce bloc. L'instruction ne fait rien si $m < n$.

Le programme suivant s'arrête donc en affichant la valeur 15.

```
test {
  int r = 0;
  for (int i = 0; i < 6; i++) {
    r = i + r;
  }
  print r;
}
```

Proposer un programme TAM qui pourrait être généré à partir de ce programme Bloc.

Une approche possible pour les questions suivantes serait de traduire la répétition de type *Pour* en une répétition de type *Tant que* et de vous appuyer sur les règles définies en cours, TD et TP. Ce n'est pas ce qui est attendu dans les réponses. Il s'agit de définir des règles spécifiques aux répétitions de type *Pour*. Vous pouvez bien sûr vous inspirer des règles définies pour les répétitions de type *Tant que*.

4.1 Construction de l'arbre abstrait

L'instruction `for` peut apparaitre dans un bloc comme toute instruction.

a/ Proposer sans les détailler une ou plusieurs classes pour représenter cette nouvelle instruction dans l'arbre abstrait.
b/ Modifier la grammaire CUP précédente pour construire l'arbre abstrait correspond à cette instruction.

4.2 Gestion de la table des symboles

a/ Expliquer quelles sont les traitements nécessaires pour gérer la table des symboles de cette nouvelle instruction.
b/ Proposer en Java des méthodes de gestion de la table des symboles de cette/ces classe(s).

4.3 Typage

a/ Expliquer quelles sont les traitements nécessaires pour gérer le typage de cette nouvelle instruction.
b/ Proposer en Java des méthodes de typage de cette/ces classe(s).

4.4 Génération de code

a/ Expliquer quelles sont les traitements nécessaires pour gérer la génération de code pour cette nouvelle instruction.
b/ Proposer en Java des méthodes de génération de code pour cette/ces classe(s).