Examen du cours “Introduction à la compilation”
Durée: 3 heures

Tout document autorisé.

Le soin apporté à la rédaction et à la présentation ainsi que la rigueur des réponses seront pris en compte dans la notation. Ce sujet se décompose en 2 parties indépendantes. La première est une application directe du cours d'analyse syntaxique. La suivante porte sur l'étude d'une version simplifiée de JAVA.

1 Analyse syntaxique (8 points)

Soient un ensemble de non-terminaux $\{S, E\}$ et un ensemble de terminaux $\{n, +, *, \$\}$. On définit la grammaire G suivante :

$$\begin{array}{lcl} S & \rightarrow & E\,\$ \\ E & \rightarrow & E\,E\,+ \mid E\,E\,* \mid n \end{array}$$

Exercice 1

1. *Pourquoi la récursion à gauche de cette grammaire pose-t-elle un problème à un analyseur syntaxique descendant de type LL ?*
2. *Transformez la grammaire G en une grammaire équivalente G' sans récursion à gauche.*
3. *La grammaire G' peut être factorisée à gauche. Pourquoi cette transformation est-elle nécessaire pour la rendre compatible avec une analyse de type LL ?*
4. *Factorisez à gauche la grammaire G' afin d'obtenir une grammaire équivalente G''.*
5. *Calculez la table d'analyse $LL(1)$ de la grammaire G''.*
6. *Utilisez la table d'analyse $LL(1)$ précédente pour analyser l'entrée « $n\,n\,n\,+\,*\$$ ». À chaque étape de l'analyse, vous donnerez d'une part la prédiction courante et d'autre part l'action (d'acceptation d'un terminal ou d'expansion d'un non terminal) qui permet de passer à l'étape suivante.*
7. *On étend la grammaire G en rajoutant la règle « $E \rightarrow E\,E\,*\,*$ ». Peut-on appliquer les transformations précédentes pour traiter cette nouvelle grammaire à l'aide de l'algorithme d'analyse syntaxique $LL(1)$? (Vous pouvez donner une explication informelle, c'est-à-dire sans refaire les transformations de façon détaillée.)*

Rappels :

– *Pour supprimer la récursion à gauche immédiate d'un non terminal A dont les règles sont de la forme :*

$$A \rightarrow A\alpha_1 \mid \ldots \mid A\alpha_n \mid \beta_1 \mid \ldots \mid \beta_m$$

où les $(\alpha_i)_{i\in[1..n]}$ sont dans $(T \cup N)^\star$ et les $(\beta_i)_{i\in[1..m]}$ sont dans $(T \cup N)^\star$ et ne commencent pas par A. On introduit un non-terminal A' dont les règles sont :

$$A' \rightarrow \epsilon \mid \alpha_1 A' \mid \ldots \mid \alpha_n A'$$

et on remplace les règles du non-terminal A par :

$$A \rightarrow \beta_1 A' \mid \ldots \mid \beta_m A'$$

– *Pour factoriser à gauche deux règles “$R_1 \equiv A \rightarrow \alpha\beta_1$” et “$R_2 \equiv A \rightarrow \alpha\beta_2$” partageant le même préfixe α, il su t d'introduire un non-terminal A' et sa définition “$A' \rightarrow \beta_1 \mid \beta_2$” puis de remplacer les deux règles R_1 et R_2 par la règle “$A \rightarrow \alpha A'$.*

□

2 Un Java épuré (12 points)

Ce sujet d'examen s'inspire fortement de l'article *Featherweight Java, A minimal core calculus for Java and GJ* écrit par Atsushi Igarashi, Benjamin Pierce et Philip Wadler pour formaliser un langage de programmation appelé FJ, une version épurée de JAVA. Dans un premier temps, on présente la syntaxe de ce langage et on vous fait écrire un premier programme. Ensuite, on introduit la relation de sous-classage établissant des relations de conversion valide entre objets. Cette relation est utilisée pour définir la sémantique opérationnelle des programmes. Enfin, on étudiera le problème de l'analyse statique des programmes de ce langage.

2.1 Syntaxe

La syntaxe abstraite de FJ est donnée par la grammaire suivante :

p	::=		***Programme***
	|	$\overline{CL}$ `in` *e*	
CL	::=		***Déclaration de classes***
	|	`class` C `extends` $C\{\ \overline{C}\,\overline{f}$; *K*; $\overline{M}\ \}$	
K	::=		***Définition d'un constructeur***
	|	$C\,(\overline{C}\,\overline{f})$ { `super`$(\overline{f})$; $this.\overline{f} = \overline{f}$; }	
M	::=		***Définition d'une méthode***
	|	$C\,m\,(\overline{C}\,\overline{f})$ { `return` *e*; }	
e	::=		***Expression***
	|	x	*Variable*
	|	*e*.f	*Sélection d'un attribut*
	|	*e*.m (*e*)	*Appel de méthode*
	|	`new` C(*e*)	*Instantiation*
	|	(C)*e*	*Conversion*

Les métavariables A, B et C sont utilisées pour dénoter des noms de classe, f et g pour dénoter des noms de champs, x et y pour dénoter des arguments formels, d et e des expressions, CL pour dénoter des déclarations de classe, K des déclarations de constructeur et M des déclarations de méthode. On écrit $\overline{X}$ pour représenter une séquence $X_1, \ldots, X_n$ d'objets dénotés par la métavariable X. On écrit aussi $\overline{C}\,\overline{f}$ pour représenter une séquence de déclarations $C_1\,f_1, \ldots, C_n\,f_n$. Enfin, $this.\overline{f} = \overline{f}$ signifie $this.f_1 = f_1; \ldots; this.f_n = f_n$.

Un programme est formé d'une suite de déclaration de classes $\overline{CL}$ et d'une expression *e*. Comme nous le verrons dans la section suivante du sujet, l'évaluation d'un programme est l'évaluation de e en prenant en compte les déclarations des classes *CL*. Ces déclarations *CL* définissent une fonction $CT(C)$ qui associe sa déclaration à chaque identificateur de classe C. On suppose qu'il n'existe pas deux déclarations avec le même identificateur. On distingue un identificateur de classe $Object$ qui est reservé et qui n'a pas d'image par la fonction CT.

Une déclaration de classe *CL* de la forme « `class` C `extends` $D\{\ \overline{C}\,\overline{f}$; *K*; $\overline{M}\ \}$ » définit une classe C qui hérite de la classe D en lui rajoutant les champs $\overline{f}$ et les définitions (ou redéfinitions) de méthodes *M*. L'identificateur $this$ est reservé et ne peut être utilisé comme nom d'attribut ou d'argument formel.

Par ailleurs, chaque classe a un constructeur de la forme « $C(\overline{C}\,\overline{f}_1)$ { `super`$(\overline{f}_2)$; $this.\overline{f}_3 = \overline{f}_3$; } ». Les arguments formels $\overline{f}_1$ correspondent à l'ensemble des attributs de la classe, comprenant ceux hérités $\overline{f}_2$ et ceux spécifiques à la classe $\overline{f}_3$. Un constructeur fait donc d'abord appel au constructeur de classe mère puis initialise les attributs propres à la classe C. On suppose que les attributs de la classe mère et de la classe fille sont distincts.

Une déclaration de méthode *M* a la forme « $C\,m\,(\overline{C}\,\overline{f})$ { `return` *e*; } » où m est le nom de la méthode définie (ou redéfinie) et les arguments formels $\overline{f}$ sont liés dans le corps de la méthode représenté par l'expression *e*.

Voici un exemple de programme écrit dans ce langage :

```
class Nat extends Object {
  Nat() {
    super();
  }
}
class Zero extends Nat { // L'entier zero.
  Zero() {
    super();
  }

  Nat add(Nat that) {
    return that;
  }
}
class Succ extends Nat { // Le successeur de l'entier "n".
  Nat n;
  Succ(Nat n) {
    super();
    this.n = n;
  }

  Nat add(Nat that) {
    return (new Succ(n.add(that));
  }
}
```

Exercice 2

1. *Quelle est la di érence entre syntaxe concrète et syntaxe abstraite ?*
2. *Modifiez les définitions de la classe $Zero$ et de la classe $Succ$ pour définir une méthode e ectuant la multiplication entre l'entier représenté par $this$ et un argument formel. (Indication : Appuyez-vous sur la méthode e ectuant l'addition.)*
3. *On rappelle qu'un « sucre syntaxique » est une extension de la syntaxe concrète qui ne nécessite pas de modification de la syntaxe abstraite pour être traitée. On reconnaît un sucre syntaxique dans un compilateur au fait qu'il est implémenté dans les actions sémantiques de l'analyse syntaxique.*

   Par exemple, en cours, nous avons parfois défini la syntaxe « `let x = e_1 in e_2` » comme un sucre syntaxique pour « `(fun x → e_2) e_1` ».

   Indiquez, en justifiant votre réponse, si les extensions suivantes sont des sucres syntaxiques :

   (a) *Le programmeur peut écrire « $e.m$ » pour dire « $e.m()$ » quand la méthode « m » n'attend pas d'argument.*

   (b) *Le programmeur peut écrire « $C\, m\ (A\, x, y)\{\ldots\}$ » pour déclarer une méthode m dont les deux arguments ont le même type, c'est-à-dire « $C\, m\ (Ax, Ay)\{\ldots\}$ ».*

   (c) *Le programmeur peut écrire « `class` $C\{\ldots\}$ » pour dire « `class` C `extends` $Object\{\ldots\}$ ».*

□

2.2 Relation de sous-classage

On définit la relation de sous-classage $A <: B$ par le système de règles d'inférence suivant :

$$\textsc{Reflexivity}\ \frac{}{A <: A} \qquad \textsc{Transitivity}\ \frac{A <: B \qquad B <: C}{A <: C}$$

$$\textsc{Inherit}\ \frac{CT(C) = \texttt{class}\ C\ \texttt{extends}\ D\{\ \overline{C}\,\overline{f};\ K;\ \overline{M}\ \}}{C <: D}$$

Exercice 3

1. *Écrire l'arbre de preuve correspondant à la dérivation du jugement $Zero <: Object$ à partir des déclarations de l'exemple précédent.*
2. *Énoncez une condition nécessaire sur les déclarations de classe pour que l'on ne puisse pas dériver à la fois $A <: B$ et $B <: A$ si A et B sont distincts. Nous appelerons cette propriété "bonne fondation de la relation de sous-classage".*

□

2.3 Sémantique opérationnelle

Pour définir la sémantique opérationnelle de FJ, deux fonctions d'extraction d'informations à partir des déclarations sont nécessaires.

La première fonction $fields$ sert à extraire les attributs, à la fois hérités et spécifiques, d'une classe. Elle est spécifiée par les règles suivantes :

$$\frac{}{fields(Object) = \bullet}$$

$$\frac{CT(C) = \texttt{class}\ C\ \texttt{extends}\ D\{\ \overline{C}\,\overline{f};\ K;\ \overline{M}\ \} \qquad fields(D) = \overline{D}\,\overline{g}}{fields(C) = \overline{D}\,\overline{g}, \overline{C}\,\overline{f}}$$

La seconde fonction $mbody$ sert à résoudre un appel de méthode à partir d'un identificateur de classe et d'un identificateur de méthode. On la spécifie ainsi :

$$\frac{CT(C) = \texttt{class}\ C\ \texttt{extends}\ D\{\ \overline{C}\,\overline{f};\ K;\ \overline{M}\ \} \quad B\,m(\overline{B}\,\overline{x})\{\ \texttt{return}\ e;\ \} \in \overline{M}}{mbody(C, m) = (\overline{x}, e)}$$

$$\frac{CT(C) = \texttt{class}\ C\ \texttt{extends}\ D\{\ \overline{C}\,\overline{f};\ K;\ \overline{M}\ \} \quad m \text{ n'est pas définie dans } \overline{M}}{mbody(C, m) = mbody(D, m)}$$

Exercice 4 (Fonctions auxiliaires)

1. *Calculez $fields(Succ)$ en donnant l'arbre de preuve correspondant.*
2. *Soient trois classes A, B et C dont voici les déclarations :*

```
class A extends Object {
  B() { super(); }
  Nat m1() { return e1; }
  Nat m2() { return e2; }
}
class B extends A {
  B() { super(); }
  Nat m1() { return e3; }
}
class C extends B {
  C() { super(); }
  Nat m2() { return e4; }
}
```

   Calculez $mbody(C, m_1)$, $mbody(B, m_1)$, $mbody(C, m_2)$ et $mbody(B, m_2)$.

3. *Pourquoi la propriété de "bonne fondation de la relation de sous-classage" est-elle nécessaire à la définition de la fonction $mbody$? Donnez l'exemple d'une relation de sous-classage mal fondée qui pose problème pour définir $mbody$.*

□

Pour évaluer un programme « *CL* `in` *e* », on commence par construire la fonction CT puis on vérifie la bonne fondation de la relation de sous-classage, et on évalue finalement l'expression e. Pour évaluer une expression e, trois cas se présentent : soit l'expression est une valeur, soit une règle de la sémantique opérationnelle à petits pas s'applique, soit l'expression est bloquée.

Une expression e est une valeur si on peut dériver le jugement « e valeur » défini par les règles suivantes :

$$\frac{}{\text{new } Object() \text{ valeur}} \qquad \frac{\forall i \quad e_i \text{ valeur}}{\text{new } C(\overline{e}) \text{ valeur}}$$

On note $[e_1/x]e_2$, l'expression e_2 dans laquelle toutes les occurrences libres de x sont remplacées par e_1.
La sémantique opérationnelle à petits pas est définie par la relation binaire « $e \to e'$ » dont les règles sont :

$$\textsc{Field}\quad \frac{fields(C) = \overline{C}\,\overline{f} \qquad \forall i \qquad \mathsf{e}_i \text{ valeur}}{(\mathsf{new}\ C(\overline{e})).f_i \to e_i}$$

$$\textsc{Call}\quad \frac{mbody(C,m) = (\overline{x}, e_0) \qquad \forall i \qquad \mathsf{e}_i \text{ valeur} \qquad \forall i \qquad \mathsf{d}_i \text{ valeur}}{(\mathsf{new}\ C(\overline{e})).m(\overline{d}) \to [\overline{d}/\overline{x}, \mathsf{new}\ C(\overline{e})/this]e_0}$$

$$\textsc{Cast}\quad \frac{C <: D \qquad \forall i \qquad \mathsf{e}_i \text{ valeur}}{(D)(\mathsf{new}\ C(\overline{e})) \to \mathsf{new}\ C(\overline{e})} \qquad \textsc{Object1}\quad \frac{e_0 \to e_0'}{e_0.f \to e_0'.f} \qquad \textsc{Object2}\quad \frac{e_0 \to e_0'}{e_0.m(\overline{e}) \to e_0'.m(\overline{e})}$$

$$\textsc{Object3}\quad \frac{e_0 \to e_0'}{(C)e_0 \to (C)e_0'} \qquad \textsc{MethodArguments}\quad \frac{e_0 \text{ valeur} \qquad \overline{e} \to \overline{e}'}{e_0.m(\overline{e}) \to e_0.m(\overline{e}')} \qquad \textsc{NewArguments}\quad \frac{\overline{e} \to \overline{e}'}{\mathsf{new}\ C(\overline{e}) \to \mathsf{new}\ C(\overline{e}')}$$

Exercice 5 (Évaluation des expressions)

1. *Parmi les règles de la sémantique, quelles sont les règles de réduction et quelles sont les règles de passage au contexte ?*
2. *Évaluez l'expression* e_1 *«* $(\mathbf{new}\ Succ(\mathbf{new}\ Zero)).add(\mathbf{new}\ Succ(\mathbf{new}\ Zero))$ *» en donnant l'arbre de preuve qui vous a permis d'obtenir le résultat.*
3. *Évaluez «* $e_1.add(e_1)$ *». (Vous pouvez bien sûr réutiliser la réponse à la question précédente.)*
4. *On se donne les déclarations de classe supplémentaires suivantes :*

```
class R extends Object {
  Nat() {
    super();
  }
  Nat m1(X that) {
    return that.m2();
  }
}
class X extends Object {
  X() {
    super();
  }
  Nat m2() {
    return (new Zero);
  }
}
class Y extends X {
  Y() {
    super();
  }
  Nat m2() {
    return (new Succ(new Zero));
  }
}
```

   Évaluez l'expression « $(\mathbf{new}\ R()).m_1(\mathbf{new}\ Y())$ *» en donnant l'arbre de preuve correspondant. Pourquoi l'appel des méthodes est-il aussi appelé "liaison tardive" dans les langages à objets ?*
5. *Pour chacune des expressions suivantes, expliquez pourquoi l'évaluation se bloque en donnant l'expression bloquée correspondante, c'est-à-dire l'expression issue de cette évaluation que l'on ne peut plus réduire mais qui n'est pourtant pas une valeur :*
   (a) $(\mathbf{new}\ Zero()).n$
   (b) $(\mathbf{new}\ R()).m_1()$

(c) (**new** R()).m_1(**new** $Zero$)

(d) ($Zero$)(**new** Nat())

□

2.4 Analyse statique

Exercice 6

1. *En vous inspirant de la définition de la fonction $mbody$, donnez une spécification pour une fonction $mtype$ associant à un nom de méthode m et à une classe C, un couple formé des types des arguments et du type de retour de m dans C.*
2. *(Hors barême) Proposez des règles de typage définissant le jugement « $\Gamma \vdash e \in C$ », qui se lit « sous l'environnement de typage Γ, l'expression e s'évalue en une instance de la classe C ».*

□