

Langages et Traducteurs

Examen du mardi 3 janvier 2017

Durée : deux heures - **Tous documents autorisés.** - Les 2 exercices sont **indépendants**.

Exercice 1 (~ 13 points)

On considère le langage **While** vu en cours (dans sa version sans procédures ni blocs imbriqués), auquel on ajoute le type **ref t** qui signifie “pointeur vers un type **t**”. Intuitivement, une variable de type **ref t** peut prendre pour valeur l’*adresse mémoire* d’une variable de type **t**. On considère donc également deux nouveaux opérateurs, inspirés du langage C :

- l’opérateur ***e**, qui fournit la valeur (de type **t**) référencée par l’expression **e** (de type **ref t**);
- l’opérateur **&x**, qui fournit l’adresse (de type **ref t**) d’une variable **x** (de type **t**).

On obtient ainsi une nouvelle syntaxe pour le langage **While**

$$\begin{aligned}
 P &::= D ; C \\
 T &::= \text{entier} \mid \text{booléen} \mid \text{ref } T \\
 D &::= \text{var } x : T \mid D ; D \\
 G &::= x \mid *G \\
 E &::= n \mid G \mid \&x \mid E + E \mid \text{true} \mid E = E \mid \text{not } E \mid E \text{ and } E \\
 C &::= G := E \mid \text{skip} \mid C ; C \mid \text{if } E \text{ then } C \text{ else } C \mid \text{while } E \text{ do } C
 \end{aligned}$$

Dans cette syntaxe x désigne un identificateur ($x \in \text{Noms}$) et n une constante entière positive ou négative ($n \in \mathbb{Z}$). Le non-terminal G désigne les expressions qui peuvent figurer en partie gauche d’une affectation, et sur lesquelles l’opérateur ***** peut être appliqué. On donne ci-dessous deux exemples de programme, l’un correct et l’autre incorrect du point de vue de la syntaxe :

<pre> var x : entier ; var y : ref entier ; var z : ref ref entier ; x := 3 ; y := &x ; z := &y ; **z := *y + x ; </pre>	<pre> var x : entier ; var y : ref entier ; var z : ref ref entier ; &x := 3 ; /* erreur de syntaxe sur la partie gauche */ y := &(x + 1) ; /* erreur de syntaxe sur la partie droite */ z := &&x ; /* erreur de syntaxe sur la partie droite */ **z := *&y ; /* erreur de syntaxe sur la partie droite */ </pre>
--	---

Partie 1 : sémantique statique

On définit l’ensemble $Type$ de la manière suivante :

$$\begin{aligned}
 TypeBase &= \{Entier, Booléen\} \\
 Type &= \{Void\} \cup (\mathbb{N} \times TypeBase)
 \end{aligned}$$

Ainsi, un élément de cet ensemble est soit le type “vide”, soit un couple (n, t_0) où n est un entier naturel et t_0 le type **Entier** ou **Booléen**. On définit alors une fonction $\mathcal{T}$ qui transforme tout élément syntaxique T en un élément de l’ensemble $Type$:

$$\begin{aligned}
\mathcal{T}(\text{entier}) &= (0, \text{Entier}) \\
\mathcal{T}(\text{booléen}) &= (0, \text{Booleen}) \\
\mathcal{T}(\text{ref } \mathbf{t}) &= \text{soit } (n, t_0) = \mathcal{T}(\mathbf{t}) \text{ dans } (n+1, t_0)
\end{aligned}$$

Intuitivement, (n, t_0) représente le type comportant “ n niveaux de références” vers le type de base t_0 . Ainsi, $\mathcal{T}(\text{ref ref booléen}) = (2, \text{Booleen})$.

On rappelle que les règles de *sémantique statique* du langage While sont définies de la manière suivante :

- Un *environnement* $\mathbf{Env}$ est une fonction partielle $\mathbf{Noms} \rightarrow \mathbf{Type}$.
- la sémantique statique d’une *déclaration* D est définie par une relation $\xrightarrow{d} \subseteq (D \cup \mathbf{Env})^2$.
- la sémantique statique d’une *expression* E est définie par une relation $\xrightarrow{e} \subseteq ((E \times \mathbf{Env}) \cup \mathbf{Type})^2$.
- la sémantique statique d’une *commande* C est définie par une relation $\xrightarrow{c} \subseteq ((E \times \mathbf{Env}) \cup \mathbf{Type})^2$.

On donne ci-dessous les (nouvelles) règles de sémantique statique associées aux déclarations :

$$\text{var } x \ t \xrightarrow{d} [x \mapsto \mathcal{T}(t)]$$

$$\frac{D1 \xrightarrow{d} \rho1 \quad D2 \xrightarrow{d} \rho2 \quad \text{Dom}(\rho1) \cap \text{Dom}(\rho2) = \emptyset}{D1 ; D2 \xrightarrow{t} \rho1 \cup \rho2}$$

Q1.

1. Complétez la règle ci-dessous qui exprime la sémantique statique de l’expression $\&x$:

$$\frac{\dots}{\langle \&x, \rho \rangle \xrightarrow{e} (n+1, t)}$$

2. Donnez un exemple de programme incorrect vis-à-vis de cette règle.
3. Proposez une règle de sémantique statique pour l’expression $\ast G$
4. Donnez un exemple de programme incorrect vis-à-vis de cette règle.

Q2.

1. On suppose dans cette partie qu’une affectation est correcte (du point de vue des types) lorsque les opérandes gauches et droits sont corrects, et de *même type*. Proposez une règle de sémantique statique pour la commande $G := E$.
2. Donnez un exemple de programme incorrect vis-à-vis de cette règle.

Partie 2 : sémantique dynamique

On note $\mathbf{Adr}$ l’ensemble des *adresses* mémoires, et on note $\mathbf{Val}$ l’ensemble des *valeurs* $\mathbb{Z} \cup \{\mathbf{tt}, \mathbf{ff}\} \cup \mathbf{Adr}$. De plus :

- Un *environnement* $\mathbf{Env}$ est une fonction partielle $\mathbf{Noms} \rightarrow \mathbf{Adr}$;
- Une *mémoire* $\mathbf{Mem}$ est une fonction partielle $\mathbf{Adr} \rightarrow \mathbf{Val}$.

Ainsi, le résultat d’une expression (et donc le contenu d’une case mémoire) peut-être soit un entier ou un booléen (pour un type de base), soit une *adresse* (pour un un type pointeur).

Les règles de *sémantique dynamique* de ce langage sont alors définies de la manière suivante :

- la sémantique dynamique d'une *déclaration* D est définie par une relation $\xrightarrow{d} \subseteq (D \cup \mathbf{Env})^2$.
- la sémantique statique d'une *expression* E est définie par une relation $\xrightarrow{e} \subseteq ((E \times \mathbf{Env} \times \mathbf{Mem}) \cup \mathbf{Val})^2$
- la sémantique statique d'une *commande* C est définie par une relation $\xrightarrow{c} \subseteq ((C \times \mathbf{Env} \times \mathbf{Mem}) \cup \mathbf{Mem})^2$.

Les règles de sémantique associées aux déclarations sont inchangées par rapport à celles vues en cours.

Q3.

1. Complétez la règle suivante qui exprime la sémantique dynamique de l'expression $\&x$:

$$\frac{\dots}{\langle \&x, \eta, \sigma \rangle \xrightarrow{e} \dots}$$

2. Proposez une règle de sémantique dynamique pour l'expression $*G$
3. Proposez une règle de sémantique dynamique pour l'affectation $G := E$.
On pourra ici distinguer deux cas (et donc écrire deux règles) selon que G est de la forme " x " ou " $*G$ ".

Partie 3 : génération de code

On propose maintenant de compléter la fonction de génération de code d'une expression pour la machine abstraite $\mathcal{M}$ vue en cours. On rappelle le profil de cette fonction : $\mathbf{GenCodeAExp} : \mathbf{AExp} \rightarrow \mathbf{Code}^* \times \mathbb{N}$. $\mathbf{GenCodeAExp}(e)$ produit un couple (C, i) où C est le code permettant de calculer la valeur de e et de la mémoriser dans le registre R_i .

Q4.

1. Ecrivez le code de la fonction $\mathbf{GenCodeAExp}(\&x)$.
2. Ecrivez le code de la fonction $\mathbf{GenCodeAExp}(*G)$.

Indication : on peut écrire une fonction *récursive*, sans re-détailler les autres constructions du langage déjà vues en cours (chaque réponse tient donc en quelques lignes!).

Partie 4 : Déréférencage implicite

On suppose maintenant que les règles de typage sont plus flexibles, et que notamment l'utilisation de l'opérateur " $*$ " (déréférencage) n'est plus obligatoire. Ces opérateurs seront "ajoutés" par le compilateur, *en nombre minimal*, partout où cela est rendu nécessaire pour que le programme soit correct du point de vue des types. Ainsi le programme suivant est désormais correct :

```
var x : entier ;
var y : ref entier ;
var z : ref ref entier ;
x := 3 ;
y := &x ;
z := &y ;
y := z + 1 ;           /* z sera déréférencé une fois pour permettre l'affectation */
x := x + y ;           /* y sera déréférencé une fois pour permettre l'addition */
x := z + 1 ;           /* z sera déréférencé deux fois pour permettre l'affectation */
if (y = z) ... ;      /* z sera déréférencé une fois pour permettre la comparaison */
```

Q5.

1. Ecrivez la nouvelle règle de sémantique statique pour l'expression $E1 + E2$
2. Ecrivez la nouvelle règle de sémantique statique pour l'expression $E1 = E2$

3. Ecrivez la nouvelle règle de sémantique statique pour la commande $G := E$
4. Quelles sont les informations qu'il est nécessaire de mémoriser lors de la vérification de la sémantique statique pour la phase de génération de code ?
 Illustrez votre réponse (informelle) en donnant le code assembleur que produirait le compilateur pour les quatres dernières instructions du programme précédent. On pourra supposer que les adresses des variables x , y et z sont obtenues en ajoutant respectivement les déplacements D_x , D_y et D_z au contenu du registre FP .

Exercice 2 (~ 7 points)

```

procedure P(void);
  var x ;

  procedure P1 (a)
    var x1 ;

    procedure P2(b, c);
      var x2 ;
      begin { P2 }
        x2 := c ;
        x := x1 + x2 + b ;    /* instruction (3) */
      end; { P2 }

    begin { P1 }
      x1 := a ;
      P2 (x+1, a) ;    /* instruction (2) */
      x := 2 ;
    end; { P1 }

  begin { P }
    x := 0 ;
    P1 (x);    /* instruction (1) */
  end { P }

```

- Q1.** Dessinez le contenu complet de la pile lors de l'exécution de la procédure P2. On précisera bien les liens statiques et dynamiques des procédures appelées.
- Q2.** Donnez le code assembleur correspondant à l'instruction **(1)** de la procédure P.
- Q2.** Donnez le code assembleur correspondant à l'instruction **(2)** de la procédure P1.
- Q3.** Donnez le code assembleur correspondant à l'instruction **(3)** de la procédure P2.