Compilateur Java

deuxième partie : analyse sémantique

à rendre avant le lundi 1er décembre 10 heures

1 Analyse sémantique

Cette deuxième partie consiste à vérifier la conformité du fichier source, une fois l'analyse syntaxique effectuée avec succès. Dans tout ce qui suit, les types sont de la forme suivante :

$$\tau ::= \texttt{void} \mid \texttt{boolean} \mid \texttt{int} \mid \texttt{double} \mid C \mid \texttt{typenull}$$

où C désigne une classe. (Il est pratique de considérer `void` comme un type, même si ce n'est pas un type au sens syntaxique.)

1.1 Héritage et sous-typage

On notera $C_1 \longrightarrow C_2$ la relation « la classe C_1 est une sous-classe de la classe C_2 », qui est la clôture réflexive-transitive de l'ensemble des déclarations d'héritage C_1 `extends` C_2 contenue dans le source.

Il y a deux classes prédéfinies, qui sont *Object* et *String*. Toute classe qui n'est pas déclarée comme héritant d'une autre hérite de la classe *Object*. La classe *String* hérite de la classe *Object*. La classe *Object* n'hérite d'aucune classe.

La relation de sous-typage $\tau_1 \sqsubseteq \tau_2$ signifie « le type τ_1 est un sous-type du type τ_2 » et est définie par les règles d'inférence suivantes :

$$\frac{\tau \in \{\texttt{boolean},\ \texttt{int},\ \texttt{double}\}}{\tau \sqsubseteq \tau} \qquad \frac{}{\texttt{int} \sqsubseteq \texttt{double}} \qquad \frac{C_1 \longrightarrow C_2}{C_1 \sqsubseteq C_2} \qquad \frac{}{\texttt{typenull} \sqsubseteq C}$$

Intuitivement, on peut interpréter la relation $\tau_1 \sqsubseteq \tau_2$ par « toute valeur de type τ_1 peut être donnée là où est attendue une valeur de type τ_2 ». On dit que les types τ_1 et τ_2 sont *compatibles*, et on note $\tau_1 \equiv \tau_2$, si $\tau_1 \sqsubseteq \tau_2$ ou $\tau_2 \sqsubseteq \tau_1$; dans ce cas, on note $\tau_1 \sqcup \tau_2$ le plus grand de ces deux types.

La notion de sous-typage est étendue aux profils (listes de types) de la manière suivante :

$$(\tau_1, \ldots, \tau_n) \sqsubseteq (\tau'_1, \ldots, \tau'_n) \text{ si et seulement si } \tau_i \sqsubseteq \tau'_i \text{ pour tout } i \in 1, \ldots n.$$

1.2 Attributs, constructeurs et méthodes d'une classe

Dans toute la suite, C_0 désigne la classe courante, c'est-à-dire la classe que l'on est en train de typer.

On note $C\texttt{\{}\ \tau\ x\ \texttt{\}}$ le fait que la classe C possède un attribut x de type τ. On a $C\texttt{\{}\ \tau\ x\ \texttt{\}}$ si et seulement si :

– la classe C déclare un attribut x de type τ ; non privé si $C \neq C_0$.
– la classe C est une sous-classe de C' et C' déclare un attribut x de type τ non privé.

On note $C\texttt{\{}\ C(\tau_1, \ldots, \tau_n)\ \texttt{\}}$ le fait que l'ensemble des constructeurs de la classe C — non privés si $C \neq C_0$ — de profils plus grands que $(\tau_1, \ldots, \tau_n)$ pour la relation $\sqsubseteq$ est non vide et admet un plus petit élément pour cette relation.

Soit C une classe, m un nom de méthode et $\tau_1, \ldots, \tau_n$ un profil. On définit l'ensemble $meth(C, m, \tau_1, \ldots, \tau_n)$ comme l'union des $(n+1)$-uplets

– $(C, a_1, \ldots, a_n)$ pour toute méthode m de C de profil $(a_1, \ldots, a_n) \sqsupseteq (\tau_1, \ldots, \tau_n)$, non privée si $C \neq C_0$; et
– $(C', a_1, \ldots, a_n)$ pour toute méthode non privée m d'une sur-classe C' de C de profil $(a_1, \ldots, a_n) \sqsupseteq (\tau_1, \ldots, \tau_n)$

On note alors $C\texttt{\{}\ \tau\ m(\tau_1, \ldots, \tau_n)\ \texttt{\}}$ le fait que l'ensemble $meth(C, m, \tau_1, \ldots, \tau_n)$ est non vide et admet un unique plus petit élément pour $\sqsubseteq$, la méthode correspondante ayant pour type de retour τ.

1.3 Typage des expressions

Un contexte Γ est une suite de déclarations de types de la forme $\tau_1\ x_1,\ \ldots,\ \tau_n\ x_n$. Il ne sera utilisé que pour les variables locales, les arguments de constructeurs et de méthodes, et éventuellement `this`. On introduit le jugement $\Gamma \vdash e : \tau$ signifiant « dans le contexte Γ, l'expression e est bien typée de type τ » et le jugement $\Gamma \vdash_l e : \tau$ signifiant « dans le contexte Γ, l'expression e est une valeur gauche bien typée de type τ ». Ces jugements sont alors définis par les règles d'inférence suivantes :

$$\frac{c \text{ constante de type } \tau}{\Gamma \vdash c : \tau} \qquad \frac{}{\Gamma \vdash \texttt{null} : \texttt{typenull}} \qquad \frac{C\ \texttt{this} \in \Gamma}{\Gamma \vdash \texttt{this} : C} \qquad \frac{\tau\ x \in \Gamma}{\Gamma \vdash_l x : \tau}$$

$$\frac{x \notin \Gamma \quad C_0\texttt{\{}\ \tau\ x\ \texttt{\}}}{\Gamma \vdash_l x : \tau} \qquad \frac{\Gamma \vdash e : C \quad C\texttt{\{}\ \tau\ x\ \texttt{\}}}{\Gamma \vdash_l e.x : \tau} \qquad \frac{C\texttt{\{}\ \tau\ x\ \texttt{\}} \quad x \text{ statique}}{\Gamma \vdash_l C.x : \tau}$$

$$\frac{\Gamma \vdash_l e : \tau}{\Gamma \vdash e : \tau} \qquad \frac{\Gamma \vdash_l e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2 \quad \tau_2 \sqsubseteq \tau_1}{\Gamma \vdash e_1\ \texttt{=}\ e_2 : \tau_1}$$

$$\frac{\Gamma \vdash_l e : \texttt{int}}{\Gamma \vdash \texttt{++}e : \texttt{int}} \qquad \frac{\Gamma \vdash_l e : \texttt{int}}{\Gamma \vdash \texttt{--}e : \texttt{int}} \qquad \frac{\Gamma \vdash_l e : \texttt{int}}{\Gamma \vdash e\texttt{++} : \texttt{int}} \qquad \frac{\Gamma \vdash_l e : \texttt{int}}{\Gamma \vdash e\texttt{--} : \texttt{int}}$$

$$\frac{\Gamma \vdash e : \tau \quad \tau \in \{\texttt{int, double}\} \quad op \in \{\texttt{+, -}\}}{\Gamma \vdash op\ e : \tau} \qquad \frac{\Gamma \vdash e : \texttt{boolean}}{\Gamma \vdash\ \texttt{!}e : \texttt{boolean}}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2 \quad \tau_1 \equiv \tau_2 \quad op \in \{\texttt{==,\ !=}\}}{\Gamma \vdash e_1\ op\ e_2 : \texttt{boolean}}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2 \quad \tau_1, \tau_2 \in \{\texttt{int, double}\} \quad op \in \{\texttt{<, <=, >, >=}\}}{\Gamma \vdash e_1\ op\ e_2 : \texttt{boolean}}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2 \quad \tau_1, \tau_2 \in \{\texttt{int, double}\} \quad op \in \{\texttt{+, -, *, /}\}}{\Gamma \vdash e_1\ op\ e_2 : \tau_1 \sqcup \tau_2}$$

$$\frac{\Gamma \vdash e_1 : \texttt{int} \quad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1\ \texttt{\%}\ e_2 : \texttt{int}} \qquad \frac{\Gamma \vdash e_1 : \texttt{boolean} \quad \Gamma \vdash e_2 : \texttt{boolean} \quad op \in \{\texttt{\&\&,\ ||}\}}{\Gamma \vdash e_1\ op\ e_2 : \texttt{boolean}}$$

$$\frac{\Gamma \vdash e_1 : \mathit{String} \quad \Gamma \vdash e_2 : \tau_2 \quad \tau_2 \in \{\texttt{int, double}, \mathit{String}\}}{\Gamma \vdash e_1\ \texttt{+}\ e_2 : \mathit{String}}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \mathit{String} \quad \tau_1 \in \{\texttt{int},\ \texttt{double},\ \mathit{String}\}}{\Gamma \vdash e_1 \texttt{ + } e_2 : \mathit{String}}$$

$$\frac{\Gamma \vdash e : \mathit{String}}{\Gamma \vdash \texttt{System.out.println(}e\texttt{)} : \texttt{void}}$$

$$\frac{\Gamma \vdash e_i : \tau_i \quad C_0\texttt{\{}\ \tau\ m(\tau_1, \ldots, \tau_n)\ \texttt{\}}}{\Gamma \vdash m(e_1, \ldots, e_n) : \tau}$$

$$\frac{\Gamma \vdash e_1 : \mathit{String} \quad \Gamma \vdash e_2 : \mathit{String}}{\Gamma \vdash e_1\texttt{.equals(}e_2\texttt{)} : \texttt{boolean}}$$

$$\frac{\Gamma \vdash e : C \quad \Gamma \vdash e_i : \tau_i \quad C\texttt{\{}\ \tau\ m(\tau_1, \ldots, \tau_n)\ \texttt{\}}}{\Gamma \vdash e.m(e_1, \ldots, e_n) : \tau}$$

$$\frac{\Gamma \vdash e_i : \tau_i \quad C\texttt{\{}\ \tau\ m(\tau_1, \ldots, \tau_n)\ \texttt{\}} \quad m \text{ statique}}{\Gamma \vdash C.m(e_1, \ldots, e_n) : \tau}$$

$$\frac{\Gamma \vdash e_i : \tau_i \quad C\texttt{\{}\ C(\tau_1, \ldots, \tau_n)\ \texttt{\}}}{\Gamma \vdash \texttt{new}\ C(e_1, \ldots, e_n) : C}$$

$$\frac{\Gamma \vdash e : \tau' \quad \tau \equiv \tau'}{\Gamma \vdash \texttt{(}\tau\texttt{)}e : \tau} \qquad \frac{\Gamma \vdash e : \tau' \quad \tau \equiv \tau' \quad \tau' \in \{C, \texttt{typenull}\}}{\Gamma \vdash e\ \texttt{instanceof}\ \tau : \texttt{boolean}}$$

Petites différences avec Java.

- `System.out.println` n'est pas ici surchargée pour accepter d'autres types que *String* en argument. Pour afficher un entier i on écrira donc `System.out.println(`i `+ "")`.
- L'opération modulo (`%`) ne s'applique qu'à des entiers, alors que Java l'autorise sur des flottants.
- La classe *String* ne possède que la méthode `equals`.
- La classe *Object* ne possède aucune méthode.
- Les classes n'ont pas de constructeur par défaut.

1.4 Typage des instructions

On introduit le jugement $\Gamma \vdash i \to \Gamma'$ signifiant « dans le contexte Γ, l'instruction i est bien typée et définit le nouveau contexte Γ' ». Ce jugement est établi par les règles d'inférence suivantes :

$$\frac{}{\Gamma \vdash\ \texttt{;} \to \Gamma} \qquad \frac{\Gamma \vdash e : \tau}{\Gamma \vdash e\ \texttt{;} \to \Gamma} \qquad \frac{x \notin \Gamma}{\Gamma \vdash \tau\ x\ \texttt{;} \to \Gamma, \tau\ x} \qquad \frac{x \notin \Gamma \quad \Gamma \vdash e : \tau' \quad \tau' \sqsubseteq \tau}{\Gamma \vdash \tau\ x \texttt{ = } e\ \texttt{;} \to \Gamma, \tau\ x}$$

$$\frac{\Gamma \vdash e : \texttt{boolean} \quad \Gamma \vdash i_1 \to \Gamma_1 \quad \Gamma \vdash i_2 \to \Gamma_2}{\Gamma \vdash \texttt{if (}e\texttt{)}\ i_1\ \texttt{else}\ i_2 \to \Gamma}$$

$$\frac{\Gamma \vdash e : \texttt{boolean} \quad \Gamma \vdash i \to \Gamma'}{\Gamma \vdash \texttt{while(}e\texttt{)}\ i \to \Gamma}$$

$$\frac{\Gamma \vdash i_1 \to \Gamma_1 \quad \Gamma_1 \vdash e : \texttt{boolean} \quad \Gamma_1 \vdash i_2 \to \Gamma_2 \quad \Gamma_1 \vdash i_3 \to \Gamma_3}{\Gamma \vdash \texttt{for(}i_1\ \texttt{;}\ e\ \texttt{;}\ i_2\texttt{)}\ i_3 \to \Gamma}$$

$$\frac{\Gamma \vdash i_1 \to \Gamma_1 \quad \Gamma_1 \vdash i_2 \to \Gamma_2}{\Gamma \vdash i_1\ \texttt{;}\ i_2 \to \Gamma_2} \qquad \frac{\Gamma \vdash i_1 \to \Gamma_1 \quad \Gamma_1 \vdash i_2 \to \Gamma_2}{\Gamma \vdash i_1\texttt{,}\ i_2 \to \Gamma_2}$$

$$\frac{\Gamma \vdash i \to \Gamma'}{\Gamma \vdash \texttt{\{i\}} \to \Gamma} \qquad \frac{}{\Gamma \vdash \texttt{return ;} \to \Gamma} \qquad \frac{\Gamma \vdash e : \tau}{\Gamma \vdash \texttt{return}\ e\ \texttt{;} \to \Gamma}$$

De plus, on a les équivalences suivantes :
- `if (`e_1`)` e_2 équivaut à `if (`e_1`)` e_2 `else ;`
- si e_1 ou e_3 est omis dans `for(`e_1 `;`e_2 `;`e_3`)` alors il équivaut à ;
- si e_2 est omis dans `for(`e_1 `;`e_2 `;`e_3`)` alors il équivaut à `true`
- $\tau\, x_1 = e_1, x_2 = e_2, \dots x_n = e_n$ équivaut à $\tau\, x_1 = e_1; \tau\, x_2 = e_2; \dots \tau\, x_n = e_n$.

1.5 Typage des classes

1.5.1 Conditions d'existence et d'unicité

Pour être sémantiquement correct, un fichier doit tout d'abord respecter les contraintes suivantes :
- chaque classe ne peut être définie qu'une seule fois ;
- une classe ne peut hériter d'une classe qui n'est pas définie, ni hériter de la classe *String* ;
- la relation d'héritage ne peut admettre de cycle.

Notez bien que l'ordre d'apparition des classes dans le fichier n'a aucune importance : en tout point, on peut référencer une classe définie ailleurs dans le fichier (avant ou après).

D'autre part, les conditions suivantes doivent être remplies :
- tous les attributs d'une même classe doivent porter des noms différents ;
- tous les constructeurs d'une même classe doivent avoir des profils différents ; de même nom que la classe et déclarés non statiques.
- deux méthodes de même nom d'une même classe doivent avoir des *arguments* différant en nombre et/ou en types.

1.5.2 Redéfinition

Une méthode m est dite *redéfinie* si elle apparaît avec un même profil (arguments de mêmes types) dans une classe C et une sous-classe C' de C. Dans ce cas, cette méthode doit vérifier les conditions suivantes :
- ne pas avoir un caractère plus privée dans C' que dans C ;
- ne pas être statique dans C et non-statique dans C', ou vice-versa ;
- ne pas avoir deux types de retour différents dans C et C'.

1.5.3 Typage des attributs, constructeurs et méthodes

Soit C_0 la classe courante. Le contexte initial de typage est $\Gamma_0 = C_0\ \texttt{this}$, ne contenant que la variable `this` de type C_0.

Typage des attributs. Un attribut est typé à la manière d'une variable locale :
- pour un attribut non initialisé $\tau\ x$, il n'y a rien à faire (on se contente de vérifier que le type τ est bien formé) ;
- pour un attribut initialisé $\tau\ x = e$, on vérifie que e est bien typée d'un type sous-type de τ. Si l'attribut x est statique (variable de classe), le typage est effectué dans le contexte vide, sinon dans Γ_0.

Remarque : on ne demande pas de tester que les attributs sont initialisés "dans le bon ordre", c'est-à-dire sans faire référence à un attribut introduit en-dessous (*forward reference*) ; on pourra supposer que c'est le cas.

Typage des constructeurs. Un constructeur C_0`(`$\tau_1\ x_1, \dots, \tau_n\ x_n$`){`$i$`}` est bien formé si tous les identificateurs sont différents deux à deux et si la liste d'instructions i est bien typée dans le contexte $\Gamma_0, \tau_1\ x_1, \dots, \tau_n\ x_n$.

Typage des méthodes. Une méthode $\tau\ m$`(`$\tau_1\ x_1, \dots, \tau_n\ x_n$`){`$i$`}` est bien formée si tous les identificateurs sont différents deux à deux et si la liste d'instructions i est bien typée dans le contexte $\Gamma_0, \tau_1\ x_1, \dots, \tau_n\ x_n$ si m est non-statique, et dans le contexte $\tau_1\ x_1, \dots, \tau_n\ x_n$ sinon.

D'autre part, toute occurrence de l'instruction `return` dans i doit retourner une valeur d'un type sous-type de τ. Enfin, lorsque le type τ n'est pas `void`, tout flot d'exécution possible dans i doit contenir (au moins) une instruction `return`.

Remarque : à la différence du compilateur Java, on ne cherchera pas à signaler le code inatteignable (i.e. derrière un `return`).

1.6 Indications

Comment procéder. On suggère fortement de réaliser l'analyse sémantique en trois phases distinctes, à savoir :

1. déclaration des classes et vérification de leur unicité,
2. construction de la relation d'héritage, déclaration des attributs, constructeurs et méthodes de toutes les classes (avec les vérifications adéquates),
3. typage des expressions d'initialisation des attributs et des corps des constructeurs et des méthodes.

Vous restez cependant libres d'utiliser toute autre méthode de votre choix qui se révélerait correcte.

Tests. En cas de doute concernant un point de sémantique, vous pouvez utiliser le compilateur Java comme référence. Vous pouvez d'ailleurs vous inspirer de ses messages d'erreur pour votre compilateur (en les traduisant ou non en français).

Anticipation. Dans la troisième partie (production de code), certaines informations provenant du typage seront nécessaires, telles que par exemple la nécessité de transformer une valeur entière en une valeur réelle, la détermination de la méthode appelée, etc. Il vous est conseillé d'anticiper ces besoins en programmant cette deuxième partie. Deux méthodes (au moins) sont envisageables :

- décorer votre arbre de syntaxe abstraite par des informations de typage (telle que le type de chaque sous-expression) ;
- construire un arbre de syntaxe abstraite plus précis.

Notez que dans les deux cas, cela signifie que vos fonctions de typage ne se contentent pas de parcourir les arbres de syntaxe abstraite mais en retournent de nouveaux.

2 Travail demandé

Adapter votre compilateur pour qu'il effectue la passe d'analyse sémantique, à l'issue de l'analyse syntaxique. En plus du fichier Java en argument, votre compilateur devra reconnaître désormais deux options, `-parse-only` et `-type-only`. La première spécifie de s'arrêter juste après l'analyse syntaxique. La seconde est sans effet dans cette deuxième partie et son utilisation sera justifiée par la suite.

En plus des erreurs lexicales et syntaxiques, votre compilateur devra maintenant afficher les erreurs sémantiques et, le cas échéant, terminer avec le code de sortie 1 (`exit 1`). Les erreurs sémantiques seront signalées de la même manière que les erreurs lexicales et syntaxiques ; par exemple :

```
File "test.java", line 21, characters 13-14:
this expression has type int but is expected to have type String
```

Le comportement en cas d'absence d'erreur ou d'erreur du compilateur ne change pas.

Modalités de remise de votre projet. Votre projet doit se présenter sous forme d'une archive tar compressée (option "z" de tar), appelée *vos_noms*`.tgz` qui doit contenir un répertoire appelé *vos_noms* (exemple : `dupont-durand.tgz`). Dans ce répertoire doivent se trouver les *sources* de votre programme (ne donnez pas les fichiers compilés). Quand on se place dans ce répertoire, la commande `make` doit créer votre compilateur, qui sera appelé `jc`. La commande `make clean` doit effacer tous les fichiers que `make` a engendré et ne laisser dans le répertoire que les fichiers sources.

Votre projet est à rendre à votre chargé de TP par e-mail[1] au plus tard le **lundi 1er décembre avant 10h00**.

[1] `sylvain.conchon@lri.fr`, `jean-christophe.filliatre@lri.fr`, `julien.forest@lri.fr`, `clement.renard@inria.fr`, `julien.signoles@lri.fr` ,`julien.narboux@inria.fr`