Compilateur Pascal avec modules
deuxième partie (v1.2): analyse sémantique

à rendre avant le jeudi 12 novembre 10 heures

1 Analyse sémantique

Cette deuxième partie consiste à vérifier la conformité du fichier source, une fois l'analyse syntaxique effectuée avec succès.

1.1 Types et environnements de typage

Dans tout ce qui suit, les expressions de types sont de la forme suivante :

$$\tau \quad ::= \quad \texttt{boolean} \mid \texttt{integer} \mid \texttt{array[}n_1\texttt{..}n_2\texttt{] of } \tau$$

où n_1 et n_2 désignent deux constantes entières. On note $\tau \neq \texttt{array}$ pour signifier que le type τ n'est pas un type de tableau.

$$\begin{array}{lcl} \Gamma & ::= & \emptyset \mid x : \delta, \Gamma \\ \delta & ::= & \tau \mid (m_1\ \tau_1, \ldots, m_n\ \tau_n) \mid \Gamma \mid (\Gamma_1, \ldots, \Gamma_n) \to \Gamma \end{array}$$

Un environnement de typage Γ est une suite de déclarations $x : \delta$ de la forme $x : \tau$ pour une variable, $x : (m_1\ \tau_1, \ldots, m_n\ \tau_n)$ pour une procédure (où les m_i désignent les spécificateurs optionnels `var`), $x : \Gamma$ pour les modules, ou $x : (\Gamma_1, \ldots, \Gamma_n) \to \Gamma$ pour les foncteurs.

Une déclaration δ_1 est *plus précise* qu'une déclaration δ_2 (noté, $\delta_1 \leq \delta_2$) si l'une des règle suivante s'applique :

$$\tau \leq \tau \qquad (m_1\ \tau_1, \ldots, m_n\ \tau_n) \leq (m_1\ \tau_1, \ldots, m_n\ \tau_n)$$

$$\frac{\Gamma' \sqsubseteq \Gamma}{\Gamma \leq \Gamma'} \qquad \frac{\Gamma_i \sqsubseteq \Gamma'_i \qquad \Gamma' \sqsubseteq \Gamma}{(\Gamma_1, \ldots, \Gamma_n) \to \Gamma \leq (\Gamma'_1, \ldots, \Gamma'_n) \to \Gamma'}$$

La relation $\Gamma_1 \sqsubseteq \Gamma_2$ est l'inclusion d'environnement. Si on a deux environnements Γ_1 et Γ_2, on dira que Γ_1 est *inclus* dans Γ_2 si pour toute déclaration $x : \delta_1$ de Γ_1, il existe une déclaration $x : \delta_2$ dans Γ_2 telle que $\delta_2 \leq \delta_1$.

On note les noms qualifiés p :

$$p \quad ::= \quad x \mid p\,.\,x$$

Aller cherche le type associée à un nom qualifié est défini par :

$$\Gamma(x) \quad = \quad \begin{cases} \delta & \text{si } \Gamma = x : \delta, \Gamma' \\ \Gamma'(x) & \text{si } \Gamma = y : \delta, \Gamma' \text{ et } x \neq y \end{cases}$$

$$\Gamma(p.x) \quad = \quad \Gamma'(x) \text{ avec } \Gamma(p) = \Gamma'$$

On peut remarquer que pour aller chercher le type associée à un nom $p.x$ dans l'environnement, il faut que p soit un module.

1.2 Typage des expressions

On introduit le jugement $\Gamma \vdash e : \tau$ signifiant « dans l'environnement Γ, l'expression e est bien typée de type τ » et le jugement $\Gamma \vdash_l e : \tau$ signifiant « dans l'environnement Γ, l'expression e est une valeur gauche bien typée de type τ ». Ces jugements sont alors définis par les règles d'inférence suivantes :

$$\frac{c \text{ constante de type } \tau}{\Gamma \vdash c : \tau} \qquad \frac{\Gamma(p) = \tau}{\Gamma \vdash_l p : \tau} \qquad \frac{\Gamma \vdash_l e : \tau}{\Gamma \vdash e : \tau}$$

$$\frac{\Gamma \vdash_l e_1 : \texttt{array[}n_1\texttt{..}n_2\texttt{] of } \tau \quad \Gamma \vdash e_2 : \texttt{integer}}{\Gamma \vdash_l e_1\texttt{[}e_2\texttt{]} : \tau}$$

$$\frac{\Gamma \vdash e : \texttt{integer} \quad op \in \{\texttt{+}, \texttt{-}\}}{\Gamma \vdash op\ e : \texttt{integer}} \qquad \frac{\Gamma \vdash e : \texttt{boolean}}{\Gamma \vdash \texttt{not } e : \texttt{boolean}}$$

$$\frac{\Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau \quad \tau \neq \texttt{array} \quad op \in \{\texttt{<}, \texttt{<=}, \texttt{>}, \texttt{>=}, \texttt{=}, \texttt{<>}\}}{\Gamma \vdash e_1\ op\ e_2 : \texttt{boolean}}$$

$$\frac{\Gamma \vdash e_1 : \texttt{integer} \quad \Gamma \vdash e_2 : \texttt{integer} \quad op \in \{\texttt{+}, \texttt{-}, \texttt{*}, \texttt{/}, \texttt{div}, \texttt{mod}\}}{\Gamma \vdash e_1\ op\ e_2 : \texttt{integer}}$$

$$\frac{\Gamma \vdash e_1 : \texttt{boolean} \quad \Gamma \vdash e_2 : \texttt{boolean} \quad op \in \{\texttt{and}, \texttt{or}\}}{\Gamma \vdash e_1\ op\ e_2 : \texttt{boolean}}$$

1.3 Typage des instructions

On introduit le jugement $\Gamma \Vdash i$ signifiant « dans l'environnement Γ, l'instruction i est bien typée ». Ce jugement est établi par les règles d'inférence suivantes :

$$\frac{\Gamma \vdash_l l : \tau \quad \Gamma \vdash e : \tau}{\Gamma \Vdash l\texttt{:=}\ e}$$

$$\frac{\Gamma(p) = (m_1\ \tau_1, \ldots, m_n\ \tau_n) \quad (\Gamma \vdash_l e_i : \tau_i \text{ si } m_i = \texttt{var},\ \Gamma \vdash e_i : \tau_i \text{ sinon})}{\Gamma \Vdash p(e_1, \ldots, e_n)}$$

$$\frac{\Gamma \vdash e_i : \tau_i \quad \tau_i \neq \texttt{array}}{\Gamma \Vdash \texttt{write}(e_1, \ldots, e_n)} \qquad \frac{\Gamma \vdash e_i : \tau_i \quad \tau_i \neq \texttt{array}}{\Gamma \Vdash \texttt{writeln}(e_1, \ldots, e_n)}$$

$$\frac{\Gamma \vdash e : \texttt{boolean} \quad \Gamma \Vdash i_1}{\Gamma \Vdash \texttt{if } e \texttt{ then } i_1} \qquad \frac{\Gamma \vdash e : \texttt{boolean} \quad \Gamma \Vdash i_1 \quad \Gamma \Vdash i_2}{\Gamma \Vdash \texttt{if } e \texttt{ then } i_1 \texttt{ else } i_2}$$

$$\frac{\Gamma \vdash e : \texttt{boolean} \quad \Gamma \Vdash i}{\Gamma \Vdash \texttt{while } e \texttt{ do } i}$$

$$\frac{\Gamma(p) = \texttt{integer} \quad \Gamma \vdash e_1 : \texttt{integer} \quad \Gamma \vdash e_2 : \texttt{integer} \quad \Gamma \Vdash i}{\Gamma \Vdash \texttt{for } p\texttt{:=}\ e_1 \texttt{ to } e_2 \texttt{ do } i}$$

$$\frac{\Gamma \Vdash i_j}{\Gamma \Vdash \texttt{begin } i_1\texttt{; } \ldots\texttt{; } i_n \texttt{ end}}$$

Note : Les règles pour les appels de procédure, `write` et `writeln` s'appliquent également dans le cas où il n'y a aucun argument ($n = 0$).

1.4 Typage des signatures

On dit qu'un type τ est *bien formé*, et on note τ bf, si les indices dans les déclarations de tableaux sont corrects. Cela correspond aux règles d'inférence suivantes :

$$\frac{}{\texttt{boolean}\ \mathsf{bf}} \qquad \frac{}{\texttt{integer}\ \mathsf{bf}} \qquad \frac{n_1 \leq n_2 \quad \tau\ \mathsf{bf}}{\texttt{array[}n_1\texttt{..}n_2\texttt{]}\ \texttt{of}\ \tau\ \mathsf{bf}}$$

On introduit ensuite le jugement $d \rightsquigarrow \Gamma$ signifiant « la suite de déclarations d est bien formée et produit un environnement Γ ». Ce jugement est définit par les règles suivantes :

$$\frac{\tau\ \mathsf{bf}}{\texttt{var}\ x_1, \ldots, x_n : \tau\texttt{;} \rightsquigarrow x_1 : \tau, \ldots, x_n : \tau}$$

$$\frac{\tau_i\ \mathsf{bf}}{\texttt{procedure}\ x(m_1\ x_1 : \tau_1, \ldots, m_n\ x_n : \tau_n)\texttt{;} \rightsquigarrow x : (m_1\ \tau_1, \ldots, m_n\ \tau_n)}$$

$$\frac{d \rightsquigarrow \Gamma}{\texttt{module}\ x : \texttt{sig}\ d\ \texttt{end;} \rightsquigarrow x : \Gamma}$$

$$\frac{d_i \rightsquigarrow \Gamma_i \qquad d \rightsquigarrow \Gamma}{\texttt{module}\ x(x_1\texttt{:}\ \texttt{sig}\ d_1\ \texttt{end}, \ldots, x_n\texttt{:}\ \texttt{sig}\ d_n\ \texttt{end}) : \texttt{sig}\ d\ \texttt{end;} \rightsquigarrow x : (\Gamma_1, \ldots, \Gamma_n) \to \Gamma}$$

1.5 Typage des modules

On introduit ensuite le jugement $\Gamma \vdash d \rightsquigarrow \Gamma'$ signifiant « dans l'environnement Γ, la suite de définitions d est bien formée et définit le nouvel environnement Γ' ».

Définitions de variables. On commence par définir le jugement $\Gamma \vdash d \rightsquigarrow \Gamma'$ pour les définitions de variables de la manière suivante :

$$\frac{\texttt{var}\ x_1, \ldots, x_n : \tau\texttt{;} \rightsquigarrow \Gamma'}{\Gamma \vdash \texttt{var}\ x_1, \ldots, x_n : \tau\texttt{;} \rightsquigarrow \Gamma'}$$

Définitions de procédures. On étend le jugement $\Gamma \vdash d \rightsquigarrow \Gamma'$ aux déclarations de procédures de la manière suivante :

$$\frac{\begin{array}{c}\texttt{procedure}\ x(m_1\ x_1 : \tau_1, \ldots, m_n\ x_n : \tau_n)\texttt{;} \rightsquigarrow \Gamma_x \\ \texttt{var}\ x_1 : \tau_1\texttt{;}\ \ldots\texttt{;var}\ x_n : \tau_n\texttt{;} \rightsquigarrow \Gamma_a \qquad d \rightsquigarrow \Gamma_v \qquad \Gamma_x, \Gamma_a, \Gamma_v, \Gamma \Vdash b\end{array}}{\Gamma \vdash \texttt{procedure}\ x(m_1\ x_1 : \tau_1, \ldots, m_n\ x_n : \tau_n)\texttt{;}\ d\ b \rightsquigarrow \Gamma_x}$$

Définitions modules. Avant d'étendre le jugement $\Gamma \vdash d \rightsquigarrow \Gamma'$ aux définitions de modules, on définit le jugement $\Gamma \vdash_m m : \Gamma'$ signifiant « dans l'environnement Γ l'expression de module m est bien formée et a pour signature Γ' ». Si l'expression de module est de la forme `struct` $d_1 \cdots d_n$ i `end`, le typage consiste à typer successivement les déclarations dans le contexte étendu par chaque nouvelle déclaration, d'où les règles :

$$\frac{\Gamma \Vdash i}{\Gamma \vdash_m \emptyset\ i : \emptyset} \qquad \frac{\Gamma \vdash d_1 \rightsquigarrow \Gamma' \quad \Gamma, \Gamma' \vdash_m d_2 \cdots d_n\ i : \Gamma''}{\Gamma \vdash_m d_1\ d_2 \cdots d_n\ i : \Gamma', \Gamma''} \qquad \frac{\Gamma(p) = \Gamma' \quad \Gamma, \Gamma' \vdash_m d_2 \cdots d_n\ i : \Gamma''}{\Gamma \vdash_m \texttt{open}\ p;\ d_2 \cdots d_n\ i : \Gamma''}$$

Si l'expression de module est une application de foncteur la règle suivante s'applique :

$$\frac{\Gamma(p) = (\Gamma_1, \ldots, \Gamma_n) \to \Gamma' \qquad \Gamma(p_i) = \Gamma'_i \qquad \Gamma_i \sqsubseteq \Gamma'_i}{\Gamma \vdash_m p(p_1, \ldots, p_n) : \Gamma'}$$

On peut maintenant étendre le jugement $\Gamma \vdash d \rightsquigarrow \Gamma'$ aux définitions de modules :

$$\frac{\Gamma \vdash_m m : \Gamma'}{\Gamma \vdash \texttt{module}\ x\texttt{:=}\ m \rightsquigarrow x : \Gamma'}$$

$$\frac{d \rightsquigarrow \Gamma_{sig} \qquad \Gamma \vdash_m m : \Gamma' \qquad \Gamma_{sig} \sqsubseteq \Gamma'}{\Gamma \vdash \texttt{module}\ x\texttt{: sig}\ d\ \texttt{end :=}\ m \rightsquigarrow x : \Gamma_{sig}}$$

$$\frac{d_i \rightsquigarrow \Gamma_i \qquad x_1 : \Gamma_1, \ldots, x_n : \Gamma_n, \Gamma \vdash_m m : \Gamma'}{\Gamma \vdash \texttt{module}\ x\texttt{(}x_1\texttt{: sig}\ d_1\ \texttt{end; } \ldots\texttt{; }x_n\texttt{: sig}\ d_n\ \texttt{end) :=}\ m \rightsquigarrow x : (\Gamma_1, \ldots, \Gamma_n) \to \Gamma'}$$

$$\frac{d_i \rightsquigarrow \Gamma_i \qquad d \rightsquigarrow \Gamma_{sig} \qquad x_1 : \Gamma_1, \ldots, x_n : \Gamma_n, \Gamma \vdash_m m : \Gamma' \qquad \Gamma_{sig} \sqsubseteq \Gamma'}{\Gamma \vdash \texttt{module}\ x\texttt{(}x_1\texttt{: sig}\ d_1\ \texttt{end; } \ldots\texttt{; }x_n\texttt{: sig}\ d_n\ \texttt{end)}\texttt{: sig}\ d\ \texttt{end :=}\ m \rightsquigarrow x : (\Gamma_1, \ldots, \Gamma_n) \to \Gamma_{sig}}$$

Fichiers. On introduit finalement le jugement $\Gamma \vdash_f d_1 \cdots d_n\ i$ signifiant « dans l'environnement Γ le fichier constitué par la suite de définitions d_1, …, d_n et l'instruction i est bien formé » :

$$\frac{\Gamma \Vdash i}{\Gamma \vdash_f \emptyset\ i} \qquad \frac{\Gamma \vdash d_1 \rightsquigarrow \Gamma' \qquad \Gamma, \Gamma' \vdash_f d_2 \cdots d_n\ i}{\Gamma \vdash_f d_1\ d_2 \cdots d_n\ i} \qquad \frac{\Gamma(p) = \Gamma' \qquad \Gamma, \Gamma' \vdash_f d_2 \cdots d_n\ i}{\Gamma \vdash_f \texttt{open}\ p;\ d_2 \cdots d_n\ i}$$

Règles d'unicité. Enfin, on vérifiera l'unicité :

- des symboles globaux (variables *globales* à un module, procédures et modules) sur l'ensemble du module ;
- des symboles locaux (paramètres formels, variables locales) dans chaque flot de contrôle de chaque procédure.

On notera qu'un symbole local peut porter le même nom qu'un symbole global. De même, un symbole global peut porter le même nom qu'un autre symbole global définit dans un sous-module du même module et dans un autre module.

Procédures prédéfinies. Les procédures suivantes sont supposées prédéfinies et devront être connues à l'analyse sémantique :

```
open_screen(w : integer; h : integer);
clear_screen;
set_color(r : integer; g : integer; b : integer);
draw_point(x : integer; y : integer);
draw_line(x1 : integer; y1 : integer; x2 : integer; y2 : integer);
draw_circle(x : integer; y : integer; r : integer);
draw_rect(x : integer; y : integer; w : integer; h : integer);
fill_rect(x : integer; y : integer; w : integer; h : integer);
usleep(n : integer);
```

1.6 Indications

Tests. En cas de doute concernant un point de sémantique, vous pouvez utiliser un compilateur Pascal comme référence sur le fragment ne comprenant pas l'extension de modules (sauf pour l'opérateur `/` et les arguments de type `array`). Vous pouvez d'ailleurs vous inspirer de ses messages d'erreur pour votre compilateur.

Anticipation. Dans la troisième partie (production de code), certaines informations provenant du typage seront nécessaires. Il vous est conseillé d'anticiper ces besoins en programmant cette deuxième partie. Une méthode envisageable est de décorer votre arbre de syntaxe abstraite par des informations de typage (telle que le type de chaque sous-expression). Notez que 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 Pascal 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.pas", line 21, characters 13-14:
this expression has type integer but is expected to have type boolean
```

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

Votre code devra comprendre au moins un fichier `typing.ml` contenant les principales fonctions d'analyse sémantique. L'introduction d'autres fichiers devra être justifiée dans le rapport.

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 `compilo`. 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 doit également s'accompagner d'un rapport (5 pages maximum) expliquant les difficultés rencontrées et les solutions techniques adoptées. Le rapport devra être rendu directement dans l'archive contenant les sources de votre projet (format pdf).

Votre projet est à rendre à votre chargé de TP par e-mail[1] au plus tard le **jeudi 12 novembre à 10h00**.

[1] `francois.bobot@lri.fr`, `delphine.longuet@lri.fr`, `louis.mandel@lri.fr`