École Normale Supérieure 2013–2014

projet de compilation

Mini C++

partie 2 — à rendre le 10 janvier

version 1 — 4 décembre 2013

L'analyse syntaxique a été réalisée dans la première partie du projet. Dans ce qui suit, on suppose que l'expression `true` (resp. `false`) a été remplacée par la constante `1` (resp. `0`) et que l'expression e`->`id a été remplacée par `(*`e`).`id.

1 Typage statique

Une fois l'analyse syntaxique effectuée avec succès, on vérifie la conformité du fichier source. Dans tout ce qui suit, les types sont de la forme suivante :

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

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. D'autre part, `typenull` est introduit pour pouvoir donner un type à l'expression `NULL`. On dit qu'un type τ est bien formé, et on note τ bf, s'il est soit `int`, soit une classe C déclarée dans le fichier source, soit un type de la forme τ'`*` avec τ' un type bien formé. On dit qu'un type τ est numérique, et on note alors $num(\tau)$, si τ est `int`, ou un type pointeur, c'est-à-dire `typenull` ou de la forme τ'`*`.

On note que les références (c'est-à-dire les types de la forme `int&`) ne sont pas incluses dans la grammaire ci-dessus. On ne les considère donc pas comme des types dans ce qui suit. On attache plutôt la présence de `&` à la variable (ou à la fonction, s'il s'agit du type de retour d'une fonction), pour signifier qu'il s'agit d'une référence. Une double référence, c'est-à-dire une déclaration de la forme τ `& &`, n'est pas admise.

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 : C_2 contenues dans le source. 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 :

$$\frac{}{\texttt{int} \sqsubseteq \texttt{int}} \qquad \frac{C_1 \longrightarrow C_2}{C_1 \sqsubseteq C_2} \qquad \frac{}{\texttt{typenull} \sqsubseteq \tau\texttt{*}} \qquad \frac{\tau_1 \sqsubseteq \tau_2}{\tau_1\texttt{*} \sqsubseteq \tau_2\texttt{*}}$$

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 ».

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 Champs, constructeurs et méthodes d'une classe

On note $C\{\ \tau\ x\ \}$ le fait que la classe C possède un *unique* champ visible x de type τ. Ce champ peut être déclaré dans la classe C ou hérité d'une sur-classe de C. Il est important de noter qu'une classe peut avoir plusieurs champs visibles de même nom x, hérités de plusieurs classes différentes ; dans ce cas, une référence à x est ambiguë et on n'a pas $C\{\ \tau\ x\ \}$.

On note $C\{\ C(\tau_1, \ldots, \tau_n)\ \}$ le fait que l'ensemble des constructeurs de la classe C de profils plus grands que $(\tau_1, \ldots, \tau_n)$ pour la relation $\sqsubseteq$ est non vide et admet un unique 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 $(a_1, \ldots, a_n) \sqsupseteq (\tau_1, \ldots, \tau_n)$; et
- $(C', a_1, \ldots, a_n)$ pour toute méthode m d'une sur-classe C' de C de profil $(a_1, \ldots, a_n) \sqsupseteq (\tau_1, \ldots, \tau_n)$.

On note alors $C\{\ \tau\ m(\tau_1, \ldots, \tau_n)\ \}$ 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 variables de la forme $\tau\ x$ et de déclarations de fonctions de la forme $\tau\ f(\tau_1, \ldots, \tau_n)$. On note $\Gamma\{\ \tau\ f(\tau_1, \ldots, \tau_n)\ \}$ le fait que l'ensemble des fonctions de Γ de profils plus grands que $(\tau_1, \ldots, \tau_n)$ pour la relation $\sqsubseteq$ est non vide et admet un unique plus petit élément pour cette relation.

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 :

$$\frac{n \text{ constante de type } \texttt{int}}{\Gamma \vdash n : \texttt{int}} \qquad \frac{}{\Gamma \vdash \texttt{NULL} : \texttt{typenull}} \qquad \frac{C\texttt{*}\ \texttt{this} \in \Gamma}{\Gamma \vdash \texttt{this} : C\texttt{*}} \qquad \frac{\tau\ x \in \Gamma}{\Gamma \vdash_l x : \tau}$$

$$\frac{x \notin \Gamma \quad C_0\texttt{*}\ \texttt{this} \in \Gamma \quad C_0\{\ \tau\ x\ \}}{\Gamma \vdash_l x : \tau} \qquad \frac{x \notin \Gamma \quad C_0\texttt{*}\ \texttt{this} \in \Gamma \quad C_0 \sqsubseteq C \quad C\{\ \tau\ x\ \}}{\Gamma \vdash_l C\texttt{::}x : \tau}$$

$$\frac{\Gamma \vdash e : C \quad C\{\ \tau\ x\ \}}{\Gamma \vdash_l e\texttt{.}x : \tau} \qquad \frac{\Gamma \vdash e : \tau\texttt{*}}{\Gamma \vdash_l \texttt{*}e : \tau} \qquad \frac{\Gamma \vdash_l e : \tau}{\Gamma \vdash \texttt{\&}e : \tau\texttt{*}} \qquad \frac{\Gamma \vdash_l e : \tau}{\Gamma \vdash e : \tau}$$

$$\frac{\Gamma \vdash_l e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2 \quad \tau_2 \sqsubseteq \tau_1 \quad num(\tau_1)}{\Gamma \vdash e_1 \texttt{ = } e_2 : \tau_1}$$

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

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

$$\frac{\Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau \quad num(\tau) \quad op \in \{\texttt{==}, \texttt{ !=}\}}{\Gamma \vdash e_1\ op\ e_2 : \texttt{int}}$$

$$\frac{\Gamma \vdash e_1 : \texttt{int} \quad \Gamma \vdash e_2 : \texttt{int} \quad op \in \{\texttt{<}, \texttt{<=}, \texttt{>}, \texttt{>=}, \texttt{+}, \texttt{-}, \texttt{*}, \texttt{/}, \texttt{\%}, \texttt{\&\&}, \texttt{||}\}}{\Gamma \vdash e_1 \ op \ e_2 : \texttt{int}}$$

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

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

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

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

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

Pour les quatre dernières règles (appel de fonction ou de méthode), le résultat est une valeur gauche (c'est-à-dire qu'il faut lire $\vdash_l$ au lieu de $\vdash$) dès lors qu'il s'agit d'une référence.

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 :

$$\frac{}{\Gamma \vdash \texttt{;} \to \Gamma} \qquad \frac{\Gamma \vdash e : \tau}{\Gamma \vdash e\texttt{;} \to \Gamma} \qquad \frac{\Gamma \vdash i_1 \to \Gamma_1 \quad \Gamma_1 \vdash i_2 \to \Gamma_2}{\Gamma \vdash i_1\texttt{;}\ i_2 \to \Gamma_2} \qquad \frac{\Gamma \vdash i \to \Gamma'}{\Gamma \vdash \texttt{\{}i\texttt{\}} \to \Gamma}$$

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

$$\frac{x \notin \Gamma \quad C\ bf \quad \Gamma \vdash e_i : \tau_i \quad C\{\ C(\tau_1, \ldots, \tau_n)\ \}}{\Gamma \vdash C\ x\ \texttt{=}\ C(e_1, \ldots, e_n)\texttt{;} \to \Gamma, C\ x}$$

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

$$\frac{\Gamma \vdash e_1\texttt{;} \to \Gamma \quad \Gamma \vdash e_2 : \texttt{int} \quad \Gamma \vdash e_3\texttt{;} \to \Gamma \quad \Gamma \vdash i \to \Gamma_1}{\Gamma \vdash \texttt{for(}e_1\texttt{;}\ e_2\texttt{;}\ e_3\texttt{)}\ i \to \Gamma}$$

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

$$\frac{e_i \text{ est une chaîne ou } \Gamma \vdash e_i : \texttt{int}}{\Gamma \vdash \texttt{std::cout <<}\ e_1\ \texttt{<<} \cdots \texttt{<<}\ e_n \to \Gamma}$$

De plus, on a les équivalences suivantes :

- La construction `if (`e_1`)` e_2 équivaut à `if (`e_1`)` e_2 `else;`.
- Dans la construction `for(`e_1`;`e_2`;`e_3`)`, e_1 et e_3 sont des listes d'expressions, qui doivent être typées (et plus tard compilées) comme si elles étaient séparées par des points-virgules.
- Si e_2 est omis dans `for(`e_1`;`e_2`;`e_3`)` alors il équivaut à `true`.

1.5 Typage des fichiers

Conditions d'existence et d'unicité. Pour être sémantiquement correct, un fichier ne doit pas introduire deux classes de même nom. D'autre part, les conditions suivantes doivent être remplies :

- Tous les champs d'une même classe doivent porter des noms différents (mais en revanche ils peuvent porter le même nom que des champs hérités d'une sur-classe).
- Tous les constructeurs d'une même classe doivent avoir des profils différents et porter le même nom que la classe. Si une classe n'introduit aucun constructeur, alors un constructeur sans argument est introduit implicitement.
- Deux méthodes de même nom d'une même classe doivent avoir des *arguments* différant en nombre et/ou en types.
- Deux fonctions de même nom doivent avoir des *arguments* différant en nombre et/ou en types.

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 et qu'elle est déclarée `virtual` dans la classe C. Dans ce cas, ces deux méthodes doivent avoir le même type de retour.

Typage des champs. Pour un champ $\tau\ x$, on doit vérifier que le type τ est bien formé. Par ailleurs, x ne doit pas être une référence.

Dans ce qui suit, Γ_g désigne l'environnement des variables globales visibles au point de programme concerné, c'est-à-dire introduites préalablement.

Typage des constructeurs. Un constructeur $C\texttt{(}\tau_1\ x_1, \ldots, \tau_n\ x_n\texttt{)\{}i\texttt{\}}$ est bien formé si tous les identificateurs x_i sont deux à deux distincts, si tous les types τ_i sont bien formés, et si la liste d'instructions i est bien typée dans le contexte $\Gamma_g, C\texttt{* this}, \tau_1\ x_1, \ldots, \tau_n\ x_n$.

Typage des méthodes. Une méthode $\tau\ C\texttt{::}m\texttt{(}\tau_1\ x_1, \ldots, \tau_n\ x_n\texttt{)\{}i\texttt{\}}$ est bien formée si tous les identificateurs x_i sont deux à deux distincts, si tous les types τ_i sont bien formés, et si la liste d'instructions i est bien typée dans le contexte $\Gamma_g, C\texttt{* this}, \tau_1\ x_1, \ldots, \tau_n\ x_n$. D'autre part, toute occurrence de l'instruction `return` dans i doit renvoyer une valeur d'un type sous-type de τ.

Typage des fonctions. Une fonction $\tau\ f\texttt{(}\tau_1\ x_1, \ldots, \tau_n\ x_n\texttt{)\{}i\texttt{\}}$ est bien formée si tous les identificateurs x_i sont deux à deux distincts, si tous les types τ_i sont bien formés, et si la liste d'instructions i est bien typée dans le contexte $\Gamma_g, \tau_1\ x_1, \ldots, \tau_n\ x_n$. D'autre part, toute occurrence de l'instruction `return` dans i doit renvoyer une valeur d'un type sous-type de τ.

Paramètres et valeur de retour. Dans les constructeurs, méthodes et fonctions, on impose que les paramètres $x_1, \ldots, x_n$ soient de type numérique ou passés par référence et que, le cas échéant, le résultat soit de type numérique ou une référence. (Ceci constitue une différence par rapport à C++, où des objets peuvent être passés par valeur.)

Fonction `main`. On vérifiera enfin l'existence d'une unique fonction de type `int main()`.

1.6 Indications

On suggère fortement de réaliser l'analyse sémantique construction par construction, c'est-à-dire de ne pas chercher à écrire tout le code d'abord pour le débugger ensuite.

Tests. En cas de doute concernant un point de sémantique, vous pouvez utiliser le compilateur C++ 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 phase suivante (production de code), certaines informations provenant du typage seront nécessaires. Il vous est conseillé d'anticiper ces besoins en programmant des fonctions de typage qui ne se contentent pas de parcourir les arbres de syntaxe abstraite issus de l'analyse syntaxique mais en renvoient de nouveaux, contenant plus d'information lorsque c'est nécessaire.

2 Limitations/différences par rapport à C++

Si tout programme Mini C++ est un programme C++ correct, le langage Mini C++ souffre néanmoins d'un certain nombre de limitations par rapport à C++. En voici quelques unes :

- Mini C++ possède moins de mots clés que C++.
- Il n'y a pas d'initialisation pour les variables globales). Pour initialiser une variable globale, il faut utiliser des instructions (dans `main` par exemple).
- On suppose que le type `int` correspond à des entiers 32 bits signés.
- Il n'y a pas d'arithmétique de pointeurs.
- Il n'y a pas de destructeur.

Votre compilateur ne sera jamais testé sur des programmes incorrects au sens de Mini C++ mais corrects au sens de C++.

3 Production de code

L'objectif est de réaliser un compilateur simple mais correct. En particulier, on ne cherchera pas à faire d'allocation de registres mais on se contentera d'utiliser la pile pour stocker les éventuels calculs intermédiaires. Bien entendu, il est possible, et même souhaitable, d'utiliser localement les registres de MIPS. On ne cherchera pas à libérer la mémoire.

3.1 Organisation des données

3.1.1 Représentation des valeurs

Types primitifs. Les entiers seront représentés de manière immédiate par les entiers MIPS (32 bits signés).

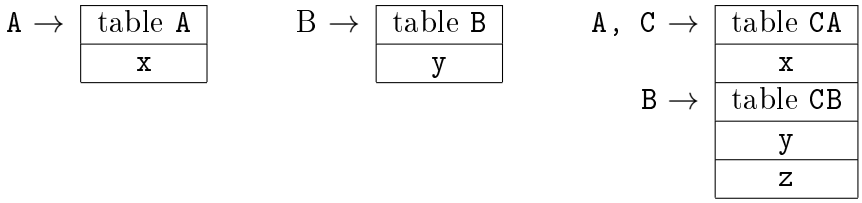
Objets. Un objet est représenté par une adresse, pointant sur un bloc alloué sur le tas. Ce bloc contient

- les valeurs des champs de l'objet ;
- des pointeurs vers des tables de méthodes virtuelles (cf section suivante).

Pour les trois classes

```
class A { public: int x; virtual void f(); };
class B { public: int y; virtual void g(); };
class C : public A, public B { public:
  int z; virtual void f(); virtual void g(); virtual void h(); };
```

on pourra adopter la représentation suivante pour des objets respectivement de classe `A`, `B` et `C` :



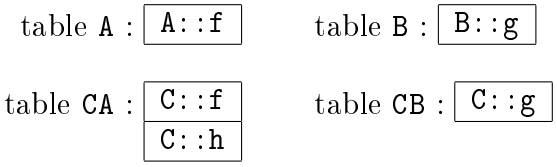
Comme on le voit, un objet de la classe `C` peut être directement considéré comme un objet de classe `A`, le champ `x` étant rangé au même endroit dans les deux objets. En revanche, pour qu'un objet de la classe `C` puisse être considéré comme un objet de classe `B`, il faut commencer par faire une arithmétique de pointeur.

Le compilateur doit maintenir une table donnant, pour chaque classe C et chaque champ de C, la position où trouver ce champ dans un objet de la classe C (sachant que le type dynamique de l'objet peut être différent du type statique que le compilateur connaît).

La valeur `NULL`. Le plus simple est de la représenter par l'entier 0.

3.1.2 Tables de méthodes virtuelles

Une table de méthodes virtuelles permet d'obtenir, à l'exécution, l'adresse du code à exécuter pour une méthode, sachant qu'elle a pu être redéfinie et qu'elle dépend donc du type dynamique de l'objet. Les tables de méthodes virtuelles sont construites statiquement et pourront être allouées dans le segment de données. Sur l'exemple donné dans la section précédente, on peut adopter les tables suivantes :



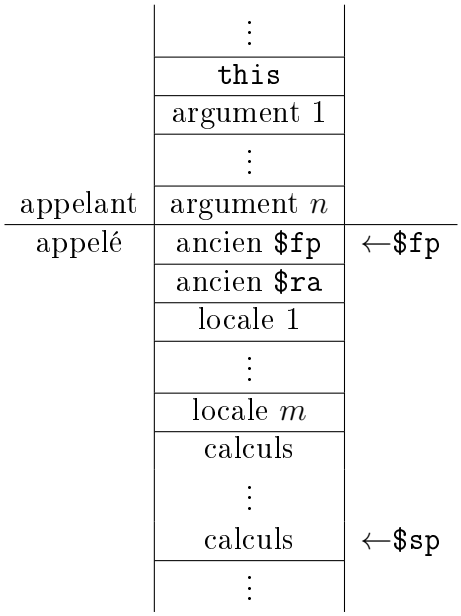
Ici, la table `CA` est utilisée autant pour accéder aux méthodes spécifiques à la classe `C` (comme `h`) que pour considérer un objet de la classe `C` comme un objet de la classe `A`. Ceci est rendu possible par le fait que le code de la méthode `f` est rangé au même endroit

dans la table `A` et dans la table `CA`. La table `CB` est utilisée pour qu'un objet de la classe `C` puisse être considéré comme un objet de la classe `B`. De même que pour les champs, le compilateur maintiendra une table donnant, pour chaque classe C et chaque méthode de C, la position où trouver le code de cette méthode dans la table des méthodes virtuelles.

3.2 Schéma de compilation

3.2.1 Fonctions, méthodes et constructeurs

Les fonctions, méthodes et les constructeurs sont représentés par des fonctions MIPS (*i.e.*, du code appelé par un `jal`). L'appelant place la valeur de `this` (le cas échéant) et les arguments sur la pile. L'appelé place la valeur de retour dans le registre `$v0`. Au retour, l'appelant se charge de dépiler `this` et les arguments. On pourra adopter le schéma suivant :



Il est important de noter que les variables locales n'occupent pas toutes nécessairement la même place sur la pile (certaines sont des objets).

Constructeurs. Le code d'un constructeur appelle implicitement, et en premier lieu, le code des constructeurs sans argument de ses super-classes, le cas échéant. Il est donc pratique d'écrire le code d'un constructeur comme une fonction MIPS qui suppose l'objet déjà alloué (et sa table de méthodes virtuelles déjà positionnée), comme expliqué en cours.

3.2.2 Ordre des opérations de compilation

On pourra effectuer les différentes opérations nécessaires à la production de code dans l'ordre suivant :

1. Construction des descripteurs de classe et de la table décrivant les objets (indiquant où se trouvent les champs au sein des objets).
2. Compilation de toutes les fonctions, constructeurs et méthodes.

Remarque importante. La correction du projet sera réalisée en partie automatiquement, à l'aide d'un jeu de petits programmes réalisant des affichages avec l'"instruction" `std::cout <<`, qui seront compilés avec votre compilateur et dont la sortie sera comparée à la sortie attendue. Il est donc très important de correctement compiler `std::cout <<`.

4 Travail demandé (pour le vendredi 10 janvier)

Étendre votre compilateur avec une nouvelle option de ligne de commande `--type-only`. Cette option indique de stopper la compilation après l'analyse sémantique (typage). Lorsqu'une erreur est détectée par votre compilateur, elle doit être signalée le plus précisément possible, par sa nature et sa localisation dans le fichier source, et le programme doit terminer avec le code de sortie 1 (`exit 1`). On adoptera le format suivant pour cette signalisation :

```
File "test.cpp", line 4, characters 5-6:
'class B' has no member named 'foo'
```

L'anglicisme de la première ligne est nécessaire pour que la fonction `next-error` d'Emacs puisse interpréter la localisation et placer ainsi automatiquement le curseur sur l'emplacement de l'erreur. (En revanche, le message d'erreur proprement dit pourra être écrit en français si vous le souhaitez.) Les localisations peuvent être obtenues pendant l'analyse syntaxique grâce aux mots-clés `$startpos` et `$endpos` de Menhir, puis conservées dans l'arbre de syntaxe abstraite.

Si le fichier d'entrée est conforme à la syntaxe et au typage décrits dans ce document, votre compilateur doit produire du code MIPS et terminer avec le code de sortie 0 (`exit 0` explicite ou terminaison normale du programme), sans rien afficher. Si le fichier d'entrée est `file.cpp`, le code MIPS doit être produit dans le fichier `file.s` (même nom que le fichier source mais suffixe `.s` au lieu de `.cpp`). Ce fichier MIPS doit pouvoir être exécuté avec la commande

```
spim -file file.s
```

ou

```
java -jar Mars_4_2.jar file.s
```

Le résultat affiché sur la sortie standard doit être identique à celui donné par l'exécution du fichier C `file.cpp`, c'est-à-dire obtenu avec les commandes

```
g++ file.cpp -o file
./file
```

(à l'exception des cinq premières lignes ajoutées par SPIM ou du dernier retour-chariot ajouté par MARS).