projet de compilation

Mini Ada

version 3 — 29 décembre 2016

I believe myself to possess a most singular combination
of qualities exactly fitted to make me pre-eminently
a discoverer of the hidden realities of nature.
Ada.

L'objectif de ce projet est de réaliser un compilateur pour un fragment du langage Ada, appelé Mini Ada par la suite, produisant du code x86-64. Il s'agit d'un fragment relativement petit du langage Ada, avec parfois même quelques petites incompatibilités. Néanmoins, votre compilateur ne sera jamais testé sur des programmes incorrects au sens de Mini Ada mais corrects au sens de Ada. Le présent sujet décrit la syntaxe et le typage de Mini Ada, ainsi que la nature du travail demandé.

1 Syntaxe

Dans la suite, nous utilisons les notations suivantes dans les grammaires :

$\langle règle \rangle^\star$	répétition de la règle $\langle règle \rangle$ un nombre quelconque de fois (y compris aucune)
$\langle règle \rangle^\star_t$	répétition de la règle $\langle règle \rangle$ un nombre quelconque de fois (y compris aucune), les occurrences étant séparées par le terminal t
$\langle règle \rangle^+$	répétition de la règle $\langle règle \rangle$ au moins une fois
$\langle règle \rangle^+_t$	répétition de la règle $\langle règle \rangle$ au moins une fois, les occurrences étant séparées par le terminal t
$\langle règle \rangle?$	utilisation optionnelle de la règle $\langle règle \rangle$ (*i.e.* 0 ou 1 fois)
(...)	parenthésage
... | ...	alternative

Attention à ne pas confondre « * » et « $^+$ » avec « `*` » et « `+` » qui sont des symboles du langage Ada. De même, attention à ne pas confondre les parenthèses avec les terminaux `(` et `)`.

1.1 Conventions lexicales

Espaces, tabulations et retour-chariots constituent les blancs. Un commentaire débute par `--` et s'étend jusqu'à la fin de la ligne. Les identificateurs obéissent à l'expression régulière $\langle ident \rangle$ suivante :

$$\begin{array}{lcl} \langle chiffre \rangle & ::= & \texttt{0–9} \\ \langle alpha \rangle & ::= & \texttt{a–z} \mid \texttt{A–Z} \\ \langle ident \rangle & ::= & \langle alpha \rangle\ (\langle alpha \rangle \mid \langle chiffre \rangle \mid \texttt{\_})^\star \end{array}$$

Les identificateurs suivants sont des mots clés :

```
access     and     begin     else    elsif    end
false      for     function  if      in       is
loop       new     not       null    or       out
procedure  record  rem       return  reverse  then
true       type    use       while   with
```

Les identificateurs et les mots clés sont insensibles à la casse. Les constantes obéissent aux expressions régulières ⟨*entier*⟩ et ⟨*caractère*⟩ suivantes :

$$\begin{array}{lcl} \langle \textit{entier} \rangle & ::= & \langle \textit{chiffre} \rangle^+ \\ \langle \textit{caractère} \rangle & ::= & \texttt{'} \ \langle \textit{tout caractère ASCII 7 bits} \rangle \ \texttt{'} \end{array}$$

Les constantes entières doivent être comprises entre 0 et 2^{31}, inclus.

1.2 Syntaxe

La grammaire des fichiers sources considérée est donnée figures 1 et 2. Le point d'entrée est le non-terminal ⟨*fichier*⟩.

$$\begin{array}{lcl}
\langle \textit{fichier} \rangle & ::= & \texttt{with Ada.Text_IO; use Ada.Text_IO;} \\
 & & \texttt{procedure} \ \langle \textit{ident} \rangle \ \texttt{is} \ \langle \textit{decl} \rangle^\star \\
 & & \texttt{begin} \ \langle \textit{instr} \rangle^+ \ \texttt{end} \ \langle \textit{ident} \rangle ? \ \texttt{;} \ \texttt{EOF} \\
\langle \textit{decl} \rangle & ::= & \texttt{type} \ \langle \textit{ident} \rangle \ \texttt{;} \\
 & | & \texttt{type} \ \langle \textit{ident} \rangle \ \texttt{is access} \ \langle \textit{ident} \rangle \ \texttt{;} \\
 & | & \texttt{type} \ \langle \textit{ident} \rangle \ \texttt{is record} \ \langle \textit{champs} \rangle^+ \ \texttt{end record} \ \texttt{;} \\
 & | & \langle \textit{ident} \rangle^+_{\texttt{,}} \ \texttt{:} \ \langle \textit{type} \rangle \ (\texttt{:=} \ \langle \textit{expr} \rangle) ? \ \texttt{;} \\
 & | & \texttt{procedure} \ \langle \textit{ident} \rangle \ \langle \textit{params} \rangle ? \ \texttt{is} \ \langle \textit{decl} \rangle^\star \\
 & & \texttt{begin} \ \langle \textit{instr} \rangle^+ \ \texttt{end} \ \langle \textit{ident} \rangle ? \ \texttt{;} \\
 & | & \texttt{function} \ \langle \textit{ident} \rangle \ \langle \textit{params} \rangle ? \ \texttt{return} \ \langle \textit{type} \rangle \ \texttt{is} \ \langle \textit{decl} \rangle^\star \\
 & & \texttt{begin} \ \langle \textit{instr} \rangle^+ \ \texttt{end} \ \langle \textit{ident} \rangle ? \ \texttt{;} \\
\langle \textit{champs} \rangle & ::= & \langle \textit{ident} \rangle^+_{\texttt{,}} \ \texttt{:} \ \langle \textit{type} \rangle \ \texttt{;} \\
\langle \textit{type} \rangle & ::= & \langle \textit{ident} \rangle \\
 & | & \texttt{access} \ \langle \textit{ident} \rangle \\
\langle \textit{params} \rangle & ::= & \texttt{(} \ \langle \textit{param} \rangle^+_{\texttt{;}} \ \texttt{)} \\
\langle \textit{param} \rangle & ::= & \langle \textit{ident} \rangle^+_{\texttt{,}} \ \texttt{:} \ \langle \textit{mode} \rangle ? \ \langle \textit{type} \rangle \\
\langle \textit{mode} \rangle & ::= & \texttt{in} \mid \texttt{in out}
\end{array}$$

FIGURE 1 – Grammaire des fichiers de Mini Ada.

$$
\begin{array}{lcl}
\langle expr \rangle & ::= & \langle entier \rangle \mid \langle caractère \rangle \mid \texttt{true} \mid \texttt{false} \mid \texttt{null} \\
& \mid & \texttt{(}\ \langle expr \rangle\ \texttt{)} \\
& \mid & \langle accès \rangle \\
& \mid & \langle expr \rangle\ \langle opérateur \rangle\ \langle expr \rangle \\
& \mid & \texttt{not}\ \langle expr \rangle \mid \texttt{-}\ \langle expr \rangle \\
& \mid & \texttt{new}\ \langle ident \rangle \\
& \mid & \langle ident \rangle\ \texttt{(}\ \langle expr \rangle^{+}_{\texttt{,}}\ \texttt{)} \\
& \mid & \texttt{character '}\ \texttt{val (}\ \langle expr \rangle\ \texttt{)} \\
\langle instr \rangle & ::= & \langle accès \rangle\ \texttt{:=}\ \langle expr \rangle\ \texttt{;} \\
& \mid & \langle ident \rangle\ \texttt{;} \\
& \mid & \langle ident \rangle\ \texttt{(}\ \langle expr \rangle^{+}_{\texttt{,}}\ \texttt{)}\ \texttt{;} \\
& \mid & \texttt{return}\ \langle expr \rangle?\ \texttt{;} \\
& \mid & \texttt{begin}\ \langle instr \rangle^{+}\ \texttt{end;} \\
& \mid & \texttt{if}\ \langle expr \rangle\ \texttt{then}\ \langle instr \rangle^{+}\ (\texttt{elsif}\ \langle expr \rangle\ \texttt{then}\ \langle instr \rangle^{+})^{\star} \\
& & (\texttt{else}\ \langle instr \rangle^{+})?\ \texttt{end if;} \\
& \mid & \texttt{for}\ \langle ident \rangle\ \texttt{in reverse}?\ \langle expr \rangle\ \texttt{..}\ \langle expr \rangle \\
& & \texttt{loop}\ \langle instr \rangle^{+}\ \texttt{end loop;} \\
& \mid & \texttt{while}\ \langle expr \rangle\ \texttt{loop}\ \langle instr \rangle^{+}\ \texttt{end loop;} \\
\langle opérateur \rangle & ::= & \texttt{=} \mid \texttt{/=} \mid \texttt{<} \mid \texttt{<=} \mid \texttt{>} \mid \texttt{>=} \\
& \mid & \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/} \mid \texttt{rem} \\
& \mid & \texttt{and} \mid \texttt{and then} \mid \texttt{or} \mid \texttt{or else} \\
\langle accès \rangle & ::= & \langle ident \rangle \mid \langle expr \rangle\ \texttt{.}\ \langle ident \rangle
\end{array}
$$

FIGURE 2 – Grammaire des instructions et expressions de Mini Ada.

Les associativités et précédences des diverses constructions sont données par la table suivante, de la plus faible à la plus forte précédence :

V3

opérateur ou construction	**associativité**
`or`, `or else`	gauche
`and`, `and then`	gauche
`not`	
`=`, `/=`	
`>`, `>=`, `<`, `<=`	
`+`, `-`	gauche
`-` (unaire)	
`*`, `/`, `rem`	gauche
`.`	gauche

Vérification supplémentaire. Dans une déclaration de procédure ou de fonction, si un identificateur suit le mot-clé `end`, alors il doit être identique au nom de la procédure ou de la fonction déclarée.

2 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 :

$$\tau ::= \texttt{integer} \mid \texttt{character} \mid \texttt{boolean} \mid R \mid \texttt{access}\ R \mid \texttt{typenull}$$

où R désigne un type enregistrement. Il s'agit là d'une notation pour la syntaxe *abstraite* des expressions de types. On note $x : \tau \in R$ pour signifier que le type enregistrement R possède un champ x de type τ.

On introduit la relation $\equiv$ sur les types comme la plus petite relation réflexive et symétrique telle que

$$\texttt{typenull} \equiv \texttt{access}\ R$$

Un environnement Γ contient des déclarations de variables, de types enregistrements, de fonctions et de procédures. On notera $d \in \Gamma$ pour spécifier qu'une certaine déclaration d appartient à Γ. Dans un environnement Γ, un type enregistrement R peut être uniquement *déclaré* (avec une déclaration de la forme `type` R`;`) ou bien être *défini* (avec une déclaration de la forme `type` R `is record ... end record;`). On notera $R \in \Gamma$ pour signifier que R est déclaré ou défini dans Γ.

2.1 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 ». Ce jugement est défini par les règles d'inférence suivantes :

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

$$\frac{\Gamma \vdash e : R \quad x : \tau \in R}{\Gamma \vdash e.x : \tau} \qquad \frac{\Gamma \vdash e : \texttt{access}\ R \quad x : \tau \in R}{\Gamma \vdash e.x : \tau}$$

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

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

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

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

$$\frac{\Gamma \vdash e : \texttt{integer}}{\Gamma \vdash \texttt{-}\ e : \texttt{integer}} \qquad \frac{\Gamma \vdash e : \texttt{boolean}}{\Gamma \vdash \texttt{not}\ e : \texttt{boolean}}$$

$$\frac{\texttt{function}\ F(x_1 : \tau'_1, \ldots, x_n : \tau'_n)\ \texttt{return}\ \tau \in \Gamma \quad \forall i, \Gamma \vdash e_i : \tau_i \wedge \tau_i \equiv \tau'_i}{\Gamma \vdash F(e_1, \ldots, e_n) : \tau}$$

$$\frac{\Gamma \vdash e : \texttt{integer}}{\Gamma \vdash \texttt{character'val}(e) : \texttt{character}}$$

$$\frac{R \in \Gamma}{\Gamma \vdash \texttt{new}\ R : \texttt{access}\ R}$$

2.2 Typage des instructions

On introduit le jugement $\Gamma \vdash^r i$ signifiant « dans l'environnement Γ, l'instruction i est bien typée, pour un type de retour r ». Un type de retour est soit $\bot$, pour signifier que l'instruction se trouve dans une procédure, soit un type τ, pour signifier que l'instruction se trouve dans une fonction dont le type de retour est τ. Ce jugement est établi par les règles d'inférence suivantes :

$$\frac{x : \tau \in \Gamma \quad \Gamma \vdash e' : \tau' \quad \tau \equiv \tau'}{\Gamma \vdash^r x \texttt{ := } e}$$

$$\frac{\Gamma \vdash e : R \quad e \text{ valeur gauche} \quad x : \tau \in R \quad \Gamma \vdash e' : \tau' \quad \tau \equiv \tau'}{\Gamma \vdash^r e.x \texttt{ := } e'} \quad \text{V2}$$

$$\frac{\Gamma \vdash e : \texttt{access } R \quad x : \tau \in R \quad \Gamma \vdash e' : \tau' \quad \tau \equiv \tau'}{\Gamma \vdash^r e.x \texttt{ := } e'}$$

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

$$\frac{\Gamma \vdash e : \texttt{boolean} \quad \Gamma \vdash^r i}{\Gamma \vdash^r \texttt{ while } e \texttt{ loop } i \texttt{ end loop}}$$

$$\frac{\Gamma \vdash e_1 : \texttt{integer} \quad \Gamma \vdash e_2 : \texttt{integer} \quad \Gamma + x : \texttt{integer} \vdash^r i}{\Gamma \vdash^r \texttt{ for } x \texttt{ in (reverse) } e_1 \texttt{..} e_2 \texttt{ loop } i \texttt{ end loop}} \quad \text{V2}$$

$$\frac{\texttt{procedure } P(x_1 : \tau'_1, \ldots, x_n : \tau'_n) \in \Gamma \quad \forall i, \Gamma \vdash e_i : \tau_i \wedge \tau_i \equiv \tau'_i}{\Gamma \vdash^r P(e_1, \ldots, e_n)}$$

$$\frac{}{\Gamma \vdash^{\bot} \texttt{return}} \qquad \frac{\Gamma \vdash e : \tau' \quad \tau' \equiv \tau}{\Gamma \vdash^{\tau} \texttt{return } e}$$

$$\frac{\forall j, \Gamma \vdash^r i_j}{\Gamma \vdash^r i_1 \ \ldots \ i_n} \qquad \frac{}{\Gamma \vdash^r \texttt{new\_line}} \qquad \frac{\Gamma \vdash e : \texttt{character}}{\Gamma \vdash^r \texttt{put}(e)}$$

2.3 Typage des déclarations

On note $\Gamma \vdash \tau$ qui signifie « le type τ est bien formé dans l'environnement Γ ». Ce jugement est défini par les règles suivantes :

$$\frac{}{\Gamma \vdash \texttt{integer}} \qquad \frac{}{\Gamma \vdash \texttt{character}} \qquad \frac{}{\Gamma \vdash \texttt{boolean}}$$

$$\frac{R \text{ est défini dans } \Gamma}{\Gamma \vdash R} \qquad \frac{R \in \Gamma}{\Gamma \vdash \texttt{access } R}$$

On introduit le jugement $\Gamma \vdash d$ qui signifie « dans l'environnement Γ, la déclaration d est bien formée ». Si $\vec{d}$ est une séquence de déclarations $d_1, \ldots, d_n$, on note $\Gamma_0 \vdash \vec{d}$ pour indiquer que $\Gamma_0 \vdash d_1$, $\Gamma_0 + d_1 \vdash d_2$, …, $\Gamma_0 + d_1 + \cdots + d_{n-1} \vdash d_n$. On a les règles suivantes :

$$\frac{\Gamma \vdash \tau}{\Gamma \vdash x : \tau} \qquad \frac{\Gamma \vdash \tau \quad \Gamma \vdash e : \tau' \quad \tau' \equiv \tau}{\Gamma \vdash x : \tau \texttt{ := } e}$$

$$\frac{\begin{array}{c}\forall i, \Gamma \vdash \tau'_i \\ \Gamma' := \Gamma + P + x_1 : \tau'_1 + \cdots + x_n : \tau'_n \quad \Gamma' \vdash \vec{d} \quad \Gamma' + \vec{d} \vdash^{\bot} i\end{array}}{\Gamma \vdash \texttt{procedure } P(x_1 : \tau'_1, \ldots, x_n : \tau'_n) \texttt{ is } \vec{d} \texttt{ begin } i \texttt{ end}} \quad \text{V2}$$

V2

$$\frac{\Gamma \vdash \tau \quad \forall i, \Gamma \vdash \tau'_i \qquad \Gamma' := \Gamma + F + x_1 : \tau'_1 + \cdots + x_n : \tau'_n \quad \Gamma' \vdash \vec{d} \quad \Gamma' + \vec{d} \vdash^{\tau} i}{\Gamma \vdash \texttt{function}\ F(x_1 : \tau'_1, \ldots, x_n : \tau'_n)\ \texttt{return}\ \tau\ \texttt{is}\ \vec{d}\ \texttt{begin}\ i\ \texttt{end}}$$

$$\frac{\Gamma \vdash \texttt{type}\ R \qquad R \in \Gamma}{\Gamma \vdash \texttt{type}\ x\ \texttt{is access}\ R}$$

V2

$$\frac{\forall i, \Gamma + \texttt{type}\ R \vdash \tau_i}{\Gamma \vdash \texttt{type}\ R\ \texttt{is record}\ x_1 : \tau_1, \ldots, x_n : \tau_n\ \texttt{end record}}$$

2.4 Vérifications supplémentaires

Enfin, les conditions suivantes doivent être vérifiées :

- Dans un appel de fonction ou de procédure, si un paramètre formel est déclaré `in out`, alors le paramètre effectif correspond doit être une valeur gauche, c'est-à-dire,
  - soit une variable,
  - soit une expression $e.f$ avec e une valeur gauche de type R,
  - soit une expression $e.f$ avec e une expression de type `access` R.

  V2
- Dans une fonction ou une procédure, si un paramètre formel X est déclaré `in` (explicitement ou par défaut), alors sa valeur ne peut être modifiée avec une affectation X `:=` e et, si X est de type R, un champ de X ne peut être modifié avec une affectation $X.f$ `:=` e. De même, sa valeur ou la valeur de son champ ne peut être modifiée en étant passée à une fonction ou procédure attendant un paramètre `in out`.

  V2
- La variable d'une boucle `for` ne peut être affectée.
- L'exécution d'une fonction doit nécessairement terminer sur une instruction `return`.
- Toutes les déclarations d'un même niveau doivent porter des noms différents, la seule exception étant celle d'un type enregistrement déclaré puis défini plus loin.
- Un type enregistrement déclaré doit être défini avant la fin des déclarations de même niveau et avant toute introduction d'un niveau de déclarations supérieur.
- Tous les champs d'un même enregistrement doivent porter des noms différents.
- Les noms `put` et `new_line` sont réservés.

2.5 Anticipation

Dans la phase suivante (production de code), certaines informations provenant du typage peuvent être 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.

3 Production de code

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

En cas de doute sur un point de sémantique du langage Ada, on pourra utiliser un compilateur Ada existant (par exemple `gnatmake` en salle info) comme référence.

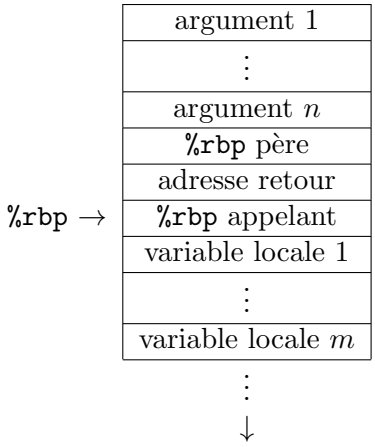
3.1 Représentation des valeurs

Types primitifs. Les entiers sont représentés de manière immédiate par des entiers machine 32 bits signés. Les booléens et les caractères sont représentés par des entiers. En particulier, 0 représente `false` et `1` représente `true`.

La valeur `null`. Elle est représentée par l'entier 0. En particulier, elle est différente de toute adresse d'un objet alloué sur le tas.

3.2 Schéma de compilation

Les arguments sont passés sur la pile. L'appelant place aussi une référence vers le tableau d'activation de la procédure père. La valeur de retour est passée dans le registre `%rax`. Le tableau d'activation a donc la structure suivante :



4 Travail demandé

Le projet est à faire seul ou en binôme. Il doit être remis par email à `filliatr@lri.fr`. 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 compilateur (inutile d'inclure les fichiers compilés). Quand on se place dans ce répertoire, la commande `make` doit créer votre compilateur, qui sera appelé `adac`. La commande `make clean` doit effacer tous les fichiers que `make` a engendrés et ne laisser dans le répertoire que les fichiers sources.

L'archive doit également contenir un rapport expliquant les différents choix techniques qui ont été faits et, le cas échéant, les difficultés rencontrées ou les éléments non réalisés. Ce rapport peut être fourni dans un format ASCII, PostScript ou PDF. Ce rapport n'a pas besoin d'être long, mais il doit être informatif.

Première partie, à rendre avant le dimanche 11 décembre 18h. Votre compilateur `adac` doit accepter sur sa ligne de commande les options `--parse-only` et `--type-only` et exactement un fichier Mini Ada portant l'extension `.adb`. Il doit alors réaliser l'analyse syntaxique du fichier. En cas d'erreur lexicale ou syntaxique, celle-ci doit être signalée le plus précisément possible, par sa nature et sa localisation dans le fichier source. On adoptera le format suivant pour cette signalisation :

```
File "test.adb", line 4, characters 5-6:
syntax error
```

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. En cas d'erreur, le compilateur doit terminer avec le code de sortie 1 (`exit 1`).

Si le fichier est syntaxiquement correct, votre compilateur doit terminer avec le code de sortie 0 si l'option `--parse-only` a été passée sur la ligne de commande. Sinon, il doit poursuivre avec le typage du fichier source. Lorsqu'une erreur de typage est détectée par votre compilateur, elle doit être signalée le plus précisément possible, de la manière suivante :

```
File "test.adb", line 4, characters 5-6:
this expression has type Integer but is expected to have type Boolean
```

Là encore, la nature du message est laissée à votre discrétion, mais la forme de la localisation est imposée. Le compilateur doit alors terminer avec le code de sortie 1 (`exit 1`). Si en revanche il n'y a pas d'erreur de typage, le compilateur doit terminer avec le code de sortie 0. En cas d'erreur du compilateur lui-même, le compilateur doit terminer avec le code de sortie 2 (`exit 2`).

Seconde partie, à rendre avant le dimanche 15 janvier 18h. Étendre votre compilateur pour stopper la compilation après l'analyse sémantique (typage) si l'option de ligne de commande `--type-only` est spécifiée. En l'absence de cette option, et si le fichier d'entrée est conforme à la syntaxe et au typage décrits dans ce sujet, votre compilateur doit produire du code x86-64 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.adb`, le code assembleur doit être produit dans le fichier `file.s` (même nom que le fichier source mais suffixe `.s` au lieu de `.adb`). Ce fichier x86-64 doit pouvoir être exécuté avec la commande

```
gcc file.s -o file
./file
```

Le résultat affiché sur la sortie standard doit être identique à celui donné par l'exécution du fichier Ada `file.adb` avec

```
gnatmake file.adb -o file
./file
```

Remarque importante. La correction du projet est réalisée en partie automatiquement, à l'aide d'un jeu de petits programmes réalisant des affichages avec les procéures `Put` et `New_Line`, qui sont compilés avec votre compilateur et dont la sortie est comparée à la sortie attendue. Il est donc très important de correctement compiler les appels à ces deux procédures.

Au delà de ce sujet. Si vous trouvez ce projet trop facile, vous êtes libres d'ajouter à votre compilateur d'autres fragments du langage Ada (en veillant à rester compatible avec les tests fournis). Voici quelques suggestions, non exhaustives : surcharge, tableaux, types dérivés, vérifications dynamiques, exceptions, paquetages. Si vous réalisez une telle extension, merci de le mentionner explicitement dans votre rapport.

Remerciements. Merci à Nathanaël Courant d'avoir relevé des coquilles dans la première version de ce sujet.