École Normale Supérieure 2022–2023

projet de compilation

Petit C

version 1 — 10 octobre 2022

L'objectif de ce projet est de réaliser un compilateur pour un fragment de C, appelé Petit C par la suite, produisant du code x86-64. Il s'agit d'un fragment contenant des entiers et des pointeurs, 100% compatible avec C au sens où tout programme de Petit C est aussi un programme C correct. Le présent sujet décrit précisément Petit C, ainsi que la nature du travail demandé.

Le projet comporte une **partie optionnelle**, à savoir le support de fonctions imbriquées. Il s'agit là d'une extension GNU C au standard C. Cette partie optionnelle est indiquée en rouge dans le sujet. Dans les tests fournis, les fichiers correspondant à cette partie optionnelle ont un nom commençant par `nested`. **La partie non optionnelle est notée sur 16.**

En cas de doute concernant un point de sémantique, vous pouvez utiliser un compilateur C comme référence (et plus spécifiquement `gcc` en présence de fonctions imbriquées). Vous pouvez d'ailleurs vous inspirer de ses messages d'erreur pour votre compilateur, en les traduisant ou non en français.

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)
$\langle règle \rangle_t^\star$	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)
($\langle règle \rangle$)	parenthésage

Attention à ne pas confondre « * » et « $^+$ » avec « `*` » et « `+` » qui sont des symboles du langage C. 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. Les commentaires peuvent prendre deux formes :

— débutant par `/*`, s'étendant jusqu'à `*/` et ne pouvant être imbriqués ;
— débutant par `//` et s'étendant jusqu'à la fin de la ligne.

Les identificateurs obéissent à l'expression régulière ⟨*ident*⟩ suivante :

⟨*chiffre*⟩ ::= `0–9`
⟨*alpha*⟩ ::= `a–z` | `A–Z`
⟨*ident*⟩ ::= (⟨*alpha*⟩ | `_`) (⟨*alpha*⟩ | ⟨*chiffre*⟩ | `_`)*

Les identificateurs suivants sont des mots clés :

```
bool break continue else false for if
int NULL return sizeof true void while
```

Enfin, les constantes littérales (entiers ou caractères) obéissent aux expressions régulières suivantes :

⟨*entier*⟩ ::= `0`
| `1–9` ⟨*chiffre*⟩*
| `'` ⟨*caractère*⟩ `'`

⟨*chiffre*⟩ ::= `0–9`

⟨*caractère*⟩ ::= tout caractère de code ASCII compris entre 32 et 127, autre que `\` et `'`
| `\\` | `\'` | `\n` | `\t`

⟨*include*⟩ ::= `#include` ⟨*espace*⟩* `<` ⟨*caractère autre que >*⟩* `>` ⟨*retour chariot*⟩

1.2 Syntaxe

La grammaire des fichiers sources considérée est donnée figure 1. Le point d'entrée est le non-terminal ⟨*fichier*⟩. Les associativités et précédences des divers opérateurs sont données par la table située en bas de la figure, de la plus faible à la plus forte précédence. Tous les fichiers de tests fournis commencent par les lignes

```
#include <stdbool.h>
#include <stdlib.h>
#include <stdio.h>
```

qui introduisent respectivement les booléens, `malloc` et `putchar`. Votre compilateur n'a pas besoin d'interpréter ces trois lignes, mais seulement de les accepter à l'analyse syntaxique.

2 Typage statique

Une fois l'analyse syntaxique effectuée avec succès, on vérifie la conformité du fichier source. Dans ce qui suit, on suppose que l'expression e_1`[`e_2`]` a été remplacée par `*(`e_1`+`e_2`)`.

⟨*fichier*⟩ ::= ⟨*include*⟩* ⟨*decl_fct*⟩* `EOF`

⟨*decl_fct*⟩ ::= ⟨*type*⟩ ⟨*ident*⟩ `(` ⟨*param*⟩$^\star_,$ `)` ⟨*bloc*⟩

⟨*type*⟩ ::= (`void` | `int` | `bool`) `*`*

⟨*param*⟩ ::= ⟨*type*⟩ ⟨*ident*⟩

⟨*expr*⟩ ::= ⟨*entier*⟩ | `true` | `false` | `NULL`
| ⟨*ident*⟩
| `*` ⟨*expr*⟩
| ⟨*expr*⟩ `[` ⟨*expr*⟩ `]`
| ⟨*expr*⟩ `=` ⟨*expr*⟩
| ⟨*ident*⟩ `(` ⟨*expr*⟩$^\star_,$ `)`
| `++` ⟨*expr*⟩ | `--` ⟨*expr*⟩ | ⟨*expr*⟩ `++` | ⟨*expr*⟩ `--`
| `&` ⟨*expr*⟩ | `!` ⟨*expr*⟩ | `-` ⟨*expr*⟩ | `+` ⟨*expr*⟩
| ⟨*expr*⟩ ⟨*opérateur*⟩ ⟨*expr*⟩
| `sizeof` `(` ⟨*type*⟩ `)`
| `(` ⟨*expr*⟩ `)`

⟨*opérateur*⟩ ::= `==` | `!=` | `<` | `<=` | `>` | `>=` | `+` | `-` | `*` | `/` | `%` | `&&` | `||`

⟨*instruction*⟩ ::= `;`
| ⟨*expr*⟩ `;`
| `if` `(` ⟨*expr*⟩ `)` ⟨*instruction*⟩
| `if` `(` ⟨*expr*⟩ `)` ⟨*instruction*⟩ `else` ⟨*instruction*⟩
| `while` `(` ⟨*expr*⟩ `)` ⟨*instruction*⟩
| `for` `(` ⟨*decl_var*⟩? `;` ⟨*expr*⟩? `;` ⟨*expr*⟩$^\star_,$ `)` ⟨*instruction*⟩
| ⟨*bloc*⟩
| `return` ⟨*expr*⟩? `;`
| `break` `;`
| `continue` `;`

⟨*bloc*⟩ ::= `{` ⟨*decl_instr*⟩* `}`

⟨*decl_instr*⟩ ::= ⟨*decl_fct*⟩ | ⟨*decl_var*⟩ `;` | ⟨*instruction*⟩

⟨*decl_var*⟩ ::= ⟨*type*⟩ ⟨*ident*⟩ (`=` ⟨*expr*⟩)?

opérateur	associativité	précédence
`=`	à droite	faible
`||`	à gauche	
`&&`	à gauche	
`==` `!=`	à gauche	
`<` `<=` `>` `>=`	à gauche	↓
`+` `-`	à gauche	
`*` `/` `%`	à gauche	
`!` `++` `--` `&` `*` (unaire) `+` (unaire) `-` (unaire)	à droite	
`[]`	—	forte

FIGURE 1 – Grammaire des fichiers C.

Types et environnements. Dans tout ce qui suit, les expressions de types sont de la forme suivante :

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

Il s'agit là d'une notation pour la syntaxe *abstraite* des expressions de types. On introduit la relation $\equiv$ sur les types comme la plus petite relation réflexive et symétrique telle que

$$\frac{\tau_1, \tau_2 \in \{\texttt{int}, \texttt{bool}\}}{\tau_1 \equiv \tau_2} \qquad \frac{}{\texttt{void*} \equiv \tau\texttt{*}}$$

Un environnement de typage 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 $\Gamma + d$ l'environnement Γ étendu avec une nouvelle déclaration d, qui écrase toute déclaration de même nom, le cas échéant.

Typage des expressions. Une expression e est une *valeur gauche* s'il s'agit d'une variable ou d'une expression de la forme $\texttt{*}e_1$. On le note alors *lvalue*(e).

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 :

$$\frac{}{\Gamma \vdash \texttt{NULL} : \texttt{void*}} \qquad \frac{c \text{ constante de type } \tau}{\Gamma \vdash c : \tau} \qquad \frac{\tau\ x \in \Gamma}{\Gamma \vdash x : \tau} \qquad \frac{\tau \not\equiv \texttt{void}}{\Gamma \vdash \texttt{sizeof(}\tau\texttt{)} : \texttt{int}}$$

$$\frac{\Gamma \vdash e : \tau \quad \mathit{lvalue}(e)}{\Gamma \vdash \texttt{\&}e : \tau\texttt{*}} \qquad \frac{\Gamma \vdash e : \tau\texttt{*}}{\Gamma \vdash \texttt{*}e : \tau}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \mathit{lvalue}(e_1) \quad \Gamma \vdash e_2 : \tau_2 \quad \tau_1 \equiv \tau_2}{\Gamma \vdash e_1 \texttt{ = } e_2 : \tau_1}$$

$$\frac{\Gamma \vdash e : \tau \quad \mathit{lvalue}(e) \quad op \in \{\texttt{++}, \texttt{--}\}}{\Gamma \vdash op\ e : \tau} \qquad \frac{\Gamma \vdash e : \tau \quad \mathit{lvalue}(e) \quad op \in \{\texttt{++}, \texttt{--}\}}{\Gamma \vdash e\ op : \tau}$$

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

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

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2 \quad \tau_1 \equiv \tau_2 \quad \tau_1 \equiv \texttt{int} \quad op \in \{\texttt{+}, \texttt{-}, \texttt{*}, \texttt{/}, \texttt{\%}, \texttt{||}, \texttt{\&\&}\}}{\Gamma \vdash e_1\ op\ e_2 : \texttt{int}}$$

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

$$\frac{\Gamma \vdash e_1 : \tau\texttt{*} \quad \Gamma \vdash e_2 : \tau\texttt{*}}{\Gamma \vdash e_1 \texttt{ - } e_2 : \texttt{int}} \qquad \frac{\Gamma \vdash e_2 \texttt{ + } e_1 : \tau}{\Gamma \vdash e_1 \texttt{ + } e_2 : \tau}$$

$$\frac{\tau\ f(\tau'_1, \ldots, \tau'_n) \in \Gamma \quad \Gamma \vdash e_i : \tau_i \quad \tau_i \equiv \tau'_i}{\Gamma \vdash f(e_1, \ldots, e_n) : \tau}$$

Typage des instructions. On introduit le jugement $\Gamma \vdash^{\tau_0} i$ signifiant « dans l'environnement Γ, l'instruction i est bien typée, pour un type de retour τ_0 ». Le type τ_0 représente le type de retour de la fonction dans laquelle se trouve l'instruction i. Ce jugement est établi par les règles d'inférence suivantes :

$$\frac{}{\Gamma \vdash^{\tau_0} \texttt{;}} \qquad \frac{\Gamma \vdash e : \tau}{\Gamma \vdash^{\tau_0} e\texttt{;}} \qquad \frac{}{\Gamma \vdash^{\texttt{void}} \texttt{return;}} \qquad \frac{\Gamma \vdash e : \tau \quad \tau \equiv \tau_0}{\Gamma \vdash^{\tau_0} \texttt{return}\ e\texttt{;}}$$

$$\frac{\Gamma \vdash e : \tau \quad \Gamma \vdash^{\tau_0} i_1 \quad \Gamma \vdash^{\tau_0} i_2}{\Gamma \vdash^{\tau_0} \texttt{if}\ \texttt{(}e\texttt{)}\ i_1\ \texttt{else}\ i_2}$$

$$\frac{\Gamma \vdash e : \tau \quad \Gamma \vdash^{\tau_0} i}{\Gamma \vdash^{\tau_0} \texttt{while(}e\texttt{)}\ i} \qquad \frac{\Gamma \vdash e : \tau \quad \forall i, \Gamma \vdash e_i : \tau_i \quad \Gamma \vdash^{\tau_0} s}{\Gamma \vdash^{\tau_0} \texttt{for(;}\ e\texttt{;}\ e_1, \ldots, e_n\texttt{)}\ s}$$

De plus, on a les équivalences suivantes :

— `if (`e`)` s équivaut à `if (`e`)` s `else;`,
— l'instruction `for(`d`;`e`;`l`)` équivaut à `{`d`;for(;`e`;`l`)}`,
— si l'expression e est omise dans `for(`d`;`e`;`l`)` alors elle équivaut à `true`.

Typage des blocs. Un bloc peut contenir des déclarations de variables et de fonctions, qui étendent localement l'environnement de typage. La portée de ces déclarations s'étend jusqu'à la fin du bloc.

$$\frac{}{\Gamma \vdash^{\tau_0} \texttt{\{\}}} \qquad \frac{\tau \not\equiv \texttt{void} \quad \Gamma + \tau\ x \vdash^{\tau_0} \texttt{\{}b\texttt{\}}}{\Gamma \vdash^{\tau_0} \texttt{\{}\tau\ x\texttt{;}b\texttt{\}}}$$

$$\frac{\Gamma' \stackrel{\text{def}}{=} \Gamma + \tau\ f(\tau_1, \ldots, \tau_n) \quad \Gamma' \vdash^{\tau} \texttt{\{}\tau_1\ x_1\texttt{;} \ldots \tau_n\ x_n\texttt{;}\ b_f\texttt{\}} \quad \Gamma' \vdash^{\tau_0} b}{\Gamma \vdash^{\tau_0} \texttt{\{}\tau\ f(\tau_1\ x_1, \ldots, \tau_n\ x_n)\ b_f\ b\texttt{\}}}$$

On remarque que le prototype d'une fonction est ajouté à l'environnement pour le typage de cette dernière, dans le but d'accepter les fonctions récursives.

Dans un même bloc, toutes les déclarations doivent porter des noms distincts, qu'il s'agisse de variables ou de fonctions. Dans une déclaration de fonction, tous les paramètres doivent avoir des noms distincts.

Typage des fichiers. On rappelle qu'un fichier est une liste de déclarations de fonctions. Pour le typer, on le considère comme un bloc contenant uniquement des déclarations de fonctions. On le type dans un environnement contenant deux fonctions prédéfinies :

```
void *malloc(int n);
int putchar(int c);
```

En particulier, il n'est pas autorisé de définir une fonction (globale) appelée `malloc` ou `putchar`. Enfin, on vérifiera la présence d'une fonction `main` avec le profil suivant :

```
int main() { ... }
```

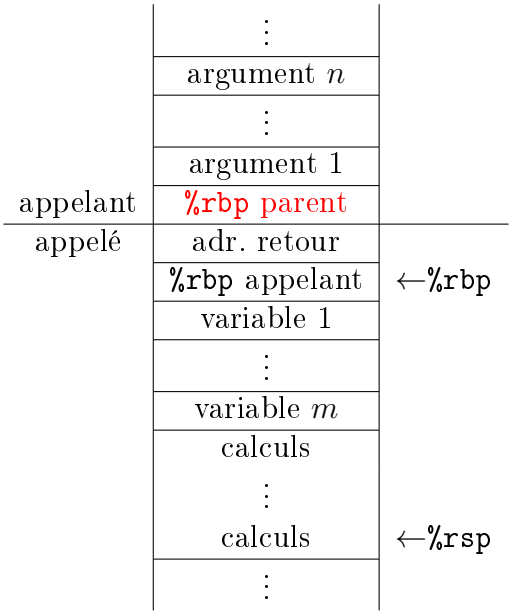
Anticipation. Dans la phase suivante (production de code), certaines informations provenant du typage seront nécessaires. Il vous est conseillé d'anticiper ces besoins en programmant des fonctions de typage qui ne se contentent pas de parcourir les arbres de syntaxe abstraite issus de l'analyse syntaxique mais en renvoient de nouveaux, contenant plus d'information lorsque c'est nécessaire.

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 (voir ci-dessous). Bien entendu, il est possible, et même souhaitable, d'utiliser localement les registres de x86-64.

Représentation des valeurs. On propose d'adopter une représentation simple, où toute valeur occupe 64 bits, qu'il s'agisse d'un entier, d'un booléen ou d'un pointeur. Ce n'est typiquement pas ce que fait un compilateur C (qui va représenter un booléen sur un seul octet et un entier sur quatre octets), mais cela simplifie beaucoup le schéma de compilation. On ne s'éloigne pas tant que cela du standard C, qui laisse notamment de la latitude dans la taille sous-jacente au type `int`.

Schéma de compilation. Les paramètres effectifs d'une fonction sont passés sur la pile. Les variables locales sont allouées sur la pile. Le schéma est donc le suivant pour une fonction ayant n paramètres et nécessitant m emplacements pour ses variables locales :



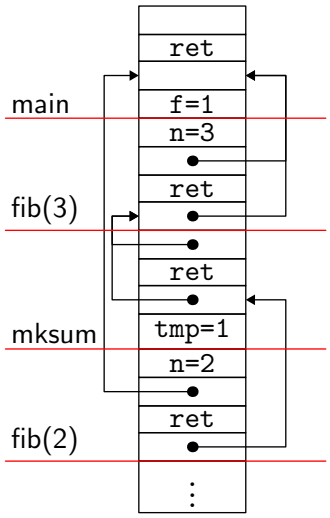
Alignement de pile. Avant d'appeler les fonctions de bibliothèque `putchar` et `malloc`, il convient d'aligner la pile, c'est-à-dire de garantir que `%rsp` est un multiple de 16 avant de faire `call`. Le plus simple pour cela consiste à définir deux petites fonctions assembleur qui appellent respectivement `putchar` et `malloc` après avoir correctement aligné la pile. Si on adopte cette approche, il n'est en revanche pas nécessaire d'aligner la pile pour ses propres fonctions.

Fonctions imbriquées. Pour autoriser les fonctions imbriquées, il faut être capable de retrouver, pour toute variable, le tableau d'activation qui la contient. Si le programme est bien typé, ce tableau d'activation existe quelque part sur la pile. Mais ce peut être arbitrairement haut. Pour qu'on puisse le retrouver, chaque appelant dépose sur la pile

un pointeur vers le tableau d'activation de son parent statique (ou une valeur non significative s'il s'agit d'une fonction globale). Pour retrouver le tableau d'activation d'une fonction de profondeur n, alors qu'on se trouve dans une fonction de profondeur $n+m$, il suffit de suivre m fois ce pointeur.

Considérons par exemple le programme C ci-dessous à gauche. On illustre à droite la pile pendant le calcul de `fib(3)`.

```
int main() {
  int f;
  void fib(int n) {
    void mksum() {
      fib(n-2);
      int tmp = f;
      fib(n-1);
      f = f + tmp;
    }
    if (n <= 1) f = n; else mksum();
  }
  fib(3);
  ...
}
```



On constate en particulier comment le code de `fib(2)` est capable de retrouver le tableau d'activation contenant la variable `f`.

On associera donc à chaque variable et à chaque fonction la profondeur de sa définition, c'est-à-dire le nombre de fonctions imbriquées à l'intérieur desquelles sa définition se trouve. On peut le faire pendant le typage. Les fonctions imbriquées elles-mêmes sont compilées exactement comme les fonctions globales.

4 Limitations/différences par rapport à C

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

- En C, la constante 0 est acceptée comme une valeur de type τ`*`, mais pas en Petit C, où on doit utiliser `NULL`.
- On représente les valeurs de type `bool` et `int` sur 64 bits, ce qui n'est pas ce qui est fait dans un compilateur C. En particulier, le standard C impose que `sizeof(bool)` vaut 1, ce qui n'est pas le cas dans Petit C.
- Petit C a un ensemble de mots clés différent du C. En particulier, on a supposé que `true`, `false`, `bool` ou encore `NULL` sont des mots clés, ce qui n'est pas le cas en C.

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

5 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 programme (inutile d'inclure les fichiers compilés, ou encore des répertoires comme `.git` ou `.vscode`). Quand on se place dans ce répertoire, la commande `make` doit créer votre compilateur, qui sera appelé `petitc`. La commande `make clean` doit effacer tous les fichiers que `make` a engendrés et ne laisser dans le répertoire que les fichiers sources. Bien entendu, la compilation du projet peut être réalisée avec d'autres outils que `make` (par exemple `dune` si le projet est réalisé en OCaml) et le `Makefile` se réduit alors à quelques lignes seulement pour appeler ces outils.

L'archive doit également contenir un court rapport expliquant les différents choix techniques qui ont été faits et, le cas échéant, les difficultés rencontrées ou les éléments non réalisés. Ce rapport pourra être fourni dans un format texte (par exemple Markdown) ou PDF.

5.1 Pour le dimanche 4 décembre 18h

Dans cette première partie du projet, le compilateur `petitc` doit accepter sur sa ligne de commande une option éventuelle (parmi `--parse-only` et `--type-only`) et exactement un fichier Petit C portant l'extension `.c`. 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.c", line 4, characters 5-6:
syntax error
```

L'anglicisme de la première ligne est nécessaire pour que la fonction `next-error` d'Emacs [1] 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, le 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 le compilateur, elle doit être signalée le plus précisément possible, de la manière suivante :

```
File "test.c", line 8, characters 3-10:
error: too many arguments to function ‘putchar’
```

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

1. Le correcteur est susceptible d'utiliser cet éditeur.

de sortie 0. En cas d'erreur du compilateur lui-même, le compilateur doit terminer avec le code de sortie 2 (`exit 2`). L'option `--type-only` indique de stopper la compilation après l'analyse sémantique (typage). Elle est donc sans effet dans cette première partie, mais elle doit être honorée néanmoins.

5.2 Pour le dimanche 15 janvier 18h

Si le fichier d'entrée est conforme à la syntaxe et au typage de ce sujet, votre compilateur doit produire du code x86-64 et terminer avec le code de sortie 0, sans rien afficher. Si le fichier d'entrée est `file.c`, le code assembleur doit être produit dans le fichier `file.s` (même nom que le fichier source mais suffixe `.s` au lieu de `.c`). Ce fichier x86-64 doit pouvoir être exécuté avec les commandes

```
gcc -no-pie file.s -o file
./file
```

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

```
gcc file.c
./a.out
```

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

La correction du projet sera également réalisée par relecture de votre code. Il est donc important d'écrire du code lisible. Des commentaires sont les bienvenus s'ils aident à comprendre les parties les plus subtiles du code. Pour le reste, un bon code se suffit à lui-même.

Conseils. Il est fortement conseillé de procéder construction par construction que ce soit pour le typage ou pour la production de code, dans cet ordre : affichage, arithmétique, variables, instructions (`if`, `while`), fonctions, fonctions imbriquées.