Mini-compilateur C [1]

première partie : analyse syntaxique
deuxième partie : typage
troisième partie : production de code

Le langage considéré dans ce TER est un sous-ensemble de C. Vos fichiers pourront être compilés aussi bien par votre compilateur que par tout autre compilateur C.

Notation Dans la suite, les symboles « $\star$ », « $+$ » et « ? » indiquent la répétition du symbole les précédant (zéro fois ou plus pour « $\star$ », zéro ou une fois pour « ? » et une fois ou plus pour « $+$ »). Attention de ne pas confondre « $\star$ » et « $+$ » avec « `*` » et « `+` » qui sont des symboles du langage C.

1 Analyse syntaxique

1.1 Analyse lexicale

Espaces, tabulations et retour-chariots constituent les blancs. Les commentaires peuvent prendre deux formes :

— débutant par `/*` et s'étendant jusqu'à `*/`, et ne pouvant être imbriqués ;
— débutant par `//` et s'étendant jusqu'à la fin de la ligne (ou du fichier s'il s'agit de la dernière ligne).

De plus on ignorera les lignes suivantes :

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

quelle que soit leur position dans le fichier.

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

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

Les identificateurs suivants sont des mots clés :

```
char   double else   extern   for      if   int   long
return short  sizeof struct   unsigned void while
```

Les constantes entières obéissent à l'expression régulière ⟨*entier*⟩ suivante :

⟨*entier*⟩ ::= ⟨*chiffre*⟩+(`u`|`U`) ?(`l`|`L`) ?

1. Ce projet reprend certains aspects du projet de compilation donné il y a trois ans, ce dernier lui-même inspiré d'un projet similaire donné par Jean-Christophe Filliâtre dans le cadre de son cours de compilation à l'École Normale Supérieure `http://www.lri.fr/~filliatr`.

Les constantes flottantes obéissent à l'expression régulière ⟨*double*⟩ suivante :

$$\langle double \rangle \quad ::= \quad (\langle chiffre \rangle + . \langle chiffre \rangle \star \mid \ . \langle chiffre \rangle +)((\texttt{e}|\texttt{E})\texttt{-}\,?\langle chiffre \rangle +)\,?$$

les littéraux caractères sont délimités par des *quotes* simples (`'`). Les caractères dont le code ASCII est celui du blanc (code ASCII décimal 32) ou supérieur sont représentés tels quels, à l'exception des caractères « `'` », « `"` » et « `\` ». Ces derniers peuvent être représentés en utilisant les séquences d'échappement « `\'` », « `\"` » et « `\\` » respectivement. Il est aussi possible d'utiliser la séquence d'échappement « `\x`xx » où x est un chiffre hexadécimal (« `0-9` », « `a-f` » ou « `A-F` »). Dans ce cas, la séquence représente le caractère de code ASCII hexadécimal xx. Enfin, la séquence « `\n` » représente un retour chariot.

Enfin les chaînes de caractères littérales sont des séquences de caractères (tels que définis ci-dessus) délimitées par des *quotes* doubles : « `"` ». Les chaînes ne peuvent pas s'étendre sur plusieurs lignes. Dans la suite, les caractères et les chaînes de caractères sont représentés par les non-terminaux ⟨*car*⟩ et ⟨*chaîne*⟩ respectivement.

1.2 Analyse syntaxique

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 suivante, de la plus faible à la plus forte précédence :

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

1.3 Localisation des erreurs

Lorsqu'une erreur lexicale ou syntaxique est détectée par votre compilateur, elle doit être signalée le plus précisémment 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
```

La première ligne du message d'erreur (« `File "test.c"` ...») doit respecter *strictement* ce format. La seconde ligne est libre, vous pouvez vous en servir pour donner un message d'explication (éventuellement en français). Les localisations peuvent être obtenues grâce aux fonctions `Lexing.lexeme_start_p`, `Lexing.lexeme_end_p` pour l'analyseur lexical et `Parsing.rhs_start_pos`, `Parsing.rhs_end_pos` pour l'analyseur syntaxique.

⟨*fichier*⟩ ::= ⟨*decl*⟩⋆ `EOF`
⟨*decl*⟩ ::= ⟨*decl_vars*⟩ | ⟨*decl_typ*⟩ | ⟨*decl_fct*⟩ | ⟨*decl_ext*⟩
⟨*decl_vars*⟩ ::= ⟨*type*⟩ ⟨*var*⟩ `;`
⟨*decl_typ*⟩ ::= `struct` ⟨*ident*⟩ `{` ⟨*decl_vars*⟩⋆ `};`
⟨*decl_fct*⟩ ::= ⟨*type*⟩ `*`⋆ ⟨*ident*⟩ `(` ⟨*arguments*⟩? `)` ⟨*bloc*⟩
⟨*decl_ext*⟩ ::= `extern` ⟨*type*⟩ `*`⋆ ⟨*ident*⟩ `(` ⟨*arguments*⟩? `);`
⟨*type*⟩ ::= `void` | ⟨*int_type*⟩ | `double` | `struct` ⟨*ident*⟩
⟨*int_type*⟩ ::= `unsigned`? (`char` | `short` | `int` | `long`)
⟨*arguments*⟩ ::= ⟨*type*⟩ ⟨*var*⟩ | ⟨*type*⟩ ⟨*var*⟩ `,` ⟨*arguments*⟩
⟨*var*⟩ ::= ⟨*ident*⟩ | `*` ⟨*var*⟩
⟨*expr*⟩ ::= ⟨*entier*⟩ | ⟨*car*⟩ | ⟨*chaîne*⟩ | ⟨*double*⟩
| ⟨*ident*⟩
| `*` ⟨*expr*⟩
| ⟨*expr*⟩ `[` ⟨*expr*⟩ `]`
| ⟨*expr*⟩ `.` ⟨*ident*⟩
| ⟨*expr*⟩ `->` ⟨*ident*⟩
| ⟨*expr*⟩ `=` ⟨*expr*⟩
| ⟨*ident*⟩ `(` ⟨*l_expr*⟩? `)`
| `++` ⟨*expr*⟩ | `--` ⟨*expr*⟩ | ⟨*expr*⟩ `++` | ⟨*expr*⟩ `--`
| `&` ⟨*expr*⟩ | `!` ⟨*expr*⟩ | `-` ⟨*expr*⟩ | `+` ⟨*expr*⟩
| ⟨*expr*⟩ ⟨*opérateur*⟩ ⟨*expr*⟩
| `sizeof (` ⟨*cplx_type*⟩ `)`
| `(` ⟨*cplx_type*⟩ `)` ⟨*expr*⟩
| `(` ⟨*expr*⟩ `)`
⟨*opérateur*⟩ ::= `==` | `!=` | `<` | `<=` | `>` | `>=` | `+` | `-` | `*` | `/` | `%` | `&&` | `||`
⟨*cplx_type*⟩ ::= ⟨*type*⟩ `*`⋆
⟨*l_expr*⟩ ::= ⟨*expr*⟩ | ⟨*expr*⟩ `,` ⟨*l_expr*⟩

⟨*instruction*⟩ ::= `;`
| ⟨*expr*⟩ `;`
| `if (` ⟨*expr*⟩ `)` ⟨*instruction*⟩
| `if (` ⟨*expr*⟩ `)` ⟨*instruction*⟩ `else` ⟨*instruction*⟩
| `while (` ⟨*expr*⟩ `)` ⟨*instruction*⟩
| `for (` ⟨*l_expr*⟩? `;` ⟨*expr*⟩? `;` ⟨*l_expr*⟩? `)` ⟨*instruction*⟩
| ⟨*bloc*⟩
| `return` ⟨*expr*⟩? `;`
⟨*bloc*⟩ ::= `{` ⟨*decl_vars*⟩⋆ ⟨*instruction*⟩⋆ `}`

FIGURE 1 – Grammaire des fichiers C

2 Typage

2.1 Types et environnements de typage

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

$$\tau ::= \texttt{void} \mid \iota \mid \texttt{double} \mid \texttt{struct}\ id \mid \tau\texttt{*} \mid \texttt{typenull}$$

$$\iota ::= \texttt{unsgigned}\ \nu \mid \nu$$
$$\nu ::= \texttt{char} \mid \texttt{short} \mid \texttt{int} \mid \texttt{long}$$

où *id* désigne un identificateur de structure. Le type `typenull` est un type particulier introduit pour pouvoir typer la constante `0` qui en C représente soit l'entier 0 soit le pointeur nul (la macro `NULL`, définie dans le fichier `stdlib.h` n'est en général qu'un alias pour `((void *)0)`). Il s'agit là d'une notation pour la syntaxe *abstraite* des expressions de types. On introduit la relation $\approx$ sur les types comme la plus petite relation réflexive et symétrique telle que

$$\frac{\tau_1, \tau_2 \in \{\iota, \texttt{typenull}, \texttt{double}\}}{\tau_1 \approx \tau_2} \qquad \frac{}{\texttt{typenull} \approx \tau\texttt{*}} \qquad \frac{}{\texttt{void*} \approx \tau\texttt{*}}$$

Attention, la relation *n'est pas transitive*. En effet, on a : `int` $\approx$ `typenull` (par la première règle) et `typenull` $\approx$ `char*` (par la deuxième règle) mais on n'a surtout pas `int` $\approx$ `char*`. Un environnement de typage Γ est une suite de déclarations de variables de la forme $\tau\ x$, de déclarations de structures de la forme `struct` S `{`$\tau_1\ x_1 \cdots \tau_n\ x_n$`}`, et de déclarations de profils de fonctions de la forme $\tau\ f(\tau_1, \ldots, \tau_n)$. On notera `struct` S `{`$\tau\ x$`}` pour indiquer que la structure S contient un champ x de type τ.

Dans la suite, on dira qu'un type est *numérique*, et on notera $num(\tau)$ si τ n'est compatible ni avec `void` ni avec un type de structure (autrement dit, τ doit être compatible avec `typenull`, un type d'entier ι `double` ou un type pointeur τ'`*`) et on dira qu'un type est *arithmétique*, et on notera $arith(\tau)$ si τ est numérique et n'est pas un type pointeur.

On définit ensuite une fonction `rank` sur les types arithmétiques. Cette dernière sert à exprimer de manière compacte les conversions entre les différents types d'entiers (et elle représente, intuitivement le nombre de bits utilisés par le type, hors bit de signe).

$$\begin{array}{lcl} rank(\texttt{char}) & = & 7 \\ rank(\texttt{short}) & = & 15 \\ rank(\texttt{int}) & = & 31 \\ rank(\texttt{long}) & = & 63 \end{array} \qquad \begin{array}{lcl} rank(\texttt{unsigned}\ \nu) & = & 1 + \texttt{rank}(\nu) \\ rank(\texttt{double}) & = & 100 \\ rank(\texttt{typenull}) & = & 0 \end{array}$$

On définit à partir de *rank*, la relation d'ordre partielle sur les types :

$$\tau_1 < \tau_2 \Leftrightarrow rank(\tau_1) < rank(\tau_2)$$

et on note $\tau_1 \leq \tau_2$ le fait que $\tau_1 < \tau_2$ ou $\tau_1 = \tau_2$ et on définit :

$$\begin{array}{lcll} max(\tau_1, \tau_2) & = & \tau_2 & \text{si } \tau_1 < \tau_2 \\ max(\tau_1, \tau_2) & = & \tau_1 & \text{si } \tau_2 < \tau_1 \end{array}$$

2.2 Typage

Bonne formation des types On dit qu'un type τ est *bien formé* dans un environnement Γ, et on note $\Gamma \vdash \tau$ `bf`, si tous les identificateurs de structures apparaissant dans τ correspondent à des structures déclarées dans Γ.

Typage des expressions En C, il existe une classe d'expressions particulières nommées « *valeurs gauches* ». Intuitivement, les valeurs gauches sont les expressions que l'on a le droit de placer *à gauche* d'une affectation (d'où leur nom). À l'inverse d'autres langages, le fait d'être une valeur gauche en C dépend du type de l'expression. C'est pourquoi

on introduit simultanément 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 définis par les règles d'inférence suivantes :

(constantes)

$$\frac{}{\Gamma \vdash \texttt{0} : \texttt{typenull}} \qquad \frac{c \text{ constante de type } \tau}{\Gamma \vdash c : \tau} \qquad \frac{\Gamma \vdash \tau \ \mathsf{bf} \quad \tau \not\equiv \texttt{void}}{\Gamma \vdash \texttt{sizeof(}\tau\texttt{)} : \texttt{unsigned long}}$$

(affectations)

$$\frac{\Gamma \vdash_l e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2 \quad \tau_1 \approx \tau_2}{\Gamma \vdash e_1 \texttt{ = } e_2 : \tau_1}$$

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

(comparaisons)

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

(arithmétique et logique)

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

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2 \quad max(\tau_1, \tau_2) \leq \texttt{unsigned long}}{\Gamma \vdash e_1 \texttt{ \% } e_2 : max(\tau_1, \tau_2)}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2 \quad \tau_1 \approx \tau_2 \quad \tau_1 \approx \texttt{double} \quad op \in \{\texttt{+}, \texttt{-}, \texttt{*}, \texttt{/}\}}{\Gamma \vdash e_1\ op\ e_2 : max(\tau_1, \tau_2)}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2 \quad \tau_1 \approx \tau_2 \quad \tau_1 \approx \texttt{double} \quad op \in \{\texttt{||}, \texttt{\&\&}\}}{\Gamma \vdash e_1\ op\ e_2 : \texttt{int}}$$

(arithmétique de pointeurs)

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2 \quad \tau_1 \equiv \tau_1'\texttt{*} \quad max(\tau_2, \texttt{unsigned long}) \leq \texttt{unsigned long} \quad op \in \{\texttt{+}, \texttt{ -}\}}{\Gamma \vdash e_1\ op\ e_2 : \tau_1'\texttt{*}}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2 \quad \tau_1 \equiv \tau_2'\texttt{*} \quad \tau_2 \equiv \tau_2'\texttt{*}}{\Gamma \vdash e_1 \texttt{ - } e_2 : \texttt{long}} \qquad \frac{\Gamma \vdash e_2 \texttt{ + } e_1 : \tau}{\Gamma \vdash e_1 \texttt{ + } e_2 : \tau} \qquad \frac{\Gamma \vdash e : \tau' \ num(\tau)\ num(\tau')\ \Gamma \vdash \tau \ \mathsf{bf}}{\Gamma \vdash \texttt{(}\tau\texttt{)}e : \tau}$$

(appel de fonction)

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

(accès)

$$\frac{\Gamma \vdash e : \texttt{struct}\ S \quad \texttt{struct}\ S\ \{\tau\ x\} \in \Gamma}{\Gamma \vdash e.x : \tau}$$

(adresse)

$$\frac{\Gamma \vdash_l e : \tau}{\Gamma \vdash \texttt{\&}e : \tau\texttt{*}}$$

(valeurs gauches)

$$\frac{\Gamma \vdash_l e : \tau}{\Gamma \vdash e : \tau} \qquad \frac{\tau\ x \in \Gamma}{\Gamma \vdash_l x : \tau} \qquad \frac{\Gamma \vdash e : \tau\texttt{*}}{\Gamma \vdash_l \texttt{*}e : \tau}$$

$$\frac{\Gamma \vdash_l e : \texttt{struct}\ S \quad \texttt{struct}\ S\ \{\tau\ x\} \in \Gamma}{\Gamma \vdash_l e.x : \tau}$$

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

$$\frac{}{\Gamma, \tau_0 \vdash_i \texttt{;}} \qquad \frac{\Gamma, \tau_0 \vdash e : \tau}{\Gamma, \tau_0 \vdash_i e\texttt{;}} \qquad \frac{}{\Gamma, \texttt{void} \vdash_i \texttt{return;}} \qquad \frac{\Gamma \vdash e : \tau_0}{\Gamma, \tau_0 \vdash_i \texttt{return}\ e\texttt{;}}$$

$$\frac{\Gamma \vdash e : \tau \quad \mathit{num}(\tau) \quad \Gamma, \tau_0 \vdash_i i_1 \quad \Gamma, \tau_0 \vdash_i i_2}{\Gamma, \tau_0 \vdash_i \texttt{if (}e\texttt{)}\ i_1\ \texttt{else}\ i_2}$$

$$\frac{\Gamma, \tau_0 \vdash_i i_1 \quad \Gamma \vdash e : \tau \quad \mathit{num}(\tau) \quad \Gamma, \tau_0 \vdash_i i_2 \quad \Gamma, \tau_0 \vdash_i i_3}{\Gamma, \tau_0 \vdash_i \texttt{for(}i_1\texttt{;}\ e\texttt{;}\ i_2\texttt{)}\ i_3}$$

$$\frac{\forall j \leq k, \Gamma \vdash \tau_j\ \mathsf{bf} \quad \tau_j \not\equiv \texttt{void} \quad \forall j \leq n, \{\tau_1\ x_1, \ldots, \tau_k\ x_k\} \cup \Gamma \vdash_i i_j}{\Gamma, \tau_0 \vdash_i \texttt{\{}\tau_1\ x_1 \cdots \tau_k\ x_k\texttt{;}i_1 \cdots i_n\texttt{\}}}$$

Cette dernière règle signifie que pour typer un *bloc* constitué de k déclarations de variables (locales au bloc) et de n instructions, on vérifie d'abord la bonne formation des déclarations puis on type chacune des instructions dans l'environnement augmenté des nouvelles déclarations. On rappelle aussi que le *parsing* a supprimé des instructions superflues (`while`, `if` sans `else`, ...).

Typage des fichiers On rappelle qu'un fichier est une liste de déclarations. On introduit le jugement « $\Gamma \vdash d \to \Gamma'$ » qui signifie « dans l'environnement Γ, la déclaration d est bien formée et produit un environnement Γ' ». Ce jugement est dérivable grâce aux règles suivantes :

Déclarations de variables (globales)

$$\frac{\Gamma \vdash \tau\ \mathsf{bf} \quad \tau \not\equiv \texttt{void}}{\Gamma \vdash \tau\ x \to \{\tau\ x\} \cup \Gamma}$$

Déclarations de structures

$$\frac{\Gamma, \texttt{struct}\ id\ \{\tau_1\ x_1 \cdots \tau_n\ x_n\} \vdash \tau_i\ \mathsf{bf}}{\Gamma \vdash \texttt{struct}\ id\ \{\tau_1\ x_1; \cdots \tau_n\ x_n;\} \to \{\texttt{struct}\ id\ \{\tau_1\ x_1 \cdots \tau_n\ x_n\}\} \cup \Gamma}$$

On vérifiera d'autre part que les types de champs τ_i ne font référence à la structure id elle-même que sous un pointeur.

Déclarations de fonctions et de fonction externes

$$\frac{\Gamma \vdash \tau_i\ \mathsf{bf} \quad \forall i > 0, \tau_i \not\equiv \texttt{void} \quad \{\tau_0\ f(\tau_1, \ldots, \tau_n), \tau_1\ x_1, \ldots, \tau_n\ x_n\} \cup \Gamma, \tau_0 \vdash_i b}{\Gamma \vdash \tau_0\ f(\tau_1\ x_1, \ldots, \tau_n\ x_n)\ b \to \{\tau_0\ f(\tau_1, \ldots, \tau_n)\} \cup \Gamma}$$

$$\frac{\Gamma \vdash \tau_i\ \mathsf{bf} \quad \forall i > 0, \tau_i \not\equiv \texttt{void}}{\Gamma \vdash \tau_0\ f(\tau_1\ x_1, \ldots, \tau_n\ x_n)\ b \to \{\tau_0\ f(\tau_1, \ldots, \tau_n)\} \cup \Gamma}$$

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

Fichiers On introduit finalement le jugement $\Gamma \vdash_f d_1 \cdots d_n$ signifiant « dans l'environnement Γ le fichier constitué par la suite de déclarations d_1, …, d_n est bien formé ». Le typage d'un fichier consiste à typer successivement les déclarations dans le contexte étendu par chaque nouvelle déclaration, d'où les règles :

$$\frac{}{\Gamma \vdash_f \emptyset} \qquad \frac{\Gamma \vdash d_1 \to \Gamma' \quad \Gamma' \vdash_f d_2 \cdots d_n}{\Gamma \vdash_f d_1\ d_2 \cdots d_n}$$

Règles d'unicité Enfin, on vérifiera l'unicité :
- des identificateurs de structures sur l'ensemble du fichier ;
- des champs de structure à l'intérieur d'une *même* structure ;
- des symboles (variables *globales* et fonctions) sur l'ensemble du fichier.

Fonction principale On vérifiera la présence d'une fonction **main** avec l'un des deux prototypes suivants :

```
int main();
int main(int argc, char** argv);
```

dans le fichier.

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 x86-64.

Les difficultés liées à la production de code dans ce projet sont les suivantes :

— les données n'ont pas toutes la même taille ; en particulier, le compilateur doit savoir calculer la taille de la représentation mémoire de chaque type.
— certaines valeurs doivent être correctement *alignées*, c'est-à-dire stockées à des adresses multiples de 8.
— les structures sont affectées, passées en argument et renvoyées comme résultat *par valeur*, ce qui nécessite de copier tout un bloc d'octets.
— Il y a beaucoup de types différents et de combinaisons à gérer ! (addition d'un `char` avec un double, d'un `unsigned long` avec un `int`, ...)

3.1 Représentation des valeurs

On commence par définir la notion d'*alignement*. On dit qu'une donnée est alignée sur n octets si l'adresse à laquelle elle se trouve est un multiple de n octets. Même si l'architecture x86-64 supporte les accès mémoire non alignés (à l'inverse d'ARM et MIPS), la norme C impose certains alignements. Afin de simplifier la disposition des objets en mémoire on appliquera la règle qu'un type simple (entier, double ou pointeur) doit toujours être aligné sur sa taille. (*i.e* les `long` et pointeurs se trouver à une adresse multiple de 8, les `int` de 4, les `short` de 2).

Une valeur de type `char` sera stockée sur 8 bits non signés et une valeur de type `int` ou de type pointeur sur 64 bits signés. Une structure est représentée comme la suite ordonnée de ses champs. Les champs sont éventuellement séparées par des octets inutilisés, dits de remplissage (*padding* en anglais), lorsque l'alignement des champs l'exige. Ainsi la structure

```
struct S { int a; char b; char c; char *p; };
```

doit être alignée sur 8 octets et sera représentée sur 16 octets, dont 2 octets de remplissage marqués X, de la manière suivante :

a	a	a	a	b	c	X	X	p	p	p	p	p	p	p	p

Une structure doit éventuellement comporter des octets terminaux de remplissage pour que la structure suivante dans un tableau soit alignée. Ainsi la structure

```
struct T { int x; char y; };
```

occupe 8 octets (dont 3 de remplissage après `y`). Ceci permet notamment d'allouer un « tableau » de `n` telles structures avec l'idiome

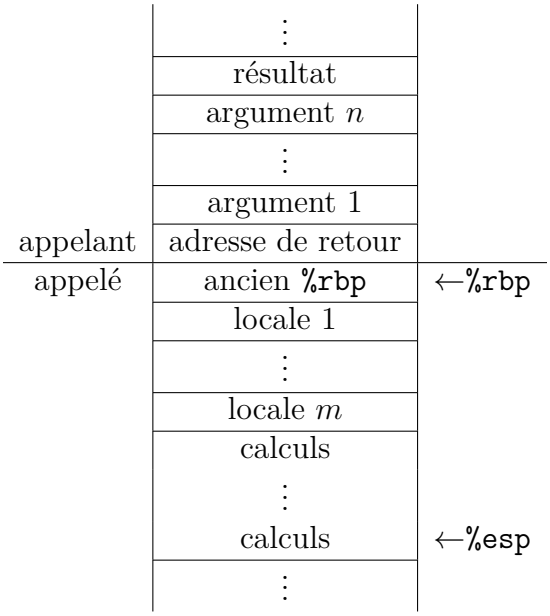
```
struct T *a = malloc(n * sizeof(struct T));
```

tout en garantissant l'alignement de chacune d'elles.

La construction `sizeof`(τ) doit renvoyer le nombre d'octets occupés par la représentation du type τ. En particulier on a `sizeof(struct S)` $= 16$ et `sizeof(struct T)` $= 8$ pour les deux structures ci-dessus.

3.2 Schéma de compilation

Les variables globales seront allouées sur le segment de données. Les variables locales seront allouées sur la pile (dans le tableau d'activation). Les arguments et résultat d'une fonction seront passés sur la pile (les données étant de taille variable, c'est le plus simple). On pourra adopter le schéma suivant :



Pour la copie de structure (passage d'argument, valeur de retour, affectation), on pourra écrire une fonction de type `memmove`, par exemple directement en assembleur.

3.3 Retour sur le typage

Afin de faciliter (grandement) la production de code, il est utile de revenir sur la passe de typage pour stocker les informations suivantes :

— pour chaque fonction gardée dans l'environnement global `fun_env`, il faut garder en plus un booléen indiquant si c'est une fonction `extern` ou non. Les déclaration `extern` nous permettent de déclarer une fonction de la bibliothèque standard C que `gcc` saura résoudre. Par exemple :

```
//affiche un caractère sur le terminal
extern  int putchar(int c);
//affiche une chaine suivie de \n dans le terminal
extern  int puts(char * s);
//renvoie un pointeur vers une zone mémoire de taille l
void * malloc(unsigned long l);
//libere une zone mémoire renvoyée par malloc
void free(void * p);

int main() {
     //le code peut utiliser toutes les fonctions
     //ci-dessus !
}
```

— pour tous les endroits où la relation de compatibilité a été utilisée, insérer un `cast` approprié. Le cas le plus complexe est celui des opérations binaires, traité ci dessous. Concrètement, étant donné un appel de fonction de la forme : $f(e_1, \ldots, e_n)$ où les arguments attendus sont de type τ_1, ..., τ_n, on le transformera en un appel de fonction $f((\tau_1)e_1, \ldots, (\tau_n)e_n)$, ce qui permet par exemple de donner des `int` a une fonction qui attend des `long` en argument. La production de code s'en trouvera alors simplifié car le code de conversion sera écrit une fois pour toute pour le *cast* et vous pourrez supposer que les fonctions ont toujours des arguments du bon type en paramètre. De même une affectation $e_1 = e_2$ où e_1 est une valeur gauche de type τ_1, sera remplacée par une affectation $e_1 = (\tau_1)e_2$

Opérateurs binaires. Les opérateurs binaires peuvent impliquer plusieurs combinaisons de types différents. La norme C est particulièrement sordide de ce point de vue là. Une fois simplifiée (pour l'architecture qui nous interesse) les conversions peuvent être exprimées comme suit. Soit une expression *bien typée* $e_1 o e_2$ où o est un opérateur binaire (addition, soustraction, comparaison, ...). On appelle τ_1 et τ_2 les types de e_1 et e_2.

— si *arith*(τ_1) et *arith*(τ_2) et $\tau_1 \neq \tau_2$ (i.e. se sont tous les deux des types doubles ou entiers, pas de pointeur)
  — si τ_1 (resp. τ_2) est `double`, e_2 est remplacé par (`double`)e_2 (resp. (`double`)e_1)
  — sinon si $\tau_1 \leq$ `int`, alors e_1 est remplacé par $e'_1 = (int)e_1$ (idem pour e_2).
  — On appelle τ'_1 et τ'_2 les nouveaux types de e'_1 et e'_2 (qui sont au moins `(int)`).
    — si τ'_1 vaut `unsigned long` alors e'_2 est remplacé par `(unsigned long)`e'_2 (idem pour e'_1)
    — sinon, si τ'_1 vaut `long` alors e'_2 est remplacé par `(long)`e'_2 (idem pour e'_1)
    — sinon, si τ'_1 vaut `unsigned int` alors e'_2 est remplacé par `(unsigned int)`e'_2 (idem pour e'_1)
    — sinon pas de conversion
— sinon, pas de conversion

Pour donner un exemple, avec les définitions suivantes :

```
long x;
char y;
double z;
```

l'expression `x * y` est équivalent à `x * ((long) ((int) y))` et l'expression `y - z` est équivalente à `((double) y) - z`.

3.4 Spécificité de l'architecture x86-64

Appels de fonction externes. L'ABI x86-64 n'utilise pas (que) la pile pour passer des paramètres lors d'appels de fonctions, mais plutôt les registres `rdi`, `rsi`, `rdx`, `rcx` `r8` et `r9` dans cet ordre, pour les 6 premiers arguments entiers et rendent leur résultat (s'il est entier) dans `rax`. Afin de pouvoir appeler des fonctions externes, il faudra donc compiler spécialement leur appel et retour en utilisant ces registres.

Registre spéciaux certains registres ont une utilisation particulière. Le registre `%esp` contient l'adresse de la dernière valeur posée sur la pile et est modifié par les instructions

`pushq` et `popq`. On peut cependant le manipuler comme tout autre registre. Le registre `%ebp` contient le pointeur vers le *tableau d'activation*. Il doit être correctement sauvegardé et initialisé lors de l'entrée dans une fonction. L'accès aux paramètres et aux variables locales se fait alors en additionnant leur décalage par rapport à `%ebp`.

Il faudra en particulier tenir compte de cela pour compiler correctement le retour de la fonction `main`. Il ne faudra pas écrire son résultat sur la pile, mais plutot le placer dans le registre `rax`. Ce dernier sera le code de sortie de votre programme.

On lira aussi de manière attentive les ressources disponibles sur la page du cours ainsi que la documentation du module `Amd64` fourni, qui permet l'écriture de code assembleur en OCaml.

3.5 Test des programmes

Une fois le fichier assembleur (extension `.s`) généré par votre compilateur, vous pouvez le compiler avec :

```
gcc -o prog.exe -g prog.s
```

Vous pouvez ensuite l'exécuter directement sur votre machine (`./prog.exe`) ou l'exécuter à travers le débugger graphique `nemiver` installé au PUIO.

4 Rendu

La date final du rendu de rapport pour le TER est fixée au :

dimanche 23 avril, 2017, 21h00

via le formulaire de rendu en ligne sur la page du cours. Les soutenances auront lieu la semaine suivante (planning à venir).

4.1 Rapport et code

Un rapport de 15 pages maximum (police 11 points) au format PDF ou `odt` doit être rendu. Ce dernier doit être synthétique (pas besoin de 2 pages d'introduction ou conclusions pour un rapport de ce type). Il doit comporter deux parties, d'égale importance.

4.1.1 Partie conception du compilateur

Vous devez y décrire votre code, en particulier tous les fichiers de votre projet ainsi que leur dépendences (e.g. `typing.ml` dépend de `ast.mli`). Pour chacune des quatres phases de compilation (analyse lexicale, analyse syntaxique, typage et production de code), vous choisirez un exemple de code C et indiquerez comment il est traité par la phase en question dans votre compilateur (évidemment vous devez prendre un exemple non trivial). Vous vous servirez de cet exemple pour décrire *succinctement l'organisation du code* (par exemple les fonctions principales du module *typing* pour le typage) ainsi que les structures de données utilisées.

4.1.2 Partie recheche

Vous devez choisir l'un des concepts proposé en fin d'énoncé et le décrire précisément via une recherche bibliographique (attention, un copier collé de wikipedia ne suffit pas et vous prendrez soin de citer vos sources). Vous illustrez ce concept en fournissant quelques scénarios d'utilisation permettant de le « tester » avec le compilateur GCC. Vous devrez aussi en implanter une version simple dans votre compilateur.

4.1.3 Rendu du code

Vous devez rendre le code. Ce dernier sera évalué sur sa forme (commentaires, indentation, élégance générale et efficacité) ainsi que sur son comportement. Pour la partie typage et analyse syntaxique, votre compilateur doit implanter fidèlement l'énoncé. Pour la partie production de code, les structures ne seront pas testée (i.e. on ne testera pas le code produit pour des fichiers sources contenant des structures. Cependant, votre compilateur ne doit pas planter dans ce cas – vous pouvez par exemple générer des `nop` pour tout code utilisant des structures).

4.1.4 Soutenance

Les soutenances durent 10 minutes par étudiant (donc 20 minutes pour un binôme). Vous consacrerez 1/3 du temps expliquer ce que contient votre compilateur (si possible avec une démo) et 2/3 du temps sur le concept de recherche.

4.1.5 Concepts

Profilage de code . Expliquer en quoi consiste le profilage de code, les différents outils (sous Unix ou autre) permettant de profiler du code (code C en particulier, mais d'autres exemples par exemple pour du code Java ou Javascript sont les bienvenus), quelles métriques le profilage de code permet de collecter et la manière de les analyser. Votre compilateur devra avoir un mode profilage permettant de profiler le code généré.

Optimisations simples . Expliquer en quoi consiste l'*inlining* de fonctions, la technique du *constant folding* et la technique du *constant propagation*. Montrez (en utilisant GCC) comment ces différentes optimisations sont complémentaires. Vous expliquerez aussi ce qui peut bloquer ces optimisations (en particulier dans les langages de haut niveau). Vous implémenterez soit un *inliner* simple, soit un évaluateur partiel d'expressions arithmétiques.

Test boîte blanche et couverture de code . Expliquer en quoi consiste le test « boîte blanche » d'un logiciel (et toutes les techniques associées). Votre compilateur devra posséder un mode de génération de code où le programme exécuté affiche, en fin d'exécution la liste de tous les blocs de code non exécutés.

Allocateur de mémoire . Expliquer comment la mémoire est gérée sur un système d'exploitation moderne (mémoire virtuelle et *paging*) puis détailler les différentes manières d'implémenter `malloc` et `free`, avec les compromis que fait chaque méthode et

leur complexité. Votre compilateur doit implémenter (en assembleur) `malloc` et `free` en réutilisant l'appel système `sbrk`.