INF564 Compilation

Mini C

version 1 — 28 janvier 2019

L'objectif de ce projet est de réaliser un compilateur pour un fragment de C, appelé Mini C par la suite, produisant du code x86-64 raisonnablement efficace. Il s'agit d'un fragment du langage C contenant des entiers et des pointeurs vers structures, 100% compatible avec C, au sens où tout programme Mini C est aussi un programme C correct. Ceci permettra notamment d'utiliser un compilateur C existant comme référence, par exemple `gcc`. Le présent sujet décrit précisément Mini C, 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)
($\langle règle \rangle$)	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 $\langle ident \rangle$ suivante :

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

Les identificateurs suivants sont des mots clés :

```
int struct if else while return sizeof
```

Enfin les constantes littérales obéissent à l'expression régulière $\langle entier \rangle$ suivante :

⟨*entier*⟩ ::= `0`
| `1–9` ⟨*chiffre*⟩*
| `0` ⟨*chiffre-octal*⟩$^+$
| `0x` ⟨*chiffre-hexa*⟩$^+$
| `'` ⟨*caractère*⟩ `'`

⟨*chiffre-octal*⟩ ::= `0–7`

⟨*chiffre-hexa*⟩ ::= `0–9` | `a–f` | `A–F`

⟨*caractère*⟩ ::= tout caractère de code ASCII compris entre 32 et 127, autre que \, ' et "
| `\\` | `\'` | `\"`
| `\x` ⟨*chiffre-hexa*⟩ ⟨*chiffre-hexa*⟩

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 ci-dessous, de la plus faible à la plus forte précédence.

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

$$
\begin{array}{lcl}
\langle \textit{fichier} \rangle & ::= & \langle \textit{decl} \rangle^{\star}\ \texttt{EOF} \\
\langle \textit{decl} \rangle & ::= & \langle \textit{decl\_typ} \rangle \ \mid\ \langle \textit{decl\_fct} \rangle \\
\langle \textit{decl\_vars} \rangle & ::= & \texttt{int}\ \langle \textit{ident} \rangle^{+}_{,}\ \texttt{;} \\
 & \mid & \texttt{struct}\ \langle \textit{ident} \rangle\ (\texttt{*}\ \langle \textit{ident} \rangle)^{+}_{,}\ \texttt{;} \\
\langle \textit{decl\_typ} \rangle & ::= & \texttt{struct}\ \langle \textit{ident} \rangle\ \texttt{\{}\ \langle \textit{decl\_vars} \rangle^{\star}\ \texttt{\}}\ \texttt{;} \\
\langle \textit{decl\_fct} \rangle & ::= & \texttt{int}\ \langle \textit{ident} \rangle\ \texttt{(}\ \langle \textit{param} \rangle^{\star}_{,}\ \texttt{)}\ \langle \textit{bloc} \rangle \\
 & \mid & \texttt{struct}\ \langle \textit{ident} \rangle\ \texttt{*}\ \langle \textit{ident} \rangle\ \texttt{(}\ \langle \textit{param} \rangle^{\star}_{,}\ \texttt{)}\ \langle \textit{bloc} \rangle \\
\langle \textit{param} \rangle & ::= & \texttt{int}\ \langle \textit{ident} \rangle \ \mid\ \texttt{struct}\ \langle \textit{ident} \rangle\ \texttt{*}\ \langle \textit{ident} \rangle \\
\langle \textit{expr} \rangle & ::= & \langle \textit{entier} \rangle \\
 & \mid & \langle \textit{ident} \rangle \\
 & \mid & \langle \textit{expr} \rangle\ \texttt{->}\ \langle \textit{ident} \rangle \\
 & \mid & \langle \textit{ident} \rangle\ \texttt{(}\ \langle \textit{expr} \rangle^{\star}_{,}\ \texttt{)} \\
 & \mid & \texttt{!}\ \langle \textit{expr} \rangle \ \mid\ \texttt{-}\ \langle \textit{expr} \rangle \\
 & \mid & \langle \textit{expr} \rangle\ \langle \textit{opérateur} \rangle\ \langle \textit{expr} \rangle \\
 & \mid & \texttt{sizeof (}\ \texttt{struct}\ \langle \textit{ident} \rangle\ \texttt{)} \\
 & \mid & \texttt{(}\ \langle \textit{expr} \rangle\ \texttt{)} \\
\langle \textit{opérateur} \rangle & ::= & \texttt{=} \mid \texttt{==} \mid \texttt{!=} \mid \texttt{<} \mid \texttt{<=} \mid \texttt{>} \mid \texttt{>=} \mid \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/} \mid \texttt{\&\&} \mid \texttt{||} \\
\langle \textit{instruction} \rangle & ::= & \texttt{;} \\
 & \mid & \langle \textit{expr} \rangle\ \texttt{;} \\
 & \mid & \texttt{if (}\ \langle \textit{expr} \rangle\ \texttt{)}\ \langle \textit{instruction} \rangle \\
 & \mid & \texttt{if (}\ \langle \textit{expr} \rangle\ \texttt{)}\ \langle \textit{instruction} \rangle\ \texttt{else}\ \langle \textit{instruction} \rangle \\
 & \mid & \texttt{while (}\ \langle \textit{expr} \rangle\ \texttt{)}\ \langle \textit{instruction} \rangle \\
 & \mid & \langle \textit{bloc} \rangle \\
 & \mid & \texttt{return}\ \langle \textit{expr} \rangle\ \texttt{;} \\
\langle \textit{bloc} \rangle & ::= & \texttt{\{}\ \langle \textit{decl\_vars} \rangle^{\star}\ \langle \textit{instruction} \rangle^{\star}\ \texttt{\}}
\end{array}
$$

FIGURE 1 – Grammaire des fichiers C.

2 Typage statique

Une fois l'analyse syntaxique effectuée avec succès, on vérifie la conformité du fichier source.

2.1 Types et environnements

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

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

où *id* désigne un identificateur de structure. 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

$$\frac{}{\texttt{typenull} \equiv \texttt{int}} \qquad \frac{}{\texttt{typenull} \equiv \texttt{struct } id \texttt{ *}} \qquad \frac{}{\texttt{void*} \equiv \texttt{struct } id \texttt{ *}}$$

Un environnement de typage Γ est une suite de déclarations de variables de la forme $\tau\ x$, de déclarations de structures de la forme $\texttt{struct } S \texttt{ \{} \tau_1\ x_1 \cdots \tau_n\ x_n \texttt{\}}$ et de déclarations de profils de fonctions de la forme $\tau\ f(\tau_1, \ldots, \tau_n)$. On notera $\texttt{struct } S \texttt{ \{} \tau\ x \texttt{\}}$ pour indiquer que la structure S contient un champ x de type τ.

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.

2.2 Typage des expressions

On définit la notion de *valeur gauche* de la manière suivante :

$$\begin{aligned} \mathit{lvalue}(x) &= \texttt{true} \text{ si } x \text{ est une variable} \\ \mathit{lvalue}(e\texttt{->}x) &= \texttt{true} \\ \mathit{lvalue}(e) &= \texttt{false}, \quad \text{sinon.} \end{aligned}$$

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 :

$$\frac{}{\Gamma \vdash \texttt{0} : \texttt{typenull}} \qquad \frac{c \text{ constante entière}}{\Gamma \vdash c : \texttt{int}} \qquad \frac{\tau\ x \in \Gamma}{\Gamma \vdash x : \tau}$$

$$\frac{\Gamma \vdash e : \texttt{struct } S * \quad \texttt{struct } S \texttt{ \{} \tau\ x \texttt{\}} \in \Gamma}{\Gamma \vdash e\texttt{->}x : \tau} \qquad \frac{\texttt{struct } S \in \Gamma}{\Gamma \vdash \texttt{sizeof(struct } S\texttt{)} : \texttt{int}}$$

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

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

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

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

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

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

2.3 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 ». Intuitivement, 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 :

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

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

$$\frac{\Gamma \vdash e : \tau \quad \Gamma \vdash^{\tau_0} i}{\Gamma \vdash^{\tau_0} \texttt{while}(e)\ i}$$

$$\frac{\forall j,\ \Gamma \vdash \tau_j\ \mathsf{bf} \quad \forall j,\ \{\tau_1\ x_1, \ldots, \tau_k\ x_k\} \cup \Gamma \vdash^{\tau_0} i_j}{\Gamma \vdash^{\tau_0} \{\tau_1\ x_1 \cdots \tau_k\ x_k ; i_1 \cdots i_n\}}$$

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.

2.4 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 structures

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

On notera que les types de champs τ_i ne peuvent faire référence à la structure id elle-même que sous un pointeur.

Déclarations de fonctions

$$\frac{\forall i,\ \Gamma \vdash \tau_i\ \mathsf{bf} \quad \{\tau_0\ f(\tau_1, \ldots, \tau_n), \tau_1\ x_1, \ldots, \tau_n\ x_n\} \cup \Gamma \vdash^{\tau_0} b}{\Gamma \vdash \tau_0\ f(\tau_1\ x_1, \ldots, \tau_n\ x_n)\ b \to \{\tau_0\ f(\tau_1, \ldots, \tau_n)\} \cup \Gamma}$$

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.

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 $d_1, \ldots, d_n$ 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 identificateurs des paramètres d'une fonction ;
- des variables à l'intérieur d'un *même* bloc ;
- des symboles de fonctions sur l'ensemble du fichier.

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

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

Point d'entrée. Enfin, on vérifiera la présence d'une fonction `main` avec le profil suivant :

```
int main();
```

2.5 Indications

Messages d'erreurs. Vous pouvez vous inspirer des messages d'erreur d'un compilateur C existant (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.

3 Production de code

L'objectif est de réaliser un compilateur produisant du code x86-64, avec allocation de registres, en suivant l'architecture présentée en cours. On pourra choisir une représentation simple des valeurs, où chaque valeur occupe exactement un mot de 8 octets (64 bits), qu'il s'agisse d'un entier ou d'un pointeur. Même si un compilateur C plus réaliste stockerait une valeur de type `int` sur 32 bits, il n'est pas incorrect de la représenter sur 64 bits. Cela simplifie certains aspects du compilateur.

4 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 :

- Il n'y a pas d'initialisation pour les variables. Pour initialiser une variable, il faut utiliser une instruction d'affectation.
- Il n'y a pas d'arithmétique de pointeurs.
- Mini C possède moins de mots clés que C.

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

5 Travail demandé (pour le dimanche 17 mars 18h)

Le projet est à faire seul ou en binôme, en Java ou en OCaml. Il doit être remis par email à `filliatr@lri.fr`, sous la 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* du compilateur (inutile d'inclure les fichiers compilés). Quand on se place dans ce répertoire, la commande `make` doit créer le compilateur, qui sera appelé `minic`. Bien entendu, la compilation du projet peut être réalisée avec d'autres outils

que `make` (par exemple `ocamlbuild` ou `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 ASCII, Markdown ou PDF.

Le compilateur `minic` doit accepter sur sa ligne de commande une option éventuelle (parmi `--parse-only` et `--type-only`) et exactement un fichier Mini 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 "file.c", 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.

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 "file.c", line 4, characters 5-6:
this expression has type int but is expected to have type struct S*
```

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.

Si le fichier est correctement typé, le compilateur doit terminer avec le code de sortie 0 si l'option `--type-only` a été passée sur la ligne de commande. Sinon, 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 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 C `file.c`, par exemple avec

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

En cas d'erreur du compilateur lui-même, le compilateur doit terminer avec le code de sortie 2.

Remarque importante. La correction du projet est réalisée en partie automatiquement, à l'aide d'un jeu de petits programmes réalisant des affichages avec la fonction `putchar`, 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 `putchar`.