

INF564 Compilation

Mini C

version 1 — 1^{er} février 2018

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 :

$\langle \text{r\`egle} \rangle^*$	répétition de la règle $\langle \text{r\`egle} \rangle$ un nombre quelconque de fois (y compris aucune)
$\langle \text{r\`egle} \rangle_t^*$	répétition de la règle $\langle \text{r\`egle} \rangle$ un nombre quelconque de fois (y compris aucune), les occurrences étant séparées par le terminal t
$\langle \text{r\`egle} \rangle^+$	répétition de la règle $\langle \text{r\`egle} \rangle$ au moins une fois
$\langle \text{r\`egle} \rangle_t^+$	répétition de la règle $\langle \text{r\`egle} \rangle$ au moins une fois, les occurrences étant séparées par le terminal t
$\langle \text{r\`egle} \rangle?$	utilisation optionnelle de la règle $\langle \text{r\`egle} \rangle$ (<i>i.e.</i> 0 ou 1 fois)
$(\langle \text{r\`egle} \rangle)$	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 $\langle \text{ident} \rangle$ suivante :

$$\begin{aligned} \langle \text{chiffre} \rangle &::= 0-9 \\ \langle \text{alpha} \rangle &::= \text{a-z} \mid \text{A-Z} \\ \langle \text{ident} \rangle &::= (\langle \text{alpha} \rangle \mid _)(\langle \text{alpha} \rangle \mid \langle \text{chiffre} \rangle \mid _)^* \end{aligned}$$

Les identificateurs suivants sont des mots clés :

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

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

$$\begin{aligned}
\langle \text{entier} \rangle &::= 0 \\
&| 1-9 \langle \text{chiffre} \rangle^* \\
&| 0 \langle \text{chiffre-octal} \rangle^+ \\
&| 0x \langle \text{chiffre-hexa} \rangle^+ \\
&| ' \langle \text{caractère} \rangle ' \\
\langle \text{chiffre-octal} \rangle &::= 0-7 \\
\langle \text{chiffre-hexa} \rangle &::= 0-9 \mid a-f \mid A-F \\
\langle \text{caractère} \rangle &::= \text{tout caractère de code ASCII compris entre 32 et 127,} \\
&\text{autre que } \backslash, ' \text{ et } " \\
&| \backslash \backslash \mid \backslash ' \mid \backslash " \\
&| \backslash x \langle \text{chiffre-hexa} \rangle \langle \text{chiffre-hexa} \rangle
\end{aligned}$$

1.2 Syntaxe

La grammaire des fichiers sources considérée est donnée figure 1. Le point d'entrée est le non-terminal $\langle \text{fichier} \rangle$. Les associativités et précédences des divers opérateurs sont données par la table ci-dessous, de la plus faible à la plus forte précedence.

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

$\langle \text{fichier} \rangle$	$::=$	$\langle \text{decl} \rangle^* \text{EOF}$
$\langle \text{decl} \rangle$	$::=$	$\langle \text{decl_typ} \rangle \mid \langle \text{decl_fct} \rangle$
$\langle \text{decl_vars} \rangle$	$::=$	$\text{int } \langle \text{ident} \rangle^+ ;$ \mid $\text{struct } \langle \text{ident} \rangle (* \langle \text{ident} \rangle)^+ ;$
$\langle \text{decl_typ} \rangle$	$::=$	$\text{struct } \langle \text{ident} \rangle \{ \langle \text{decl_vars} \rangle^* \} ;$
$\langle \text{decl_fct} \rangle$	$::=$	$\text{int } \langle \text{ident} \rangle (\langle \text{param} \rangle^*) \langle \text{bloc} \rangle$ \mid $\text{struct } \langle \text{ident} \rangle * \langle \text{ident} \rangle (\langle \text{param} \rangle^*) \langle \text{bloc} \rangle$
$\langle \text{param} \rangle$	$::=$	$\text{int } \langle \text{ident} \rangle \mid \text{struct } \langle \text{ident} \rangle * \langle \text{ident} \rangle$
$\langle \text{expr} \rangle$	$::=$	$\langle \text{entier} \rangle$ \mid $\langle \text{ident} \rangle$ \mid $\langle \text{expr} \rangle \rightarrow \langle \text{ident} \rangle$ \mid $\langle \text{ident} \rangle (\langle \text{expr} \rangle^*)$ \mid $! \langle \text{expr} \rangle \mid - \langle \text{expr} \rangle$ \mid $\langle \text{expr} \rangle \langle \text{opérateur} \rangle \langle \text{expr} \rangle$ \mid $\text{sizeof } (\text{struct } \langle \text{ident} \rangle)$ \mid $(\langle \text{expr} \rangle)$
$\langle \text{opérateur} \rangle$	$::=$	$= \mid == \mid != \mid < \mid <= \mid > \mid >= \mid + \mid - \mid * \mid / \mid \&\& \mid \mid\mid$
$\langle \text{instruction} \rangle$	$::=$	$;$ \mid $\langle \text{expr} \rangle ;$ \mid $\text{if } (\langle \text{expr} \rangle) \langle \text{instruction} \rangle$ \mid $\text{if } (\langle \text{expr} \rangle) \langle \text{instruction} \rangle \text{ else } \langle \text{instruction} \rangle$ \mid $\text{while } (\langle \text{expr} \rangle) \langle \text{instruction} \rangle$ \mid $\langle \text{bloc} \rangle$ \mid $\text{return } \langle \text{expr} \rangle ;$
$\langle \text{bloc} \rangle$	$::=$	$\{ \langle \text{decl_vars} \rangle^* \langle \text{instruction} \rangle^* \}$

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 :

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

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

$$\overline{\text{typenull} \equiv \text{int}} \quad \overline{\text{typenull} \equiv \text{struct } id *} \quad \overline{\text{void} * \equiv \text{struct } id *}$$

Un environnement de typage Γ est une suite de déclarations de variables de la forme τx , 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 $\tau f(\tau_1, \dots, \tau_n)$. On notera **struct** $S \{ \tau x \}$ 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 $\Gamma \vdash \tau \text{ bf}$, 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 :

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

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 :

$$\begin{array}{c} \overline{\Gamma \vdash 0 : \text{typenull}} \quad \overline{c \text{ constante entière} \quad \Gamma \vdash c : \text{int}} \quad \overline{\tau x \in \Gamma \quad \Gamma \vdash x : \tau} \\ \overline{\Gamma \vdash e : \text{struct } S * \quad \text{struct } S \{ \tau x \} \in \Gamma \quad \text{struct } S \in \Gamma} \quad \overline{\Gamma \vdash \text{sizeof}(\text{struct } S) : \text{int}} \\ \overline{\Gamma \vdash e_1 : \tau_1 \quad lvalue(e_1) \quad \Gamma \vdash e_2 : \tau_2 \quad \tau_1 \equiv \tau_2} \\ \quad \Gamma \vdash e_1 = e_2 : \tau_1 \\ \overline{\Gamma \vdash e : \tau \quad \tau \equiv \text{int}} \quad \overline{\Gamma \vdash e : \tau} \\ \quad \Gamma \vdash - e : \text{int} \quad \Gamma \vdash ! e : \text{int} \\ \overline{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2 \quad \tau_1 \equiv \tau_2 \quad op \in \{==, !=, <, <=, >, >=\}} \\ \quad \Gamma \vdash e_1 op e_2 : \text{int} \\ \overline{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2 \quad op \in \{||, \&\&\}} \\ \quad \Gamma \vdash e_1 op e_2 : \text{int} \\ \overline{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2 \quad \tau_1 \equiv \text{int} \quad \tau_2 \equiv \text{int} \quad op \in \{+, -, *, /\}} \\ \quad \Gamma \vdash e_1 op e_2 : \text{int} \\ \overline{\tau f(\tau'_1, \dots, \tau'_n) \in \Gamma \quad \forall i, \Gamma \vdash e_i : \tau_i \quad \tau_i \equiv \tau'_i} \\ \quad \Gamma \vdash f(e_1, \dots, e_n) : \tau \end{array}$$

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 :

$$\frac{}{\Gamma \vdash^{\tau_0} ;} \quad \frac{\Gamma \vdash e : \tau}{\Gamma \vdash^{\tau_0} e;} \quad \frac{\Gamma \vdash e : \tau \quad \tau \equiv \tau_0}{\Gamma \vdash^{\tau_0} \text{return } e;} \\ \frac{\Gamma \vdash e : \tau \quad \Gamma \vdash^{\tau_0} i_1 \quad \Gamma \vdash^{\tau_0} i_2}{\Gamma \vdash^{\tau_0} \text{if } (e) i_1 \text{ else } i_2} \\ \frac{\Gamma \vdash e : \tau \quad \Gamma \vdash^{\tau_0} i}{\Gamma \vdash^{\tau_0} \text{while}(e) i} \\ \frac{\forall j, \Gamma \vdash \tau_j \text{ bf} \quad \forall j, \{\tau_1 x_1, \dots, \tau_k x_k\} \cup \Gamma \vdash^{\tau_0} i_j}{\Gamma \vdash^{\tau_0} \{\tau_1 x_1 \dots \tau_k x_k; i_1 \dots i_n\}}$$

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 $\Gamma \vdash d \rightarrow \Gamma'$ 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

$$\frac{\forall i, \Gamma, \text{struct } id \{\tau_1 x_1 \dots \tau_n x_n\} \vdash \tau_i \text{ bf}}{\Gamma \vdash \text{struct } id \{\tau_1 x_1; \dots \tau_n x_n\} \rightarrow \{\text{struct } id \{\tau_1 x_1 \dots \tau_n x_n\}\} \cup \Gamma}$$

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

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

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 $\Gamma \vdash_f d_1 \dots d_n$ signifiant « dans l'environnement Γ le fichier constitué par la suite de déclarations d_1, \dots, 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 :

$$\frac{}{\Gamma \vdash_f \emptyset} \quad \frac{\Gamma \vdash d_1 \rightarrow \Gamma' \quad \Gamma' \vdash_f d_2 \dots d_n}{\Gamma \vdash_f d_1 d_2 \dots d_n}$$

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 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).