École Normale Supérieure 2012–2013

projet de compilation

Mini C

version 1 — 10 octobre 2012

L'objectif de ce projet est de réaliser un compilateur pour un fragment de C, appelé Mini C par la suite, produisant du code MIPS. Il s'agit d'un fragment contenant des entiers, des pointeurs, des chaînes de caractères et des structures. Il s'agit d'un fragment 100% compatible avec C, au sens où tout programme de 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 :

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

Les identificateurs suivants sont des mots clés :

```
char else for if int return sizeof struct union void while
```

Enfin les constantes littérales (entiers ou chaînes de caractères) obéissent aux expressions régulières ⟨*entier*⟩ et ⟨*chaîne*⟩ suivantes :

⟨*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*⟩
⟨*chaîne*⟩	::=	`"` ⟨*caractère*⟩* `"`

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.

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`)` et que l'expression e`->`id a été remplacée par `(*`e`).`id.

2.1 Types et environnements

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

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

où id désigne un identificateur de structure ou d'union et s une constante entière. 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

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

On définit par ailleurs le prédicat suivant sur les types :

$$num(\tau) \quad \Leftrightarrow \quad \tau \equiv \texttt{int} \quad \text{ou} \quad \tau \equiv \tau'\texttt{*}$$

⟨*fichier*⟩ ::= ⟨*decl*⟩* `EOF`
⟨*decl*⟩ ::= ⟨*decl_vars*⟩ | ⟨*decl_typ*⟩ | ⟨*decl_fct*⟩
⟨*decl_vars*⟩ ::= ⟨*type*⟩ ⟨*var*⟩$^+_,$ `;`
⟨*decl_typ*⟩ ::= `(` `struct` | `union` `)` ⟨*ident*⟩ `{` ⟨*decl_vars*⟩* `}` `;`
⟨*decl_fct*⟩ ::= ⟨*type*⟩ `*`* ⟨*ident*⟩ `(` ⟨*argument*⟩$^\star_,$ `)` ⟨*bloc*⟩
⟨*type*⟩ ::= `void` | `int` | `char` | `struct` ⟨*ident*⟩ | `union` ⟨*ident*⟩
⟨*argument*⟩ ::= ⟨*type*⟩ ⟨*var*⟩
⟨*var*⟩ ::= ⟨*ident*⟩ | `*` ⟨*var*⟩
⟨*expr*⟩ ::= ⟨*entier*⟩ | ⟨*chaîne*⟩
| ⟨*ident*⟩
| `*` ⟨*expr*⟩
| ⟨*expr*⟩ `[` ⟨*expr*⟩ `]`
| ⟨*expr*⟩ `.` ⟨*ident*⟩
| ⟨*expr*⟩ `->` ⟨*ident*⟩
| ⟨*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` `(` ⟨*expr*⟩$^\star_,$ `;` ⟨*expr*⟩? `;` ⟨*expr*⟩$^\star_,$ `)` ⟨*instruction*⟩
| ⟨*bloc*⟩
| `return` ⟨*expr*⟩? `;`
⟨*bloc*⟩ ::= `{` ⟨*decl_vars*⟩* ⟨*instruction*⟩* `}`

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

FIGURE 1 – Grammaire des fichiers C.

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

On dit qu'un type τ est *bien formé* dans un environnement Γ, et on note $\Gamma \vdash \tau\ \mathsf{bf}$, si tous les identificateurs de structures ou d'unions apparaissant dans τ correspondent à des structures ou des unions déclarées dans Γ.

2.2 Typage des expressions

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

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

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 :

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

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

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

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

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

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

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

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

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

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

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2 \quad \tau_1 \equiv \tau_2'\texttt{*} \quad \tau_2 \equiv \tau_2'\texttt{*}}{\Gamma \vdash e_1 \texttt{ - } e_2 : \texttt{int}} \qquad \frac{\Gamma \vdash e_2 \texttt{ + } e_1 : \tau}{\Gamma \vdash e_1 \texttt{ + } e_2 : \tau}$$

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

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 la quelle se trouve l'instruction i. Ce jugement est établi par les règles d'inférence suivantes :

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

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

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

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

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

Cette dernière règle signifie que pour type 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.

De plus, on a les équivalences suivantes :

- `if (`e_1`)` e_2 équivaut à `if (`e_1`)` e_2 `else;`
- si i_1 ou i_3 est omis dans `for(`i_1`;`e_2`;`i_3`)` alors il équivaut à `;`
- si e_2 est omis dans `for(`i_1`;`e_2`;`i_3`)` alors il équivaut à `1`
- dans la construction `for`, une liste d'expressions $e_1, e_2, \ldots, e_n$ équivaut au bloc `{`e_1`;`e_2`;` $\ldots$ `;`e_n`}`.

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 variables globales

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

Déclarations de structures et d'unions

$$\frac{\Gamma, \texttt{struct}\ id\ \texttt{\{}\tau_1\ x_1 \cdots \tau_n\ x_n\texttt{\}} \vdash \tau_i\ \texttt{bf} \qquad \tau_i \not\equiv \texttt{void}}{\Gamma \vdash \texttt{struct}\ id\ \texttt{\{}\tau_1\ x_1\texttt{;} \cdots \tau_n\ x_n\texttt{;\}} \to \{\texttt{struct}\ id\ \texttt{\{}\tau_1\ x_1 \cdots \tau_n\ x_n\texttt{\}}\} \cup \Gamma}$$

$$\frac{\Gamma, \texttt{union}\ id\ \texttt{\{}\tau_1\ x_1 \cdots \tau_n\ x_n\texttt{\}} \vdash \tau_i\ \texttt{bf} \qquad \tau_i \not\equiv \texttt{void}}{\Gamma \vdash \texttt{union}\ id\ \texttt{\{}\tau_1\ x_1\texttt{;} \cdots \tau_n\ x_n\texttt{;\}} \to \{\texttt{union}\ id\ \texttt{\{}\tau_1\ x_1 \cdots \tau_n\ x_n\texttt{\}}\} \cup \Gamma}$$

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

Déclarations de fonctions

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

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, …, 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 identificateurs d'unions 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.

On notera qu'une structure et une union peuvent avoir le même identificateur.

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 l'un des deux profils suivants :

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

2.5 Indications

Tests. En cas de doute concernant un point de sémantique, vous pouvez utiliser un compilateur C comme référence. Vous pouvez d'ailleurs vous inspirer de ses messages d'erreur pour votre compilateur (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 (locales ou globales). Pour initialiser une variable, il faut utiliser des instructions.
- On suppose que le type `int` correspond à des entiers 32 bits signés.
- Mini C possède moins de mots clés que C.
- On ne se servira jamais de la valeur renvoyée par `putchar`.

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

4 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 MIPS.

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 4.
- les structures (et les unions) sont affectées, passées en argument et renvoyées comme résultat *par valeur*, ce qui nécessite de copier tout un bloc d'octets.

4.1 Représentation des valeurs

On commence par définir la notion d'*alignement*. Un entier de type `int` ou un pointeur doit être aligné. Une structure ou une union doit être alignée si l'un quelconque de ses champs doit être aligné. Une donnée alignée doit être placée en mémoire à une adresse multiple de 4.

Une valeur de type `char` sera stockée sur 8 bits non signés et une valeur de type `int` ou de type pointeur sur 32 bits signés. Une structure est représentée comme la juxtaposition 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 et sera représentée sur 12 octets, dont 2 octets de remplissage marqués X, de la manière suivante :

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

Une union est représentée comme la superposition des représentations de ses différents champs. Une structure ou une union qui doit être alignée occupe nécessairement un nombre d'octets multiple de 4. Ainsi la structure

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

occupe 8 octets. Ceci permet notamment d'allouer un « tableau » de `n` telles structures avec l'idiome

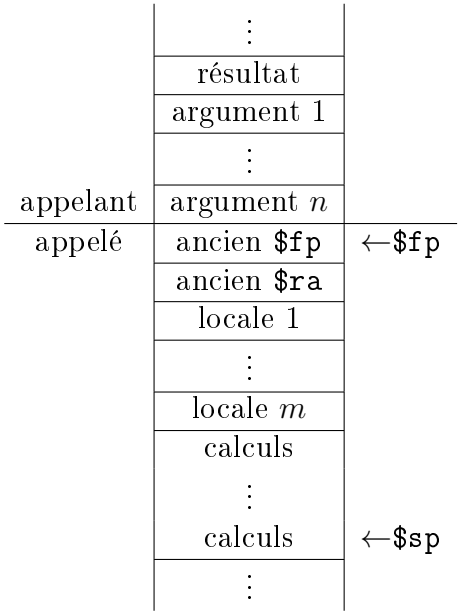
```
struct T *a = sbrk(n * sizeof(struct T));
```

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)` $= 12$ et `sizeof(struct T)` $= 8$ pour les deux structures ci-dessus.

4.2 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 MIPS.

Les deux fonctions prédéfinies `putchar` et `sbrk` seront respectivement réalisées avec les appels systèmes 11 et 9 et MIPS. On ne cherchera pas à positionner la valeur de retour de `putchar`.

Suggestion d'organisation. On pourra procéder selon les étapes suivantes :

1. calculer la taille de toutes les structures et unions ;
2. pour chaque fonction f du code :
   (a) calculer l'emplacement de ses arguments/résultat sur la pile,
   (b) calculer l'emplacement de ses variables locales sur la pile,

   (c) produire le code MIPS de la fonction f ;
3. ajouter le code de `putchar`, `sbrk` et des fonctions auxiliaires, le cas échéant ;
4. allouer les variables globales sur le tas.

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`.

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). Quand on se place dans ce répertoire, la commande `make` doit créer votre compilateur, qui sera appelé `minic`. 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 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, PostScript ou PDF.

5.1 Pour le vendredi 30 novembre (avant minuit)

Écrire un compilateur, appelons-le `minic`, acceptant sur sa ligne de commande exactement un fichier Mini C (portant l'extension `.c`) et éventuellement l'option `-parse-only` ou l'option `-type-only`. Ces deux options indiquent respectivement de stopper la compilation après l'analyse syntaxique (section 1) et l'analyse sémantique (section 2).

En cas d'erreur lexicale, syntaxique ou de typage, celle-ci doit être signalée (de la manière indiquée ci-dessous) et le programme doit terminer avec le code de sortie 1 (`exit 1`). En cas d'autre erreur (une erreur du compilateur lui-même), le programme doit terminer avec le code de sortie 2 (`exit 2`).

Lorsqu'une erreur est détectée par votre compilateur, elle 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 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.) Les localisations peuvent être obtenues pendant l'analyse syntaxique grâce aux mots-clés `$startpos` et `$endpos` de Menhir, puis conservées dans l'arbre de syntaxe abstraite.

5.2 Pour le vendredi 11 janvier (avant minuit)

Étendre le compilateur précédent de la manière suivante. Si le fichier d'entrée est conforme à la syntaxe et au typage décrits dans ce document, votre compilateur doit produire du code MIPS 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.c`, le code MIPS doit être produit dans le fichier `file.s` (même nom que le fichier source mais suffixe `.s` au lieu de `.c`). Ce fichier MIPS doit pouvoir être exécuté avec la commande

```
spim -file file.s arg1 arg2 ...
```

ou

```
java -jar Mars_4_2.jar file.s pa file.s arg1 arg2 ...
```

Les chaînes de caractères `arg1`, `arg2`, etc., sont les arguments passés au programme. Le résultat affiché sur la sortie standard doit être identique à celui donné par l'exécution du fichier C `file.c`, c'est-à-dire obtenu avec les commandes

```
gcc file.c -o file
./file arg1 arg2 ...
```

(à l'exception des cinq premières lignes ajoutées par SPIM ou du dernier retour-chariot ajouté par MARS). Le premier argument de ligne de commande (qui est le nom de fichier `file.s` ou `./file` dans les commandes ci-dessus) ne sera jamais utilisé dans les tests de votre compilateur.