projet de compilation

Petit Rust

version 2 — 30 novembre 2017

L'objectif de ce projet est de réaliser un compilateur pour un fragment de Rust, appelé Petit Rust par la suite, produisant du code x86-64. Il s'agit d'un fragment contenant des entiers, des pointeurs, des tableaux et des structures. Il s'agit d'un fragment 100% compatible avec Rust, au sens où tout programme de Petit Rust est aussi un programme Rust correct. Ceci permettra notamment d'utiliser un compilateur Rust existant comme référence. Le présent sujet décrit la syntaxe de Petit Rust.

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

Les identificateurs obéissent à l'expression régulière $\langle ident \rangle$ suivante :

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

Les identificateurs suivants sont des mots clés :

```
else false fn if let mut return struct true while
```

Enfin, les constantes littérales (entiers ou chaînes de caractères) obéissent aux expressions régulières $\langle entier \rangle$ et $\langle chaîne \rangle$ suivantes :

$$\begin{array}{lll} \langle entier \rangle & ::= & \langle chiffre \rangle^\star \\ \langle caractère \rangle & ::= & \text{tout caractère autre que } \texttt{\textbackslash} \text{ et } \texttt{"} \\ & \mid & \texttt{\textbackslash\textbackslash} \mid \texttt{\textbackslash"} \mid \texttt{\textbackslash n} \\ \langle chaîne \rangle & ::= & \texttt{"}\ \langle caractère \rangle^\star\ \texttt{"} \end{array}$$

1.2 Syntaxe

La grammaire des fichiers sources 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	faible
`||`	à gauche	
`&&`	à gauche	
`==` `!=` `<` `<=` `>` `>=`	—	↓
`+` `-`	à gauche	
`*` `/` `%`	à gauche	
`!` `*` (unaire) `-` (unaire) `&` `&mut`	—	
`[]`	—	
`.`	—	forte

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 :

$$\tau ::= \texttt{()} \mid \texttt{i32} \mid \texttt{bool} \mid S \mid \texttt{Vec<}\tau\texttt{>} \mid \texttt{\&}m\,\tau$$

où S désigne un identificateur de structure et m un indicateur de mutabilité, c'est-à-dire `mut` ou rien. Il s'agit là d'une notation pour la syntaxe *abstraite* des expressions de types. Un type de la forme $\texttt{\&}m\,\tau$ est appelé un *type emprunt* (*borrow* en anglais).

On introduit sur les types la relation $\tau_1 \leq \tau_2$ qui signifie « toute valeur de type τ_1 peut être utilisée là où une valeur de type τ_2 est attendue ». On la définit avec les règles suivantes :

$$\frac{}{\tau \leq \tau} \qquad \frac{}{\texttt{\&mut}\,\tau \leq \texttt{\&}\,\tau}$$

Un environnement de typage Γ est une suite de déclarations de variables de la forme $m\,x : \tau$, de déclarations de structures de la forme $\texttt{struct}\ S\ \texttt{\{}x_1 : \tau_1, \ldots, x_n : \tau_n\texttt{\}}$ et de déclarations de profils de fonctions de la forme $f(\tau_1, \ldots, \tau_n) \to \tau$. On notera $\texttt{struct}\ S\ \texttt{\{}x : \tau\texttt{\}}$ 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 Règles de typage

On note τ_r le type de retour de la fonction en cours de vérification. On introduit les trois jugements de typage suivants :

$\Gamma \vdash e : \tau$, pour « l'expression e est bien typée de type τ »;

$\Gamma \vdash_l e : \tau$, pour « l'expression e est une valeur gauche bien typée de type τ »;

$\Gamma \vdash_{\texttt{mut}} e$, pour « l'expression e est mutable ».

$$
\begin{array}{lcl}
\langle fichier \rangle & ::= & \langle decl \rangle^\star \texttt{ EOF} \\
\langle decl \rangle & ::= & \langle decl\_fun \rangle \mid \langle decl\_struct \rangle \\
\langle decl\_struct \rangle & ::= & \texttt{struct } \langle ident \rangle \texttt{ \{ } (\langle ident \rangle \texttt{ : } \langle type \rangle)^\star_{\texttt{,}} \texttt{ \}} \\
\langle decl\_fun \rangle & ::= & \texttt{fn } \langle ident \rangle \texttt{ (} \langle argument \rangle^\star_{\texttt{,}} \texttt{) } (\texttt{->} \langle type \rangle)? \ \langle bloc \rangle \\
\langle type \rangle & ::= & \langle ident \rangle \mid \langle ident \rangle \texttt{ < } \langle type \rangle \texttt{ > } \mid \texttt{ \& } \langle type \rangle \mid \texttt{ \& mut } \langle type \rangle \\
\langle argument \rangle & ::= & \texttt{mut}? \ \langle ident \rangle \texttt{ : } \langle type \rangle \\
\langle bloc \rangle & ::= & \texttt{\{ } \langle instr \rangle^\star \ \langle expr \rangle? \texttt{ \}} \\
\langle instr \rangle & ::= & \texttt{ ;} \\
& \mid & \langle expr \rangle \texttt{ ;} \\
& \mid & \texttt{let mut}? \ \langle ident \rangle \texttt{ = } \langle expr \rangle \texttt{ ;} \\
& \mid & \texttt{let mut}? \ \langle ident \rangle \texttt{ = } \langle ident \rangle \texttt{ \{ } (\langle ident \rangle \texttt{ : } \langle expr \rangle)^\star_{\texttt{,}} \texttt{ \} ;} \\
& \mid & \texttt{while } \langle expr \rangle \ \langle bloc \rangle \\
& \mid & \texttt{return } \langle expr \rangle? \texttt{ ;} \\
& \mid & \langle if \rangle \\
\langle if \rangle & ::= & \texttt{if } \langle expr \rangle \ \langle bloc \rangle \ (\texttt{else } (\langle bloc \rangle \mid \langle if \rangle))? \\
\langle expr \rangle & ::= & \langle entier \rangle \mid \texttt{ true } \mid \texttt{ false} \\
& \mid & \langle ident \rangle \\
& \mid & \langle expr \rangle \ \langle binaire \rangle \ \langle expr \rangle \\
& \mid & \langle unaire \rangle \ \langle expr \rangle \\
& \mid & \langle expr \rangle \texttt{ . } \langle ident \rangle \\
& \mid & \langle expr \rangle \texttt{ . len ( )} \\
& \mid & \langle expr \rangle \texttt{ [} \langle expr \rangle \texttt{]} \\
& \mid & \langle ident \rangle \texttt{ (} \langle expr \rangle^\star_{\texttt{,}} \texttt{)} \\
& \mid & \texttt{vec ! [} \langle expr \rangle^\star_{\texttt{,}} \texttt{]} \\
& \mid & \texttt{print ! (} \langle chaîne \rangle \texttt{)} \\
& \mid & \langle bloc \rangle \\
& \mid & \texttt{(} \langle expr \rangle \texttt{)} \\
\langle binaire \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{ || } \mid \texttt{ =} \\
\langle unaire \rangle & ::= & \texttt{- } \mid \texttt{ ! } \mid \texttt{ * } \mid \texttt{ \& } \mid \texttt{ \& mut}
\end{array}
$$

FIGURE 1 – Grammaire des fichiers Petit Rust.

Ces jugements de typage sont définis par les règles d'inférence suivantes :

$$\frac{c \text{ constante de type } \tau}{\Gamma \vdash c : \tau} \qquad \frac{m\, x : \tau \in \Gamma}{\Gamma \vdash_l x : \tau} \qquad \frac{\Gamma \vdash_l e : \tau}{\Gamma \vdash e : \tau}$$

$$\frac{\Gamma \vdash_l e_1 : \tau_1 \quad \Gamma \vdash_{\texttt{mut}} e_1 \quad \Gamma \vdash e_2 : \tau_2 \quad \tau_2 \leq \tau_1}{\Gamma \vdash e_1 \texttt{ = } e_2 : \texttt{()}}$$

$$\frac{\Gamma \vdash e : \texttt{i32}}{\Gamma \vdash \texttt{-}\ e : \texttt{i32}} \qquad \frac{\Gamma \vdash e : \texttt{bool}}{\Gamma \vdash \texttt{!}\ e : \texttt{bool}}$$

$$\frac{\Gamma \vdash e_1 : \texttt{i32} \quad \Gamma \vdash e_2 : \texttt{i32} \quad op \in \{\texttt{==}, \texttt{!=}, \texttt{<}, \texttt{<=}, \texttt{>}, \texttt{>=}\}}{\Gamma \vdash e_1\ op\ e_2 : \texttt{bool}}$$

$$\frac{\Gamma \vdash e_1 : \texttt{i32} \quad \Gamma \vdash e_2 : \texttt{i32} \quad op \in \{\texttt{+}, \texttt{-}, \texttt{*}, \texttt{/}, \texttt{\%}\}}{\Gamma \vdash e_1\ op\ e_2 : \texttt{i32}}$$

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

$$\frac{\Gamma \vdash_l e : \tau}{\Gamma \vdash \texttt{\&}\, e : \texttt{\&}\, \tau} \qquad \frac{\Gamma \vdash_l e : \tau \quad \Gamma \vdash_{\texttt{mut}} e}{\Gamma \vdash \texttt{\&mut}\, e : \texttt{\&mut}\, \tau} \qquad \frac{\Gamma \vdash e : \texttt{\&}m\, \tau}{\Gamma \vdash_l \texttt{*}e : \tau}$$

$$\frac{\forall i.\ \Gamma \vdash e_i : \tau}{\Gamma \vdash \texttt{vec![}e_1, \ldots, e_n\texttt{]} : \texttt{Vec<}\tau\texttt{>}} \qquad \frac{\Gamma \vdash e : \texttt{Vec<}\tau\texttt{>}}{\Gamma \vdash e\texttt{.len()} : \texttt{i32}} \qquad \frac{\Gamma \vdash_l e_1 : \texttt{Vec<}\tau\texttt{>} \quad \Gamma \vdash e_2 : \texttt{i32}}{\Gamma \vdash_l e_1\texttt{[}e_2\texttt{]} : \tau}$$

$$\frac{\texttt{struct}\ S\ \texttt{\{}x_1 : \tau_1, \ldots, x_n : \tau_n\texttt{\}} \in \Gamma \quad \pi \text{ une permutation} \quad \forall i.\ \Gamma \vdash e_i : \tau_i}{\Gamma \vdash S\texttt{\{}x_{\pi(1)} : e_{\pi(1)}, \ldots, x_{\pi(n)} : e_{\pi(n)}\texttt{\}} : S}$$

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

$$\frac{f(\tau'_1, \ldots, \tau'_n) \to \tau \in \Gamma \quad \forall i.\ \Gamma \vdash e_i : \tau_i \quad \tau_i \leq \tau'_i}{\Gamma \vdash f(e_1, \ldots, e_n) : \tau} \qquad \frac{}{\Gamma \vdash \texttt{print!(}s\texttt{)} : \texttt{()}}$$

$$\frac{\Gamma \vdash e : \texttt{bool} \quad \Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \texttt{if}\ e\ e_1\ \texttt{else}\ e_2 : \tau} \qquad \frac{\Gamma \vdash e : \texttt{bool} \quad \Gamma \vdash e_1 : \texttt{()}}{\Gamma \vdash \texttt{while}\ e\ e_1 : \texttt{()}}$$

$$\frac{}{\Gamma \vdash \texttt{return} : \texttt{()}} \qquad \frac{\Gamma \vdash e : \tau_r}{\Gamma \vdash \texttt{return}\ e : \texttt{()}}$$

$$\frac{}{\Gamma \vdash \texttt{\{\}} : \texttt{()}} \qquad \frac{\Gamma \vdash e : \tau}{\Gamma \vdash \texttt{\{}\ e\ \texttt{\}} : \tau} \qquad \frac{\Gamma \vdash \texttt{\{}\ b\ \texttt{\}} : \tau}{\Gamma \vdash \texttt{\{}\ \texttt{;}\ b\ \texttt{\}} : \tau}$$

$$\frac{\Gamma \vdash e : \tau \quad \Gamma, m\, x : \tau \vdash \texttt{\{}\ b\ \texttt{\}} : \tau'}{\Gamma \vdash \texttt{\{ let}\ m\ x = e\texttt{;}\ b\ \texttt{\}} : \tau'} \qquad \frac{e \neq \texttt{let} \quad \Gamma \vdash e : \tau \quad \Gamma \vdash \texttt{\{}\ b\ \texttt{\}} : \tau'}{\Gamma \vdash \texttt{\{}\ e\texttt{;}\ b\ \texttt{\}} : \tau'}$$

Dans ces dernières règles, b désigne le contenu d'un bloc, c'est-à-dire une séquence d'expressions séparées par des points-virgules, éventuellement terminée par une dernière expression non suivie d'un point-virgule.

On ajoute également trois règles d'*auto-déréférencement*, qui permettent respectivement d'écrire $e\texttt{.}x$ au lieu de $(\texttt{*}e)\texttt{.}x$, $e\texttt{[}e_2\texttt{]}$ au lieu de $(\texttt{*}e)\texttt{[}e_2\texttt{]}$ et $e\texttt{.len()}$ au lieu de $(\texttt{*}e)\texttt{.len()}$. Ces règles sont les suivantes :

$$\frac{\Gamma \vdash e : \texttt{\&}m\, \texttt{Vec<}\tau\texttt{>}}{\Gamma \vdash e\texttt{.len()} : \texttt{i32}} \qquad \frac{\Gamma \vdash e : \texttt{\&}m\, \texttt{Vec<}\tau\texttt{>} \quad \Gamma \vdash e_2 : \texttt{i32}}{\Gamma \vdash_l e\texttt{[}e_2\texttt{]} : \tau}$$

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

Enfin, les règles de mutabilité sont les suivantes :

$$\frac{\texttt{mut}\ x : \tau \in \Gamma}{\Gamma \vdash_{\texttt{mut}} x} \qquad \frac{\Gamma \vdash_{\texttt{mut}} e}{\Gamma \vdash_{\texttt{mut}} e\texttt{[}e_2\texttt{]}} \qquad \frac{\Gamma \vdash_{\texttt{mut}} e}{\Gamma \vdash_{\texttt{mut}} e\texttt{.}x} \qquad \frac{\Gamma \vdash e : \texttt{\&mut}\, \tau}{\Gamma \vdash_{\texttt{mut}} \texttt{*}e}$$

Dans ces dernières règles, on suppose que l'auto-déréférencement a déjà été appliqué, c'est-à-dire que e a été remplacé par $\texttt{*}e$ dans $e\texttt{.}x$ ou $e\texttt{[}e_2\texttt{]}$ si e est de type $\texttt{\&}m\, \tau$.

2.3 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.

Déclaration de structure. La validité d'une déclaration de structure revient essentiellement à vérifier la validité des types de ses champs :

$$\frac{\forall i.\ \Gamma, \texttt{struct}\ S\ \texttt{\{}x_1 : \tau_1, \ldots, x_n : \tau_n\texttt{\}} \vdash \tau_i\ \mathsf{bf} \quad \tau_i \text{ ne contient pas d'emprunt}}{\Gamma \vdash \texttt{struct}\ S\ \texttt{\{}x_1 : \tau_1, \ldots x_n : \tau_n\texttt{\}} \Rightarrow \{\texttt{struct}\ S\ \texttt{\{}x_1 : \tau_1, \ldots x_n : \tau_n\texttt{\}}\} \cup \Gamma}$$

On vérifiera d'autre part que les types τ_i des champs ne font référence à la structure S elle-même que sous un type `Vec`. On vérifiera enfin l'unicité des identificateurs $x_1, \ldots, x_n$.

Déclaration de fonction. Lorsque le type de retour τ_r d'une fonction est omis dans la syntaxe, il s'agit du type `()`.

$$\frac{\begin{array}{c}\Gamma \vdash \tau_r\ \mathsf{bf} \quad \tau_r \text{ ne contient pas d'emprunt} \quad \forall i.\ \Gamma \vdash \tau_i\ \mathsf{bf} \\ \{f(\tau_1, \ldots, \tau_n) \to \tau_r,\ x_1 : \tau_1, \ldots, x_n : \tau_n\} \cup \Gamma \vdash \texttt{\{}b\texttt{\}} : \tau \\ \tau = \texttt{()} \text{ et l'évaluation de } b \text{ aboutit toujours à un } \texttt{return}\text{, ou bien } \tau = \tau_r\end{array}}{\Gamma \vdash \texttt{fn}\ f(x_1 : \tau_1, \ldots, x_n : \tau_n) \to \tau_r\ \texttt{\{}b\texttt{\}} \Rightarrow \{f(\tau_1, \ldots, \tau_n) \to \tau_r\} \cup \Gamma}$$

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. On vérifiera par ailleurs l'unicité des identificateurs $x_1, \ldots, x_n$.

Fichier. 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 :

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

On vérifiera par ailleurs l'unicité des identificateurs de structures sur l'ensemble du fichier et l'unicité des identificateurs de fonctions sur l'ensemble du fichier.

Point d'entrée. Enfin, on vérifiera la présence d'une fonction `main` avec le profil `fn main()`.

2.4 Typage des ressources

Le principal intérêt du langage Rust, par rapport à un langage comme C, se situe dans les garanties fortes que son typage apporte dans la gestion des ressources, en particulier la mémoire. Par exemple, le typage de Rust garantit qu'on ne cherchera jamais à libérer deux fois un même bloc de mémoire alloué sur le tas (double `free`) ou encore qu'on ne suivra jamais une référence fantôme (une référence vers un bloc mémoire qui a été libéré). Cette section décrit ces vérifications supplémentaires. Il s'agit de vérifications *statiques*, c'est-à-dire effectuées pendant la compilation. On suggère de les effectuer comme une seconde phase de typage, une fois toutes les vérifications décrites dans les sections précédentes effectuées.

Les valeurs sont séparées en deux familles, selon leur type. Les valeurs de type `()`, `i32`, `bool` ou `&`τ sont dites *duplicables* (*copy* en anglais). Les valeurs de type `&mut`τ, structure ou `Vec<`τ`>`

sont dites *déplaçables* (*move* en anglais). Dans la suite, on appelle *ressource* toute valeur de cette seconde catégorie.

Toute ressource a un unique propriétaire, qui est une variable. Lorsque la variable propriétaire atteint la fin de sa portée, la ressource est libérée et n'est plus accessible. La portée d'une variable est ainsi définie : la portée d'une variable introduite par `let` s'étend jusqu'à la fin du bloc où elle est introduite ; la portée d'un paramètre formel s'étend jusqu'à la fin de la fonction. À la fin d'un bloc, les ressources sont libérées dans l'ordre inverse de la déclaration de leurs propriétaires.

Le propriétaire d'une ressource v peut changer avec le temps, lors
- d'une affectation de la forme $l = v$;
- d'une affectation à une variable `let` x `=` v`;` ;
- d'une affectation implicite dans la construction d'un tableau `vec![`$\ldots, v, \ldots$`]` ou d'une structure S `{`$\ldots,\ c : v, \ldots$`}` ;
- d'un passage comme paramètre effectif d'un appel $f(\ldots, v, \ldots)$.

Considérons les deux programmes suivants :

```
let v = vec![1,2,3];
let w = v;
let a = w[0];
```

```
let v = vec![1,2,3];
let w = v;
let a = v[0];
```

Celui de gauche est accepté car on accède à la ressource (le tableau) par son nouveau propriétaire `w`, mais celui de droite est refusé car on tente d'accéder à la ressource par son ancien propriétaire `v`.

Pour éviter les changements intempestifs de propriétaires en pratique, Rust introduit la notion de *référence*. Une référence est un pointeur vers une valeur de type τ et son type est `&`$m\,\tau$. Le booléen m indique si la référence est immuable (type `&`τ) ou au contraire mutable (type `&mut` τ). La création d'une référence vers une ressource v de type τ, avec la construction `&`$m\,v$ dite d'« emprunt », s'accompagne de plusieurs contraintes :
- la portée d'une référence ne doit pas excéder celle du propriétaire de la ressource ;
- il peut exister plusieurs références immuables (`&`τ) vers une même ressource, à un instant donné, mais au plus une référence mutable (`&mut` τ).

Parmi les tests fournis, le répertoire `typing2/` contient les tests spécifiques au typage des ressources (tests positifs dans `typing2/good/` et tests négatifs dans `typing2/bad/`).

2.5 Indications

Tests. En cas de doute concernant un point de sémantique, vous pouvez utiliser le compilateur `rustc` 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 Restrictions/différences par rapport à Rust

Si tout programme Petit Rust est un programme Rust correct, le langage Petit Rust souffre néanmoins de quelques restrictions par rapport à Rust. En particulier,
- Petit Rust possède moins de mots clés que Rust.

— les indices pour accéder au type `Vec` dans Petit Rust sont de type `i32`, alors qu'ils sont de type `usize` en Rust ; de même pour le résultat de la méthode `len`.

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

4 Production de code

Pour un fichier correctement typé, la dernière phase du compilateur consiste à produire du code x86-64 pour son exécution. 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.

4.1 Sémantique

Le mode de passage de Petit Rust est *par valeur*, quel que soit le type. En particulier, les structures sont affectées, passées en argument et renvoyées comme résultat *par valeur*, ce qui nécessite de copier tout un bloc d'octets. La section suivante précise ce qu'est une valeur.

La plupart des constructions (arithmétique, logique, conditionnelle, boucle, fonction) sont identiques à celles que l'on trouve dans d'autres langages, notamment C, et compilées sans surprise.

Un bloc peut être terminé par une dernière expression non suivie d'un point-virgule et a dans ce cas la valeur de cette expression. Cette dernière expression peut être une conditionnelle. Le corps d'une fonction est un bloc et, en l'absence de `return`, la valeur renvoyée par la fonction est la valeur de ce bloc.

La construction `&`$m\,e$ désigne l'adresse où se situe en mémoire la valeur de e. La valeur de m ne joue pas de rôle ici (mais seulement au typage). La construction `*`e donne la valeur qui se trouve en mémoire à l'adresse désignée par e.

4.2 Représentation des valeurs

Une des difficultés de ce projet est que 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.

On suggère d'adopter, au moins dans un premier temps, la représentation simple et uniforme suivante.

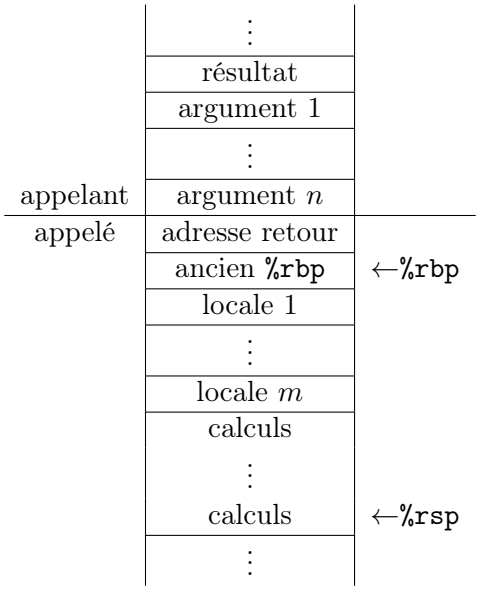
— Les valeurs de type `()`, `i32`, `bool` et `&`τ sont toutes représentées sur 64 bits, de la manière suivante :
  — une valeur de type `()` est quelconque (il n'est pas possible de comparer des valeurs de type `()` dans Petit Rust) ;
  — une valeur de type `i32` est un entier 32 bits signés stocké sur 64 bits (avec extension de signe) [1] ;
  — une valeur de type `bool` est un entier 64 bits qui faut 1 (pour `true`) ou 0 (pour `false`) ;
  — une valeur de type `&`τ est un pointeur vers une valeur de type τ.
— Une valeur de type S, où S est une structure, est la juxtaposition des valeurs de tous les champs de S, dans un ordre laissé à la liberté du compilateur.

1. Oui, c'est paradoxal, mais ne manipuler que des registres 64 bits simplifiera la production de code et il n'est pas incorrect de représenter un entier 32 bits par un entier 64 bits. On pourra supposer l'absence de débordement arithmétique dans les tests fournis.

— Une valeur de type `Vec<`τ`>` est représentée comme une structure possédant deux champs, le premier étant le nombre n d'éléments et le second étant un pointeur vers une zone mémoire allouée sur le tas de $n \times s$ octets, où s est la taille de la représentation d'une valeur de type τ.
Concernant ce dernier type, il est important de comprendre que la copie d'une valeur de type `Vec<`τ`>` n'implique que la copie des 16 octets contenant le nombre d'éléments et le pointeur, mais pas la copie de la zone du tas contenant les éléments.

4.3 Schéma de compilation

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 x86-64. La fonction prédéfinie `print!` pourra être réalisée par un appel à la fonction de bibliothèque `printf` (avec les bons arguments).

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

1. calculer la représentation (emplacement des champs et taille totale) de toutes les structures ;
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 x86-64 de la fonction f ;
3. ajouter le code de `print!` et des fonctions auxiliaires, le cas échéant ;
4. allouer les chaînes dans le segment de données.

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 `print!`, 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 `print!`.

5 Travail demandé (pour le mercredi 10 janvier 18h)

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 `prustc`. 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.

Votre compilateur doit accepter sur sa ligne de commande exactement un fichier Petit Rust (portant l'extension `.rs`) et éventuellement une option parmi `--parse-only`, `--type-only` et `--no-asm`. Ces options indiquent respectivement de stopper la compilation après l'analyse syntaxique (section 1), le typage simple (jusqu'à 2.3) ou juste avant la production de code (et donc avec le typage des ressources).

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.rs", 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.

Si le fichier d'entrée est conforme à la syntaxe et au typage décrits dans ce document, votre compilateur doit produire du code x86-64 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.rs`, le code x86-64 doit être produit dans le fichier `file.s` (même nom que le fichier source mais suffixe `.s` au lieu de `.rs`). Ce fichier x86-64 doit pouvoir être compilé et exécuté avec les commandes

```
gcc file.s -o file
./file
```

Le résultat affiché sur la sortie standard doit être identique à celui obtenu avec les commandes

```
rustc file.rs
./file
```