École Normale Supérieure 2020–2021

projet de compilation

Petit Julia

version 2 — 19 octobre 2020

L'objectif de ce projet est de réaliser un compilateur pour un fragment de Julia, appelé Petit Julia par la suite, produisant du code x86-64. Il s'agit d'un fragment relativement petit du langage Julia, avec parfois même quelques petites incompatibilités. Le présent sujet décrit la syntaxe et le typage de Petit Julia, 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)
(...)	parenthésage
... | ...	alternative

Attention à ne pas confondre « * » et « $^+$ » avec « `*` » et « `+` » qui sont des symboles du langage Julia. De même, attention à ne pas confondre les parenthèses avec les terminaux `(` et `)`.

1.1 Analyse lexicale

Espaces, tabulations et retours chariot constituent les blancs. Les commentaires commencent par `#` et s'étendent jusqu'à la fin de la ligne. Les identificateurs obéissent à l'expression régulière $\langle ident \rangle$ suivante :

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

Les identificateurs suivants sont des mots clés :

```
else      elseif  end      false   for
function  if      mutable  return  struct
true      while
```

Les constantes obéissent aux expressions régulières ⟨*entier*⟩ et ⟨*chaîne*⟩ suivantes :

⟨*entier*⟩ ::= ⟨*chiffre*⟩$^+$
⟨*car*⟩ ::= tout caractère de code ASCII compris entre 32 et 126 (inclus), autre que \ et "
| \\ | \" | \n | \t
⟨*chaîne*⟩ ::= " ⟨*car*⟩* "

Les constantes entières ne doivent pas dépasser 2^{63}. Enfin, les expressions régulières suivantes définissent des lexèmes impliqués dans la syntaxe de l'arithmétique et des appels de fonctions :

⟨*entier-ident*⟩ ::= ⟨*entier*⟩⟨*ident*⟩
⟨*ident-parg*⟩ ::= ⟨*ident*⟩(
⟨*entier-parg*⟩ ::= ⟨*entier*⟩(
⟨*pard-ident*⟩ ::=)⟨*ident*⟩

Dans chacune de ces expressions régulières, il n'y a pas de blanc entre les deux composantes.

Point-virgule automatique. Pour épargner au programmeur la peine d'écrire des points-virgules, l'analyseur lexical de Petit Julia insère automatiquement un point-virgule lorsqu'il rencontre un retour chariot et que le lexème précédemment émis faisait partie de l'ensemble suivant :

⟨*ident*⟩ | ⟨*entier*⟩ | ⟨*entier-ident*⟩ | ⟨*pard-ident*⟩ | ⟨*chaîne*⟩ | `true` | `false` | `return` | `)` | `end`

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 diverses constructions sont données par la table suivante, de la plus faible à la plus forte précédence :

opérateur ou construction	**associativité**
`return`	—
`=`	droite
`||`	gauche
`&&`	gauche
`==`, `!=`, `>`, `>=`, `<`, `<=`	gauche
`+`, `-`	gauche
`*`, `%`	gauche
`-` (unaire), `!`	—
`^`	droite
`.`	gauche

À cette grammaire s'ajoute la contrainte que le lexème `else` ne doit pas être suivi du lexème `if` (il faut utiliser `elseif`).

$$
\begin{array}{lcl}
\langle \textit{fichier} \rangle & ::= & \langle \textit{decl} \rangle^{\star}\ \texttt{EOF} \\
\langle \textit{decl} \rangle & ::= & \langle \textit{structure} \rangle \mid \langle \textit{fonction} \rangle \mid \langle \textit{expr} \rangle\ \texttt{;} \\
\langle \textit{structure} \rangle & ::= & \texttt{mutable}?\ \texttt{struct}\ \langle \textit{ident} \rangle\ (\ \langle \textit{param} \rangle ?\ )^{\star}_{\texttt{;}}\ \texttt{end ;} \\
\langle \textit{fonction} \rangle & ::= & \texttt{function}\ \langle \textit{ident-parg} \rangle\ \langle \textit{param} \rangle^{\star}_{\texttt{,}}\ \texttt{)}\ (\ \texttt{::}\ \langle \textit{ident} \rangle\ )?\ \langle \textit{bloc} \rangle\ \texttt{end ;} \\
\langle \textit{param} \rangle & ::= & \langle \textit{ident} \rangle\ (\ \texttt{::}\ \langle \textit{ident} \rangle\ )? \\
\langle \textit{expr} \rangle & ::= & \langle \textit{entier} \rangle \mid \langle \textit{chaîne} \rangle \mid \texttt{true} \mid \texttt{false} \\
& \mid & \langle \textit{entier-ident} \rangle \\
& \mid & \langle \textit{entier-parg} \rangle\ \langle \textit{bloc}_1 \rangle\ \texttt{)} \\
& \mid & \texttt{(}\ \langle \textit{bloc}_1 \rangle\ \texttt{)} \\
& \mid & \texttt{(}\ \langle \textit{expr} \rangle\ \langle \textit{pard-ident} \rangle \\
& \mid & \langle \textit{ident-parg} \rangle\ \langle \textit{expr} \rangle^{\star}_{\texttt{,}}\ \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 & \langle \textit{lvalue} \rangle \\
& \mid & \langle \textit{lvalue} \rangle\ \texttt{=}\ \langle \textit{expr} \rangle \\
& \mid & \texttt{return}\ \langle \textit{expr} \rangle ? \\
& \mid & \texttt{for}\ \langle \textit{ident} \rangle\ \texttt{=}\ \langle \textit{expr} \rangle\ \texttt{:}\ \langle \textit{expr} \rangle\ \langle \textit{bloc} \rangle\ \texttt{end} \\
& \mid & \texttt{while}\ \langle \textit{expr} \rangle\ \langle \textit{bloc} \rangle\ \texttt{end} \\
& \mid & \texttt{if}\ \langle \textit{expr} \rangle\ \langle \textit{bloc} \rangle\ \langle \textit{else} \rangle \\
\langle \textit{lvalue} \rangle & ::= & \langle \textit{ident} \rangle \mid \langle \textit{expr} \rangle\ \texttt{.}\ \langle \textit{ident} \rangle \\
\langle \textit{else} \rangle & ::= & \texttt{end} \\
& \mid & \texttt{else}\ \langle \textit{bloc} \rangle\ \texttt{end} \\
& \mid & \texttt{elseif}\ \langle \textit{expr} \rangle\ \langle \textit{bloc} \rangle\ \langle \textit{else} \rangle \\
\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{bloc} \rangle & ::= & (\ \langle \textit{expr} \rangle ?\ )^{\star}_{\texttt{;}} \\
\langle \textit{bloc}_1 \rangle & ::= & \langle \textit{expr} \rangle\ (\ \texttt{;}\ \langle \textit{bloc} \rangle\ )?
\end{array}
$$

FIGURE 1 – Grammaire des fichiers Petit Julia.

Lexèmes combinés. On remarque que `f(1)` est accepté (et désigne un appel de fonction) mais que `f (1)` provoque une erreur de syntaxe (espace entre le nom de la fonction et la parenthèse ouvrante).

Sucre syntaxique. On a les équivalences suivantes :
- l'expression `if` e b `end` équivaut à `if` e b `else end` ;
- l'expression `elseif` e b s équivaut à `else if` e b s `end` ;
- l'expression `println`$(e_1, e_2, \ldots, e_n)$ équivaut à `print`$(e_1, e_2, \ldots, e_n$`,"\n")` ;
- dans une déclaration de fonction `function` $f(x_1 :: \tau_1, \ldots, x_n :: \tau_n) :: \tau_{n+1}$, si un type τ_i est omis, il équivaut à `Any`.

2 Typage statique

Le langage Julia appartient clairement à la famille des langages *dynamiquement typés*, c'est-à-dire où les types sont associés aux *valeurs* et utilisés pendant l'exécution — par opposition aux langages *statiquement typés* où les types sont associés aux *expressions* et utilisés pendant la compilation. Cela étant, il reste possible de réaliser tout de même *un peu* de typage statique sur notre langage Petit Julia, afin de
- *refuser* à la compilation un programme qui contient une expression qui, si elle était exécutée, provoquerait une erreur à coup sûr [1] ;
- *optimiser* le code produit, notamment en appelant directement une fonction sans passer par le mécanisme de *dispatch* dynamique (voir la section suivante).

Il n'y a pas de limite a priori au typage statique que l'on peut faire sur les programmes Petit Julia, si ce n'est notre temps ou notre imagination. Ainsi, on pourrait faire de l'inférence de types, de la propagation de types, etc. Cependant, on suggère de rester raisonnable dans ce projet, en se limitant aux règles proposées ci-dessous.

Types. Un type est un identifiant, qui peut être `Any`, un type prédéfini parmi `Nothing`, `Int64`, `Bool`, `String`, ou enfin un type désignant une structure S introduite avec `struct` S. Un type est noté τ.

$$\tau \ ::= \ \texttt{Any} \mid \texttt{Nothing} \mid \texttt{Int64} \mid \texttt{Bool} \mid \texttt{String} \mid S$$

Le type `Any` est utilisé pour typer, à la compilation, toute variable ou expression dont le type n'est pas connu. Il n'aura pas de réalité à l'exécution. Une relation de compatibilité entre deux types, notée $\tau \equiv \tau'$, est définie de la manière suivante :

$$\tau \equiv \tau' \ := \ \tau = \texttt{Any} \vee \tau' = \texttt{Any} \vee \tau = \tau'$$

Une remarque importante est que l'on suppose ici *l'ensemble des différents types possibles* connu au moment de la compilation. En effet, on compile un programme complet, qui n'est ni une bibliothèque ni un programme utilisant une bibliothèque. Cela nous permet des vérifications supplémentaires. Ainsi, une expression comme `e.f` ne sera acceptée que s'il existe au moins un type ayant un champ `f` (et que l'expression `e` a un type

1. Il est important de préciser ici « si elle était exécutée ». En effet, on ne peut décider en général si un morceau de code va être exécuter. Sans cette précision, notre typage statique serait fortement limité.

compatible, bien évidemment). De la même façon, on connaît également l'ensemble des fonctions définies dans le programme.

Un contexte de typage Γ contient un ensemble de structures, de fonctions et de variables de la forme $x : \tau$. Une fonction de Γ est notée $f(\tau_1, \ldots, \tau_n) \Rightarrow \tau$, avec $n \geq 0$. On note $S(\tau_1, \ldots, \tau_n)$ une structure S déclarée avec n champs de types $\tau_1, \ldots, \tau_n$, dans cet ordre. On note $x : \tau \in S$ le fait que la structure S contient un champ x de type τ.

Dans un environnement Γ, une variable x est soit globale, soit locale. On note $\Gamma + e$ l'environnement obtenu en ajoutant dans Γ toutes les variables affectées dans l'expression e, comme autant de variables locales (éventuellement en cachant alors une variable globale de même nom, mais sans introduire de nouvelle variable locale si une variable locale de même nom existe déjà). Le contexte global contient initialement une variable `nothing` de type `Nothing`.

Typage d'une expression. On introduit le jugement $\Gamma \vdash e : \tau$ signifiant « dans le contexte Γ, l'expression e est bien typée de type τ ». Ce jugement est défini par les règles suivantes.

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

$$\frac{\Gamma \vdash e : \tau \quad x : \tau' \in S \quad \tau = \texttt{Any} \vee \tau = S}{\Gamma \vdash e\,.\,x : \tau'}$$

$$\frac{\Gamma \vdash e : \tau \quad \tau \in \{\texttt{Any}, \texttt{Int64}\}}{\Gamma \vdash \texttt{-}\ e : \texttt{Int64}} \qquad \frac{\Gamma \vdash e : \tau \quad \tau \in \{\texttt{Any}, \texttt{Bool}\}}{\Gamma \vdash \texttt{!}e : \texttt{Bool}}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2 \quad op \in \{\texttt{==}, \texttt{!=}\}}{\Gamma \vdash e_1\ op\ e_2 : \texttt{Bool}}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2 \quad \tau_1, \tau_2 \in \{\texttt{Any}, \texttt{Int64}\} \quad op \in \{\texttt{+}, \texttt{-}, \texttt{*}, \texttt{\%}, \texttt{\^{}}\}}{\Gamma \vdash e_1\ op\ e_2 : \texttt{Int64}}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2 \quad \tau_1, \tau_2 \in \{\texttt{Any}, \texttt{Int64}\}}{\Gamma \vdash \texttt{div}(e_1, e_2) : \texttt{Int64}}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2 \quad \tau_1, \tau_2 \in \{\texttt{Any}, \texttt{Int64}, \texttt{Bool}\} \quad op \in \{\texttt{<}, \texttt{<=}, \texttt{>}, \texttt{>=}\}}{\Gamma \vdash e_1\ op\ e_2 : \texttt{Bool}}$$

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

$$\frac{\forall i,\ \Gamma \vdash e_i : \tau_i}{\Gamma \vdash \texttt{print}(e_1, \ldots, e_n) : \texttt{Nothing}}$$

$$\frac{x : \tau \in \Gamma \quad \Gamma \vdash e : \tau' \quad \tau \neq \texttt{Any} \Rightarrow \tau = \tau'}{\Gamma \vdash x \texttt{ = } e : \tau'}$$

$$\frac{\Gamma \vdash e : \tau \quad x : \tau' \in S \quad \tau = \texttt{Any} \vee \tau = S \quad S \text{ mutable} \quad \Gamma \vdash e' : \tau'' \quad \tau' \equiv \tau''}{\Gamma \vdash e.x \texttt{ = } e' : \tau'}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \tau_1 \in \{\texttt{Any}, \texttt{Bool}\} \quad \Gamma \vdash e_2 : \tau \quad \Gamma \vdash e_3 : \tau}{\Gamma \vdash \texttt{if}\ e_1\ e_2\ \texttt{else}\ e_3\ \texttt{end} : \tau}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \tau_1 \in \{\texttt{Any}, \texttt{Bool}\} \quad \Gamma \vdash e_2 : \tau_2 \quad \Gamma \vdash e_3 : \tau_3 \quad \tau_2 \neq \tau_3}{\Gamma \vdash \texttt{if}\ e_1\ e_2\ \texttt{else}\ e_3\ \texttt{end} : \texttt{Any}}$$

$$\frac{}{\Gamma \vdash \texttt{return} : \texttt{Any}} \qquad \frac{\Gamma \vdash e : \tau}{\Gamma \vdash \texttt{return}\ e : \texttt{Any}} \quad \text{V2}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \tau_1 \in \{\texttt{Any}, \texttt{Bool}\} \quad \Gamma + e_2 \vdash e_2 : \tau_2}{\Gamma \vdash \texttt{while}\ e_1\ e_2\ \texttt{end} : \texttt{Nothing}}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2 \quad \tau_1, \tau_2 \in \{\texttt{Any}, \texttt{Int64}\} \quad \Gamma + e_3 + \{x : \texttt{Int64}\} \vdash e_3 : \tau_3}{\Gamma \vdash \texttt{for}\ x = e_1 : e_2\ e_3\ \texttt{end} : \texttt{Nothing}}$$

$$\frac{}{\Gamma \vdash \langle \text{bloc vide} \rangle : \texttt{Nothing}} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1\ ;\ e_2 : \tau_2}$$

$$\frac{\forall i,\ \Gamma \vdash e_i : \tau_i' \quad \text{une seule } f(\tau_1, \ldots, \tau_n) \Rightarrow \tau \in \Gamma \text{ telle que } \tau_i' \equiv \tau_i}{\Gamma \vdash f(e_1, \ldots, e_n) : \tau}$$

$$\frac{\forall i,\ \Gamma \vdash e_i : \tau_i' \quad \text{plusieurs } f(\tau_1, \ldots, \tau_n) \Rightarrow \tau \in \Gamma \text{ telle que } \tau_i' \equiv \tau_i}{\Gamma \vdash f(e_1, \ldots, e_n) : \texttt{Any}}$$

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

Typage d'un fichier. Les déclarations de fonctions peuvent apparaître dans n'importe quel ordre. Les structures, en revanche, doivent être déclarées avant d'être utilisées. Il est suggéré de procéder en deux temps :

1. On parcourt les déclarations du fichier dans l'ordre et on construit un environnement global contenant les structures, les fonctions et les variables globales de la manière suivante :
   (a) Pour une déclaration de structure S de la forme
   $$\texttt{struct}\ S\ \ x_1 :: \tau_1; \ldots; x_n :: \tau_n\ \texttt{end}$$
   on vérifie qu'il n'y a pas déjà une structure de nom S, que les x_i sont deux à deux distincts et que tous les types τ_i sont bien formés. Un type est bien formé s'il est prédéfini, déclaré précédemment ou bien égal à S. On vérifie en outre que les noms de champs sont uniques sur l'ensemble du fichier.
   (b) Pour une déclaration de fonction f de la forme
   $$\texttt{function}\ f(x_1 :: \tau_1, \ldots, x_n :: \tau_n)\ :: \tau\ b\ \texttt{end}$$
   on vérifie que $f \notin \{\texttt{div}, \texttt{print}, \texttt{println}\}$, que les x_i sont deux à deux distincts et que tous les types τ_i et τ sont bien formés. Mais on ne cherche pas à typer le corps b de la fonction.
   (c) Pour une expression globale e, on ajoute à l'environnement global toutes les variables qui sont affectées quelque part dans l'expression e.
2. On parcourt une nouvelle fois les déclarations du fichier dans l'ordre et on procède cette fois aux vérifications suivantes :

(a) Pour une déclaration de fonction de la forme

$$\texttt{function}\ f(x_1 :: \tau_1, \ldots, x_n :: \tau_n)\ :: \tau\ b\ \texttt{end}$$

on construit un nouvel environnement Γ en ajoutant à l'environnement global toutes les variables $x_i : \tau_i$ et toutes les variables affectées à l'intérieur de b, comme autant de variables locales. Puis on vérifie que le bloc b est bien typé dans Γ, d'un type compatible avec τ. On vérifie par ailleurs que toute instruction `return` dans b renvoie bien un résultat d'un type compatible avec τ.

(b) Pour une expression globale e, on vérifie qu'elle est bien typée dans l'environnement global.

Anticipation. Dans la phase suivante (production de code), certaines informations provenant du typage seront nécessaires, telles que par exemple la portée, la détermination de la fonction appelée, etc. 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 simple mais correct. En particulier, on ne cherche pas à faire d'allocation de registres mais on se contente d'utiliser la pile pour stocker les éventuels calculs intermédiaires. Bien entendu, il est possible d'utiliser localement les registres. On ne cherche pas à libérer la mémoire.

3.1 Représentation des valeurs

Il est suggéré d'adopter une représentation simple et uniforme, où toute valeur est un pointeur vers un bloc alloué sur le tas. Ce bloc est composé de $n + 1$ mots de 64 bits, de la façon suivante :

classe	champ 1	...	champ n

Le premier mot désigne le type de la valeur (`Nothing`, `Int64`, `Bool`, `String` ou S). Pour une valeur de type `Nothing`, il n'y a pas de champ ($n = 0$). Pour une valeur de type `Int64` ou `Bool`, il y a un seul champ ($n = 1$) contenant la valeur. La valeur d'un booléen est un entier valant 0 ou 1. Pour une valeur de type `String`, il y a un seul champ ($n = 1$) contenant un pointeur vers la chaîne de caractères. Enfin, pour une valeur d'un type structure S, on trouve les valeurs des n champs de la structure.

On peut faire une exception pour la valeur `nothing`, en la représentant directement par l'entier 0.

3.2 Vérifications dynamiques

Lorsqu'une variable est utilisée, il faut vérifier qu'elle a bien reçue une valeur auparavant. Si ce n'est pas le cas, le programme doit échouer (par exemple avec un message

comme `x not defined`). Une façon simple de faire cette vérification consiste à initialiser les variables avec une valeur distincte de toute valeur légale (par exemple un entier impair). On note que le test dynamique est inutile dans certains cas, comme pour un paramètre de fonction ou la variable d'une boucle `for`. On pourra chercher à faire cette optimisation.

Lorsqu'on affecte un champ de structure déclaré de type $\tau \neq$ `Any` avec une valeur v, ou lorsqu'une fonction déclarée avec un type de retour $\tau \neq$ `Any` renvoie une valeur v, le programme doit échouer si la valeur v n'est pas du type τ. De même, le programme doit échouer si les deux bornes d'une boucle `for` ne sont pas des entiers. On note que le test dynamique est inutile dans certains cas, comme par exemple `function f()::Int64 42 end` ou encore `for n = 1 : 2 ... end`, mais on ne cherchera pas forcément à faire d'optimisation ici.

3.3 Dispatch multiple

Dans un appel de fonction $f(e_1, \ldots, e_n)$, il n'est pas toujours possible de déterminer statiquement quelle est la fonction qui doit être appelée. En effet, il peut exister plusieurs fonctions f prenant n arguments avec des types compatibles avec les types trouvés pour $e_1, \ldots, e_n$. Voici un exemple simple :

```
function foo(x::Int64) 42 end
function foo(x::Bool ) 43 end
function bar(x) foo(x) end
```

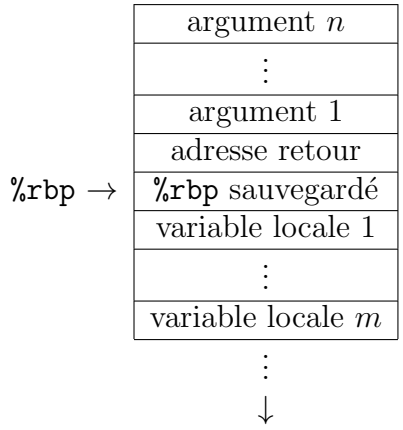
Lorsque l'on compile l'appel à `foo` contenu dans la fonction `bar`, on ne peut pas choisir entre les deux versions de la fonction `foo`. On ne pourra le faire qu'à l'exécution, en examinant le type *dynamique* de `x`. Pour faire cela, il est suggéré de produire à la compilation un code qui ressemble à ceci :

```
function foo_1(x::Int64) 42 end
function foo_2(x::Bool ) 43 end
function foo(x)
  if     typeof(x) == Int64 foo_1(x)
  elseif typeof(x) == Bool  foo_2(x)
  else                      error    end end
function bar(x) foo(x) end
```

D'autres exemples impliquant un tel *dispatch* sont contenus dans les tests fournis. On note qu'il peut y avoir une ambiguïté, *i.e.*, plusieurs fonctions peuvent être appelées. Dans ce cas, le programme doit échouer dynamiquement.

3.4 Schéma de compilation

L'appelant place tous les arguments sur la pile avant de faire `call`. L'appelé sauvegarde `%rbp` sur la pile et le positionne à cet endroit. Il alloue éventuellement de la place sur la pile pour ses variables locales. La valeur de retour est placée dans le registre `%rax`. Au retour, l'appelant se charge de dépiler les arguments. On a donc un tableau d'activation de la forme suivante :



La valeur d'une expression est compilée en utilisant la pile si besoin et en plaçant sa valeur finale dans `%rdi` ou en sommet de pile (au choix).

Le code assembleur produit au final doit ressembler à quelque chose comme

```
        .text
        .globl main
main:   évaluation des expressions du programme
        xorq %rax, %rax
        ret
...     fonctions du programme
...     fonctions écrites en assembleur, le cas échéant
        .data
...     chaînes de caractères
...     variables globales
```

4 Travail demandé

Le projet est à faire seul ou en binôme. Il doit être remis par email à `filliatr@lri.fr`, sous la forme d'une archive compressée (avec `tar` ou `zip`), appelée `vos_noms{.tgz,.zip}` qui doit contenir un répertoire appelé `vos_noms` (exemple : `dupont-durand`). 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 `pjuliac`. La commande `make clean` doit effacer tous les fichiers que `make` a engendrés et ne laisser dans le répertoire que les fichiers sources. Bien entendu, la compilation du projet peut être réalisée avec d'autres outils que `make` (par exemple `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, les difficultés rencontrées, les éléments non réalisés et plus généralement toute différence par rapport à ce qui a été demandé. Ce rapport pourra être fourni dans un format ASCII, Markdown ou PDF.

Partie 1 (à rendre pour le dimanche 6 décembre 18:00). Dans cette première partie du projet, le compilateur `pjuliac` doit accepter sur sa ligne de commande une

option éventuelle (parmi `--parse-only` et `--type-only`) et exactement un fichier Petit Julia portant l'extension `.jl`. 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 "test.jl", line 4, characters 5-6:
syntax error
```

L'anglicisme de la première ligne est nécessaire pour que la fonction `next-error` d'Emacs [2] 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 (`exit 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 "test.jl", line 4, characters 5-6:
this expression has type Int64 but is expected to have type String
```

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 (`exit 1`). Si en revanche il n'y a pas d'erreur de typage, le compilateur doit terminer avec le code de sortie 0. En cas d'erreur du compilateur lui-même, le compilateur doit terminer avec le code de sortie 2 (`exit 2`). L'option `--type-only` indique de stopper la compilation après l'analyse sémantique (typage). Elle est donc sans effet dans cette première partie, mais elle doit être honorée néanmoins.

Partie 2 (à rendre pour le dimanche 17 janvier 18:00). Si le fichier d'entrée est conforme à la syntaxe et au typage de ce sujet, 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.jl`, le code assembleur doit être produit dans le fichier `file.s` (même nom que le fichier source mais suffixe `.s` au lieu de `.jl`). Ce fichier x86-64 doit pouvoir être exécuté avec la commande

```
gcc -no-pie file.s -o file
./file
```

Le résultat affiché sur la sortie standard doit être identique à celui donné par l'interprétation du fichier Julia `file.jl` avec

```
julia file.jl
```

2. Le correcteur est susceptible d'utiliser cet éditeur.

Remarque importante. La correction du projet est réalisée en partie automatiquement, à l'aide d'un jeu de petits programmes réalisant des affichages avec l'instruction `print` ou `println`, 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 `print` et `println`.

Conseils. Il est fortement conseillé de procéder construction par construction que ce soit pour le typage ou pour la production de code, dans cet ordre : affichage, arithmétique, variables globales, instructions (`if`, `for`, `while`), fonctions, structures.