École Normale Supérieure 2019–2020

projet de compilation

Petit Go

version 1.1 — 9 octobre 2019

L'objectif de ce projet est de réaliser un compilateur pour un fragment de Go, appelé Petit Go par la suite, produisant du code x86-64. Il s'agit d'un fragment relativement petit du langage Go, avec parfois même quelques petites incompatibilités. Néanmoins, votre compilateur ne sera jamais testé sur des programmes incorrects au sens de Petit Go mais corrects au sens de Go. Le présent sujet décrit la syntaxe et le typage de Petit Go, 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 Go. 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 peuvent prendre deux formes :

— débutant par `/*` et s'étendant jusqu'à `*/` (mais non imbriqués) ;

— débutant par `//` et s'étendant jusqu'à la fin de la ligne.

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

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

Les identificateurs suivants sont des mots clés :

```
else    false  for      func    if
import  nil    package  return  struct
true    type   var
```

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

⟨*hexa*⟩ ::= `0`–`9` | `a`–`f` | `A`–`F`
⟨*entier*⟩ ::= ⟨*chiffre*⟩$^+$
| (`0x` | `0X`) ⟨*hexa*⟩$^+$
⟨*car*⟩ ::= tout caractère de code ASCII compris entre 32 et 126 (inclus), autre que `\` et `"`
| `\\` | `\"` | `\n` | `\t`
⟨*chaîne*⟩ ::= `"` ⟨*car*⟩* `"`

Les constantes entières doivent être comprises entre -2^{63} et $2^{63}-1$.

Point-virgule automatique. Pour épargner au programmeur la peine d'écrire des points-virgules à la fin des lignes qui contiennent des instructions, l'analyseur lexical de Petit Go 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*⟩ | ⟨*chaîne*⟩ | `true` | `false` | `nil` | `return` | `++` | `--` | `)` | `}`

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**
`||`	gauche
`&&`	gauche
`==`, `!=`, `>`, `>=`, `<`, `<=`	gauche
`+`, `-`	gauche
`*`, `/`, `%`	gauche
`-` (unaire), `*` (unaire), `&`, `!`	—
`.`	gauche

Sucre syntaxique. On a les équivalences suivantes :
— l'instruction `for` b équivaut à `for true` b (boucle infinie).
— l'instruction `for` i_1`;` e`;` i_2 b équivaut à { i_1`;` `for` e { b i_2 }}.
— l'instruction `if` e b équivaut à `if` e b `else` {}.
— l'instruction $x_1, \ldots, x_n$`:=`$e_1, \ldots, e_m$ équivaut à `var` $x_1, \ldots, x_n$ `=` $e_1, \ldots, e_m$.

$$
\begin{array}{lcl}
\langle \textit{fichier} \rangle & ::= & \texttt{package main ;} \ (\texttt{import "fmt" ;})? \ \langle \textit{decl} \rangle^\star \ \texttt{EOF} \\
\langle \textit{decl} \rangle & ::= & \langle \textit{structure} \rangle \mid \langle \textit{fonction} \rangle \\
\langle \textit{structure} \rangle & ::= & \texttt{type} \ \langle \textit{ident} \rangle \ \texttt{struct \{} \ (\langle \textit{vars} \rangle^+_{\texttt{;}} \ \texttt{;}?)? \ \texttt{\} ;} \\
\langle \textit{fonction} \rangle & ::= & \texttt{func} \ \langle \textit{ident} \rangle \ \texttt{(} \ (\langle \textit{vars} \rangle^\star_{\texttt{,}} \ \texttt{,}?)? \ \texttt{)} \ \langle \textit{type\_retour} \rangle? \ \langle \textit{bloc} \rangle \ \texttt{;} \\
\langle \textit{vars} \rangle & ::= & \langle \textit{ident} \rangle^+_{\texttt{,}} \ \langle \textit{type} \rangle \\
\langle \textit{type\_retour} \rangle & ::= & \langle \textit{type} \rangle \\
& \mid & \texttt{(} \ \langle \textit{type} \rangle^+_{\texttt{,}} \ \texttt{,}? \ \texttt{)} \\
\langle \textit{type} \rangle & ::= & \langle \textit{ident} \rangle \\
& \mid & \texttt{*} \ \langle \textit{type} \rangle \\
\langle \textit{expr} \rangle & ::= & \langle \textit{entier} \rangle \mid \langle \textit{chaîne} \rangle \mid \texttt{true} \mid \texttt{false} \mid \texttt{nil} \\
& \mid & \texttt{(} \ \langle \textit{expr} \rangle \ \texttt{)} \\
& \mid & \langle \textit{ident} \rangle \\
& \mid & \langle \textit{expr} \rangle \ \texttt{.} \ \langle \textit{ident} \rangle \\
& \mid & \langle \textit{ident} \rangle \ \texttt{(} \ \langle \textit{expr} \rangle^\star_{\texttt{,}} \ \texttt{)} \\
& \mid & \texttt{fmt . Print (} \ \langle \textit{expr} \rangle^\star_{\texttt{,}} \ \texttt{)} \\
& \mid & \texttt{!} \ \langle \textit{expr} \rangle \mid \texttt{-} \ \langle \textit{expr} \rangle \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 \\
\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 & ::= & \texttt{\{} \ (\langle \textit{instr} \rangle)^+_{\texttt{;}} \ \texttt{;}?)? \ \texttt{\}} \\
\langle \textit{instr} \rangle & ::= & \\
& \mid & \langle \textit{instr\_simple} \rangle \mid \langle \textit{bloc} \rangle \mid \langle \textit{instr\_if} \rangle \\
& \mid & \texttt{var} \ \langle \textit{ident} \rangle^+_{\texttt{,}} \ \langle \textit{type} \rangle? \ (\texttt{=} \ \langle \textit{expr} \rangle^+_{\texttt{,}})? \\
& \mid & \texttt{return} \ \langle \textit{expr} \rangle^\star_{\texttt{,}} \\
& \mid & \texttt{for} \ \langle \textit{bloc} \rangle \\
& \mid & \texttt{for} \ \langle \textit{expr} \rangle \ \langle \textit{bloc} \rangle \\
& \mid & \texttt{for} \ \langle \textit{instr\_simple} \rangle? \ \texttt{;} \ \langle \textit{expr} \rangle \ \texttt{;} \ \langle \textit{instr\_simple} \rangle? \ \langle \textit{bloc} \rangle \\
\langle \textit{instr\_simple} \rangle & ::= & \langle \textit{expr} \rangle \\
& \mid & \langle \textit{expr} \rangle \ (\texttt{++} \mid \texttt{--}) \\
& \mid & \langle \textit{expr} \rangle^+_{\texttt{,}} \ \texttt{=} \ \langle \textit{expr} \rangle^+_{\texttt{,}} \\
& \mid & \langle \textit{ident} \rangle^+_{\texttt{,}} \ \texttt{:=} \ \langle \textit{expr} \rangle^+_{\texttt{,}} \\
\langle \textit{instr\_if} \rangle & ::= & \texttt{if} \ \langle \textit{expr} \rangle \ \langle \textit{bloc} \rangle \ (\texttt{else} \ (\langle \textit{bloc} \rangle \mid \langle \textit{instr\_if} \rangle))?
\end{array}
$$

FIGURE 1 – Grammaire des fichiers Petit Go.

2 Typage statique

Une fois l'analyse syntaxique effectuée avec succès, on vérifie la conformité du fichier source. Dans tout ce qui suit, les types sont de la forme suivante :

$$\tau \ ::= \ \texttt{int} \mid \texttt{bool} \mid \texttt{string} \mid S \mid \texttt{*}\tau$$

où S désigne un nom de structure. 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 $f(\tau_1, \ldots, \tau_n) \Rightarrow \tau'_1, \ldots, \tau'_m$, avec $n \geq 0$ et $m \geq 0$.

Bonne formation d'un type. Le jugement $\Gamma \vdash \tau \ bf$ signifie « le type τ est bien formé dans l'environnement Γ ». Il est défini ainsi :

$$\frac{}{\Gamma \vdash \texttt{int} \ bf} \qquad \frac{}{\Gamma \vdash \texttt{bool} \ bf} \qquad \frac{}{\Gamma \vdash \texttt{string} \ bf} \qquad \frac{S \in \Gamma}{\Gamma \vdash S \ bf} \qquad \frac{\Gamma \vdash \tau \ bf}{\Gamma \vdash \texttt{*}\tau \ bf}$$

Champs d'une structure. On note $S\{\ x : \tau\ \}$ le fait que la structure S possède un champ x de type τ.

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 ». Le jugement $\Gamma \vdash_l e : \tau$ signifie de plus que e est une valeur gauche. Ces jugements sont définis par les règles suivantes.

$$\frac{c \text{ constante de type } \tau}{\Gamma \vdash c : \tau} \qquad \frac{\Gamma \vdash \tau \ bf}{\Gamma \vdash \texttt{nil} : \texttt{*}\tau} \qquad \frac{x : \tau \in \Gamma}{\Gamma \vdash_l x : \tau} \qquad \frac{\Gamma \vdash_l e : \tau \quad e \neq \_}{\Gamma \vdash e : \tau}$$

$$\frac{\Gamma \vdash e : S \quad S\{\ x : \tau\ \}}{\Gamma \vdash e.x : \tau} \qquad \frac{\Gamma \vdash e : \texttt{*}S \quad e \neq \texttt{nil} \quad S\{\ x : \tau\ \}}{\Gamma \vdash e.x : \tau}$$

$$\frac{\Gamma \vdash_l e : \tau' \quad \Gamma \vdash e.x : \tau}{\Gamma \vdash_l e.x : \tau} \qquad \frac{\Gamma \vdash e : \texttt{int}}{\Gamma \vdash \texttt{-}\, e : \texttt{int}} \qquad \frac{\Gamma \vdash e : \texttt{bool}}{\Gamma \vdash \texttt{!}e : \texttt{bool}}$$

$$\frac{\Gamma \vdash e : \texttt{*}\tau \quad e \neq \texttt{nil}}{\Gamma \vdash \texttt{*}e : \tau} \qquad \frac{\Gamma \vdash_l e : \tau}{\Gamma \vdash \texttt{\&}e : \texttt{*}\tau} \qquad \frac{S \in \Gamma}{\Gamma \vdash \texttt{new(}S\texttt{)} : \texttt{*}S}$$

$$\frac{\Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau \quad op \in \{\texttt{==},\ \texttt{!=}\} \quad e_1 \neq \texttt{nil} \vee e_2 \neq \texttt{nil}}{\Gamma \vdash e_1 \ op \ e_2 : \texttt{bool}}$$

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

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

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

$$\frac{\Gamma \vdash f(e_1, \ldots, e_n) \Rightarrow \tau_1}{\Gamma \vdash f(e_1, \ldots, e_n) : \tau_1}$$

Typage d'un appel. Le jugement $\Gamma \vdash f(e_1, \ldots, e_n) \Rightarrow \tau_1, \ldots, \tau_m$ signifie « dans le contexte Γ, l'appel de fonction $f(e_1, \ldots, e_n)$ est bien typé et renvoie m valeurs de types $\tau_1, \ldots, \tau_m$ ». Il est défini ainsi :

$$\frac{f(\tau_1, \ldots, \tau_n) \Rightarrow \tau'_1, \ldots, \tau'_m \in \Gamma \quad \forall i,\ \Gamma \vdash e_i : \tau_i}{\Gamma \vdash f(e_1, \ldots, e_n) \Rightarrow \tau'_1, \ldots, \tau'_m}$$

$$\frac{n \geq 2 \quad f(\tau_1, \ldots, \tau_n) \Rightarrow \tau'_1, \ldots, \tau'_m \in \Gamma \quad \Gamma \vdash g(e_1, \ldots, e_k) \Rightarrow \tau_1, \ldots, \tau_n}{\Gamma \vdash f(g(e_1, \ldots, e_k)) \Rightarrow \tau'_1, \ldots, \tau'_m}$$

Cette seconde règle permet de passer directement les n résultats d'une fonction g en arguments d'une fonction f.

Typage d'une instruction. Le jugement $\Gamma \vdash s$ signifie « dans le contexte Γ, instruction s est bien typée ». Il est défini par les règles suivantes.

$$\frac{\Gamma \vdash_l e : \texttt{int}}{\Gamma \vdash e\texttt{++}} \qquad \frac{\Gamma \vdash_l e : \texttt{int}}{\Gamma \vdash e\texttt{--}}$$

$$\frac{\forall i,\ \Gamma \vdash e_i : \tau_i}{\Gamma \vdash \texttt{fmt.Print(}e_1, \ldots, e_n\texttt{)}} \qquad \frac{n \geq 2 \quad \Gamma \vdash f(e_1, \ldots, e_k) \Rightarrow \tau_1, \ldots, \tau_n}{\Gamma \vdash \texttt{fmt.Print(}f(e_1, \ldots, e_k)\texttt{)}}$$

$$\frac{\forall i,\ \Gamma \vdash_l e_i : \tau_i \quad \forall i,\ \Gamma \vdash e'_i : \tau_i}{\Gamma \vdash e_1, \ldots, e_n\texttt{=}e'_1, \ldots, e'_n} \qquad \frac{\forall i,\ \Gamma \vdash_l e_i : \tau_i \quad \Gamma \vdash f(e'_1, \ldots, e'_m) \Rightarrow \tau_1, \ldots, \tau_n}{\Gamma \vdash e_1, \ldots, e_n\texttt{=}f(e'_1, \ldots, e'_m)}$$

$$\frac{\Gamma \vdash e : \texttt{bool} \quad \Gamma \vdash b_1 \quad \Gamma \vdash b_2}{\Gamma \vdash \texttt{if(}e\texttt{)}\ b_1\ \texttt{else}\ b_2} \qquad \frac{\Gamma \vdash e : \texttt{bool} \quad \Gamma \vdash b}{\Gamma \vdash \texttt{for}\ e\ b}$$

$$\frac{\forall i,\ \Gamma \vdash e_i : \tau_i}{\Gamma \vdash \texttt{return}\ e_1, \ldots, e_n} \qquad \frac{\Gamma \vdash f(e_1, \ldots, e_k) \Rightarrow \tau_1, \ldots, \tau_n}{\Gamma \vdash \texttt{return}\ f(e_1, \ldots, e_k)}$$

$$\frac{\Gamma \vdash \tau\ bf \quad \Gamma + x_1 : \tau, \ldots, x_n : \tau \vdash \{s_2; \ldots; s_m\}}{\Gamma \vdash \{\texttt{var}\ x_1, \ldots, x_n\ \tau; s_2; \ldots; s_m\}}$$

$$\frac{\Gamma \vdash \tau\ bf \quad \forall i,\ \Gamma \vdash e_i : \tau \quad \Gamma + x_1 : \tau, \ldots, x_n : \tau \vdash \{s_2; \ldots; s_n\}}{\Gamma \vdash \{\texttt{var}\ x_1, \ldots, x_n\ \tau = e_1, \ldots, e_n; s_2; \ldots; s_n\}}$$

$$\frac{\forall i,\ e_i \neq \texttt{nil} \quad \forall i,\ \Gamma \vdash e_i : \tau_i \quad \Gamma + x_1 : \tau_1, \ldots, x_n : \tau_n \vdash \{s_2; \ldots; s_m\}}{\Gamma \vdash \{\texttt{var}\ x_1, \ldots, x_n = e_1, \ldots, e_n; s_2; \ldots; s_m\}}$$

$$\frac{\Gamma \vdash \tau\ bf \quad \Gamma \vdash f(e_1, \ldots, e_k) \Rightarrow \tau^n \quad \Gamma + x_1 : \tau, \ldots, x_n : \tau \vdash \{s_2; \ldots; s_m\}}{\Gamma \vdash \{\texttt{var}\ x_1, \ldots, x_n\ \tau = f(e_1, \ldots, e_k); s_2; \ldots; s_m\}}$$

$$\frac{\Gamma \vdash f(e_1, \ldots, e_k) \Rightarrow \tau_1, \ldots, \tau_n \quad \Gamma + x_1 : \tau_1, \ldots, x_n : \tau_n \vdash \{s_2; \ldots; s_m\}}{\Gamma \vdash \{\texttt{var}\ x_1, \ldots, x_n = f(e_1, \ldots, e_k); s_2; \ldots; s_m\}}$$

$$\frac{}{\Gamma \vdash \{\}} \qquad \frac{\Gamma \vdash s_1 \quad \Gamma \vdash \{s_2; \ldots; s_n\}}{\Gamma \vdash \{s_1; \ldots; s_n\}}$$

Par ailleurs, toutes les variables introduites dans un *même* bloc doivent porter des noms différents. Une exception est faite pour la variable _.

Typage d'un fichier. Les déclarations d'un fichier peuvent apparaître dans n'importe quel ordre. En particulier, les fonctions et les structures sont mutuellement récursives. Il est suggéré de procéder en trois temps :

1. On ajoute dans l'environnement toutes les structures (mais pas leurs champs), en vérifiant l'unicité des noms de structures.
2. (a) On ajoute dans l'environnement toutes les fonctions, en vérifiant l'unicité des noms de fonctions. Pour une déclaration de fonction de la forme

$$\texttt{func}\ f(x_1 : \tau_1, \ldots, x_n : \tau_n)\ (\tau'_1, \ldots, \tau'_m)\ \{b\}$$

   on vérifie que les x_i sont deux à deux distincts et que tous les types τ_i et τ'_j sont bien formés.

   (b) On vérifie et on ajoute dans l'environnement tous les champs de structures. Pour une déclaration de structure S de la forme

$$\texttt{type}\ S\ \texttt{struct}\ \{\ x_1 : \tau_1, \ldots, x_n : \tau_n\ \}$$

   on vérifie que les x_i sont deux à deux distincts et que tous les types τ_i sont bien formés.
3. (a) Pour chaque déclaration de fonction de la forme

$$\texttt{func}\ f(x_1 : \tau_1, \ldots, x_n : \tau_n)\ (\tau'_1, \ldots, \tau'_m)\ \{b\}$$

   on construit un nouvel environnement Γ en ajoutant toutes les variables $x_i : \tau_i$ à l'environnement contenant les structures et les fonctions et on type le bloc b dans Γ, *i.e.*, on vérifie $\Gamma \vdash b$. On vérifie également
   — que toute instruction `return` dans b renvoie bien un résultat du type attendu $\tau'_1, \ldots, \tau'_m$;
   — si $m > 0$, que toute branche du flot d'exécution dans b aboutit bien à une instruction `return` ;
   — que toute variable locale introduite dans b, autre que _, est bien utilisée.

   (b) On vérifie qu'il n'y a pas de structure "récursive" c'est-à-dire de structure S possédant un champ de type (qui contient un champ de type, qui contient un champ de type, etc.) S.

Enfin, on vérifie qu'il existe une fonction `main` sans paramètres et sans type de retour et que le fichier contient `import "fmt"` si et seulement s'il y a au moins une instruction `fmt.Print`.

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

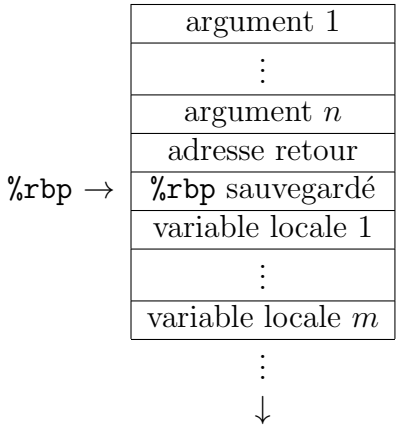
Types primitifs. On peut représenter un entier directement par un entier machine, en l'occurrence un entier 64 bits signé. Les booléens sont représentés par des entiers : 0 représente `false` et une valeur non nulle représente `true`. Une chaîne est représentée par un pointeur vers une chaîne allouée dans le segment de données (car, dans notre fragment, il n'est pas possible de construire de chaînes dynamiquement).

Structures. Une structure est représentée par un bloc mémoire alloué sur la pile ou sur le tas. Ce bloc contient les valeurs des champs de la structure, dans un ordre arbitraire. Le compilateur maintient donc une table donnant, pour chaque structure S et chaque champ de S, la position où trouver ce champ dans une valeur de type S.

La valeur `nil`. Elle est représentée par l'entier 0. En particulier, elle est différente de toute adresse d'une structure allouée.

3.2 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 peut être placée dans le registre `%rax` si cela est possible. 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.

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

```
        .text
        .globl main
main:   appel de la fonction main
        xorq %rax, %rax
        ret
  ...   fonctions Petit Go
  ...   fonctions écrites en assembleur, le cas échéant
        .data
  ...   chaînes de caractères
```

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 `tar` compressée (option "`z`" de `tar`), appelée *vos_noms*`.tgz` qui doit contenir un répertoire appelé *vos_noms* (exemple : `dupont-durand.tgz`). Dans ce répertoire doivent se trouver les *sources* du compilateur (inutile d'inclure les fichiers compilés). Quand on se place dans ce répertoire, la commande `make` doit créer le compilateur, qui sera appelé `pgoc`. 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 ou `dune` si le projet est réalisé en OCaml) et le `Makefile` se réduit alors à quelques lignes seulement pour appeler ces outils.

L'archive doit également contenir un court rapport expliquant les différents choix techniques qui ont été faits, 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 8 décembre 18:00). Dans cette première partie du projet, le compilateur `pgoc` doit accepter sur sa ligne de commande une option éventuelle (parmi `--parse-only` et `--type-only`) et exactement un fichier Petit Go portant l'extension `.go`. 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.go", line 4, characters 5-6:
syntax error
```

L'anglicisme de la première ligne est nécessaire pour que la fonction `next-error` d'Emacs [1] 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

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

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.go", line 4, characters 5-6:
this expression has type int but is expected to have type bool
```

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.

Partie 2 (à rendre pour le dimanche 12 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.go`, le code assembleur doit être produit dans le fichier `file.s` (même nom que le fichier source mais suffixe `.s` au lieu de `.go`). 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'exécution du fichier Go `file.go` avec

```
go run file.go
```

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 `fmt.Print`, 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 `fmt.Print`.

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 locales, fonctions, structures.