Compilation de NanoGo

Projet 2ème partie

L'objectif de ce projet est de réaliser un compilateur produisant du code x86-64 pour un fragment de Go[1], appelé NanoGo par la suite. 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 NanoGo mais corrects au sens de Go.

Le présent sujet décrit la syntaxe, le système de types, ainsi que la sémantique opérationnelle à grands pas de NanoGo. Il fixe également les contraintes du projet et la nature du travail demandé. En particulier, une contrainte importante est que vous devez coder votre projet en remplaçant uniquement les commentaires `TODO` dans l'archive qui est distribuée avec ce sujet.

1 Syntaxe

Le NanoGo contient des notions déjà vues dans le langage C (variable, structures, pointeur, fonction), mais beaucoup simplifiées. Un exemple de programme NanoGo est donné ci-dessous (fichier `tests/good/fact_rec.go`) :

```
package main
import "fmt"
// version recursive de la factorielle
func fact(n int) int {
  if n <= 1 {
    return 1;
  }
  return n * fact(n-1);
}

func main() {
  for n := 0; n <= 10; n++ {
    fmt.Print(fact(n));
    fmt.Print("\n")
  }
}
```

1. Go est un langage inspiré de C et Pascal `https://golang.org/doc/`

1.1 Analyse lexicale

Le lexique du NanoGo est formalisé dans l'annexe A.1. Vous l'utiliserez uniquement pour écrire vos tests, car l'analyseur lexical du NanoGo vous est donné dans le fichier `src/lexer.mll`. Il est compilé avec `ocamllex` pour générer le code OCaml de l'analyseur.

Ainsi, l'ensemble des constantes Σ contient les constantes booléennes `true` et `false` (dans l'ensemble $\Sigma_{\texttt{bool}}$), les constantes entières (dans l'ensemble $\Sigma_{\texttt{int}}$) et les constantes chaînes de caractères (dans l'ensemble $\Sigma_{\texttt{string}}$.

1.2 Analyse syntaxique

La syntaxe concrète des programmes NanoGo est donnée dans la figure 1 de l'annexe A.2. Vous utiliserez cette grammaire pour écrire les tests de votre compilateur.

L'analyseur syntaxique du NanoGo est codé dans le fichier `src/parser.mly`, compilé avec `menhir` pour générer le code OCaml de l'analyseur. L'analyseur syntaxique construit un arbre de syntaxe abstraite (AST), valeur de type `pfile` dans le fichier `src/ast.mli`. Cet arbre peut être affiché grâce au module `src/pretty.ml` [2]. Dans la définition du typage et de la sémantique, nous utiliserons une syntaxe abstraite, codée par les types définis dans `src/ast.mli` et `src/tast.mli` respectivement.

2 Typage statique

La première partie de vos travaux consiste à coder une analyse sémantique des programmes NanoGo qui consiste à s'assurer de la conformité de typage. L'analyse doit être codée dans les fichiers `src/tast.mli` et `src/typing.ml`.

Le système de types de NanoGo est un système très simple, sans relation de sous-typage entre les types.

Contexte de typage. 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 (introduit avec la déclaration `type`).

Un contexte de typage Γ contient un ensemble de structures, de fonctions et de variables. Dans le contexte Γ, les noms sont liés à des informations de typage :

- Un nom de structure est lié à la liste et le type de ses champs ; on note $\Gamma \ni S\{\ x : \tau\ \}$ le fait que la structure S possède au moins un unique champ nommé x de type τ dans Γ.
- Un nom de fonction est lié à la liste des paramètres typés et des résultats ; on note $\Gamma \ni f(\tau_1, \ldots, \tau_n) \Rightarrow \tau'_1, \ldots, \tau'_m$, avec $n \geq 0$ et $m \geq 0$ l'information sur f dans Γ.
- Un nom de variable est lié à son type, $x : \tau$.

2. L'affichage en format DOT de ce module a été ajouté au projet par Vincent Lafeychine.

L'opération $\Gamma + x : \tau$ est définie par $\forall x'$, $(\Gamma + x : \tau)(x') = \Gamma(x')$ si $x \neq x'$ et $(\Gamma + x : \tau)(x) = \tau$ sinon. Le fichier `src/tast.mli` code pour le moment que le contexte pour les variables. Vous devez le compléter avec le codage des autres parties du contexte.

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

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

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 (*l-value*) bien typée et de type τ ». Ces jugements sont définis par les règles suivantes :

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

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

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

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

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

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

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

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

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

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 :

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

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

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 :

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

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

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

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

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

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

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

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

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

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

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

Par ailleurs, toutes les variables introduites dans un *même* bloc doivent porter des noms différents. Une exception est faite pour la variable « `_` »qui a un statut spécial : elle n'a pas besoin d'être initialisée avec « `vars` »et ne peut apparaître qu'en partie gauche de l'affectation pour marquer que la valeur de l'expression correspondant n'est pas importante. Ceci est utile pour appeler des procédures (fonctions avec 0 résultats) ou tout simplement résultat n'est pas est vide (dans une expression.

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 le contexte de typage toutes les structures (mais pas leurs champs), en vérifiant l'unicité des noms de structures.

2. (a) On ajoute dans le contexte 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 le contexte de typage 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 contexte 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

i. que toute instruction `return` dans b renvoie bien un résultat du type attendu $\tau'_1, \ldots, \tau'_m$;

ii. si $m > 0$, que toute branche du flot d'exécution dans b aboutit bien à une instruction `return` ;

iii. 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 (mais peut contenir des champs de type $*S$).

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`.

Cette analyse sera codée dans le fichier `src/typing.ml`.

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 arbres, contenant les informations calculées par le typage et d'autres informations qui peuvent être nécessaires à la production de code. Le type des arbres syntaxiques enrichis par le typage sera complété dans le fichier `src/tast.mli` ; ces arbres seront produits par les fonctions du module `src/typing.ml` et peuvent être affichés grâce au module `src/pretty.ml`.

3 Sémantique d'exécution

La sémantique à grands pas de NanoGo est très proche de celle du C.

Le NanoGo travaille sur des valeurs dans $\mathbf{Z}_k$, avec $k = 64$, le domaine des entiers signés sur k bits. NanoGo utilise des pointeurs, ce qui nécessite d'introduire :

— une notion d'adresse dans un domaine $Addr$, qu'on fixe être l'ensemble des entiers en $\mathbf{Z}_k$ multiples de $k/8$;

— une notion de mémoire μ, une fonction partielle qui associe à des adresses $a \in Addr$ une valeur en $\mathbf{Z}_k$; $\texttt{dom}(\mu)$ dénote l'ensemble d'adresses a telles que $\mu(a)$ est définie.

Comme NanoGo supporte l'allocation dynamique et l'appel de fonction, nous devons gérer la pile et le tas. On suppose que la pile est disjointe du tas, par exemple la pile est dans les adresses négatives et le tas dans celles strictement positives ; l'adresse 0 est réservée pour `nil`. On suppose que le tas est géré aussi comme une pile (car il n'y a pas d'opération de libération de la mémoire). L'opération d'allocation de mémoire, $\texttt{alloc}(\mu, n)$ avec n la taille de la plage d'adresses réservées sur les tas, plage renvoyée comme résultat de l'opération. Aux adresses réservées sur le tas les valeurs ne sont pas initialisées, donc une valeur `any` sera utilisée dans la sémantique pour cela.

L'environnement ρ associe à chaque variable x l'adresse $a = \rho(x)$ à laquelle est stockée la variable x, donc la valeur de x est obtenue par $\mu(\rho(x))$. Les variables définies dans l'environnement ρ sont celles dont la portée lexicale comprend l'expression ou la commande. L'environnement ρ_{glob} dénote l'environnement des variables globales d'un programme π ; on suppose qu'il est une application injective, en associant à des variables globales distinctes des adresses distinctes, qui est inclue dans l'environnement courant.

Pour les valeurs structure, on utilise l'information de typage pour associer à chaque nom de champ f sa position $\mathit{offset}(f)$ depuis de début du bloc mémoire occupé par les valeurs de la structure. À noter que $\mathit{offset}(f)$ doit être un multiple de $k/8$ car il sera utilisé pour calculer des adresses. La taille d'une structure, notée $\mathit{sizeof}(S)$, est fixée par le typage et doit être également un multiple de $k/8$.

Les expressions NanoGo n'ont pas des effets de bord, sauf l'expression $\texttt{new(}S\texttt{)}$ qui modifie le domaine de la fonction mémoire. Les instructions ont des effets de bord dans la mémoire et dans le flux de sortie standard. On modélise cet effet par une suite de caractères ω sur laquelle l'opération classique de concaténation $\ll \cdot \gg$s'applique.

Les règles de sémantique sont écrites dans la syntaxe abstraite du langage, celle définie dans le fichier `tast.mli`, où les préfixes `TE` et `TD` ont été supprimés du nom des constructeurs. Dans la suite, on suppose que le programme π est fixée une fois pour toutes, afin d'alléger l'écriture des jugements : à la place de $\rho, \mu \vdash^{\pi} c \Rightarrow \rho', \mu'$ on écrira $\rho, \mu \vdash c \Rightarrow \rho', \mu'$.

En conclusion, jugements de la sémantique opérationnelle sont :

— $\rho, \mu \vdash_l e \Rightarrow v, \mu'$: dans l'environnement ρ et la mémoire μ, l'expression e est évaluée à la valeur v et la mémoire μ' (nouveau tas),

— $\rho, \mu \vdash_l e \Rightarrow a$: dans l'environnement ρ et la mémoire μ, l'expression e est une valeur gauche située à l'adresse a,

— $\rho, \mu, \omega \vdash c \Rightarrow v, \mu', \omega'$: dans l'environnement ρ, la mémoire μ, et le flot de sortie ω,

l'instruction c produit le vecteur (éventuellement vide) de valeurs $\vec{v}$, la mémoire μ' et le flot de sortie ω'.

Sémantique d'une expression.

$$\frac{}{\rho, \mu \vdash n \Rightarrow n \bmod 2^k, \mu} \qquad \frac{\mu \text{ contient } s \text{ à } a}{\rho, \mu \vdash s \Rightarrow a, \mu} \qquad \frac{}{\rho, \mu \vdash \texttt{nil} \Rightarrow 0, \mu}$$

$$\frac{}{\rho, \mu \vdash_l x \Rightarrow \rho(x)} \qquad \frac{\rho, \mu \vdash_l e \Rightarrow a}{\rho, \mu \vdash_l e\texttt{.}f \Rightarrow a + \mathit{offset}(f)} \qquad \frac{\rho, \mu \vdash_l e \Rightarrow a \quad \mu(a) \in \texttt{dom}(\mu)}{\rho, \mu \vdash_l \texttt{*}e \Rightarrow a, \mu}$$

$$\frac{\rho, \mu \vdash_l e \Rightarrow a}{\rho, \mu \vdash e \Rightarrow \mu(a), \mu} \qquad \frac{\rho, \mu \vdash_l e \Rightarrow a}{\rho, \mu \vdash \texttt{\&}e \Rightarrow a, \mu}$$

$$\frac{\rho, \mu \vdash e \Rightarrow n, \mu'}{\rho, \mu \vdash \texttt{-}e \Rightarrow -n, \mu''} \qquad \frac{\rho, \mu \vdash e \Rightarrow b, \mu'}{\rho, \mu \vdash \texttt{!}e \Rightarrow \mathit{not}\ b, \mu''}$$

$$\frac{[a, a + \mathit{sizeof}(S)) = \mathit{alloc}(\mu)}{\rho, \mu \vdash \texttt{new(}S\texttt{)} \Rightarrow a, \mu[\,[a, a + \mathit{sizeof}(S)) \mapsto \texttt{any}]}$$

$$\frac{\rho, \mu \vdash e_1 \Rightarrow n_1, \mu' \quad \rho, \mu' \vdash e_2 \Rightarrow n_2, \mu'' \quad \mathit{cmp} \in \{\texttt{<}, \texttt{<=}, \texttt{>}, \texttt{>=}\}}{\rho, \mu \vdash e_1\ \mathit{cmp}\ e_2 \Rightarrow n_1\ \mathit{cmp}\ n_2, \mu''}$$

$$\frac{\rho, \mu \vdash e_1 \Rightarrow n_1, \mu' \quad \rho, \mu' \vdash e_2 \Rightarrow n_2, \mu'' \quad \mathit{op} \in \{\texttt{+}, \texttt{-}, \texttt{*}\}}{\rho, \mu \vdash e_1\ \mathit{op}\ e_2 \Rightarrow (n_1\ \mathit{op}\ n_2)\ \mathit{mod}\ 2^k, \mu''}$$

$$\frac{\rho, \mu \vdash e_1 \Rightarrow n_1, \mu' \quad \rho, \mu' \vdash e_2 \Rightarrow n_2, \mu'' \quad n_2 \neq 0}{\rho, \mu \vdash e_1\ \texttt{/}\ e_2 \Rightarrow \lfloor n_1/n_2 \rfloor\ \mathit{mod}\ 2^k, \mu''}$$

$$\frac{\rho, \mu \vdash e_1 \Rightarrow n_1, \mu' \quad \rho, \mu' \vdash e_2 \Rightarrow n_2, \mu'' \quad n_2 \neq 0}{\rho, \mu \vdash e_1\ \texttt{\%}\ e_2 \Rightarrow (n_1 - \lceil n_1/n_2 \rceil)\ \mathit{mod}\ 2^k, \mu}$$

La règle pour les constantes chaînes de caractère peut paraître curieuse, mais elle indique seulement que la constante doit être stockée préalablement dans la mémoire et l'évaluation de l'expression renvoie cette adresse. Les règles pour les constantes `true` et `false` ne sont pas données, vous pouvez en proposer une sémantique cohérente avec celle du langage C.

Pour les opérations binaires de comparaison, la sémantique est celle des opérations classiques sur $\mathbf{Z}_k$.

Les bizarreries dans la sémantique de la division et du reste sont faites pour correspondre à la sémantique des mêmes opérations en assembleur x86 (et de la plupart des autres processeurs).

Comme la section suivante (production de code) l'indique, les valeurs des opérandes des expressions binaires seront stockées sur la pile pour éviter l'allocation de registres.

Sémantique de l'appel d'une fonction. On considère qu'il existe une unique fonction de nom f dans le programme NanoGo déclarée par

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

L'ordre d'évaluation des arguments est inverse à celui écrit, mais il ne compte vraiment pas car les expressions n'ont pas d'effet de bord sur la mémoire déjà allouée.

$$\frac{\rho, \mu \vdash e_n \Rightarrow v_n, \mu_n \quad \ldots \quad \rho, \mu_2 \vdash e_1 \Rightarrow v_1, \mu_1 \quad \rho[x_1 \mapsto v_1, \ldots, x_n \mapsto v_n], \mu_1, \epsilon \vdash b \Rightarrow (v'_1, \ldots, v'_m), \mu', \omega'}{\rho, \mu \vdash f(e_1, \ldots, e_n) \Rightarrow (v'_1, \ldots, v'_m), \mu', \omega'}$$

Sémantique d'une instruction.

$$\frac{\rho, \mu \vdash_l e \Rightarrow a \quad n = \mu(a)}{\rho, \mu, \omega \vdash e\texttt{++} \Rightarrow (), \mu[a \mapsto (n+1)], \omega} \qquad \frac{\rho, \mu \vdash_l e \Rightarrow a \quad n = \mu(a)}{\rho, \mu, \omega \vdash e\texttt{--} \Rightarrow (), \mu[a \mapsto (n-1)], \omega}$$

$$\frac{\rho, \mu \vdash e \Rightarrow n, \mu'}{\rho, \mu, \omega \vdash \texttt{fmt.Print(}e\texttt{)} \Rightarrow (), \mu'', \omega \cdot n}$$

$$\frac{\rho, \mu \vdash_l e_1 \Rightarrow a_1 \quad \ldots \quad \rho, \mu \vdash_l e_n \Rightarrow a_n \qquad \rho, \mu \vdash e'_1 \Rightarrow v_1, \mu_1 \quad \ldots \quad \rho, \mu_{n-1} \vdash e'_n \Rightarrow v_n, \mu_n \quad \mu'' = \mu_n[a_1 \mapsto v_1, \ldots, a_n \mapsto v_n]}{\rho, \mu, \omega \vdash e_1, \ldots, e_n\texttt{=}e'_1, \ldots, e'_n \Rightarrow (), \mu'', \omega}$$

$$\frac{\rho, \mu \vdash e \Rightarrow n, \mu' \quad \rho, \mu'', \omega \vdash b_1 \Rightarrow \vec{v}, \mu_1, \omega' \quad n \neq 0}{\rho, \mu, \omega \vdash \texttt{if } e\ b_1\ b_2 \Rightarrow \vec{v}, \mu_1, \omega'}$$

$$\frac{\rho, \mu \vdash e \Rightarrow n, \mu' \quad \rho, \mu'', \omega \vdash b_2 \Rightarrow \vec{v}, \mu_2, \omega' \quad n = 0}{\rho, \mu, \omega \vdash \texttt{if } e\ b_1\ b_2 \Rightarrow \vec{v}, \mu_2, \omega'}$$

$$\frac{\rho, \mu \vdash e \Rightarrow n, \mu' \quad \rho, \mu', \omega \vdash \{b; \texttt{for } e\ b\} \Rightarrow \vec{v}, \mu_1, \omega' \quad n \neq 0}{\rho, \mu, \omega \vdash \texttt{for } e\ b \Rightarrow \vec{v}, \mu_1, \omega'} \qquad \frac{\rho, \mu \vdash e \Rightarrow n, \mu' \quad n = 0}{\rho, \mu, \omega \vdash \texttt{for } e\ b \Rightarrow (), \mu', \omega'}$$

$$\frac{\rho, \mu \vdash e_1 \Rightarrow v_1, \mu_1 \quad \ldots \quad \rho, \mu \vdash e_n \Rightarrow v_n, \mu_n}{\rho, \mu, \omega \vdash \texttt{return } e_1, \ldots, e_n \Rightarrow (v_1, \ldots, v_n), \mu'', \omega}$$

$$\frac{\rho, \mu \vdash f(e_1, \ldots, e_n) \Rightarrow (v_1, \ldots, v_k), \mu', \omega'}{\rho, \mu, \omega \vdash \texttt{return } f(e_1, \ldots, e_n) \Rightarrow (v_1, \ldots, v_k), \mu, \omega'}$$

$$\frac{\rho_1 = \rho[x_1 \mapsto \texttt{any}] \quad \ldots \quad \rho_n = \rho[x_n \mapsto \texttt{any}] \quad \rho_n, \mu_n, \omega \vdash \{s_2; \ldots; s_m\} \Rightarrow \vec{v}, \mu', \omega'}{\rho, \mu, \omega \vdash \{\texttt{var } x_1, \ldots, x_n\ \tau; s_2; \ldots; s_m\} \Rightarrow \vec{v}, \mu'', \omega'}$$

$$\frac{\begin{array}{c}\rho, \mu \vdash e_1 \Rightarrow v_1, \mu_1 \quad \ldots \quad \rho, \mu_{n-1} \vdash e_n \Rightarrow v_n, \mu_n \\ \forall i,\ a_i \text{ adresse non utilisï¿½e de la pile} \\ \mu_0 = \mu_n[a_1 \mapsto v_1, \ldots, a_n \mapsto v_n] \quad \rho' = \rho[x_1 \mapsto a_1, \ldots, x_n \mapsto a_n] \\ \rho', \mu_0, \omega \vdash \{s_2; \ldots; s_m\} \Rightarrow \vec{v}, \mu', \omega' \quad \mu'' = \mu' \setminus \{a_1, \ldots, a_n\}\end{array}}{\rho, \mu, \omega \vdash \{\texttt{var } x_1, \ldots, x_n = e_1, \ldots, e_n; s_2; \ldots; s_m\} \Rightarrow \vec{v}, \mu'', \omega'}$$

$$\frac{\begin{array}{c}\rho, \mu, \omega \vdash f(e_1, \ldots, e_k) \Rightarrow (v_1, \ldots, v_n), \mu', \omega' \\ \forall i,\ a_i \text{ adresse non utilisée de la pile} \\ \mu_0 = \mu'[a_1 \mapsto v_1, \ldots, a_n \mapsto v_n] \quad \rho' = \rho[x_1 \mapsto a_1, \ldots, x_n \mapsto a_n] \\ \rho', \mu_0, \omega' \vdash \{s_2; \ldots; s_m\} \Rightarrow \vec{v}, \mu'_0, \omega'' \quad \mu'' = \mu'_0 \setminus \{a_1, \ldots, a_n\}\end{array}}{\rho, \mu, \omega \vdash \{\texttt{var } x_1, \ldots, x_n = f(e_1, \ldots, e_k); s_2; \ldots; s_m\} \Rightarrow \vec{v}, \mu'', \omega''}$$

$$\frac{}{\rho, \mu, \omega \vdash \{\} \Rightarrow (), \mu, \omega}$$

$$\frac{\rho, \mu, \omega \vdash c_1 \Rightarrow (), \mu', \omega' \quad \rho, \mu', \omega' \vdash \{c_2; \ldots; c_n\} \Rightarrow \vec{v}, \mu'', \omega''}{\rho, \mu, \omega \vdash \{c_1; \ldots; c_n\} \Rightarrow \vec{v}, \mu'', \omega''} \qquad \frac{\rho, \mu, \omega \vdash c_1 \Rightarrow \vec{v}, \mu', \omega' \quad \vec{v} \neq ()}{\rho, \mu, \omega \vdash \{c_1; \ldots; c_n\} \Rightarrow \vec{v}, \mu', \omega'}$$

4 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.

La génération de code x86-64 se fera dans le fichier `src/compile.ml`. On vous recommande d'utiliser le module `src/x86_64.ml` distribué avec le sujet qui permet de construire l'arbre abstrait du programme assembleur et imprimer cet arbre dans un fichier.

4.1 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`. Un texte (suite de caractères) est représenté par l'adresse d'un `string` alloué dans le segment de données. En effet, NanoGo ne permet pas de construire de textes dynamiquement.

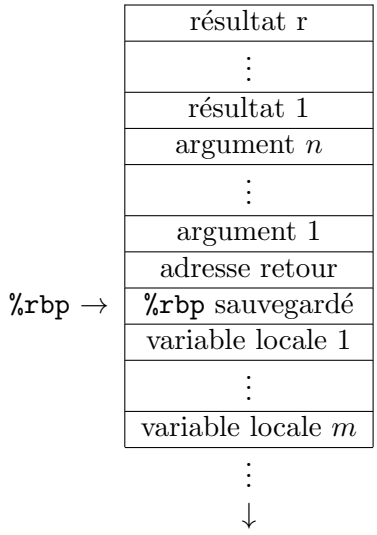
Structures. Une structure est représentée par un bloc mémoire alloué sur la pile (variable locale) ou sur le tas (expression `new`). Ce bloc contient les valeurs des champs de la structure. Le compilateur maintient une table donnant, pour chaque structure S et chaque champ de S, la position de ce champ (son *offset*) 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.

4.2 Schéma de compilation

L'appelant place tous les arguments sur la pile avant de faire `call`. Il réserve également de la place pour les résultats. Si la fonction a un seul résultat, le registre `%rax` peut être utilisé. 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. Au retour,

l'appelant se charge de dépiler les résultats et les arguments. On a donc un tableau d'activation de la forme suivante :



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 NanoGo
  ...   fonctions écrites en assembleur, le cas échéant
        .data
  ...   chaînes de caractères
```

5 Travail demandé

Le projet est à faire seul. Vous devez récupérer sur eCampus les sources initiales du projet et **modifier uniquement** les fichiers indiqués dans les portions marquées `TODO`.

Dépôt du projet. Chaque version du projet doit être remise sous eCampus, dans l'espace du cours dédié au projet, sous la forme d'une archive `tar` compressée (option "`z`" de `tar`), appelée *votre_nom*`.tgz` qui doit contenir un répertoire appelé *votre_nom* (exemple : `dupont.tgz`). Dans ce répertoire doivent se trouver les *sources* du compilateur (inutile d'inclure les fichiers compilés) incluant les fichiers sources distribués. Quand on se place dans ce répertoire, la commande `make` crée le compilateur (grâce au fichier `Makefile` distribué), qui sera appelé `ngoc`. La commande `make clean` efface tous les fichiers que `make` a engendrés et ne laisse dans le répertoire que les fichiers sources. Le fichier `Makefile` distribué utilise `dune`, mais vous pouvez le réécrire pour éviter l'installation de ce paquet.

L'archive du projet doit également contenir un rapport de maximum 3 pages expliquant les différents choix techniques qui ont été faits, les difficultés rencontrées, les éléments réalisés et leur test, 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 lundi 5 décembre 2022 à 10h00). Dans cette première partie du projet, le compilateur `ngoc` doit appeler, si les analyses lexicale et syntaxique se terminent sans erreur, l'analyse de typage. Lorsqu'une erreur de typage est détectée par le compilateur, elle doit être signalée le plus précisément possible, par sa nature et sa localisation dans le fichier source, 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 de typage. Elle est donc sans effet dans cette première partie.

Pour amener à bien cette partie, vous devez tester votre analyse avec, au minimum, les tests distribués dans le répertoire `tests` et d'autres tests que vous devez **écrire et les inclure** dans le rendu du projet. Pour vous aider à la mise au point de ces tests, le code fourni permet, grâce à l'option `--parse-only` de réaliser l'analyse syntaxique du fichier. En cas d'erreur lexicale ou syntaxique, celle-ci est signalée et le typage n'est pas appelé, mais le compilateur se 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.

Pour cette partie, nous vous recommandons de procéder de manière graduelle, construction par construction, dans l'ordre suivant :

I. typage de la déclaration de la fonction `main`, typage des expressions et des instructions simples, ainsi que l'affichage (appels à `fmt.Print`), mais sans appel de fonction utilisateur et sans utilisation de structures ; *cette étape fait partie de la version minimale du projet*,

II. typage des déclarations de fonction utilisateur et des expressions et instructions qui utilisent des appels de ces fonctions,

III. typage des déclarations de structures et leur utilisation dans les expressions.

Le fichier `src/typing.ml` contient une ébauche du typage de `fmt.Print`.

Partie 2 (à rendre pour le lundi 18 janvier 2023 à 10h00). 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
```

Pour cette partie du projet, l'ordre recommandé est le suivant :

IV. traduction de `main` et des instructions et expressions arithmétiques qu'il utilise ainsi que l'affichage (appels à `fmt.Print`) ; *cette étape fait partie de la version minimale du projet*,

V. traduction du corps des fonctions définies par l'utilisateur et des appels à ces fonctions,

VI. traduction de déclarations de structure et leur utilisation dans les expressions.

Le fichier `src/compile.ml` contient une ébauche de la production de code pour `fmt.Print`.

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, comme indiqué et de s'assurer à chaque étape que votre code compile et passe les tests donnés dans le répertoire `tests`.

Afin de ne pas perdre des versions qui fonctionnent, nous vous recommandons d'utiliser un gestionnaire de versions comme git, ou tout simplement de sauvegarder les versions correctes sur un ou plusieurs disques.

Même si vous travaillez seul sur ce projet, vous devez documenter votre code et écrire un journal de votre développement (le format Markdown est très facile pour cette tâche). Ces notes et commentaires vous permettrons de générer rapidement le rapport du projet.

Respectez le calendrier et les étapes proposées afin d'avancer régulièrement.

La soutenance sera faite en 15 minutes par un enseignant du module, après avoir lu votre rapport et testé votre compilateur. Il faut donc lui faire une rapide démonstration de votre compilateur qui relève les points les plus saillants. L'examinateur peut vous poser des questions sur toute partie du code, y compris le code qui vous a été fourni.

A Eléments de langage pour NanoGo

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)
$\langle règle \rangle_t^\star$	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 `)`.

A.1 Lexique de NanoGo

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 :

$\langle chiffre \rangle$ ::= `0`–`9`
$\langle alpha \rangle$ ::= `a`–`z` | `A`–`Z` | `_`
$\langle ident \rangle$ ::= $\langle alpha \rangle$ ($\langle alpha \rangle$ | $\langle chiffre \rangle$)*

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 $\langle entier \rangle$ et $\langle chaîne \rangle$ suivantes :

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

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

Ainsi, l'ensemble des constantes Σ contient les constantes booléennes `true` et `false`, ainsi que les constantes entières et les constantes que sont les chaines de caractères, déterminées lors de l'analyse lexicale.

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 NanoGo 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` | `++` | `--` | `)` | `}`

A.2 Syntaxe concrète de NanoGo

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{\}} \ \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 NanoGo.

B Utiliser OCaml avec OPAM

Les sources fournis pour réaliser ce projet utilisent dune pour compiler les fichiers OCaml. Pour plus de faciliter, il est préférable d'installer OCaml avec OPAM. Comme OCaml est déjà installé sur les machines du département, cette annexe vous explique comment configurer OCaml à partir du gestionnaire de paquets OPAM.

1. Lancez $ opam init, et attendez quelques instants.
2. OPAM devrait proposer d'éditer le fichier [3] ∼/.bash_profile pour établir les variables nécessaires à OCaml au démarrage. Acceptez, à deux reprises, en appuyant sur la touche "y". Vérifiez que l'un des fichiers contient une ligne de la forme :

   ```
   # opam configuration
   test -r $HOME/.opam/opam-init/init.sh &&
   . $HOME/.opam/opam-init/init.sh > /dev/null 2> /dev/null || true
   ```

   Si ce n'est pas le cas, ajoutez ces lignes au fichier ∼/.bash_profile.
3. Comme OPAM vous l'indique, exécutez : $ eval $(opam env)
4. Etant donné que ∼/.bash_profile est lu à l'ouverture de la session, pour charger l'environnement OCaml, il faut ou bien vous déloger et reloger, ou bien charger le ∼/.bash_profile via $. ∼/.bash_profile ou bien évaluer la configuration que propose OPAM $ eval $(opam config env)
5. Pour s'assurer que OPAM est bien détecté, on peut vérifier que la variable d'environnement PATH contient un chemin vers OPAM. Pour se faire, il faut vérifier que la sortie de la commande $ echo $PATH contienne une entrée contenant un .opam.
6. On peut maintenant installer les paquets nécessaires, avec
   $ opam install tuareg merlin dune.
7. La configuration de l'éditeur peut-être gérée par OPAM avec
   $ opam user-setup install.

3. Le nom fichier de ce fichier peut différer d'une machine à une autre. Si le fichier n'existe pas, essayez ∼/.bashrc ou ∼/.profile