Langages de programmation, interprétation, compilation
Projet 2025–26

Micro Go

Partie 1 — 10 novembre 2025

L'objectif de ce projet est de réaliser un interprète (et un compilateur produisant du code MIPS) pour un fragment de Go, appelé Micro Go par la suite. Il s'agit d'un fragment relativement petit du langage Go, avec parfois même quelques petites incompatibilités.

Les programmes Go peuvent être exécutés en ligne, par exemple sur le site `https://www.mycompiler.io/fr/new/go`. Il est recommandé de tester des programmes simples pour se familiariser avec la syntaxe et la sémantique du langage.

1 Exemples

Notre langage Micro Go comporte un noyau classique de langage de programmation impératif (expressions arithmétiques et logiques, conditionnelles, boucles).
Il permet de définir et utiliser des structures (tableaux avec champs nommés, possiblement récursifs pour représenter des structures chaînées). Comme en C, le langage Go permet de manipulés des valeurs de type structure avec plusieurs champs et dispose d'un mécanisme explicite de pointeurs (expressions $*e$ et $\&e$). Ici, dans Micro Go, on ne va considérer comme valeur que des pointeurs vers des structures.

Une autre particularité est de pouvoir renvoyer des n-uplets de valeurs comme résultat d'une fonction. Il est alors possible d'utiliser ce n-uplet (qui n'est pas une valeur) dans une affectation multiple ou en argument d'une autre fonction.

Ces différents points sont illustrés par le programme ci-dessous qui effectue la division euclidienne de deux entiers de plusieurs manières différentes.

```
package main;
import "fmt";

/* a, b entiers naturels, b > 0 */
func div1(a,b int) (int,int) {
    if (a < b) { return 0, a } else
    { x,y := div1(a-b,b); return x+1,y }
  };
func div2(a,b int) (int,int) {
    q := 0;
    for (a >= b) { q++; a=a-b };
    return q, a
  };
```

```
/* Version avec retour d'une structure */
type res struct { quo int; rem int };
func div3(a,b int) *res {
    r := new(res);
    r.quo = 0;
    for (a >= b) { r.quo++; a=a-b };
    r.rem = a;
    return r
  };
func main() {
        fmt.Print(div1(45,6)); fmt.Print("\n");
        fmt.Print(div2(45,6)); fmt.Print("\n");
        r:=div3(45,6); fmt.Print(r.quo,r.rem); fmt.Print("\n");
        fmt.Print(r); fmt.Print("\n");
};
```

2 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 les signes « * », « $^+$ » et les parenthèses utilisées dans la syntaxe des expressions régulières avec les symboles correspondants du langage Go « * », « + », « (» et «) ».

2.1 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 :

⟨*chiffre*⟩ ::= `0–9`
⟨*alpha*⟩ ::= `a–z` | `A–Z` | `_`
⟨*ident*⟩ ::= ⟨*alpha*⟩ (⟨*alpha*⟩ | ⟨*chiffre*⟩)*

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 (Bonus). Pour épargner au programmeur la peine d'écrire des points-virgules à la fin des lignes qui contiennent des instructions, l'analyseur lexical de Micro 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` | `++` | `--` | `)` | `}`

En particulier le **else** d'une conditionnelle doit être placé sur la même ligne que le bloc précédent. Par exemple :

```
if (a < b) { return 0, a }
else { x,y := div1(a-b,b); return x+1,y }
```

est transformé en :

```
if (a < b) { return 0, a } ;
else { x,y := div1(a-b,b); return x+1,y }
```

et l'analyse syntaxique échoue sur ce programme. Il faut donc écrire :

```
if (a < b) { return 0, a } else
{ x,y := div1(a-b,b); return x+1,y }
```

Dans un premier temps, vous pourrez supposer que l'utilisateur entre lui-même les points-virgules.

2.2 Analyse syntaxique

La grammaire des fichiers sources considérés est donnée dans la figure 1. Le point d'entrée est le non-terminal ⟨*fichier*⟩.

Un *fichier* Micro Go débute par la déclaration du package `main`, de manière optionnelle l'import de la bibliothèque `"fmt"` puis comporte une suite de déclarations.

Une *déclaration* est soit une déclaration d'un type de structures introduite par le mot-clé `type`, soit une déclaration de fonction introduite par le mot-clé `func`. Les deux déclarations se terminent par un point-virgule.

— Une *structure* possède un nom puis une liste de champs (identificateurs) associés à un type. Les déclarations de champs sont séparées par des `;` comme dans

type res **struct** { quo **int**; rem **int** };

Les déclarations de même type peuvent être regroupées, on peut écrire de manière équivalente **type** res **struct** { quo, rem **int** };

— Une *fonction* possède un nom, une liste de paramètres typés entre parenthèses, séparés par des virgules, un type de retour et un bloc d'instructions. Le type de retour est optionnel (pour les fonctions ne renvoyant pas de valeur), il peut aussi correspondre au retour de plusieurs valeurs auquel cas c'est une liste de types entre parenthèses, séparés par des virgules.

Types. Les *types* sont soit `int`, `bool` ou `string` (pour les entiers, booléens et chaînes de caractères) soit de la forme $*S$ avec S le nom d'une structure déclarée dans l'environnement.

Instructions. Une *instruction* est de l'une des formes suivantes :

— une *instruction simple* peut être une expression, possiblement suivie de `++` ou `--` pour incrémenter ou décrémenter une valeur, ou bien une affectation.
L'affectation de base peut se faire sur plusieurs variables, par exemple `x,y = div1(a–b,b)`, la forme `x,y := div1(a–b,b)` est juste une abbréviation pour combiner la déclaration des variables `x` et `y` et leur initialisation. C'est donc équivalent à **var** `x,y = div1(a–b,b)`.
Règle : L'instruction $x_1, \ldots, x_n$:=$e_1, \ldots, e_m$ équivaut à `var` $x_1, \ldots, x_n = e_1, \ldots, e_m$.
La vérification de cohérence entre le nombre de variables et le nombre de valeurs ne se fera qu'au typage, ce nombre peut être différent dans le cas d'un appel de fonction.

— un *bloc d'instructions* est une suite d'instructions entre accolades `{}` séparées par des points-virgules.

— une instruction conditionnelle commence par `if` e `{...}` avec e une expression et peut se poursuivre par une branche `else` qui est suivie soit d'un bloc, soit d'une autre expression conditionnelle. L'absence de branche `else` équivaut à une branche sur un bloc d'instructions vide.
Règle : l'instruction `if` $e\ b$ équivaut à `if` $e\ b$ `else {}`.

— une *déclaration de variables* est introduite par le mot clé `var`. On peut déclarer plusieurs variables (séparées par des virgules) et de manière optionnelle, déclarer leur type et

donner leur valeur initiale. On pourra de manière équivalente écrire

```
var r *res; r = new(res)
```

ou en intégrant l'initialisation à la déclaration

```
var r *res = new(res)
```

ou encore en omettant la déclaration du type de la variable

```
var r = new(res)
```

— une *boucle* est introduite par le mot-clé `for`. La construction de base est `for` e `{b}` avec e une expression et b une suite d'instructions. Cela correspond à une boucle `while` (tant que e s'évalue en vrai, effectuer les instructions du bloc).
Le langage autorise deux variantes syntaxiques :
— l'instruction `for` b équivaut à `for true` b (boucle infinie).
— l'instruction `for` i_1`;` e`;` i_2 b, avec i_1 et i_2 des instructions simples, équivaut à
`{` i_1`; for` e `{` b i_2 `}}` pour simuler une boucle `for` classique :

```
for k:=0; k< 10; k++ { .... }
```

— une instruction `return` est suivi d'une liste d'expressions séparées par des virgules.

Expressions. La syntaxe des expressions est assez classique avec des constantes (dont `nil` pour représenter le pointeur nul), des opérateurs unaires (`!` pour la négation booléenne, `-` pour le moins unaire), des opérations binaires, des appels de fonctions.
L'expression `new(`S`)` représente un nouvel objet de la structure S et l'accès au champs x d'une structure e est noté $e.x$.
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), `!`	—
`.`	gauche

On notera dans la grammaire, la possibilité d'ajouter des signes de ponctuation (optionnels) à la fin de certaines séquences non-vides. Ainsi dans un bloc, où une déclaration de structure, il est autorisé de mettre un point-virgule avant l'accolade fermante, et dans les arguments d'une fonction, ou dans un type de retour à plusieurs composantes, on peut mettre une virgule finale avant la parenthèse fermante.

⟨*fichier*⟩ ::= `package main ;` (`import "fmt" ;`)? ⟨*decl*⟩* `EOF`

⟨*decl*⟩ ::= ⟨*structure*⟩ | ⟨*fonction*⟩

⟨*structure*⟩ ::= `type` ⟨*ident*⟩ `struct {` (⟨*vars*⟩ `;`)*⟨*vars*⟩? `} ;`

⟨*fonction*⟩ ::= `func` ⟨*ident*⟩ `(` (⟨*vars*⟩ `,`)*⟨*vars*⟩? `)` ⟨*type_retour*⟩? ⟨*bloc*⟩ `;`

⟨*vars*⟩ ::= ⟨*ident*⟩$^+_{,}$ ⟨*type*⟩

⟨*type_retour*⟩ ::= ⟨*type*⟩
| `(` ⟨*type*⟩$^+_{,}$ `,?` `)`

⟨*type*⟩ ::= `int` | `bool` | `string` | `*` ⟨*ident*⟩

⟨*expr*⟩ ::= ⟨*entier*⟩ | ⟨*chaîne*⟩ | `true` | `false` | `nil`
| `(` ⟨*expr*⟩ `)`
| ⟨*ident*⟩
| ⟨*expr*⟩ `.` ⟨*ident*⟩
| ⟨*ident*⟩ `(` ⟨*expr*⟩$^\star_{,}$ `)`
| `fmt.Print (` ⟨*expr*⟩$^\star_{,}$ `)`
| `!` ⟨*expr*⟩ | `-` ⟨*expr*⟩
| ⟨*expr*⟩ ⟨*op*⟩ ⟨*expr*⟩

⟨*op*⟩ ::= `==` | `!=` | `<` | `<=` | `>` | `>=`
| `+` | `-` | `*` | `/` | `%` | `&&` | `||`

⟨*bloc*⟩ ::= `{` (⟨*instr*⟩ `;`)*⟨*instr*⟩? `}`

⟨*instr*⟩ ::= ⟨*instr_simple*⟩ | ⟨*bloc*⟩ | ⟨*instr_if*⟩
| `var` ⟨*ident*⟩$^+_{,}$ ⟨*type*⟩? (`=` ⟨*expr*⟩$^+_{,}$)?
| `return` ⟨*expr*⟩$^\star_{,}$
| `for` ⟨*bloc*⟩
| `for` ⟨*expr*⟩ ⟨*bloc*⟩
| `for` ⟨*instr_simple*⟩? `;` ⟨*expr*⟩ `;` ⟨*instr_simple*⟩? ⟨*bloc*⟩

⟨*instr_simple*⟩ ::= ⟨*expr*⟩
| ⟨*expr*⟩ (`++` | `--`)
| ⟨*expr*⟩$^+_{,}$ `=` ⟨*expr*⟩$^+_{,}$
| ⟨*ident*⟩$^+_{,}$ `:=` ⟨*expr*⟩$^+_{,}$

⟨*instr_if*⟩ ::= `if` ⟨*expr*⟩ ⟨*bloc*⟩
| `if` ⟨*expr*⟩ ⟨*bloc*⟩ `else` ⟨*bloc*⟩
| `if` ⟨*expr*⟩ ⟨*bloc*⟩ `else` ⟨*instr_if*⟩

FIGURE 1 – Grammaire des fichiers Micro Go.

3 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 :

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

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 $m \geq 0$.

Sur le plan lexical, on a autorisé l'identifiant _ comme nom de variable. Cette variable a un rôle spécial en lien avec les fonctions qui renvoient plusieurs valeurs. Par exemple si la fonction `f` retourne 3 valeurs, et que seule la seconde nous intéresse alors on pourra écrire `_,y,_:=f()`, seule la variable `y` sera ajoutée à l'environnement. L'opération $\Gamma + x : \tau$ d'ajout d'une déclaration à un environnement qui est utilisée pour la bonne formation des fonctions et des instructions sera telle que $\Gamma + \_ : \tau = \Gamma$

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 :

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

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 $\Gamma \vdash f(e_1, \ldots, e_n) \Rightarrow \tau_1, \ldots, \tau_m$ est utilisé pour les fonctions qui renvoient plusieurs valeurs. Il signifie « dans le contexte Γ, l'appel de fonction $f(e_1, \ldots, e_n)$ est bien typé et renvoie m valeurs de types $\tau_1, \ldots, \tau_m$ ».

Ces jugements sont définis par les règles suivantes.

— Les constantes entières ont pour type `int`, les constantes booléennes ont pour type `bool` et les constantes chaînes de caractères ont pour type `string`. La constante `nil` représente un pointeur nul dont le type peut correspondre à n'importe quelle structure déclarée.

$$\frac{c \text{ constante de type } \tau}{\Gamma \vdash c : \tau} \qquad \frac{S \in \Gamma}{\Gamma \vdash \texttt{nil} : *S}$$

— Les variables de l'environnement ont le type déclaré dans l'environnement.

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau}$$

Comme la variable _ n'est jamais entrée dans l'environnement, ce n'est pas une expression bien typée et elle ne peut donc pas apparaître dans une expression correcte.

— La construction `new` crée une nouvelle structure et renvoie un pointeur sur cette structure. Si une expression a pour type uns structure qui a été déclarée avec un champs x de type τ, on peut accéder à la valeur de ce champs.
La valeur `nil` est problématique car on peut lui donner pour type n'importe quelle structure déclarée, par ailleurs l'expression `nil.x` n'a pas de sens sur le plan opérationnel, cela nous amène à ne pas accepter une telle expression.

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

— Les opérations unaires et binaires ont le comportament attendu. Le test d'égalité ou d'inégalité peut se faire entre des expressions de type quelconque. Il n'est néanmoins pas autorisé de comparer `nil` à lui-même.

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

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

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

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

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

— appel de fonction : si une fonction renvoie un seul élément, alors le résultat de l'appel est considéré comme une expression du type correspondant

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

Règle de bonne formation d'un appel de fonction $\Gamma \vdash f(e_1, \ldots, e_n) \Rightarrow \tau_1, \ldots, \tau_m$:

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

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

Cette seconde règle permet de passer directement les n résultats d'une fonction g en arguments d'une fonction f. Dans le cas où $n = 1$ l'appel de fonction $g(e_1, \ldots, e_k)$ est typable comme une expression et la première règle s'applique, dans le cas où $n = 0$ la fonction g ne renvoie pas de valeur et ne peut pas être passée en argument.

Valeurs gauches. On utilise la notation $\Gamma \vdash_l e : \tau$ pour représenter le fait que $\Gamma \vdash e : \tau$ et de plus e est une valeur gauche, c'est-à-dire une expression de la forme $\langle \textit{ident} \rangle (. \langle \textit{ident} \rangle)^*$.

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

— on peut incrémenter/décrémenter des valeurs gauches de type entier

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

— On peut imprimer une suite de valeurs soit donnée par une suite d'expressions soit résultat de l'appel d'une fonction à résultat multiple.

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

— Affectation : les expressions à gauche de l'affectation doivent être des valeurs gauches et leur type déclaré doit correspondre aux types des valeurs données. On ignore les e_i de la forme _.

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

— Conditionnelle et boucle (on ne considère que le cas général, les situations particulières ayant été résolues au moment de l'analyse syntaxique).

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

— Instruction `return`

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

— Déclarations de variables. Les variables peuvent être déclarées avec ou sans leur type et avec ou sans une initialisation, par ailleurs l'initialisation peut se faire par une liste d'expressions ou bien par l'appel d'une fonction. L'initialisation sans contrainte de typage ne peut se faire avec la valeur `nil` qui n'a pas un type unique. Par ailleurs il faut qu'un type soit donné explicitement ou bien qu'il y ait une initialisation. La portée des variables est limitée à la suite du bloc.

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

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

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

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

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

Par ailleurs, toutes les variables introduites au même niveau dans un *même* bloc d'instructions doivent porter des noms différents. Une exception est faite pour l'identifiant _ qui n'introduit pas explicitement de variable dans le contexte et peut donc être utilisé plusieurs fois.

— Typage d'un bloc d'instructions

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

Typage d'un fichier. Les déclarations d'un fichier peuvent apparaître dans n'importe quel ordre. En particulier, les fonctions sont mutuellement récursives et de même 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. 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` ;

— **Bonus** vérifier que toute variable locale introduite dans b, autre que `_`, est bien utilisée.

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`.

4 Travail demandé

On fournit un squelette de code contenant les éléments suivants.

Fichier	Contenu	Commentaire
mgoast.ml	syntaxe abstraite	
mgolexer.mll	analyse lexicale	à compléter (ocamllex)
mgoparser.mly	analyse grammaticale	à compléter (menhir)
typechecker.ml	vérification des types	à compléter (caml)
mgoc.ml	programme principal	
dune/dune-project	configuration	
tests	dossier de tests	à compléter (go)

Votre travail principal consiste à compléter les fichiers `mgolexer.mll`, `mgoparser.mly` et `typechecker.ml` en respectant les descriptions données ci-dessus. Vous obtiendrez alors avec le programme `mgoc` un analyseur complet pour Micro Go (il restera à faire la partie interprétation et compilation). On s'attend à ce que vous ajoutiez également de nouveaux tests.

Vous pouvez travailler seul ou en binôme. Votre projet devra être rendu sur ecampus (sous la forme d'une archive zip respectant le format du squelette donné). Joignez à votre projet un rapport décrivant ce qui a été réalisé, ce qui fonctionne ou non, et les difficultés que vous avez pu rencontrer. Le rapport doit également détailler les éventuelles extensions que vous avez traitées. Le rapport peut prendre la forme d'un simple fichier `README.txt`. ou d'un fichier `pdf`.

Le squelette de code est conçu pour traiter correctement un programme minimal. En plus de ces éléments, vous pouvez réutiliser sans limitations les fragments de code présentés dans le cours. Attention en revanche : tout emprunt de code d'une autre source que le cours doit être documenté dans votre rapport.

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.

Utilisation du programme. Dans cette première partie du projet, le compilateur mgoc accepte sur sa ligne de commande une option éventuelle (parmi `--parse-only` et `--type-only`) et exactement un fichier Micro 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 peut être modifié. En cas d'erreur, le compilateur doit terminer avec le code de sortie 1 (`exit 1`).

Si le fichier est syntaxiquement correct, le compilateur termine avec le code de sortie 0 si l'option `--parse-only` a été passée sur la ligne de commande. Sinon, il poursuit 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 termine avec le code de sortie 2 (`exit 2`). L'option `--type-only` indique de stopper la compilation après l'analyse sémantique (typage). Elle est sans effet dans cette première partie.

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