École Polytechnique 2025–2026

CSC_52064 Compilation

Mini Go

version 1 — January 9, 2026

The goal is to build a compiler for a tiny fragment of the Go programming language, called Mini Go in the following, to x86-64 assembly. This fragment contains integers, Booleans, strings, structures, and pointers. It is compatible with Go. This means that Go can be used as a reference when needed.

The syntax of Mini Go is described in Sec. 1. A parser is provided, together with abstract syntax. You have to implement static type checking (Sec. 2) and code generation (Sec. 3).

1 Syntax

We use the following notations in grammars:

$\langle rule \rangle^\star$	repeats $\langle rule \rangle$ an arbitrary number of times (including zero)
$\langle rule \rangle^\star_t$	repeats $\langle rule \rangle$ an arbitrary number of times (including zero), with separator t
$\langle rule \rangle^+$	repeats $\langle rule \rangle$ at least once
$\langle rule \rangle^+_t$	repeats $\langle rule \rangle$ at least once, with separator t
$\langle rule \rangle?$	use $\langle rule \rangle$ optionally
($\langle rule \rangle$)	grouping

Be careful not to confuse "*" and "$^+$" with "`*`" and "`+`" that are Go symbols. Similarly, do not confuse grammar parentheses with terminal symbols `(` and `)`.

1.1 Lexical Conventions

Spaces, tabs, and newlines are blanks. Comments are of two kinds:

- delimited by `/*` and `*/` (and not nested);
- starting from `//` and extending to the end of line.

Identifiers follow the regular expression $\langle ident \rangle$:

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

The following identifiers are keywords:

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

Integer literals follow the regular expression ⟨*integer*⟩:

$$\begin{array}{lcl} \langle \textit{hexa} \rangle & ::= & \texttt{0–9} \mid \texttt{a–f} \mid \texttt{A–F} \\ \langle \textit{integer} \rangle & ::= & \langle \textit{digit} \rangle^+ \\ & \mid & (\texttt{0x} \mid \texttt{0X})\ \langle \textit{hexa} \rangle^+ \end{array}$$

Integer literals must be in the range -2^{63} to $2^{63}-1$. A string literal ⟨*string*⟩ is enclosed with quotes (`"`). There are four escape sequences: `\"` (for the character `"`), `\n` (for a newline character), `\t` (for a tabulation character), and `\\` (for the character `\`).

Automatic insertion of semi-colons. To save the programmer the trouble of writing semicolons at the end of lines that contain instructions, the lexical analyzer automatically inserts a semicolon when it encounters a carriage return and the previously emitted token was part of the following set:

⟨*ident*⟩ | ⟨*integer*⟩ | ⟨*string*⟩ | `true` | `false` | `nil` | `return` | `++` | `-` | `)` | `}`

1.2 Syntax

The grammar of source files is given in Fig. 1. The entry point is ⟨*file*⟩. Associativity and priorities are given below, from lowest to strongest priority.

operation	**associativity**
`||`	left
`&&`	left
`==`, `!=`, `>`, `>=`, `<`, `<=`	left
`+`, `-`	left
`*`, `/`, `%`	left
`-` (unary), `*` (unary), `&`, `!`	—
`.`	left

Syntactic sugar. We have the following equivalences:

- instruction `for` b stands for `for true` b (infinite loop).
- instruction `for` i_1`;` e`;` i_2 b stands for `{` i_1`; for` e `{` b i_2 `}}`.
- instruction `if` e b stands for `if` e b `else {}`.
- instruction $x_1, \ldots, x_n$ `:=` $e_1, \ldots, e_m$ stands for `var` $x_1, \ldots, x_n$ `=` $e_1, \ldots, e_m$.

Syntactic sugar is eliminated at parsing time (in the provided parser).

$$
\begin{array}{lcl}
\langle \mathit{file} \rangle & ::= & \texttt{package main ;} \ (\texttt{import "fmt" ;})? \ \langle \mathit{decl} \rangle^\star \ \texttt{EOF} \\
\langle \mathit{decl} \rangle & ::= & \langle \mathit{structure} \rangle \mid \langle \mathit{function} \rangle \\
\langle \mathit{structure} \rangle & ::= & \texttt{type} \ \langle \mathit{ident} \rangle \ \texttt{struct \{} \ (\langle \mathit{vars} \rangle^+_{\texttt{;}} \ \texttt{;}?)? \ \texttt{\}} \ \texttt{;} \\
\langle \mathit{function} \rangle & ::= & \texttt{func} \ \langle \mathit{ident} \rangle \ \texttt{(} \ (\langle \mathit{vars} \rangle^\star_{\texttt{,}} \ \texttt{,}?)? \ \texttt{)} \ \langle \mathit{return\_type} \rangle? \ \langle \mathit{bloc} \rangle \ \texttt{;} \\
\langle \mathit{vars} \rangle & ::= & \langle \mathit{ident} \rangle^+_{\texttt{,}} \ \langle \mathit{type} \rangle \\
\langle \mathit{return\_type} \rangle & ::= & \langle \mathit{type} \rangle \\
& \mid & \texttt{(} \ \langle \mathit{type} \rangle^+_{\texttt{,}} \ \texttt{,}? \ \texttt{)} \\
\langle \mathit{type} \rangle & ::= & \langle \mathit{ident} \rangle \\
& \mid & \texttt{*} \ \langle \mathit{type} \rangle \\
\langle \mathit{expr} \rangle & ::= & \langle \mathit{integer} \rangle \mid \langle \mathit{string} \rangle \mid \texttt{true} \mid \texttt{false} \mid \texttt{nil} \\
& \mid & \texttt{(} \ \langle \mathit{expr} \rangle \ \texttt{)} \\
& \mid & \langle \mathit{ident} \rangle \\
& \mid & \langle \mathit{expr} \rangle \ \texttt{.} \ \langle \mathit{ident} \rangle \\
& \mid & \langle \mathit{ident} \rangle \ \texttt{(} \ \langle \mathit{expr} \rangle^\star_{\texttt{,}} \ \texttt{)} \\
& \mid & \texttt{fmt . Print (} \ \langle \mathit{expr} \rangle^\star_{\texttt{,}} \ \texttt{)} \\
& \mid & \texttt{!} \ \langle \mathit{expr} \rangle \mid \texttt{-} \ \langle \mathit{expr} \rangle \mid \texttt{\&} \ \langle \mathit{expr} \rangle \mid \texttt{*} \ \langle \mathit{expr} \rangle \\
& \mid & \langle \mathit{expr} \rangle \ \langle \mathit{operator} \rangle \ \langle \mathit{expr} \rangle \\
\langle \mathit{operator} \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 \mathit{bloc} \rangle & ::= & \texttt{\{} \ ((\langle \mathit{stmt} \rangle)^+_{\texttt{;}} \ \texttt{;}?)? \ \texttt{\}} \\
\langle \mathit{stmt} \rangle & ::= & \langle \mathit{simple\_stmt} \rangle \mid \langle \mathit{bloc} \rangle \mid \langle \mathit{stmt\_if} \rangle \\
& \mid & \texttt{var} \ \langle \mathit{ident} \rangle^+_{\texttt{,}} \ \langle \mathit{type} \rangle? \ (\texttt{=} \ \langle \mathit{expr} \rangle^+_{\texttt{,}})? \\
& \mid & \texttt{return} \ \langle \mathit{expr} \rangle^\star_{\texttt{,}} \\
& \mid & \texttt{for} \ \langle \mathit{bloc} \rangle \\
& \mid & \texttt{for} \ \langle \mathit{expr} \rangle \ \langle \mathit{bloc} \rangle \\
& \mid & \texttt{for} \ \langle \mathit{simple\_stmt} \rangle? \ \texttt{;} \ \langle \mathit{expr} \rangle \ \texttt{;} \ \langle \mathit{simple\_stmt} \rangle? \ \langle \mathit{bloc} \rangle \\
\langle \mathit{simple\_stmt} \rangle & ::= & \langle \mathit{expr} \rangle \\
& \mid & \langle \mathit{expr} \rangle \ (\texttt{++} \mid \texttt{--}) \\
& \mid & \langle \mathit{expr} \rangle^+_{\texttt{,}} \ \texttt{=} \ \langle \mathit{expr} \rangle^+_{\texttt{,}} \\
& \mid & \langle \mathit{ident} \rangle^+_{\texttt{,}} \ \texttt{:=} \ \langle \mathit{expr} \rangle^+_{\texttt{,}} \\
\langle \mathit{stmt\_if} \rangle & ::= & \texttt{if} \ \langle \mathit{expr} \rangle \ \langle \mathit{bloc} \rangle \ (\texttt{else} \ (\langle \mathit{bloc} \rangle \mid \langle \mathit{stmt\_if} \rangle))?
\end{array}
$$

Figure 1: Grammar of Mini Go.

2 Static Typing

Static types τ are given by the following abstract syntax:

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

where S is a structure name. A typing context Γ contains structures (written S), variables (written $x : \tau$), and functions (written $f(\tau_1, \ldots, \tau_n) \Rightarrow \tau'_1, \ldots, \tau'_m$ with $n \geq 0$ and $m \geq 0$).

Well-formed types. The judgment $\Gamma \vdash \tau \ \mathit{bf}$ means "the type τ is well-formed in environment Γ". It is defined as follows:

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

Structure fields. We write $S\texttt{\{ } x : \tau \texttt{ \}}$ the fact that structure S has a field x of type τ.

Typing expressions. The judgment $\Gamma \vdash e : \tau$ means "in the context Γ, the expression e is well typed with type τ". The judgment $\Gamma \vdash_l e : \tau$ additionally means that e is a left value. These two judgments are defined by the following set of rules.

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

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

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

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

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

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

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

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

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

Typing function calls. The judgment $\Gamma \vdash f(e_1, \ldots, e_n) \Rightarrow \tau_1, \ldots, \tau_m$ means "in the context Γ, the function call $f(e_1, \ldots, e_n)$ is well typed and returns m values of types $\tau_1, \ldots, \tau_m$". It is defined as follows:

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

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

The second rule allows us to pass the n results of function g to a function f expecting n parameters.

Typing statements. The judgment $\Gamma \vdash s$ means "in the context Γ, the statement s is well typed". It is defined as follows:

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

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

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

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

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

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

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

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

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

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

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

In addition, all the variables introduced in the *same* block must have distinct names, with the exception of variables named `_`.

Typing a file. The declarations in a file can appear in any order. In particular, functions and structures are mutually recursive. It is suggested to proceed in three steps:

1. We add all the structures (but not their fields) to the environment, checking the uniqueness of the structure names.

2. (a) We add all the functions to the environment, checking the uniqueness of the function names. For a function declaration of the form

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

   we check that the x_i are pairwise distinct and that all types τ_i and τ'_j are well-formed.

   (b) We check and add all the structure fields to the environment. For a structure declaration of the form

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

   we check that the x_i are pairwise distinct and that all types τ_i are well-formed.

3. (a) For each function declaration

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

   we construct a new environment Γ by adding all the variables $x_i : \tau_i$ to the environment containing the structures and functions, and we type the block b in Γ, *i.e.*, we check $\Gamma \vdash b$. Additionally, we check

   - that every `return` statement in b returns a list of m values of the expected types $\tau'_1, \ldots, \tau'_m$;
   - if $m > 0$, that any control-flow branch in b reaches a `return` statement;
   - that any local variable introduced in b, other than `_`, is further used.

   (b) We check that there is no "recursive" structure, that is, no structure S with a field (which contains a field, which contains a field, etc.) of type S, without passing through a pointer.

Finally, we check that there exists a function `main` without parameters and without return type and that the file contains `import "fmt"` if and only if there is at least one `fmt.Print` instruction somewhere in the file.

3 Code Generation

The aim is to produce a simple but correct compiler. In particular, we do not attempt to do any kind of register allocation, but simply use the stack to store any intermediate calculations. Of course, it is possible, and even desirable, to use some x86-64 registers locally. Memory is allocated using `malloc` and no attempt will be made to free memory.

Simplification. To make things simpler, you can assume that your project will not be tested on programs passing structures by value, returning structures from functions, or assigning structures. Beside, some program transformations are performed on the typed syntax trees, between static typing and code generation, to

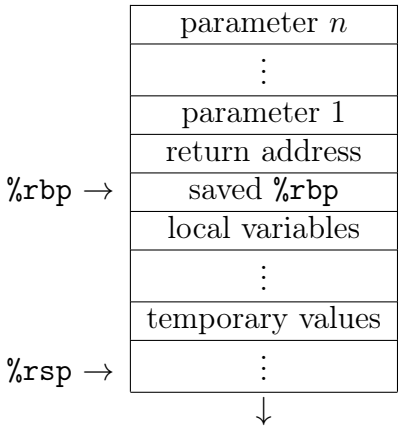
- get rid of multiple assignments;
- get rid of multiple returns;
- allocate all structures on the heap;
- allocate on the heap any variable whose address is taken with `&`.

(This is the purpose of module `Rewrite`, already implemented and used in `minigo.ml`.)

Value Representation. We propose a simple compilation scheme but you are free to use any other. A value of type `int`, `bool`, `string`, or $*\tau$ is a 64-bit word. The value `nil` is the integer 0. The values `false` and `true` are the integers 0 and 1, respectively. A string is a pointer to a 0-terminated string allocated on the data segment (since it is not possible to build strings dynamically in our fragment).

A structure is a heap-allocated block of a size large enough to accommodate all fields. Your compiler maintains, for each field, its position within the structure, relative to the start of the structure. Note that structures can be nested.

Stack Layout. We suggest a compilation scheme where all parameters are passed on the stack (each of them being a 64-bit word, since structures are never passed), and where the return value is in register `%rax` (again, no structure is returned). The stack frame is depicted beside. Local variables are allocated on the stack. (But note that no structure is allocated on the stack, thanks to the rewriting performed prior to code generation; see above.) The top of the stack is used to store intermediate computations, such as the value of the first operand during the evaluation of a binary operation.



The assembly code generated by your compiler should look like:

```
        .text
        .globl main
main:   call of Mini Go function main
        xorq %rax, %rax
        ret
...     code of Mini Go functions
...     assembly functions, if needed
        .data
...     string literals
```

Stack alignment. With recent versions of the `libc`, it is important to have a 16-byte stack alignment when calling library functions such as `malloc` or `printf` (this is required by the System V Application Binary Interface). Since it is not always easy to ensure stack alignment when calling library functions (because of intermediate computations temporarily stored on the stack), it may be convenient to introduce wrappers around library functions, as follows:

```
malloc_
      pushq   %rbp
      movq    %rsp, %rbp
      andq    $-16, %rsp  # 16-byte stack alignment
      call    malloc
      movq    %rbp, %rsp
      popq    %rbp
      ret
```

These wrappers are simply concatenated to the generated assembly code — and of course any call to `malloc` is replaced with a call to `malloc_`.

Here is a list of functions from the C standard library that you may want to use (feel free to use any other):

`void *malloc(size_t size);`
`malloc(`n`)` returns a pointer to a freshly heap-allocated block of size n.
You don't have to free memory.

`void *calloc(size_t nmemb, size_t size);`
`calloc(`n`, `s`)` returns a pointer to a freshly heap-allocated block of size $n \times s$.
The memory is set to zero. *You don't have to free memory.*

`int printf(const char *format, ...);`
`printf(`f`,...)` writes to standard output according to the format string
(ignore the return value). *Register `%rax` must be set to zero before calling `printf`.*

Important Notice. Grading involves (for one part only) some automated tests using small Go programs with `fmt.Print` commands. They are compiled with your compiler, and the output is compared to the expected output. This means you should be careful in compiling calls to `fmt.Print`.

4 Project Assignment (due March 15, 6pm)

The project must be done **alone or in pair**. It must be delivered on Moodle, as a compressed archive containing a directory with your name(s) (*e.g.* `dupont-durand`). Inside this directory, source files of the compiler must be provided (no need to include compiled files). The command `make` must create the compiler, named `minigo`. The compilation may involve any tool (such as `dune` for OCaml) and the `Makefile` can be as simple as a call to such a tool. The command `minigo` may be a script to run the compiler, for instance if the compiler is not implemented in OCaml.

The archive must also contain **a short report** explaining the technical choices and, if any, the issues with the project and the list of whatever is not delivered. The report can be in format ASCII, Markdown, or PDF.

The command line of `minigo` accepts an option (among `--debug`, `--parse-only` and `--type-only`) and exactly one file with extension `.go`. If the file is parsed successfully, the compiler must terminate with code 0 if option `--parse-only` is on the command line. Otherwise, the compiler moves to static type checking. Any type error must be reported as follows:

```
file.go:4:6:
bad arity for function f
```

The location indicates the filename name, the line number, and the column number. Feel free to design your own error messages. The exit code must be 1.

If the file is type-checked successfully, the compiler must exit with code 0 if option `--type-only` is on the command line. Otherwise, the compiler generates x86-64 assembly code in file `file.s` (same name as the input file, but with extension `.s` instead of extension `.go`). The x86-64 file will be compiled and run as follows

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

possibly with option `-no-pie` on the `gcc` command line. The output must be identical to that of the command

```
go run file.go
```

For your convenience, the expected output is provided in `file.out` for each test file.

Tips. It is strongly recommended to proceed construction by construction, whether for typing or code generation, in this order: output, arithmetic, local variables, functions, structures.