# Compilation Project - Part 1: PetitGo

Megi Dervishi

December 8, 2019

## 1 Organisation

The project is written in `Ocaml` using also `Ocamllex` and `Menhir`. The github page for the project can be seen in this link. The project is composed by the below files

**main** Execution of the program

**go_ast** Declaratation of the abstract syntax used during parsing

**go_parser , go_lexer** Parsing and lexing

**go_typer** Typing functions

**Makefile**

The project contains as well the folder `tests` and inside of it the automatic testing script `tester.sh`.

### 1.1 Compiling the project

There are three options that the program could use before compiling:

**-v** Mode verbeux.

**-parse-only** L'éxécution s'arrête après le parsage.

**-type-only** L'éxécution s'arrête après le typage.

In order to test a particular function type it (copy/paste) in the file `test.go` and run `make`. To run the automatic tests then execute

```
make tests
```

# 2 Implementation

## 2.1 Abstrac Syntactic tree

As previously mentioned the abstract syntax is defined in the file `go_ast.ml` following the rules explained in the project. In particular below there are the representation of the expression and instruction types,

```
and constant =
| Eint64   of int64
| Estring  of string
| Ebool    of bool
| Econst   of constant
| ENil

and expr =
| Econst   of constant
| Eident   of string
| Emethod  of expr loc * ident loc
| Eprint   of expr loc list
| Ecall    of ident loc * expr loc list
| Eunop    of unop * expr loc
| Ebinop   of binop * expr loc * expr loc

and instr =
  Inop
| Iexpr   of expr loc
| Iasgn   of expr loc * expr loc
| Iblock  of instruction list
| Idecl   of ident loc list * ty loc * expr loc
| Ireturn of expr loc
| Ifor    of expr loc * instr
| Iif     of expr loc * instr * instr
```

Furthermore to handle the localisation of types I introduced `'a loc` which is helps in keeping track of the position of the particular type.

```
type tuple = Lexing.position * Lexing.position
type 'a loc = 'a * tuple
```

## 2.2 Lexer and Parser

The lexer and parser were also implemented as on the rules described in the subject report. In the lexer there are two hash-tables one of which implements the tokens that require a semicolon after their calling and the other that do not. Then I use the reference bool `semicolon` which is `true` if semicolon is required and the function `check_semicolon` which updates the pointer after an empty

line. In the parser there are certain helpful functions (`blabla` ) which were implemented to `option` types i.e. supporting `None` as part of their syntax.

### 2.3 Typing

Certainly the typer was more challenging than the parser and lexer. The typer has the following main/representative types:

```
type typ =
| Tint
| Tbool
| Tstring
| Tstruct of ident
| Tstar   of typ
| Tnone
| Tvoid   (* used for "_" *)

type gotype =
| Tsimpl  of typ
| Tmany   of gotype
```

The environment stores the struct, functions and variables of the program in the following way.

```
type tstruct = (typ Smap.t ) Smap.t
type tfunct  = (gotype * gotype) Smap.t
type tvars   = typ Smap.t
type typenv  = { structs: tstruct; funct : tfunct; vars : tvars }
```

The most important typer functions are the following:

```
1) type_expr:  typenv -> 'a * 'b -> gotype * 'c * 'b * bool
2) type_instruction : 'a -> 'b -> 'c -> 'a * 'd * bool * bool
3) add_functions_to_env : typenv -> 'a -> typenv
4) add_struct_to_env:  typenv -> '_a -> typenv
5) type_prog:  '_a list -> typenv * '_b lis
```

For `type_expr` if the bool is true then the expression was a left value. For `type_instruction` there are two bool types. The first one is the `return_bool` which is `true` if there was a `return` in the instructions and the second one is `print_bool` which is `true` if there was a `Print` in the instructions.

## Improvements

Well the first improvement from this moment is to make a fully functioning Typer which passes all the tests. Furthermore at this moment I am mostly using the exception of `Typing_error`. My goal is create a much more eloquent

error handling. Consequently also would like to create better printing functions for the error messages which are more precise in localising/visualizing where the error is to the user. For example:

```
1:
2: func f() { var x = &3 }
   -------------------^---
```

## Conclusion

Overall the project was an interesting challenge which sometimes was a bit difficult as I have learned to code in `Ocaml` only recently and do not posses yet all the "programming tricks". My parser passed all the automatic tests which I am very happy about. The Typer was definetly more difficult especially since I had many small errors with the `loc` which were not allowing me to compile the Typer. Nonetheless I will continue working on it and improving as mentioned above.