# µGo: A Simple Go Programming Language

**Compiler 2022 Programming Assignment II**
**Syntactic and Semantic Definitions for µGo**
**Due Date: May 12, 2022 at 23:59**

Your assignment is to build the parser for the µGo language that supports print IO, arithmetic operations and some basic constructs for µGo. You will have to define the token classes and the grammar for µGo using the given Lex and Yacc codes, respectively, to produce the parser. You are welcome to make any changes of the given codes to meet your expectations. In addition to the µGo syntax, your produced parser will do simple checking for semantic correctness of the testing cases.

## 1. Yacc Definitions

In the previous assignment, you have built the Lex code to split the input text stream into tokens that will be accepted by Yacc. For this assignment, you must build the code to analyze these tokens and check the syntax validity based on the given grammar rules.

Specifically, you will do the following three tasks in this assignment.

1. Define tokens and types (Section 1.1)
2. Design µGo grammar and implement the related actions (Section 1.2)
3. Handle semantic errors (Section 1.3)

### 1.1 Define Tokens and Types

#### 1.1.1 Tokens

The tokens need to be defined in both Lex and Yacc code. Lex recognizes a token when it gets one, and Lex forwards the occurrence of the token to Yacc. You should make sure the *consistency* of the token definitions in both Lex and Yacc code. You are welcome to add/modify the token definitions in the given Lex code.

Some tips for token definition (in Yacc) are listed below:

- Declare tokens using `%token`.
- The name of grammar rule, which is not declared as a token, is assumed to be a nonterminal.

#### 1.1.2 Types

Type refers to one of the µGo data types: **integer**, **float**, **string** and **boolean**. Useful tips for defining a type are listed below.

- Define a type for `yylval` using `%union` by yourself. For example, `%union { int i_val; }` means `yylval` is able to be accessed via the `int` type using the `i_val` variable.
- Define a type for token using `%type` and give the type name within the less/greater than symbols, `<` / `>`; for example, `%type <i_val> INT_LIT` means the token `INT_LIT` has the `int` type.

```
%union {
    int i_val;
    float f_val;
    char* s_val;
}
%type <i_val> INT_LIT
%type <f_val> FLOAT_LIT
%type <s_val> STRING_LIT
```

## 1.2 Design Grammar and Implement Actions

### 1.2.1 Grammar

The concept of **CFG (context-free grammar)** that you learned in the courses should be used to design the grammar for print IO, arithmetic operations and basic constructs. The conversion from the productions of a CFG to the corresponding Yacc rules is illustrated as below.

- Grammar productions for A:
  $A \rightarrow B_1 B_2 \ldots B_m$
  $A \rightarrow C_1 C_2 \ldots C_n$
  $A \rightarrow D_1 D_2 \ldots D_k$

- Yacc rules:
  $A$
  : $B_1 B_2 \ldots B_m$
  | $C_1 C_2 \ldots C_n$
  | $D_1 D_2 \ldots D_k$
  ;

**Hint:** The link is ANSI C grammar rules, you could design your parser grammar base on it.

### 1.2.2 Actions

An action is C statement(s) that should be performed as soon as the parser recognizes the production rule from the input stream. The C code surrounded by `{` and `}` is able to handle input/output, call sub-routines, and update the program states. Occasionally, it is useful to put an action *in the middle of a rule*. The following code snippet shows that integer literal will be printed out after token `INT_LIT` is recognized.

```
literal
    : INT_LIT   { printf("type %s value %d", "int32", $<i_val>1); }
    | FLOAT_LIT { printf("type %s value %f", "float32", $<f_val>1); }
;
```

### 1.3 Handle Semantic Errors

Your Yacc code needs to detect semantic errors during parsing the given µGo code. When errors occur, your parser should detect and display error messages upon the termination of the parsing procedure. The messages will include the *type* of the semantic error and the *line number* of the code that causes the error.

To be precise, in this assignment, you should at least handle the following four cases:

1. Variable errors:
    - Operate on *any* undeclared variable
    - Re-define *any* existed variable
2. Type errors:
    - Handle modulo operation ( `%` ) involving any floating point number or variable
    - Handle simple type checking for the mismatching (e.g., `3 + 3.14` ) and the condition of "if" and "for" statements that the values must be the *boolean* type

**Hint:** Your `%union` may need to use `struct` with some fields to record the types of the value.

# 2. Symbol Table

### 2.1 Functions

Symbol table needs to be built in the Yacc program so as to perform the following tasks.

1. Create a symbol table when entering a new scope. `create_symbol`
2. Insert an entry for a variable declaration. `insert_symbol`
3. Look up an entry in the symbol table. `lookup_symbol`
4. Dump all contents in the symbol table of current scope and its entries when exiting a scope. `dump_symbol`

**Hint:** You may add some data fields in the table to facilitate semantic error handling or scoping check.

**Hint:** You may need to link and organize multiple tables as the operation of a stack.

## 2.2 Scope Level

The global scope level is zero and is increased by one when entering a new block. When the program leaving a block, you need to dump the symbol table of current level then decrease the level by one. You can find the example at section 2.4 below.

Note that the level indecate the depth of the scope; however the scopes with the same level cannot communicate with each other. For example, in example 2.4, the scope levels of A and C are 2, but we cannot access the floating-point veriable `width` in C.

## 2.3 Table Fields

The structure of the example symbol table is listed below:

- Index: the variable index in attached symbol table, and should be unique in that symbol table.
- Name: the name of the variable.
- Type: the type of the variable.
- Addr: abbreviation of address, it should be unique in the whole program. Note that the Addr of function is always `-1`.
- Lineno: the line number where define the variable.
- Func_sig: the function signature contains return type and type of parameters. For example, the prototype `func foo(x int32, y float32, g int32) int32` should be recorded as `(IFI)I` in Func_sig.

| Index | Name | Type | Addr | Lineno | Func_sig |
|---|---|---|---|---|---|
| 0 | x | int32 | 0 | 1 | - |
| 1 | y | float32 | 1 | 2 | - |

## 2.4 Symbol Table Example

- Example µGo code:

```
package main

func main() {
    var height int32 = 99
    {
        var width float32 = 3.14
    } // A. Exit... dump scope level 2
    var length float32
    {
        var length string = "hello world"
        {
            var length bool
        } // B. Exit... dump scope level 3
        var width int32 = 66
    } // C. Exit... dump scope level 2
} // D. Exit... dump scope level 1
// E. Exit... dump scope level 0
```

- Example output of the symbol table:

```

Scope level: 2
Index     Name      Type      Addr      Lineno    Func_sig
0         width     float32   1         6         -

Scope level: 3
Index     Name      Type      Addr      Lineno    Func_sig
0         length    bool      4         12        -

Scope level: 2
Index     Name      Type      Addr      Lineno    Func_sig
0         length    string    3         10        -
1         width     int32     5         14        -

Scope level: 1
Index     Name      Type      Addr      Lineno    Func_sig
0         height    int32     0         4         -
1         length    float32   2         8         -

Scope level: 0
Index     Name      Type      Addr      Lineno    Func_sig
0         main      func      -1        3         ()V
```

## 3. What Should Your Parser Do?

**Each test case is 10 and the total score is 110pt**

1. Handle arithmetic operations, where brackets and precedence should be considered. (20pt, `in01`, `in02`)
2. Implement the scoping check function in your parser. To get the full credits for this feature, your parser is expected to correctly handle the scope of the variables defined by the µGo language. (10pt, `in03`)
3. Support the variants of the assignment operators. (i.e., `=`, `+=`, `-=`, `*=`, `/=`, `%=`) (10pt, `in04`)
4. Handle the type conversion between integer and floating-point. (10pt, `in05`)
5. Support if statements. (10pt, `in06`)

6. Support for statements. (10pt, `in07`)
7. Detect semantic error(s) and display the error message(s). The parser should display at least the error type and the line number. (20pt, `in08`, `in09`)
8. Support function define and function call. (10pt, `in10`)
9. Support switch statements. (10pt, `in11`)

## 3.1 Example

Example input code and the expected output from your parser.

- Input #1:

```go
package main
func main() {
  var sum int32 = 0
  var i int32
  for i = 0; i <= 10; i++ {
      sum += i
  }
  println(sum) // 55
}
```

- Output #1:

```
> Create symbol table (scope level 0)
package: main
func: main
> Create symbol table (scope level 1)
func_signature: ()V
> Insert `main` (addr: -1) to scope level 0
INT_LIT 0
> Insert `sum` (addr: 0) to scope level 1
> Insert `i` (addr: 1) to scope level 1
IDENT (name=i, address=1)
INT_LIT 0
ASSIGN
IDENT (name=i, address=1)
INT_LIT 10
LEQ
IDENT (name=i, address=1)
INC
> Create symbol table (scope level 2)
IDENT (name=sum, address=0)
IDENT (name=i, address=1)
ADD

> Dump symbol table (scope level: 2)
```

```
Index     Name      Type      Addr      Lineno    Func_sig

IDENT (name=sum, address=0)
PRINTLN int32

> Dump symbol table (scope level: 1)
Index     Name      Type      Addr      Lineno    Func_sig
0         sum       int32     0         3         -
1         i         int32     1         4         -


> Dump symbol table (scope level: 0)
Index     Name      Type      Addr      Lineno    Func_sig
0         main      func      -1        2         ()V

Total lines: 10
```

- Input #2 (with error):

```
package main
func main() {
  var y float32
  x + y
  100 % y
}
```

- Output #2:

```
> Create symbol table (scope level 0)
package: main
func: main
> Create symbol table (scope level 1)
func_signature: ()V
> Insert `main` (addr: -1) to scope level 0
> Insert `y` (addr: 0) to scope level 1
error:4: undefined: x
IDENT (name=y, address=0)
error:4: invalid operation: ADD (mismatched types ERROR and float32)
ADD
INT_LIT 100
IDENT (name=y, address=0)
error:5: invalid operation: (operator REM not defined on float32)
REM

> Dump symbol table (scope level: 1)
Index     Name      Type      Addr      Lineno    Func_sig
0         y         float32   0         3         -
```

```
> Dump symbol table (scope level: 0)
Index     Name      Type      Addr      Lineno    Func_sig
0         main      func      -1        2         ()V

Total lines: 7
```

## 4. Submission

- Hand in your homework with Moodle.
- Allow only `.zip` formats for file compression.
- The directory organization should be exactly as follows.

```
Compiler_StudentID_HW2.zip/
└── Compiler_StudentID_HW2/
    ├── compiler_hw2.l
    ├── compiler_hw2.y
    ├── compiler_hw_common.h
    └── Makefile
```

**!!! Incorrect format will lose 10pt. !!!**

## 5. Appendix: µGo Specification

In this specification, the syntax is specified using Extended Backus-Naur Form (EBNF). The following
table lists the operators defined in EBNF.

```
|   alternation
()  grouping
[]  option (0 or 1 times)
{}  repetition (0 to n times)
```

### 5.1 Types

A type determines a set of values together with operations and methods specific to those values.

- "int32": the set of all signed 32-bit integers (-2147483648 to 2147483647)
- "float32": the set of all IEEE-754 32-bit floating-point numbers
- "string": a (possibly empty) sequence of bytes
- "bool": the set of Boolean truth values denoted by the predeclared constants true and false

```
Type      = "int32" | "float32" | "string" | "bool"
```

## 5.2 Expressions

```
Expression = UnaryExpr | Expression binary_op Expression
UnaryExpr  = PrimaryExpr | unary_op UnaryExpr

binary_op  = "||" | "&&" | cmp_op | add_op | mul_op
cmp_op     = "==" | "!=" | "<" | "<=" | ">" | ">="
add_op     = "+" | "-"
mul_op     = "*" | "/" | "%"

unary_op   = "+" | "-" | "!"
```

**Note:** Arithmetic operations are written in infix notation, and the precedence for the operators is defined as below (the smaller number has the higher precedence).

| Category | Operators | Precedence |
|---|---|---|
| Unary | `+` `-` `!` | 1 |
| Multiplication | `*` `/` `%` | 2 |
| Addition | `+` `-` | 3 |
| Comparison | `<` `>` `<=` `>=` `==` `!=` | 4 |
| Logical AND | `&&` | 5 |
| Logical OR | `\|\|` | 6 |

**Note:** 1. The expression within `()` needs to be evaluate first. 2. `++` `--` `=` `+=` `-=` `*=` `/=` `%=` are invalid in an expression. 3. `!`, `&&`, and `||` are only for boolean type and boolean type does not support arithmetic operation, e.g., multiplication, addition, and comparison. 4. `%` is only for integer numbers.

### Primary expressions

```
PrimaryExpr = Operand | IndexExpr | ConversionExpr
Operand     = Literal | identifier | "(" Expression ")"
Literal     = int_lit | float_lit | bool_lit | string_lit
```

### Conversions (Type casting)

A conversion changes the type of an expression to the type specified by the conversion.

```
ConversionExpr = Type "(" Expression ")"
```

Example:

```
int32(3.2)
float32(x + 3)
```

## 5.3 Statements

```
Statement =
    DeclarationStmt NEWLINE
    | SimpleStmt NEWLINE
    | Block
    | IfStmt
    | ForStmt
    | SwitchStmt
    | CaseStmt
    | PrintStmt NEWLINE
    | ReturnStmt NEWLINE
    | NEWLINE

SimpleStmt = AssignmentStmt | ExpressionStmt | IncDecStmt
```

### Declarations statements

A variable declaration creates one variables, binds corresponding identifiers to them, and gives a type and an initial value.

```
DeclarationStmt = "var" identifier Type [ "=" Expression ]
```

Example:

```
var i int32
var k float32 = 3.14
```

### Assignments statements

Each left-hand side operand must be addressable.

```
AssignmentStmt = Expression assign_op Expression
assign_op      = "=" | "+=" | "-=" | "*=" | "/=" | "%="
```

Example:

```
a = 99
b -= c + a
```

### Expression statements

In µGo, an expression statement is composed of an Expression, which consists of several expressions with operators (Section 5.2).

```
ExpressionStmt = Expression
```

Example:

```
x + y - z * 10 + 100 / 5
```

```
x >= y && x <= z
```

### IncDec statements

The "++" and "--" statements increment or decrement their operands by the untyped constant 1. As with an assignment, the operand must be addressable. You can assume that the Expression in this statement must be an identifier in our assignment, i.e., valid statements are like `x++`, `i--`, and `3++`.

```
IncDecStmt = Expression ( "++" | "--" )
```

Example:

```
x++
y--
```

## Block

A block is a possibly empty sequence of declarations and statements within matching brace brackets.

```
Block         = "{" StatementList "}"
StatementList = { Statement }
```

Example:

```
{
    x += y
    y--
}
```

## If statements

"If" statements specify the conditional execution of two branches according to the value of a boolean expression. If the `Condition` evaluates to true, the "if" branch is executed, otherwise, if present, the "else" branch is executed.

```
IfStmt = "if" Condition Block [ "else" ( IfStmt | Block ) ]

Condition = Expression
```

Example:

```
if x > max {
    max = x
}
```

```
if (x < max) {
    x = max
} else {
    max = x
}
```

```
if (x <= y) {
    x++
} else if (x > z) {
    y++
} else {
    z++
}
```

### For statements

A "for" statement specifies repeated execution of a block. There are two forms: the iteration may be controlled by a single condition or a "for" clause.

```
ForStmt   = "for" ( Condition | ForClause ) Block

ForClause = InitStmt ";" Condition ";" PostStmt
InitStmt  = SimpleStmt
PostStmt  = SimpleStmt
```

Example:

```
for a < b {
    a *= 2
}
```

```
for i = 0; i < 10; i++ {
    sum += i
}
```

### Switch statements

A "switch" statement is a shorter way to write a sequence of `if-else` statements. It runs the first case whose value is equal to the condition expression.

```
SwitchStmt = "switch" Expression Block
CaseStmt = ( "case" INT_LIT | DEFAULT ) ":" Block
```

Example:

```
switch num {
    case 0: {
        println("A")
    }
    default: {
        println(num)
    }
}
```

### Function

A declaration begins with the `func` keyword, followed by the name you want the function to have, a pair of parentheses (it takes zero or more arguments), and then a block containing the function's code.

```
GlobalStatement = PackageStmt NEWLINE | FunctionDeclStmt | NEWLINE
PackageStmt = PACKAGE IDENT

FunctionDeclStmt = FuncOpen "(" ParameterList ")" ReturnType FuncBlock
ParameterList = [ IDENT Type | ParameterList "," IDENT Type ]
FuncBlock = "{" StatementList "}"
ReturnStmt = RETURN [Expression]
```

Example:

```
package main

func foo() {
    println("foo")
    return
}
func main() {
    foo()
    return
}
```

### Print statements

Implementation restriction: "print" and "println" need not to accept arbitrary argument types, but printing of boolean, numeric, and string types must be supported.

- "print": prints evaluated expression.
- "println": like print but prints a newline at the end.

```
PrintStmt = ( "print" | "println" ) "(" Expression ")"
```

Example:

```
print(x)
println(y + x * z)
```

## 6. References

- The Go (not μGo) Programming Language Specification: https://golang.org/ref/spec
- ANSI C Yacc grammar: http://www.quut.com/c/ANSI-C-grammar-y.html