µGo: A Simple Go Programming Language Specification

Compiler Construction, Spring 2018

This document is ***µGo*** language specification, which is a simplified language base on ***Go***. The µGo inherits basic features in Go, but remove the complex features. This file presents the language specification (e.g., operators and syntaxes) that your parser should implement; in other words, your scanner and parser do not have to handle the undefined behaviors.

1. The Basic Components in µGo

In general, a µGo program consists of the following parts: Variables, Declarations, Statements and Expressions, Comments. In the remainder of this document, we introduce the notation that we used for defining the language (in 2. Representation of the Syntax in µGo). We introduce the details of µGo. In particular, we define the types of tokens that your scanner should identify (in 3. Lexical Element), the data types for µGo (in 4. Types), the code blocks (in 5. Blocks), the declarations of identifiers and the scope of a declared identifier (in 6. Declarations and Scope), the computation of a value to operands (in 7. Expressions), the code statements (in 8. Statement), the built-in functions that are supported natively by µGo (in 9. Built-in Functions) and the operator precedence (in 10. Operator Precedence).

2. Representation of the Syntax in µGo

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)
```

The following terms are used to denote specific Unicode character and decimal digits.

```
decimal_digit    =   "0" … "9"
unicode_letter   =   /* a Unicode code point classified as "Letter" */
unicode_char     =   /* an arbitrary Unicode code point except newline */
letter           =   unicode_letter | "_"
```

3. Lexical Element

• Identifiers

Identifiers name program entities such as variables and types. An identifier is a sequence of one or more letters and digits. The first character in an identifier must be a letter.

```
identifier    =    letter { letter | decimal_digit }
```

• Keywords

The following keywords are reserved and may not be used as identifiers.

```
else, if, for, var
```

• Predeclared identifiers

The following identifiers are implicitly declared by default in μGo programs.

```
print, println
```

• Comments

Comments serve as program documentation. There are two forms:

- Line comments start with the character sequence // and stop at the end of the line.
- General comments start with the character sequence /* and stop with the first subsequent character sequence */.

• Operators and punctuation

The following character sequences represent operators (including assignment operators) and punctuation.

```
Arithmetic    +     -     *     /     %     ++     --
Relational    <     >     <=    >=    ==    !=
Assignment    =     +=    -=    *=    /=    %=
Logical       &&    ||
```

- **Integer literals**

An integer literal is a sequence of digits representing an integer constant.

```
decimal_lit    =    ( "1" … "9" ) { decimal_digit }
```

Example:

```
42
1234567890
17014118346046923173
```

- **Floating-point literals**

A floating-point literal is a decimal representation of a floating-point constant. It has an integer part, a decimal point and a fractional part. The integer and fractional part comprise decimal digits.

```
decimals    =    decimal_digit { decimal_digit }
float_lit    =    decimals "."  decimals
```

Example:

```
4.2
1.234567890
170141.18346046923173
```

- **String literals**

A string literal represents a string constant obtained from concatenating a sequence of characters.

```
string_lit    =    “\”” { unicode_char } “\””
```

Example:

```
“Hello, world!”
“Hey, world.”
```

4. Types

- **Numeric types**

A numeric type represents sets of integer or floating-point values. The predeclared architecture-

independent numeric types are as below.

```
Type     =  int  |  float32
int      =  /* the set of all signed 32-bit integers (-2147483648 to 2147483647) */
float32  =  /* the set of all IEEE-754 32-bit floating-point */
```

5. Blocks

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

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

In addition to explicit blocks in the source code, there are implicit blocks:

- The *universe block* encompasses all Go source text.
- Each "if" and "for" statement is considered to be in its own implicit block.
- Blocks nest and influence scoping.

6. Declarations and Scope

• Variable declarations

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

```
Declaration  =  "var" VarSpec
VarSpec      =  Type [ "=" Expression ]
```

Example:

```
var i int
var k float32 = 0
```

7. Expressions

• Arithmetic operators

Arithmetic operators apply to numeric values and yield a result of the same type as the first operand. The four standard arithmetic operators (+, -, *, /) apply to integer and floating-point.

```
 +    sum             integers, floats
 -    difference      integers, floats
 *    product         integers, floats
 /    quotient        integers, floats
 %    remainder       integers
```

- **Comparison operators**

Comparison operators compare two operands and yield an untyped boolean value.

```
==      equal
!=      not equal
 <      less
<=      less or equal
 >      greater
>=      greater or equal
```

- **Logical operators**

Logical operators apply to boolean values and yield a result of the same type as the operands. The right operand is evaluated conditionally.

```
&&   conditional AND      E.g., p && q is  "if p then q else false"
 ||   conditional OR      E.g., p || q  is  "if p then true else q"
```

The syntaxes for the above operators are as below.

```
Expression    =  Operand  |  Expression  binary_op  Expression

Operand       =  Literal  |  identifier  |  "("  Expression  ")"
Literal       =  int_lit  |  float_lit  |  string_lit

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

8. Statement

- **Expression statements**

In μGo, an expression statement means single identifier, arithmetic, comparison and logical operations.

```
ExpressionStmt = Expression
```

Example:

```
x + y - z * 10 + 100 / 5
x >= y
x || y - 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 or a map index expression.

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

The following assignment statements are semantically equivalent:

```
IncDec Statement        Assignment
x++                     x += 1
x--                     x -= 1
```

- **Assignments statements**

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

Each left-hand side operand must be addressable. Example:

```
x = 10
x += y + 10
x %= 200
```

- **For statements**

A "for" statement specifies repeated execution of a block.

```
ForStmt = "for" Expression Block
```

Example:

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

- **If statements**

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

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

Example:

```
case1   if x > max {
          x = max
        }

case2   if (x > max) {
          x = max
        } else {
          max = x
        }

case3   if (x < y) {
          x++
        } else if (x > z) {
          y++
        } else {
          z++
        }
```

- **Print statements**

Please refer to the Built-in function for details.

The above statements are represented as follows.

```
Statement  =  ExpressionStmt | IncDecStmt | Assignment | ForStmt | IfStmt
              | PrintStmt | PrintlnStmt
```

9. Built-in Functions (i.e., Print Statements)

Built-in functions are predeclared. They are called like any other function but some of them accept a type instead of an expression as the first argument.

In μGo, the built-in functions are defined as the print statement.

```
PrintStmt    =  “print” “(” string_lit “)”
PrintlnStmt  =  “println” “(” string_lit “)”
```

print	prints all arguments; formatting of arguments is implementation-specific
println	like print but prints spaces between arguments and a newline at the end

Example:

```
print (“Hello world !”)
println (“Hello world !”)
```

10. Operator Precedence

There are six precedence levels for operators and the precedence level 6 is of the highest precedence. Postfix operators bind more tightly than others, i.e., multiplication, addition, comparison, logical AND (&&), and logical OR (||).

Category	**Precedence**	**Operator**
postfix	6	++ --
multiplication	5	* / %
addition	4	+ -
comparison	3	== != < <= > >=
logical AND	2	&&
logical OR	1	||