The Go Programming Language Specification

Group Members:
Shubh Agrawal 190828
Shubhankar Gambhir 190835
Tarun Kanodia 190902
V Pramodh Gopalan 190933

Group - 24

Source language: Go
Implementation language: C++
Target language: x86 assembly

The Go Programming Language Specification

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 in Golang.

```
identifier = letter { letter | unicode_digit } .
```

Native Data types (integer, boolean, character)

- byte (used as char)
- bool
- int

Keywords

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

```
break      func      struct      else      package      if
continue   for       import      return    var          new
const
```

Handling I/O in Go

Input is taken from `STDIN` using `"fmt.Scanln"` while output is printed to `STDOUT` using `"fmt.Println".` Formatted input can be taken using “`fmt.Scanf`” and Formatted output can be printed using “`fmt.Printf`”.

```
//following is code to print something onto the output stream
package main
import "fmt" // need to import this library
func main() {
    fmt.Println("!... Hello World ...!")
}
```

```
// Printing Variables
var first int = 5
fmt.Println(first)
// Taking Inputs
```

```
var first int
fmt.Scanln(&first)
fmt.Scanf(“%d”,&first)
// %d is for int , %c is for byte , %t is for bool
```

Variable declaration -

Variables can be declared in Go using the following syntax:

```
// declaring and initializing the variable
var a int = 30

// declaring and initializing the variable  of char type
var my_char byte = 'a'
 var b bool = true
```

Shorthand declarations are also supported in Go:

```
// declaring and initializing the variable and is of type int
    a:= 30
```

```
// declaring and initializing the variable  of boolean type
```

```
    b:= true
```

Expressions

Primary expressions:

Primary expressions are the operands for unary and binary expressions.
Example: x, 2, `f(3.1415, true) etc.`

Operators:

Go supports all standard arithmetic operators like addition, subtraction, multiplication, division
The expressions are of the form:

```
Expression = UnaryExpr | Expression binary_op Expression
```

where `binary_op` denotes the binary operators

`List of unary operators: "+" , "-" , "!" , "^" , "*" , "&"`
`List of binary operators: "||" , "&&" , "==" , "!=" , "<" , "<=" , ">" , ">=" , "+" , "-" , "|" , "^", "*" , "/" , "%" , "<<" , ">>" , "&".`

Operator precedence:

Unary operators have the highest precedence. As the **`++`** and **`--`** **o**perators form statements, not expressions, they fall outside the operator hierarchy. As a consequence, statement **`*p++`** is the same as **`(*p)++.`**
There are five precedence levels for binary operators. Multiplication operators bind strongest, followed by addition operators, comparison operators, **`&& (logical AND),`** and finally **`|| (logical OR).`**

Conditionals

Single `if` condition:

```
var v int = 700
     if(v < 1000) {
         // print the following if condition evaluates to true
        fmt.Printf("v is less than 1000\n")
    }
```

if else condition

```
var v int = 700
     if(v < 500) {
         // print the following if condition evaluates to true
        fmt.Printf("v is less than 500\n")
    } else if (v<=1000){
   // print the following if we have the if condition evaluates
   //to false and else if condition evaluates to true
        fmt.Printf("v is less than 1000\n")
    }
```

if-else if-else Condition

```
 var v int = 700
  if(v < 500) {
        fmt.Printf("v is less than 500\n")
    } else if (v<=600){
        fmt.Printf("v is less than 600\n")
    } else {
         // print the following if condition and the else if
          //condition evaluates to true
         fmt.Printf(" yoyo")
    }
```

Loops:

General:

```
for [condition |  ( init; condition; increment )] {
    // statements
}
```

for loops:

```
for i := 0; i < 4; i++{
    // statements
 }
```

The above code runs all the statements inside the for loop 4 times.

while loops:

```
 j:=0
 for j<10{
         fmt.Println(j)
         j+=1
  }
```

The above code prints all numbers from 0 to 9, each number on new line.

break statements:

```
j:=0
for true {
     fmt.Println(j)
      j+=1
     if (j==5){
      // when j==5, this code segment is executed,
      // thus exiting the loop
         break
     }
}
```

The above code prints all numbers from 0 to 4, each number on new line.

continue statements:

```
j:=0
for j<=5 {
       j+=1
       if(j==2){
        // when j==2, this code segment is executed, thus
        // skipping the later instruction(print instruction here)
        // for this iteration of the loop
             continue
       }
       fmt.Println(j)
}
```

Array:

General:

```
var variable_name[SIZE] variable_type
```

Example:

```
var myarr[3] int
// Elements are assigned using index
myarr[0] = 561
myarr[1] = 872
myarr[2] = 1289
// Accessing the elements of the array using index value
fmt.Println("Elements of Array:")
fmt.Println("Element 1: ", myarr[0])
fmt.Println("Element 2: ", myarr[1])
fmt.Println("Element 3: ", myarr[2])
```

```
--------Output-----------
Elements of Array:
Element 1:  561
Element 2:  872
Element 3:  1289
```

Functions:

General:

```
func <name of function> (params, ... ) <return type>{
      // statements
}
```

params is of the form = `<variable name> <variable type>`

Example:

```
func area(length int, width int)int{
    Ar := length* width
    return Ar
}
```

This function takes 2 inputs which are length and width both of type integer and it just multiplies and returns the result.

Structures:

Declaration:

General:

```
type <struct name> struct{
      Member1 datatype
     Member2 datatype
     Member3 datatype
      ...
}
```

Example:

```
type Point struct{
    X int
    Y int
    Z int
    label byte
}
```

Assigning values:

General:

```
var <var name> <struct name>
<var name>.<Member1> = val1
<var name>.<Member2> = val2
<var name>.<Member3> = val3
```

Example:

```
var a Point;
```

```
a.X = 0
a.Y = 1
a.Z = 2
a.label = ‘a’
```

Accessing values:

General:

```
<var name>.<Member>
```

Example:

```
fmt.Println("X: ", a.X)
fmt.Println("Y: ", a.Y)
fmt.Println("Z: ", a.Z)
fmt.Println("Label: ", a.label)
```

```
--------Output-----------
X:  0
Y:  1
Z:  2
Label:  a
```

Pointers:

Declaration:

General:

```
var <var name> *type
```

Example:

```
var a *int
```

Usage:

General:

```
*<var name> = val
```

Example:

```
var b int = 3                    //dummy value
```

```
*a = &b                     //assigned address of a variable
fmt.Printf("Address of b variable: %x\n", &b  )
fmt.Printf("Address stored in a variable: %x\n", a )
fmt.Printf("Value of *a variable: %d\n", *a )
```

```
--------Output-----------
Address of b variable: 11788000
Address stored in a variable: 11788000
Value of *a variable: 3
```

Multilevel pointers can also be declared in Go using automatic type inference. Example:

```
x := 10                    //dummy value
px := &x                   //assigned address of a variable
ppx := &px                 //assigned address of another pointer
```

Dynamic Memory Allocation:

Dynamic Memory can be allocated using “new” keyword.
Example: Dynamic memory allocation for struct:

```
type st struct{
    a int
    next *st
}

// Dynamic struct
 s := new(st)
```