CMSC 245 – Principles of Programming Languages
Lab#1: Go, code go!

Structure of a Go Program

The general structure of a Go program is shown below:

```
1  package main
2
3  <import packages>
4
5  func main() {
6    <code for main() goes here>
7 } // main()
```

Here are some salient features:

- **Line 1:** All main programs reside in a package `main`.
- **Line 3:** All packages used in the program below have to be imported. There are two ways of doing this (we will import the **`fmt`** and **`math`** packages):

  ```
  import "fmt"
  import "math"
  ```

  or, you can use:

  ```
  import (
     "fmt"
     "math"
  )
  ```

  Package names are enclosed in double-quotes, and listed in alphabetical order.

- **Lines 5-7:** Defines the `main()` function. This is where execution begins.

To run a Go program use the command:

```
> go run <filename>.go
```

Types and Operators

Go provides several built-in basic data types: numbers (integers, floating-point numbers), Booleans, and strings:

- **Numbers-integers:** The built-in types **`uint8`**, **`uint16`**, **`uint32`**, **`uint64`** represent 8, 16, 32, and 64-bit unsigned integers. The types **`int8`**, **`int16`**, **`int32`**, **`int64`** represent signed 8, 16, 32, and 64 bit integers. The type **`int`** represents integers whose size is machine-dependent (32 or 64 bit. On CS servers and on my Laptop, it is 64 bits. **`int`** is the type you will use for most common applications. The standard operators defined on integers are: **`+`**, **`-`**, **`*`**, **`/`**, and **`%`**.
- **Numbers-floating point numbers:** The built-in types **`float32`** and **`float64`** are used to define floating point variables. The standard operators defined on floating-point numbers are: **`+`**, **`-`**, **`*`**, and **`/`**.
- **Booleans:** The type **`bool`** is used to define Boolean values (**`false`**, **`true`**). The three standard logical operators available are: **`&&`** (and), **`||`** (or), and **`!`** (not).
- **Strings:** The type **`string`** is used to define strings. String literals are written enclosed in double-quotes `"Bryn Mawr", "Haverford"`, etc. The operations on string type include: **`+`** (contcatenation), **`s[i]`** (i-the character of a string **`s`**), and **`len(s)`** for finding out the length of a string **`s`**. Strings in Go can also be compared using **`<`**, **`<=`**, **`>`**, **`>=`**, **`==`**, and **`!=`**.

Variables and Declarations

Variable names: All variable names must start with a letter, and may contain letters, numbers, or _ (underscore). CamelCase (or camelCase) names are good practice. Although, Go assigns special meanings to names beginning with an uppercase letter. More on that later when we do packages.

Scope: Variables can be defined anywhere after the **`import`** statement. Go uses lexical scoping. A variable is visible after it is defined, in the block that it is defined.

There are many different ways to declare variables in Go.

1. **Basic: `var <name> <type>`**

   ```
   var n int
   var name string
   ```

   All variables are initialized by default: 0 for integers, 0.0 for floats, **`""`** for strings, and **`false`** for Boolean type.

2. **Initializing variables: `var <name> <type> = <value>`**

   ```
   var n int = 10
   ```

```
var name string = "Bryn Mawr"
```

3. **Type Inference: `var <name> = <value>`**

   ```
   var n = 10
   var name = "Bryn Mawr"
   ```

   If you define a variable with an initial value, Go infers its type using the value provided and you do not need to specify the type. Above, Go will infer **`n`** to be an **`int`** and **`name`** to be of type **`string`**.

4. **Multiple variables** can be defined on a single line:
   `var <name1>, name2>, … <type>`

   ```
   var n, m int
   ```

   Multiple variable can also de initialized:

   ```
   var n, m = 10, 20
   ```

   Above, **`n`** will be initialized to 10 and **`m`** to 20. Go will also infer their type to **`int`**. If needed, the type can also be specified.

5. **Group Definitions:** Go also provides a syntax for defining multiple variables in groups:

   ```
   var (
        n = 10
        name = "Bryn Mawr"
        rank int
   )
   ```

   Groups are often used to define related entities, like a group of constants, error codes, etc.

6. **Short-hand Declarations: `<var> := <value>`**

   n := 10

   Typically, short variable declarations are useful in defining loop control valriiables. They can also be used anywhere to define and initialize variables. The **`:=`** operator is not to be used as an assignment. That is, for a given variable, it can only be used once (at the time of its definition). Multiple assignment (see below) is permitted.

Control Structures

- **Assignment: `<var> = <expression>`**

  ```
  area = width * height
  ```

- **Multiple Assignment:**
  ```
  <var1>, <Var2>, …, <varN> = <expr1>, <expr2>, …, <exprN>
  <var1>, <Var2>, …, <varN> := <expr1>, <expr2>, …, <exprN>
  ```

  ```
  n, m := 10, 20
  ```

  Defines **n** and **m** to be **int** (by inference), initialized to 10 and 20, respectively.

  Multiple assignment can also be used to swap two variables:

  ```
  n, m = n, n
  ```

  The above swaps the contents of **n** and **m**. This is similar to the behavior in Python. Often, in Go, many functions return multiple values (we will see when we do functions) where multiple assignment can also be used.

- **Conditional Statements- If**
  ```
  if <condition> {                if <condition> {
       <statement(s)>                       <statement(s)>
  }                               } else {
                                       <statement(s)
                                  }


  max = b                         if a > b {
  if a > b {                           max = a
       max = a                    } else {
  }                                    max = b

                             }
  ```

  In Go if- statements, you do not have to surround the <condition> with parentheses (), but you can. However, braces **{}** are required for every block.

- **Conditional Statements- switch/case**

```
switch <expression> {
case <expression1> : <statement(s)>
case <expression2>: <statement(s)>
…
default: <statement(s)>
}

switch month {
case 1, 3, 5, 7, 8, 10, 12:
      daysInmonth = 12
case 4, 6, 9, 11:
      daysInMonth = 30
case 2:
      if (leapYear(year) {
            daysInMonth = 29
      } else {
            daysInMonth = 28
      }
default: fmt.Printf("Illegal value for month: %v\n", month)
os.Exit(1)
}
```

The **`<expression>`** in switch is optional. And, cases can refer to any variable in the current scope (see below). Case expressions can be any value, or an expression (see below). There is no break statement required (as in C, C++, Java, etc.). Only the case selected is executed.

```
r = rand.float64()   // generates a random float in [0.0..1.0)
switch {
case r < 0.5:
      outcome = HEADS
case r >= 0.5:
      outcome = TAILS
}
```

The switch/case statement also provides the use of a **`fallthrough`** statement in case there is a need to mimic the default fallthrough behavior of C/Java. Though I haven't yet found a good example of its use.

- **Loops- for**

  The only loop statement Go has is the for- statement. The general form of a for-statement is:

  ```
  for <init>; <condition>; <post> {
      <statements>
  }
  ```

  Unlike C, C++, Java, etc. there are no parentheses around the three control components of the loop. As is the case for if- statement, the braces **{}** are always required. Here are some examples:

  ```
  for i := 0; i < n; i++ {
      sum += i
  }
  ```

  Notice the use of short-hand declaration to define, initialize, and use the loop control variable, **i**. The for- statement can be used as a while- loop (i.e no **<init>** and no **<post>**):

  ```
  for a != b {
      if a > b {
          a = a – b
      } else {
          b – b – a
      }
  }
  ```

  You can omit all three loop control components:

  ```
  for {
      <statement(s)>
  }
  ```

  The above represents an infinite loop. **<statement(s)>** will be repeated forever. With arrays and slices (we'll discuss these later), you can write a *range loop* to iterate through all elements of an array, **A**:

  ```
  for i, x := range A {
      fmt.Prinf("index:%d, element: %v\n", i, x)
  }
  ```

  Above, **i** is set to the index, and **x** to the **A[i]** for all elements of **A**, starting with the first (at index, 0).

All loops can use the `continue` and `break` statements to skip to the next iteration of the loop, or to exit the loop, respectively.

Exercises

1. The Notorious FizzBuzz Problem: Write a program that prints numbers from 1 to 100. However, for numbers that are multiples of three it prints the word "Fizz" instead of the number, for multiples of five, it prints "Buzz". And, for numbers that are multiples of both three and five, it prints "FizzBuzz".

2. Find out how to generate random numbers in Go. Write two programs that simulate the following:

- rolls one six-sided die 100 million times and computes the empirical probability of getting a 6.
- rolls of two six-sided dice 100 million times and computes the empirical probability of getting a sum of 7.

What to Hand in:

Submit a PDF document in Dropbox containing the following:

1. A printout of the code and sample runs of the Go programs in Exercises 1 and 2.