> *Compilation date: 24.01.2022*

Introduction

`Go` is a modern-day language that has a refreshing approach to programming. Its simplicity and robustness make it one of today's most popular and loved languages. Moreover, due to its easy code maintenance and readability, almost every major organization is adopting it into its codebase.

1. Lexical Elements

i. Comments

Comments are helpful to document a code and explain its working. There are two ways to comment a code:

- 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 `*/`.

ii. Tokens

Tokens form the vocabulary of the Go language. There are four allowable classes: `identifiers`, `keywords`, `operators` and `punctuation`, and `literals`. White space, formed from spaces (U+0020), horizontal tabs (U+0009), carriage returns (U+000D), and newlines (U+000A), is ignored except as it separates tokens that would otherwise combine into a single token. Also, a newline or end of file may trigger the insertion of a semicolon. While breaking the input into tokens, the next token is the longest sequence of characters that form a valid token.

iii. Identifiers

```
identifier = ( letter | underscore ) { letter | unicode_digit |
underscore }*
```

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. For example,

```
a
x78pe
_Num
```

iv. Keywords

Keywords are predefined, reserved words used in programming that have special meanings to the compiler. These may not be used as identifiers.
List of all keywords in Go:

```
break       break
default     func
func        case
select      else
case        switch
struct      if
else        type
package     for
switch      return
```

v. Operators and punctuation

An operator is a symbol that tells the compiler to perform specific mathematical or logical functions. The operators follow the standard precedence rule. The following character sequences represent operators and punctuation:

```
+    &     +=    &=     &&    ==    !=    (    )
-    |     -=    |=     ||    <     <=    [    ]
*    ^     *=    ^=     <-    >     >=    {    }
/    <<    /=    <<=    ++    =     :=    ,    ;
%    >>    %=    >>=    --    !     ...   .    :
```

vi. White spaces

White space is the term used for the collective class of the the space character, the tab character and the newline character. It is ignored (outside of string and character constants) and is used to separate tokens. For example,

```
    x = y + z
```

2. Expressions

Expressions are combinations of atleast one operand with zero or more operators. Operands can be functions with a return value, constants or variables. For example,

```
1234
A + B -11
AREA(2, 3) + 10
```

3. Constants

Constants in `Go` are the fixed values that are used in a program, and its value remains the same during the entire execution of the program.

Syntax: Constants are declared using `const` keyword. For example,

```
const untypedInteger                    = 123
const untypedFloating                   = 123.12
const typedInteger  int                 = 123
const typedFloatingPoint  float64       = 123.12
```

Constants can be declared with or without a type in Go. A constant may be given a type explicitly by a constant declaration or conversion, or implicitly when used in a variable declaration or an assignment or as an operand in an expression. An untyped constant has a default type which is the type to which the constant is implicitly converted in contexts where a typed value is required, for example, `i := 0`, where there is no explicit type.

There are three types of constants in `Go`:

- **Numeric constants** : These include integer and floating-point. Integer constants may be represented in octal (with prefix 0), hexadecimal (with prefix 0x or 0X) and decimal. They can be followed by the suffix U and L to denote unsigned and long respectively. E.g. 12, 0x4b , 0213 , 30UL, 3.124.
- **String constant**: These are enclosed with double quotes. E.g. "hello", "i am learning cs335a"
- **Boolean constants**: `true` , `false`

4. Input / Output

`Printf()` will be used for standard output, whereas `Scanf()` will be used for standard input.
Formatting verbs are:

```
%d for int
%f for float
%s for string
```

Example:

```
    var x int
    var str string = “World\n”
    Scanf(“%d”, &x)
    Printf(“x = %d”, x)
    Printf(“Hello %s”, str)
```

5. Functions

```
func function_name(Parameter-list)(Return_type){
    // function body.....
}
```

Functions are the block of codes or statements in a program which gives the user the ability to reuse the same code. This saves the excessive use of memory, saves time and provides better readability of the code. For example,

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

i. Single value return

The returned values are passed back using the return keyword. In our implementation of `Go`, we are only focussing on `single-value return`, like `int`, `float`, `boolean`, `string`, `array` or `struct`. (Note that Go does not have function pointers, and we would maintain it as such.)

ii. Recursion

With the given grammar for language, a function can call itself, i.e., we support `recursion`.
Here is an example,

```
func factorial ( a int ) int {
    if ( a < 1 ){
        return 1
    }
    return factorial( a-1 )
}
```

6. Main function

Any given Golang program requires a `main` function.It does not take any argument nor return anything. It is the entry point of the executable programs. Golang will automatically call the `main()` function. Hence it needs not to be called explicitly.

```
func main() {
    // code block
}
```

7. Data Types

i. Primitive Data Types

The following are the primitive data types in `Go`

```
    uint8       the set of all unsigned  8-bit integers
    uint16      the set of all unsigned 16-bit integers
    uint32      the set of all unsigned 32-bit integers
    uint64      the set of all unsigned 64-bit integers (0 t

    int8        the set of all signed  8-bit integers
    int16       the set of all signed 16-bit integers
    int32       the set of all signed 32-bit integers
    int64       the set of all signed 64-bit integers

    float32     the set of all IEEE-754 32-bit floating-point numbers
    float64     the set of all IEEE-754 64-bit floating-point numbers

    string     A string value is a (possibly empty) sequence of bytes.
    boolean  Boolean truth values denoted by the predeclared constants
true and  false
```

a. Initialization and declaration of variables:

1. Only Declaration : We have to specify the type of the variable that is being declared . Example of declaring an integer variable:

```
var  my_variable int
```

2. Declaration and Initialization : We need not specify the variable type to be specified if we are initializing the variable at the time of declaration Examples of declaration and initialization:

```
var  my_string = "abcdef"
my_string2 :=  "abcdef"
```

ii. Structure

A structure is a user defined data type consisting of variables of other data types, possibly other user defined data types.

a. Defining Structure

You define a structure using `type` and `struct` keywords followed by the struct's members enclosed in braces. Each member of the struct is defined using an identifier followed by data type. Different members are separated using a newline.

```
type student struct {
    Name string
    Roll_no int
}
```

b. Declaring Structure

A `struct` is declared just like any other data type

```
var Student1 student
```

c. Accessing Structure Members

After declaration of a `struct` variable, the member of the struct can be accessed using the dot (`.`) operator.

```
Student1.Name = "ABC"
```

iii. Arrays

An array is a data structure that lets you store one or more elements consecutively in memory. In `Go`, array elements are indexed beginning at position zero, not one.

a. Declaring Arrays

You declare an array by specifying the data type for its elements, its name, and the number of elements it can store. Here is an example that declares an array named `arr` that can store `5` integers:

```
var arr[5] int
```

The function `len(arr)` returns the length of the array named `arr`.

b. Initializing Arrays

You can initialize the elements in an array when you declare it by listing the initializing values, separated by commas, in a set of braces. Here is an example:

```
my_array := [5] int {1,2,3,4,5}
```

You don't have to explicitly initialize all of the array elements. For example, this code initializes the first three elements as specified, and then initializes the last two elements to a default value of zero:

```
my_array := [5] int {1,2,3}
```

c. Accessing Array Elements

You can access the elements of an array by specifying the array name, followed by the element index, enclosed in brackets. Remember that the array elements are numbered starting with zero.

```
my_array[0] = 5
```

That assigns the value 5 to the first element in the array, at position zero. You can treat individual array elements like variables of whatever data type the array is made up of.

d. Multidimensional Arrays

You can make multidimensional arrays, or "arrays of arrays". You do this by adding an extra set of brackets and array lengths for every additional dimension you want your array to have. For example, here is a declaration for a two-dimensional array that holds five elements in each dimension (a two-element array consisting of five-element arrays):

```
var arr[2][5] int
my_array := [2][5] int {{1,2,3,4,5},{-1,-2,-3,-4,-5}}
```

Multidimensional array elements are accessed by specifying the desired index of both dimensions: `arr [1][3] = 12;`

iv. Slices :

Slices are a key data type in `Go`, giving a more powerful interface to sequences than arrays.

a. Declaring Slices

We can use the built-in function to declare slices in `Go`. Here we make a slice of strings of length 3.

```
s : = make([ ] string ,3)
```

b. Initializing Slices

You can initialize the elements in a slice when you declare it by listing the initializing values, separated by commas, in a set of braces. Here is an example:

```
s  := [ ] string {"we","are","building","go","compiler"}
```

The function `len(s)` returns the length of Slice `s`.
We can set and get just like with arrays

```
s[0] = "custom"
s[1] = "set"
s[2] = "get"
```

c. Dynamic Memory (*):

The `append` function of the slice data structure is similar to the dynamic memory in other languages.

Dynamic Nature of Slices (the `append` function)

In addition to these basic operations, slices support several more that make them richer than arrays. One is the builtin append, which returns a slice containing one or more new values. Note that we need to accept a return value from append as we may get a new slice value. Here are a few examples:

```
s = append(s, "d")
s = append(s, "e", "f")
```

These operations append the strings "d", "e", "f" at the end of the original slice `s`.

The Slice functionality (`:` operator)

Slices support a "slice" operator with the syntax `slice[low:high]`.
This gets a slice of the elements from `[low , high)`
For Example :

```
s  := [ ] string {"this","is","course","CS335","A"}

t  := s[2:4]    //  t = ["course", "CS335"]
t   = s[2:]     // t = ["course", "CS335", "A"]
t   = s[:4]     // t = ["this", "is", "course", "CS335"]
```

Multidimensional Slices

Just like arrays slices can also be multi-dimensional and the size of each dimension can be increased using the append operator as was the case with single dimension.

```
twoD := make([][] int, 3)
```

The above code declares a slice of slices of initial size 3 .

```
twoD[2] = make([]int,4)
```

The above code initializes the third slice to be a slice of size 4. We can access the elements of the 2-D slices similar to the 2-D arrays .

```
twoD[2][3] = 7
```

We can even change the sizes of the slices in all dimensions using the append operator.

```
twoD[2] = append (twoD[2],3,4,5)
twoD = append(twoD,make([]int,6))
```

v. Pointers

Pointers are used to store the direct memory address of another variable.

a. Defining Pointers

Pointers are defined using asterisk (*) operator followed by the data type of the variable pointed by this pointer.

```
var point1 *int32
var point2 *string
```

b. Initialising Pointers

& operator can be used to get the address of a defined variable. Then the assignment operator = can be used to initialise the pointer with that address.
Eg:

```
var x int = 5
var int_ptr *int = &x
```

c. Dereferencing Pointers

Asterisk (`*`) can be used to access the value of a variable using pointers.

```
var x int = 5
var int_ptr *int = &x
*int_ptr = 10
```

8. If-Else

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

The conditional execution of two branches according to the value of a boolean expression is specified by `if` statements.
The `if` branch is run if the expression evaluates to true; otherwise, the `else` branch is executed if it is present. The `else if` statement to specify a new condition if the first condition is false. We can also have if statements inside if statements, also called a nested `if`.

```
if condition {
    // execute this block of code
}
```

A simple statement that executes before the expression is evaluated can be used to precede the expression. For example,

```
if x := f(); x < y {
    return x
} else if z < x {
    return y
}
```

9. Switch

```
ExprSwitchStmt = "switch" [ SimpleStmt ";" ] [ Expression ] "{" {
ExprCaseClause } "}" .
ExprCaseClause = ExprSwitchCase ":" StatementList .
ExprSwitchCase = "case" ExpressionList | "default" .
```

`switch` statement acts as an alternative of `if-else` and allows to select one of many code blocks to be executed. `switch` statements in Go resemble those in C, C++, etc. But a key difference lies in that it only runs the matched case, and so there is no need for a break statement. An expression is evaluated and its value is compared with each case, and block of code associated with matched case gets executed. In case of no match, code in `default` case if executed, if it exists.

```
switch expression {
    case x:
    // code block
    case y:
    // code block
    default:
    // code block
}
```

It is possible to associate the same code block with multiple values, and is known as multi-case `switch`.

```
switch expression {
    case x,y:
    // code block if expression is evaluated to x or y
    case v,w:
    // code block if expression is evaluated to v or w
    default:
    // code block if expression is not found in any cases
}
```

10. For loop

```
ForStmt = "for" [ Condition | ForClause | RangeClause ] Block .
Condition = Expression .
```

As common to most of the other languages, `for` statement commands repeated execution of a block of code. In go, we can specify iteration using either a single condition, a `for` clause, or a `range` clause. It is possible to place a loop inside another loop, also called "nested loops".

i. Single condition

Similar to `while` loop is `C/C++`, it executes code block as long as the single condition evaluates to `true`.

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

ii. For clause

In this, `statement1` initialise loop counter value, `statement2` is the check condition, and `statement3` modifies the loop counter value.

```
for statement1; statement2; statement3 {
   // code to be executed for each iteration
}
```

Two common statements used in a `for` loop are

- `continue`: skip that iteration in the loop
- `break`: breaks out of the loop execution

iii. Introducing multi-level breaks (*)

We have extended the break functionality . Now the break keyword can be followed by an integer argument. The integer argument depicts the number of nested loops that we break out of.

```
for (int i = 1 : 10) {
    for (int j = 1 : 10) {
        for (int k = 1 : 10) {
            if (k > 3) {
                break 2;
            }
        }
    }
}
```

In the example shown , the control breaks out of the two inner nested loops and continues execution in the outermost loop Note that if the integer argument is greater than the number of nested loops , the compiler gives an error.

11. Math library (*)

The following basic math functions will be supported by our compiler (names are self-explanatory).

```
MIN(x, y), MAX(x, y), POW(x, y), ABS(x) - here x and y are integer data
types.
```

12. String library (*)

The following basic string functions will be supported by our compiler (names are self-explanatory).

```
UPPER(x), LOWER(x), REVERSE(x), LEN(x) - here x is string.
```

13. Code Optimizations (*)

- **Dead Code Elimination** : This involves removing parts of code from the executable (essentially ignoring the code while compiling) that never gets executed .

For example, consider the following :

```
x:= 3

if x == 2  {
    // this code block never gets executed
}

// execution starts here
```

The code inside the if condition never gets executed, so we can remove it.

15. Glossary

* : Advanced and cool features, that would be implemented if time and feasibility permits.

16. Changes

As we have started building our compiler, we needed to make some minor changes in our specs along the way. Following points contain some of the limitations our compiler has, as well as, some of the features which aren't incorporated yet but will be in future iterations :-

- We have not allowed for struct element access using `->`.
- Standard way of struct initialization not supported (plans to include some other syntax support for that)
- Array bound checks not yet included (will do it during run time)
- Semantic checks for constant variable types not yet done
- Only single statement allowed inside switch cases
- No handling for slices done as of yet, will be done in future
- We enforce strict type checking for expressions and arithmetic operators . (3, 3.0 treated differently) i.e. 3 is of type `int64` , 3.0 is of type `float64` , 3 won't be treated as a float in operations. But 3 can be used in arithmetic operations with any other type of int
- We don't support overloading of functions or defining functions with different return types
- We can't have a variable as the same name as function
- Functions have to be ordered i.e. we cannot call a function `foo()` earlier and then declare it afterwards.
- Have to store size in symbol table, will be done in future
- We have plans to include `package` and `import` keyword in future
- Typecasting is not handled as of now.