

Milestone 1

Group 15

1 Native Data types

1.1 Integer types

An integer type represents sets of integer values. The predeclared architecture-independent numeric types are:

uint8	the set of all unsigned 8-bit integers (0 to 255)
uint16	the set of all unsigned 16-bit integers (0 to 65535)
uint32	the set of all unsigned 32-bit integers (0 to 4294967295)
uint64	the set of all unsigned 64-bit integers (0 to 18446744073709551615)
int8	the set of all signed 8-bit integers (-128 to 127)
int16	the set of all signed 16-bit integers (-32768 to 32767)
int32	the set of all signed 32-bit integers (-2147483648 to 2147483647)
int64	the set of all signed 64-bit integers (-9223372036854775808 to 9223372036854775807)
float32	the set of all IEEE-754 32-bit floating-point numbers
float64	the set of all IEEE-754 64-bit floating-point numbers
byte	alias for uint8
rune	alias for int32

1.2 Boolean types

A boolean type represents the set of Boolean truth values denoted by the predeclared constants `true` and `false`. The predeclared boolean type is `bool`.

1.3 Character types

Go doesn't have a `char` data type. It uses `byte` and `rune` to represent character values. The `byte` data type represents ASCII characters and the `rune` data type represents a more broader set of Unicode characters that are encoded in UTF-8 format.

2 Variables

A variable is a storage location for holding a value. The set of permissible values is determined by the variable's type.

2.1 Declaration

Static type declaration is declared using the `var` keyword and the type of the variable is the type given in its declaration

Syntax: `var variable_name type = expression`

Variables of interface type also have a distinct **dynamic type**, which is the concrete type of the value assigned to the variable at run time.

3 User defined types

3.1 structs

Since there is no class type in Go, `struct` will be our main user defined data type. A struct is a sequence of named elements, called fields, each of which has a name and a type. The declaration starts with the keyword `type`, then a name for the new struct, and finally the keyword `struct`. Within the curly brackets, a series of data fields are specified with a name and a type. For eg:

```
type identifier struct{
    field1 data_type
    field2 data_type
    field3 data_type
}
```

Using var keyword and the identifier used above we can initialise a variable of the define type.Using dot notation we can access values of the struct.

```
var variable_name identifier
variable_name.field1 = "Hello"
```

We will support nested structures as well,and an example of nested structure type will be:

```
type geometry struct {
    area int
    perimeter int
}
type rectangle struct {
    length int
    breadth int
    metrics geometry
}
```

We will also support copying Struct Type Using Value and Pointer Reference

```
type rectangle struct {
    length int
    breadth int
}
var r2,r1 rectangle;
r2=r1
r3=&r1
```

r2 will be the same as r1, it is a copy of r1 rather than a reference to it. Any changes made to r2 will not be applied to r1 and vice versa. When r3 is updated, the underlying memory that is assigned to r1 is updated.

4 Expressions

An expression specifies the computation of a value by applying operators and functions to operands.

4.1 Operators

Operators combine operands into expressions.

4.2 Unary Operators

Unary operators act on a single operand to produce a value.

&

*

+

-

!

4.2.1 Arithmetic operators

+	sum	integers, floats, complex values, strings
-	difference	integers, floats, complex values
*	product	integers, floats, complex values
/	quotient	integers, floats, complex values
%	remainder	integers
&	bitwise AND	integers
	bitwise OR	integers
^	bitwise XOR	integers
&^	bit clear (AND NOT)	integers
<<	left shift	integer << integer >= 0
>>	right shift	integer >> integer >= 0

4.2.2 Assignment Operators

=	ASSIGN
*=	MUL ASSIGN
/=	DIV ASSIGN
%=	MOD ASSIGN
+=	ADD ASSIGN
-=	SUB ASSIGN
&=	AND ASSIGN
=	XOR ASSIGN
=	OR ASSIGN

4.2.3 Comparison Operators

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

4.2.4 Logical operators

&&	conditional AND	p && q is “if p then q else false”
	conditional OR	p q is “if p then true else q”
!	NOT	!p is “not p”

4.3 Operator precedence

Operators with higher precedence appear higher here.

Precedence	Operator
Postfix	(), [], - >, ., ++, --
Unary	+, -, !,
Multiplicative	*, /
Additive	+, -
Shift	<<, >>
Relational	<, <=, >, >=
Equality	==, !=
Bitwise AND	&
Bitwise XOR	^
Bitwise OR	
Logical AND	&&
logical OR	
Assignment	=, + =, - =, * =, / =, % =, >> =, << =, & =, ^ =, =,
Comma	,

Binary operators of the same precedence associate from left to right.

5 If-Else statements

If-else statements are used to conditionally execute code based on the value of a boolean expression. If the expression is true then the code in the if block is executed otherwise the code in the else block is executed. An if statement need not be followed by an else statement.

IfStatement = "if" [SimpleStatement ";"] Expression Block ["else" (IfStatement | Block)]

```
if x>y{
    state = 0
}
```

```
if x>y{
    return x
} else if x>z{
    return z
} else {
    return y
}
```

6 For statement

For statements are used to repeatedly execute a block. There are three ways the iteration is determined, a single condition, a 'for' clause, or a 'range' clause.

6.1 For statement with single condition

The block executes as long as a boolean condition evaluates to true. If no condition is true, it is considered to be true.

```
for x<50{
    x = x+10
}
```

6.2 For statement with 'for' clause

This type of for statement has a boolean condition, but an 'init' and 'post' statement may also be present. The init statement is executed once before the first iteration, before the condition is evaluated, and the post statement is executed after each iteration. The init statements and post statements can't be variable declaration, that is var x int=0, can't be used but we can use x:=0 or any expression such as x=0.

```
var y[n] int
for x := 0; x<n; x++){
    y[x] = x+1
}
```

7 Break statement

This statement breaks the execution of the innermost 'for' . The 'for' statement should be within the same function.

"break" [label]

Example:

```
for a<2 {
    x++
    if x>50 {
```

```
        break
    }
}
```

8 Continue statement

This statement begins the next iteration of the enclosing innermost 'for' loop. The for loop should be within the same function.

```
for x: = 0; x<n; x++){
    if a[x] != 45{
        continue
    }
    f()
}
```

9 Functions

Function body in go has following structure.

9.1 Regular Functions

```
func function_name(parameter-list) return-type{
    function-body
}
```

Following is simple function

```
func add(x int, y int){
    return x+y
}
```

This is called as add(3, 4)

10 Pointers

Pointers are special variables which store address of variable and can be dereferenced to access value stored at that location. Pointer to variable of type T will be stored in type *T

```
var x int = 5
var y *int = &x    // & operator is used to get address of the variable
*y += 1            // * operator is used to get value stored at address pointed by pointer
```

11 Arrays

11.1 Arrays

Arrays in go have compile time constant size and are stored in data type [n]T where n is the size and T is the type. Multidimensional arrays are also supported

```
var x [3]int
var y [2]int = {0, 1}    // Arrays can be initialized with list of literals in brackets
// [3]int and [2]int are completely different types
```

12 Restriction

All statements should be ended with semicolon,including if else,function definition,for loop ..

13 Reference

- 1.<https://www.golangprograms.com/go-language/struct.html>
- 2.<https://www.callicoder.com/golang-basic-types-operators-type-conversion/>
- 3.<https://www.geeksforgeeks.org/how-to-add-a-method-to-struct-type-in-golang/>
- 4.<https://go.dev/ref/spec>