Operators

This lesson discusses operators supported by Go.

WE'LL COVER THE FOLLOWING

Introduction

A symbol that is used to perform *logical* or *mathematical* tasks is called an **operator**. Go provides the following built-in operators:

- Arithmetic operators
- Logical operators
- Bitwise operators

Arithmetic operators

The common binary operators +, -, * and / that exist for both integers and floats in Golang are:

- Addition operator `+`
- Division operator `/`
- Modulus operator `%`
- Multiplication operator `*`

The `+` operator also exists for strings. The `/` operator for integers is (floored) integeral division. For example, $9/4$ will give you 2, and $9/10$ will give you 0.

The modulus operator `%` is *only* defined for integers. For example, 9%4 gives 1 .

Integer division by 0 causes the program to crash, and a run-time panic occurs (in many cases the compiler can detect this condition). Division by 0.0 with floating-point numbers gives an infinite result: +**Inf**.

There are shortcuts for some operations. For example, the statement:

```
b = b + a
```

can be shortened as:

```
b += a
```

The same goes for `-=`, `*=`, `/=` and `%=` .

As unary operators for integers and floats we have:

- Increment operator `++`
- Decrement operator `--`

However, these operators can only be used after the number, which means: **i**++ is short for **i**+=**1** which is, in turn, short for **i**=**i**+**1**. Similarly, **i**– is short for **i**-=**1** which is short for **i**=**i**-**1**.

Moreover, ++ and -- may only be used as statements, not expressions; so **n** = **i**++ is invalid, and subtler expressions like **f(i**++**)** or **a[i]**=**b[i**++**]**, which are accepted in C, C++ and Java, *cannot* be used in Go.

No error is generated when an *overflow* occurs during an operation because high bits are simply *discarded*. Constants can be of help here. If we need integers or rational numbers of unbounded size (only limited by the available memory), we can use the **math/big** package from the standard library, which provides the types **big.Int** and **big.Rat**.

Logical operators

Following are logical operators present in Go:

- Equality operator `==`
- Not-Equal operator `!=`

- Less-than operator `<`

- Greater-than operator `>`
- Less-than equal-to operator `<=`
- Greater-than equal-to operator `>=`

The following illustration describes how they work.

A	B	A==B
2	5	False
10	10	True
7	3	False

Returns true if both sides of operator have same values

Equality Operator

1 of 6

A	B	A!=B
2	5	True
10	10	False
7	3	True

Returns true if both sides of operator have different values

Not Equal Operator

2 of 6

A	B	A < B
2	5	True
10	10	False
7	3	False

Returns true if left side of operator is less than right side

Less Than Operator

A	B	A > B
2	5	False
10	10	False
7	3	True

Returns true if left side of operator is greater than right side

Greater Than Operator

A	B	A <= B
2	5	True
10	10	True
7	3	False

Returns true if left side of operator is less than or equal to right side

Less Than or Equal To Operator

5 of 6

A	B	A >= B
2	5	False
10	10	True
7	3	True

Returns true if left side of operator is greater than or equal to right side

Greater Than or Equal To Operator

6 of 6

Go is very strict about the values that can be compared. It demands that values have to be of the same type. If one of them is a constant, it must be of a type compatible with the other. If these conditions are not satisfied, one of the values has first to be converted to the other's type.

<, <=, >, >=, == and != not only work on number types but also on strings. They

are logical because the result value of these operators is of type *bool*. Run the following program to see how these operators work:

```
package main
import "fmt"

func main(){
    b3 := 10 > 5          // greater than operator
    fmt.Println(b3)
    b3 = 10 < 5           // less than operator
    fmt.Println(b3)
    b3 = 5 <= 10          // less than equal to
    fmt.Println(b3)
    b3 = 10 != 10         // not equal to
    fmt.Println(b3)
}
```

Logical Operators

As you can see, we use >, <, <= and != operators and change the value of `b3` one by one at **line 5**, **line 7**, **line 9** and **line 11** to evaluate the results. The results are either *true* or *false* based on the comparison made.

Boolean constants and variables can also be combined with logical operators to produce a boolean value. Such a logical statement is not a complete Go-statement on itself. Go has three boolean logical operators:

- **AND** operator (`&&`)
- **OR** operator (`||`)
- **NOT** operator (`!`)

The following illustration describes how they work.

A	B	A&&B
True	True	True
True	False	False
False	True	False
False	False	False

Returns true if both operands are true

AND Operator

1 of 3

A	B	A||B
True	True	True
True	False	True
False	True	True
False	False	False

Returns true if one of the operands or both operands are true

OR Operator

2 of 3

A	!A
True	False
False	True

Returns the opposite of the value

NOT Operator

3 of 3

`AND` and `OR` are binary operators where `NOT` is a unary operator. The && and || operators behave in a shortcut way that when the value of the left side is known, and it is sufficient to deduce the value of the whole expression (false with && and true with ||), then the right side is not computed anymore. For

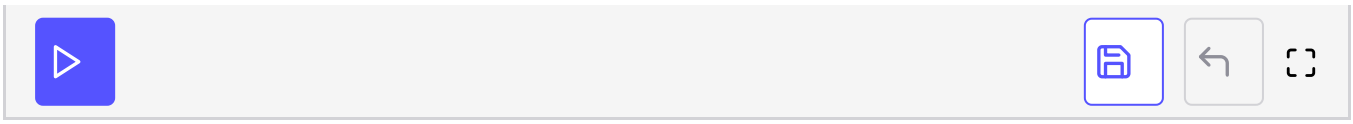
with && and true with ||), then the right side is not computed anymore. For

that reason, if one of the expressions involves a long-lasting calculation, put this expression on the right side.

Run the following program to see how these three operators work:

```
package main
import "fmt"

func main(){
    b3 := 10 > 5 && 7 < 15     // AND operator
    fmt.Println(b3)
    b3 = 10 < 5 || 2 > 7       // OR operator
    fmt.Println(b3)
    b3 = !b3                   // NOT operator
    fmt.Println(b3)
}
```



Boolean Logical Operators

As you can see, we use AND, OR and NOT operators and change the value of `b3` one by one at **line 5**, **line 7**, **line 9** to evaluate the results. The results were either *true* or *false* based on the comparison made.

Bitwise operators

They work only on integer variables having bit-patterns of equal length. **%b** is the format-string for bit-representations. Following are some bitwise operators:

- Bitwise AND operator `&`
- Bitwise OR operator `|`
- Bitwise XOR operator `^`
- Bit CLEAR operator `&^`
- Bitwise COMPLEMENT operator `^`

These are explained in the illustration below:

A	B	A&B
0	0	0
0	1	0
1	0	0
1	1	1

Bits in the same position are and-ed together

Bitwise AND Operator

A	B	A|B
0	0	0
0	1	1
1	0	1
1	1	1

Bits in the same position are or-ed together

Bitwise OR Operator

A	B	A^B
0	0	0
0	1	1
1	0	1
1	1	0

Bits in the same position are xor-ed together

Bitwise XOR Operator

A	B	^B	A&^B
0	0	1	0
0	1	0	0
1	0	1	1
1	1	0	0

Use to clear a bit to 0, by definition: **A&^B = A &(NOT B)**

Bit Clear Operator

A	^A
0	1
1	0

When used as a unary operator, it applies complement on all bits

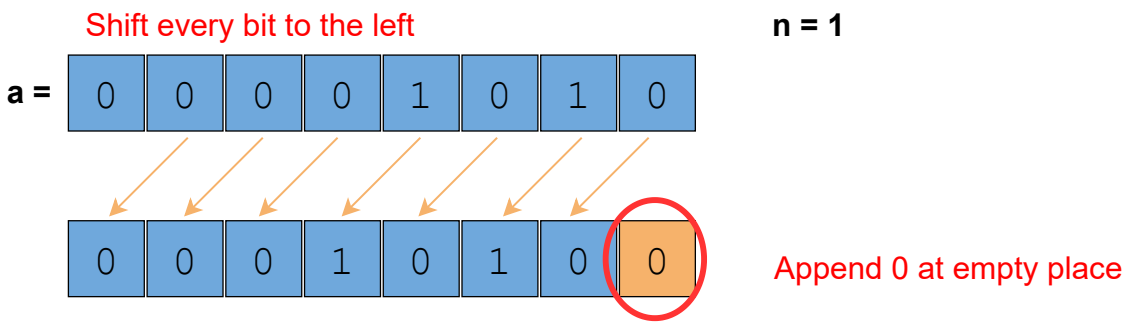
Bitwise Complement Operator

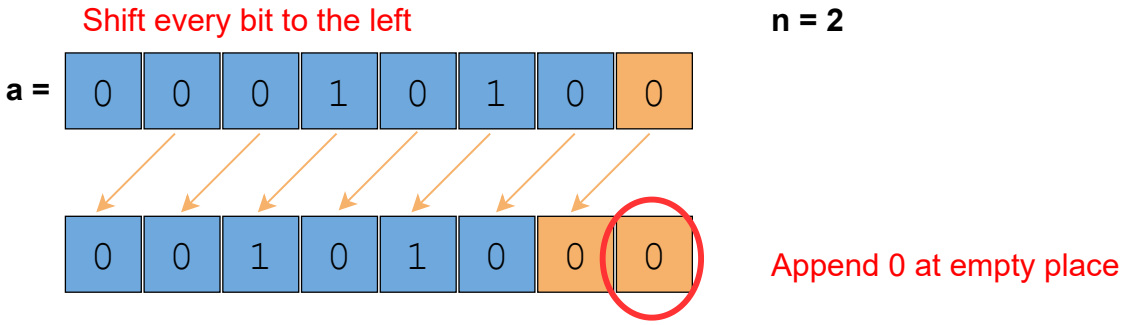
Bitwise AND, OR, XOR, and CLEAR are *binary* operators which means they require *two* operands to work on. However, the complement operator is a unary operator.

There are other two major bitwise operators used for *shifting*:

- Left shift operator `<<`
- Right shift operator `>>`

Assume `a` holds **10**. Let's see how left shifting by **2** works:

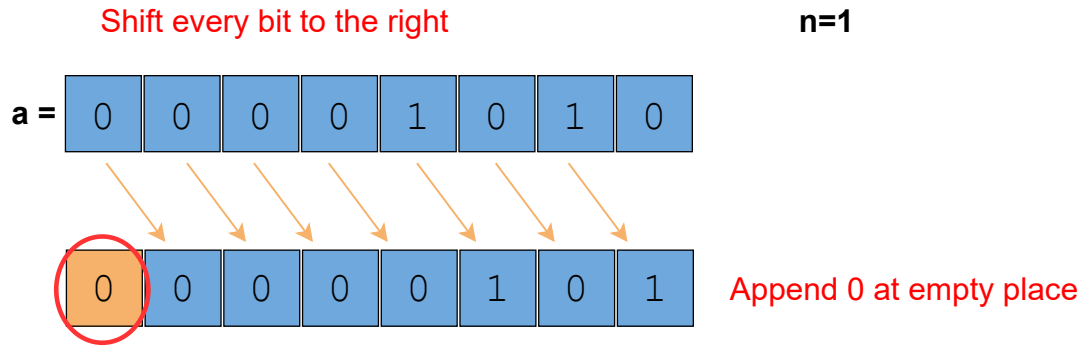




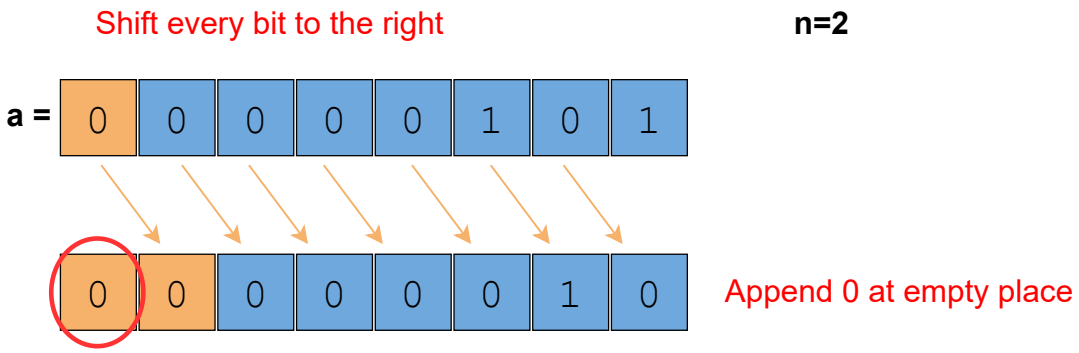
Left Shift Operator

Left Shift Operator

Assume `a` holds **10**. Let's see how right shifting by **2** works:



Right Shift Operator



Right Shift Operator

2 of 2

Left shift by `n` causes multiplication by 2^n, and right shift by `n` causes division by 2^n. For example, if `n` is **2**, the number is *multiplied* by **4** for left shifting and *divided* by **4** for right shifting.

Operators and precedence

Some operators have higher priorities (precedence) than others. Binary operators of the same precedence associate from left to right. The following table lists all the operators and their precedence (much shorter and clearer than in C or Java), *top to bottom* is *highest to lowest*:

Precedence	Operator(s)
7	`^` `!`
6	`*` `/` `%` `<<` `>>` `&` `&^`
5	`+` `-` `|` `^`
4	`==` `!=` `<` `<=` `>=` `>`
3	`<-`

2	`&&`
1	`||`

Using ( ) is of course allowed to clarify expressions, to indicate priority in operations as expressions contained in ( ) are always evaluated first.

That’s it about operators. There is a challenge in the next lesson for you to solve.