Expressions and Statements in Go.

Session 04

Golang course by Exadel

13 Oct 2022

Sergio Kovtunenko
Lead backend developer, Exadel

Agenda

- Go Expressions overview
- Expressions: **Composite** literals
- Expressions: **Index** expressions
- Expressions: **Operators**
- Go **Statements** overview
- Statements: **assignments**
- Statements: `if`, `for`, `switch`, `break`, `continue`, `goto`
- Next time...

Introduction

Introduction

▶️ In my mind: **language is a tool to express (and share) ideas**.

▶️ Language related to a philosophy and design.

▶️ Some statements will not be covered in the current session, such as:

- **send/receive** to/from the channels statements
- **select** statement
- **type switch**
- **go** statement
- **return** statement
- **defer** statement

Expressions in Go language

Go Expressions overview

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

▶️ **Operands** denote the *elementary values in an expression*.

▶️ **Operators** combine operands into *expressions*.

✓ Operator precedence
- **Unary operators** have the highest precedence
- **Binary operators** of the same precedence associate from left to right:

Precedence	Operator
5	* / % << >> & &^
4	+ - | ^
3	== != < <= > >=
2	&&
1	||

▶️ **DANGER:** *Operator precedence* in Go is not the same as in C language!

Comparison operators

- ✓ Comparison operators compare two operands and yield an untyped boolean value.

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

- ✓ The **equality operators** `==` and `!=` apply to operands that are comparable.
- ✓ The **ordering operators** `<`, `<=`, `>`, and `>=` apply to operands that are ordered.
- ✓ Slice, map, and function values are **not comparable**.
  - But a slice, map, or function and other values may be compared to the predeclared identifier `nil`.

Logical operators

- ✓ Logical operators apply to *boolean* values and yield a result of the same type as the operands.
- ✓ The right operand is evaluated conditionally.

```
&&    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"
```

Address operator

- ✓ For an operand `x` of type `T`, the address operation `&x` generates a pointer of type `*T` to `x`.
- ✓ For an operand `x` of pointer type `*T`, the pointer indirection `*x` denotes the variable of type `T` pointed to by `x`.
  - `nil` pointer dereference will cause a run-time panic.
- ✓ Example:

```
&x
&a[f(2)]
&Point{2, 3}
*p
*pf(x)

var x *int = nil
*x   // causes a run-time panic
&*x  // causes a run-time panic
```

Index Expressions

- ✓ A primary expression of the form **a[x]** denotes the element (indexed by x) of the:
  - array
  - pointer to array
  - slice
  - string
  - map.
- ✓ The value x is called the index or map key, respectively.
- ✓ If a is not a map:
  - the index x must be of *integer type* or *an untyped constant*.
  - it must be in range 0 <= x < len(a).
    - Otherwise, runtime panic will occur.
  - a *constant index* must be non-negative and *representable by a value of type* ***int***.
- ✓ For a of string type:
  - a[x] is the non-constant byte value at index x and the type of a[x] is byte
  - a[x] may not be assigned to
- ✓ For a of map type M:
  - if the **map** contains an entry with key x, a[x] is the **map** element with key x and the type of a[x] is the element type of M.
  - if the **map** is **nil** or does not contain such an entry, a[x] is the zero value for the element type of M.

Composite Literals (1/2)

- ✓ Composite literals construct values for: structs, arrays, slices and maps and create a new value each time they are evaluated.
  - Each element may *optionally* be preceded by a corresponding key.
    - For map literals, all elements must have a key.
- ✓ Examples of valid **struct** literals:

```
// For given types declaration:
type Point3D struct { x, y, z float64 }
type Line struct { p, q Point3D }
// we can create values like:
origin := Point3D{}                            // zero value for Point3D
line := Line{origin, Point3D{y: -4, z: 12.3}}  // zero value for line.q.x
```

Composite Literals (2/2)

✓ Examples of valid **array**, **slice**, and **map** literals:

```
// list of prime numbers
primes := []int{2, 3, 5, 7, 9, 2147483647}

// vowels[ch] is true if ch is a vowel
vowels := [128]bool{'a': true, 'e': true, 'i': true, 'o': true, 'u': true, 'y': true}

// the array [10]float32{-1, 0, 0, 0, -0.1, -0.1, 0, 0, 0, -1}
filter := [10]float32{-1, 4: -0.1, -0.1, 9: -1}

// frequencies in Hz for equal-tempered scale (A4 = 440Hz)
noteFrequency := map[string]float32{
    "C0": 16.35, "D0": 18.35, "E0": 20.60, "F0": 21.83,
    "G0": 24.50, "A0": 27.50, "B0": 30.87,
}
```

Statements in Go language

Statements overview

▶️ Statements control program flow execution.

▶️ *Statements* are **not** *Expressions*.

▶️ Interesting facts about statement in Golang:

- **mandatory** braces

- **no parentheses** for conditionals

- **implicit break** in switches

- **no** semicolons

- multiple assignments

Assignments (1/2)

✓ Examples of assignments:

```
a, b = b, a  // exchange a and b

x := []int{1, 2, 3}
i := 0
i, x[i] = 1, 2  // set i = 1, x[0] = 2

i = 0
x[i], i = 2, 1  // set x[0] = 2, i = 1

x[0], x[0] = 1, 2  // set x[0] = 1, then x[0] = 2 (so x[0] == 2 at end)

x[1], x[3] = 4, 5  // set x[1] = 4, then panic setting x[3] = 5.

type Point struct { x, y int }
var p *Point
x[2], p.x = 6, 7  // set x[2] = 6, then panic setting p.x = 7

i = 2
x = []int{3, 5, 7}
for i, x[i] = range x {  // set i, x[2] = 0, x[0]
    break
}
// after this loop, i == 0 and x == []int{3, 5, 3}
```

Assignments (2/2)

- ✓ A tuple assignment assigns the *individual elements* of a multi-valued operation to a list of variables.
  - The number of operands on the left hand side must match the number of values.
  - In the first form:

```
func f() (int, int) {
    return 1, 2
}
var x, y = f() // function f() returns 2 results!
```

  - In the second form:

```
var one, two, three = '一', '二', '三'
```

- ✓ The *blank identifier* _ provides a way to ignore right-hand side values in an assignment:

```
_ = x       // evaluate x but ignore it
x, _ = f()  // evaluate f() but ignore second result value
```

Increment / Decrement statements

- ✓ The "++" and "--" statements **increment** or **decrement** their operands by the untyped constant 1.
- ✓ The following assignment statements are semantically equivalent:

```
IncDec statement    Assignment
x++                 x += 1
x--                 x -= 1
```

`if` statements (1/2)

- ✓ "**if**" statements specify the conditional execution.
  - If the expression *evaluates to true*, the "**if**" branch is executed
  - otherwise (if present) the "**else**" branch is executed.
  - Example:

```
if x > max {
    x = max
} else {
    max = x
}
```

- ✓ Like **for** and **switch**, the **if** statement can start with a short statement *to execute before* the condition.

```
if v := math.Pow(x, n); v < lim {
    a = v
}
```

  - Variables declared by the **if** statement are only in scope until the end of the **if**.
  - Variables declared inside an **if** statement are also available inside any of the **else** blocks.

`if` statements (2/2)

- ✓ Keep the normal code path at a minimal indentation:
  - While it's not considered bad practice to use **else**, it's actually fairly uncommon to see an **else** or **else if**.

```go
func badOriginalCode() {
    // ...
    if _, ok := f.dirs[dir]; !ok {
        f.dirs[dir] = new(feedDir)
    } else {
        f.addErr(fmt.Errorf("..."))
        return
    }
    // some code
}
func revisedCode() {
    // ...
    if _, found := f.dirs[dir]; found {
        f.addErr(fmt.Errorf("..."))
        return
    }
    f.dirs[dir] = new(feedDir)
    // some code
}
```

Source: "When in Go, do as Gophers do" by Fumitoshi Ukai (https://go.dev/talks/2014/readability.slide#27)

Switch statements types

▶️ There are **two** forms:

- **expression** switches:
  - with the cases contain expressions that are **compared against the value** of the switch expression.
- **type** switches:
  - with the cases contain types that are **compared against the type** of a specially annotated switch expression.
  - **NOTE:** this will be covered on a separate session about interfaces in Go.

Expression switches (1/2)

- ✓ Switch cases evaluate cases from top to bottom, stopping when a case succeeds (the other cases are skipped)
  - If no case matches and there is a "**default**" case, its statements are executed.
    - There can be at most one "**default**" case;
    - "**default**" case may appear anywhere in the "**switch**" statement;

```
func main() {
    fmt.Print("Go runs on ")
    switch os := runtime.GOOS; os {
    case "darwin":
        fmt.Println("OS X.")
    case "linux":
        fmt.Println("Linux.")
    default:
        // freebsd, openbsd,
        // plan9, windows...
        fmt.Printf("%s.", os)
    }
}
```

- ✓ A **case** body breaks automatically, unless it ends with a **fallthrough** statement (possibly labeled).

Expression switches (2/2)

- ✓ **Switch** without a condition is the same as `switch true`.
  - This construct can be a clean way to write *long if-then-else chains:*

```
t := time.Now()
switch { // same as switch true {...}
case t.Hour() < 12:
     fmt.Println("Good morning!")
case t.Hour() < 17:
     fmt.Println("Good afternoon.")
default:
     fmt.Println("Good evening.")
}
```

`fallthrough` statement

- ✓ A "**fallthrough**" statement transfers control to the **first statement of the next** `case` **clause** in an expression "**switch**" statement.
- ✓ It may be used only as the **final non-empty statement** in such a clause.

`for` loop statement

- A "**for**" statement specifies **repeated** execution of a **block**.
- Go has *only one looping construct*, the **for** loop.

Basic `for` loop

- ✓ The basic `for` loop has three components separated by semicolons:
  - the **init** statement: executed **before the first** iteration
  - the **condition** expression: evaluated **before** every iteration
  - the loop will stop iterating once the boolean condition evaluates to `false`
  - the **post** statement: executed **at the end** of every iteration

- ✓ *Unlike* other languages like C, Java, or Javascript there are no parentheses surrounding the three components of the for statement and the braces **{ }** are *always required*.
  - Also the *semicolons* are required unless there is *only a* ***condition***.

- ✓ For statements with single condition (*emulate traditional while loop*):

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

  - Explanations:

```
for cond { S() }    // is the same as    for ; cond ; { S() }
for      { S() }    // is the same as    for true     { S() }
```

- ✓ Forever loop:

```
for {
    // loop forever
}
```

`for range` loop

- ✓ To iterate over **arrays**, **slices**, **maps**, **strings** or values received on a **channels** we should use **range** statement.
  - *Reminder*: unused variables can be omitted with _ variable name.

```
for i, num := range numbers { ... }
for i := range numbers { ... }
for range numbers { ... }
for _, pop := range population { ... }
```

- ✓ For each iteration, iteration values are produced as follows if the respective iteration variables are present:

Range expression			1st value			2nd value	
array or slice	a	[n]E, *[n]E, or []E	index	i	int	a[i]	E
string	s	string type	index	i	int	see below	rune
map	m	map[K]V	key	k	K	m[k]	V
channel	c	chan E, <-chan E	element	e	E		

Common pitfall with `for range` loop

- ✓ Common pitfall with range for loop: Go uses a **copy of the value** *instead of the value itself* within a **range** clause:
  - Mutation of the loop variable state can take no effect.
  - Don't get an address of the loop variable.
  - This copy gets reused throughout the range clause, which leaves our `listOfPointers` slice full of three references to the same pointer (the copy pointer).
  - To solve this you can use temporary variable **OR** address variable using index expression:

```go
func main() {
    list := []string{"one", "two", "three"}
    var listOfPointers []*string = make([]*string, len(list))
    for i, value := range list {
        listOfPointers[i] = &value // Never obtain address of for-range value!

        // To fix this issue you can address variable using index OR use temporary variable:
        // listOfPointers[i] = &list[i]                // OK
        // tmp := value; listOfPointers[i] = &tmp      // OK
    }

    for _, v := range listOfPointers {
        fmt.Print(*v, " ") // will always print "three three three"
    }
}
```

`for range` with array, pointer to array or slices copy

- ✓ For an *array*, *pointer to array*, or *slice* value, the *index iteration* values are produced in increasing order, starting at element index **0**.
- ✓ If at most one iteration variable is present, the *range loop* produces *iteration values* from **0** up to **len(a)-1** and does not index into the array or slice itself.
- ✓ For a **nil slice**, the number of iterations is **0**.

`for range` with strings

- ✓ For a *string* value, the "range" clause iterates over the Unicode code points (**runes**) in the string starting at byte index **0**.
  - This loop will decode one UTF-8-encoded **rune** on each iteration.
- ✓ On successive iterations:
  - the *index value* will be the **index of the first byte** of successive UTF-8-encoded code points in the string
  - the *second value*, of type rune, will be the value of the corresponding code point (**rune**).

```
func main() {
    const nihongo = "日本語"
    for index, runeValue := range nihongo {
        fmt.Printf("%#U starts at byte position %d\n", runeValue, index)
    }
    // OUTPUT:
    // U+65E5 '日' starts at byte position 0
    // U+672C '本' starts at byte position 3
    // U+8A9E '語' starts at byte position 6
}
```

- ✓ If the iteration encounters an **invalid UTF-8 sequence**, the second value will be '**0xFFFD**', the Unicode replacement character, and the next iteration will advance a single byte in the string.

- ✓ If a **for range loop** isn't sufficient for your purposes, chances are the facility you need is provided by a package in the library.
  - The most important such package is unicode/utf8, which contains helper routines to *validate*, *disassemble*, and *reassemble* UTF-8 strings.

`for range` with maps

- ✓ The **iteration order** over *maps* is **not specified** and is not guaranteed to be the same from one iteration to the next.
- ✓ If a map entry that has not yet been reached is **removed during iteration**, the corresponding iteration value will not be produced.
- ✓ If map entry is **created during iteration**, that entry may be produced during the iteration **or** may be skipped.
  - The choice may vary for each entry created and from one iteration to the next.
- ✓ If the map is nil, the number of iterations is **0**.

`for range` with channels

- ▶️ **Outside of the scope of today's session.**
- ▶️ This will be covered on the session about Goroutines and Channels.

`break` statement

- ✓ A "**break**" statement terminates execution of the innermost "**for**", "**switch**", or "**select**" statement within the same function.
- ✓ "**break**" statements can be used with **labels**.
  - Labels must be that of an enclosing "**for**", "**switch**", or "**select**" statement, and that is the one whose execution terminates.
  - Example:

```
OuterLoop:
for i = 0; i < n; i++ {
    for j = 0; j < m; j++ {
        switch a[i][j] {
        case nil:
            state = Error
            break OuterLoop
        case item:
            state = Found
            break OuterLoop
        }
    }
}
```

`continue` statement

- ✓ A "**continue**" statement begins the next iteration of the innermost "**for**" loop at its *post statement*.
- ✓ The "**for**" loop must be *within the same function*.
- ✓ "**continue**" statements can be used with **labels**.
  - Labels must be that of an enclosing "**for**" statement, and that is the one whose execution advances.
  - Example:

```
RowLoop:
for y, row := range rows {
    for x, data := range row {
        if data == endOfRow {
            continue RowLoop
        }
        row[x] = data + bias(x, y)
    }
}
```

`goto` statement

- ✓ A "**goto**" statement transfers control to the statement with the corresponding label within the same function.
- ✓ Executing the "**goto**" statement must not cause any variables to come into scope that were not already in scope at the point of the **goto**.
  - Incorrect example (*because the jump to label `L` skips the creation of `v`*):

```
goto L  // BAD
v := 3
L:
```

- ✓ A "goto" statement outside a block **cannot jump to a label inside that block.**
  - Incorrect example (*because the label `L1` is inside the "for" statement's block but the `goto` is not*):

```
if n % 2 == 1 {
    goto L1 // BAD
}
for n > 0 {
    f()
    n--
    L1:
    f()
    n--
}
```

What was NOT covered?

▶️ Some statements were not be covered:

- **send/receive** to/from the channels statements
- **select** statement
- **type switch**
- **go** statement
- **return** statement
- **defer** statement

▶️ They will be discussed in the appropriate session :)

Next time...

▶️ Session05:

Error handling and best practices, panic, and recovery

- Panic
- Recovery from panic
- Error handling
- Libraries to help with error handling
- Best practices

Closing words

▶️ We shared the Go Course feedback form with you, please take your time and provide a feedback

▶️ Homework?

- Clone the repository: cdarwin/go-koans (https://github.com/cdarwin/go-koans)
- Run `go test`
- Make the failing tests pass, by replacing these types of `__variables__` with real values.

Thank you

Golang course by Exadel

13 Oct 2022

Sergio Kovtunenko
Lead backend developer, Exadel
skovtunenko@exadel.com (mailto:skovtunenko@exadel.com)
https://github.com/skovtunenko (https://github.com/skovtunenko)
@realSKovtunenko (http://twitter.com/realSKovtunenko)