Programming in Go

Matt Holiday
Christmas 2020

Basic Types

Keywords & symbols

Only 25 keywords; you may not use these as names:

```
break        default      func         interface    select
case         defer        go           map          struct
chan         else         goto         package      switch
const        fallthrough  if           range        type
continue     for          import       return       var
```

Plus a bunch of operators & symbols:

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

Predeclared identifiers

You can use these as names, shadowing the built-in meaning, but you really don't want to do that!

Constants:

```
true false iota nil
```

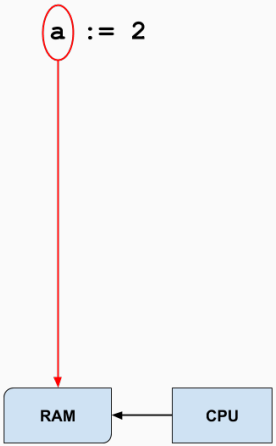
Types:

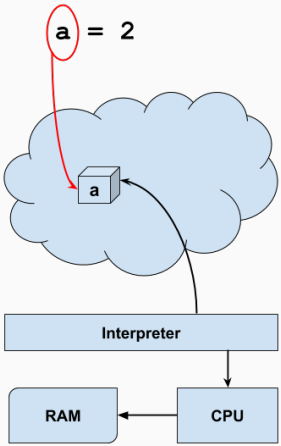
```
int int8 int16 int32 int64
uint uint8 uint16 uint32 uint64 uintptr
float32 float64 complex64 complex128
bool byte rune string error
```

Functions:

```
make len cap new append copy close delete
complex real imag
panic recover
```

Machine-native vs interpreted





Integers

“Unsized” integers default to the machine’s natural wordsize:

- 64 bits on my laptop
- 32 bits on my Raspberry Pi

`int` is the default type for integers in Go, even lengths

Signed	Unsigned
`int`	`uint`
`int64`	`uint64`
`int32`	`uint32`
`int16`	`uint16`
`int8`	`uint8`

“Real” and “imaginary” numbers

Non-integers are represented in floating point:

- floating point numbers:
  `float32 float64`

- complex (imaginary) floating point numbers:
  `complex64 complex128`

Don’t use floating point for monetary calculations!

Try a package like Go money

Simple declarations

Anywhere:

```
var a int

var (
    b = 2
    f = 2.01
)
```

Only inside functions:

```
c := 2
```

Number conversions

Conversions may change the value

```
func main() {
    a := 2
    b := 2.01

    fmt.Printf("a: %-4v %[1]T\n", a)   // fmt.Printf("a: %-4d %[1]T\n", a)
    fmt.Printf("b: %-4v %[1]T\n", b)   // fmt.Printf("b: %-4.2f %[1]T\n", b)

    var size float32 = 1.9

    y := int(size)       // truncated to 1
    z := float32(y)      // still 1.0 from 1
}
```

Once the number's been rounded down, it stays that way

Simple types

Special types:

- `bool` (boolean) has two values `false`, `true`

  these values are **not** convertible to/from integers!

- `error`: a special type with one function, `Error()`

  an `error` may be nil or non-nil

- Pointers are physically addresses, logically opaque

  a pointer may be nil or non-nil
  no pointer manipulation except through package `unsafe`

Initialization

Go initializes all variables to “zero” by default if you don’t:

- All numerical types get `0` (float `0.0`, complex `0i`)
- `bool` gets `false`
- `string` gets `""` (the empty string, length 0)
- Everything else gets `nil` :
  - pointers
  - slices
  - maps
  - channels
  - functions (function variables)
  - interfaces
- For aggregate types, all members get their “zero” values

Constants

Only numbers, strings, and booleans can be constants (immutable)

Constant can be a literal or a compile-time function of a constant

```
const (
    a = 1                   // int
    b = 2 * 1024            // 2048
    c = b << 3              // 16384

    g uint8 = 0x07          // 7
    h uint8 = g & 0x03      // 3

    s = "a string"
    t = len(s)              // 8
    u = s[2:]               // SYNTAX ERROR
)
```

Examples

```
// x and y get the values passed in by the caller

func do(x, y int) int {
    const t = 21            // type int by default
    const z = false         // type bool from the value

    var i uint8 = 255       // explicit type uint8
    var j = 256             // type int by default
    var k int               // 0 by default

    var m                   // SYNTAX ERROR, no type/value
    var n = nil             // SYNTAX ERROR, no type

    v := 0                  // short declaration, int
    w := func() { . . . }   // short declaration, function

    return k
}
```

Examples

```
// explicit conversion is required for integer types

func do(x, y int) int {
    k := x + y              // k int
    m := uint32(k)          // int conversion
    n := 3 * m              // still uint32

    k = m                   // TYPE MISMATCH
    k = int(m)              // int conversion

    var i uint8 = 255

    j := i++                // SYNTAX ERROR
    b := k = 0              // SYNTAX ERROR

    return m                // TYPE MISMATCH
    return int(m)           // int conversion
}
```