Carl Kingsford, 02-201, Fall 2015

Lecture 6: Types and Expressions

Types

Since the computer only operates on bits, we need a way to specify how to interpret particular sets of bits. Do these bits represent an integer? a string? a real number?

Go (and many programming languages) do this via types, which we saw a little bit last time. Now, we'll see more of the types that Go has built in.

Number-based types

Type	Data	Examples
int	Positive or negative integers	3,-200,40,42
uint	Non-negative integers (u = unsigned)	0, 3, 7, 11, 13
bool	Holds true or false	true
float64	Real, floating point number	3.14159, 12e-3, 0.23
complex128	Complex number (real, imaginary)	10 + 3i
string	Holds a sequence of characters	"Hello, world"

Some example variable definition:

```
var m uint = 10
var small bool = true
var big bool = m > 10
var e, pi float64 = 2.7182818285, 3.14
var name string = "Carl"
var root complex64 = 3 + 7i
```

Literals

Explicit values for variables are called literals.

- Integer literals: a sequence of digits 0,1,2,...,9

```
72
6402
000734
0xFF    // hexadecimal literals start with 0x
```

- String literals: a sequence of characters between quotes "

```
"Hi there”
"😄"        // unicode characters supported in strings
"1+三=4"
"3.14159"   // this is a string NOT a number
```

- `bool` (Boolean) literals: either `true` or `false`

```
true
false
```

- Floating point (real) literals: a number with a "." or "e"

```
7.
7.0
.32456
1.21212121
12E2
10E+3
11e-2
```

aEb means a × 10b.

- Imaginary literal: floating point literal with `i` after it

```
7.0i
7i
1e-5i
```

Items in an expression must have the same type

```
var a float64 = 3.0          // ok!
var b float64 = "4.0"        // ERROR! “4.0” is a string
var c int = 3.0              // ERROR! 3.0 is not an int
var d string = 7000           // ERROR!
var e int = 2                 // ok
var f uint = e             // ERROR! e is an int not a uint
var ok bool = 0                  // ERROR! 0 is not a bool
var ok2 bool = e > 1        // ok: boolean expression

var scale int = 2               // ok
var t float64 = 2.3*scale   // ERROR! 2.3 is a float, scale is an integer
var t2 float64 = 2*scale        // ERROR! 2*scale is an integer
```

Everything in a Go expression must have the same type.

Type converstions

You can change the type of a variable in an expression by type casting.

You use the syntax: TYPE(EXPRESSION) to change the type of EXPRESSION to TYPE.

```
var time float64 = 7.2      // ok
var r int = time                // ERROR! time not an int
var round int = int(time)   // ok!!! round will equal 7
```

You know that time is 7.2, but Go doesn’t know that, so it trusts you that you want to change time to an int.

When converting a floating point number to an int, Go will throw away the fractional part.

Conversion challenges

```
var a, b float64 = 7.6, -13.9
var c, d int = int(a), int(b)
```

Q: What values do c and d have?

```
var u int = -70
var q uint = uint(u)
```

Q: What value does q have?

Variables have limited range

Type	Min	Max
int	-9223372036854775808	9223372036854775807
uint	0	18446744073709551615
float64	-1.797693134862315708145274237317043567981e+30	1.797693134862315708145274237317043567981e+308

Question: What does this print?

```
var i int = 9223372036854775807
fmt.Println(i+1)
```

Go has types that let you specify how many bits they use:

Type	Number of bits
int	32 or 64 depending on your computer
uint	32 or 64 depending on your computer (but always same size as int)
int8 / uint8	8
int16 / uint16	16
int32 / uint32	32
int64 / uint64	64
float32	32
float64	64
complex64	32 for each of the real and imaginary parts
complex128	64 for each of the real and imaginary parts
byte	another word for int8
rune	another word for int32

Tip: use int, float64, and complex128 unless you have memory limitations.

`string` types

A variable that can hold a string has type `string`:

```
1 | var name string = "Carl"
```

Summary of types

- Types in an expression must agree.
- Be sure you don't corrupt your data by converting to the wrong type.
- Everything else is basically details that you have to know to program, but that shouldn't be forefront in your mind.

Expressions & Operators

Operations on integers

`a + b` addition

`a - b` subtraction

`a * b` multiplication

`-a` negation

`+a` doesn't do anything, but available for symmetry with -

`a / b` ***integer*** division: 2/3 = 0; 10/3 = 3; -10/3 = -3
results are truncated toward 0

If q = x / y and r = x % y then
x = q*y + r and |r| < |y|

`a % b` *modulus* (aka remainder):
13 % 2 = 1; 10 % 2 = 0; 10 % 3 = 1
-10 % 3 = -1 (since -10 = 3*(-3) - 1)

Increment and decrement on integers

Adding and subtracting 1 is so common there is a special notation for it:

`a++` is the same as `a = a + 1`

`a--` is the same as `a = a - 1`

Operations on real values

The standard mathematical operations on real values are supported:

Real-valued division is of limited precision:

```
2.0/3.0 = 0.6666666666666666
10.0/3.0 = 3.3333333333333335
-10.0/3.0 = -3.3333333333333335
```

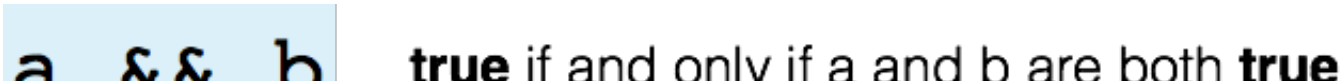
Example expressions

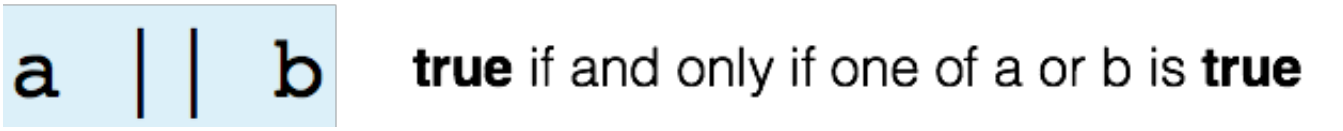
```
a + b / 3 + 2
(a+b) / 3 + 2
-a*(3+c - d)
```

Boolean variables and operators

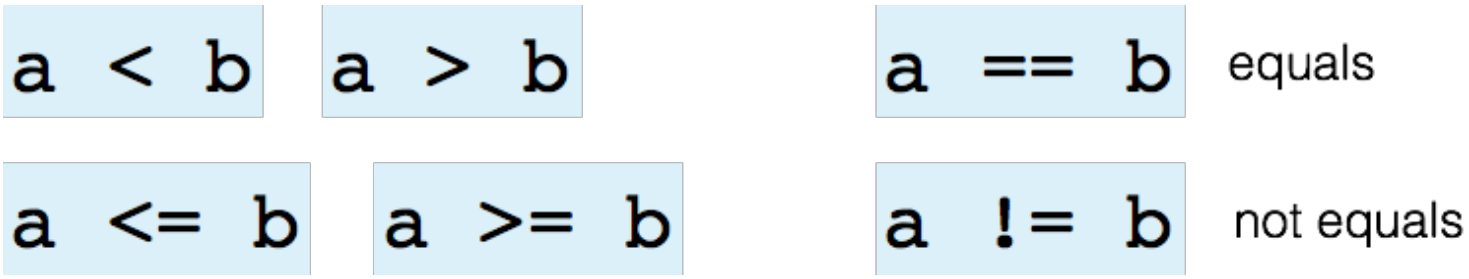
A Boolean variable is a variable that can hold two possible values: either `true` or `false`.

Boolean operators combine boolean variables:





Go also has comparison operators, the result of which is a Boolean value:



Examples with Boolean values:

Examples, true or false?

```
a=10
b=50
```

`a>10 && b > 20` **false**

`a==10 && b < 100 && a*b > 1000` **false**

`a>20 || b < 51 || b-a*b > 0` **true**

`a=10 && b=50` **syntax error!**

`a==10 && b >= 100 || b == 50` **true**

`b==50 || a == 10 && b >= 100` **true**

`a>5 && b>20 || a==0 && b==0` **true**

`a>5 || b>20 && a==0 || b==0` **true**

`a>5 || (b>20 && a==0) || b==0` **true**

Packages

- Packages are collections of functions you can use in your program.
- Go provides many built-in packages: see http://golang.org/pkg/
- Enable the use of a package with:

```
import "packageName"
```

at the top of your program.

- If you need to import lots of packages, you can write:

```
import (
    "package1"
    "package2"
    "package3"
)
```

at the top of your program.

- The `fmt` package provides the `fmt.Print` and `fmt.Println` functions we've used a lot.

The `math` package

The `math` package contains many funtions related to mathematical operations. For example:

```
func Abs(x float64) float64
func Min(x, y float64) float64
func Pow(x, y float64) float64
```

To use the math package, put `import "math"` at the top of your program. Then these functions are avaliable with the following syntax:

```
math.Abs(x)
math.Min(x,y)
math.Pow(x,y)
```

When using a function `F` from a package `X`, you write `X.F`.

The `strings` package

The `strings` package provides many functions to operate on strings. This example tests whether one string has another as a substring:

```
var a string = "hi, there"
var b string = "the"
if strings.Contains(a, b) {
    fmt.Println("String a contains string b!")
}
```

- A very large part of programming in practice is looking up how to use functions in existing packages.

The `strconv` package and `error`

Suppose we have an string "42" that we want to convert to an `int`. We cannot do:

```
// BAD CODE BAD CODE BAD CODE BAD CODE
var s string = "42"
var x int64
x = int64(s) // WRONG!
// BAD CODE BAD CODE BAD CODE BAD CODE
```

The reason we can't do this is that converting a string to an integer involves more than just relabeling its type. It requires interpreting the string in a way we understand as decimal notation. Instead, we have to use a function to do this conversion.

```
var err error
x, err = strconv.ParseInt(s, 10, 64)
```

What's going on here? `strconv.ParseInt` is a function that takes 3 parameters:

- the string to convert
- the base of the integer that the string is written in
- the number of bits used to represent the integer

It also returns **2** values:

- the value in the string as an `int64`
- an error value that indicates whether the operation work

What is an `error` value? `error` is a type that can hold either an error code or the special value `nil` indicating no error.

> **Test yourself:** What happens if you run this code:
>
> ```
> var x int64
> var err error
> x, err = strconv.ParseInt("StampyCat", 10, 64)
> fmt.Println("x=", x, "err=", err)
> ```

Finally, note that `strconv.ParseInt` always returns an `int64` --- if you want a different size int, you must convert the result you get.

The `strconv` package contains many functions to convert between various types and `string`s.

Next time, we'll see how to handle the case when you get an error like the one above.

> **On your own**: What happens when you run this code:
>
> ```
> var x string
> x = string(0x1f601)
> fmt.Println(x)
> ```

> Why?

Summary

- Every variable has a type that tells Go how to interpret the bits that represent that variable.
- In Go, everything in an expression must have the same type.
- You can convert between types by using the type name like a function: int(myFloatVar).
- Consistent with other types: string(103) reinterprets the number 103 as a string. It does not turn 103 into a string "103" of the decimal representation of 103.
- Packages provide lots of useful pre-defined functions, one of which is to convert numbers into strings (and vice versa).

Glossary

- package: a group of functions that you can import to use in your program
- literal: an explict value like 23, "hi", or 2.3 in your program
- type casting: changing the type of an expression
- error: is a special type that holds an error code
- Boolean: a type that can hold `true` or `false`