

Orale Vigna

Zucchero sintattico

Con "zucchero sintattico" si intendono i costrutti sintattici che non cambiano la struttura del programma ma soltanto la forma. L'assegnazione breve è un caso evidente di zucchero sintattico.

Program Structure

Go is an imperative language: the program has a state that is manipulated during its execution. The state is represented by the variables.

A main function as starting is required (usually the first func), otherwise a build-error will occur.


`func main` must have no arguments and no return values results.

When the program executes, after initializations the first function called (the entry-point of the program) will be `main.main()`. The program exits, immediately and successfully, when `main.main` returns.

1 Declarations

A declaration names a program entity and specifies some or all of its properties.

There are four major kinds of declarations: `var`, `const`, `type`, and `func`.

 Declaration Example

```
1 //package main
2 //import . "fmt"
3
4 const boilingF = 212.0
5
6 func main() {
7     var f = boilingF
8     var c = (f - 32) * 5 / 9
9
10    Printf("boiling point = %g°F or %g°C\n", f, c)
11 }
12
13 /* The constant boiling is a package-level declaration (as is main), whereas the
14 variables f and c are local to the function main. The name of each package-level
15 entity is visible not only throughout the source file that contains its
16 declaration, but throughout all the files of the pack-age. By contrast, local
17 declarations are visible only within the function in which they are declared and
18 perhaps only within a small part of it */
```

1.1 Function declaration

```
func function_name(parameter_list) return_value_list { body }
```

A function declaration has:

- a name
- an optional list of parameters (the variables whose values are provided by the function's callers). They are separated by a comma, and after the name of each parameter must come its type.
- an optional list of results, and the function body, which contains the statements that define what the function does. The result list is omitted if the function does not return anything

The body of the functions is enclosed between braces `{ }`

Execution of the function begins with the first statement and continues until it encounters a return statement or reaches the end of a function that has no results. Control of the program and any results are then returned to the caller.

2 Variables

A var declaration creates a variable of a particular type, attaches a name to it, and sets its initial value. Each declaration has the general form `var name type = expression`

Either the type or the `= expression` part may be omitted, but not both.

- If the type is omitted, it is determined by the initializer expression.
- If the expression is omitted, the initial value is the zero value for the type, which is:
 - `0` for numbers, `false` for booleans, `""` for strings
 - `nil` for interfaces and reference types (slice, pointer, map, channel, function).The zero value of an aggregate type like an array or a struct has the zero value of all of its elements or fields.

This way a variable always holds a defined value of its type: there are no uninitialized variables

A variable is a piece of storage containing a value. When they are created by declarations, they are identified by a name. Some others are however identified only by expressions like `x[i]` or `x.f`.

- All these expressions read the value of a variable (except when they appear on the left side of an assignment, in which case a new value is assigned to the variable)

2.1 Short variable declaration (Dichiarazione breve)

```
go
1  t := 1 // var t int = 1 // variable_name = [expression] // the type is based on the expression
2  x, y := 3, 4 // multiple assignments (var x, y int = 5, 9)
```

Within a function, an alternate form called a short variable declaration may be used to declare and initialize local variables. It takes the form `name := expression`, and the type of *name* is determined by the type of *expression*.

It's important to remember that `:=` is a declaration, whereas `=` is an assignment. A multi-variable declaration should not be confused with a tuple assignment, in which each variable on the left-hand side is assigned the corresponding value from the right-hand side:

```
i, j = j, i // swap values of i and j
```

2.1.1 Indebolimento dell'assegnamento multiplo parallelo

L'assegnamento multiplo parallelo ha un indebolimento della regola della variabile nuova:

```
var x int; x := 1 non compila, x è già dichiarato.
```

`var x int; x, y := 5, 6` compila, perchè il Go assegna un valore alla variabile `x` già dichiarata, e invece dichiara `y`. Per fare però in modo che assegni un valore a una variabile già dichiarata, nell'assegnamento multiplo deve esserci almeno una nuova variabile a sinistra da dichiarare. Esempio:

```
var x, y int; x, y := 5, 6 non compila.
```

From the book: "One subtle but important point: a short variable declaration does not necessarily declare all the variables on its left-hand side. If some of them were already declared in the same lexical block, then the short variable declaration acts like an assignment to those variables."

2.2 Lifetime of variables

The escaping concept [...] page 55 of the book -> how does the compiler decide if it's better to allocate in the heap or in the stack.

Not relevant for the exam

2.3 Scope of variables

A variable (constant, type, function) is only known in a certain range of the program, called the scope.

- Variables declared outside of any function have global (or package) scope: they are visible and available in all source files of the package.
- Variables declared in a function have local scope: they are only known in that function, the same goes for parameters and return-variables.

3 Assignment

The value held by a variable is updated by an assignment statement, which in its simplest form has a variable on the left of the `=` sign and an expression on the right: `variable = expression`

GO

```
1  x = 1           // named variable
2  *p = true       // indirect variable
3  person.name = "bob" // struct field
4  count[x] = count[x] * scale // array or slice or map element
```

3.1 Assignability

Assignments statements are an explicit form of assignment, but there are many places in a program where an assignment occurs implicitly:

- a function call implicitly assigns the argument values to the corresponding parameter variables
- a return statement implicitly assigns the return operands to the corresponding result variables
- a literal expression for a composite type such as `medals := []string{"gold", "silver"}` implicitly assigns each element as if it had been written like this: `medals[0] = "gold"; medals[1] = "silver"`

3.2 Tuple assignment (Assegnamento tupla)

Another form of assignment, known as tuple assignment, allows several variables to be assigned at once. All of the right-hand side expressions are evaluated before any of the variables are updated, making this form most useful when some of the variables appear on both sides of the assignment, as happens, for example, when swapping the values of two variables: `x, y = y, x`

4 Pointers

A pointer value is the address of a variable. A pointer is thus the location at which a value is stored. Not every value has an address, but every variable does.

With a pointer, we can read or update the value of a variable indirectly, without using or even knowing the name of the variable, if indeed it has a name.

If a variable is declared `var x int`, the expression `&x` (address of `x`) yields a pointer to an integer variable, that is a value of type `*int`.

The expression `*p` yields the value of the variable pointed to by `p`. But since `*p` denotes a variable, it may also appear on the left side of an assignment, in which case the assignment updates the value of the variable pointed to.

- Each component of a variable of aggregate type, a field of a struct or an element of an array, is also a variable and thus has an address too.
- Variables are addressable values. Expressions that denote variables are the only ones to which the address operator `&` may be applied.
- Two pointers are equal if and only if they both point to the same variable or if they are both `nil`

The zero value for a pointer of any type is `nil`

- `p != nil` is true if `p` points to a variable

[My notes]

4.1 Aliasing

Each time we take the address of a variable or copy a pointer, we create new aliases or ways to identify the same variable. Pointer aliasing is useful because it allows us to access a variable without using its name.

[!] It's not just pointers that create aliases; aliasing also occurs when we copy values of other reference types like slices, maps, and channels, and even structs, arrays, and interfaces that contain these types.

4.2 New() function

Another way to create a variable is to use the built-in function `new`. The expression `new(T)` creates an unnamed variable of type `T`, initializes it to the zero value of `T`, and returns its address, which is a value of type `*T`.

5 Type declaration

A type declaration defines a new named type that has the same underlying type as an existing type. The named type provides a way to separate different and perhaps incompatible uses of the underlying type so that they can't be mixed unintentionally: `type name underlying_type`

Type declarations most often appear at package level, where the named type is visible throughout the package, and if the name is exported (it starts with an upper-case letter), it's accessible from other packages as well.

For every type `T`, there is a corresponding conversion operation `T(x)` that converts the value `x` to type `T`. A conversion from one type to another is allowed if both have the same underlying type, or if both are unnamed pointer types that point to variables of the same underlying type; these conversions change the type but not the representation of the value.

The underlying type of a named type determines its structure and representation, and also the set of intrinsic operations it supports, which are the same as if the underlying type had been used directly.

```
go    Utilizzo dei Type Definition per dichiarare unità di temperatura diverse

1  type Celsius float64
2  type Fahrenheit float64
3
4  func CtoF(c Celsius) Fahrenheit {
5      return Fahrenheit(float64(c) * 9/5 + 32)
6  }
7
8  func main() {
9      c := Celsius(32.5) //var c Celsius = 32.5
10     f := Fahrenheit(CtoF(c)) //var f Fahrenheit = CtoF(c) = 90.5
11     Println(c, f)
12 }
```

5.1 Type alias

Type Alias (alias di tipo): `type intero = int`. Serve a definire dei type alias: sono dei nomi per richiamare i tipi diversamente. I type alias quindi denotano lo stesso tipo e sono solo modi diversi per riferirci ad esso.

- La `rune` e il `byte` sono rispettivamente il type alias dell' `int32` e dell' `int8`: si riferiscono quindi allo stesso tipo ma con un nome diverso. È zucchero sintattico.

```
go    Type Alias

1  type intero = int //creo un type alias intero di int (stessa cosa)
2  type rune = int32 //rune è di base un type alias di int32
3  type byte = int8 //byte è un type alias di int8
4
5  func main() {
6      var x intero = 5 //x è effettivamente un intero
7      Println(x)
8  }
```

6 Scope

A declaration associates a name with a program entity, such as a function or a variable. The scope of a declaration is the part of the source code where a use of the declared name refers to that declaration.

Scope and lifetime are not the same thing:

- the scope of a declaration is a region of the program text: it's a compile time property
- the lifetime of a variable is the range of time during execution when the variable can be referred to by other parts of the program: it's a run time property

6.1 Syntactic block

A syntactic block is a sequence of statements enclosed in braces like those that surround the body of a function or loop. A name declared inside a syntactic block is not visible outside that block. The block encloses its declarations and determines their scope.

- We can generalize this notion of blocks to include other groupings of declarations that are not explicitly surrounded by braces in the source code; we'll call them all lexical blocks. There is a lexical block for each for, if and switch statement, for each case in a switch statement, etc...

Declarations outside any function, that is at package level, can be referred to from any file in the same package.

- Imported packages, such as `fmt` in the `tempconv` example, are declared at the file level, so they can be referred to from the same file, but not from another file in the same package without another import.

6.1.1 Shadowing

Dichiarare in un contesto lessicale più piccolo (in un sotto-blocco) una variabile che esiste in un contesto lessicale più ampio.

```
GO
1 func main() {
2     n := 3;
3     if n > 0 {
4         n := 2
5         Println(n)
6     }
7     Println(n)
8     //Output: 2, 3. n è in shadowing
9 }
```

```
GO
1 func main() {
2     n := 3;
3     if n > 0 {
4         n, a := 2, 5
5         Println(n, a)
6     }
7     Println(n)
8     /* n non è in shadowing per via dell'indebolimento dell'assegnamento multiplo
9        parallelo: n non viene ridichiarata in oscuramento ma solo riassegnata,
10       mentre a viene dichiarata. */
11 }
```

A program may contain multiple declarations of the same name so long as each declaration is in a different lexical block. For example, you can declare a local variable with the same name as a package level variable.

When the compiler encounters a reference to a name, it looks for a declaration, starting with the innermost enclosing lexical block and working up to the universe block. If the compiler finds no declaration, it reports an "undeclared name" error. If a name is declared in both an outer block and an inner block, the inner declaration will be found first. In that case, the inner declaration is said to shadow or hide the outer one, making it inaccessible.

6.2 For scope

Not all lexical blocks correspond to explicit brace-delimited sequences of statements; some are merely implied.

The for loop creates two lexical blocks: the explicit block for the loop body, and an implicit block that additionally encloses the variables declared by the initialization clause, such as `i`. The scope of a variable declared in the implicit block is the condition,

post-statement (i++), and body of the for statement.

6.3 If, switch scope

Like for loops, if statements and switch statements also create implicit blocks in addition to their body blocks.

```
1 func main() {
2     f := func() int { return 2 }
3     g := func(x int) int { return x+1-1 }
4
5     if x := f(); x == 0 {
6         Println(x)
7     } else if y := g(x); x == y { // the x declared above is also visible here
8         Println(x, y)
9     } else { // the x and y declared above are also visible here
10        Println(x)
11    }
12    //Println(x, y) // compile error: x and y are not visible here
13 }
```

The second if statement is nested within the first, so variables declared within the first statement's initializer are visible within the second.

- Similar rules apply to each case of a switch statement: there is a block for the condition and a block for each case body.

Types can be:

- primitive like int, float, bool, string
- structured like struct, array, slice, map, channel
- interfaces which only describe the behavior of a type

Basic Data Types

1 Integers

Go provides both signed and unsigned integer arithmetic.

- There are four distinct sizes of signed integers: 8, 16, 32, and 64 bits represented by the types ints, int16, int32, and int64
- And corresponding unsigned versions uint8, uint16, uint32, and uint64

There are also two types called just int and uint that are the natural or most efficient size for signed and unsigned integers on a particular platform.

The type rune is a type alias for int32 and conventionally indicates that a value is a Unicode code point.

Signed numbers are represented in 2's complement form, in which the range of values of an n -bit number is from -2^{n-1} to 2^{n-1} . Unsigned integers use the full range of bits for non negative values and thus have the range that goes from 0 to 255.

2 Floats

Go provides two sizes of floating-point numbers, float32 and float64. Their arithmetic properties are governed by the IEEE 754 standard implemented by all modern CPUs.

- float32 provides approximately 6 decimal digits of precision
- float64 provides about 15 digits

Digits may be omitted before the decimal point (.104) or after it (1.)

In go non puoi dividere un float per un intero. Si risolve con `n := x.0`.

- A meno che non si faccia tra una costante e il float, in questo caso non c'è interferenza di tipo. Es: `2 * float` funziona.

- `n := 2; n * float` non funziona. Il go nel primo caso intende in maniera autonoma il 2 come 2.0 in ordine da portare a termine l'operazione.

3 Booleans

A value of type `bool`, or `boolean`, has only two possible values, `true` and `false`.

The conditions in `if` and `for` statements are `booleans`, and comparison operators like `==` and `<` produce a `boolean` result. The unary operator `!` is logical negation, so `!true` is `false`.

3.1 Algebra di boole

Con delle operazioni logiche che per i `booleans` sono diverse da quelle dell'algebra normale. Da questa derivano gli operatori logici: `||` or, `&&` and, `!` not

- `||` e `&&` sono binari (prendono 2 argomenti: `EXPR || EXPR`)
- `!` è unario (prende un argomento: `!EXPR`)

Funzionamento:

- `&&` restituisce `true` se entrambe le condizioni, e quindi entrambi i `booleans`, sono veri: `T && T = T`.
- `||` restituisce `false` se almeno una delle condizioni (e quindi almeno uno dei due `bool`) è vera. E' l'oppure inclusivo
- `!` prende un'espressione e lo converte nel valore booleano opposto. `!(a==b)` è uguale ad `a!=b`

3.1.1 Operatori di comparazione

In programming, *comparison operators* are used to compare values and evaluate down to a single `Boolean` value of either `true` or `false`.

- `<=`, `>=`, `!=`, `==`, `<`, `>`

3.2 Short-circuit behaviour

`Boolean` values can be combined with the `&&` (AND) and `||` (OR) operators, which have short-circuit behavior: if the answer is already determined by the value of the left operand, the right operand is not evaluated.

- This makes it safe to write expressions like this: `s != "" && s[0] == 'x'` where `s[0]` would panic if applied to an empty string.

3.4 Strings

Strings are a sequence of UTF-8 characters (the 1 byte ASCII-code is used when possible, a 2-4 byte UTF-8 code when necessary -> ASCII-characters are still stored using only 1 byte).

- A Go string is thus a sequence of variable-width characters (each 1 to 4 bytes), contrary to strings in other languages as C++, Java or Python that are fixed-width.
- The advantages are that Go strings and text files occupy less memory/disk space, and since UTF-8 is the standard, Go doesn't need to encode and decode strings as other languages have to do.

Strings are value types and immutable: once created you cannot modify the contents of the string; formulated in another way: strings are immutable arrays of bytes.

[Notes]

Structured types

1 Arrays

[Notes]

2 Slices

[Notes]

2.1 Append

[Notes]

2.2 Subslicing

[Notes]

3 Maps

The hash table is one of the most ingenious and versatile of all data structures. It is an unordered collection of key/value pairs in which all the keys are distinct, and the value associated with a given key can be retrieved, updated, or removed using a constant number of key comparisons on the average, no matter how large the hash table.

In Go, a map is a reference to a hash table, and a map type is written `map[K]V`, where `K` and `V` are the types of its keys and values. All of the keys in a given map are of the same type, and all of the values are of the same type, but the keys need not be of the same type as the values. The key type `K` must be comparable using `==`, so that the map can test whether a given key is equal to one already within it.

The built-in function `make` can be used to create a map:

```
ages := make(map[string]int)
```

We can also use a map literal to create a new map populated with some initial key/value pairs:

```
GO
1 ages := map[string]int{
2     "alice": 31,
3     "charlie": 34,
4 }
5 // which is equivalent to
6 ages := make(map[string]int)
7 ages["alice"] = 31
8 ages["charlie"] = 34
```

Map elements are accessed with this notation: `ages["alice"] = 32` and they are removed with the built-in function `delete`:

```
delete(ages, "alice") //remove element ages["alice"]
```

- Those operations are safe even if the element isn't in the map: a map lookup using a key that isn't present returns the zero value for its type.
- The order of map iteration is unspecified -> the order is random
- We can use the for range loop to iterate maps over their keys and their values

4 Structs

A struct is an aggregate data type that groups together zero or more named values of arbitrary types as a single entity. Each value is called a field. All of these fields are collected into a single entity that can be copied as a unit, passed to functions and returned by them, stored in arrays, and so on.

These two statements declare a struct type called `Employee` and a variable called `dilbert` that is an instance of an `Employee`:

```
GO
1 type Employee struct {
2     ID int
3     Name string
4     Address string
5     Position string
6     Salary int
7 }
8
9 var dilbert Employee
```

- The individual fields of `dilbert` are accessed using dot notation like `dilbert.Name`.

- Because `dilbert` is a variable, its fields are variables too, so we may assign to a field:
`dilbert.Salary -= 5000 ;`
or take its address and access it through a pointer:
`position := &dilbert.Position; *position = "Senior " + *position`
- The dot notation also works with a pointer to a struct:

GO

```
1 var employeeOfTheMonth *Employee = &dilbert
2 employeeOfTheMonth.Position += " (proactive team player)"
3 // last statement is equal to:
4 (*employeeOfTheMonth).Position += " (proactive team player)"
```

[Notes]

4.1 Struct literals

A value of a struct type can be written using a struct literal that specifies values for its fields.

GO

```
1 type Point struct{X, Y int}
2 p := Point{1, 2}
3 p2 := Point{Y: 3}
```

- The first form requires that a value is specified for every field and in the right order.
- In the second form a struct value is initialized by listing some or all of the fields names and their corresponding values.
 - If a field is omitted in this kind of literal, it is set to the zero value for its type.
 - The two forms cannot be mixed.

Struct values can be passed as arguments to functions and returned from them.

- When the structs are large, it's preferred to pass or to return them from functions using pointers for efficiency. `pp := &Point{1, 2}` same thing as `pp := new(Point); *pp = Point{1, 2}`

Functions

A function lets us wrap up a sequence of statements as a unit that can be called from elsewhere in a program, perhaps multiple times. Functions make it possible to break a large problem which requires many code lines into a number of smaller tasks. Furthermore, the same task can be invoked several times, so a function promotes code reuse.

1 Function declarations

A function declaration has a name, a list of parameters, an optional list of results and a body:

```
func name(parameter_list) (result_list) { body }
```

- The parameter list specifies the names and types of the function's parameters, which are the local variables whose values or arguments are supplied by the caller.
- The result list specifies the types of the values that the function returns.
 - If the function returns one unnamed result or no results at all, parentheses are optional and usually omitted.
 - If the result is left off the function will be called only for its effects.
 - Also the results may be named. In that case, each name declares a local variable initialized to the zero value for its type.

A function that has a result list must end with a `return` statement (unless execution cannot reach the end of the function due to an infinite for loop with no break or because the function ends with a call to panic).

The invocation happens in the code of another function: the calling function.

Signature

The type of a function is sometimes called its signature. Two functions have the same type or signature if they have the same sequence of parameter types and the same sequence of result types.

Every function call must provide an argument for each parameter, in the order in which the parameters were declared. Parameters are local variables within the body of the function, with their initial values set to the arguments supplied by the caller.

Call by value, call by reference

- Arguments are passed by value, so the function receives a copy of each argument: therefore modifications to the copy do not affect the caller.
- However, if the arguments contains some kind of reference, like a pointer, slice, map or function, then the caller may be affected by any modifications the function makes to variables indirectly referred by the argument.
 - That's because if you pass the memory address of that variable with `&`, then a pointer is passed to the function. Therefore the pointer is copied, not the data that it points to, and through the pointer the function changes the original value.
 - It's cheaper to pass a pointer rather than making a copy of the object.

1.1 Recursion

Functions may be recursive, that is, they may call themselves, either directly or indirectly.

1.2 Multiple return values

A function can return more than one result. The result of calling a multi-valued function is a tuple of values. The caller of such a function must explicitly assign the values to variables if any of them are to be used. To ignore one of the values, it's sufficient to assign it to the blank identifier:

```
num, err := strconv.Atoi(n); num, _ := strconv.Atoi(n)
```

In a function with named results, the operands of a return statement may be omitted. This is called a bare return.

- A bare return is a shorthand way to return each of the named result variables in order.

1.2.1 Blank identifier

The blank identifier `_` can be used to discard values, effectively assigning the right-side value to nothing

1.3 Functions as parameters

Functions can be used as parameters in another function. The passed function can then be called within the body of that function, and that's why it's commonly called a callback.

```
1 func Add(a, b int) {
2     Println("The sum of", a, "and", b, "is:", a+b)
3 }
4
5 func callback(y int, f func(int, int)) {
6     f(y, 2) // this becomes Add(1, 2) when called by line 10
7 }
8
9 func main() {
10     callback(1, Add)
11 }
```

For example, from the package `sort`, the function `sort.Slice(x any, func(i, j int) bool)` accepts a callback function to determine the behaviour of the sorting.

2 Functional types

They are used to organize not data like the other types, but rather the code.

They are declared like `var name func(arguments_types) (return_types)`

```
1 func nothing (x int, s string) (bool, int) {
```

```

2     return true, x*2
3 }
4
5 func main() {
6     var x func(int, string) (bool, int)
7     x = nothing
8     // also x := nothing
9     Println(x(5, "hi"))
10 }

```

The zero value of a function type is `nil`. Calling a nil function value causes a panic.

- Function values may be compared with nil, but are not comparable between each other.

We can change the behaviour of the functions depending on the code that it's passed to them:

```

GO
1 func product(x int, f func(int) int) int {
2     return f(x) // the function type f has access to the variable x because
3                 // they are both located in the same lexical block
4 }
5
6 func double(x int) int { return 2*x }
7
8 func triple(x int) int { return 3*x }
9
10 func main() {
11     Println(product(5, double))
12     Println(product(5, triple))
13     l := 100
14
15     f := func(x int) int { // function literal
16         return l*4 // f has access to all the variables declared within the
17                   // lexical block where its declared
18     }
19
20     Println(f(4), product(5, f), l) // 400 400
21
22     Println(product(10, func(y int) int { // anonymous function
23         return y*y
24     })))
25 }

```

2.1 Anonymous functions / Function literals (closures)

Named functions can be declared only at the package level, but we can use a function literal to denote a function value within any expression. A function literal is written like a function declaration, but without a name following the `func` keyword. It is an expression, and its value is called an anonymous function.

Function literals let us define a function at its point of use.

A function literal represents an anonymous function: `func(x, y int) int { return x + y }`

A function literal can be assigned to a variable or invoked directly:

```

GO
1 func main() {
2     z := 3
3     f := func(x, y int) int {
4         z++
5         return x + y + z
6     }
7     /* I can use z inside of the anonymous function because f it's a closure: it
8        can refer to variables in its lexical block and change them even outside */
9     Println(f(3, 2), z) // 9 4
10 }

```

```

11     func(m, n int) (bool, int) {
12         return true, m * n
13     }(10, 2) // true 20
14 }

```

That's because it's not a function that can stand on its own, but it can be assigned to a variable that is a reference to that function, and then it can be invoked as if the name of the variable was the name of the function.

- Like all functions they can be with or without parameters:

GO

```

1 func main() {
2     v := "Hello World"
3     func (u string) {
4         Println(u)
5     }(v) // Hello World
6 }

```

Function literals are *closures*: they may refer to variables defined in a surrounding function/lexical block. Those variables are then shared between the surrounding function and the function literal, and they survive as long as they are accessible.

- A closure inherits the scope of the function in which it is created.

2.1.1 Functions returning other functions

GO

```

1 func main() {
2     var f = Adder()
3     var d = Adder()
4
5     var AdderLiteral = func() func(int) int {
6         var x int
7         return func(delta int) int {
8             x += delta
9             return x
10        }
11    }
12
13    g := AdderLiteral()
14    Println("g:", g(1), g(20), g(300)) // 1 21 321
15    Println("f:", f(4), f(-30)) // 4 -26
16    Println("d:", d(3)) // 3
17
18 }
19
20 func Adder() func(int) int {
21     var x int
22     return func(delta int) int {
23         x += delta
24         return x
25     }
26 }

```

GO

```

1 func main() {
2     f := squares()
3     Println(f()) // "1"
4     Println(f()) // "4"
5     Println(f()) // "9"
6     Println(f()) // "16"
7 }
8
9 func squares() func() int {
10     var x int

```

```

11     return func() int {
12         x++
13         return x * x
14     }
15 }

```

Functions defined in this way have access to the entire lexical environment, so the inner function can refer to variables from the enclosing function as shown in the upper example.

- The function `squares` returns another function of type `func() int`. A call to `squares` creates a local variable `x` and returns an anonymous function that, each time it is called, increments `x` and returns its square. A second call to `squares` would create a second `x` and return a new anonymous function which increments that variable.

Functions therefore are not just code but can have a state. The anonymous inner function can access and update the local variables of the enclosing function `squares`. These hidden variable references are why we classify functions as reference types and why function values are not comparable.

Here the lifetime of a variable is not determined by its scope: the variable `x` exists after `squares` has returned within `main`, even though `x` is hidden inside `f`.

GO Lexical Closure for Exam

```

1  func g(f func()) func() {
2      return f
3  }
4
5  func main() {
6      x := 0
7      f := func() {
8          x++
9      }
10
11     Println(x) // 0
12     g(f)()
13     Println(x) // 1
14 }

```

3 Defer, Panic, Recover

Methods

A method is a function associated with a particular type.

Method declaration

A method is declared with a variant of the ordinary function declaration in which an extra parameter appears before the function name. The parameter attaches the function to the type of that parameter.

GO

```

1  package geometry
2
3  import "math"
4
5  type Point struct{ X, Y float64}
6
7  // traditional function
8  func Distance(p, q Point) float64 {
9      return math.Hypot(q.X - p.X, q.Y - p.Y)
10 }
11
12 // same thing, but as a method of the Point type
13 func (p Point) Distance(q Point) float64 {

```

```

14     return math.Hypot(q.X - p.X, q.Y - p.Y)
15 }

```

The extra parameter `p` is called the method's receiver.

- In Go, differently from other object-oriented languages, we don't use special names like `this` or `self` for the receiver, but we choose their names like we would for any other parameter.

In a method call, the receiver argument appears before the method name. This parallels the declaration, in which the receiver parameter appears before the method name.

GO

```

1  p := Point{1, 2}
2  q := Point{4, 6}
3  fmt.Println(Distance(p, q)) // "5" function call
4  fmt.Println(p.Distance(q)) // "5" method call

```

- There's no conflict between the two declaration of functions called `Distance` above. The first declares a package-level function called `geometry.Distance`, while the second declares a method of the type `Point`, so its name is `Point.Distance`.

Methods with a pointer receiver

Calling a function makes a copy of each argument value, so if a function need to update a variable or if one of its arguments is so large that it would be preferred to avoid copying it, then we must pass the address of the variable using a pointer.

The same thing applies to methods that need to update the receiver variable: we attach them to the pointer type, such as `*Point`

GO

```

1  func (p *Point) ScaleBy(factor float64) {
2      p.X *= factor
3      p.Y *= factor
4  }

```

- The name of this method is `(*Point).ScaleBy` (the parentheses are necessary).

Methods values and expressions

The selector `p.Distance` yields a method value, a function that binds a method (`Point.Distance`) to a specific receiver value `p`. This function can then be invoked without a receiver value, it needs only the non-receiver arguments.

GO

```

1  distanceFromP := p.Distance // method value
2  fmt.Println(distanceFromP(q)) // "5"
3
4  var origin Point // {0, 0}
5  fmt.Println(distanceFromP(origin)) // "√5"
6
7  scaleP := p.ScaleBy // method value
8  scaleP(2) // p becomes (2, 4)
9  scaleP(10) // p becomes (20, 40)

```

Domande orale Vigna

- Operatori `&`, `||` e `!`
- Varie forme di ciclo `for` in `go`
 - `For range`
 - 3 tipi di `for range`: stringhe, slice/array, mappe
 - Ciclo `for`: come viene eseguito?

- Break - interrompe ciclo; continua - interrompe l'esecuzione del corpo e passi all'istruzione successiva
- Come è la rappresentazione interna delle stringhe in go?
 - Conversione di una slice di rune
- Slice? Tutto a riguardo (appunti)
 - Differenza tra array e slice
 - Subslicing di una slice
 - operatore di subslice e a cosa si può applicare - slice di byte, stringhe, vettori che danno una slice
- Dichiarare una funzione dentro una funzione
- Puntatori
- Bit byte in int64 int int32
- In una mappa la chiave deve essere sempre diversa, non può essere uguale, ma il valore invece può esserlo.
- Switch (== se non c'è niente prima delle graffe vale true)
 - Switch x (x variabile o condizione)
- Differenza tra passare una slice / vettore a una funzione
- defer, panic e recover per gestire i segmentation fault
- Strutture e metodi

Esercizi

- Fare struct e fare una funzione che restituisca l'età moltiplicata per due
- Contare le parole di una stringa di rune usando comandi tipo unicode.IsLetter
- Trovare elementi comuni di due slice e stamparli
- Dichiarare una funzione che prende due argomenti da int a interi e restituisce una funzione da int e int
- Verificare la distanza tra due elementi di una mappa

Programma Programmazione I - 2023/2024

Il programma di massima del corso è il seguente (TWG=The Way to Go):

1. Introduzione al corso. Architettura del calcolatore. Che cos'è l'informatica. Linguaggi di programmazione (macchina, assembly, alto livello). Il calcolatore come macchina programmabile.
2. La macchina di von Neumann. Informazione (bit, byte,...). Caricamento in RAM del programma, fetch-decode-execute. Architettura della CPU: ALU e CU. Un esempio di CPU con relativo linguaggio assembly.
3. Ciclo di vita del software. Strumenti per la programmazione. Storia di go. Il primo programma in go [TWG4]. Il go tool. Compilazione. Esecuzione. Formattazione. Documentazione. [TWG3]
4. Discussione degli aspetti lessicali e sintattici. Commenti [TWG4]. Struttura generale di un programma go: programma, pacchetti, sorgenti. La libreria standard. [TWG4]
5. Variabili: nome, tipo, valore, visibilità (scope). Tipi. Classificazione dei tipi (tipi di base, tipi composti, interfacce). Dichiarazione, assegnamenti e assegnamenti multipli, short-assignment. [TWG4]
6. I/O di base: fmt.Println, fmt.Print, fmt.Scan. Tipi di base numerici (int, float64). Espressioni numeriche. Conversioni. Variabili inutilizzate e blank variable. [TWG4]
7. Selezione binaria (if). Il tipo bool e gli operatori booleani. Esercizi. [TWG5]
8. Ancora sull'if: variabili locali all'if (locali ai blocchi; locali al costrutto). Esempi.
9. Il ciclo (for): versione unaria, ternaria, zeraria. Esercizi. [TWG5]
10. I caratteri (ASCII, Unicode, UTF-8). Tipo rune. Tipo string: differenze fra raw e UTF-8. Funzione len. Quarta forma del ciclo for (range). [TWG4]
11. Funzioni: parametri, segnatura argomenti. Passaggio per valore. Valori restituiti. Valori restituiti con nome. [TWG6]
12. Esercizi con i cicli semplici e funzioni. Istruzioni break e continue. [TWG5]
13. Esercizi con i cicli annidati.
14. Rappresentazione dell'informazione. Notazione posizionale. Rappresentazione degli interi negativi. Range di rappresentazione, overflow. Tipi interi a lunghezza fissa. Cenni alla rappresentazione dei reali: virgola fissa e mobile (standard IEEE 754). Cenni al tipo complex. [Dispense, TWG4, TWG5]
15. Selezione multiaria (switch). [TWG5]
16. Esercizi. Pacchetto strconv e pacchetto strings. [TWG]

17. Puntatori: operatori * e &. La funzione new. [TWG4]
18. Type: alias e definizioni. Struct. Esercizi con puntatori e struct.
19. Array e slice. Inizializzatori. Applicazione dei for range. Funzione append. [TWG7]
20. Esercizi. Subslicing. fmt.Printf. Argomenti da riga di comando.
21. Generazione numeri pseudocasuali. Pacchetto math. Esercizi.
22. Mappe. Applicazione dei for range. Conversione di string a []rune. Esercizi. [TWG8]
23. Ricorsione. Stack di esecuzione. [TWG6]
24. Esercizi sulla ricorsione.
25. Metodi. Interfacce (cenni). Esempi: Stringer, Reader, Writer. [TWG10, TWG11 (cenni)]
26. Grafica con il pacchetto github.com/holizz/terrapin. Esempio semplice. Frattali e curva di Koch. Un motore di ricerca.
27. Tipi funzione e chiusure (cenni). Esempi dalle librerie (ordinamento, shuffling, ricerca in stringhe). L'esempio dell'integrazione numerica (metodo Monte-Carlo). [TWG6]
28. Ordinamento e ricerca tramite funzioni di libreria. I/O di base. [TWG6, TWG12, TWG13]
29. I/O avanzato. File, panic, defer e recover. [TWG6, TWG12, TWG13]
30. Pacchetti e struttura. Visibilità. Documentare un pacchetto.
31. Esercitazione: lettura di un file di testo con formato prestabilito, espressioni regolari, gestione degli errori di I/O e di parsing.
32. Testing unitario e funzionale. (E2E) [TWG13]
33. Il linguaggio C. Il gcc. Differenze sintattiche: punto-e-virgola, parentesi nelle strutture di controllo, dichiarazioni di variabili, tipi e funzioni. Differenze nelle strutture di controllo del flusso (switch, while, do-while). Uso di istruzioni semplici nelle strutture di controllo. Inclusione vs. importazione.
34. Assenza di stringhe, slice, mappe. Uso dei char[] per le stringhe. Tipi elementari e dipendenza dal compilatore; uso di tipi specifici (stdint.h, bool.h). Cast impliciti. Definizione di macro. Funzioni di libreria. Parametri da riga di comando.
35. Puntatori, aritmetica dei puntatori. Gestione della memoria: malloc, free.
36. Goroutine e canali.