Closures

This lesson covers the important concept of naming functions via closures.

WE'LL COVER THE FOLLOWING

- Using function literals
- The `defer` keyword and lambda functions

Using function literals

Sometimes, we do not want to give a function a name. Instead, we can make an **anonymous function** (also known as a *lambda* function, a *function literal*, or a *closure*), for example:

```
func(x, y int) int { return x + y }
```

Such a function cannot stand on its own (the compiler gives the error: `non-declaration statement outside function body`), but it can be assigned to a variable which is a reference to that function:

```
fplus := func(x, y int) int { return x + y }
```

Then it can be invoked as if `fplus` was the name of the function:

```
fplus(3,4)
```

or it can be invoked directly:

```
func(x, y int) int { return x + y } (3, 4)
```

Here is a call to a lambda function, which calculates the sum of integer floats till 1 million. The *gofmt* reformats the lambda function in this way:

```
func() {
    sum = 0.0

    for i := 1; i<= 1e6; i++ {
        sum += i
    }
}()
```

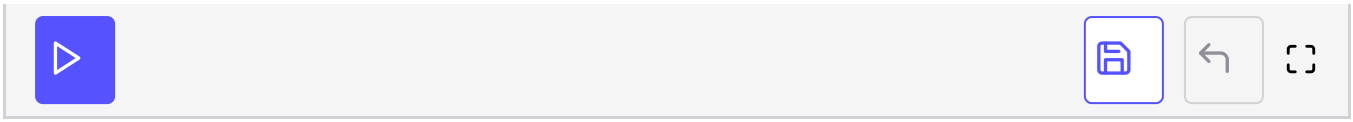
The first **( )** is the parameter-list, and it follows immediately after the keyword `func` because there is no function name. The **{ }** comprise the function body, and the last pair of **( )** represent the *call to the function*.

Here is an example of assigning a function literal to a variable:

```
package main
import "fmt"

func main() {
    f()
}

func f() {
    for i := 0; i < 4; i++{
        g := func(i int) {fmt.Printf("%d ", i)} // g assigned to the function literal
        g(i) // g used as a function
        fmt.Printf(" - g is of type %T and has value %v\n", g, g)
    }
}
```



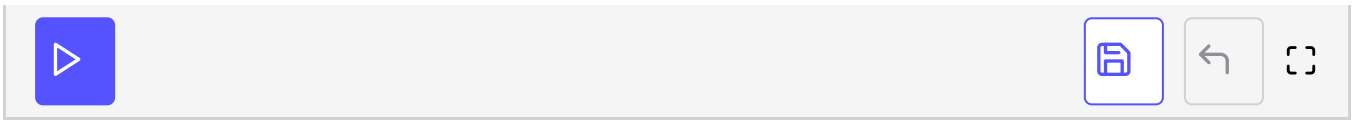
Function Literal Assignment

In the above code, in `main` at **line 5**, we call a function `f`. At **line 8**, we declare `f`. There is a for loop iterating **4** times. In that loop, we declare an anonymous function taking the `i` iterator as a parameter and printing the value. The function is assigned to variable `g`. In the next line (**line 11**) we called `g` as the function literal we declared at **line 10**. This `g` will now perform the task expected by the closure to perform at **line 10**. This makes it clear that the type of `g` is `func(int)`, and its value is the address where it's placed in memory.

You can verify the type of such a function too. Look at the following program.

```
package main
import (
    "fmt"
```

```
func main() {
    fv := func() {
        fmt.Println("Hello World!")
    }
    fv()
    fmt.Printf("The type of fv is %T", fv)
}
```



Lambda Value

In the above code, in `main()`, **line** 7 defines a lambda function and assigns it to a variable `fv`, which prints **Hello World!**. **Line 10** calls the function with `fv`, causing **Hello World!** to be printed on screen. In the next line (**line 11**), type of `fv` (`func()`) using *%T* format specifier is printed.

Anonymous functions, like all functions, can be with or without parameters. In the following example, there is a value `v` passed as a parameter to `u`:

```
func (u string) {
fmt.Println(u)
...
}(v)
```

The `defer` keyword and lambda functions

The `defer` keyword is often used with lambda functions. It can then also change return values, provided that you are using named result parameters. Lambda functions can also be launched as goroutines with *go* (see Chapter 12).

Lambda functions are also called closures (a term from the theory of functional languages). They may refer to variables defined in a surrounding function. A closure is a function that captures some external state, for example, the state of the function in which it is created. Another way to formulate this is by using a closure that inherits the scope of the function in which it is declared. That state (those variables) is then shared between the surrounding function and the closure, and the variables survive as long as they are accessible.

Closures are often used as wrapper functions. They predefine 1 or more of the

arguments for the wrapped function. Another useful application is using

closures in performing clean error-checking.

Now that we know what closures are, let's see how a function can be returned from another function using closures.