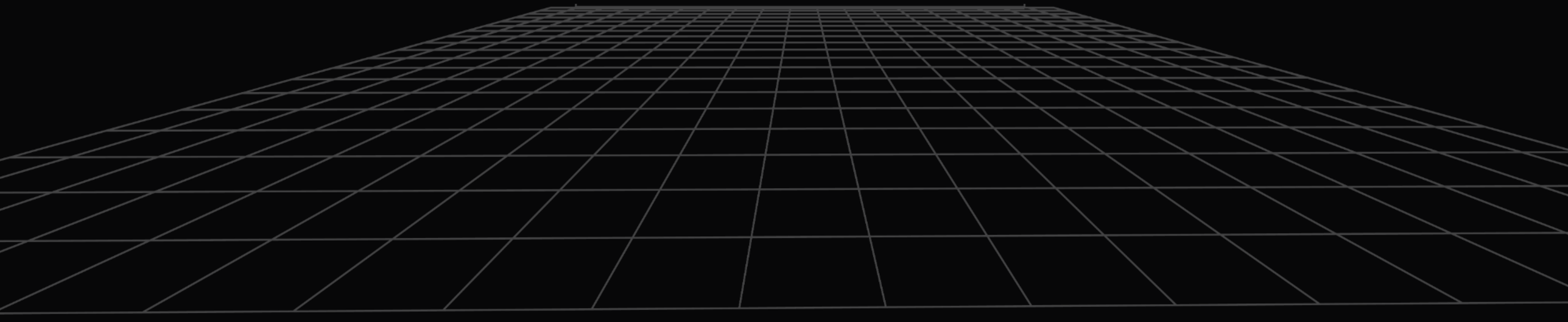
ФУНКЦИИ

Balun.Courses

ПРОВЕРЬ ЗАПИСЬ

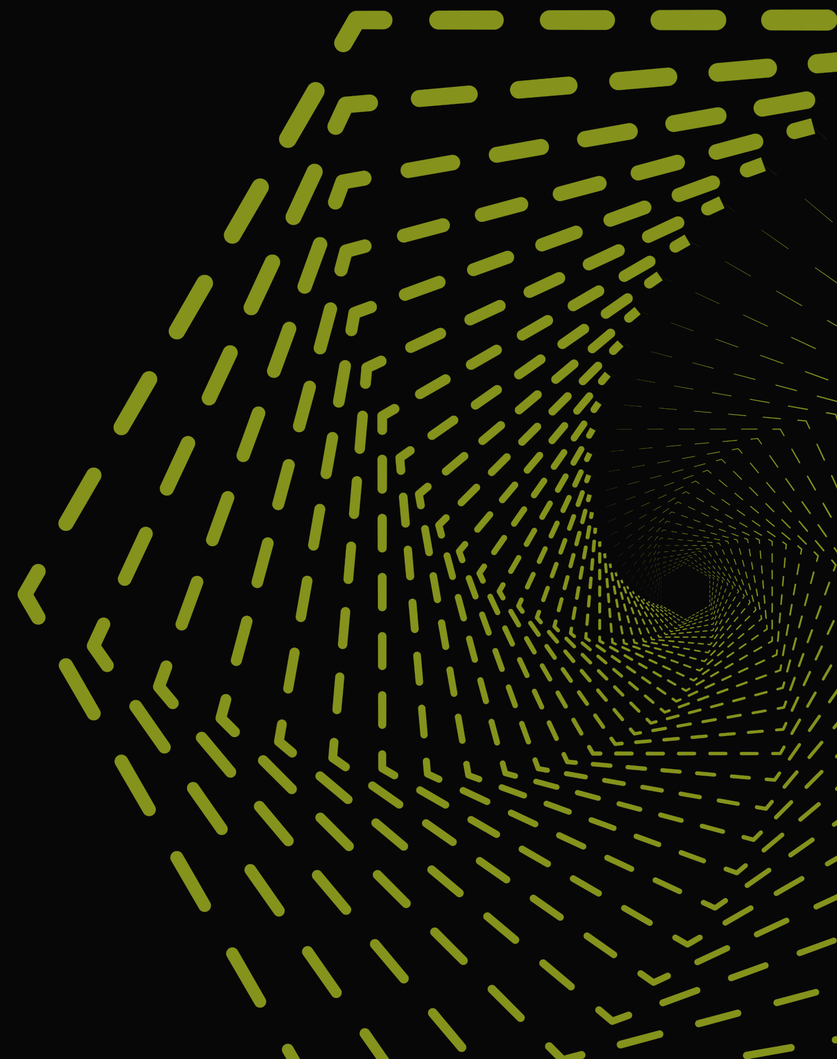


ПРАВИЛА ЗАНЯТИЯ

1. вопросы в чате можно задавать **в любое время**

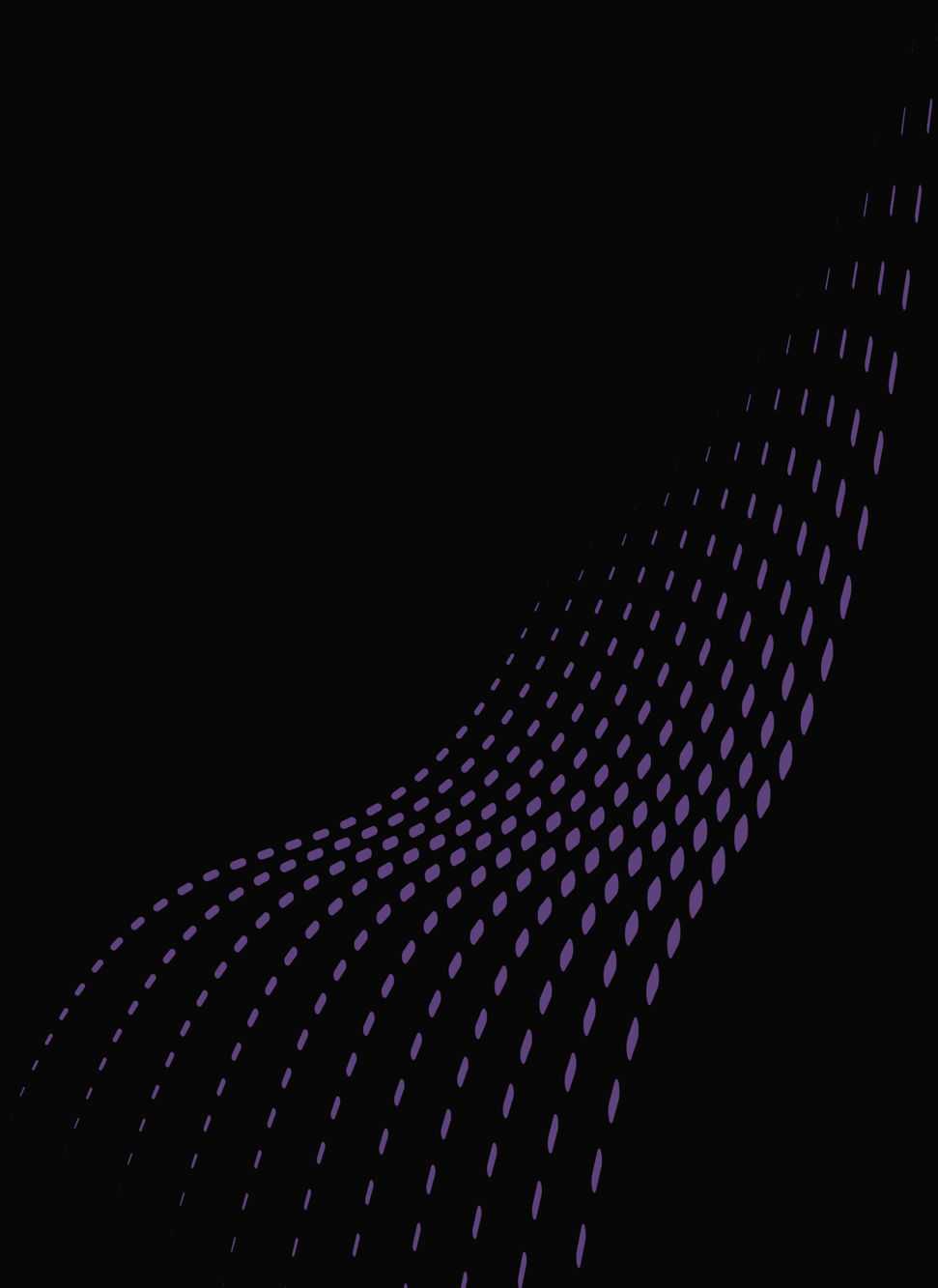
2. вопросы голосом задаем **по поднятой руке в Zoom**

3. ответы на вопросы будут **в запланированных местах**



ПЛАН ЛЕКЦИИ

1. Функции
2. Функциональное программирование
3. Паттерны и приемы
4. Устройство стека



Terminal: deep_golang

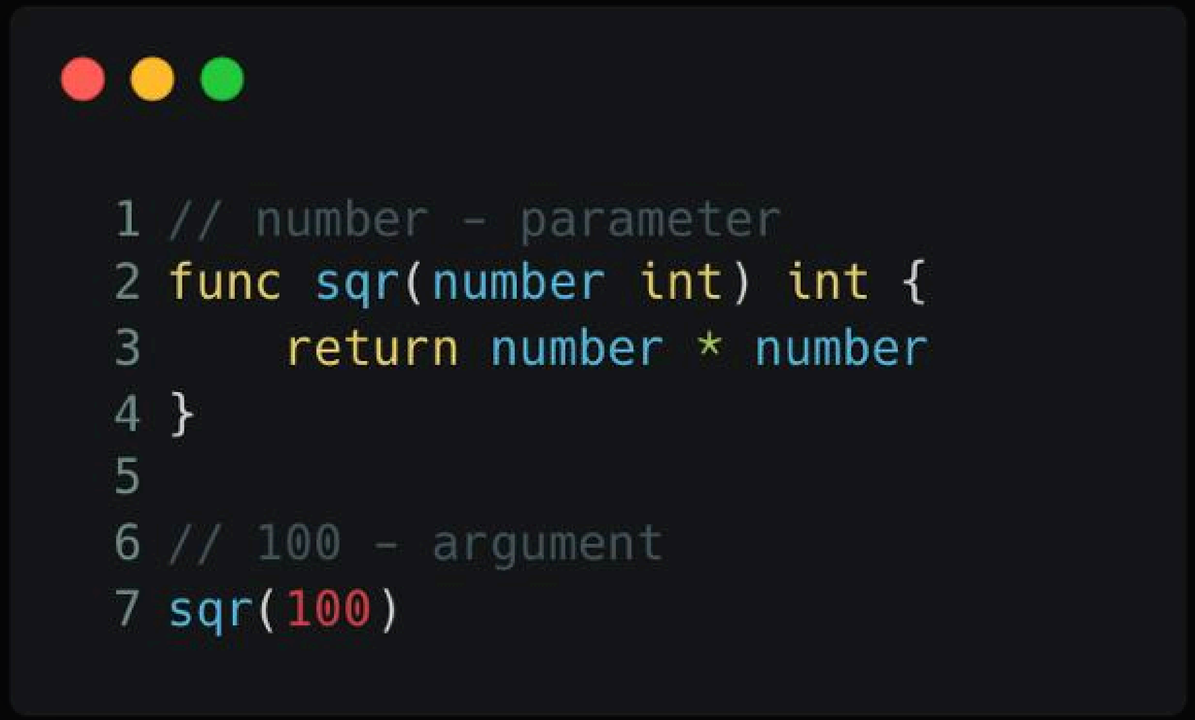
ФУНКЦИИ

Фрагмент кода, к которому можно обратиться из другого места программы
(часто называют подпрограммой)

С именем функции связан адрес первой инструкции функции, которой передается управление при обращении к функции

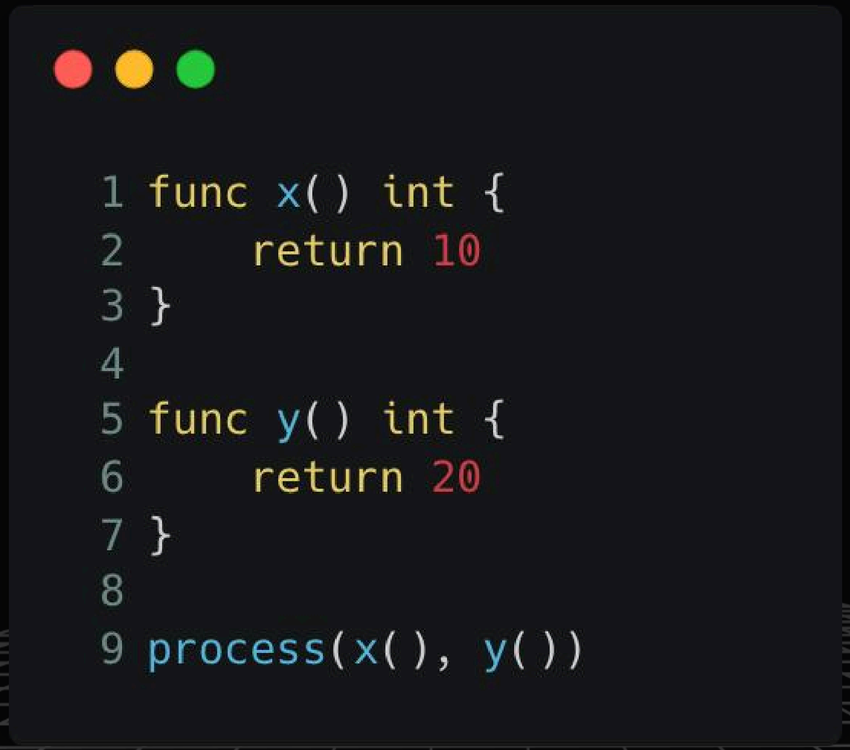
Параметры – переменные в объявлении функции

Аргументы – значения, переданные при вызове



```
1 // number - parameter
2 func sqr(number int) int {
3     return number * number
4 }
5
6 // 100 - argument
7 sqr(100)
```

Аргументы вычисляются **слева направо**



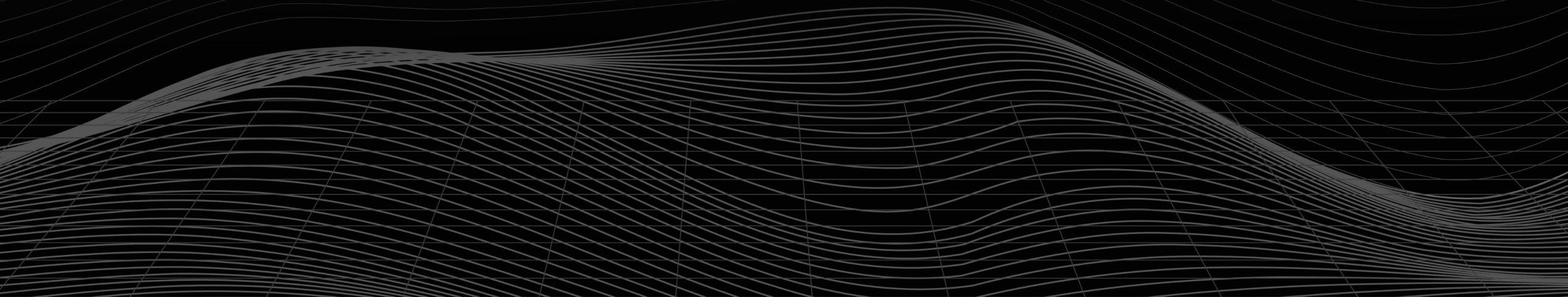
```
1 func x() int {
2     return 10
3 }
4
5 func y() int {
6     return 20
7 }
8
9 process(x(), y())
```

Terminal: deep_golang

СИГНАТУРА ФУНКЦИИ

Список входных параметров функции и результирующих значений функции (название функции, тело функции, а также названия входных параметров и названия результирующих значений не являются сигнатурой)

Все аргументы при передаче
в функцию **копируются!**



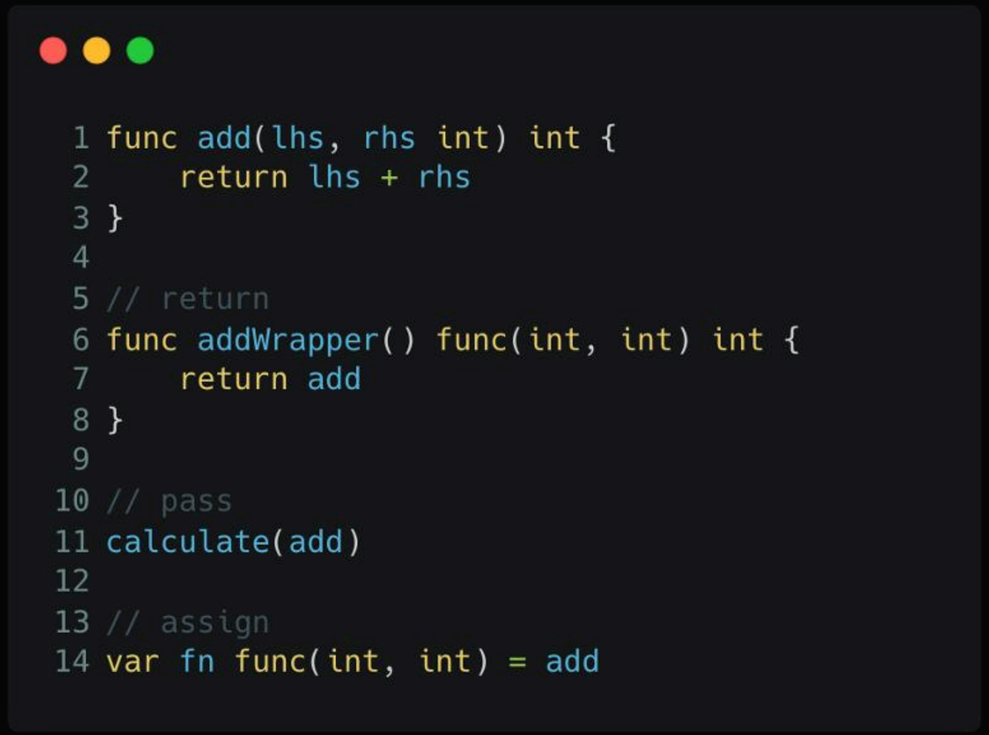
Встроенные функции

```
func append(slice []Type, elems ...Type) []Type
func cap(v Type) int
func clear[T ~[]Type | ~map[Type]Type1](t T)    // 1.21
func close(c chan<- Type)
func complex(r, i FloatType) ComplexType
func copy(dst, src []Type) int
func delete(m map[Type]Type1, key Type)
func imag(c ComplexType) FloatType
func len(v Type) int
func make(t Type, size ...IntegerType) Type
func max[T cmp.Ordered](x T, y ...T) T          // 1.21
func min[T cmp.Ordered](x T, y ...T) T          // 1.21
func new(Type) *Type
func panic(v any)
func print(args ...Type)
func println(args ...Type)
func real(c ComplexType) FloatType
func recover() any
```

В GO ФУНКЦИИ ПЕРВОГО ПОРЯДКА

Они ведут себя как переменные

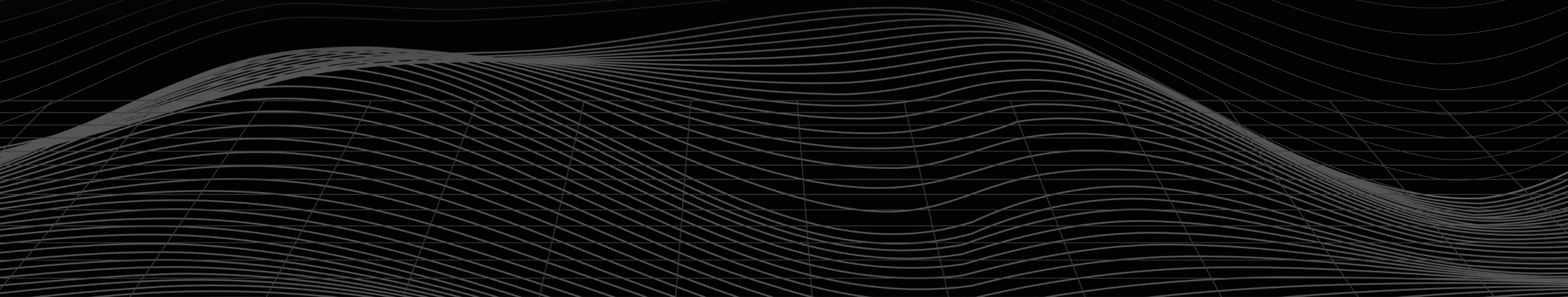
их можно присваивать, передавать и возвращать, но **нельзя сравнивать** и **брать адрес у функции**



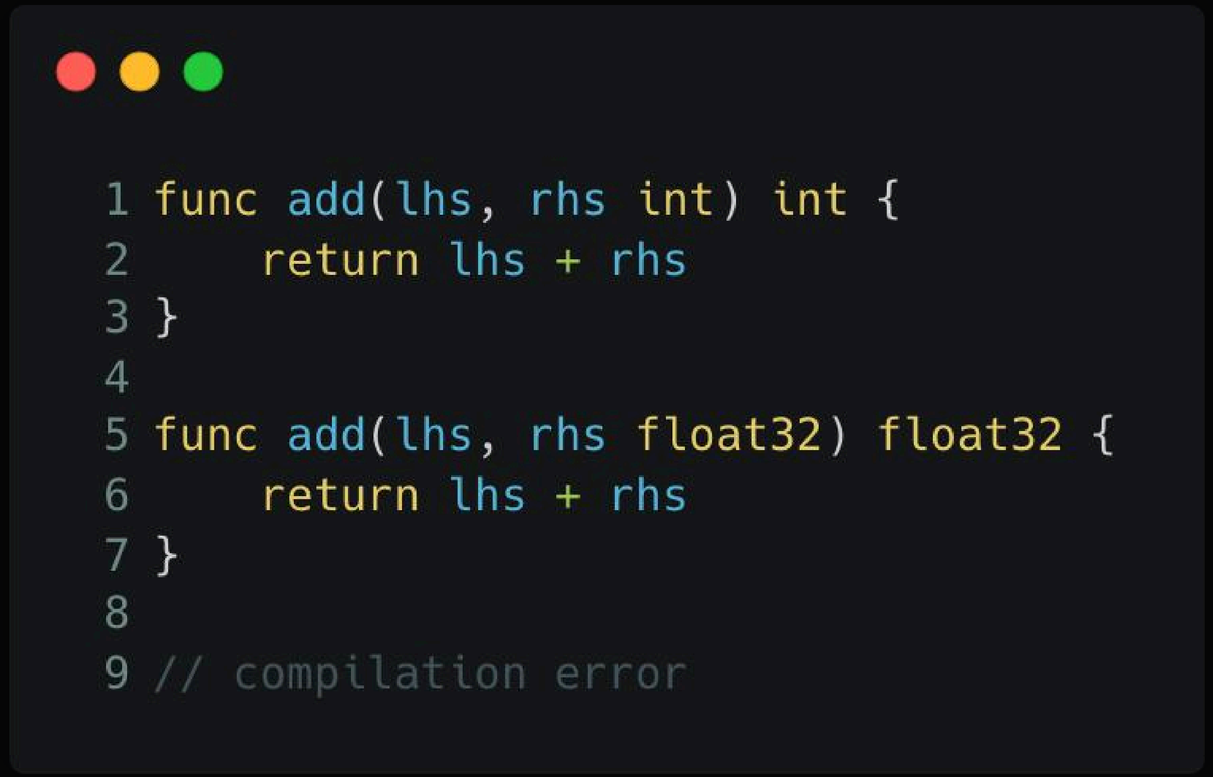
```go
func add(lhs, rhs int) int {
    return lhs + rhs
}

// return
func addWrapper() func(int, int) int {
    return add
}

// pass
calculate(add)

// assign
var fn func(int, int) = add
```

Нулевое значение типов функций равно nil

(также встроенные функции и функцию init нельзя использовать в качестве значений)

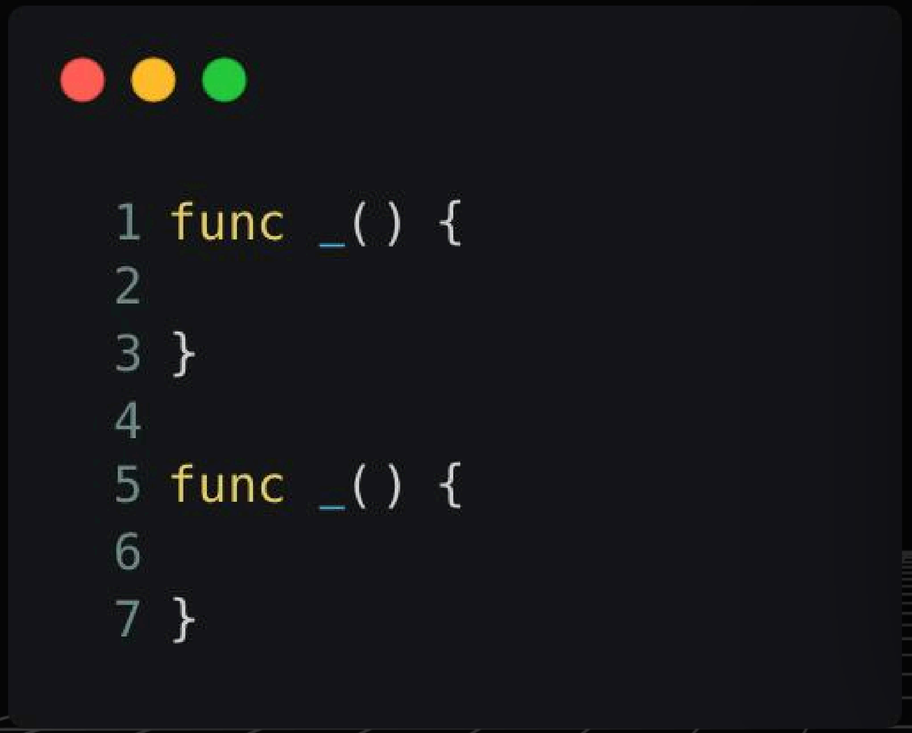


В GO НЕТ ПЕРЕГРУЗКИ ФУНКЦИЙ

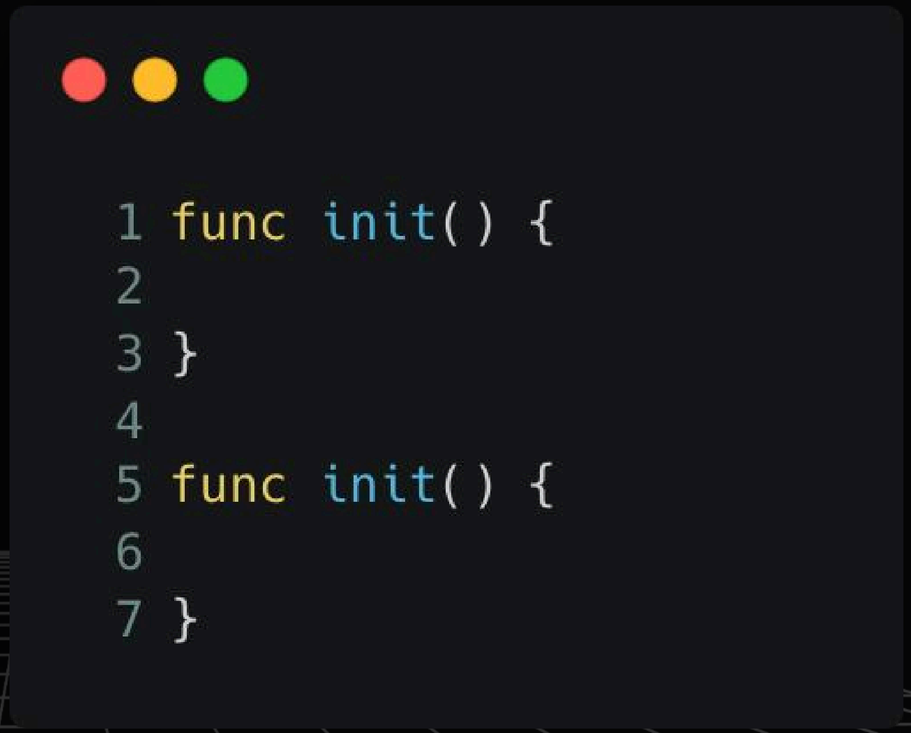


```
1 func add(lhs, rhs int) int {
2     return lhs + rhs
3 }
4
5 func add(lhs, rhs float32) float32 {
6     return lhs + rhs
7 }
8
9 // compilation error
```

Но можно создать несколько функций
с одинаковыми именами **(init или _)**

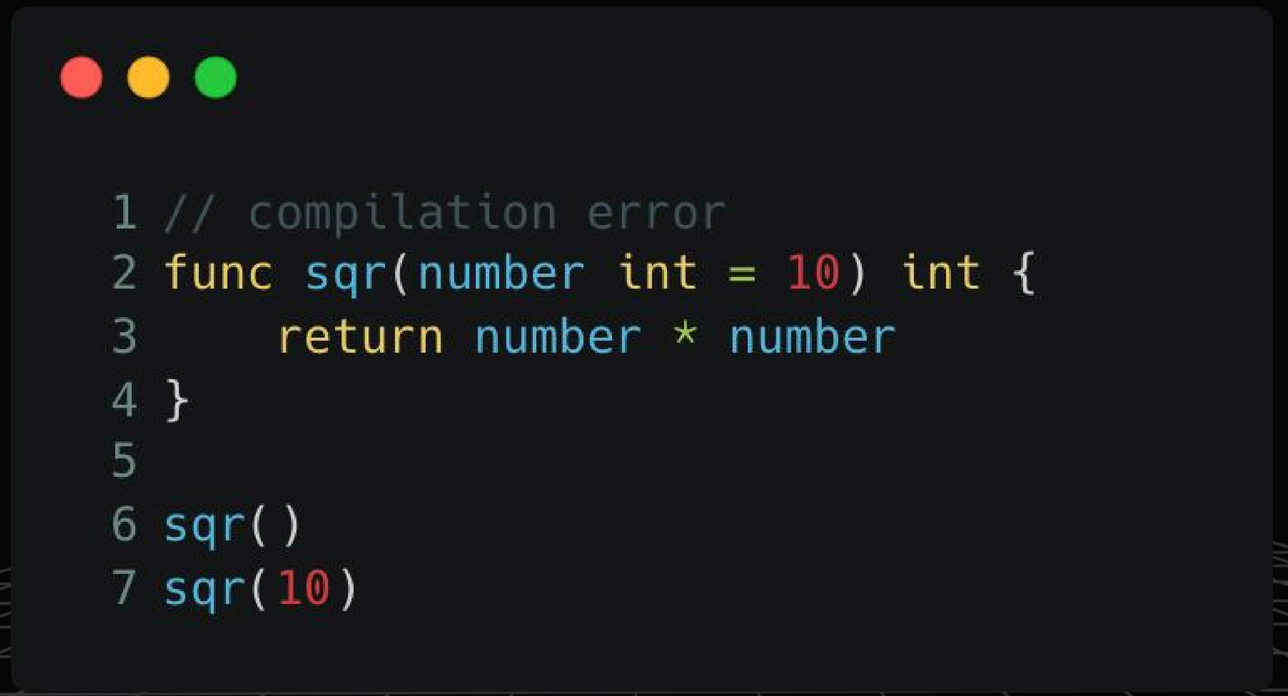


```
1 func _() {
2
3 }
4
5 func _() {
6
7 }
```



```
1 func init() {
2
3 }
4
5 func init() {
6
7 }
```

В GO НЕТ ПАРАМЕТРОВ ПО УМОЛЧАНИЮ



```
1 // compilation error
2 func sqr(number int = 10) int {
3     return number * number
4 }
5
6 sqr()
7 sqr(10)
```

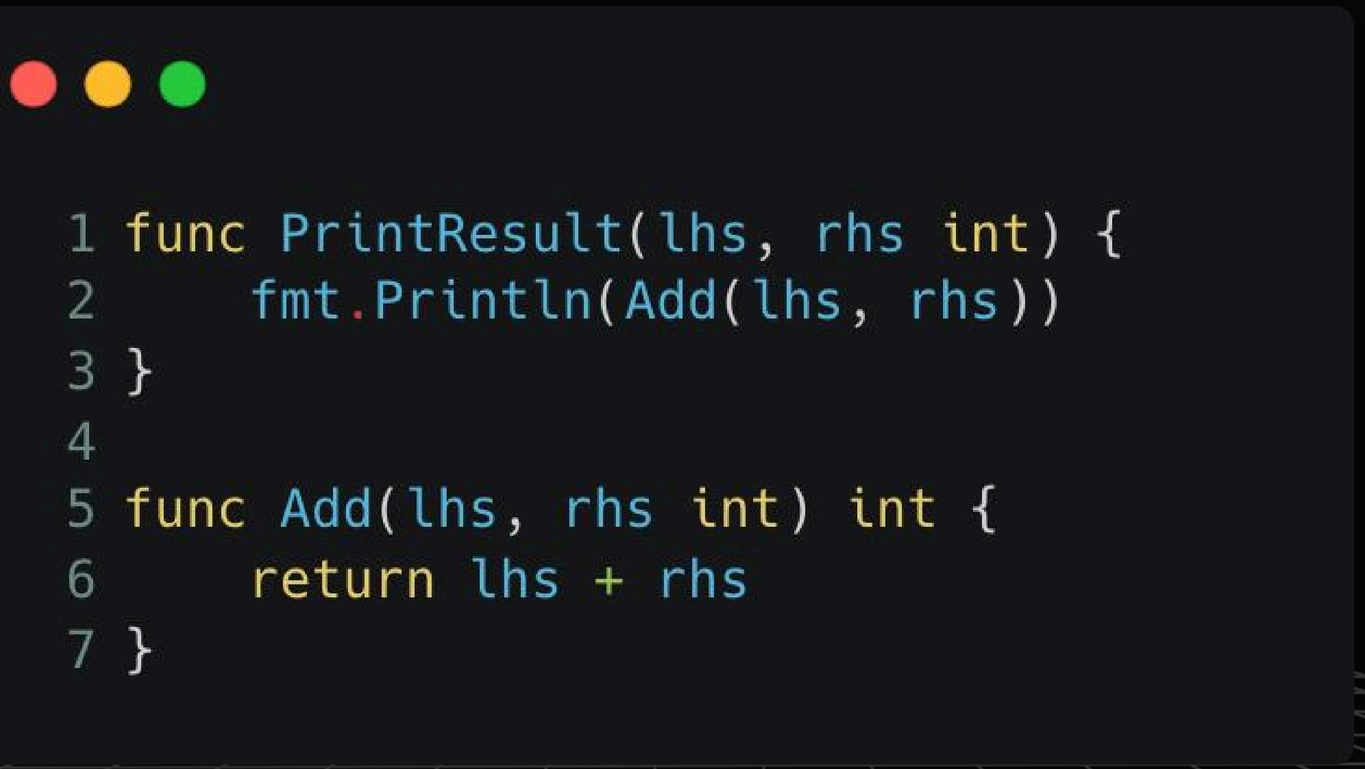
COPY ELISION

В Go нет RVO (Return Value Optimization)
и NRVO (Named Return Value Optimization)

```go
func Get() Point {
    return Point { x: 4, y: 5 }
}

x = Get()
```

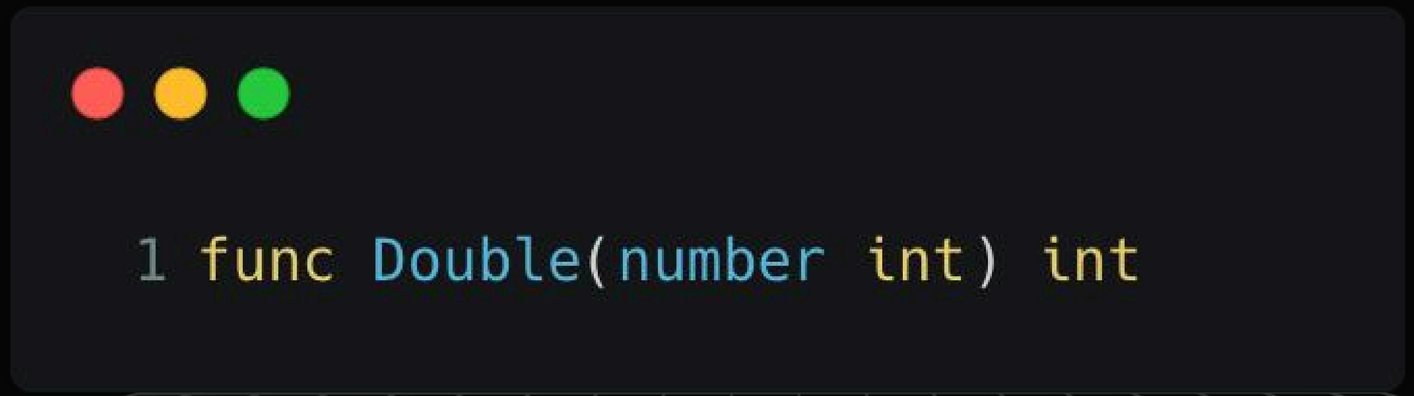
```go
func GetElided(out *Point) {
    out.x = 4
    out.y = 5
}

GetElided(&x)
```

В Go можно использовать функции, которые определены ниже (прототипы не нужны)



```go
func PrintResult(lhs, rhs int) {
    fmt.Println(Add(lhs, rhs))
}

func Add(lhs, rhs int) int {
    return lhs + rhs
}
```

Но тем не менее, в **Go есть прототипы**
(объявление функции без тела функции)



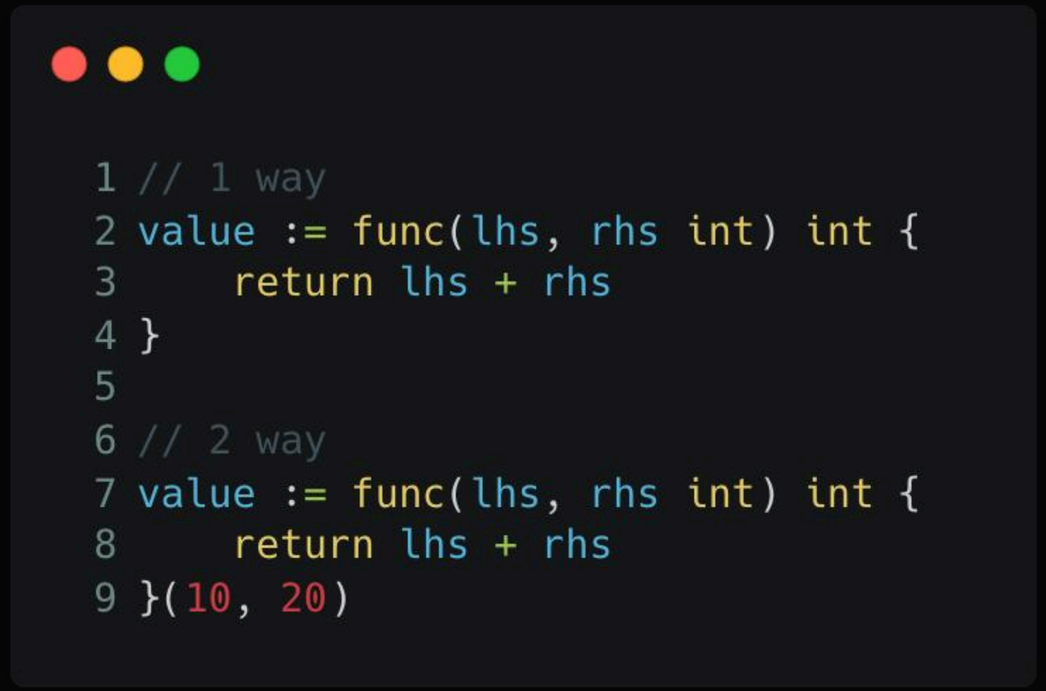
Terminal: question +  –

ЗАЧЕМ НУЖНЫ ПРОТОТИПЫ ФУНКЦИЙ?

В GO ЕСТЬ АНОНИМНЫЕ ФУНКЦИИ

Особый вид функций, которые определяются в месте использования и не получают имени

Анонимные функции можно вызвать сразу после определения



```go
// 1 way
value := func(lhs, rhs int) int {
    return lhs + rhs
}

// 2 way
value := func(lhs, rhs int) int {
    return lhs + rhs
}(10, 20)
```

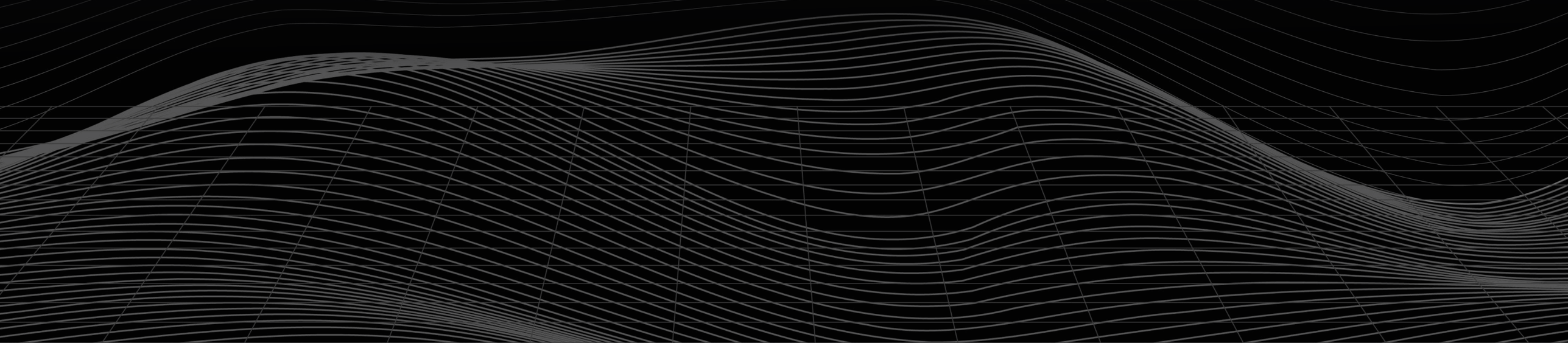
В GO ЕСТЬ VARIADIC PARAMETERS

Могут быть только последними в списке аргументов и их не может быть больше одного

```go
func process(data string, tokens ...int) {
    // implementation...
}

func main() {
    process("")
    process("", 1)
    process("", 1, 2, 3)
    process("", []int{ 1, 2 }...)
}
```

EXAMPLE

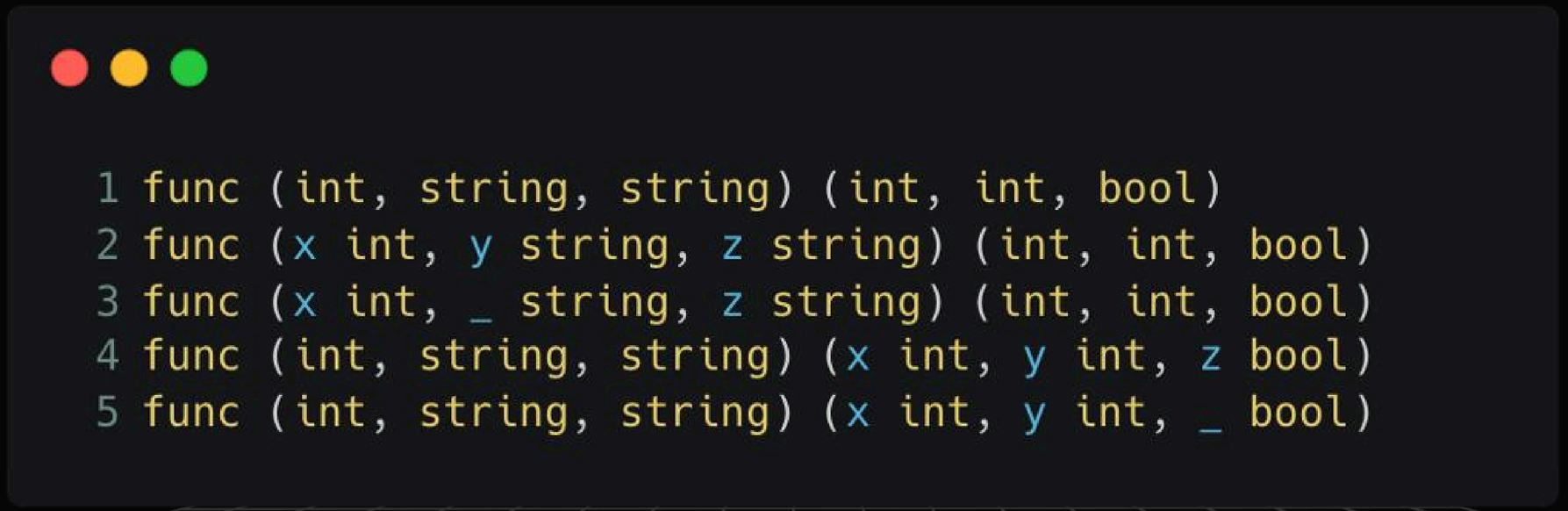
Variadic parameters



```go
func Prepare(
  arg1 string,
  arg2 string,
  arg3 string,
  arg4 string,
  arg5 string,
) {
    // implementation
}
```

```go
func Prepare(arg1, arg2, arg3, arg4, arg5 string) {
    // implementation
}
```

Имена параметров и возвращаемых
значений можно пропускать



```
1 func (int, string, string) (int, int, bool)
2 func (x int, y string, z string) (int, int, bool)
3 func (x int, _ string, z string) (int, int, bool)
4 func (int, string, string) (x int, y int, z bool)
5 func (int, string, string) (x int, y int, _ bool)
```

ИМЕНОВАННЫЕ ПАРАМЕТРЫ ВОЗВРАЩАЕМОГО ЗНАЧЕНИЯ

Terminal: question

КОГДА ИСПОЛЬЗОВАТЬ

именованные параметры возвращаемого значения?

Можно было бы вернуть несколько результатов
в виде структуры, но это **не всегда возможно** –
например, в случае существующего интерфейса

```go
// Use case #1
type navigator interface {
    GetLocation(address string) (lat, lon float32, err error)
}

// Use case #2
type mobileNavigator struct{}

func (n mobileNavigator) GetLocation(address string) (lat, lon float32, err error) {
    return 0., 0., nil
}
```

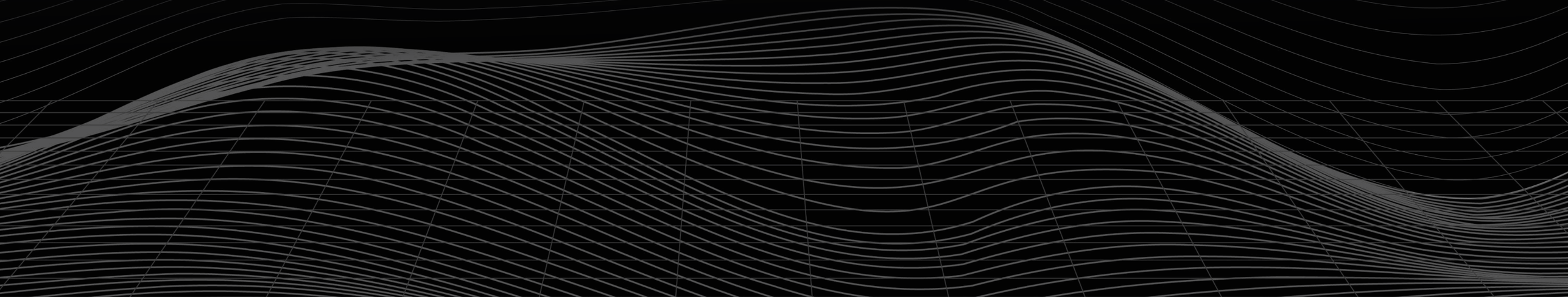
Использование именованных параметров результата зависит от контекста

если нет уверенности, что их использование делает код более читабельным, не используйте их

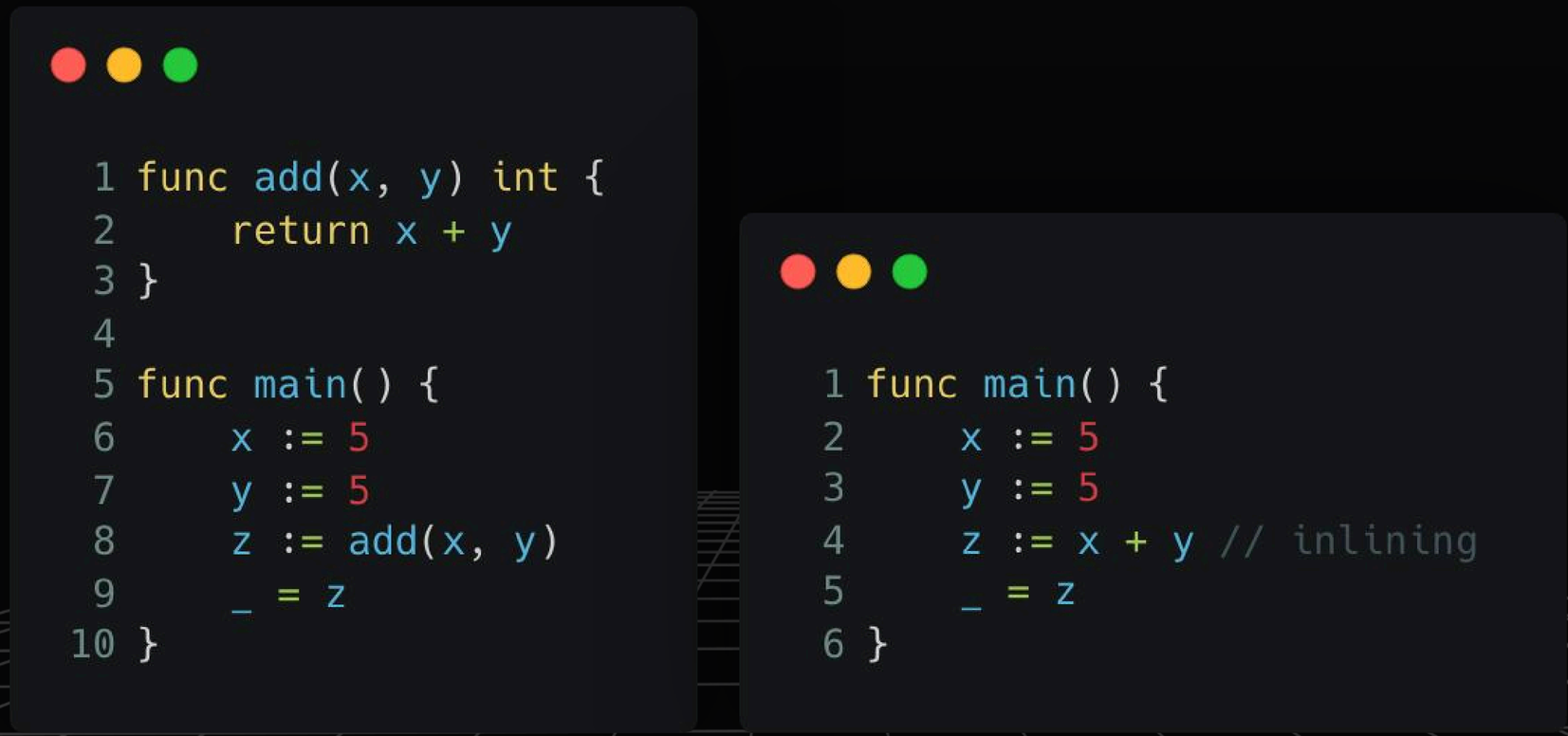
EXAMPLE

Named return values

Не нужно мешать именованный и неименованный **return** в рамках одной функции!



КОМПИЛЯТОР GO УМЕЕТ ВСТРАИВАТЬ ФУНКЦИИ

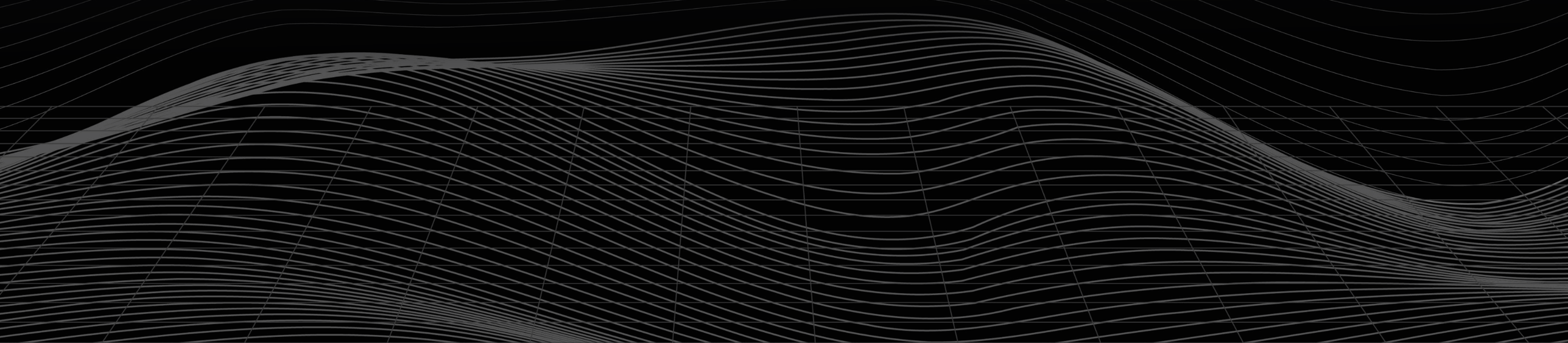


```go
func add(x, y) int {
    return x + y
}

func main() {
    x := 5
    y := 5
    z := add(x, y)
    _ = z
}
```

```go
func main() {
    x := 5
    y := 5
    z := x + y // inlining
    _ = z
}
```

EXAMPLE

Functions inlining



Terminal: question + ⚙ –

КАКИЕ МИНУСЫ ЕСТЬ И ВСТРАИВАНИЯ ФУНКЦИЙ?

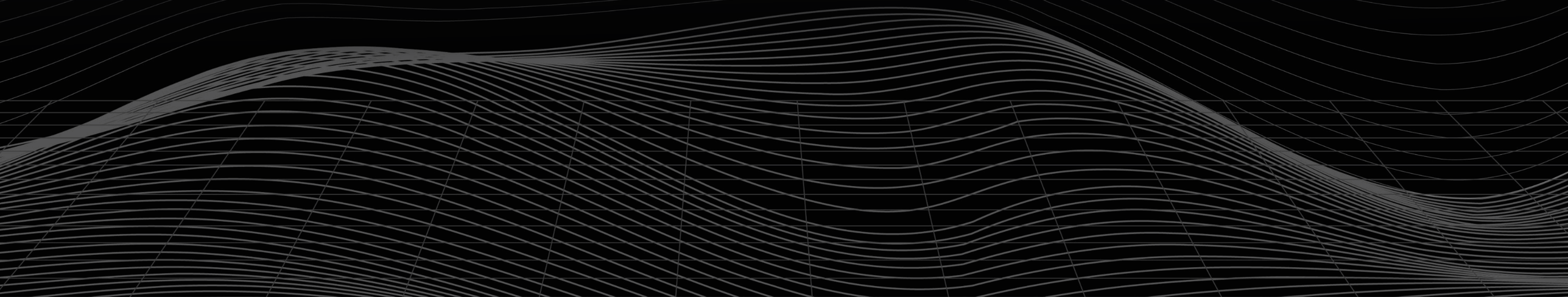
В Go инлайнинг реализуется на этапе компиляции, и его идея максимально простая – основана на **budget model, или cost-based model**

> Суть: у нас есть бюджет, равный 80, на встраивание одной функции. Считаем стоимость функции – если очень грубо, то это сумма всех инструкций, помноженных на какой-то вес. Если стоимость меньше или равна бюджету, мы можем встроить функцию. Если нет, то нет.

Terminal: question

КАКАЯ ОТВЕТСТВЕННОСТЬ ДОЛЖНА БЫТЬ У ФУНКЦИИ?

Функция должна делать что-то одно (если в названии появляется **and**, то явно она делает что-то еще)



ПРАВИЛО №1

ФУНКЦИИ ДОЛЖНЫ БЫТЬ КОМПАКТНЫМИ

ПРАВИЛО №2

ФУНКЦИИ ДОЛЖНЫ БЫТЬ ЕЩЕ КОМПАКТНЕЕ

Я не могу научно обосновать свое утверждение. Не ждите от меня ссылок на исследования, доказывающие, что очень маленькие функции лучше больших. Я могу всего лишь сказать, что я почти четыре десятилетия писал функции всевозможных размеров. Мне доводилось создавать кошмарных монстров в 3000 строк. Я написал бесчисленное множество функций длиной от 100 до 300 строк. И я писал функции от 20 до 30 строк. Мой практический опыт научил меня (ценой многих проб и ошибок), что функции должны быть очень маленькими. Однако **строки не должны состоять из 150 символов**, а функции из 100 строк. Желательно, чтобы **длина функции не превышала 20 строк**

В GO ХОРОШО ПИСАТЬ НЕБОЛЬШИЕ ФУНКЦИИ

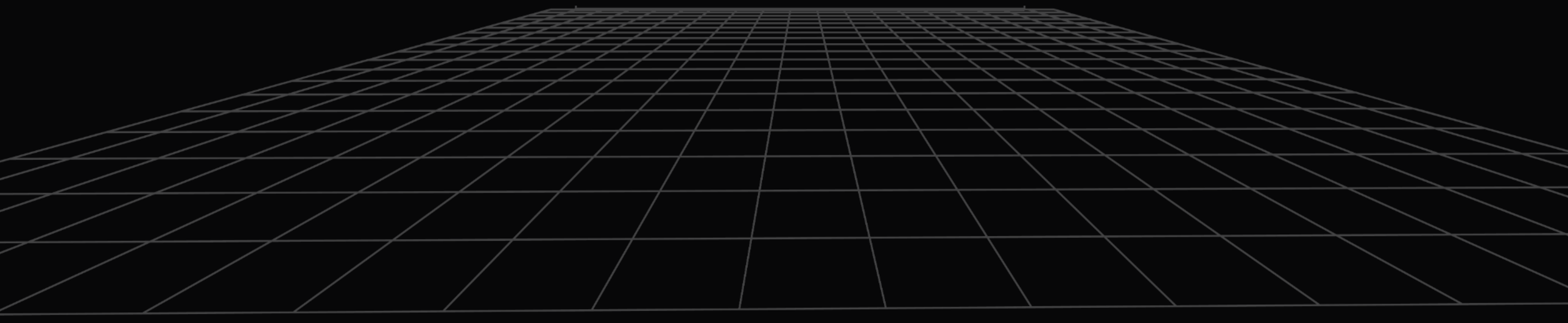
Не только чтобы было проще покрывать их тестами, делать код читаемым и поддерживаемым, но и чтобы Go мог эффективнее производить оптимизации

EXAMPLE

Filename like argument

FAQ

Функции



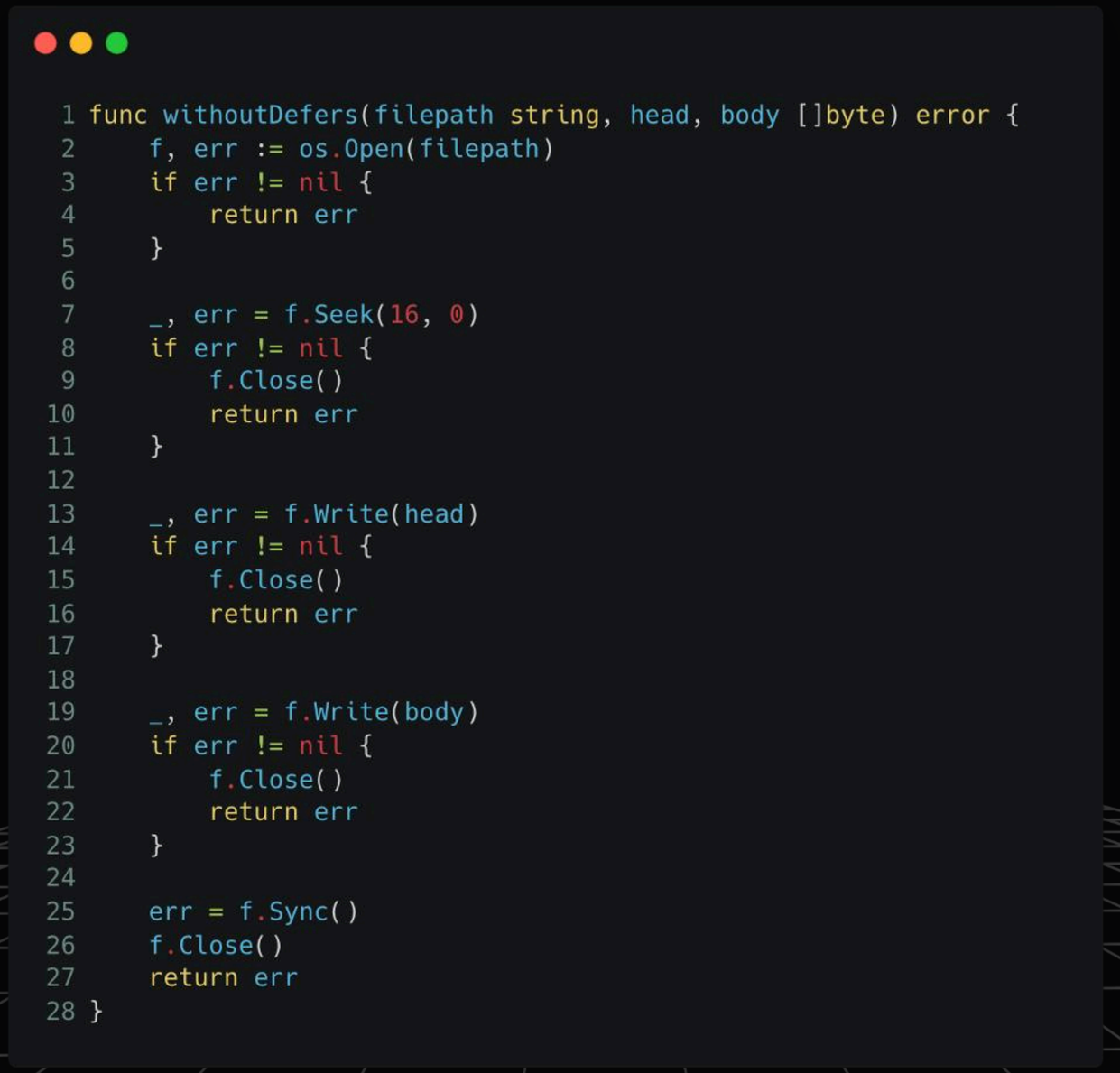
DEFER

Terminal: deep_golang

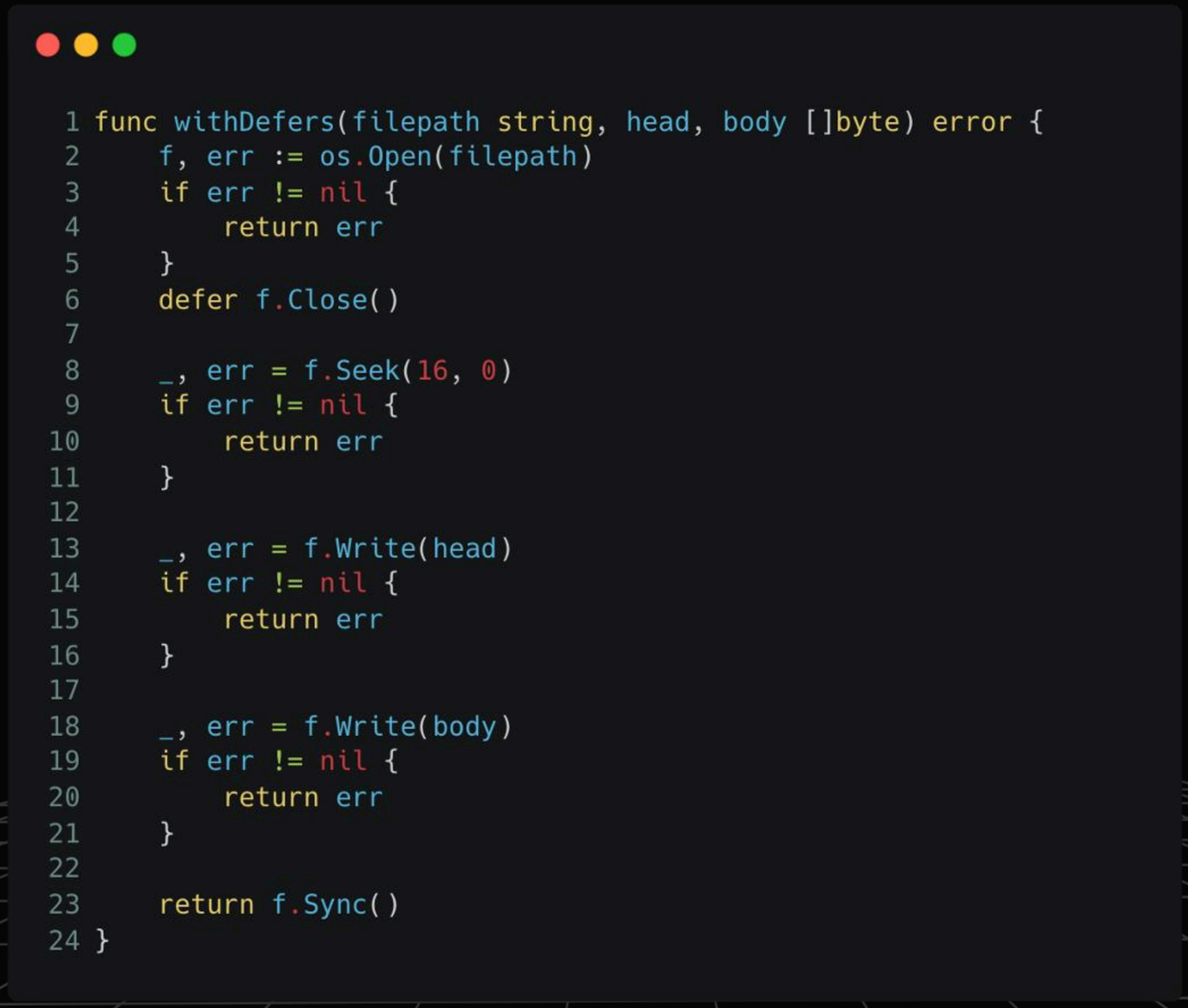
DEFER

Ключевое слово **defer в Go** позволяет нам запланировать запуск функции до ее возврата - несколько функций могут быть отложены из функции

(defer часто используется для очистки ресурсов или завершения задач)

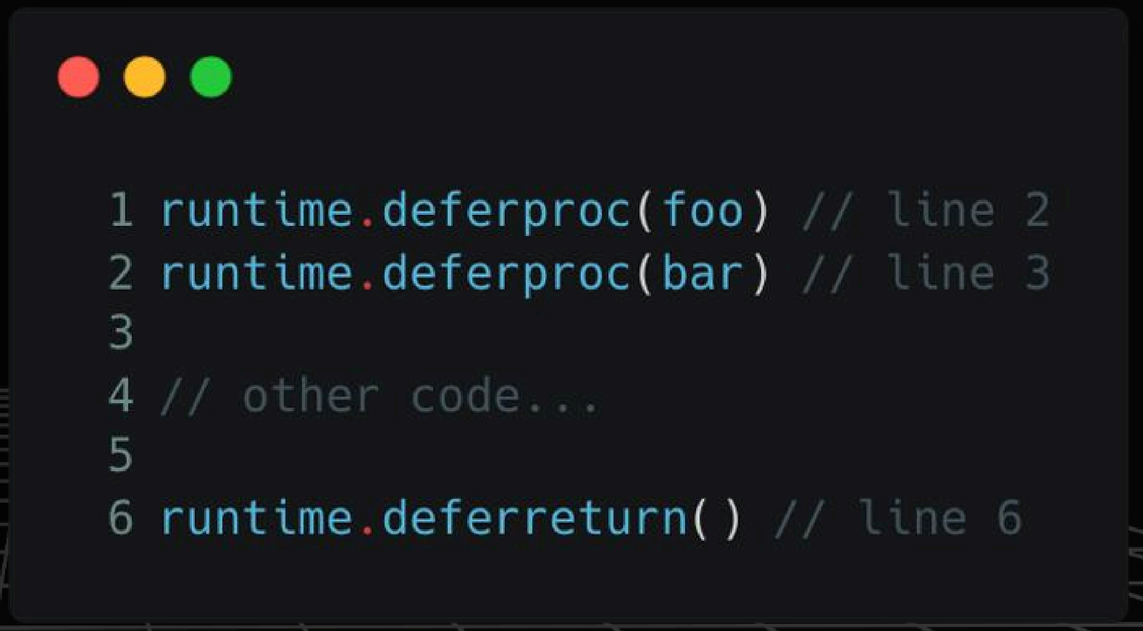


```go
func withoutDefers(filepath string, head, body []byte) error {
    f, err := os.Open(filepath)
    if err != nil {
        return err
    }

    _, err = f.Seek(16, 0)
    if err != nil {
        f.Close()
        return err
    }

    _, err = f.Write(head)
    if err != nil {
        f.Close()
        return err
    }

    _, err = f.Write(body)
    if err != nil {
        f.Close()
        return err
    }

    err = f.Sync()
    f.Close()
    return err
}
```



```go
func withDefers(filepath string, head, body []byte) error {
    f, err := os.Open(filepath)
    if err != nil {
        return err
    }
    defer f.Close()

    _, err = f.Seek(16, 0)
    if err != nil {
        return err
    }

    _, err = f.Write(head)
    if err != nil {
        return err
    }

    _, err = f.Write(body)
    if err != nil {
        return err
    }

    return f.Sync()
}
```

```go
func run() {
    defer foo()
    defer bar()

    fmt.Println("hello")
}
```



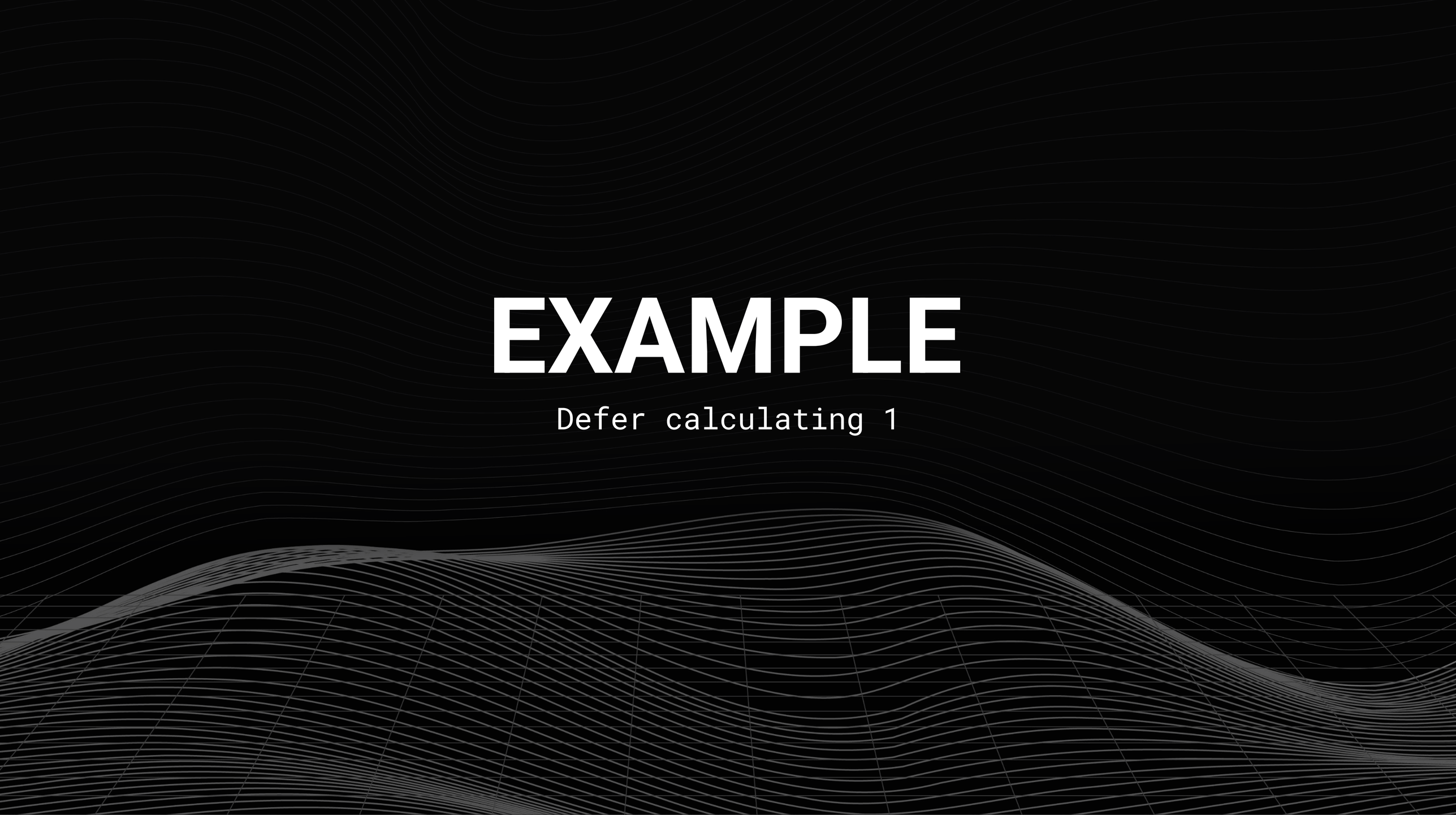
```go
runtime.deferproc(foo) // line 2
runtime.deferproc(bar) // line 3

// other code...

runtime.deferreturn() // line 6
```

EXAMPLE

Defer with function

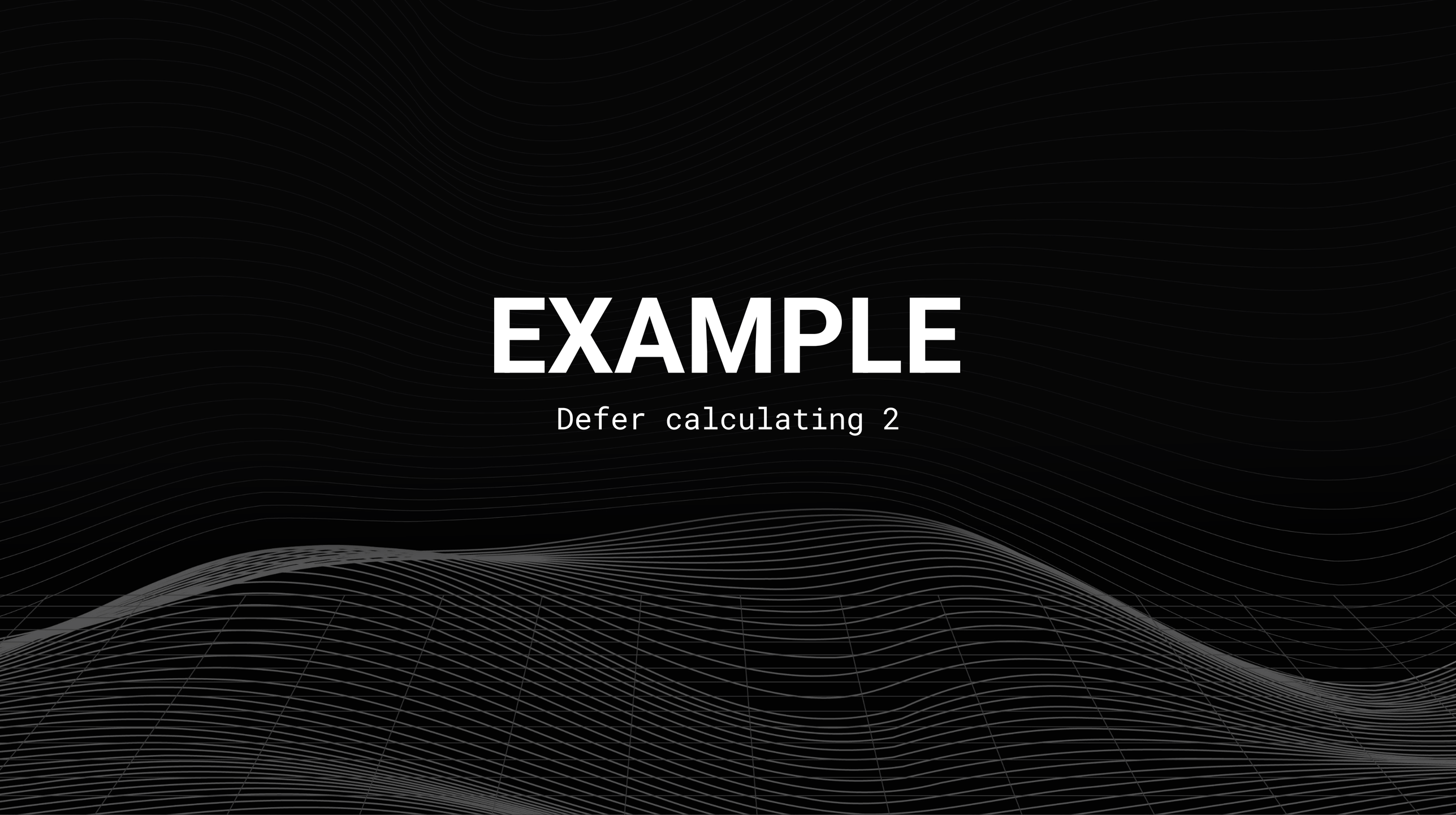
EXAMPLE

Defer calculating 1

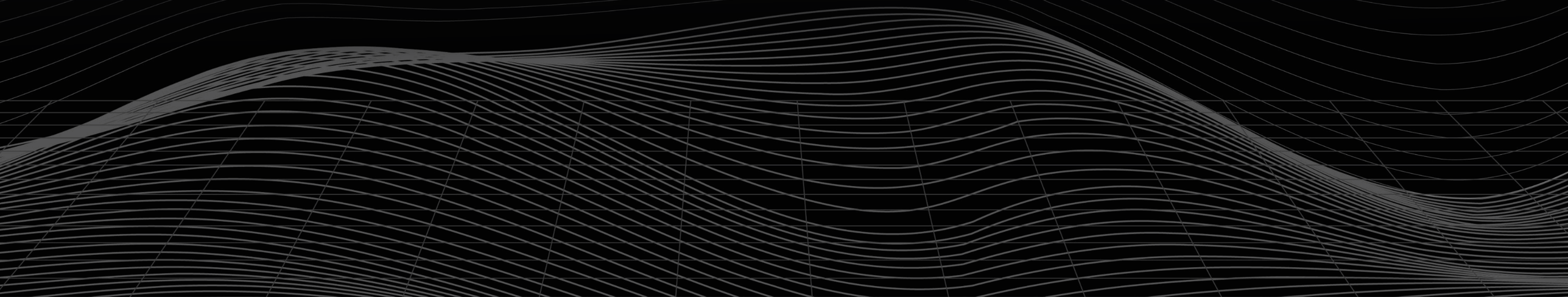


EXAMPLE

Defer calculating 2



Вычисление аргументов функции defer происходит сразу же,
а не после выхода из окружающей функции (следовательно,
Go откладывает выполнение функции с тем значением,
которое было передано при вызове defer)



РЕШЕНИЕ:

1. Использовать указатели
2. Использовать замыкания

EXAMPLE

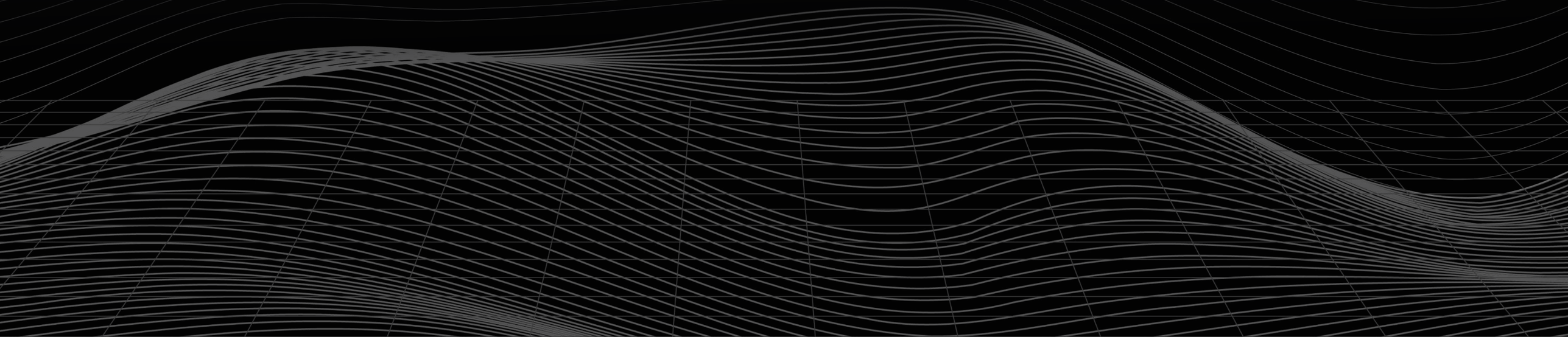
Defer evaluation moment

EXAMPLE

Defer nil

EXAMPLE

Defer inside loop



EXAMPLE

Defer inactive

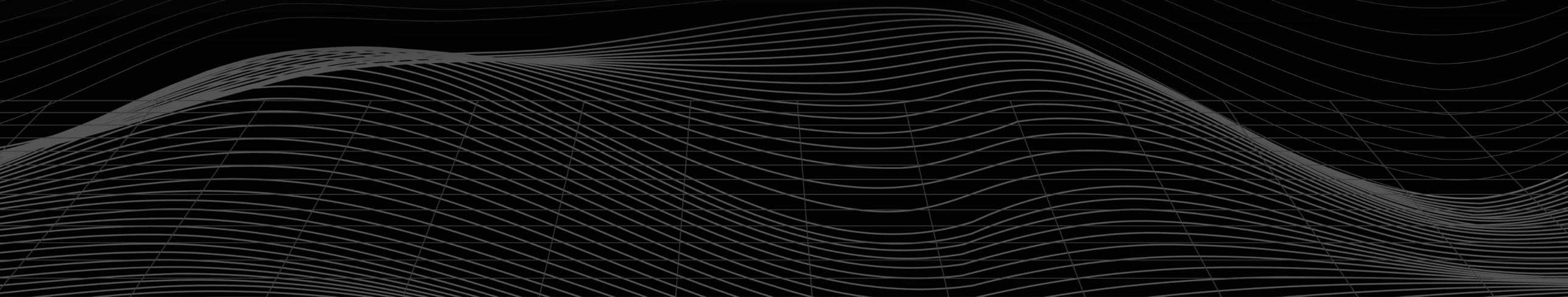
EXAMPLE

Defer union

EXAMPLE

Defer order

Defer может модифицировать именованные параметры возвращаемого значения



EXAMPLE

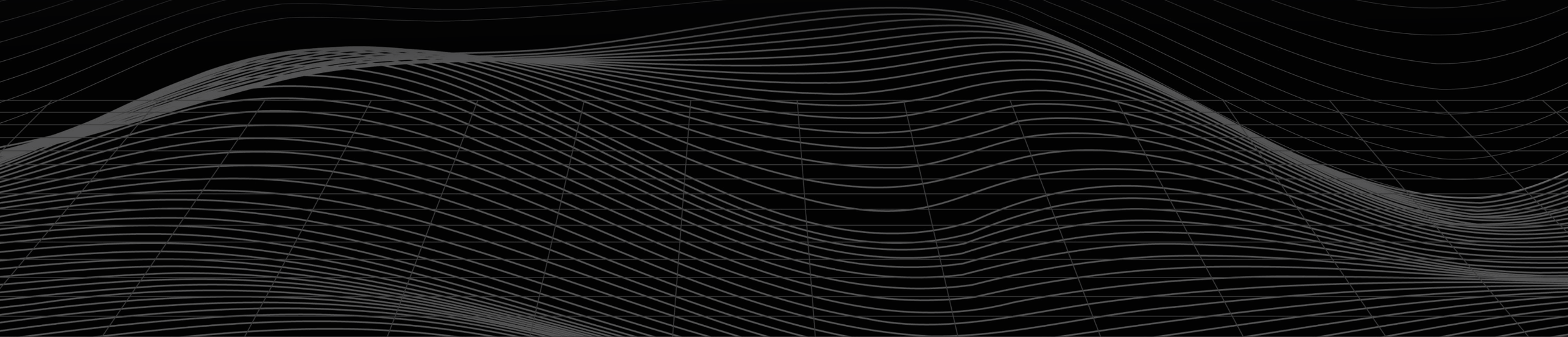
Defer modification 1

EXAMPLE

Defer modification 2

EXAMPLE

Defer performance



GO 1.14 – DEFER INLINING

```go
func run() {
    defer foo()
    defer bar()

    fmt.Println("hello")
}
```

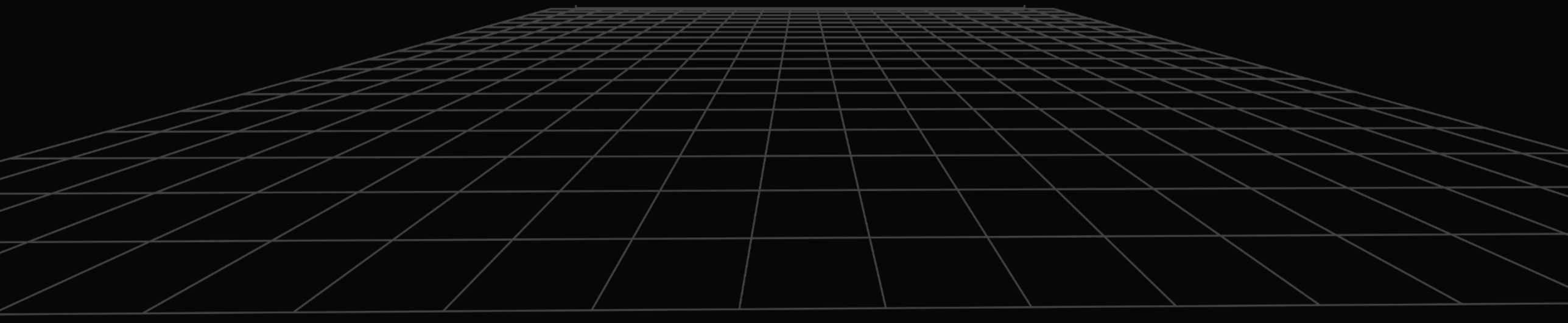
```go
// other code...

bar() // line 6
foo() // line 6
```

It is possible to do this improvement only in static cases. For example, in a loop where the execution is determined by the input size dynamically, the compiler doesn’t have the chance to generate code to inline all the deferred functions.

But in simple cases (e.g. deferring at the top of the function or in conditional blocks if they are not in loops), **it is possible to inline the deferred functions.**

FAQ

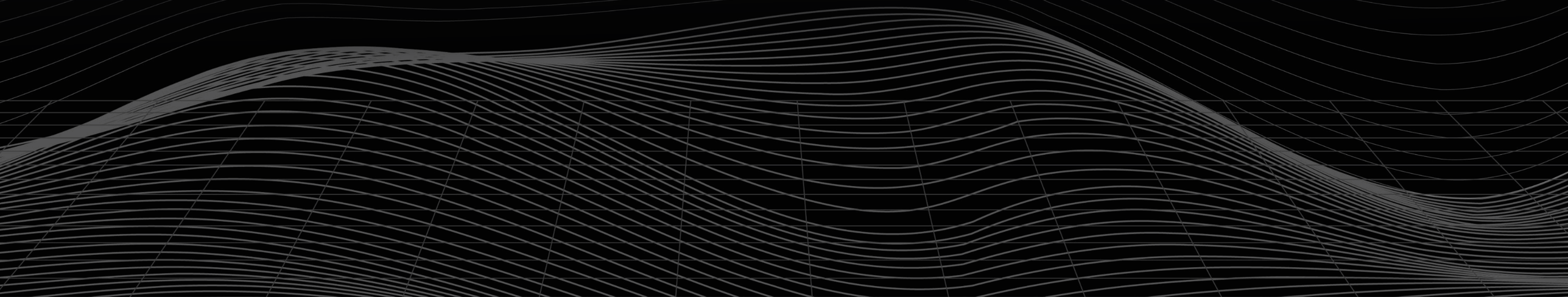
Defer



ПЕРЕРЫВ 5 МИНУТ

ФУНКЦИОНАЛЬНОЕ ПРОГРАММИРОВАНИЕ

Императивный подход описывает, каким образом ты что-то делаешь, а **декларативный** подход описывает, что именно ты делаешь

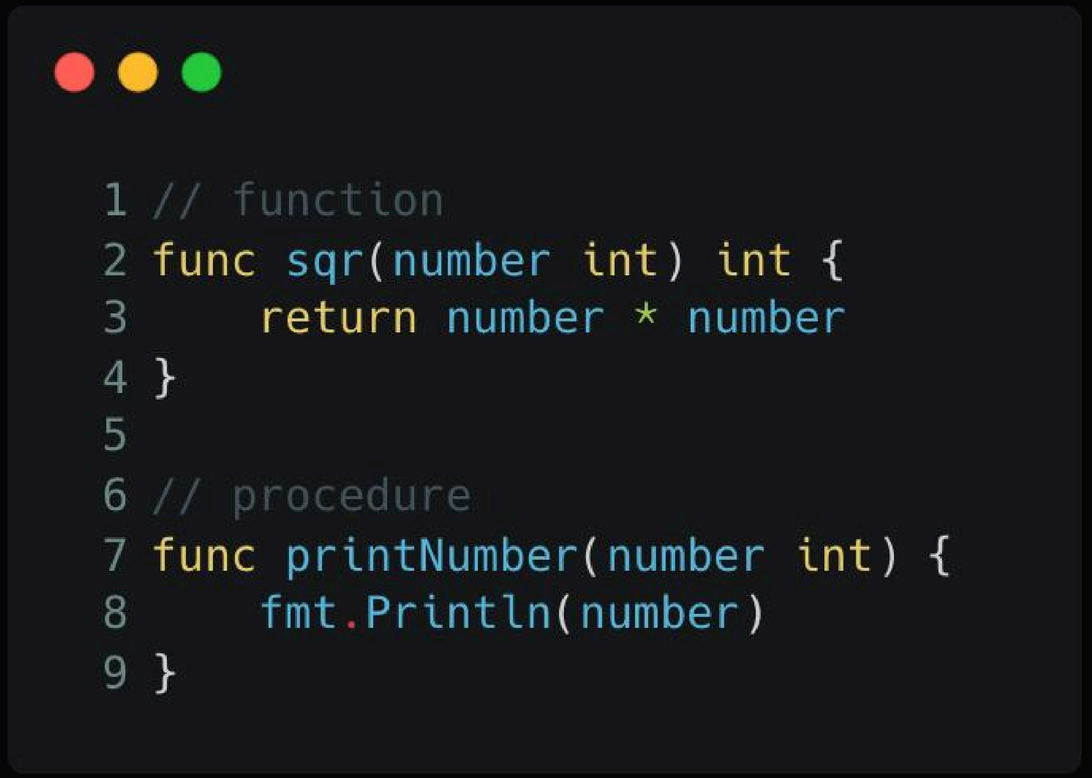


EXAMPLE

Imperative - declarative

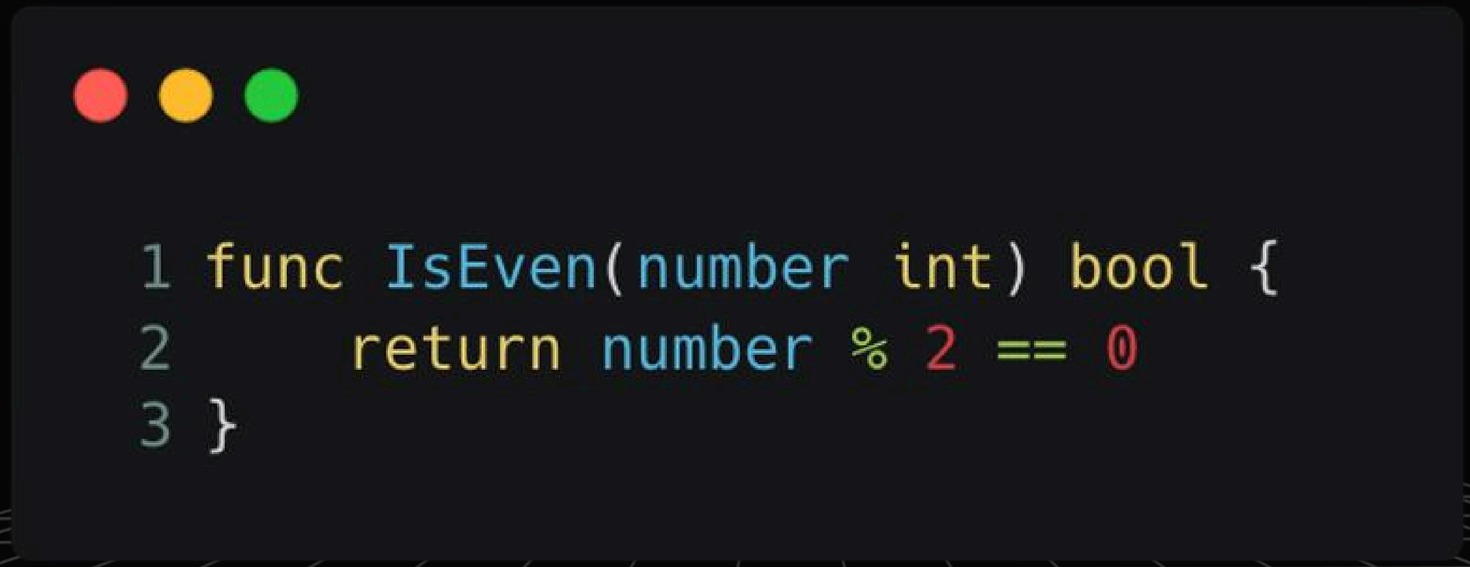
Функция – возвращает значение

Процедура – вызывается ради побочных эффектов



```go
// function
func sqr(number int) int {
    return number * number
}

// procedure
func printNumber(number int) {
    fmt.Println(number)
}
```

Предикат — функция, возвращающая логическое значение



```go
func IsEven(number int) bool {
    return number % 2 == 0
}
```

EXAMPLE

Predicate

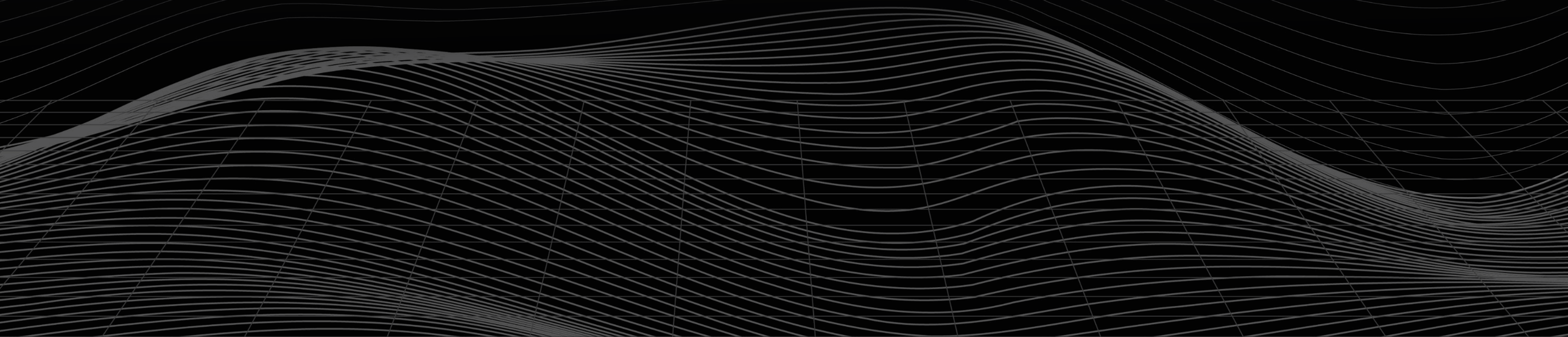
Terminal: deep_golang

ФУНКЦИЯ ВЫСШЕГО ПОРЯДКА

Функция, принимающая в качестве аргументов другие функции или возвращающая другие функции

EXAMPLE

High order function



Terminal: deep_golang

ЧИСТАЯ ФУНКЦИЯ

Функция, которая является детерминированной
и не обладает побочными эффектами

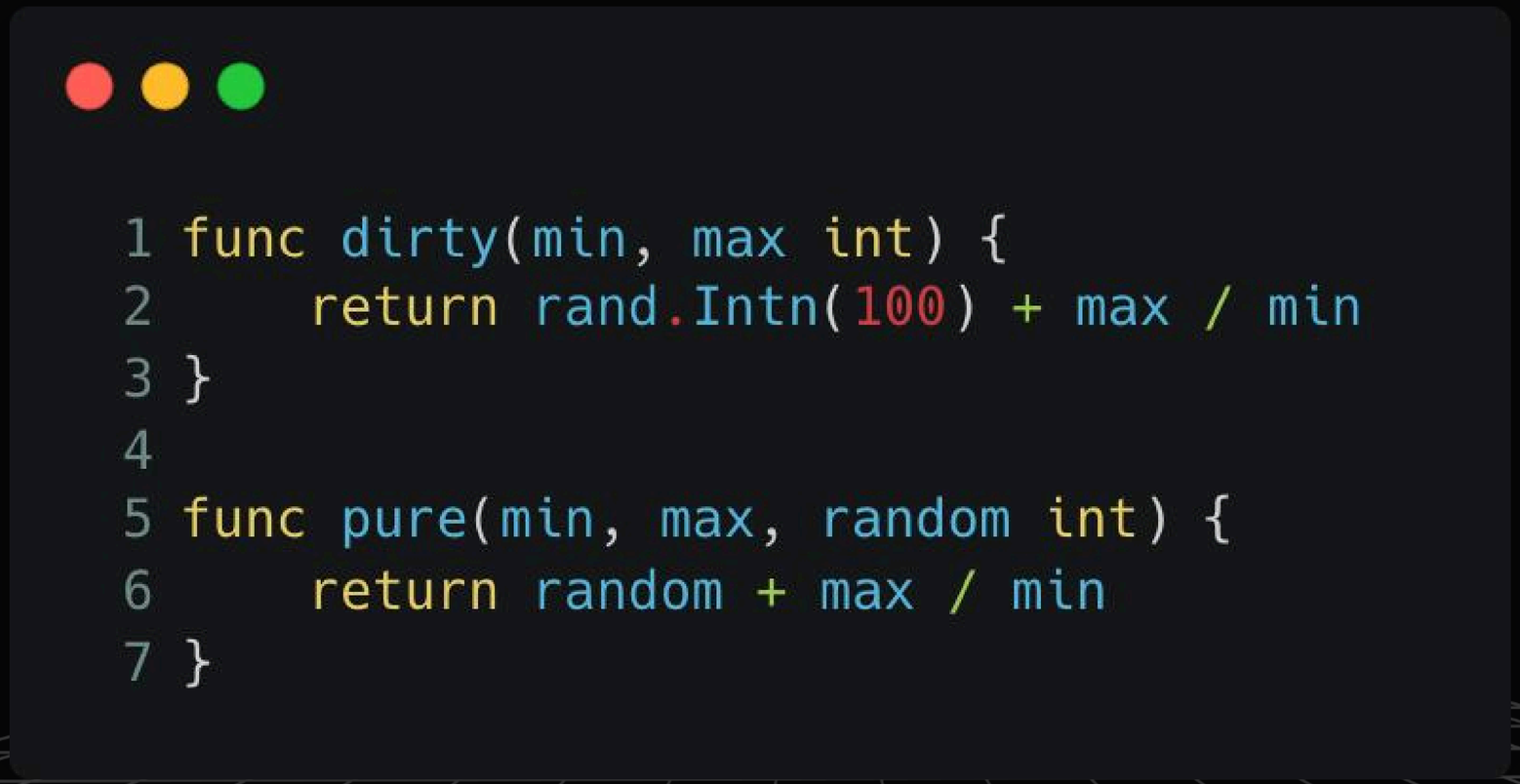
EXAMPLE

Pure function

ПРЕИМУЩЕСТВА

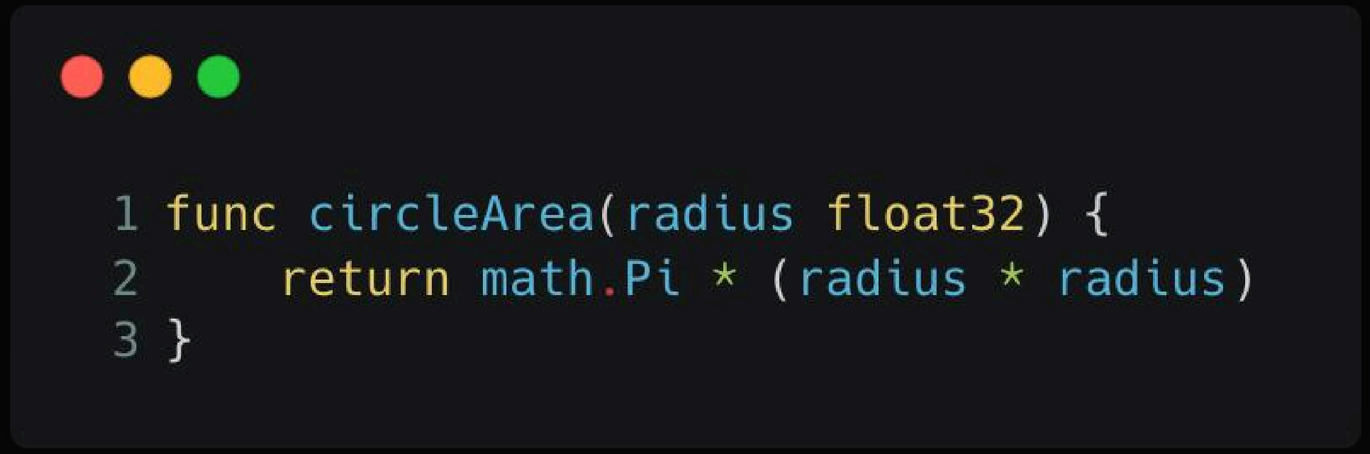
- Проще разобрать, как устроена функция
- Просто распараллелить
- Просто тестировать

Можно выносить «грязь» из функции и принимать,
в виде параметра, либо работать с «грязью» из вне



```go
func dirty(min, max int) {
    return rand.Intn(100) + max / min
}

func pure(min, max, random int) {
    return random + max / min
}
```

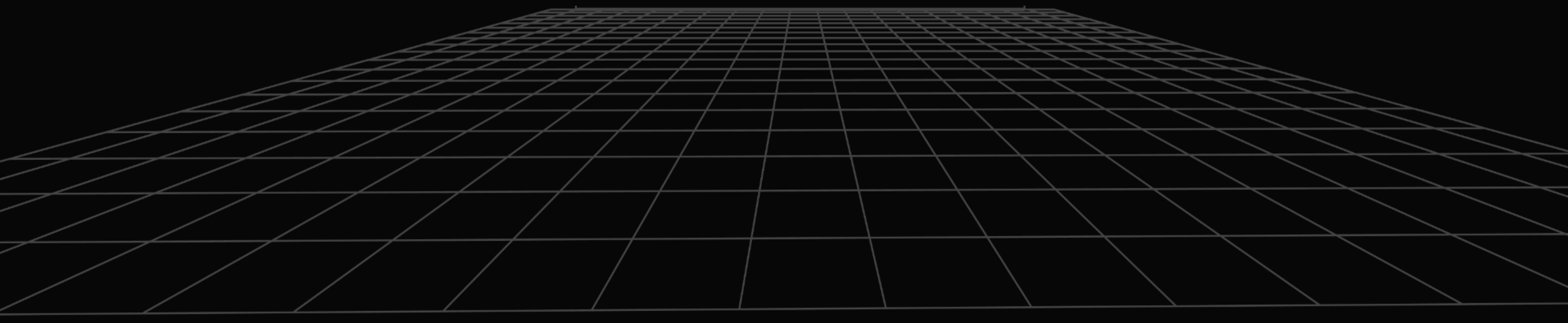
Не нужно вдаваться в крайности и пытаться достичь абсолютной чистоты функции



```
1 func circleArea(radius float32) {
2     return math.Pi * (radius * radius)
3 }
```

FAQ

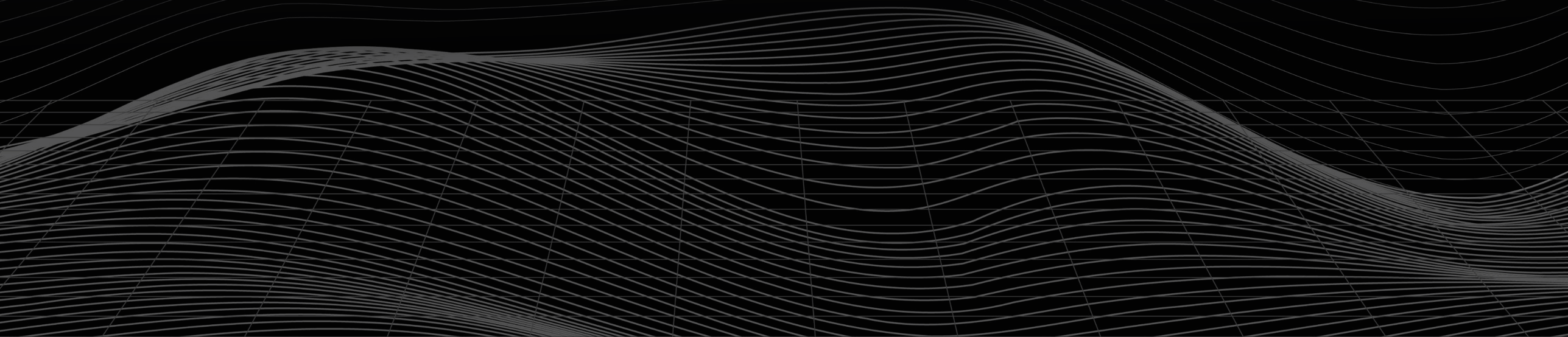
Функциональное программирование



ПАТТЕРНЫ И ПРИЕМЫ

EXAMPLE

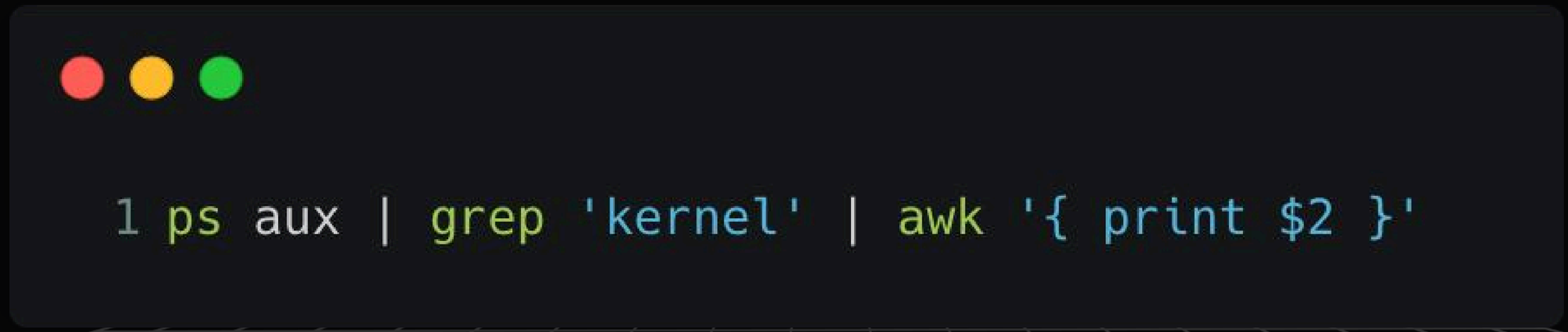
Decorator



EXAMPLE

Composition

LINUX PIPES



EXAMPLE

Conveyor

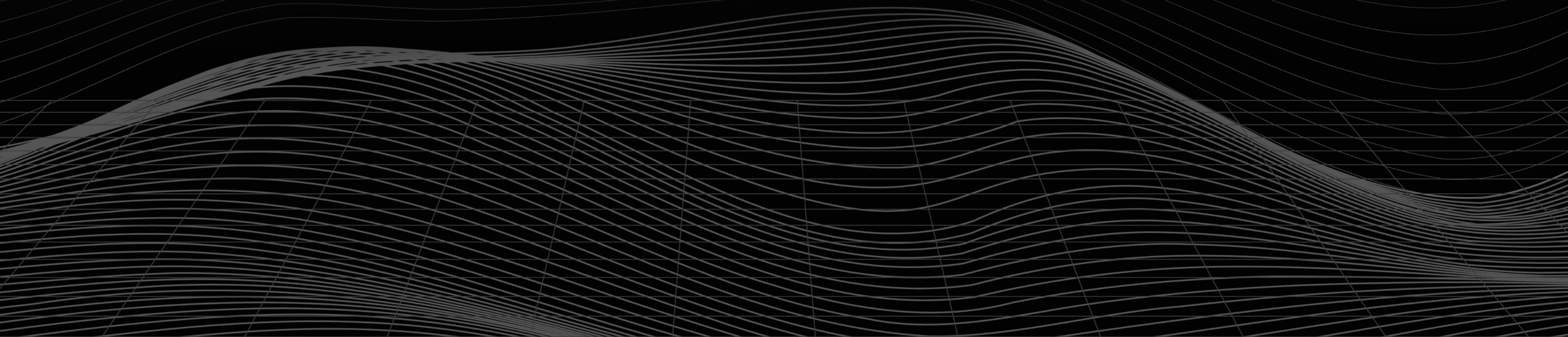
Terminal: deep_golang

ЗАМЫКАНИЕ

Функция, которая ссылается на свободные переменные области видимости своей родительской функции

EXAMPLE

Generator



EXAMPLE

Random generator

Переменные переживающие вызов функции аллоцируются в куче

Terminal: deep_golang

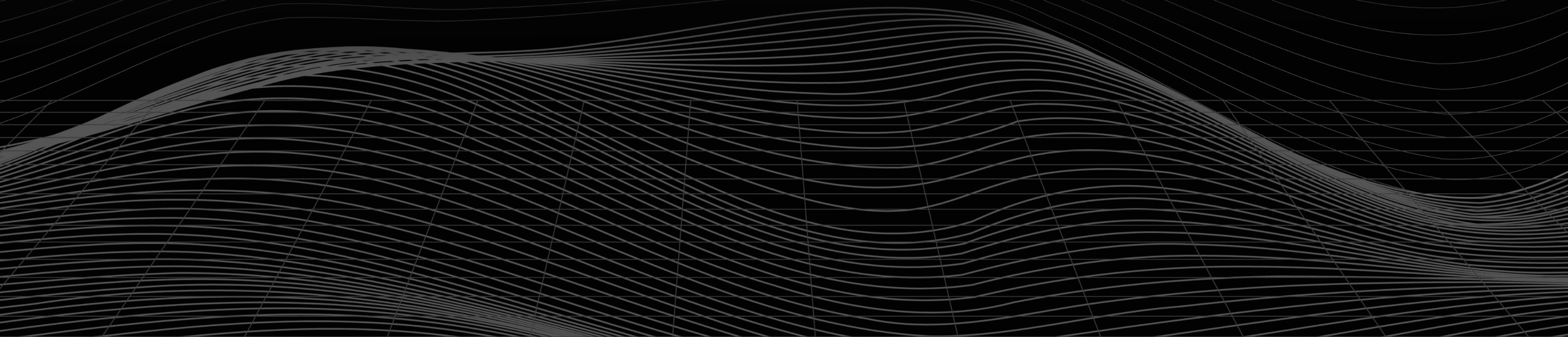
КАРРИРОВАНИЕ

Преобразование функции от многих аргументов
в набор вложенных функций, каждая из которых
является функцией от одного аргумента

изменение функции от вида
func(a,b,c) до вида **func(a)(b)(c)**

EXAMPLE

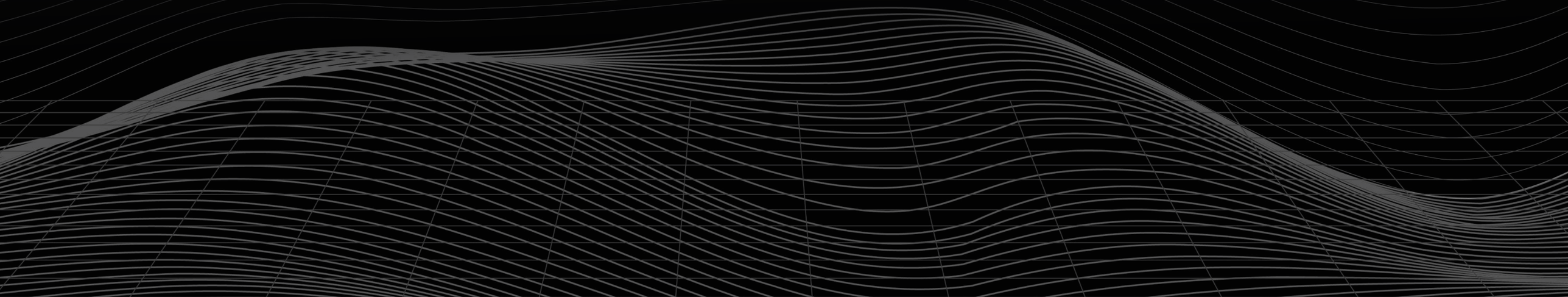
Currying



Terminal: question + ⚙ –

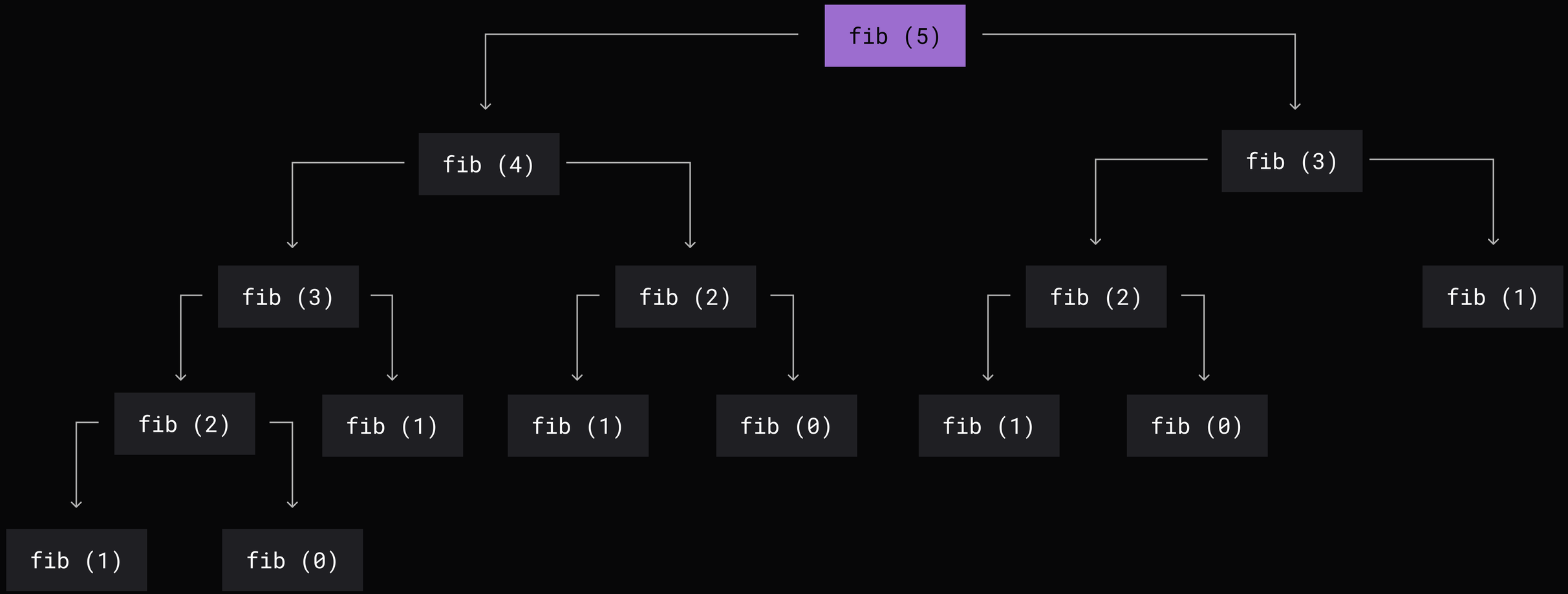
ЗАЧЕМ МОЖЕТ ПОНАДОБИТЬСЯ КАРРИРОВАНИЕ?

Например, у нас есть функция логирования
log(section, level, message)...



EXAMPLE

Lazy evaluation


fib (5)
fib (4)
fib (3)
fib (3)
fib (2)
fib (2)
fib (1)
fib (2)
fib (1)
fib (1)
fib (0)
fib (1)
fib (0)
fib (1)
fib (0)

EXAMPLE

Continuation

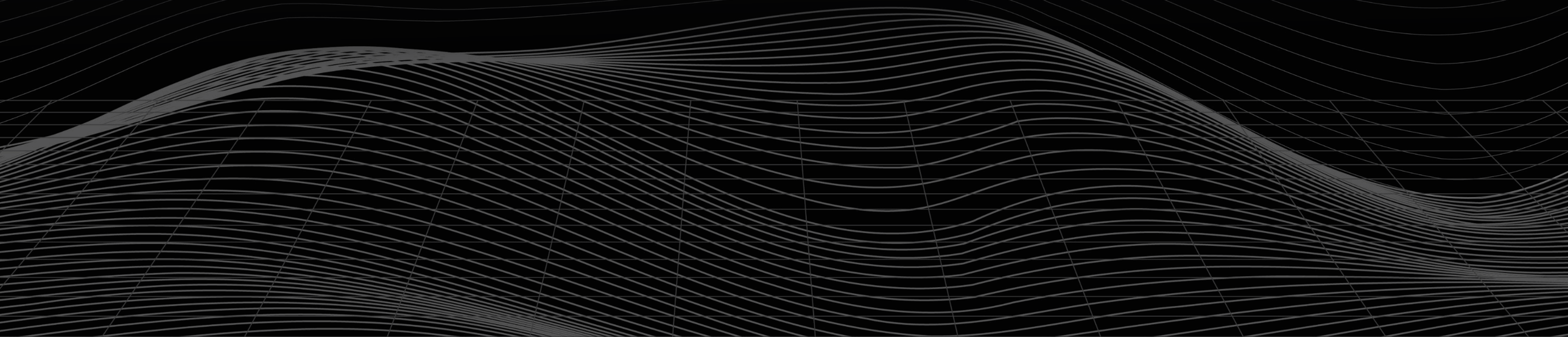
РЕКУРСИЯ

Функция, которая вызывает саму себя, может быть:

- **прямой** – когда функция вызывает себя
- **косвенная** – когда одна функция вызывает внутри себя другую функцию,которая затем вызывает первую

EXAMPLE

Recursion



Terminal: deep_golang

ХВОСТОВАЯ РЕКУРСИЯ

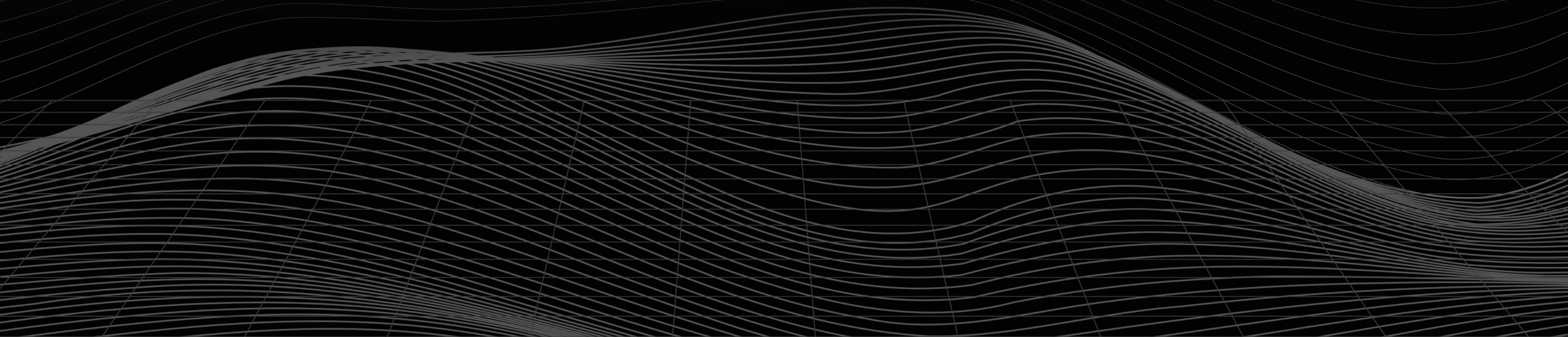
Особый вид рекурсии, когда функция заканчивается вызовом самой себя без дополнительных операторов

Когда это условие выполняется, компилятор разворачивает рекурсию в цикл с одним стек-фреймом, просто меняя локальные переменные от итерации к итерации

«Это не нужно в языке с циклами. Когда программист пишет рекурсивный код, он хочет представлять стек вызовов или он пишет цикл»

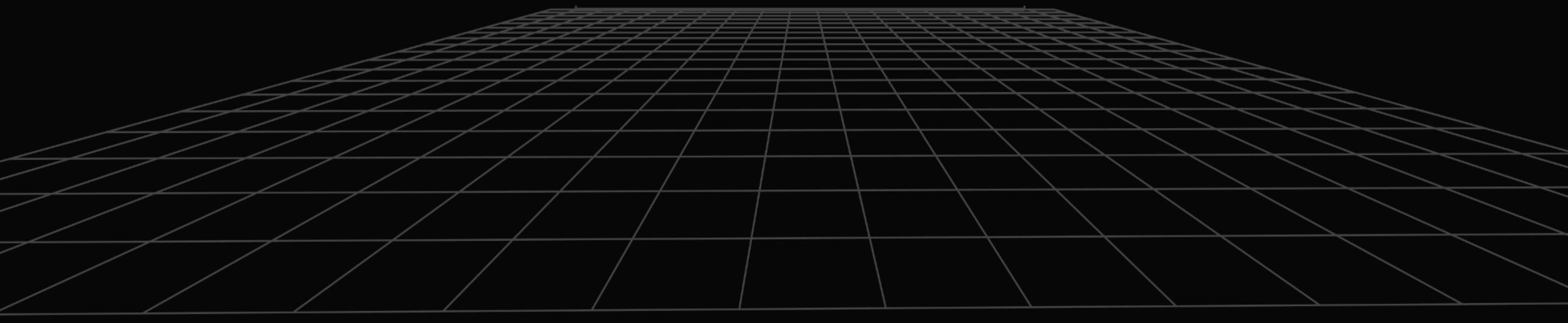
EXAMPLE

Memoization



FAQ

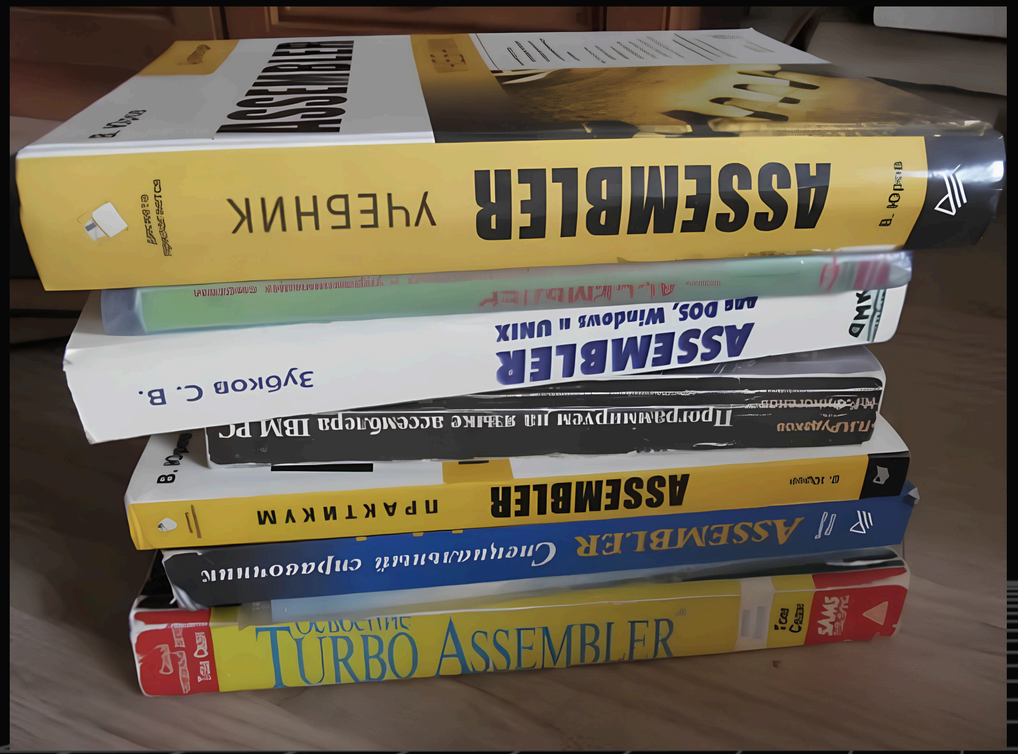
Паттерны и приемы



УСТРОЙСТВО СТЕКА

СТЕК

Специально отведенная область памяти для хранения временных данных (еще называют аппаратным стеком)

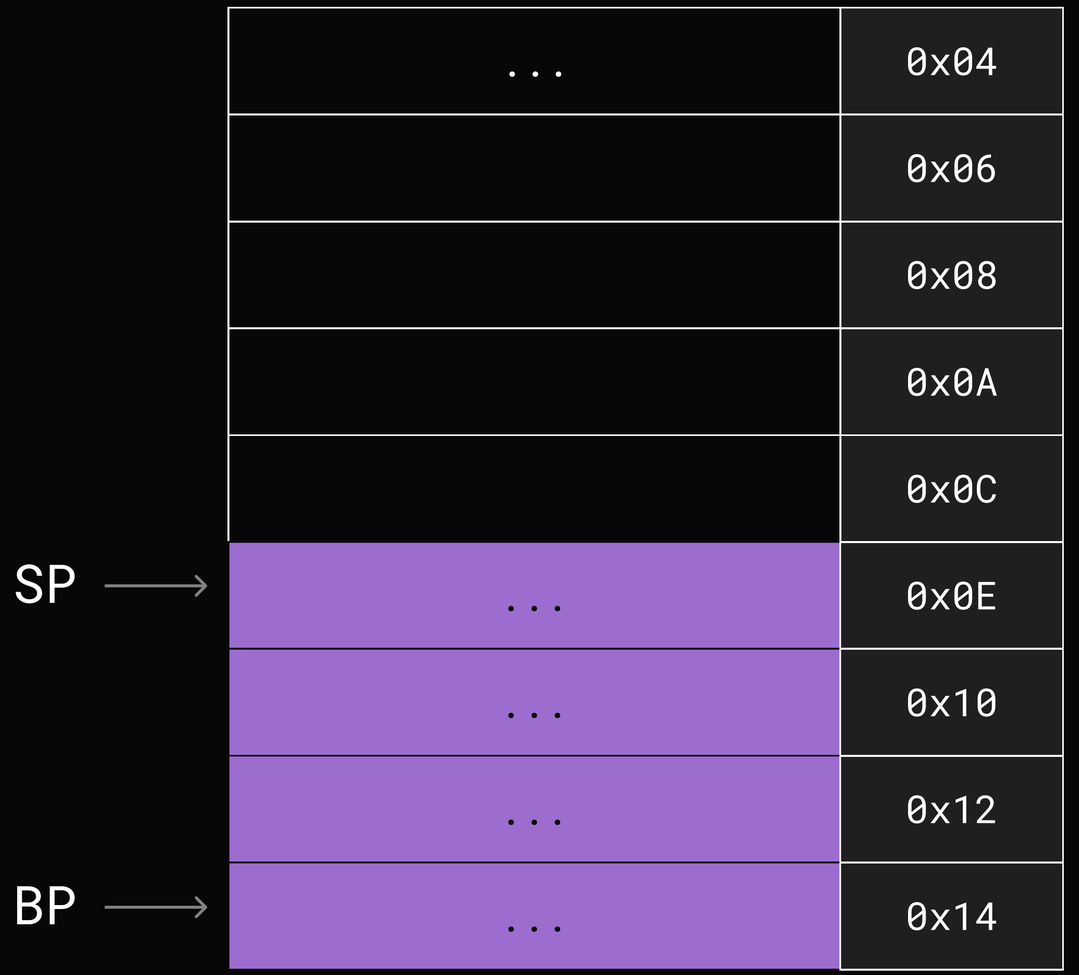


SP (STACK POINTER) РЕГИСТР

это вершина стека или адрес самого последнего добавленного элемента

BP (BASE POINTER) РЕГИСТР

это начало фрейма или адрес,
с которого в стек вносятся
или изменяются значения



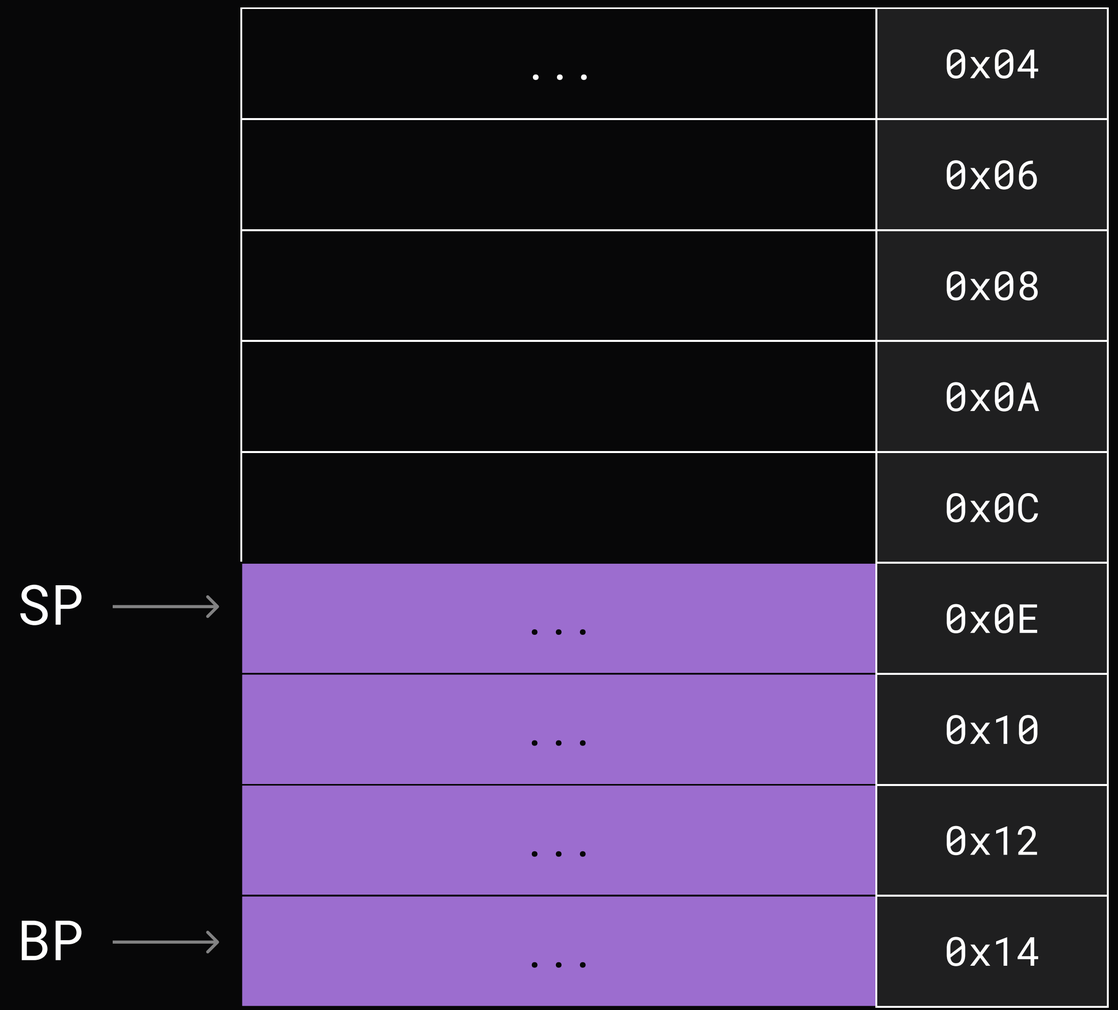
ОСНОВНЫЕ КОМАНДЫ

PUSH инструкция — помещает данные на вершину стека

POP инструкция — снимает данные с вершины стека

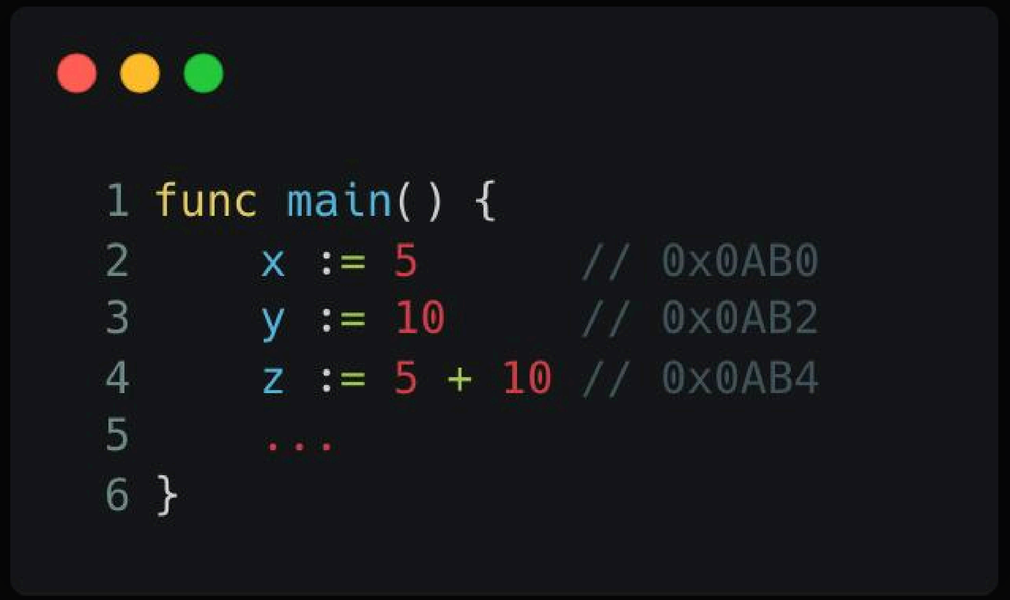
PUSH 5

POP AX

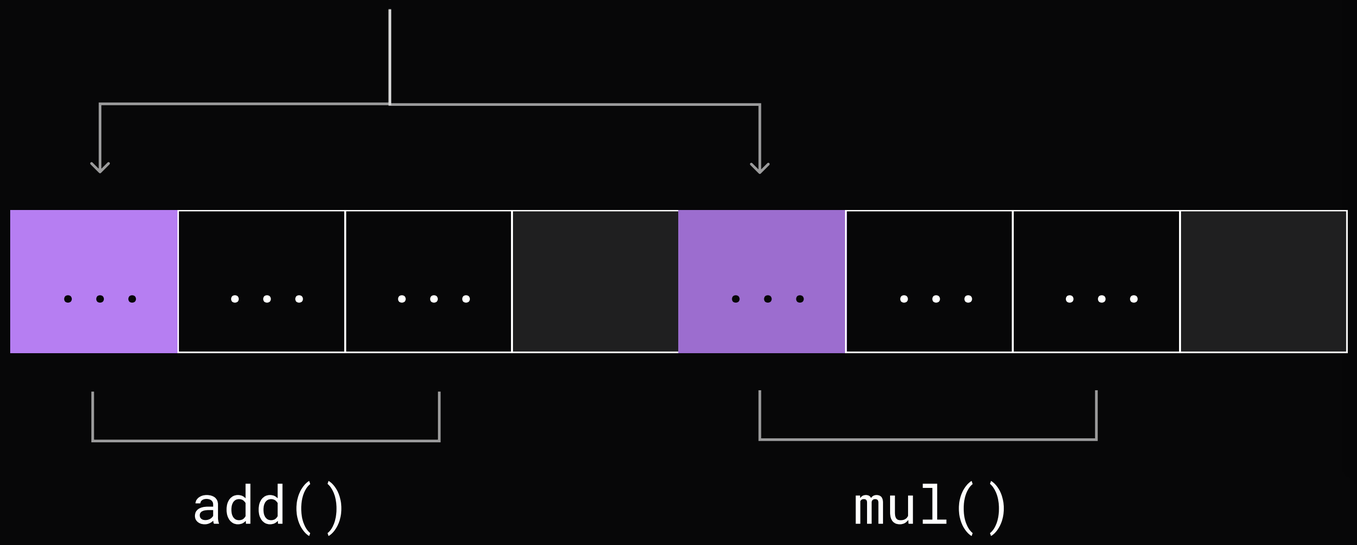


IP (INSTRUCTION POINTER) РЕГИСТР

это указатель на команду,
которая будет выполняться
следующей



Адрес начала функции

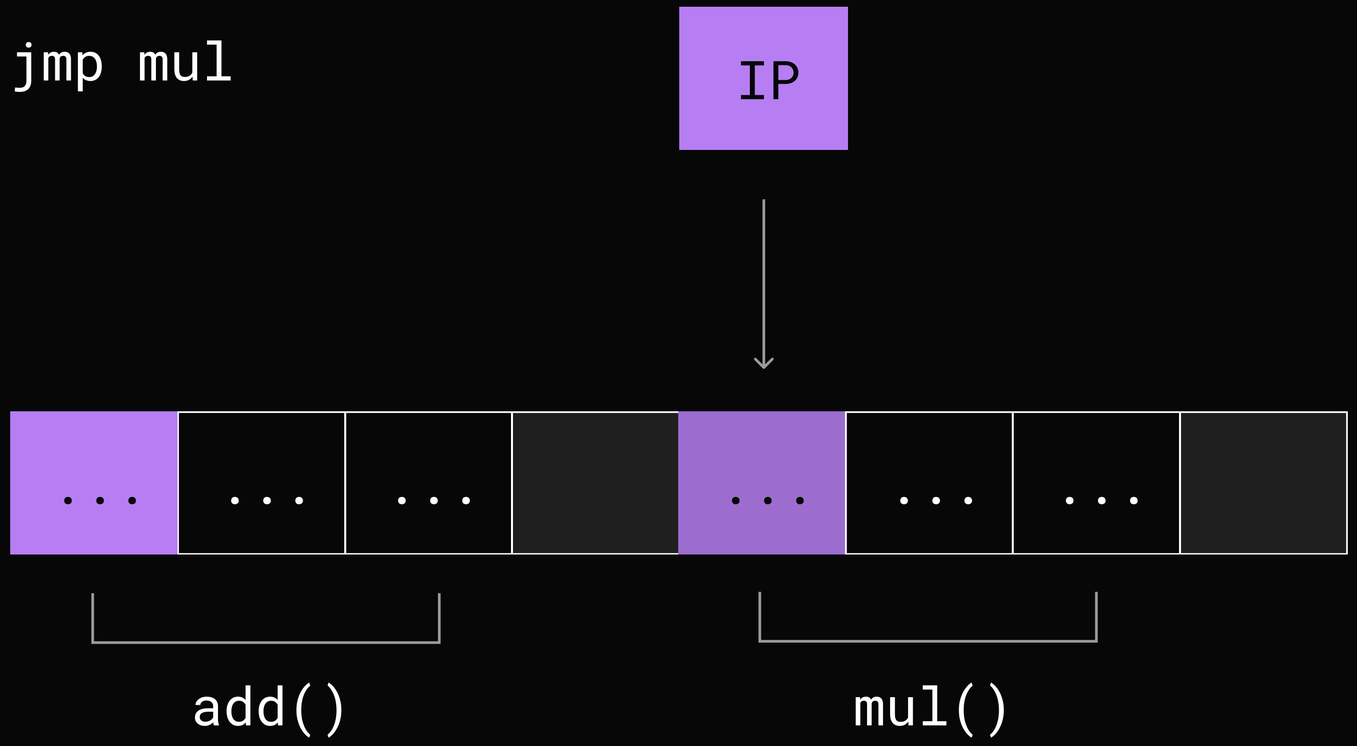



```go
func sum(x, y int) int {
    return x + y
}

func mul(x, y int) int {
    return x * y
}
```

Для того, чтобы начать выполнить функцию –
нужно загрузить ее адрес начала в **регистр IP**

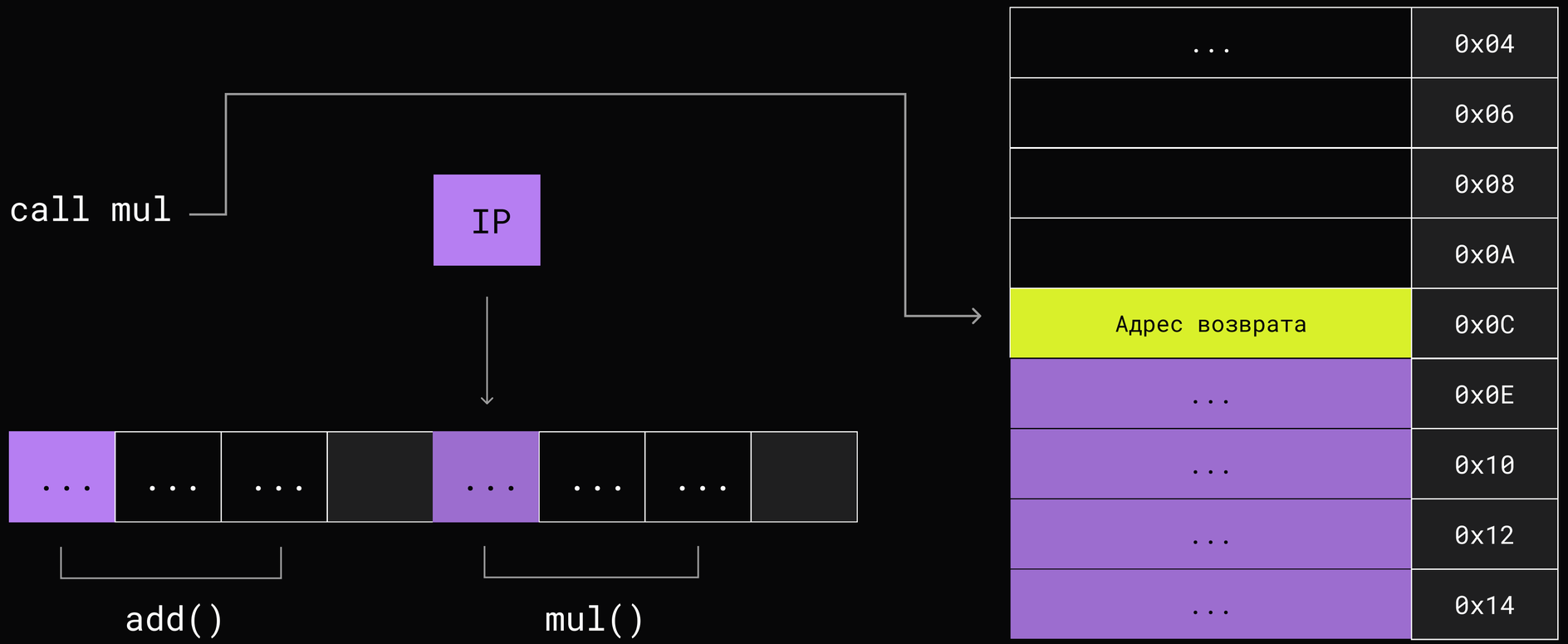
JMP инструкция - просто прыгает из одного участка кода в другой по определенному адресу (меняя при этом IP)

```
jmp mul
```

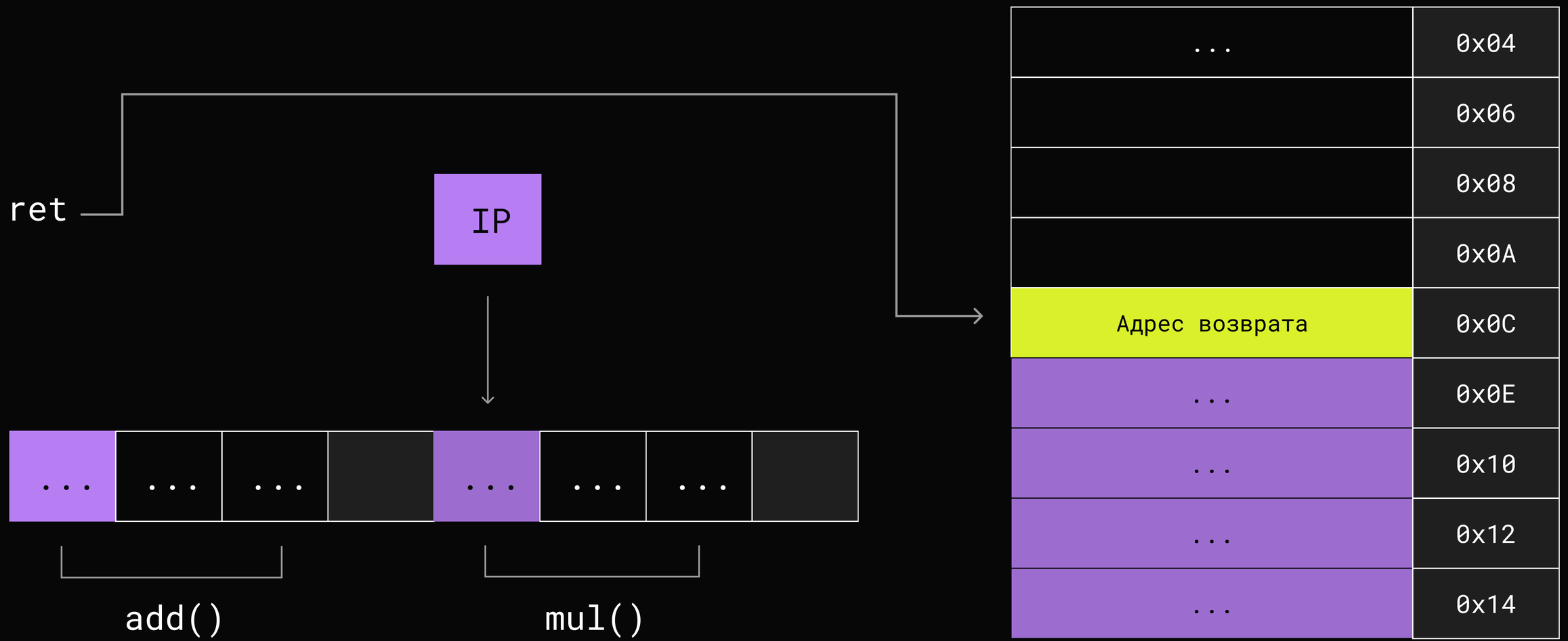


ДОСТАТОЧНО ЛИ ИНСТРУКЦИИ JMP ДЛЯ ВЫЗОВА ФУНКЦИИ?

CALL инструкция - перед тем, как прыгнуть по какому-то адресу – заносит в стек адрес возврата, после чего происходит прыжок по указанному адресу (адрес возврата – это адрес инструкции следующий за инструкцией CALL)

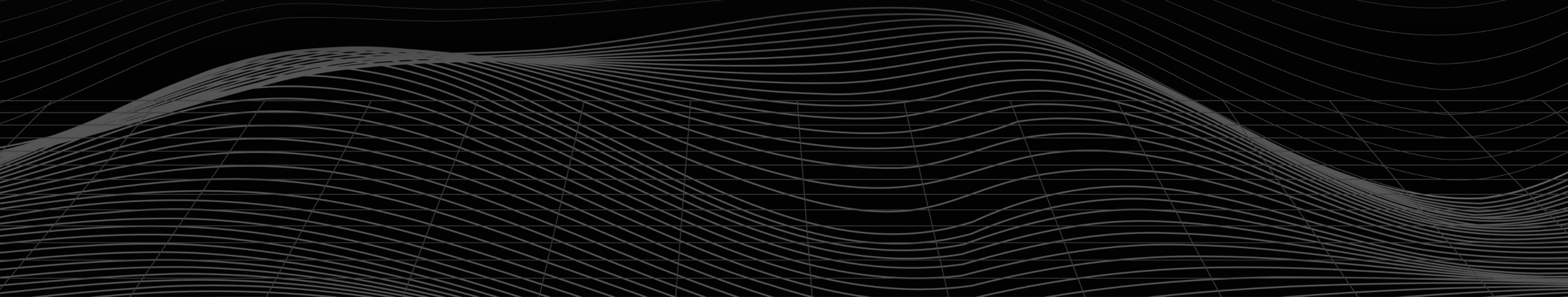


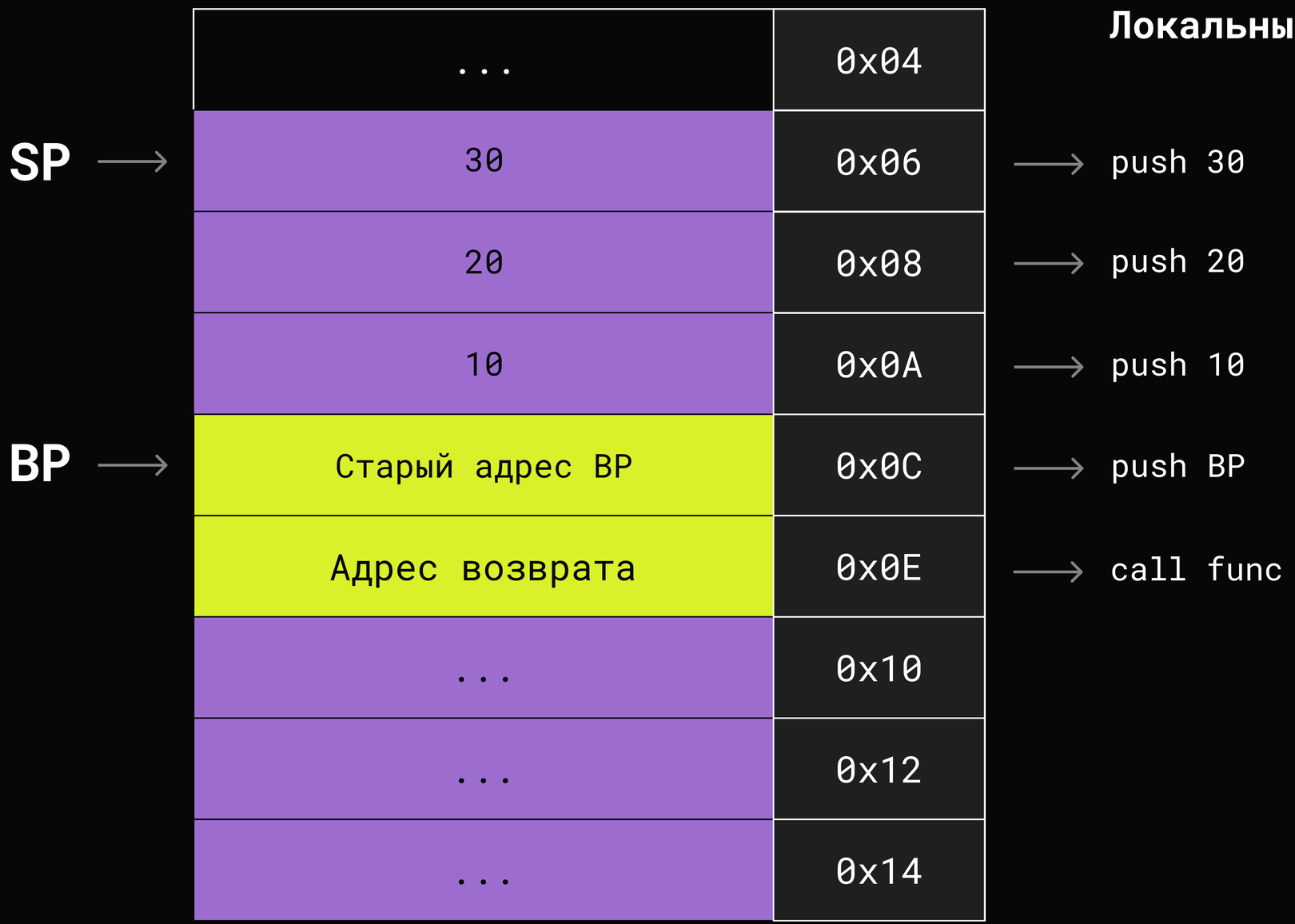
RET инструкция - вытаскивает из стека адрес возврата и затем переходит по нему



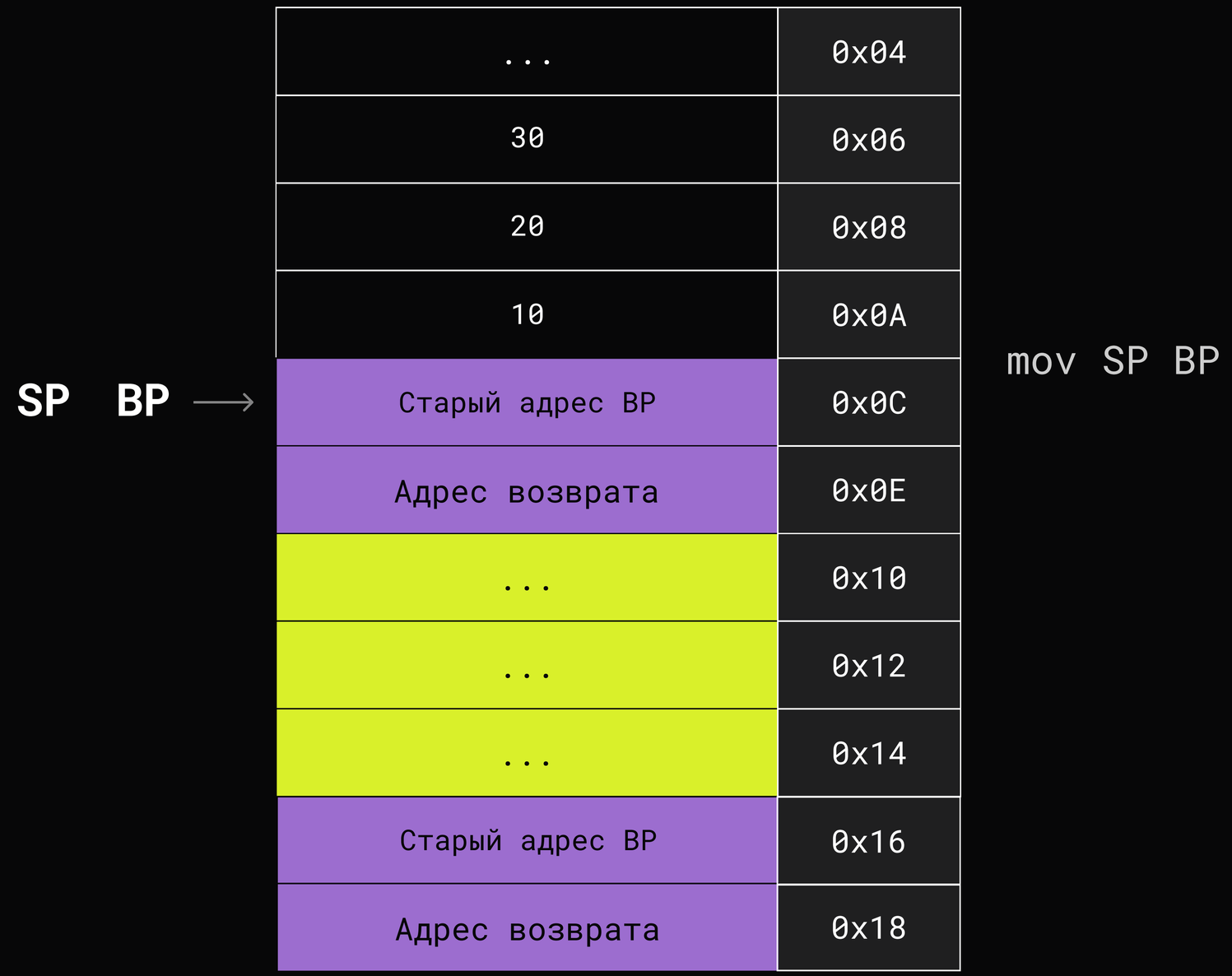
Фрейм функции – это локальное окружение функции в виде ее переменных

Регистр BP всегда статичен, то есть будет указывать на одну и ту же ячейку




Локальные переменные
SP
BP
...
0x04
30
0x06
push 30
20
0x08
push 20
10
0x0A
push 10
Старый адрес BP
0x0C
push BP
Адрес возврата
0x0E
call func
...
0x10
...
0x12
...
0x14

КАК ПРОИСХОДИТ ВОЗВРАТ ИЗ ФУНКЦИИ?

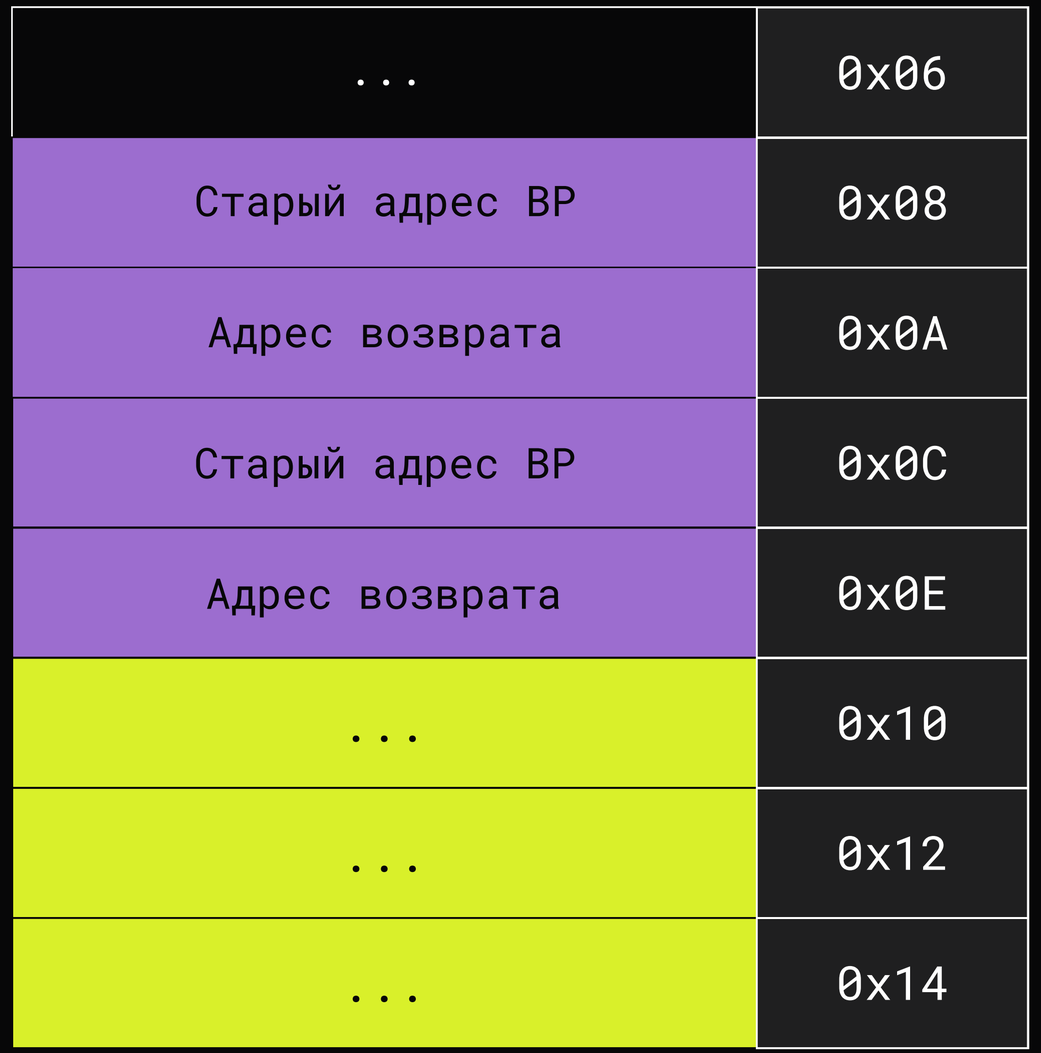

...
0x04
30
0x06
20
0x08
10
0x0A
SP BP
Старый адрес BP
0x0C
Адрес возврата
0x0E
...
0x10
...
0x12
...
0x14
Старый адрес BP
0x16
Адрес возврата
0x18
mov SP BP


```
mov SP BP
pop BP
```


```
mov SP BP
pop BP
ret
```

ПЕРЕПОЛНЕНИЕ СТЕКА

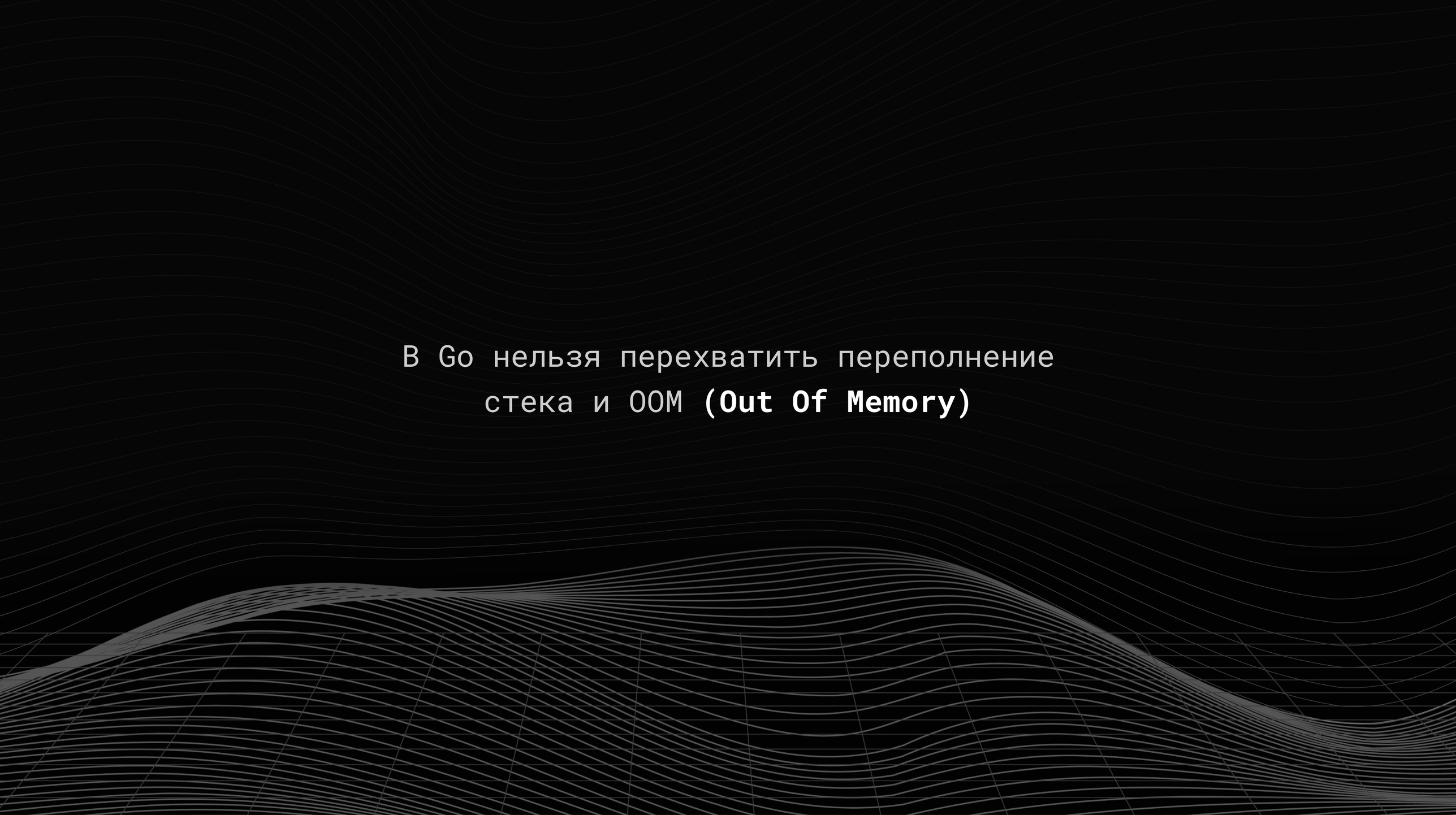
```
1 func recursion() {
2     recursion()
3 }
```



EXAMPLE

Stack overflow

В Go нельзя перехватить переполнение стека и OOM **(Out Of Memory)**



Terminal: question + ⚙ –

КАК ПЕРЕДАТЬ АРГУМЕНТЫ В ФУНКЦИЮ?

STDCALL

Аргументы передаются перед вызовом в обратном порядке

Terminal: question

КТО БУДЕТ СДВИГАТЬ SP

на нужное количество байт при возврате из функции?

ВЫЗЫВАЕМАЯ СТОРОНА

ВЫЗЫВАЮЩАЯ СТОРОНА

СМЕЩЕНИЕ ИДЕТ ОТ BP,
ТАК КАК ОН ВСЕГДА СТАТИЧЕН

BP-4 - локальная переменная 2

BP-2 - локальная переменная 1

BP - указатель на базу

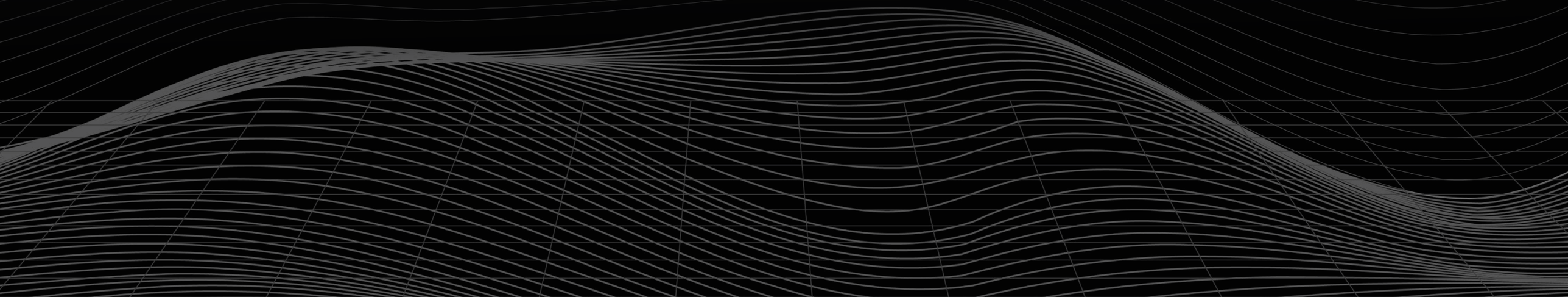
BP+2 - адрес возврата

BP+4 - аргумент функции 1

BP+6 - аргумент функции 2

...	0x06
20	0x08
10	0x0A
Старый адрес BP	0x0C
Адрес возврата	0x0E
1	0x10
2	0x12

Можно передавать аргументы не только через стек,
но и через регистры и глобальные переменные

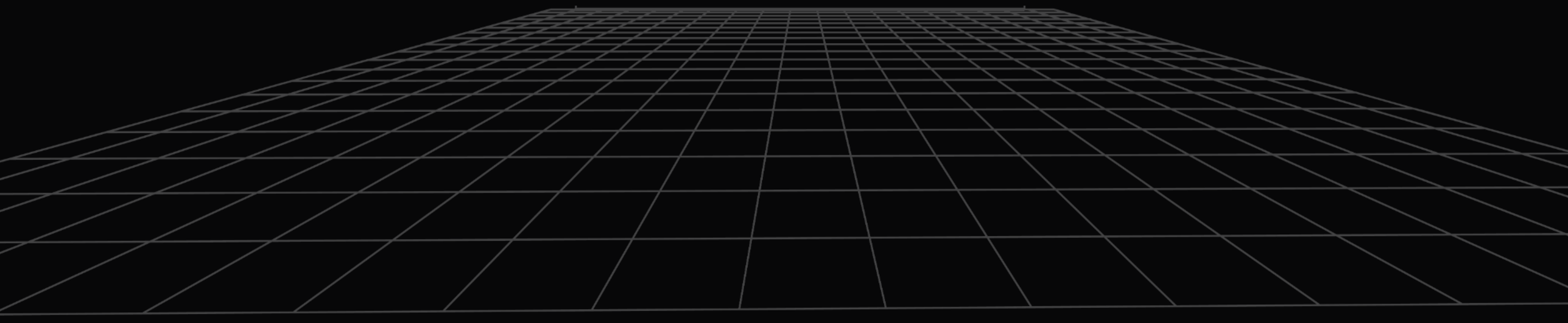


В GO 1.17 для архитектуры AMD64 stack-based ABI был изменен на register-based

На искуственных бенчмарках доступ к аргументам в регистрах был на 40% быстрее, чем к аргументам на стеке

FAQ

Устройство стека



ПОЖАЛУЙСТА, ЗАПОЛНИ ОПРОС О ЗАНЯТИИ

Ссылка в чате и в группе участников