Instituto de Matemática e Estatística
Universidade de São Paulo

Linguagem Go

MAC5742/0219
Introdução à Programação Concorrente, Paralela e Distribuída

Professor: Alfredo Goldman vel Lejbman

Eduardo Gonçalves Pinheiro — 8936798
Leonardo Pereira Macedo — 8536065
Ricardo Akira Tanaka — 9778856

29 de junho de 2017

Sumário

1 Introdução

1.1 Organização do trabalho

Este trabalho tem como objetivo aprender e apresentar uma linguagem de programação recente chamada *Go*. Analisaremos desde as ferramentas que ela oferece para um programador até a forma como é estruturada.

O restante desta seção 1 irá explicar como a linguagem veio a ser desenvolvida, bem como suas principais características. A seção 2 apresenta rapidamente instruções para instalar e compilar programas em *Go*, com um foco voltado para sistemas *Unix*, além de informações sobre o compilador em si. Nas seções 3 a 6, estudaremos a sintaxe e ferramentas da linguagem, passando por variáveis, laços, funções, etc. Na seção 7, daremos atenção especial à parte de concorrência existente em *Go*, uma vez que a monografia é destinada para uma disciplina deste tópico. Por fim, a seção 8 apresenta uma breve conclusão em relação ao trabalho como um todo.

1.2 História de *Go*

Go, também conhecida como *golang*, é uma linguagem de código aberto, desenvolvida na Google por três engenheiros de software (figura 1):

- **Robert Griesemer** trabalhou na máquina virtual do *Java Hotspot* [1].
- **Rob Pike** trabalhou no *Bell Labs*, braço de pesquisa e desenvolvimento nos EUA de onde surgiram *C*, *C++* e o transistor, dentre outras criações. Nesse local, Pike era integrante da equipe de *Unix* [2].
- **Ken Thompson** também trabalhou no *Bell Labs*. Além de ter implementado o sistema operacional *Unix* original, Thompson desenvolveu a linguagem *B*, predecessora direta da linguagem *C*, e foi o co-criador do UTF-8, juntamente com Rob Pike [3].

Figura 1: Da esquerda para a direita, Robert Griesemer, Rob Pike e Ken Thompson [4]

Em 2007, a Google passava por dificuldades em fazer manutenção de seus vários servidores, com várias linhas escritas em *C++*. Desejava-se desenvolver uma linguagem produtiva que pudesse ser melhor usada no contexto, tendo as seguintes características: [5]

- Deve ser **escalável** para programas de grande porte com diversas dependências, com equipes de vários programadores trabalhando em conjunto.
- Deve ser **familiar**, **rápida de se aprender**, pois diversos programadores da Google estão familiares com linguagens da família *C*.
- Deve ser **moderna**. *C*, *C++* e, de certa forma, *Java* foram criadas antes do surgimento de desenvolvimento Web e de computadores com vários processadores. Uma linguagem que fosse criada com redes e multiprocessamento em mente certamente seria benéfica.

Tendo os fatores acima em mente, os três desenvolvedores também buscaram se basear em aspectos positivos de outras linguagens (uma das principais motivações citadas pelo trio foi a complexidade desnecessária de *C++*). *Go* foi anunciada abertamente em novembro de 2009, mas sua primeira versão (*Go 1.0*) só foi lançada em 2012 [6].

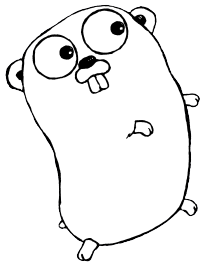
1.3 Principais características

Dentre suas principais características, podemos citar:

- Segue parcialmente a tradição de *C*;
- Linguagem compilada e estaticamente tipada;
- Coleta de lixo;
- Preocupação com causas de problemas ao acessar a memória (como ponteiros e *buffers*);
- Programação concorrente no estilo CSP, isto é, *Communicating Sequential Processes* [7]. Este padrão estabelece que componentes denominados processos interagem entre si e com o ambiente através de uma comunicação definida por interfaces.
- Filosofia *"Share memory by communicating"* (com o uso de *channels*, por exemplo, que veremos na seção 7.2), oposta a *"Communicate by sharing memory"* (o que ocorre em linguagens que usam variáveis compartilhadas e *mutex*) [8].

O logo da linguagem (figura 2), criado pela americana Renée French, é um roedor (*gopher*) sem nome, apesar de referirem-se a ele como *Go gopher*.

Figura 2: Mascote da linguagem *Go*, o *Go gopher* [9]



Go possui um site oficial [10] bastante completo e atualizado, onde é possível ler a documentação e acompanhar mudanças sendo feitas na linguagem, dentre outras possibilidades. Na época em que este relatório foi escrito, a linguagem encontrava-se na versão 1.8, lançada em fevereiro de 2017 [11].

1.4 Para que *Go* é usada?

Apesar de ser uma linguagem de propósito geral, a existência de coleta de lixo torna *Go* inadequada para *software* embutido ou em tempo real, pois perde-se um pouco de velocidade [12].

Para sabermos o quão rápido é o desempenho de *Go*, referenciamos um site de *benchmarks*, que compara a linguagem com outras de uso comum para programas como *Mandelbrot*, cálculo de dígitos de π e árvores binárias [13]. A conclusão obtida é que *Go* possui uma velocidade bem parecida com *Java*; isto é, mais lenta que *C*/*C++*, mas mais rápida que linguagens de *script*, como *Python* e *Ruby*.

Por outro lado, a linguagem possui uma boa estrutura para desenvolvimento de servidores com elevada concorrência, sistemas grandes e distribuídos e sistemas de rede altamente escaláveis. Nessa situação, é uma linguagem que pode competir com *C++*, pois algumas ferramentas da linguagem (*goroutines* e *channels*) são bem mais simples e intuitivas de usar se comparadas a *threads* em *C++*.

Dentro da Google, diversos novos servidores estão sendo feitos em *Go*, enquanto algumas equipes consideram até em reescrever código existente nessa linguagem [14].

2 Compilador *Go*

Explicamos, nesta seção, como instalar o compilador de *Go*, seus comandos básicos e algumas de suas características.

2.1 Instalação

O site oficial da linguagem conta com instruções para diversos Sistemas Operacionais. Em um sistema *Unix*, a maneira mais simples de instalar o compilador de *Go* é pelo terminal. Por exemplo, para *Ubuntu* ou *Debian*, basta digitar:

```
$ sudo apt-get install golang  # Instala a versão mais recente da linguagem
```

Também é possível instalar *Go* por partes; envolve clonar um repositório, alterar algumas variáveis de ambiente e executar alguns comandos. Caso o leitor tenha interesse, recomenda-se consultar este tutorial.

2.2 Comandos básicos

Por ser uma linguagem compilada, *Go* conta com o compilador *gc*, usado na linha de comando como `go`. A maneira como é usado depende do argumento digitado logo após chamá-lo. Os comandos mais simples que explicaremos aqui são:

- **build**: Usado para compilar código da linguagem (por padrão, arquivos com extensão `.go`). Por exemplo, para compilar o arquivo `meuprograma.go`, basta digitar no terminal:

  ```
  $ go build meuprograma.go  # Gera o binário "meuprograma"
  ```

- **run**: Compila e executa código *Go*, mas sem gerar um binário como em `go build`.
- **fmt**: Formata arquivos de código fornecidos como argumento, salvando-os com o mesmo nome (isto é, sem criar *backup*).
- **version**: Imprime a versão do compilador *Go* em uso.

2.3 Mais detalhes sobre o compilador

O compilador *gc* foi inicialmente escrito em *C*, sendo que em 2015 foi reescrito em *Go* para a versão 1.5 da linguagem. Também citamos que há um compilador *frontend* para *C*, *gccgo*, escrito em *C++* [15].

Uma das características que definem *Go* é o tempo de compilação, que é fonte constante de otimizações, sendo capaz de compilar em segundos ao invés de minutos para um programa análogo em *C++*. Na verdade, a compilação de *C/C++* é mais lenta devido à necessidade de analisar diversos arquivos *header*, enquanto *Go* usa gerenciamento de dependências. Em outras palavras, só é necessário olhar para os pacotes que foram diretamente importados, sem se preocupar com o que eles importam. Com isso, o número de dependências fica linear, acelerando a compilação [16].

A *LLVM Project* também disponibiliza um *frontend* para *Go*, o `llgo` [17]. Segundo o FAQ da linguagem, foi cogitado utilizar o LLVM como compilador, mas ele se mostrou lento, levando à decisão de desenvolver o *gc*. Uma das características de programas em *Go* é o tamanho do seu binário, que é causado pelas bibliotecas estáticas. Logo, todos os binários são monolíticos e incluem o *runtime* e suas informações de suporte, como checagem de tipos.

2.4 *Runtime*

Go possui um *runtime* que na verdade é uma biblioteca incluída em todo binário da linguagem [18]. Este implementa o Coletor de Lixo, concorrência, gerenciamento de pilhas e outras funções críticas, sendo análogo ao *libc*. É necessário reforçar que o termo *runtime* é utilizado de maneira liberal, pois não existe máquina virtual para execução do *Go*. Os programas são compilados para código de máquina nativo, e o *runtime* fornece serviços críticos aos binários.

Assim como o compilador *gc*, inicialmente o *runtime* foi implementado em *C*, mas atualmente já foi traduzido para *Go*, com trechos em *Assembly*.

3 Componentes básicas de *Go*

As próximas seções são inspiradas em uma sequência de tutoriais disponíveis no próprio site da linguagem, explicando desde como declarar variáveis, funções e pacotes até estruturas mais complexas [19].

3.1 Pacotes

Todo programa em *Go* é composto por diferentes pacotes. Pode-se entendê-los como os módulos na linguagem *C* (isto é, o *header* e a sua implementação).

Pacotes podem ser importados e usados por outros pacotes. Por exemplo, para podermos escrever o clássico *Hello, world!*, precisamos do pacote `fmt`, que contém funções de impressão na tela:

```
1 package main  // Nome do pacote representado por este arquivo
2
3 // Para importar pacotes, use "import"
4 import "fmt"
5
6 func main() {
7   // Vamos usar a função Println() do pacote fmt
8   fmt.Println("Hello, world!")
9 }
```

Go não é uma linguagem de *script*, então precisa da função `main()` como indicadora de onde o programa começa. Deve-se destacar que todo arquivo *precisa* ter um nome de pacote, como é o caso do `package main` no exemplo acima.

O leitor pode ter estranhado o uso de uma letra maiúscula em `Println()`, e isso é justificado. Em *Go*, um nome é *exportado* se ele começar com letra maiúscula. Em outras palavras, nomes iniciados por uma letra maiúscula podem ser usadas externamente ao pacote, enquanto termos que começam com uma letra minúscula só existem localmente. Usar `println()` no exemplo acima daria um erro de compilação, pois a função não seria encontrada.

No caso de existirem diversos pacotes para importar, é boa prática utilizar um `import` "fatorado":

```
import (
  "fmt"
  "math"
)
```

3.2 Funções

Funções são trechos de código altamente reutilizáveis e uma ferramenta importante para qualquer programador. Assim como em diversas linguagens, funções em *Go* podem conter zero ou mais argumentos. Se quisermos fazer uma função que soma dois inteiros, usamos a palavra reservada **`func`** e escrevemos:

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

Para programadores acostumados com *C/C++* e *Java*, pode parecer estranho escrever o tipo da variável/-função após sua declaração. A vantagem principal está na facilidade de leitura [20]; suponha, por exemplo, o seguinte ponteiro para função em *C*, sem os nomes das variáveis:

```
int (*fp)(int (*)(int, int), int)
```

A expressão acima é um ponteiro para uma função que retorna um valor inteiro. Seus dois argumentos são um ponteiro para uma função inteira, que recebe dois números inteiros, e um outro número inteiro. Mesmo para um programador experiente na linguagem, a notação não é de simples leitura.

Vamos comparar com o caso análogo em *Go*, que seria uma variável de função. Note a facilidade maior de leitura no estilo "esquerda para direita":

```
f func(func(int, int) int) int
```

Go possui também algumas simplificações e recursos adicionais para funções:

- Se dois ou mais parâmetros consecutivos possuírem o mesmo tipo, pode-se declarar este tipo apenas para o último parâmetro da sequência. Por exemplo, podemos simplificar um pouco a função `soma()` escrita acima:

```
// x e y possuem o mesmo tipo, então só y é declarado explicitamente como int
func soma(x, y int) int {
  return x + y
}
```

- Assim como em algumas linguagens mais modernas, como *Python* e *Ruby*, pode-se retornar mais de um resultado:

```
// Note o tipo de retorno
func troca(aqui, ali string) (string, string) {
  return ali, aqui
}
```

- As variáveis de retorno de uma função podem ser nomeadas. Tal prática leva elas a serem implicitamente declaradas no início da função. Além disso, torna-se opcional escrever tais variáveis após o comando `return`. Esta prática é conhecida como *naked return*, e não é recomendada para funções longas por atrapalhar a legibilidade do código.

  ```
  func triplica(x int) (triplo int) {
    triplo = 3*x
    return  // Naked return
  }
  ```

  É importante notar que o `return` acima é necessário, com ou sem argumentos, uma vez que a função retorna algo. A situação é diferente de uma função que não retorna nada:

  ```
  func imprime_numero(x int) {
    fmt.Println(x)
    return  // Opcional
  }
  ```

Por fim, ressaltamos que funções em *Go* são de primeira classe, isto é, pode-se passá-las como argumento para outras funções. Também é possível definir funções que retornam funções. Veremos mais sobre isso na seção 5.7.

3.3 Variáveis

Como o nome diz, variáveis referem-se ao que *varia*. O uso delas permite guardar estado dentro de um programa, possivelmente modificando os resultados obtidos ao executar o mesmo.

3.3.1 Declaração

Variáveis em *Go* são declaradas usando a palavra reservada **`var`**. O tipo da variável pode ser declarado após seu nome, ou ser implícito se um valor for usado em sua inicialização.

```
var x int
var y = 1  // y é inferido como int

func main() {
  // Inicialização de várias variáveis na mesma linha
  var str, pi, n = "oi", 3.14, 10

  // É possível declarar várias variáveis do mesmo tipo em uma linha...
  var a, b, c int

  // Mas NÃO é possível declarar variáveis de tipos diferentes na mesma linha!
  var alpha float32, beta int  // ERRADO!
}
```

Go possui *escopo estático*. Em outras palavras, mudanças em uma variável local não são refletidas em uma variável global de mesmo nome. Veja o trecho de código abaixo:

```
var a = 1  // Variável global

func muda_var() {
    a = 2  // Modifica variável global
    fmt.Println("Em muda_var(), a =", a)
}

func main() {
    var a = 0  // Variável no escopo de main()
    fmt.Println("No main(), a =", a)
    muda_a()
    fmt.Println("Depois de chamar muda_var(), a =", a)
}
```

Veja que outra variável `a` é declarada dentro de `main()`. A mudança feita em `muda_var()` vale somente para a variável global, não tendo como afetar algo que está no escopo de outra função. Se executássemos este trecho, obteríamos na saída:

```
No main(), a = 0
Em muda_var(), a = 2
Depois de chamar muda_var(), a = 0
```

Há também uma notação curta para declarar variáveis, válida somente dentro de funções: `:=`. Ao ser usada, omite-se a palavra `var`.

```
x := 1  // ERRADO! x não está dentro de uma função!

func main() {
  n, y, str := 4, 2.0, "oi"
}
```

3.3.2 Tipos existentes

A tabela 1 contém os principais tipos de variáveis existentes em *Go*. Veja que tipos numéricos permitem a especificação de seu tamanho com um sufixo.

Tabela 1: Principais tipos de variáveis existentes em *Go*

Tipo de dado	Palavras reservadas	Exemplos de valores	Observação
Variável booleana	`bool`	`true` `false`	—
Cadeia de caracteres	`string`	“batata” “tomate”	Não podem ser escritas com aspas simples (como `'batata'`)
Inteiros com sinal	`int`, `int8`, `int16`, `int32`, `int64`	42 -1729	O tamanho de `int` depende do sistema operacional (32 ou 64 bits)
Inteiros sem sinal (*unsigned*)	`uint`, `uint8`, `uint16`, `uint32`, `uint64`	0 8001	O tamanho de `uint` depende do sistema operacional (32 ou 64 bits)
Ponto flutuante	`float32`, `float64`	3.1416 2.72	`float64` equivale ao `double` de outras linguagens
Números complexos	`complex64`, `complex128`	10+3i 1-2i	O tamanho especificado no tipo é dividido em duas partes iguais, para guardar os valores real e complexo

Também há os seguintes tipos especiais:

- **`byte`**: Equivalente a `uint8`.
- **`rune`**: Equivalente a `int32`, representa um ponto de código em *Unicode*.
- **`uintptr`**: Tipo usado para tratar um ponteiro (assunto que veremos na seção 5.1) para um `uint`.

O tipo da variável importa se ela for declarada sem um valor inicial explícito. Nessas situações, *Go* a inicializa com seu *valor zero*: 0 para variáveis numéricas, “” (*string* vazia) para `string` e `false` para `bool`.

É importante tambem destacar que *Go* não realiza conversões implícitas entre tipos, como ocorre em linguagens como *C/C++* e *Java*. Deve-se fazer a conversão manualmente, usando o tipo da variável como uma “função”:

```
var cat1, cat2 int = 3, 4

// Sqrt() demanda float64 como argumento
var hipot float64 = math.Sqrt(float64(cat1*cat1 + cat2*cat2))
```

3.3.3 Constantes

Constantes são variáveis imutáveis, isto é, não podem ser alteradas depois de declaradas e inicializadas.

Para criar uma constante em *Go*, usa-se a palavra reservada **`const`**. Ressalta-se que *não* é possível declarar constantes com a notação `:=`.

```
const pi = 3.14
const verdade = true
```

4 Controle de fluxo

Linguagens de programação costumam contar com mecanismos de controle de fluxo para direcionar e reaproveitar código. Apresentamos, nesta seção, os mecanismos existentes em *Go*.

4.1 Laços

Laços são usados para executar repetidamente trechos de código, muitas vezes alterando a cada passo as variáveis envolvidas. A palavra reservada usada para criar laços é o **`for`**. Seu uso é semelhante a *C/C++* e *Java*, com três cláusulas:

- A primeira cláusula é executada antes da primeira iteração.
- A segunda cláusula é verificada antes de começar cada iteração. Se a condição for verdadeira, o laço é encerrado.
- A terceira cláusula é executada após cada iteração.

O trecho abaixo, por exemplo, soma os números de 1 a 10 em `soma`.

```
soma := 0
for i := 0; i < 10; i++ {
  soma += i
}
```

O uso de chaves `{ }` é obrigatório para indicar qual o trecho a ser executado em laço. Ressalta-se, também, que não é necessário sempre usar as três cláusulas. Inclusive, o laço **`while`** encontrado em diversas linguagens é feito da seguinte forma em *Go*:

```
soma := 1
for soma < 1000 {
  soma += soma
}
```

Um laço infinito é criado se o `for` não tiver nenhuma cláusula. Uma forma de sair desse trecho de código é usar a palavra reservada **`break`**, usada para sair do laço mais interno no código.

```
for  {
    break;  // Sai do laço mais interno no código
}
```

4.2 Condicionais

Condicionais surgem da necessidade de divergir certos pontos do código, de acordo com o estado em que uma ou mais variáveis se encontram.

4.2.1 `if/else`

A forma mais simples de criar uma condição no código é através da palavra reservada **`if`**, que executa seu bloco apenas se sua condição for satisfeita:

```
func verifica_positivo(x int) bool {
  if x >= 0 {
    return true
  }
  return false
}
```

Em *Go*, é possível realizar uma inicialização antes da condição do `if`, similar à primeira cláusula do `for`.

```
func verifica_soma_10(x int, y int) bool {
  if soma := x + y; soma == 10 {
    return true
  }
  return false
}
```

Por fim, há também a palavra reservada **`else`**, que executa código se a condição do `if` não for satisfeita.

```
func verifica_maior_int(x, y int) {
    if x > y {
        fmt.Printf("%d é maior que %d\n", x, y)  // Equivalente ao printf() de C
    } else if x < y {
        fmt.Printf("%d é menor que %d\n", x, y)
    } else {
        fmt.Printf("Os dois números são iguais: %d\n", x)
    }
}
```

4.2.2 `switch`

O uso de **`switch`** equivale a uma cadeia de `if/else`. Cada `case` verifica uma condição em relação ao termo após o `switch`, executando código se a verificação for satisfeita.

```
func escreve_numero(x int) string {
  switch x {
    case 1:
      return "um"
    case 2:
      return "dois"
    default:  // Equivale a "if true"
        return "Cansei :P"
    }
}
```

Ao contrário de *C/C++* e *Java*, não é necessário colocar um *break* no final do `case` para o programa sair do `switch`; *Go* automaticamente faz isso quando uma condição é satisfeita.

4.3 O comando `defer`

Conforme o nome diz, a palavra reservada **`defer`** defere a execução de uma função, colocando-a numa pilha. Quando a função na qual o referido `defer` está termina, executa-se cada chamada existente na mencionada estrutura de dados, seguindo uma ordem LIFO (*Last In, First Out*).

Como exemplo, veja a função `main()` abaixo:

```
func main() {
  fmt.Println("Vamos contar?")

  for i := 0; i < 3; i++ {
    defer fmt.Println(i)
  }

  fmt.Println("O main() acabou")
}
```

A tabela 2 representa a pilha do `defer` quando a função `main()` terminar.

Tabela 2: Pilha do `defer`

(Topo da pilha)
`fmt.Println(2)`
`fmt.Println(1)`
`fmt.Println(0)`

Com a tabela anterior, fica fácil ver que a saída do programa será:

```
Vamos contar?
O main() acabou
2
1
0
```

`defer` é útil para simplificar funções que realizam operações de limpeza/término em seu final. Por exemplo, pode-se usá-lo para fechar a conexão de um *socket* após o fim (inesperado ou não) de uma comunicação.

5 Tipos avançados

Nesta seção, serão apresentados tipos de dados mais complexos presentes em *Go*. Também aprofundaremos um pouco mais o tópico de funções no final desta parte.

5.1 Ponteiros

Assim como *C*, *Go* possui ponteiros, e a forma de uso é bem parecida:

```
func main() {
  i := 42
  p := &i          // Aponta para i
  *p = *p + 1687   // Muda o valor de i através do ponteiro p
  fmt.Println(*p)
}
```

O código acima imprimiria 1729 ao ser executado.

Porém, *Go* não possui aritmética de ponteiros como em *C*/*C++*. Segundo os criadores, os motivos principais são por segurança e para facilitar a implementação da coleta de lixo [21].

```
func main() {
  i := 42
  p := &i
  p++  // ERRADO! Não é aceito na linguagem
  fmt.Println(*p)
}
```

5.2 *Struct*

Uma **`struct`** pode ser entendida como uma coleção de campos que representarão um novo tipo de dados. Pode-se declarar uma `struct` com a palavra reservada `type`:

```
type Vertice struct {
  X int
  Y int
}
```

Para acessar um campo de uma `struct`, utilizamos o caractere de ponto "`.`".

```
func main() {
  var p Vertice
  p.X = 42
}
```

Uma ***struct literal*** indica um valor de `struct` recém alocado, sendo usado para facilitar sua criação:

```
func main() {
  // Cria uma struct literal (X = 1, Y = 2)
  p = Vertice{1, 2}
}
```

`Structs` podem ser acessadas por ponteiros da mesma maneira que variáveis comuns. No entanto, ao contrário de *C*, que usa `->` para simplificar a escrita (`struct->campo` em vez de `(*struct).campo`), o caractere de ponto "`.`" também é usado nessa situação.

5.3 *Array*

Como grande parte das linguagens de programação, *Go* possui **`arrays`**. Sua utilização é bem simples:

```
var x [42]int  // Cria um array de int, com tamanho 42
```

Sua declaração é pouco usual em linguagens: o tamanho precede o tipo.

5.4 *Slice*

Um conceito peculiar a *Go* é o de **`slices`**, representando a forma que os desenvolvedores criaram para flexibilizar a utilização de `arrays`. Como o nome sugere, um `slice` cria um pedaço de tamanho dinâmico de um `array`.

5.4.1 Uso

Um *slice* pode ser criado a partir de um *array*.

```
func main() {
  primos := [6]int{2, 3, 5, 7, 11, 13}  // Array

  var slc []int = primos[1:4]  // Slice
  fmt.Println(slc)
}
```

O resultado impresso pelo trecho anterior é:

```
[3 5 7]
```

Um `slice` não guarda valor algum; ele apenas descreve uma seção de um `array`, funcionando como uma referência. Dessa forma, modificar elementos de um `slice` também altera os elementos do `array` ao qual ele faz referência.

```
func main() {
  primos := [6]int{2, 3, 5, 7, 11, 13}  // Array

  var slc []int = primos[1:4]  // Slice
  slc[2] = 42
  fmt.Println(slc)
}
```

A saída do código acima é:

```
[3 5 42]
```

Assim como `arrays`, é possível criar ***slices literais***, que são utilizadas da mesma forma:

```
// Cria um array, fazendo com que slc seja uma slice que o referência
slc := []int{2, 3, 5, 7, 11, 13}
```

5.4.2 Tamanho e capacidade

Um `slice` possui um *tamanho* e uma *capacidade*. O *tamanho* é a quantidade de elementos no `slice`; a *capacidade* é o tamanho do `array` ao qual ele faz referência.

```
slc := []int{2, 3, 5, 7, 11, 13}
len(slc)  // Retorna o tamanho de slc
cap(slc)  // Retorna a capacidade do array de slc
```

5.4.3 `make` e `append`

É possível criar `slices` com a função nativa **`make()`**. Esta é a forma de criar dinamicamente `arrays`:

```
a := make([]int, 5)  // Cria um array de int com tamanho 5
```

`Slices` podem conter qualquer tipo, incluindo outros `slices`. Com isso, podemos criar matrizes facilmente e acessá-las de forma semelhante ao que é feito em outras linguagens:

```
matriz := [][]int{[]int{1, 2, 3}, []int{4, 5, 6}, []int{7, 8, 9}}
matriz[1][1]  // Acessa o valor na posição (1, 1), neste caso 5
```

Para adicionar elementos a um `slice`, *Go* fornece a função nativa **`append()`**. Caso seja ultrapassada a capacidade do *array*, uma nova alocação e referência ocorrem, redimensionando-se a estrutura:

```
slc := make([]int, 5)
slc = append(slc, 5, 6, 7)  // Adiciona os valores 5, 6 e 7 ao slice
```

5.5 `range`

`range` funciona como um *iterator* de *Java*:

```
var pot = []int{1, 2, 4, 8, 16, 32, 64, 128}

func main() {
  // Itera por todos os elementos de pot
  for i, v := range pot {
    // i é o index no array; v é o valor
    fmt.Printf("2**%d = %d\n", i, v)
  }
}
```

5.6 *Map*

Um **`map`** é um *hash*, mapeando chaves para valores. Assim como `arrays` dinâmicas, é criado com a função **`make()`**:

```
// Cria um map onde chaves são strings e valores são inteiros
mapa := make(map[string]int)

// Adiciona um par (chave, valor) a mapa
mapa["Ramanujan"] = 1729
mapa["universo"] = 42

// Acessa o valor que possui a referida chave
fmt.Println(mapa["Ramanujan"])  // 1729
```

Além de adicionar elementos, podemos alterar um valor, deletar um par (chave, valor) e verificar se uma chave está presente em um conjunto.

```
// Altera o valor para a chave "Ramanujan"
mapa["Ramanujan"] = 1729

// Deleta a chave e o valor de "Ramanujan"
delete(mapa, "Ramanujan")

// ok será true caso "Ramanujan" esteja em mapa, ou false caso contrário
ok := mapa["Ramanujan"]
```

5.7 Funções como valores

Conforme comentado na seção de funções (seção 3.2), funções também podem ser valores em *Go*. Isto é, podem ser passadas como argumento para outras funções e até serem valor de retorno das mesmas:

```
1  /*
2   * Função que recebe como argumento uma função que possui dois
3   * valores float64 como argumento. Retorna um float64.
4   */
5  func calcula(fn func(float64, float64) float64) float64 {
6    return fn(3, 4)
7  }
8
9  func main() {
10   // hipot é uma variável que faz referência a uma função
11   hipot := func(x, y float64) float64 {
12     // Retorna uma chamada de função
13     return math.Sqrt(x*x + y*y)
14   }
15
16   // hipot é utilizada como uma função normal
17   fmt.Println(hipot(5, 12))
18
19   fmt.Println(calcula(hipot))
20   fmt.Println(calcula(math.Pow))
21 }
```

Na linha 19 do código anterior, `calcula()` recebe a função que `hipot` faz referência, e realiza uma chamada a ela com os valores 3 e 4. Esta função, por sua vez, retorna `math.Sqrt(x*x + y*y)`. Como `x = 3` e `y = 4`, o resultado impresso é **5**.

Já na linha 20, `math.Pow()` é chamado em `calcula()` com valores 3 e 4. Com isso, calcula-se 3^4, imprimindo-se **81**.

5.8 Funções como *closures*

Uma função em *Go* também pode ser uma ***closure***, ou seja, um valor na função que faz referência a um valor fora de seu escopo. A função pode acessar e alterar o valor referenciado:

```
1  /*
2   * A função somador retorna uma closure que possui um argumento int e
3   * retorna um int.
4   */
5  func somador() func(int) int {
6    // soma está na função somador, mas não está na função retornada func
7    soma := 0
8    return func(x int) int {
9      soma += x  // Pode acessar o valor de soma, pois é uma closure
```

```
    return soma
  }
}

func main() {
  // Cada closure tem sua própria variável soma
  pos, neg := somador(), somador()

  for i := 0; i < 10; i++ {
    fmt.Println(
      pos(i),     // Soma os valores de 0 até 10 = 45
      neg(-2*i),  // Soma os valores da forma -2*i = -90
    )
  }
}
```

6 Sofisticando tipos

Go não possui Orientação a Objetos. No entanto, algumas ferramentas da linguagem permitem sofisticar tipos e seus comportamentos, assunto que será abordado nesta seção.

6.1 Métodos

São definidos sobre tipos, servindo basicamente como uma função com um *receiver* como argumento. Funciona como açúcar sintático para atrelar uma função exclusivamente a um tipo. Segue padrões usuais, sendo acessado por um operador ".", e deve ter o tipo declarado dentro do mesmo pacote para funcionar. Também pode acessar tipos através de ponteiros, alterando variáveis por referência sem ter que copiar estruturas potencialmente grandes.

```
// Define um retângulo numa estrutura
type rect struct {
    largura, altura int
}

// Métodos para calcular área e perímetro
func (r *rect) area() int {
    return r.largura * r.altura
}
func (r rect) perim() int {
    return 2*r.largura + 2*r.altura
}

// Acesso aos métodos
r := rect{largura: 10, altura: 5}
rp := &r
area := rp.area()
perimetro := rp.perim()
```

6.2 Interfaces

Podemos definir **interface** como um conjunto de assinaturas de métodos, mas elas também são tipos e neste caso podem ter métodos com *receiver* `nil`, análogo ao `void*` de *C*.

```
// Conjunto de métodos
type geometria interface {
    area() float64
    perim() float64
}

// Estruturas que definem figuras geométricas
type rect struct {
    largura, altura float64
}
type circulo struct {
    raio float64
}

// Métodos
func (r rect) area() float64 {
    return r.largura * r.altura
```

```
}
func (r rect) perim() float64 {
    return 2*r.largura + 2*r.altura
}
func (c circulo) area() float64 {
    return math.Pi * c.raio * c.raio
}
func (c circulo) perim() float64 {
    return 2 * math.Pi * c.raio
}

// Chamadas genéricas
func calcula_medidas(g geometria) {
    fmt.Println(g)
    fmt.Println(g.area())
    fmt.Println(g.perim())
}

func main() {
    r := rect{largura: 3, altura: 4}
    c := circulo{raio: 5}
    calcula_medidas(r)
    calcula_medidas(c)
}
```

Uma interface vazia acontece quando é definido um tipo `interface` e o mesmo não possui métodos. Com isso, a interface aceita qualquer parâmetro, e o runtime de *Go* faz a conversão de tipo quando necessário – por isso a analogia ao `void*` de *C*. Combinado ao `switch`, pode-se fazer algumas construções interessantes.

```
func faz(i interface{}) {
  switch v := i.(type) {
  case int:
    fmt.Printf("Dobro de %v é %v\n", v, v*2)
  case string:
    fmt.Printf("%q tem %v bytes\n", v, len(v))
  default:
    fmt.Printf("Não reconheço o tipo %T!\n", v)
  }
}

func main() {
  faz(21)      // Dobro de 21 é 42
  faz("hello") // "hello" tem 5 bytes
  faz(true)    // Não reconheço o tipo bool!
}
```

7 Concorrência

Conforme mencionado na introdução (seção 1), a linguagem *Go* possui recursos poderosos e de simples uso para implementação de concorrência. Nesta seção, veremos algumas dessas ferramentas, bem como exemplos de seu uso.

7.1 *Goroutines*

Uma ***goroutine*** é uma *thread* leve, gerenciada pelo *runtime* do *Go*. Por ser uma tarefa mínima e com diversas primitivas de comunicação, pode ser criada de maneira liberal, sendo multiplexada nos *threads* do sistema operacional e sem a preocupação do *overhead* prejudicar o desempenho[22], como ocorre quando há excesso de *threads* em execução concorrente.

Uma *goroutine* é criada ao usarmos a palavra reservada **go**, passando uma chamada de função como argumento. A *goroutine* atual avaliará a função e seus parâmetros, mas a execução da mesma ocorre na nova *thread*.

Abaixo, mostramos um exemplo de uso dessa funcionalidade para escrever um servidor *daytime* (isto é, um servidor que devolve sua data e horário aos clientes que nele se conectarem). O código foi baseado em uma implementação já existente.

```
1 import "net"   // Contém funções para sockets
2 import "time"  // Usada para adquirir horário do servidor
3
4 func main() {
5   // Cria socket de escuta
```

```
6   escuta, err := net.Listen("tcp", "127.0.0.1:13")
7   defer escuta.Close()
8
9   for {
10    // Cria nova goroutine para cada conexão aceita
11    conn, err := escuta.Accept()
12    go gerenciaConexao(conn)
13  }
14 }
```

Primeiro, criamos um *socket* de escuta através da função `net.Listen()`. Esta recebe o protocolo da camada de transporte a ser usado (TCP) e em qual IP/porta queremos esperar por conexões (porta 13 para servidores *daytime*). Devolve-se o *socket* de escuta e um erro, que não trataremos aqui para simplificar o exemplo. Note, também, o uso de `defer` para automaticamente fechar o *socket* após o fim da função `main()`.

Em seguida, a partir da linha 9, temos o laço infinito do *socket* de escuta. Se uma conexão com um cliente for estabelecida, criamos um novo *socket* com `escuta.Accept()`, e tratamos tal cliente em uma nova *goroutine*, chamando `gerenciaConexao()`. O código desta função é bem simples no servidor *daytime*:

```
1 func gerenciaConexao(conn net.Conn) {
2   // Formata data atual para uma string
3   t := time.Now().Format(time.RFC3339)
4
5   // Envia data ao cliente e encerra conexão
6   conn.Write([]byte(t))
7   conn.Close()
8 }
```

O uso de *threads* para a simples tarefa de obter e devolver a data atual pode soar um pouco exagerado, pois o próprio `main()` poderia rapidamente tratar disso. De toda forma, deve permanecer a ideia de que *goroutines* poderiam ser facilmente usadas na criação de servidores bem mais complexos.

7.2 *Channels*

Channels são condutos tipados, tipicamente usados para comunicação entre *goroutines*.

Internamente, são implementados com filas, onde cada célula contém um ponteiro para sua *goroutine* e o tipo de dado que o *channel* trata. Também usa-se uma estrutura de exclusão mútua para controlar o fluxo, podendo ser um *mutex* ou um semáforo, a depender do Sistema Operacional [23]. O código de implementação dos *channels*, disponível no GitHub do *Go* [24], deixa clara a estrutura de dados utilizada na implementação.

```
type hchan struct {
  qcount   uint           // total data in the queue
  dataqsiz uint           // size of the circular queue
  buf      unsafe.Pointer // points to an array of dataqsiz elements
  elemsize uint16
  closed   uint32
  elemtype *_type // element type
  sendx    uint   // send index
  recvx    uint   // receive index
  recvq    waitq  // list of recv waiters
  sendq    waitq  // list of send waiters

  // lock protects all fields in hchan, as well as several
  // fields in sudogs blocked on this channel.
  //
  // Do not change another G's status while holding this lock
  // (in particular, do not ready a G), as this can deadlock
  // with stack shrinking.
  lock mutex
}
```

7.2.1 Criação e uso

O primeiro passo para usar um *channel* passa por sua criação. Assim como *maps*, são gerados pelo comando `make`. Abaixo, criamos um *channel* (tipo **chan**) que trabalha com o tipo `int`.

```
ch := make(chan int)
```

Feito isso, basta usar o operador **<-** para obter ou enviar valores pelo *channel*. A direção da seta indica para onde os dados fluem:

```
ch <- var    // Envia var para o channel ch
var := <-ch  // Recebe de ch, e guarda o valor na nova variável var
```

É importante destacar que *channels* **bloqueiam** até o outro lado estar pronto para receber/enviar algo. Tal procedimento facilita a sincronização de *goroutines* sem que variáveis adicionais (*mutex* ou *condicionais*, por exemplo) sejam necessárias.

7.2.2 *Channels* com *buffer*

É possível criar um *buffer* para um *channel*, isto é, especificar quantos elementos do tipo do *channel* podem ser armazenados. Para isso, especificamos o tamanho do *buffer* no segundo argumento do `make`:

```
// Buffer armazena 10 ints
ch := make(chan int, 10)
```

A grande diferença de *channels* com *buffer* é que envios bloqueiam se o *buffer* estiver cheio, e recepções bloqueiam se o *buffer* estiver vazio. Exemplificando, veja os três casos abaixo:

```
ch := make(chan int, 2)
ch <- 1
ch <- 2
fmt.Println(<-ch)
fmt.Println(<-ch)
```

```
ch := make(chan int, 2)
ch <- 1
ch <- 2
ch <- 3 // Cheio!
fmt.Println(<-ch)
fmt.Println(<-ch)
```

```
ch := make(chan int, 2)
ch <- 1
ch <- 2
fmt.Println(<-ch)
fmt.Println(<-ch)
fmt.Println(<-ch) // Vazio!
```

Nos três casos, cria-se um *channel* com um *buffer* para 2 inteiros. O trecho de código da esquerda é o único que funciona, pois não insere mais elementos que a capacidade do *buffer* (como no código do meio), nem tenta receber elementos de um *buffer* vazio (como no código da direita).

Se tentarmos executar um dos trechos de código com problema, o programa é interrompido e gera um erro de *deadlock*:

```
fatal error: all goroutines are asleep - deadlock!
```

7.2.3 `range` e `close`

Um problema recorrente em *channels* envolve receber dados repetidamente. A forma de fazer isso em *Go* é criar um `for` e iterar sobre o *channel* com a palavra reservada `range`:

```
for i := range ch {
  // Faz algo com a variável i recebida
}
```

Se não houver nenhum dado a ser recebido, o laço bloqueia até chegar algum valor.

Quando não há mais dados para enviar, é prática comum que o emissor feche o *channel* com a função `close()`. Esta chamada leva ao término do laço acima no receptor.

No exemplo abaixo, criamos uma *goroutine* para calcular e enviar números de Fibonacci para a *goroutine* principal.

```
1  func fibonacci(n int, ch chan int) {
2    x, y := 0, 1
3    for i := 0; i < n; i++ {
4      ch <- x
5      x, y = y, x+y
6    }
7    close(ch)  // Cálculo de Fibonacci acabou; fecha o channel
8  }
9
10 func main() {
11   ch := make(chan int, 10)
12   go fibonacci(cap(ch), ch)  // cap() devolve o tamanho do buffer de um channel
13   for i := range ch {
14     fmt.Println(i)
15   }
16 }
```

7.2.4 `select`

Go oferece uma ferramenta poderosa para tratar vários *channels* simultaneamente: a palavra reservada `select`. Sua sintaxe é idêntica à do `switch`, com cada `case` representando uma tentativa de enviar ou receber algum dado. Se o *channel* estiver bloqueado, o respectivo `case` é pulado.

Assim como no `switch`, `select` também possui o `default` para quando todos os `case` são ignorados. A diferença é que o código deste bloco é executado se todos os outros *channels* estiverem bloqueados.

```
x := 0
for {
  select {
    case ch <- x:
      // Consigo enviar x para ch
      x++
    case <-ch_tchau:
      // Recebi algo de ch_tchau
      fmt.Println("É hora de dar tchau!")
      return
    default:
      fmt.Println("Todos os channels estão bloqueados!")
  }
}
```

7.3 *Mutex*

Apesar de não ser uma estrutura muito recomendada do ponto de vista da filosofia *"Share memory by communicating"* de *Go*, há momentos onde queremos que apenas uma *goroutine* acesse uma certa variável por vez para proteger regiões críticas. Para isso, usamos uma estrutura que garante *exclusão mútua*: o *mutex*.

Em *Go*, o pacote `sync` contém a estrutura `Mutex`, que possui dois métodos importantes: `Lock` e `Unlock`. Um trecho de código cercado por um `Lock()/Unlock()` será executado em exclusão mútua.

```
import "sync"

var mutex sync.Mutex

// Na goroutine...
mutex.Lock()
// Seção crítica!
mutex.Unlock()
```

8 Conclusão

Devido ao foco em servidores, seria imaginável que *Go* teria um grande poder de desempenho, mas isso não ocorre e *C/C++* seguem com a liderança neste quesito.

Um dos criadores de *Go*, Rob Pike, defende que a motivação da linguagem é utilizar o que a tecnologia fornece atualmente – concorrência e paralelismo – e maximizar o desempenho do programador[25]. Ao priorizar simplicidade de código e velocidade de compilação, Pike defende a minimização do esforço do programador em contraponto a minimizar os ciclos de CPU.

A comunidade também argumenta que, por ser uma linguagem recente, não recebeu tantas otimizações como outras mais tradicionais. No entanto, fica claro que funcionalidades complexas (como o Coletor de Lixo) e limitações ao acesso de baixo nível ao programador talvez façam com que *Go* nunca seja o expoente em desempenho. Se essa perda for compensada com aumento de produtividade, os criadores da linguagem possivelmente terão atingido seu objetivo.

Com suporte de um gigante da computação (Google) no seu desenvolvimento e crescente adoção no mercado[26], a linguagem *Go* provavelmente será mantida e atualizada por muito tempo. Anualmente existe a GopherCon[27], uma conferência que tem como foco a linguagem, com palestras, eventos e *"hackathons"* focadas em aprofundar o conhecimento dos interessados, além de debater o futuro da mesma.

É importante citar que muitos aspectos de *Go* não foram abordados nesta monografia. Caso o leitor queira se aprofundar no assunto, um tópico interessante para se citar é suporte a testes automatizados na linguagem[28], assunto importante na área de engenharia de software e na metodologia ágil.

Acreditamos que esta monografia dá uma visão geral sobre a linguagem e a intenção de seus criadores, além de oferecer caminhos aos leitores interessados no tema. De modo geral, temos a percepção de que *Go* pode ser usado para fins específicos, notoriamente a criação de servidores e para tarefas envolvendo computação paralela e concorrente.

Referências

[1] Computer Hope. "Robert Griesemer" (2017). URL: https://www.computerhope.com/people/robert_griesemer.htm.

[2] "Rob Pike" (dez. de 2016). URL: https://en.wikipedia.org/wiki/Rob_Pike.

[3] "Ken Thompson" (jun. de 2017). URL: https://en.wikipedia.org/wiki/Ken_Thompson.

[4] Ahlemeier, Cunjuca, Hinojos, Fortuno. "Google Go" (2013). URL: http://cse360fall13.wikispaces.asu.edu/Google+Go.

[5] Rob Pike. "Go at Google: Language Design in the Service of Software Engineering" (2012). URL: https://talks.golang.org/2012/splash.article.

[6] "Go (programming language)" (jun. de 2017). URL: https://en.wikipedia.org/wiki/Go_(programming_language).

[7] C. A. R. Hoare. *Communicating Sequential Processes*. 2015. URL: http://www.usingcsp.com/cspbook.pdf.

[8] Andrew Gerrand. "Share Memory By Communicating" (jul. de 2010). URL: https://blog.golang.org/share-memory-by-communicating.

[9] *The Go Programming Language (Go Gopher image)*. 2015. URL: https://golang.org/doc/gopher.

[10] *The Go Programming Language*. Mar. de 2017. URL: https://golang.org.

[11] "Go 1.8 Release Notes" (2017). URL: https://golang.org/doc/go1.8.

[12] Ilya Pavlenkov. *For which purpose is the Go programming language used?* Jul. de 2014. URL: https://www.quora.com/For-which-purpose-is-the-Go-programming-language-used.

[13] *The Computer Language Benchmarks Game*. 2017. URL: http://benchmarksgame.alioth.debian.org.

[14] Robert Love. *How is Go used at Google?* Nov. de 2013. URL: https://www.quora.com/How-is-Go-used-at-Google.

[15] Artelius, zneak. *What language is the Go programming language written in?* 2015. URL: https://stackoverflow.com/questions/3327676/what-language-is-the-go-programming-language-written-in.

[16] el.pescado, Larry OBrien. *How does Go compile so quickly?* 2013. URL: https://stackoverflow.com/questions/2976630/how-does-go-compile-so-quickly.

[17] The LLVM Project. *LLVM-based compiler for Go*. 2017. URL: https://github.com/go-llvm/llgo.

[18] Golang.org. *runtime2.go*. 2017. URL: https://github.com/golang/go/blob/master/src/runtime/runtime2.go.

[19] "A Tour of Go" (). URL: https://tour.golang.org/list.

[20] Rob Pike. "Go's Declaration Syntax" (jul. de 2010). URL: https://blog.golang.org/gos-declaration-syntax.

[21] "The Go Programming Language: Frequently Asked Questions (FAQ)" (2017). URL: https://golang.org/doc/faq#no_pointer_arithmetic.

[22] Traun Leyden. *Goroutines vs Threads*. 2014. URL: http://tleyden.github.io/blog/2014/10/30/goroutines-vs-threads/.

[23] mna, Matt. *How are Go channels implemented?* 2015. URL: https://stackoverflow.com/questions/19621149/how-are-go-channels-implemented.

[24] Golang.org. *chan.go*. 2017. URL: https://github.com/golang/go/blob/master/src/runtime/chan.go.

[25] Rob Pike. *Less is exponentially more*. 2012. URL: https://commandcenter.blogspot.com.br/2012/06/less-is-exponentially-more.html.

[26] Golang.org. *Cases do Go*. 2017. URL: https://github.com/golang/go/wiki/SuccessStories.

[27] Gopher Academy. *GopherCon*. 2017. URL: https://www.gophercon.com/.

[28] Golang.org. *Testing*. 2017. URL: https://golang.org/pkg/testing/.