PONTIFÍCIA UNIVERSIDADE CATÓLICA DE MINAS GERAIS
Instituto de Ciências Exatas e de Informática

Trabalho de Linguagem de Programação - Linguagem Go*

Gabriel Luciano Gomes[1]
Geovane Fonseca de Sousa Santos[2]
Isabelle Hirle Alves Langkammer[3]
Luigi Domenico Cecchini Soares[4]

*Artigo apresentado ao Instituto de Ciências Exatas e Informática da Pontifícia Universidade Católica de Minas Gerais como pré-requisito para obtenção do título de Bacharel em Ciência da Computação.

[1]Aluno, Ciência da Computação, Brasil, glgomes@sga.pucminas.br.
[2]Aluno, , Ciência da Computação, Brasil, , geovane.fonseca@sga.pucminas.br.
[3]Aluno, , Ciência da Computação, Brasil, , isabelle.langkammer@sga.pucminas.br.
[4]Aluno, , Ciência da Computação, Brasil, , luigi.domenico@sga.pucminas.br.

Sumário

Lista de Figuras

1 INTRODUÇÃO

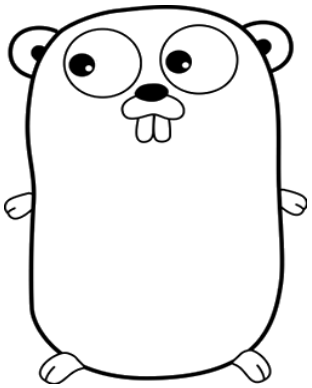
A linguagem Go tem se tornado cada vez mais popular, estando entre as melhores linguagens de programação e uma das mais usadas pelo mundo. Dentre outras características marcantes, ela beneficia os desenvolvedores pelo seu domínio de operações concorrentes e a simplicidade da construção.

Conhecida como a linguagem da Google, foi desenvolvida pelos engenheiros da empresa para solucionar problemas de desenvolvimento de *software*. A linguagem foi lançada com o nome Go!, mas, poucos dias após o lançamento, Fancis McCabe solicitou a mudança do nome. Esta mudaçna ocorreu, pois, em 2003, ele havia criado uma linguagem com o mesmo nome, porém não a registrou. Go é conhecida como uma linguagem concisa, em que problemas complexos são resolvidos em poucas linhas de código, tendo como propósito aumentar a produtividade em projetos.

O mascote de Go é chamado de Go Gopher. A ideia por trás do mascote surgiu quando pediram para Renee French desenhar o logotipo e, para o lançamento da linguagem, ela sugeriu adpatar o gopher WFMU, que foi criado para um evento beneficiente há mais de 15 anos. O mascote representa os programadores de Go e a linguagem, sendo bem popular. A figura 1 mostra tanto o mascote quanto o logotipo atual referente a linguagem.

Figura 1 – Mascote e Logotipo atual

(a) Mascote Go



Fonte: Golang

(b) Logotipo Go

Fonte: blog.golang.org

2 HISTÓRICO

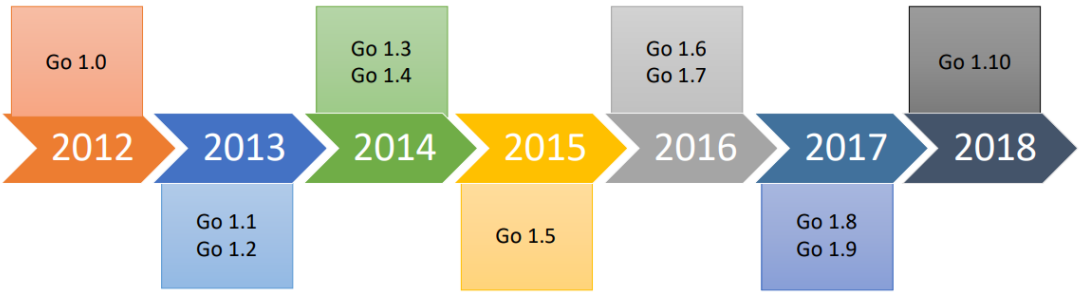
Go começou a ser desenvolvida em tempo parcial em 2007 por Robert Griesemer, Rob Pike e Ken Thompson, engenheiros da google, em 2008 o desenvolvimento passou a ser um projeto em tempo integral do Google e no ano seguinte se tornou Open source e logo depois começou a ser adotada por desenvolvedores fora da google.

A linguagem nasceu como resposta aos problemas de desenvolvimento de software rela-

cionados com o uso de sistemas distribuídos em redes, computação web, computação massiva e processadores multicore se tornando mainstream. A construção de códigos extensos e complexos faz com que possuam um grande número de dependências, resultando em uma compilação demorada. Tentaram resolver o problema com outras linguagens como C++, Java e Python, mas apesar de terem os requisitos desejáveis, nenhuma se mostrou totalmente eficiente. O objetivo de Go é ser uma linguagem em que os códigos sejam eficientes e desenvolvidos rapidamente.

A primeira versão de Go foi lançada em 2012, a última versão lançada foi a 1.10 em fevereiro de 2018. Na figura a seguir tem o histórico com a data de lançamento de cada versão.

Figura 2 – Histórico de versão da linguagem Go



2.1 Influência

Go é Influenciada pelas linguagens Limbo, Oberon, C, Alef, Pascal, Modula e Python. Possui uma sintaxe semelhante a C e é tão eficiente quanto. Possui recursos de linguagens dinâmicas como Python . Uma das características da linguagem são a simplicidade, ou seja, Go apresenta uma sintaxe limpa e concisa enquanto linguagens como C++ e Java os códigos ficam enormes e volumosos. Possui recursos como coleta de lixo e reflexão, mas não é uma linguagem orientada a objetos pois não é uma solução eficaz para os problemas e os criadores queriam ter uma abordagem melhor para programas extensíveis. Suporta o “Duck typing” que é quando uma struct pode implementar uma interface automaticamente.

3 PARADIGMAS

Go é uma linguagem de programação que adota o paradigma imperativo em seu conceito. Para exemplificar esse conceito, pode-se usar como modelo as linguagens naturais, como o português, que apresentam frases imperativas, nas quais tem-se a ideia de ordem, ação ou obrigação. Tendo em vista isso, podemos associar esse conceito a computação como sendo vista como ações, enunciados ou comandos que mudam o estado (variável) de um programa.

```
1  var i int = 10
2
3  // Comando de atribuicao
4  i = 5
5
6  // Loop
7  for j := 0; j < 10; j++ {
8      i += 1
9      ...
10 }
```

4 CARACTERÍSTICAS

4.1 Sintaxe e Semântica

A linguagem Go é uma nova linguagem de programação que ainda está em fase de desenvolvimento, para criação de compiladores, Servidores Web, aplicações de propósito geral, como arquivos XML e processamento de dados. Contudo, por enquanto, nenhuma aplicação a utiliza. Junto a isso, a sintaxe e a sêmantica do Go foram criadas como um híbrido de diversas linguagens famosas, dentre elas as linguagens C e Python. Como essas duas compõem estruturas bastante intuitivas e simples, Go herdou essas características delas.

Além disso, em contraposição a linguagens *C-like*, que são de grande volume, Go é uma linguagem "limpa" e apresenta uma sintaxe concisa. Ela utiliza uma inferência implícita de tipos, amarrando o tipo a um identificador de acordo com o primeiro valor atribuído a ele. Quando for necessário realizar amarrações, a sintaxe de declaração é simples e diferente de C. A sintaxe é descrita pela gramática de Backus-Naur Estendido (família de notações meta-sintaxe qualquer que pode ser usada para expressar uma gramática livre de contexto).

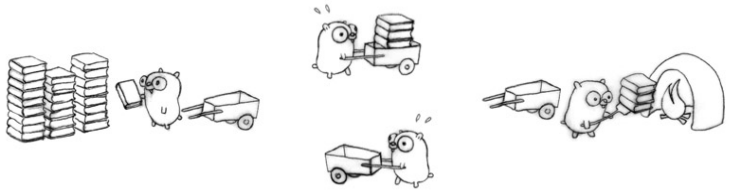
4.2 Concorrência:

Go é uma linguagem concorrente, pois suporta canais de comunicação, baseados em Processos Sequenciais de Comunicação de Hoare (CSP), que é uma linguagem formal para descrever padrões de interação em sistemas concorrentes. Além disso, CSP é um membro da família de teorias matemáticas, conhecido como Álgebra de Processos. Tal suporte a concorrência é diferente da programação de aproximação Lock-based, encontrada em pthreads, Java locks e outros.

4.2.1 Concorrência x Paralelismo

A concorrência explora a possibilidade de resolver vários problemas ao mesmo tempo. Isso se dá a medida que, neste paradigma, é visado a multiplicidade de tarefas que um sistema pode executar. Por exemplo, em um processo de incineramento de arquivos, podemos desempilhar arquivos, transportá-los para o incinerador e então queimá-los. Cada uma dessas ações são tarefas distintas e, não necessariamente, precisam ser executadas simultaneamente. Supondo que cada uma dessas funções é executada por agentes distintos, ao adicionar mais funcionários, mais trabalho poderá ser executado, mas cada um com sua devida responsabilidade e tempo de execução. Esta é a ideia que a concorrência abstrai: vários trabalhos a serem executados de forma concorrente, mas não necessariamente simultâneos.

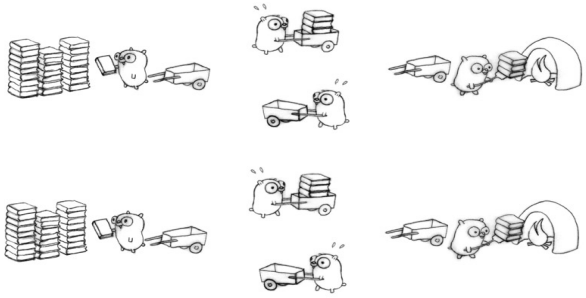
Figura 3 – Modelo de Concorrência de tarefas



Fonte: The Go Blog
Disponível em: <https://talks.golang.org/2012/waza.slide#19>;. Acesso em maio. 2018.

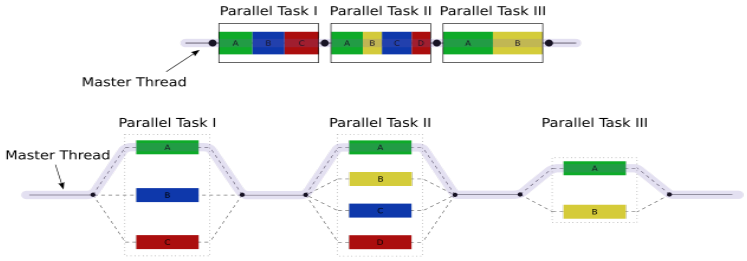
Tomando como base o conceito de uma estrutura concorrente, o paralelismo explora a ideia de que pelo menos duas tarefas estão sendo executadas ao mesmo tempo. É com essa aplicação que a concorrência possibilita a simultaneidade, uma vez que n processos podem ser executados ao mesmo tempo. Além disso, a paralelização pode explorar também a divisão de um único problema em frações distintas, com o objetivo de ter uma execução mais rápida, otimizando o processamento. As imagens abaixo exemplificam a ideia de paralelização de processos e de uma tarefa.

Figura 4 – Modelo de paralelização de tarefas



Fonte: The Go Blog
Disponível em: <https://talks.golang.org/2012/waza.slide#16>;. Acesso em maio. 2018.

Figura 5 – Modelo de paralelização de um problema



Fonte: Wikipedia - OpenMP
Disponível em: <https://en.wikipedia.org/wiki/OpenMP>;. Acesso em maio. 2018.

4.3 *Goroutines*

São co-rotinas semelhantes as *threads* implementadas por outras linguagens, como Java. Porém, as *goroutines* são muito mais leves que as *threads* comuns. Outro ponto importante é que elas são executadas no mesmo espaço de enderaçamento e, portanto, acessos a memória compartilhada devem ser sincronizados.

```
1  func f(from string) {
2      for i := 0; i < 3; i++ {
3          fmt.Println(from, ":", i)
4          time.Sleep(100 * time.Millisecond)
5      }
6  }
7
8  func main() {
9      // Go: utilizado para iniciar uma goroutine
10     go f("goroutine")
11     f("main")
12 }
```

4.4 Comunicação

4.4.1 Channels

Go implementa vias de comunicação entre processos que usufruem de informações compartilhadas, por meio de canais tipados. Estes permitem a conversa entre *goroutines*, ao possiblitar o envio e a recepção de valores. Por padrão, envios e recebimentos causam bloqueios até que o outro lado da comunicação esteja preparado. Tal funcionamento permite que as *gorouti-nes* sejam sincronizadas, sem que seja necessário bloqueios explícitos.

```
1  // Criando um canal do tipo float.
2  ch := make(chan float32)
3
4  // Criando um canal do tipo inteiro, com tamanho 10.
5  // Conhecido como buffered channel.
6  buffered_ch := make(chan int, 10)
7
8  // Enviando v para o canal ch.
9  ch <- v
10
11 // Recebendo um valor do canal ch e inicializa v com esse valor.
12 v := <-ch
13
14 // Fechando um canal.
15 close(ch)
16
17 // Verificando se canal foi fechado.
18 // Open recebe o valor false, caso o canal tenha sido fechado.
19 v, open := <-ch
```

4.4.2 Mutex

Channels são ótimos para comunicação entre *goroutines*. Mas, e se essa troca de informações não se faz necessária? E se é preciso apenas garantir que somente uma *goroutine* por vez irá acessar uma variável, para evitar conflitos? Esse conceito é chamado de Exclusão Mútua, e o nome convencional para a estrutura de dados que provê essa exclusão é *mutex* (*Mutual Exclusion*).

```
1  // SafeCounter is safe to use concurrently.
2  type SafeCounter struct {
3      v   map[string]int
4      mux sync.Mutex
5  }
6
7  ...
8  var counter = SafeCounter{v: make(map[string]int)}
9  // Bloqueando o contador para uma goroutine utilizar
10 counter.mux.Lock()
11
12 // Liberando o contador para uma outra goroutine utilizar
13 counter.mux.Unlock()
```

4.4.3 Select

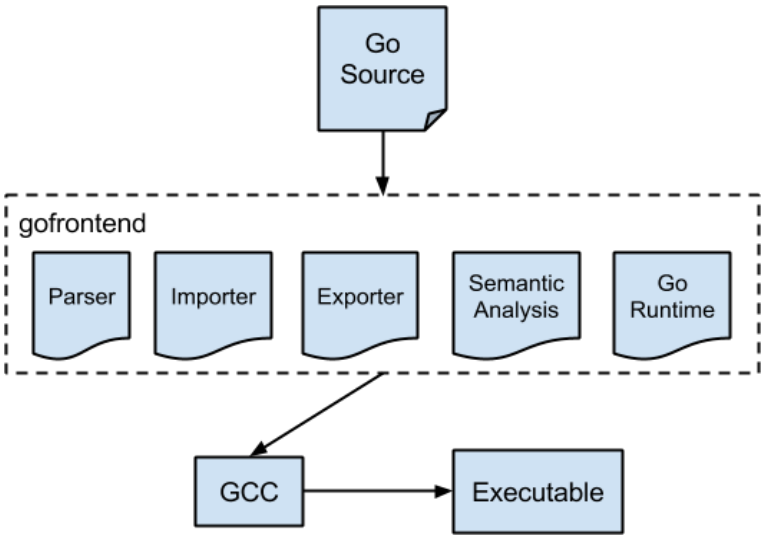
Comando de bloco que permite uma *goroutine* esperar por várias operações de comunicação (realizadas através dos *channels*). Enquanto nenhum dos casos especificados no *select* estiverem prontos para serem executados, a *goroutine* é bloqueada. Tal bloqueio pode ser transpassado ao se utilizar um caso como *default*, que será executado caso nenhum dos outros casos estejam prontos.

```
1  func fibonacci(c, quit chan int) {
2      x, y := 0, 1
3      for {
4          select {
5          case c <- x:
6              x, y = y, x+y
7          case <-quit:
8              fmt.Println("quit")
9              return
10         }
11     }
12 }
13
14 func main() {
15     c := make(chan int)
16     quit := make(chan int)
17     go func() {
18         for i := 0; i < 10; i++ {
19             fmt.Println(<-c)
20         }
21         quit <- 0
22     }()
23     fibonacci(c, quit)
24 }
```

4.5 Compilador

Utiliza como compilador o gccgo para gerar um arquivo executável. O compilador é responsável por fazer o parse do código fonte, importar, exportar, analisar a semântica e executar o runtime do Go seguindo o seguinde esquema.

Figura 6 – Modelo de compilação - Linguagem Go



Fonte: Golang/Project

4.6 Duck typing (Tipagem “pato”)

Duck typing é a possibilidade da estrutura implementar uma interface automaticamente. Esse é um estilo de linguagem em que os comportamentos, métodos e propriedades de um objeto determinam a semântica válida, em vez de sua herança ou uma interface explícita. Essa técnica é poderosa e é um novo recurso implementado na linguagem Go.

4.7 Recursos modernos

Apesar de ser uma linguagem de programação para sistemas, Go suporta recursos modernos como coleta de lixo, reflexão (Métodos e/ou atributos que serão utilizados de uma outra classe), dentre outros. A reflexão é um recurso que devemos ter cuidado ao utiliza-lo, pois:

- A reflexão apresenta alguns riscos ao utilizar sua implementação. A redução de desempenho pode ocorrer quando o código em utilização está sendo requerido múltiplas vezes com alta frequência, aumentando a latência. Além disso, há possibilidade de restrição de segurança, uma vez que pode ser executado em outros ambientes que tenham regras específicas de seguranças, dificultando um acesso geral à esta. E, por fim, pode expor a estrutura interna dos objetos, pois com esse recurso, temos acesso a todos os métodos e atributos ofertadas pelo objeto em questão.
- Por outro lado, a reflexão também apresenta benefícios quando utilizamos de forma correta e adequada esse recurso. Pode facilitar a manutenção, pois a responsabilidade é coesa; minimiza a geração de erros, já que o código é modificado apenas em sua classe origem e não nos que o utilizam; há um ganho de produtividade; gera uma padroniza-

ção, uma vez que tais processos evitam geração de erros provenientes de manutenções; extensibilidade, a aplicação pode ser utilizada em vários outros programas, ampliando a utilização e o reaproveitamento do mesmo.

4.8 Arquivos binários

A linguagem Go gera arquivos binários para sua aplicação com todas as aplicações embutidas (built-in). Isso remove a necessidade de instalação de outros softwares para execução da aplicação. Isso gera um "alívio" nas tarefas de desenvolvimento das aplicações e providencia atualizações necessárias dentre outra milhares de instalações. Além de que os arquivos gerados são compatíveis com múltiplos sistemas operacionais, o que gera um ganho enorme para a linguagem.

4.9 Design da Linguagem

Algumas decisões de estruturas foram feitas frente-a-frente com o paradigma Orientado a Objetos, que mantem os recursos limitados e curtos. Em função disso, a linguagem é opinativa e recomenda várias maneiras de produção das coisas desejadas. É preferida a utilização de composições do que heranças, e seu sistema de tipos é elegante e permite um comportamento a ser adicionado sem a realização de casamentos fortes em componentes.

4.10 Biblioteca Padrão

Go utiliza uma biblioteca padrão poderosa, que é distribuída em pacotes. Essa biblioteca possui a maioria dos recursos que os desenvolvedores esperam ao migrar para uma outra linguagem. Além disso, há diversos suportes em fóruns e blogs da Google, junto a páginas de usuários da linguagem na internet, que auxiliam os programadores a explorar o máximo de sua capacidade.

4.11 Gerência de Pacotes

A linguagem Go combina ambientes de trabalhos de novos desenvolvedores, por meio de projetos open source, e inclui as maneiras de trabalhar com os pacotes externos. Permite também a publicação de pacotes próprios, com comandos fáceis, para incorporação em sua biblioteca padrão. Além disso, todos os programas, em Go, tem como início de sua execução o pacote *main* e a função *main*.

```
1 package main
2 import "fmt"
3
4 func main() {
5     fmt.Println("Hello World!")
6 }
```

4.11.1 Exportação de nomes

Um nome declarado dentro de um determinado pacote é exportado se, e somente se, esse identificador for inicado com letra maíuscula. Caso contrário, o nome será privado ao pacote em que está contido. Sendo assim, referenciar uma estrutura que não é exportada resulta em erros durante a compilação.

```
1 package main
2 import (
3     "fmt"
4     "math"
5 )
6
7 func main() {
8     fmt.Println(math.pi)
9 }
```

ERRO: cannot refer to unexported name math.pi
ERRO: undefined: math.pi

5 ELEMENTOS DA LINGUAGEM

5.1 Erros por não utilização

Em go, variáveis e/ou importações que não estejam sendo utilizadas geram erros em tempo de compilação.

```
1 package main
2
3 import "fmt"
4 func main() {
5     v := 10
6 }
```

ERRO: imported and not used: "fmt"
ERRO: v declared and not used

5.2 Valores e Tipos:

Um valor diz respeito a qualquer entidade que exista durante uma computação. Em outras palavras, tudo que possa ser avaliado, armazenado, passado como parâmetro para uma função ou retornado por uma função. Pensando em uma melhor organização em relação aos estudos dos valores, costuma-se agrupá-los em tipos.

Tipos são especificações das classes de valores que podem ser associados à variável. Além disso, dizem respeito as operação que podem ser usadas para criar, acessar e modificar esses valores. No geral, linguagens de programação oferecem um conjunto de tipos primitivos, além de mecanimos para se estruturar tipos compostos.

5.2.1 Tipos primitivos

- Inteiros

```
1  // Int (32 ou 64 bits, de acordo com a arquitetura)
2  var i int
3  i = 10
4
5  // Variantes do tipo inteiro, com sinal (8, 16, 32, 64)
6  var i8 int8
7  i8 = -2
8
9  // Variantes do tipo inteiro, sem sinal (8, 16, 32, 64)
10 var u8 uint8
11 u8 = 2
```

- Reais

```
1  // Float
2  var f float32
3  f = 1.5
4
5  // Double
6  var d float64
7  d = 3.343343
```

- *String*

```
1  var str string
2  str = "Hello world!"
```

- *Boolean*

```
var t, f bool
t, f = true, false
```

- Complexos

```
import "math/cmplx"
...
var z complex128 = cmplx.Sqrt(-5 + 12i)
```

5.2.2 *Tipos Compostos*

- *Arrays*
  São mapeamentos finitos entre um conjunto de índices e um outro conjunto finito, de valores de um tipo T qualquer. Em Go, arranjos suportam apenas o mapeamento entre índices e valores de um mesmo tipo.

```
primes := [6]int{2, 3, 5, 7, 11, 13}
```

- *Slices*
  São como uma espécie de referência para um *array*. Sendo assim, não armazenam nenhum valor, apenas descrevem a seção de um *array*. Realizar modificações em uma *slice* resulta em alterações nos elementos do *array* correspondente.

```
primes := [6]int{2, 3, 5, 7, 11, 13} // Array
var s []int = primes[1:4] // Slice - posicoes 1 a 4
                          // (1 inclusivo, 4 exclusivo)
```

- *Structs*
  Representam um produto cartesiano de n conjuntos S_1, S_2, S_3, ... S_n, denotado por $S_1 X S_2 X S_3 X ... X S_n$, em que seus elementos são n-tuplas ordenadas $(s_1, s_2, s_3, ...s_n)$, onde $s_i \in S_i$.

```
type Vertex struct {
    X, Y float64
}
```

- *Maps*
  Assim como os *arrays*, consistem em uma função de mapeamento entre dois conjuntos finitos de valores S em valores de um tipo T. Em outras palavras, o tipo *Map*, em Go,

mapeia chaves de um tipo T_1 a valores de um tipo T_2, podendo T_1 e T_2 serem diferentes.

```
var m map[string]Vertex
m["Bell Labs"] = Vertex{
    40.68433, -74.39967,
}
```

5.2.3 *Checagem de Tipos*

Go é uma linguagem estaticamente tipada. Isto é, os tipos associados a cada identificador são verificados em tempo de compilação. Além disso, a linguagem possui como característica uma tipagem forte, uma vez que erros de tipos são sempre detectados. O exemplo a seguir apresenta um erro decorrente de uma tentativa de atribuir um valor de um tipo diferente do que está associado ao identificador.

```
func main(){
    var nome string = "nome"
    nome = 10
}
```

ERRO: cannot convert 10 to type string

5.2.4 *Equivalência de Tipos*

Go dá suporte apenas à equivalência nominal, deixando de lado qualquer implementação relacionada a checagem de tipos a partir de suas estruturas. Na linguagem em questão, dois tipos com estruturas semelhantes, mas com nomes diferentes são considerados como distintos. Para exemplificar, o exemplo a seguir mostra o erro causado por uma tentativa de utilizar um método associado ao tipo distinto do declarado.

```
package main

type Peca struct {cor int, dama bool}
type Tupla struct {tamanho int, comida bool}

func (peca *Peca) ehDama () (bool){
    return peca.dama
}

func main(){
    var tupla *Tupla = new Tupla
```

```
12      tupla.ehDama()
13  }
```

ERRO: tupla.ehDama undefined

5.2.5 *Inferência de tipos*

Variáveis podem ser declaradas sem que seu tipo apareça de forma explícita. Nesse caso, o tipo a ser amarrado a entidade será inferido a partir do valor apresentado ao lado direito do comando de atribuição.

- Exemplo de declaração explícita

```
1 var i, j int
2 i, j = 10, 12
```

- Exemplo de declaração implícita

```
1 var i, s, b = 5, "Calculo lambda", true
```

5.3 Estruturas de repetição

Go não implementa as estruturas apresentadas por outras linguagens, como *while* ou *do-while*. Ao invés disso, é implementado uma única estrutura (*for*). Um *for* apresenta em sua sintaxe 3 componentes: comando de inicialização, expressão condicional e um comando pós-iteração. Todos esses componentes podem ser omitidos, fazendo com que o loop passe a funcionar como um *while* ou uma repetição infinita.

- *For* básico

```
1 sum := 0
2 for i := 0; i < 10; i++ {
3     sum += i
4 }
```

- *While*

```
1 sum := 1
2 for sum < 1000 {
3     sum += sum
4 }
```

- *Loop* infinito

```
sum := 1
for {
    sum += sum
}
```

5.4 Abstrações

É o processo de identificar as qualidades ou propriedades importantes associados a modelagem que está sendo realizada. Em outras palavras, uma abstração faz distinção entre **o que** uma parte de um programa faz e **como** isso é implementado. Dessa forma, sua importância se mostra evidente na construção de programas grandes, onde o programador pode introduzir diversos níveis de abstração, através da implementação de funções e procedimentos.

5.4.1 Abstração de função

Uma abstração de função incorpora uma expressão a ser avaliada. Quando chamada, ela produz um valor como resultado. Em Go, uma função é um valor de primeira classe. Portanto, pode ser retornado por outra função ou ser passada como parâmetro.

```
func compute(fn func(float64, float64) float64) float64 {
    return fn(3, 4)
}

func main() {
  hypot := func(x, y float64) float64 {
      return math.Sqrt(x*x + y*y)
  }
  fmt.Println(compute(hypot))
}
```

- Retorno múltiplo
  Em Go, uma abstração pode retornar múltiplos valores.

```
func swap(x, y string) (string, string) {
    return y, x
}

func main() {
```

```
6       a, b := swap("hello", "world")
7       fmt.Println(a, b)
8   }
```

- Retorno de valores nomeados
  Os valores retornados por uma função podem receber nomes. Com isso, deixa-se de ter a obrigação de passar argumentos para o retorno propriamente dito. Assim, quando nenhum argumento é passado para o retorno, a função devolve como valor todos os argumentos nomeados anteriormente.

```
1  func split(sum int) (x, y int) {
2       x = sum * 4 / 9
3       y = sum - x
4       return
5  }
```

5.4.2 *Abstração de procedimento*

Uma abstração de procedimento incorpora um conjunto de comandos a serem executados. Quando chamada, ela altera o valor de variáveis.

```
1  func swap(x, y string) (string, string) {
2      return y, x
3  }
```

5.5 *Closures*

Técnica de implementação de amarração de entidades com escopo estático. Em termos práticos, é um registro que armazena uma função e o ambiente em que ela foi definida. Assim, as variáveis livres, utilizadas pela função, são persistidas seguindo o tempo de vida da mesma.

```
1  func adder() func(int) int {
2      sum := 0
3      return func(x int) int {
4          sum += x
5          return sum
6      }
7  }
8
9  pos := adder()
```

```
10 for i := 0; i < 10; i++ {
11     fmt.Println(pos(i))
12 }
```

5.6 Métodos

Go não é uma linguagem que incorpora o paradigma orientado a objetos, portanto, não existe a ideia de classes na linguagem. Apesar disso, é possível definir métodos associados a um tipo específico. Um método é uma função que apresenta um arguemnto receptor especial, podendo este ser um ponteiro, permitindo, assim, alterações nas informações amarradas ao tipo.

```
1  type Vertex struct {
2      X, Y float64
3  }
4
5  func (v Vertex) Abs() float64 {
6      return math.Sqrt(v.X*v.X + v.Y*v.Y)
7  }
8
9  func (v *Vertex) Scale(f float64) {
10     v.X = v.X * f
11     v.Y = v.Y * f
12 }
13
14 v := Vertex{3, 4}
15 v.Scale(10)
```

5.7 Interfaces

Uma interface é um tipo definido como um conjunto de assinaturas de métodos. Interfaces são implementadas implicitamente de forma que um tipo implementa uma interface ao implementar seus métodos. Uma variável que tenha como tipo determinada interface pode receber qualquer valor que implemente os métodos da interface.

Interfaces podem conter qualquer quantidade de métodos, inclusive zero. Estas são chamadas de interfaces vazias, uma vez que não implementa nenhum método, podendo, assim, receber valores de qualquer tipo (todo tipo implementa pelo menos zero métodos). Interfaces vazias são usadas em códigos que suportam valores de tipos desconhecidos. Por exemplo, **fmt.Print** recebe uma quantidade variável de argumentos do tipo interface.

```
1 type Abser interface {
2     Abs() float64
```

```
}
```

6 LINGUAGENS SEMELHANTES

As principais linguagens semelhantes ao Go são:

6.1 Python

Existem algumas semelhanças entre o Go e o Python. É possível encontrá-las entre os tipos de alto nível - *slices* e *maps* em Go, que são como as listas e os dicionários em Python, mas com tipagem estática. O *range* de Go funciona como o *enumeration* do Python.

As diferenças entre as duas linguagens são muito mais numerosas. Alguns deles podem ser chocantes para um desenvolvedor de Python. Por exemplo, em Go, não há *try-except*. Em vez disso, o Go permite que as funções retornem um tipo de erro além de um resultado.

```
// getUUID retorna um UUID estavel baseado no primeiro endereco MAC
func getUUID() (*uuid.UUID, error) {
    itfs, err := net.Interfaces()
    if err != nil {
        return nil, err
    }
    // pega a primeira interface com o endereo MAC
    for _, itf := range itfs {
        if len(itf.HardwareAddr) == 0 {
            continue
        }
        id := []byte(itf.HardwareAddr.String() + "-" + NAME)
        return uuid.NewV5(uuid.NamespaceOID, id)
    }
    return nil, errors.New("Nao pode encontrar a interface
    com o endereco MAC fornecido")
}
```

Dessa forma, quando a função for invocada deverá ser feito a verificação se um erro foi retornado ou não:

```
deviceUUID, err := getUUID()
if err != nil {
    logger.Fatal(err)
}
```

Há mais algumas diferenças em Go em relação a Python, como:

- Ponteiros
- Estruturas (tipos compostos)
- Co-rotinas
- Canais (envia mensagens entre co-rotinas)
- Métodos (Go não tem classes, mas pode definir métodos em tipos)
- *closure*

Porém, entre suas semelhanças e diferenças, o que torna as linguagens facilmente migráveis entre uma e outra é que o design das duas seguem o mesmo princípio. Isso se dá pelo fato das duas tentarem reduzir a complexidade e a desordem do códigos.

6.2 Linguagem C

As linguagens C e Go são semelhantes, uma vez que ambas apresentam paradigma imperativo. Dessa forma, analisando a sintaxe das duas é possível realizar uma comparação da estrutura de suas expressões, de seus comandos e de suas atribuições. O exemplo a seguir apresenta o código de uma soma de inteiros em ambas as linguagens.

- Exemplo em C:

```c
#include <stdio.h>

int main(void)
{
    int a, b, c;
    scanf("%d", &a);
    scanf("%d", &b);
    c = a + b;
    printf("%d\n", c);
    return 0;
}
```

- Exemplo em Go:

```go
package main

import "fmt"

func main(){
    var a, b, c int
```

```
7       fmt.Scan(&a)
8       fmt.Scan(&b)
9       c = a + b
10      fmt.Println(c)
11  }
```

Outra grande semelhança com C é o uso de ponteiros em Go, porém na linguagem da Google não é permitido aritmética com eles. Apesar disso, ponteiros são fundamentais nas duas linguagens para manipulação de vetores e arquivos.

7 EXEMPLOS DE PROGRAMAS

- Exemplo 1: *Mutex*

```
1  package main
2
3  import (
4      "fmt"
5      "sync"
6      "time"
7  )
8
9  // SafeCounter is safe to use concurrently.
10 type SafeCounter struct {
11     v   map[string]int
12     mux sync.Mutex
13 }
14
15 // Inc increments the counter for the given key.
16 func (c *SafeCounter) Inc(key string) {
17     c.mux.Lock()
18     // Lock so only one goroutine at a time can access the map c.v.
19     c.v[key]++
20     c.mux.Unlock()
21 }
22
23 // Value returns the current value of the counter for the given key.
24 func (c *SafeCounter) Value(key string) int {
25     c.mux.Lock()
26     // Lock so only one goroutine at a time can access the map c.v.
27     // Defer delays Unlock execution until value ends
28     defer c.mux.Unlock()
29     return c.v[key]
30 }
31
```

```
func main() {
    c := SafeCounter{v: make(map[string]int)}
    for i := 0; i < 100; i++ {
        go c.Inc("somekey")
    }

    time.Sleep(time.Second)
    fmt.Println(c.Value("somekey"))
}
```

- Exemplo 2: Comparação entre árvores binárias

```
package main

import (
    "fmt"
    "golang.org/x/tour/tree"
)

// Walk walks the tree t sending all values
// from the tree to the channel ch.
func Walk(t *tree.Tree, ch chan int) {
    if t != nil {
        Walk(t.Left, ch)
        ch <- t.Value
        Walk(t.Right, ch)
    }
}

// Same determines whether the trees
// t1 and t2 contain the same values.
func Same(t1, t2 *tree.Tree) bool {
    ch1 := make(chan int, 10)
    ch2 := make(chan int, 10)
    same := true
    go Walk(t1, ch1)
    go Walk(t2, ch2)

    for i := 0; i < cap(ch1) && same; i++ {
        if <-ch1 != <-ch2 {
            same = false
        }
    }

    return same
}

func main() {
```

```
37      fmt.Println(Same(tree.New(1), tree.New(1)))
38      fmt.Println(Same(tree.New(1), tree.New(2)))
39  }
```

8 CONCLUSÃO

Go é uma linguagem de programação nova, multiparadigma, destacada por abranger várias caracteristicas, como tipagem forte e estática. Além disso, possui recursos de implementação que possibilitam o desenvolvedor a ter uma melhor experiência quanto sua simplicidade e sua performance. Ademais, por ser uma linguagem open source e possuir grande apoio da comunidade, vários novos recursos vêm sendo adcionados a sua biblioteca padrão, que já é de grande potência, se tornando ainda mais atrativa a outros programadores.

Por outro lado, a linguagem Go ainda não possui nenhuma aplicação que seja desenvolvida inteiramente em sua gramática. Mas, o suporte para tal evento tem sido cada vez mais crescente e a utilização da mesma com fins didáticos tem sido cada vez mais explorado, devido sua sintática e semântica de fácil compreenção. Com isso, pode-se explorar ainda mais a aplicabilidade da linguagem e o desenvolvimento de uma aplicação completa nesta se torna cada vez mais próxima.

Conclui-se, portanto, que apesar de ser uma linguagem recente, ela tem mostrado seu grande potencial e atraído cada vez mais desenvolvedores. Concomitantemente a isso, seus recursos despertam maior interesse em outras linguagens de programação, pois Go explora de forma otimizada os recursos apresentados por outras. Não atoa, a comunidade Go tem crescido cada vez mais e os recursos por ela acrescentado ou requerido, se torna gradativamente mais corrente.

Referências

DEITEL, Harvey M.; DEITEL, Paul J. **Java How to Program (6th Edition)**. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 2004. ISBN 0131483986.

DEVMEDIA. **Conhecendo Java Reflection**. 2018. Disponível em: <https://www.devmedia.com.br/conhecendo-java-reflection/29148>.

GANESH, S.G. **Let's GO: A First Look at Google's Go Programming Language**. 2010. Disponível em: <https://opensourceforu.com/2010/05/first-look-at-google-go-programming-language/>.

GOOGLE. **Go by Example**. 2017. Disponível em: <https://gobyexample.com/>.

GOOGLE. **A linguagem de programação Go**. 2017. Disponível em: <http://www.golangbr.org/>.

GOOGLE. **Tour Golang**. 2017. Disponível em: <https://tour.golang.org>.

M.R., Costa. **Linguagens de Programação - Notas de aula**. 2018. Disponível em: <https://www.sistemas.pucminas.br/sga>.

OPENSOURCE. **Let's GO: A First Look at Google's Go Programming Language**. 2010. Disponível em: <https://opensourceforu.com/2010/05/first-look-at-google-go-programming-language/>.

QUORA. **Some unique features of Golang**. 2017. Disponível em: <https://www.quora.com/What-are-some-unique-features-of-Golang>.

SARDA, Deepak. **The Zen of Python**. Berkeley, CA, USA: Apress, 2017. ISBN 9781484232347.

SEBESTA, R.W. **Conceitos de Linguagens de Programação - 9.ed.:**. Grupo A - Bookman, 2009. ISBN 9788577808625. Disponível em: <https://books.google.com.br/books?id=vPldwBmt-9wC>.

XORIENT. **Go Programming Language: Key Features**. 2016. Disponível em: <https://www.xoriant.com/blog/product-engineering/go-programming-language-key-features.html>.