

**CENTRO UNIVERSITÁRIO CARIOCA**

**RAPHAEL LIMA VICARONE**

**THIAGO FERNANDES DA SILVA ALMEIDA**

**LINGUAGEM DE PROGRAMAÇÃO GO: HISTÓRIA, ANÁLISE E CONSTRUÇÃO**

**DE UMA APLICAÇÃO**

**RIO DE JANEIRO**

**2021RAPHAEL LIMA VICARONE**

**THIAGO FERNANDES DA SILVA ALMEIDA**

**LINGUAGEM DE PROGRAMAÇÃO GO: HISTÓRIA, ANÁLISE E CONSTRUÇÃO**

**DE UMA APLICAÇÃO**

Trabalho de Conclusão de Curso apresentado ao Centro Universitário Carioca, como requisito parcial para obtenção do grau de Bacharel em Ciência da Computação.

Orientador: Prof. D.Sc Sérgio Assunção Monteiro

**RIO DE JANEIRO**

**2021**

**RAPHAEL LIMA VICARONE**

**THIAGO FERNANDES DA SILVA ALMEIDA**

**LINGUAGEM DE PROGRAMAÇÃO GO: HISTÓRIA, ANÁLISE E CONSTRUÇÃO DE UMA APLICAÇÃO.**

Trabalho de Conclusão de Curso apresentado ao Centro Universitário Carioca, como requisito parcial para obtenção do grau de Bacharel em Ciência da Computação

.

BANCA EXAMINADORA

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Prof. Sérgio Assunção Monteiro, D.Sc - Orientador

Centro Universitário Carioca

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Prof. André Luiz Avelino Sobral, M.Sc

Centro Universitário Carioca

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Prof. Alberto Tavares da Silva, D.Sc

Centro Universitário CariocaAGRADECIMENTOS

Jhsdgwruohg ekjgsogijsg skgjlsigja lrjhg lsjglsgj irirfuhfg itijrigj erg kerjg kejrgeirjg eigj ekjgoirgjwpoeijgpw oirgjwoeirjgwçejg wet2gweg

Wet rjehgowrgjwirgj kljgço irgj weoirg ojrlgqjorgij wreoiwjergu5uuurjg wierjgweuhoujgijr53yt298rug w985uo u hwrohwpo 8r towurt uoitworut 985 rjtiowetw0985u twoierjg weprigj w85u wlekgwçeoritjwçoit v

Weoit woeirjt wortjw peuirtj8ijtirjtijtijjirjituw´085uwe5jtp8u´t09a~pifjçofghjwoeif jg eroigjw98u03jpifjg reijg woeirjgpw9urtjwçerjtwo irjtoeirjtoeirjgo erijt oi vov otrjpwoirtpw858tujoergjwoeirtjpwoerjtwpiorejtv woeritjwçoeit´wety

**RESUMO**

Desenvolver software nunca foi uma tarefa fácil. Escrever um programa, não importa o quão pequeno seja, requer uma dose alta de paciência e atenção aos detalhes e as vezes parece não haver solução para o problema. E nada disso parece ter mudado ao longo do tempo. Novas linguagens de programação e novas ferramentas todos os dias são criadas por programadores em todo o mundo e o desafio não diminuiu, pelo contrário, o número de núcleos dos processadores vem aumentado mesmo em dispositivos móveis, desenvolvedores e administradores de sistema sofrem para aproveitar ao máximo a máquina e não afetar as tarefas dos usuários.

Conforme o número de usuários aumenta, fica mais difícil encontrar alguma linguagem de programação que resolva todos esses problemas facilmente. A linguagem Go aparece em um ambiente onde tais problemas são ainda mais extremos: Google.

Embora a linguagem Go seja inspirada na linguagem C, ela possui recursos de alto nível, como abstração de certas estruturas de dados, coleta de lixo e duck typing, além de trazer métodos modernos e elegantes para a criação de aplicativos concorrentes. Go também inclui uma extensa biblioteca padrão com ferramentas para comunicação em redes, servidores HTTP, expressões regulares, leitura e escrita de arquivos e muito mais.

O objetivo desta pesquisa é apresentar os recursos da linguagem Go e partes importantes da biblioteca padrão, uma análise de sua sintaxe e semântica, apresentar sua história e objetivos em sua concepção, além da construção de uma aplicação completa, uma Rede Social.

**Palavras chave:** Google, Sintaxe, Semântica, Goroutines, Paradigma Funcional, Concorrência, Escalabilidade.

**ABSTRACT**

Developing software has never been an easy task. Writing a program, no matter how small, requires a high dose of patience and attention to detail and sometimes there seems to be no solution to the problem. And none of that seems to have changed over time. New programming languages ​​and new tools are raised every day by programmers around the world and the challenge has not diminished, on the contrary, the number of cores for those adapted has increased even on mobile devices. , developers and system administrators fully recover the machine and do not affect users' tasks.

As the number of users increases, it becomes more difficult to find any programming language that solves all these problems easily. The Go language appears in environment where such problems are even more extreme: Google.

Although the Go language is inspired by the C language, it has high-level features, such as abstraction of data structures, garbage collection and duck typing, in addition to bringing modern and elegant methods for creating competing applications. Go also includes an extensive standard library with tools for communicating over networks, HTTP servers, regular expressions, reading and writing files and more.

The objective of this research is to present the resources of the Go language and important parts of the standard library, an analysis of its syntax and semantics, to present its history and objectives in its conception, in addition to the construction of a complete application, a Social Network.

**Keywords:** Google, Syntax, Semantics, Goroutines, Functional Paradigm, Competition, Scalability.

**LISTA DE ILUSTRAÇÕES**

**Figuras**

[Figura 1 – Código Fonte Hello Word 15](#_30j0zll)

[Figura 2 - figura 20](#_1fob9te)

[Figura 3 - figura 21](#_3znysh7)

[Figura 4 - figura 24](#_2et92p0)

[Figura 5 - figura 25](#_tyjcwt)

[Figura 6 - figura 26](#_3dy6vkm)

[Figura 7 - I figura 27](#_1t3h5sf)

[Figura 8 - T figura 28](#_4d34og8)

[Figura 9 - figura 29](#_2s8eyo1)

[Figura 10 - figura 30](#_17dp8vu)

[Figura 11 - figura 32](#_3rdcrjn)

[Figura 12 - figura 34](#_26in1rg)

[Figura 13 - figura 36](#_lnxbz9)

[Figura 14 - figura 37](#_35nkun2)

[Figura 15 - figura 38](#_1ksv4uv)

[Figura 16 - figura 39](#_44sinio)

[Figura 17 - figura 40](#_2jxsxqh)

[Figura 18 - figura 42](#_z337ya)

[Figura 19 - figura 43](#_3j2qqm3)

[Figura 20 - figura 44](#_1y810tw)

[Figura 21 - figura 45](#_4i7ojhp)

[Figura 22 - figura 46](#_2xcytpi)

[Figura 23 - figura 47](#_1ci93xb)

[Figura 24 - figura 48](#_3whwml4)

[Figura 25 - figura 49](#_2bn6wsx)

[Figura 26 - figura 50](#_qsh70q)

**Sumário**

1. INTRODUÇÃO 10

1.1. Porquê GO 11

2. ESTRUTURA DO TRABALHO 12

1. **INTRODUÇÃO**

Go é uma linguagem extremamente simples que favorece a criação de programas rápidos, leves, confiáveis, fáceis de entender, documentar e de manter. A primeira  impressão é de que lhe falta funcionalidades mais complexas que outras linguagens possuem, no entanto, o intuito de seus criadores foi exatamente o inverso, que GO fosse uma linguagem simples onde só existe uma forma ou poucas formas  de realizar uma tarefa, sendo assim, GO é uma linguagem fácil de aprender e que não possui muito a gerar discussão entre os programadores  do tipo, essa funcionalidade é melhor para tal tarefa.

Não é uma linguagem pensada para programas curtos, mas é muito bem otimizada para programas escaláveis, pois foi criada pela gigante Google para a própria Google, pensando em uma linguagem que otimize e melhore a produção e compilação de código por seus programadores em suas milhares de linhas de código.

Não existe linguagem que seja capaz de resolver todos os problemas de forma fácil, porém o GO foi criado em um ambiente em que esses problemas são extremamente grandes, o Google.

Apesar de bastante inspirada na linguagem C, Go possui características de mais alto nível, como abstrações para algumas estruturas de dados, garbage collection e duck typing, além de  trazer uma  abordagem moderna e elegante para a criação de aplicações concorrentes. Go também inclui uma extensa biblioteca padrão com ferramentas para comunicação em redes, servidores HTTP, expressões regulares, leitura e escrita de arquivos e muito mais.

**2. PORQUE GO?**

Go nasceu como um projeto interno no Google, iniciado  em 2007 por  Rob Pike, Ken Thompson e Robert Griesemer, e posteriormente lançado como um projeto de código aberto em novembro de 2009. Pike e Thompson já haviam trabalhado juntos no Bell Labs, o berço do Unix e do Plan 9, o sistema operacional que deu origem às ferramentas de compilação usadas como base do desenvolvimento da linguagem Go.

No artigo "Go at Google: Language Design in  the Service of Software Engineering" (<http://talks.golang.org/2012/splash.article>), Rob Pike  atribui os longos períodos de compilação e a dificuldade em escalar o desenvolvimento de grandes aplicações como sendo os principais motivadores para a criação da nova linguagem.  Segundo ele, a maior parte do problema é a forma com que as linguagens C e C++ tratam o gerenciamento de dependências entre os diversos arquivos-fonte.

Dentro deste cenário, Go foi concebida com o objetivo de tornar o desenvolvimento de servidores no Google uma tarefa mais produtiva e eficiente.

Para evitar os problemas de compilação, Go  implementa um controle rigoroso e inteligente de dependências, baseado na definição e uso  de package (pacotes).

Com uma sintaxe bastante limpa, se comparada com outras linguagens inspiradas em C, como C++ e Java, Go permite a escrita de programas concisos e legíveis, além de facilitar bastante a escrita de ferramentas que interagem com o código-fonte. Bons exemplos de tais ferramentas são o  go fmt  (formata o código de acordo com o guia de estilo da linguagem) e go fix (reescreve partes do código que usa APIs depreciadas para que usem as novas APIs introduzidas em versões mais recentes).

Go possui tipagem forte e estática, porém introduz uma forma curta de declaração de variáveis baseada em  inferência de tipos, evitando redundância e produzindo código muito mais sucinto do que linguagens estaticamente tipadas tradicionais. Além disso, Go traz uma implementação de duck typing: se faz "quack" como um pato e anda como um pato, então provavelmente é um pato. Baseado em interfaces, permitindo a criação de tipos bastante flexíveis. Alguns tipos de coleção de dados como slices, listas de tamanho dinâmico, e maps, dicionários de dados associativos, são nativos à linguagem, que também fornece um conjunto de funções embutidas para a manipulação destes tipos, arrays, listas de tamanho fixo, também estão disponíveis para casos em que um nível maior de controle é necessário.

Go suporta o uso de ponteiros, referências a endereços de memória, o que torna ainda mais fácil a criação de poderosos tipos customizados. Entretanto, aritmética sobre ponteiros não é suportada.

Apesar de toda a flexibilidade, através do uso de um coletor de lixo Go reduz drasticamente a complexidade no gerenciamento de memória das aplicações, tornando-as mais robustas e evitando vazamentos descuidados. Go também fornece recursos que permitem a escrita de programas totalmente procedurais, orientados a objetos, através da definição de novos tipos e de funções  que operam sobre tais tipos, ou funcionais, funções são membros de primeira classe em Go, que também suporta a criação de closures, funções que herdam o contexto de onde elas foram definidas.

A abordagem de Go para concorrência é um dos maiores diferenciais da linguagem. Inspirada no famoso paper de C.A.R. Hoare Communicating Sequential Processes [ref csp-hoare], Go implementa goroutines que são processos extremamente leves que se comunicam através de channels, evitando o uso de memória compartilhada e dispensando o uso de travas, semáforos e outras técnicas de sincronização de processos.

**2.1 ORIGENS DO GO**

Podemos observar no mundo as espécies das mais variadas de animais que existem hoje e delas podemos destacar características que foram herdadas de seus ancestrais durante o processo evolutivo das espécies, assim também acontece com as linguagens de programação e não foi diferente com o Go.

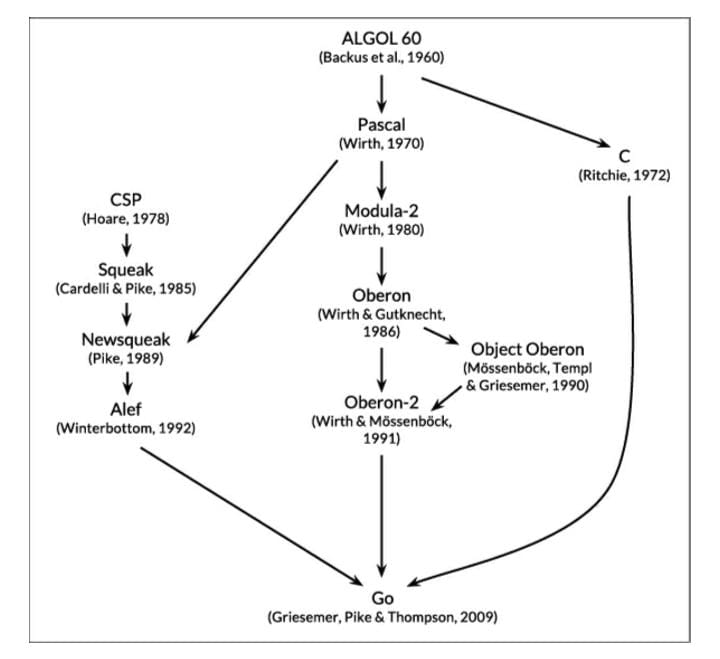


Figure : Ascendência GO

Alguns descrevem GO como uma linguagem tipo C outros dizem que é o C do século XXI, pois GO herdou de C a sintaxe de suas expressões, as instruções de controle de fluxo, tipos primitivos, passagem de parâmetro por valor, ponteiros e a ênfase em programas compilados que geram código de máquina eficientes e cooperem naturalmente com as abstrações dos sistemas operacionais atuais.

Existem outras linguagens que deram origem ao GO. Uma linha principal de influência vem das linguagens de Niklaus Wirth, começando com Pascal. Modula-2 inspirou o conceito de pacote. Oberon eliminou a distinção entre arquivos de interface de módulos e arquivos de implementação de módulos. Oberon-2 influenciou a sintaxe de pacotes, importações e declarações, e Object Oberon forneceu a sintaxe para declarações de métodos. Outra linhagem entre os ancestrais de Go, e uma que distingue Go das linguagens de programação recentes, é uma sequência pouco conhecida de linguagens de pesquisa, desenvolvida no Bell Labs, todas inspiradas no conceito de CSP (communicating sequential processes, ou processos sequenciais comunicantes), do artigo de Tony Hoare, de 1978, sobre os fundamentos da concorrência. Em CSP, um programa é uma composição paralela de processos que não têm estados compartilhados; os processos se comunicam e se sincronizam usando canais. Porém, o CSP de Hoare era uma linguagem formal para descrever os conceitos fundamentais de concorrência, e não uma linguagem de programação para escrever programas executáveis.

Rob Pike e outras pessoas começaram a realizar testes com implementações de CSP como linguagens de verdade. A primeira recebeu o nome de Squeak (‘‘A language for communicating with mice’’1 ), que era uma linguagem para tratar eventos de mouse e de teclado, com canais criados estaticamente. A essa linguagem seguiu-se a Newsqueak, que oferecia instruções e sintaxe de expressões semelhantes a C e notação de tipos semelhante a Pascal. Era uma linguagem puramente funcional com coleta de lixo, novamente com o propósito de administrar eventos de teclado, mouse e janelas. Os canais tornaram-se valores de primeira classe, criados dinamicamente e armazenáveis em variáveis. O sistema operacional Plan 9 levou essas ideias adiante em uma linguagem chamada Alef. A linguagem Alef tentou transformar a Newsqueak em uma linguagem de programação de sistemas viável, mas a ausência da coleta de lixo automática dificultou muito a concorrência. Outras construções em Go mostram a influência de genes não ancestrais aqui e ali; por exemplo, iota, de modo geral, é proveniente de APL, e o escopo léxico com funções aninhadas vem de Scheme (e da maioria das linguagens desde então). Também encontramos novas mutações. As fatias (slices) inovadoras de Go oferecem arrays dinâmicos com acesso aleatório eficiente, mas também possibilitam arranjos sofisticados de compartilhamento de memória que lembram listas ligadas, já instrução defer surgiu com Go.

**2.2 PROJETO GO**

Todas as linguagens de programação refletem a filosofia de programação de seus criadores, o que, frequentemente, inclui um componente significativo de reação às deficiências percebidas em linguagens anteriores. O projeto Go nasceu da frustração com vários sistemas de software no Google, que sofriam de uma explosão de complexidade. (Esse problema, de forma alguma, é exclusivo do Google.) Como afirma Rob Pike, ‘‘a complexidade é multiplicativa’’: corrigir um problema deixando uma parte do sistema mais complexo, lentamente, mas de forma certeira, acrescenta complexidade a outras partes. Com uma pressão constante para acrescentar recursos, opções e configurações, e disponibilizar código rapidamente, é fácil negligenciar a simplicidade, mesmo que, em longo prazo, ela seja a chave para um bom software. Simplicidade exige mais trabalho no início de um projeto para reduzir uma ideia à sua essência, e mais disciplina durante o tempo de vida de um projeto para diferenciar mudanças boas de mudanças ruins ou nocivas. Com esforço suficiente, uma mudança boa pode ser acomodada sem comprometer o que Fred Brooks chamava de “integridade conceitual” do design, mas não uma mudança ruim, e uma mudança nociva troca a simplicidade por sua prima leviana, a conveniência. Somente por meio da simplicidade do design um sistema pode permanecer estável, seguro e coerente à medida que cresce. O projeto Go inclui a linguagem em si, suas ferramentas e bibliotecas-padrão e, por último, mas não menos importante, uma proposta cultural de simplicidade radical. Por ser uma linguagem recente de alto nível, Go tem a vantagem de usufruir de experiências anteriores, e o básico foi muito bem-feito: ela tem coleta de lixo, um sistema de pacotes, funções de primeira classe, escopo léxico, uma interface para chamadas de sistema e strings imutáveis em que o texto geralmente é codificado em UTF-8. Porém, ela tem poucos recursos quando comparada a outras linguagens e é pouco provável que vá acrescentar outros. Por exemplo, Go não tem conversões numéricas implícitas, construtores ou destrutores, sobrecarga de operadores, valores default para parâmetros, herança, genéricos, exceções, macros, anotações de funções nem armazenamento local para threads. A linguagem é madura e estável, e garante retro compatibilidade: programas Go mais antigos podem ser compilados e executados com versões mais novas de compiladores e bibliotecas-padrão. Go tem um sistema de tipos suficiente para evitar a maioria dos erros por descuido que aflige programadores de linguagens dinâmicas, porém seu sistema é mais simples que os de linguagens tipadas comparáveis a ela. Essa abordagem, às vezes, pode resultar em regiões isoladas de programação ‘‘não tipada’’ em um framework mais amplo de tipos, e programadores que usam Go não chegam ao ponto de fazer o que programadores de C++ e Haskell fazem para expressar propriedades de segurança como provas baseadas em tipos. Na prática, porém, Go oferece aos programadores uma boa dose de proteção e de vantagens quanto ao desempenho em tempo de execução, resultantes de um sistema de tipagem relativamente forte sem o peso de um sistema complexo. Go incentiva um conhecimento do design contemporâneo dos sistemas de computadores, particularmente da importância da localidade. Seus tipos de dados embutidos e a maioria das estruturas de dados das bibliotecas são concebidos para funcionar naturalmente sem inicialização explícita ou construtores implícitos, portanto há relativamente pouca alocação de memória e escritas em memória ocultas no código. Os tipos agregados de Go (structs e arrays) armazenam seus elementos diretamente, ou goroutines, são inicialmente tão pequenas que criar uma gorrotina custa pouco e criar um milhão delas é prático. A biblioteca-padrão de Go, muitas vezes descrita como uma biblioteca com ‘‘pilhas incluídas’’, oferece blocos de construção claros e APIs para E/S, processamento de texto, imagens, criptografia, rede e aplicações distribuídas, com suporte para vários formatos de arquivo e protocolos-padrão. As bibliotecas e ferramentas fazem uso intenso de convenções para reduzir a necessidade de configurações e explicações, simplificando assim a lógica dos programas; isso deixa programas Go diversificados mais semelhantes uns aos outros e, desse modo, mais fáceis de aprender. Projetos compilados com a ferramenta go usam apenas nomes de arquivo e de identificadores e, ocasionalmente, um comentário especial para determinar todas as bibliotecas, executáveis, testes, benchmarks, exemplos, variantes específicas de plataforma e documentação para um projeto, o próprio código-fonte em Go contém a especificação para gerar o projeto.

**2.3** **O PRIMEIRO PROGRAMA EM GO**

Os em GO podem ser escritos em qualquer editor de textos que tenham suporte à codificação UTF-8, os arquivos precisam ser salvos nessa fonte para que o compilador consiga processá-los. Neste trabalho foi utilizado o VS CODE para edição dos códigos fontes dos exemplos e da aplicação.

Para criar o primeiro programa em GO foi criado um diretório chamado Olá Mundo, e criado um novo arquivo chamado helloWord.go com o seguinte conteúdo:

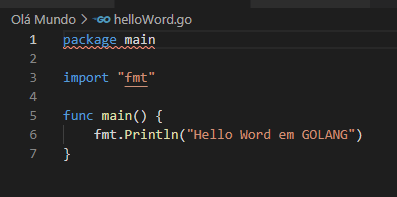


Figure : Hello Word

A linguagem C exerceu uma forte influência sobre GO, e o “Hello Word” expressa muitas dessas influências. O GO é uma linguagem compilada, onde as ferramentas da linguagem convertem o código fonte e suas dependências em um código de máquina na linguagem nativa de um computador. Essas ferramentas são acessadas por um único comando chamado go, e nele existem vários subcamandos, um subcomando que é muito utilizado é o run, que compila o código fonte de arquivos com a extensão .go, faz a ligação com a biblioteca e executa o programa.

Nesta pesquisa foi utilizado $ para representar o prompt de comando do Windows. Ao executar o comando $ go run helloWord.go será impresso na tela do prompt Hello Word em GOLANG.

O sub comando go build, cria um arquivo binário chamado helloWord, que pode ser executado a qualquer momento sem a necessidade de processamento adicional.

Go trata o UNICODE nativamente, portanto é capaz de processar qualquer linguagem do mundo.

Todo código em GO é organizado em pacotes que se assemelha com as bibliotecas ou módulos de outras linguagens. Um pacote é constituído de um ou mais arquivos fonte .go em um único diretório que define o que o pacote faz.

Cada arquivo-fonte começa com uma declaração package que nesse caso, é o package main, que define a qual pacote o arquivo pertence, seguida de uma lista de outros pacotes importados e das declarações do programa armazenadas nesse arquivo. A biblioteca-padrão de Go tem mais de cem pacotes para tarefas comuns como entrada e saída, ordenação e manipulação de texto.

O pacote fmt é um exemplo, contém funções para exibir saídas formatadas e para processar entradas. Println é uma das funções básicas de saída em fmt; ela exibe um ou mais valores separados por espaços, com um caractere de quebra de linha para que a saída seja exibida em uma única linha.

O pacote main define um programa executável independente, e não uma biblioteca. No pacote main, a função main é onde a execução do programa começa. O que main faz é o que o programa faz. É claro que main, normalmente, chamará funções de outros pacotes para fazer boa parte do trabalho, por exemplo, a função fmt.Println. Devemos dizer ao compilador quais pacotes são necessários ao arquivo-fonte; esse é o papel da declaração import que vem depois da declaração package. O programa ‘‘hello, world’’ usa apenas uma função de outro pacote, mas a maioria dos programas importará mais pacotes.

GO não permite que seja declarado pacotes, funções ou variáveis que não estejam sendo usados, isso evita que referência a o que não foi usado, se acumule com a evolução do programa.

As declarações import devem estar após a declaração package. Depois disso, um programa é constituído de declarações de funções, variáveis, constantes e tipos (introduzidos pelas palavras reservadas func, var, const e type); na maioria das vezes, a ordem das declarações não importa. Esse programa é tão pequeno quanto possível, pois declara apenas uma função que, por sua vez, chama apenas outra função.

Uma declaração de função é constituída da palavra reservada func, o nome da função, uma lista de parâmetros (vazia para main), uma lista de resultados (também vazia nesse caso) e o corpo da entre chaves. Go não exige ponto-e-vírgula no final de instruções ou declarações, exceto quando duas ou mais estiverem na mesma linha. Na verdade, quebras de linha após determinados elementos sintáticos são convertidas em ponto-e-vírgula, portanto o local em que as quebras de linha são colocadas é importante para a análise correta de código em Go. Por exemplo, a chave de abertura { da função deve estar na mesma linha que o final da declaração func, e não em uma linha separada; na expressão x + y, uma quebra de linha é permitida após o operador +, mas não antes.

Go tem uma posição rígida quanto à formatação de código. A ferramenta gofmt reescreve código em um formato padrão, e o subcomando fmt da ferramenta go aplica gofmt a todos os arquivos do pacote especificado, ou àqueles que estão no diretório atual, por padrão.

Estabelecer um formato padrão por decreto elimina muitos debates sem sentido sobre trivialidades e, acima de tudo, permite várias transformações automáticas de código-fonte que seriam impraticáveis se uma formatação arbitrária fosse permitida. Muitos editores de texto podem ser configurados para executar gofmt sempre que salvar um arquivo, de modo que seu código-fonte estará sempre devidamente formatado. Uma ferramenta relacionada, goimports, adicionalmente administra a inserção e a remoção de declarações de importação conforme for necessário. Essa ferramenta não é parte da distribuição padrão, mas pode ser obtida através do comando $ go get golang.org/x/tools/cmd/goimports.

**2.4 ESTRUTURA DE PROGRAMAS E SINTAXE**

Em Go, assim como em outras linguagens de programação, criamos programas grandes a partir de um conjunto pequeno de construções básicas. Variáveis armazenam valores. Expressões simples são combinadas para formar expressões maiores com operações como adição e subtração. Tipos básicos são reunidos em agregados como arrays e estruturas. Expressões são usadas em instruções cuja ordem de execução é determinada por instruções de controle de fluxo como if e for. Instruções são agrupadas em funções para isolamento e reutilização. Funções são reunidas em arquivos-fonte e em pacotes.

**2.5 NOMES**

Em GO, existem 25 nomes que são reservados e só poderão ser utilizadas em momentos específicos em que a sintaxe da linguagem permite ou exige, além disso GO é case sensitive, por ser nativamente padronizado para texto UNICODE, aceita qualquer nome desde que não seja da lista de reservados e diferenciando maiúscula

Tabela

Descrição gerada automaticamentede minúscula.

Além disso existem aproximadamente 30 nomes pré-declarados como *init* e *true* para constantes, tipos e funções embutidas.

Texto

Descrição gerada automaticamente

**Figura 4** – Nomes pré-declarados

Esses nomes não são reservados, podem ser usados, porém, podem causar confusão e dificultar a leitura para outros programadores, dificultando assim a manutenção do código.

Se uma entidade for declarada em uma função, ela será local a essa função, porém, se for declarada fora de uma função, ela será visível a todos os arquivos do pacote ao qual ela pertence.

O fato de a primeira letra do nome ser maiúscula ou minúscula determina sua visibilidade além das fronteiras dos pacotes. Se o nome começa com uma letra maiúscula, ele é exportado, o que significa que é visível e acessível fora de seu próprio pacote e pode ser referenciado por outras partes do programa, como Printf do pacote fmt. Nomes de pacotes usam sempre letras minúsculas.

Não há limite para o tamanho dos nomes, mas, segundo a convenção e o estilo de programas Go, a tendência é usar nomes curtos, especialmente para variáveis locais com escopos pequenos, porém, segundo Robert C Martin (2008) o nome de uma variável, pacote ou função deve ser clara, ou seja, tornar o propósito da variável bem definido pra qualquer um que for ler o código:

Escolher bons nomes leva tempo, mas economiza mais. Portanto cuide de seus nomes e troque-os quando encontrar melhores. Todos que lerem o seu código (incluído você mesmo) ficarão agradecidos.

O nome de uma variável deve responder a todas as grandes questões. Ele deve lhe dizer porque existe, o que faz e como é usado. Se um nome requer um comentário então não revela o seu propósito (MARTIN, 2008, p. 18).

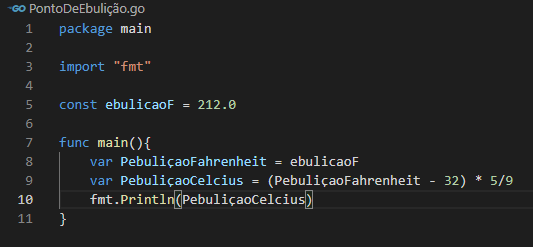
Geralmente, quanto maior o escopo de um nome, mais longo e mais significativo ele deve ser. Do ponto de vista de estilo, programadores de Go usam ‘‘camel case’’ para compor nomes por combinação de palavras, isto é, letras maiúsculas no interior do nome são preferíveis a underscores no meio. Desse modo, as bibliotecas-padrão têm funções com nomes como QuoteRuneToASCII e parseRequestLine, mas jamais quote\_rune\_to\_ASCII ou parse\_request\_line. As letras de siglas e acrônimos como ASCII e HTML são todas representadas com maiúscula ou minúscula, portanto, uma função pode se chamar htmlEscape, HTMLEscape ou escapeHTML, mas nunca escapeHtml.

**2.6 DECLARAÇÕES**

Declarações especificam uma ou mais entidades do programa, seu nome e suas propriedades, ***func***, ***var***, ***const*** e ***type*** são as principais declarações em GO.

Um programa em GO começa com a declaração do package que é o pacote a qual o arquivo pertence e após isso declara-se o import, que é o comando onde importamos pacotes tanto da biblioteca padrão GO quanto de pacotes criados pelo programador. Em seguida segue-se a declaração de variáveis, tipos, funções e constantes sem uma ordem definida e são declaradas a nível de pacote (*package level*). Por exemplo, o código a seguir apresenta uma constante, uma função e duas variáveis.

**Figura 5 –** PontoDeEbulição.go

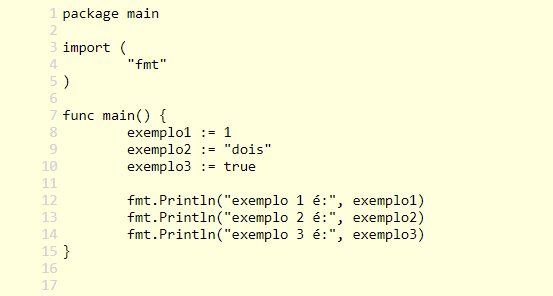


Fonte: Autoria Própria

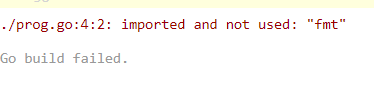
A constante “ebulicaoF” é uma declaração a nível de pacote, ou seja, é visível por todos dentro do pacote assim com a função main. Já as variáveis “PebuliçaoFahrenheit” e “PebuliçaoCelcius” foram declaradas dentro da função main e, portanto, só são visíveis dentro da função.

A função é definida pela palavra reservada ***func***, logo após declara-se o nome da função e os parâmetros, que serão preenchidos por quem chama a função e após isso pode ou não conter um retorno, em GO uma função pode ter mais de um retorno, então segue-se as instruções que serão executadas na função. A função executa a instrução até encontrar um ***return*** ou um resultado, caso não haja ***return***, que poderá ser usado no código que chamou a função.

Declarações também podem ser do tipo curtas ou implícitas, utilizando-se do operador “ **:= ”** (dois pontos e sinal de igualdade), onde a variável será inicializada ao mesmo tempo que é criada e seu tipo será inferido de acordo com o conteúdo da sua atribuição. Sua utilização é limitada ao uso de variáveis, constantes não podem ter uma declaração do tipo curta.



Importante ser mencionado que dentro da linguagem GO, durante o processo de compilação, caso o código possua uma ou mais declarações não utilizadas, isto retornará um erro, pois uma das diretrizes da linguagem é a sua eficiência que obriga no contexto de sua execução, o não desperdício de memória.



**2.7 TIPOS DE DADOS**

Os dados na linguagem GO podem ser boolean, numeric integer, numeric float, string, array, slice, struct, pointer, interface, map. É possível se atribuir uma função a uma variável, sendo assim considerada uma tipagem. Dentro do tipo numeric, podemos classificá-los quanto a sua natureza, estática ou dinâmica, quanto a sua assinatura para definir o se aceitará valores negativos dentro do intervalo possível e também designar a alocação de bits que irá compô-lo, como exemplo usar 8, 16, 32 ou 64 bits de memória.

**3. OBJETIVOS DO TRABALHO**

Este trabalho tem o objetivo de apresentar uma análise completa da linguagem GO, apresentar os principais conceitos da linguagem e da programação funcional. Os conceitos da linguagem GO juntamente com a programação funcional contribuem para que GO seja extremamente eficiente em programação concorrente, aliando toda a facilidade de se usar essa linguagem por ser simples, robusta e poderosa, pensada para criar programas escaláveis e que podem ser facilmente reutilizados e manuteníveis.

**3.1 PARADIGMA FUNCIONAL**

A programação funcional tem como principal característica a aplicação direta sobre os dados. No paradigma funcional um programa estabelece uma relação clara e explícita entre entrada e saída, Augusto Sampaio e Antônio Maranhão.

Interface gráfica do usuário, Texto, Aplicativo

Descrição gerada automaticamenteAo contrário do paradigma imperativo, que usa procedimentos, e pode ser chamado também de procedural, que usa sub-rotinas para alterar o estado dos dados. Enquanto em um programa funcional este mapeamento é direto, através de funções definidas como na matemática, em um programa imperativo tal mapeamento é indireto, via estados explicitamente modelados por variáveis globais e locais do programa em execução, Augusto Sampaio e Antônio Maranhão.

**Figura 8:** modelo computacional dos paradigmas imperativo e funcional

A programação funcional é descrita por alguns autores como Deam Wampler, O'Reilly Novatec São Paulo 2019, que descreve no seu livro Programação funcional para desenvolvedores java, a grata surpresa que foi quando resolveu buscar aprofundar-se em estudos sobre o paradigma que o fez mudar sua forma de programar que era predominantemente regido pelo paradigma Orientado a Objetos.

A medida que aprendia sobre programação funcional, encontrava boas ideias para implementar concorrência, conforme havia imaginado, mas também descobri que ela clareava meu pensamento sobre o projeto de tipos e funções, Deam Wampler.

Deam wampler também se mostra surpreso com o quanto a programação funcional é apropriada para os desafios únicos de nossa época, como trabalhar conjuntos massivos de dados e mesmo assim não perder a agilidade já que os requisitos mudam cada vez mais rapidamente e os cronogramas ficam cada vez mais curtos. E exatamente pensando no trabalho em cima de dados massivos é que foi pensada a linguagem GOLANG, pois como dito anteriormente no capítulo 1, GO é a linguagem feita pela Google e para a Google onde uma quantidade de dados gigantesca é compilado, codificado e trabalhado todos os dias, e justamente pela necessidade de trabalhar esses dados mais rapidamente e de uma maneira um pouco menos complexa que surgiu GO.

**3.2 CONCORRÊNCIA COM GOROUTINES E CHANNELS**

Caio Filipini em seu livro, Programando em GO, 2020 – Casa do Código, afirma que GO se mostrou no inicio como uma linguagem que parecia lhe faltar algumas funcionalidades encontradas em linguagens mais sofisticadas o incomodava e que funcionalidades como tipos genéricos e tratamentos de erros simples geravam certa polêmica, mas que com o passar do tempo pode ver que percebeu que essa falta de funcionalidades na verdade é algo que torna a linguagem simples, pois há somente uma forma de realizar uma tarefa na maioria das vezes.

Goroutines e Channels são funcionalidades que chamam atenção para Go no que se refere a programação concorrente e se mostram funcionalidades que tornam GO uma linguagem muito poderosa, eficiente e simples para esse tipo de programação. Go tem sido fundamental no meu dia a dia escrevendo programas concorrentes e plataformas (systems programming). É uma linguagem que favorece a criação de programas paralelos, leves, rápidos, simples de entender, de distribuir (um simples binário compilado estaticamente) e de manter, Caio Filipini.

Programação concorrente é um tema bastante delicado e complexo. Se você já escreveu programas que dependiam de múltiplas threads ou múltiplos processos sabe que é uma tarefa que exige muita atenção e paciência, especialmente quando é necessário compartilhar estado entre as diferentes linhas de execução. A maior parte das linguagens de programação implementa threads de alguma forma, seja através de um escalonador próprio ou delegando o controle para o sistema operacional hospedeiro. O compartilhamento de estado em um programa multithreaded é normalmente implementado através de variáveis globais e/ou compartilhadas e exige algum mecanismo de trava ou semáforo para evitar condições de corrida (race conditions).

Sistemas concorrentes devem lidar com atividades separadas que estão progredindo ao mesmo tempo. Informalmente, dizemos que duas atividades são concorrentes se em algum momento ambas tenham sido iniciadas, mas não terminadas, programação concorrente é a atividade de construir um programa contendo múltiplas atividades que progridem em paralelo, Dilma Menezes da Silva.

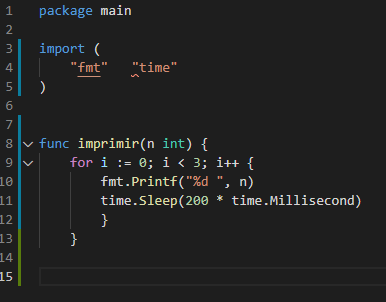
Go implementa um modelo de concorrência baseado em **goroutines** que se comunicam através de **channels,** sendo que o próprio ambiente de execução garante que apenas uma goroutine acesse um channel em um determinado momento.

*Do not communicate by sharing memory; instead, share memory by communicating* (não comunique através de memória compartilhada; em vez disso, compartilhe memória através da comunicação), effective Go.

**3.2.1 GOROUTINES**

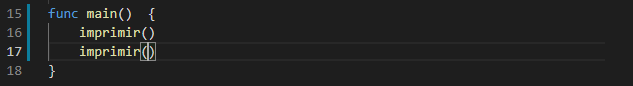
Uma goroutine é um processo muito leve que, na prática, é muito parecida com uma thread, porém as goroutines são gerenciadas pelo ambiente de execução da linguagem, e esse ambiente determina quando associar a thread do sistema operacional.

A seguinte função imprimir() recebe um número inteiro e o imprime 3 vezes com um espaço de 200 milissegundos entre eles.



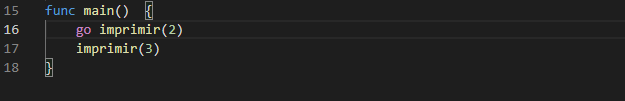
**Figura 9:** Função imprimir()

Agora considere a seguinte função main() que chama a função imprimir() duas vezes com números diferentes:



**Figura 9:** Função main() chamando duas vezes a função imprimir()

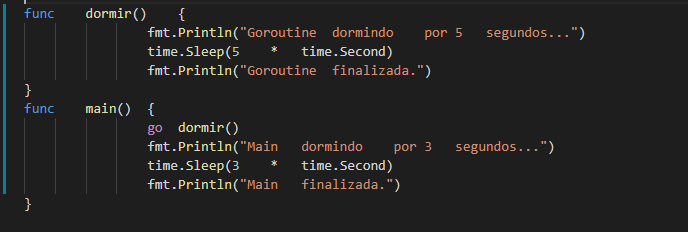
Ao executar a função main(), obteremos o resultado 2 2 2 3 3 3 agora se alterarmos o código para que seja executado uma goroutine teremos a seguinte situação:



**Figura 11:** Chamada da função main() com goroutines.

Apenas adicionando o comando go, aplicamos uma goroutine ao código fazendo com que a execução se torne concorrente e mudando da saída de pós execução do código para 3 2 3 2 3 2.

Uma goroutine depende da função main() para continuar a execução, dessa forma, caso a execução da função main() for encerrada a goroutine se encerra também, como podemos ver na figura abaixo que mostra um código que implementa uma goroutine que “dorme” por 5 segundos e a função main() dorme por 3 segundos.



**Figura 11:** Exemplo de execução de uma goroutine sendo encerrada junto com a função main()

Neste exemplo a goroutine nunca é finalizada pois a execução da função main() é encerrada antes do término da execução da goroutine, gerando a seguinte saída:

Main dormindo por 3 segundos...

Goroutine dormindo por 5 segundos...

Main finalizada.

**3.2.1.1 GOROUTINES X THREADS**

A diferença entre gorrotinas e threads, embora sejam essencialmente quantitativas, são grandes o suficiente para tornar-se qualitativa, Caio Filipini.

Uma Thread de sistema operacional tem um bloco de memória de tamanho fixo para a sua pilha (stack). Essa pilha de tamanho fixo é ao mesmo tempo grande demais ou pequena demais. Uma pilha de 2MB seria pequena demais para uma gorrotina pequena que, por exemplo espera uma condição para encerrar um channel, há também situações não raras em que o programa Go cria inúmeras gorrotinas sendo assim, uma thread com uma pilha fixa de 2MB seria pequena demais, especialmente

se possuir funções com muitas chamadas recursivas. Já uma pilha de uma gorrotina não é de tamanho fixo, ela possui um tamanho dinâmico e que se define de acordo com a necessidade.

A Thread é controlada pelo kernel do sistema operacional, que através do escalonador (scheduling), interrompe a execução de uma thread, salva seu estado atual, seleciona outra Thread, restaura seu estado anterior e executa. já uma goroutine utiliza um escalonador próprio e com uma técnica chamada escalonamento m:n (m:n scheduling), essa técnica escalona m gorrotinas em n threads do sistema operacional. A tarefa do escalonador Go é semelhante ao escalonador do kernel do sistema operacional, porém ele somente se ocupa com as gorrotinas de um único programa Go. Se uma gorrotina executa um Time.sleep ou é bloqueado em um canal ou uma operação mutex, o escalonador Go, para essa goroutine e executa outra até chegar o momento de executar novamente a primeira. Já que ele não precisa fazer a mudança para o contexto do kernel, reescalonar um gorrotina é mais simples do que reescalonar uma thread, logo é mais eficiente e mais rápida.

**3.2.1.2 GOMAXPROC**

O escalonador de Go usa um parâmetro chamado GOMAXPROCS para determinar quantas threads de sistema operacional podem estar ativas executando código Go simultaneamente. Seu valor default é o número de CPUs do computador, portanto, em um computador com oito CPUs, o escalonador escalonará código Go em até oito threads do sistema operacional ao mesmo tempo. (GOMAXPROCS é o n no escalonamento m:n.) Gorrotinas que estão dormindo ou que estejam bloqueadas em uma comunicação não precisam de uma thread.

**3.2.2 CHANNELS**

Executar diferentes goroutines concorrentemente é muito importante e problemas que exigem alta performance e grande eficiência no uso de recursos de processamento são tratados de forma mais consistente. Raramente as gorrotinas são disparadas de forma que não há interação uma com as outras.

Os channels foram pensados e criados como uma abstração para viabilizar esta comunicação de forma segura e eficiente um channel é um canal que conduz informações de um tipo válido em GO.

A função make() é utilizada para criar um novo canal. Exemplo, para criar um canal que capaz de trafegar dados do tipo inteiros utilizamos o comando c:= make (chan int).

Para interagir com um canal, utilizamos o operador <- (conhecido como

arrow operator, ou operador seta). A posição do canal em relação à seta indica a direção do fluxo da comunicação. Por exemplo, para enviar valores int para o canal c, utilizamos a seguinte notação: c <- 33

E para receber um valor enviado para o canal c: valor := <- c

Tela de computador com fundo preto

Descrição gerada automaticamente

Inicialmente criamos um canal para trafegar valores do tipo int. Em seguida, disparamos uma goroutine executando a função produzir(), que recebe um canal como argumento e simplesmente envia um número inteiro par o canal recebido.

Por padrão, operações de envio e recebimento em um canal bloqueiam até que o outro lado estar pronto. Este fato permite que a própria comunicação entre duas goroutines garanta a sincronização entre elas, sem que nenhum mecanismo de travas seja necessário.

Por este motivo, a próxima linha da função main(), que recebe um valor do canal, fará com que a linha de execução principal fique bloqueada até que algum valor seja enviado para o canal c. Assim que o valor seja enviado pela função produzir(), a linha de execução principal será então desbloqueada, o valor 33 será consumido, atribuído à variável valor e impresso no console.

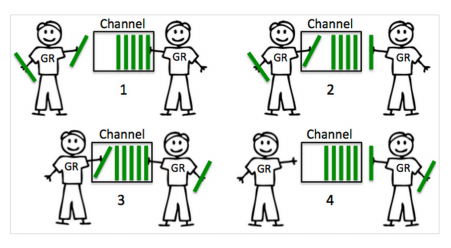
**3.2.2.1 BUFFERED CHANNEL**

Channels são como conduítes tipados pelo qual pode-se enviar e receber informações de uma goroutine a outra. Existem dois tipos de channels, os não buferizados (unbuffered channels) e os buferizados (buffered channels).

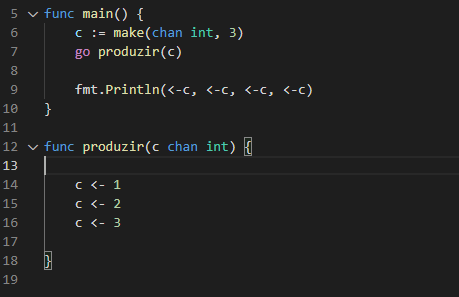
Channels buferizados são usados para envio e recebimento assíncrono de informações entre 2 ou mais goroutines. Buferizar um channel significa informar seu tamanho na sua declaração e desta forma ele poderá conter 0 ou até seu limite de informações.

Uma goroutine pode colocar informações em um channel até seu limite. Se um channel está cheio e uma goroutine tenta colocar informações nele, ela só será liberada quando liberar um espaço e puder completar seu envio.

Uma Goroutine pode pegar informações de um channel enquanto o canal não estiver vazio, se uma goroutine tentar pegar uma informação e o canal estiver vazio, ela ficara bloqueada até que haja informação a ser recuperada, então, o resgate da informação será completado e a goroutine é liberada.

Para criar um channel buferizados podemos utilizar o mesmo comando conforme a figura 12 adicionando apenas um novo parâmetro que define o tamanho buffer: c := make (chan int, 5). Neste caso o tamanho do buffer foi especificado como 5, logo a goroutine que utiliza esse canal fica bloqueada até que o buffer se complete.

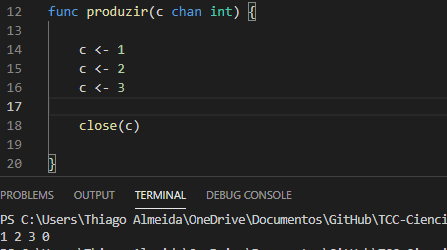
Na figura a seguir veremos um outro exemplo de channel buferizado com tamanho igual a 3 que dispara uma gorrotina que envia 3 valores pelo canal criado. Em seguida recebe 4 valores do canal e os imprimimos no console. Qual será o resultado desse programa?



Causamos um *deadlock*. A goroutine produtora encerrou sua execução logo após produzir os três valores; por causa do buffer, nenhuma das operações de envio fez com que a execução fosse bloqueada.



No entanto, ao tentarmos receber um quarto valor, que nunca foi produzido, pelo canal, a linha principal ficou bloqueada. O ambiente de execução detectou o *deadlock* e encerrou a execução do programa com um erro. O produtor precisa indicar de alguma forma que não enviará mais nenhum valor pelo canal. Para isso existe a função embutida close() . Ao alterar a função produzir() para fechar o canal após produzir os três valores o código e a saída produzida no console ficará da seguinte forma:



Dessa forma, a função close() fecha o canal sempre que não houver mais valores a ser produzidos.