

explain the y-combinator in pt-br with the techniques used

< O Y-combinator é uma técnica poderosa da programação funcional, que permite a implementação de recursão anônima em linguagens que não possuem suporte nativo para essa funcionalidade. O conceito do Y-combinator foi introduzido por Haskell Curry e recebeu esse nome em referência ao matemático combinator Alonzo Church.

Para entender o Y-combinator, primeiro precisamos entender a ideia de recursão anônima. A recursão anônima é um tipo de recursão em que uma função se chama a si mesma sem referenciar seu próprio nome. Isso pode ser um desafio em linguagens funcionais que exigem que as funções sejam nomeadas.

O truque do Y-combinator está em criar uma função de ordem superior que aceita uma função e retorna uma versão recursiva dessa função, permitindo que ela se chame a si mesma. Essa função de ordem superior é chamada de combinator Y.

Vamos explicar o Y-combinator passo a passo, usando uma linguagem funcional imaginária chamada L.

1. Definimos uma função auxiliar chamada Y, que recebe uma função **f** como argumento: **Y = fun(f) =>**
2. Dentro da função Y, retornamos uma nova função anônima que aceita um argumento **x**: **fun(x) =>**
3. Agora, dentro dessa função anônima, chamamos a função **f**, mas ao invés de passar **x** diretamente, passamos uma chamada recursiva para Y passando **x** como argumento: **f(Y(f))(x)**
4. Fechamos todas as chaves e parênteses: **Y = fun(f) => (fun(x) => f(Y(f))(x))**

Agora, podemos usar o Y-combinator para implementar recursão anônima em nossa linguagem L. Suponha que desejamos calcular o fatorial de um número usando recursão anônima. Podemos fazer isso da seguinte maneira:

1. Definimos uma função **fatorial** que recebe uma função **f** e um número **n**: **fatorial = fun(f) => fun(n) =>**
2. Em seguida, verificamos se **n** é igual a zero. Se for, retornamos 1, caso contrário, chamamos a função **f** passando **n - 1** como argumento e multiplicamos o resultado por **n**: **if (n == 0) then 1 else n * f(n - 1)**
3. Fechamos todas as chaves e parênteses: **fatorial = fun(f) => fun(n) => if (n == 0) then 1 else n * f(n - 1)**

Agora, podemos usar o Y-combinator para obter uma versão recursiva da função **fatorial**: **fatorialRecursivo = Y(fatorial)**

A função **fatorialRecursivo** será uma versão recursiva da função **fatorial** que pode ser chamada sem referenciar seu próprio nome.

Essa é uma breve explicação do Y-combinator e como ele pode ser usado para implementar recursão anônima em linguagens funcionais. Essa técnica é fundamental na programação funcional e pode ser usada para resolver problemas complexos que requerem recursão.

detail in pt-br 10 programming language ecosystems today in the benchmarking game, breaking down relevant categories

1. Python:

- A linguagem de programação Python é extremamente popular devido à sua sintaxe clara e fácil de aprender. É amplamente utilizado em ciência de dados, aprendizado de máquina e desenvolvimento web. No Jogo de Benchmarking, Python tende a ter um desempenho inferior em termos de velocidade em comparação com linguagens como C ou Java, mas sua eficiência de desenvolvimento geralmente supera essas deficiências.

2. Java:

- Java é uma linguagem orientada a objetos que é amplamente usada em desenvolvimento empresarial e aplicativos Android. Tem um bom desempenho no Jogo de Benchmarking devido à sua eficiência e velocidade, especialmente com o uso da Máquina Virtual Java (JVM).

3. JavaScript:

- JavaScript é a linguagem de programação principal para desenvolvimento web front-end. Com a introdução do Node.js, JavaScript também se tornou popular para o desenvolvimento back-end. No Jogo de Benchmarking, pode não ser a linguagem mais rápida, mas sua universalidade e versatilidade tornam-na indispensável.

4. C:

- C é uma das linguagens de programação mais antigas e ainda é amplamente usada hoje, especialmente em sistemas operacionais e hardware de baixo nível. É conhecida por sua velocidade e eficiência, tendo um desempenho forte no Jogo de Benchmarking.

5. C++:

- C++ é uma extensão de C que adiciona recursos de programação orientada a objetos. É amplamente usado em software de alto desempenho, como jogos. No Jogo de Benchmarking, C++ geralmente tem um desempenho muito bom, oferecendo um bom equilíbrio entre eficiência de código e velocidade de execução.

6. Ruby:

- Ruby é uma linguagem de programação de script de alto nível conhecida por sua sintaxe clara e limpa. É mais famoso pelo framework de desenvolvimento web, Ruby on Rails. No Jogo de Benchmarking, o desempenho de Ruby pode não ser o mais forte, mas a facilidade de uso e a produtividade do desenvolvedor são seus pontos fortes.

7. Swift:

- Swift é a linguagem de programação primária para desenvolvimento no ecossistema Apple, incluindo iOS, MacOS, watchOS e tvOS. Embora seja uma linguagem mais recente, ela foi projetada para ter um bom desempenho, com melhorias notáveis sobre seu predecessor, Objective-C, no Jogo de Benchmarking.

8. Go:

- Go é uma linguagem de programação desenvolvida pelo Google que se destaca pela concorrência eficiente e facilidade de uso. No Jogo de Benchmarking, Go geralmente tem um bom desempenho, com um tempo de execução rápido e uso eficiente dos recursos do sistema.

9. Rust:

- Rust é uma linguagem de sistema que se destaca pela segurança de memória e desempenho. É projetado para ser uma alternativa mais segura a C e C++. No Jogo de Benchmarking, Rust normalmente apresenta desempenho superior em termos de velocidade e eficiência.

10. Kotlin:

- Kotlin é uma linguagem de programação moderna que é totalmente interoperável com Java e é agora a linguagem recomendada para o desenvolvimento de aplicativos Android. No Jogo de Benchmarking, Kotlin tende a ter um desempenho semelhante ao Java, pois ambos funcionam na JVM.

Lembre-se de que o desempenho no Jogo de Benchmarking é apenas uma das muitas métricas que podem ser usadas para avaliar uma linguagem de programação. A escolha da linguagem de programação também deve levar em consideração outros fatores, como a natureza do projeto, a equipe de desenvolvimento e o ecossistema da linguagem.

but in which categories these ecosystems compete using intricate tools in the entire range of algorithms and techniques

< Os ecossistemas de linguagens de programação competem em uma variedade de categorias, incluindo, mas não se limitando a:

1. Desempenho: Algumas linguagens, como C e Rust, são projetadas para ter alto desempenho. Elas são frequentemente usadas em aplicações onde a velocidade é crucial, como sistemas operacionais, jogos e processamento de dados em grande escala.
2. Facilidade de uso: Linguagens como Python e Ruby são famosas por sua sintaxe clara e intuitiva. Elas são muitas vezes escolhidas para o protótipo rápido ou para a introdução de novatos na programação.
3. Segurança: Linguagens como Rust e Swift foram projetadas com um forte enfoque na segurança, incluindo prevenção de erros comuns, como acesso à memória após a liberação (dangling pointers).
4. Concorrência: Go e Rust são notáveis por suas funcionalidades de concorrência. A linguagem Go apresenta goroutines, enquanto Rust oferece uma abordagem de propriedade única para gerenciar o acesso à memória em cenários concorrentes.
5. Versatilidade: Algumas linguagens, como JavaScript e Python, são usadas em uma ampla gama de aplicações, desde o desenvolvimento web até a ciência de dados e aprendizado de máquina.
6. Suporte a bibliotecas e frameworks: A disponibilidade de bibliotecas e frameworks pode tornar o desenvolvimento mais rápido e fácil. Python é famoso por sua vasta gama de bibliotecas para ciência de dados e aprendizado de máquina, enquanto JavaScript possui numerosos frameworks para o desenvolvimento web.
7. Portabilidade: Linguagens como Java e Python são altamente portáveis, com a capacidade de executar o mesmo código em diferentes plataformas.
8. Comunidade e suporte: Uma comunidade forte pode fazer uma grande diferença em termos de suporte e disponibilidade de recursos de aprendizado. Linguagens como JavaScript e Python possuem grandes comunidades de desenvolvedores.
9. Interoperabilidade: Linguagens como Kotlin e TypeScript oferecem interoperabilidade com Java e JavaScript, respectivamente, permitindo que os desenvolvedores tirem proveito das características dessas linguagens enquanto aproveitam os benefícios das novas.
10. Orientação a objetos vs funcional: Algumas linguagens, como Java e C++, são fortemente orientadas a objetos, enquanto outras, como Haskell, são fortemente orientadas ao funcional. Ainda outras, como JavaScript e Python, suportam ambos os paradigmas.

list 10 languages released after 2015 detailing intricate details in how they can affect future benchmarks

< Desde 2015, várias novas linguagens de programação foram lançadas, cada uma trazendo recursos e inovações únicos que poderiam impactar futuros benchmarks. Aqui estão 10 linguagens notáveis lançadas após 2015:

1. Ballerina (2017):
 - Ballerina é uma linguagem de programação de código aberto projetada para a era da nuvem. Ela introduz um novo paradigma que une o código e a infraestrutura. Em benchmarks futuros, isso poderia alterar a forma como medimos o desempenho do código, considerando também a configuração da infraestrutura.

2. Elm (2016):

- Elm é uma linguagem de programação funcional que compila para JavaScript. É conhecida pela sua forte garantia de ausência de erros de tempo de execução. Em futuros benchmarks, Elm pode influenciar a maneira como medimos a confiabilidade do código, além da performance.

3. Zig (2016):

- Zig é uma linguagem de programação de uso geral e de sistema projetada para otimizar a manutenção de software e melhorar a produtividade do desenvolvedor. Suas características de segurança de memória e simplicidade podem afetar futuros benchmarks de desempenho e produtividade.

4. Julia (2018):

- Julia é uma linguagem de programação de alto nível e alto desempenho para computação técnica. Tem a ambição de combinar a velocidade de C com a usabilidade de Python. Isso poderia mudar a forma como comparamos linguagens de programação em termos de desempenho e facilidade de uso em benchmarks futuros.

5. ReasonML (2016):

- ReasonML é uma nova sintaxe para OCaml, criada pelo Facebook. É projetada para ser acessível para desenvolvedores JavaScript e tem forte interoperabilidade com o JavaScript. Isso pode influenciar futuros benchmarks de linguagens que se integram a ecossistemas existentes.

6. Crystal (2019):

- Crystal é uma linguagem de programação de propósito geral inspirada em Ruby, mas com desempenho e eficiência semelhantes a C. A combinação de uma sintaxe amigável com desempenho de alto nível pode impactar futuros benchmarks, especialmente na comparação entre eficiência de desenvolvimento e desempenho.

7. Dark (2019):

- Dark é uma linguagem de programação orientada a serviços, projetada para tornar a implantação de código uma coisa do passado. Seu impacto em futuros benchmarks pode ser na forma como medimos a eficiência da implantação e manutenção do código.

8. Gleam (2020):

- Gleam é uma linguagem de programação estática e fortemente tipada que roda na Erlang VM. Suas características poderiam influenciar futuros benchmarks, especialmente no contexto de sistemas distribuídos e concorrentes.

9. Odin (2016):

- Odin é uma linguagem de programação de propósito geral, de alto desempenho e estática, projetada com o objetivo de produzir código eficiente. Ela pode influenciar futuros benchmarks em termos de eficiência de desempenho e código.

10. V (2019):

- V é uma linguagem de programação de sistema simples e rápida, semelhante a Go. Ela promete segurança, simplicidade e eficiência, o que pode afetar futuros benchmarks em termos de desempenho e segurança.

Essas linguagens representam novas abordagens e filosofias na programação. O impacto delas em futuros benchmarks dependerá de como essas filosofias e características serão valorizadas e medidas.

if actions used toml, convert this yaml to toml to test

< The YAML content provided represents a GitHub Actions workflow file. It is defined in a specific structure expected by GitHub.

TOML (Tom's Obvious, Minimal Language) is a simple language that provides a configuration file format. However, as of my knowledge cutoff in September 2021, GitHub Actions does not natively support TOML for defining workflows, hence the conversion might not be useful in practice. Nevertheless, here's how the provided YAML might look in TOML:

```
[name]
value = "Release"

[on]
[on.push]
branches = ["main"]

[jobs]
[jobs.deploy-widgets]
runs-on = "ubuntu-latest"
name = "Make a function call on mainnet"

[jobs.deploy-widgets.env]
NEAR_NETWORK_CONNECTION = "mainnet"
NEAR_CONTRACT_ACCOUNT_ID = "${{ vars.NEAR_CONTRACT_ACCOUNT_ID }}"
NEAR_SIGNER_ACCOUNT_ID = "${{ vars.NEAR_SIGNER_ACCOUNT_ID }}"
NEAR_SIGNER_ACCOUNT_PUBLIC_KEY = "${{ vars.NEAR_SIGNER_ACCOUNT_PUBLIC_KEY }}"
NEAR_SIGNER_ACCOUNT_PRIVATE_KEY = "${{ secrets.NEAR_SIGNER_ACCOUNT_PRIVATE_KEY }}"

[[jobs.deploy-widgets.steps]]
name = "Checkout repository"
uses = "actions/checkout@v2"

[[jobs.deploy-widgets.steps]]
name = "Install near CLI"
run = '''
curl --proto 'https' --tlsv1.2 -LsSf https://github.com/near/near-cli-rs/releases/download/
v0.3.1/near-cli-rs-v0.3.1-installer.sh | sh
'''

[[jobs.deploy-widgets.steps]]
name = "Call some function"
run = '''
near contract call-function as-transaction "$NEAR_CONTRACT_ACCOUNT_ID" 'function_name_here'
json-args '{}' prepaid-gas '100 TeraGas' attached-deposit '0 NEAR' sign-as "$NEAR_SIGNER_-
ACCOUNT_ID" network-config "$NEAR_NETWORK_CONNECTION" sign-with-plaintext-private-key --signer-
public-key "$NEAR_SIGNER_ACCOUNT_PUBLIC_KEY" --signer-private-key "$NEAR_SIGNER_ACCOUNT_-
PRIVATE_KEY" send
'''
```

Keep in mind that GitHub Actions won't recognize this TOML configuration. It's simply a demonstration of how the YAML could be translated to TOML.

how to make the conversion even more concise? considering toml's idiomatic features

< The TOML format is indeed designed to be more human-readable and concise, but it doesn't provide a way to shorten or simplify the content of your specific YAML configuration, because the nature of the data being represented is still the same.

However, you can try to structure the TOML to reduce visual clutter, even though it may not reduce the line count significantly. Here is another attempt, which groups related keys:

```

[name]
value = "Release"

[on.push]
branches = ["main"]

[jobs.deploy-widgets]
runs-on = "ubuntu-latest"
name = "Make a function call on mainnet"

[jobs.deploy-widgets.env]
NEAR_NETWORK_CONNECTION = "mainnet"
NEAR_CONTRACT_ACCOUNT_ID = "${{ vars.NEAR_CONTRACT_ACCOUNT_ID }}"
NEAR_SIGNER_ACCOUNT_ID = "${{ vars.NEAR_SIGNER_ACCOUNT_ID }}"
NEAR_SIGNER_ACCOUNT_PUBLIC_KEY = "${{ vars.NEAR_SIGNER_ACCOUNT_PUBLIC_KEY }}"
NEAR_SIGNER_ACCOUNT_PRIVATE_KEY = "${{ secrets.NEAR_SIGNER_ACCOUNT_PRIVATE_KEY }}"

[jobs.deploy-widgets.steps.checkout]
name = "Checkout repository"
uses = "actions/checkout@v2"

[jobs.deploy-widgets.steps.install]
name = "Install near CLI"
run = '''
curl --proto '=https' --tlsv1.2 -LsSf https://github.com/near/near-cli-rs/releases/download/
v0.3.1/near-cli-rs-v0.3.1-installer.sh | sh
'''

[jobs.deploy-widgets.steps.call]
name = "Call some function"
run = '''
near contract call-function as-transaction "$NEAR_CONTRACT_ACCOUNT_ID" 'function_name_here'
json-args '{}' prepaid-gas '100 TeraGas' attached-deposit '0 NEAR' sign-as "$NEAR_SIGNER_-
ACCOUNT_ID" network-config "$NEAR_NETWORK_CONNECTION" sign-with-plaintext-private-key --signer-
public-key "$NEAR_SIGNER_ACCOUNT_PUBLIC_KEY" --signer-private-key "$NEAR_SIGNER_ACCOUNT_-
PRIVATE_KEY" send
'''

```

This way, each step has its own unique section, but it's less idiomatic than the previous version because TOML arrays are typically used to represent multiple similar objects.

Remember, GitHub Actions as of my knowledge cutoff in September 2021, won't recognize this TOML configuration.

if an official implementation were to be implemented, turn the yaml sample into the most concise toml equivalent

< If GitHub Actions were to support TOML and considering the language's features for conciseness, the configuration might look something like this:

```

name = "Release"

[on.push]
branches = ["main"]

[jobs.deploy-widgets]
runs-on = "ubuntu-latest"
name = "Make a function call on mainnet"

[jobs.deploy-widgets.env]

```

```

NEAR_NETWORK_CONNECTION = "mainnet"
NEAR_CONTRACT_ACCOUNT_ID = "${{ vars.NEAR_CONTRACT_ACCOUNT_ID }}"
NEAR_SIGNER_ACCOUNT_ID = "${{ vars.NEAR_SIGNER_ACCOUNT_ID }}"
NEAR_SIGNER_ACCOUNT_PUBLIC_KEY = "${{ vars.NEAR_SIGNER_ACCOUNT_PUBLIC_KEY }}"
NEAR_SIGNER_ACCOUNT_PRIVATE_KEY = "${{ secrets.NEAR_SIGNER_ACCOUNT_PRIVATE_KEY }}"

[[jobs.deploy-widgets.steps]]
name = "Checkout repository"
uses = "actions/checkout@v2"

[[jobs.deploy-widgets.steps]]
name = "Install near CLI"
run = '''
curl --proto '=https' --tlsv1.2 -LsSf https://github.com/near/near-cli-rs/releases/download/
v0.3.1/near-cli-rs-v0.3.1-installer.sh | sh
'''

[[jobs.deploy-widgets.steps]]
name = "Call some function"
run = '''
near contract call-function as-transaction "$NEAR_CONTRACT_ACCOUNT_ID" 'function_name_here'
json-args '{}' prepaid-gas '100 TeraGas' attached-deposit '0 NEAR' sign-as "$NEAR_SIGNER_-
ACCOUNT_ID" network-config "$NEAR_NETWORK_CONNECTION" sign-with-plaintext-private-key --signer-
public-key "$NEAR_SIGNER_ACCOUNT_PUBLIC_KEY" --signer-private-key "$NEAR_SIGNER_ACCOUNT_-
PRIVATE_KEY" send
'''

```

This configuration is similar to the previous examples, but it drops some of the nesting (like the ‘on’ table) for a more flattened and concise format. Note that this would only be possible with changes to the underlying software interpreting the configuration files, which as of my knowledge cutoff in September 2021, does not support TOML in GitHub Actions.