Universidade do Estado do Rio de Janeiro
Centro de Tecnologia e Ciências
Instituto de Matemática e Estatística

Gabriella Silva Montenegro dos Santos

ANÁLISE DO USO DE CORROTINAS EM LINGUAGENS ESTRUTURADAS

Rio de Janeiro
AAAA

Gabriella Silva Montenegro dos Santos

ANÁLISE DO USO DE CORROTINAS EM LINGUAGENS ESTRUTURADAS

Trabalho de Conclusão de Curso apresentado, como requisito parcial para obtenção do título de Cientista da Computação, à Instituto de Matemática e Estatística, da Universidade do Estado do Rio de Janeiro.

Orientador: Prof. Daniel José Nahid Mansur Chalhub, DSc
Coorientador: cargo nome sobrenome, titulação

Rio de Janeiro
AAAA

Página da Ficha Catalográfica:

A biblioteca deverá providenciar a ficha catalográfica. Salve a ficha no formato PDF.

Substitua esse arquivo ***Ficha.pdf*** na pasta ***B.PreTextual*** pelo pdf da sua ficha catalográfica enviado pela biblioteca.

CATALOGAÇÃO NA FONTE

UERJ / REDE SIRIUS / BIBLIOTECA CTC/B

XXXX XXXX XXXX XXXX XXXX XXXX XXXX XXXX XXXX

XXXX XXXX XXXX XXXX XXXX XXXX XXXX XXXX XXXX

XXXX XXXX XXXX XXXX XXXX XXXX XXXX XXXX XXXX

XXXX XXXX XXXX XXXX XXXX XXXX XXXX XXXX XXXX

XXXX XXXX XXXX XXXX XXXX XXXX XXXX XXXX XXXX

XXXX XXXX XXXX XXXX XXXX XXXX XXXX XXXX XXXX

_____ _____

Assinatura Data

Gabriella Silva Montenegro dos Santos

ANÁLISE DO USO DE CORROTINAS EM LINGUAGENS ESTRUTURADAS

Trabalho de Conclusão de Curso apresentado, como requisito parcial para obtenção do título de Cientista da Computação, à Instituto de Matemática e Estatística, da Universidade do Estado do Rio de Janeiro.

Aprovada em DD de MMMMMM de AAAA.
Banca Examinadora:

Prof. Daniel José Nahid Mansur Chalhub, DSc (Orientador)
Universidade do Estado do Rio de Janeiro (UERJ) - PPG-EM

---

cargo nome sobrenome, titulação (Coorientador)
unidade – instituição

Prof. Norberto Mangiavacchi, Ph.D.
Universidade do Estado do Rio de Janeiro (UERJ) - PPG-EM

Eng. João José, M.Sc.
Instituição

Eng. Marcos João, B.Sc.
Instituição

---

quarto membro titular da banca
Instituição

Rio de Janeiro

AAAA

DEDICATÓRIA

Eu dedico essa tese para uma pessoa muito especial.

AGRADECIMENTOS

Texto de agradecimento texto de agradecimento texto de agradecimento texto de agradecimento texto de agradecimento texto de agradecimento texto de agradecimento texto de agradecimento texto de agradecimento texto de agradecimento texto de agradecimento texto de agradecimento texto de agradecimento texto de agradecimento texto de agradecimento texto de agradecimento.

Feliz, feliz, feliz... Estou tão feliz

Uma criança feliz

RESUMO

SILVA MONTENEGRO DOS SANTOS, Gabriella. *Análise do uso de corrotinas em linguagens estruturadas.* AAAA. 23 f. Trabalho de Conclusão de Curso (Graduação em Ciência da Computação) – Instituto de Matemática e Estatística, Universidade do Estado do Rio de Janeiro, Rio de Janeiro, AAAA.

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum. Sed ut perspiciatis unde omnis iste natus error sit voluptatem accusantium doloremque laudantium, totam rem aperiam, eaque ipsa quae ab illo inventore veritatis et quasi architecto beatae vitae dicta sunt explicabo. Nemo enim ipsam voluptatem quia voluptas sit aspernatur aut odit aut fugit, sed quia consequuntur magni dolores eos qui ratione voluptatem sequi nesciunt. Neque porro quisquam est, qui dolorem ipsum quia dolor sit amet, consectetur, adipisci velit, sed quia non numquam eius modi tempora incidunt ut labore et dolore magnam aliquam quaerat voluptatem. Ut enim ad minima veniam, quis nostrum exercitationem ullam corporis suscipit laboriosam, nisi ut aliquid ex ea commodi consequatur? Quis autem vel eum iure reprehenderit qui in ea voluptate velit esse quam nihil molestiae consequatur, vel illum qui dolorem eum fugiat quo voluptas nulla pariatur?

xxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxx teste de hiphenização: monumental bla bla bla

Palavras-chave: primeira palavra chave; segunda palavra chave; terceira palavra chave; quarta palavra chave (se houver).

ABSTRACT

SILVA MONTENEGRO DOS SANTOS, Gabriella. *Title of dissertation in English.* AAAA. 23 f. Trabalho de Conclusão de Curso (Graduação em Ciência da Computação) – Instituto de Matemática e Estatística, Universidade do Estado do Rio de Janeiro, Rio de Janeiro, AAAA.

Happiness deserves a English description. Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum. Sed ut perspiciatis unde omnis iste natus error sit voluptatem accusantium doloremque laudantium, totam rem aperiam, eaque ipsa quae ab illo inventore veritatis et quasi architecto beatae vitae dicta sunt explicabo. Nemo enim ipsam voluptatem quia voluptas sit aspernatur aut odit aut fugit, sed quia consequuntur magni dolores eos qui ratione voluptatem sequi nesciunt. Neque porro quisquam est, qui dolorem ipsum quia dolor sit amet, consectetur, adipisci velit, sed quia non numquam eius modi tempora incidunt ut labore et dolore magnam aliquam quaerat voluptatem. Ut enim ad minima veniam, quis nostrum exercitationem ullam corporis suscipit laboriosam, nisi ut aliquid ex ea commodi consequatur? Quis autem vel eum iure reprehenderit qui in ea voluptate velit esse quam nihil molestiae consequatur, vel illum qui dolorem eum fugiat quo voluptas nulla pariatur?

xxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxx Hyphenation test: monumental bla bla bla

Keywords: first keyword; second keyword; third keyword; fourth keyword (if any).

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

SUMÁRIO

INTRODUÇÃO

Corrotinas consistem em uma abstração de controle de execução que permite que um trecho de código seja suspenso e retomado posteriormente, tornando-as colaborativas por natureza. Apesar de ser uma técnica antiga, poucos conceitos formais foram apresentados ao longo do tempo.

Em sua tese de doutorado, (MARLIN, 1980) definiu duas características fundamentais das corrotinas: a persistência dos valores das variáveis locais entre chamadas sucessivas e a suspensão da execução quando o controle sai da corrotina, permitindo que esta seja retomada exatamente do ponto onde parou. No entanto, essa descrição corresponde à concepção comum sobre corrotinas, mas deixa em aberto questões relevantes sobre sua implementação e variações (MOURA; RODRIGUEZ; IERUSALIMSCHY, 2009). Os autores destacam três aspectos essenciais que diferenciam os tipos de corrotinas: o mecanismo de transferência de controle, que pode ser simétrico ou assimétrico; a forma como as corrotinas são disponibilizadas na linguagem, podendo ser objetos de primeira classe ou estruturas restritas; e se a implementação é stackful, ou seja, se permite a suspensão da execução dentro de chamadas aninhadas.

Com isso, diferentes implementações surgiram ao longo do tempo, adaptando o conceito de corrotinas às necessidades específicas de cada linguagem de programação. Algumas abordagens priorizam simplicidade na geração de valores, enquanto outras oferecem mecanismos mais flexíveis para troca de dados e controle de execução.

Em JavaScript e Python, as corrotinas são introduzidas meio dos geradores, que permitem suspender e retomar a execução do código, mas com um comportamento predominantemente orientado à produção de valores. Essa característica faz com que os geradores sejam frequentemente usados de forma unidirecional, onde o controle de execução avança passo a passo gerando novos valores. No entanto, embora sejam essencialmente produtores, os geradores podem também receber dados durante sua execução, tornando o fluxo de controle bidirecional quando estruturados para isso.

Por outro lado, em Lua, as corrotinas são implementadas seguindo um modelo assimétrico, no qual o controle de execução é explicitamente transferido entre as corrotinas de forma dinâmica e bidirecional. Além disso, ao contrário dos geradores de linguagens como JavaScript e Python, que são *stackless* e só podem ser suspensos no próprio escopo da função geradora, as corrotinas de Lua são *stackful*, permitindo a suspensão e retomada mesmo dentro de chamadas aninhadas.

Em geral, a capacidade de suspender e retomar a execução de uma função de forma controlada é particularmente útil em cenários que envolvem operações de I/O, onde o tempo de resposta pode variar significativamente. Ao permitir que a execução de uma tarefa seja pausada enquanto aguarda a conclusão de uma operação externa—como

a leitura de um arquivo ou a comunicação com um servidor—essa técnica evita o bloqueio da aplicação e melhora a eficiência do uso dos recursos disponíveis. Em modelos tradicionais baseados em threads, essa coordenação pode exigir mecanismos mais complexos, como bloqueios e filas de espera. Já com corrotinas e geradores, o fluxo de dados pode ser tratado de maneira mais natural e eficiente, garantindo que a execução progrida de forma assíncrona e cooperativa, sem a necessidade de alternância forçada entre diferentes unidades de processamento.

Objetivo

O objetivo deste trabalho é investigar o papel das corrotinas no contexto das linguagens de programação estruturadas, analisando como suas características impactam o controle de fluxo e a troca de dados. Através de uma análise comparativa e semântica nas diferentes implementações nativas de corrotinas nas linguagens estruturadas, busca-se entender como as corrotinas, com suas diferentes nuances em cada linguagem, podem facilitar ou não a implementação de problemas que envolvem o fluxo de dados.

Justificativa

O uso de corrotinas, apesar de ser um conceito antigo, é um tema pouco explorado tanto na literatura acadêmica quanto na prática de desenvolvimento de software, especialmente em linguagens estruturadas como Python, JavaScript e Lua. Embora existam implementações nativas nessas linguagens, o entendimento formal e a comparação das abordagens existentes são escassos, o que dificulta a avaliação de suas reais vantagens e limitações.

Este estudo busca, portanto, preencher essa lacuna, oferecendo uma análise detalhada das diferentes implementações de corrotinas em linguagens de programação. Ao investigar o impacto dessas técnicas em termos de modularidade, expressividade e simplicidade, espera-se fornecer uma base mais sólida para a compreensão do papel das corrotinas na resolução de problemas.

Metodologia

Organização da Dissertação

1 CONCEITOS BÁSICOS

2 GERADORES E CORROTINAS

Neste capítulo, são apresentadas as operações básicas e diferenças entre corrotinas e geradores, conforme suas implementações nativas nas linguagens em JavaScript e Lua.

2.1 Operações Básicas

2.1.1 Geradores

Os geradores possibilitam a suspensão do código com a palavra-chave `yield` e a retomada por meio do método `next()`. Este método pertence ao objeto retornado pela função geradora, permitindo continuar a execução a partir do ponto de interrupção.

O exemplo a seguir ilustra um gerador simples, no qual a execução é suspensa com `yield` e retomada progressivamente através de chamadas sucessivas ao método `next()` do objeto retornado pelo gerador. Inicialmente, a função geradora permanece suspensa e só é executada quando o primeiro `next()` é chamado quando é suspensa novamente ao encontrar a declaração `yield` e retomada posteriormente com a declaração do segundo `next()`.

```
function* gerador() {
    console.log("Inicio do gerador");
    yield;
    console.log("Retomando o gerador");
}

const gen = gerador();
gen.next(); // Executa ate o primeiro yield
gen.next(); // Retoma a execucao
```

No código acima, a palavra-chave `function*` define uma função geradora, que pode ser pausada e retomada. O operador `yield`, que atua como um ponto de suspensão da execução, apenas pode ser utilizado dentro de funções geradoras. Isso ocorre porque o JavaScript segue o modelo de *event loop*, executado em browsers web, onde a execução do código é baseada em um fluxo assíncrono (RAJANI et al., 2015), garantindo que uma tarefa só seja processada quando a anterior for concluída, sem bloquear a interface do usuário.

Além disso, os geradores são *First-Class Objects*, ou seja, podem ser atribuídos a variáveis, passados como argumentos para outras funções e retornados de funções. Essa

flexibilidade viabiliza padrões de controle de fluxo, como iteradores personalizados como o `for...of`.

2.1.2 Corrotinas

As corrotinas em Lua são manipuladas por meio da biblioteca nativa `coroutine`, que oferece algumas operações básicas como: interrupção do código com `coroutine.yield`, retomada com `coroutine.resume` e criação de corrotinas com `coroutine.create`.

Quando uma corrotina é criada, é alocada uma pilha separada para sua execução, garantindo o comportamento *stackful*. A função `coroutine.create` recebe como argumento o código a ser executado na corrotina, que pode ser definido por uma função nomeada ou anônima, e retorna uma referência à corrotina criada, que é um *First-Class Value*.

Uma corrotina é retomada com a função `coroutine.resume`, que recebe como argumento a referência da corrotina. Se a corrotina estiver suspensa, ela será retomada do ponto onde foi interrompida. Caso tenha finalizado sua execução, novas chamadas a `coroutine.resume` não terão efeito.

A suspensão de uma corrotina é feita utilizando `coroutine.yield`. Quando a corrotina executa essa chamada, ela é interrompida e retorna o controle ao código que a chamou. Isso permite que a corrotina seja suspensa em pontos específicos de sua execução e retome posteriormente de onde parou. Assim como geradores, a corrotina começa suspensa, apontando para a primeira declaração do seu corpo e só é executada quando chamada por `coroutine.resume`.

Para verificar o estado de uma corrotina, Lua fornece a função `coroutine.status`, que recebe como argumento a referência da corrotina e retorna uma das seguintes strings:

- `"running"`: indica que a corrotina está em execução.
- `"suspended"`: significa que a corrotina foi criada, mas ainda não iniciou sua execução, ou que foi interrompida por um `yield`.
- `"dead"`: indica que a corrotina concluiu sua execução ou encontrou um erro.

No código a seguir é apresentado o ciclo de vida de uma corrotina em Lua, passando pelos estados de criação, suspensão, retomada e finalização. Inicialmente, a corrotina é criada com a função `coroutine.create`, que recebe como argumento a função `corrotina`. No momento da criação, seu estado é `"suspended"`, indicando que ainda não foi executada.

Quando a corrotina é retomada pela primeira vez com `coroutine.resume(co)`, sua execução se inicia e a mensagem `"Início da corrotina"` é exibida. No entanto, ao atingir `coroutine.yield()`, sua execução é suspensa e o controle retorna ao código

principal, mantendo a corrotina no estado `"suspended"`. Isso significa que ela pode ser retomada posteriormente.

Na segunda chamada de `coroutine.resume(co)`, a corrotina continua sua execução a partir do ponto onde foi suspensa, imprimindo a mensagem `"Retomando a corrotina"`. Como não há mais instruções a serem executadas, a corrotina finaliza sua execução e seu estado passa a ser `"dead"`, indicando que não pode mais ser retomada.

```
function corrotina()
    print("Início da corrotina")
    coroutine.yield()
    print("Retomando a corrotina")
end

co = coroutine.create(corrotina)

print(coroutine.status(co)) -- "suspended"
coroutine.resume(co)        -- Executa até o yield
print(coroutine.status(co)) -- "suspended"
coroutine.resume(co)        -- Retoma a execução
print(coroutine.status(co)) -- "dead"
```

Embora `coroutine.create` e `coroutine.resume` sejam as formas mais comuns de manipular corrotinas, Lua também oferece a função `coroutine.wrap`, que simplifica esse processo. A principal diferença é que `coroutine.wrap` retorna uma função, em vez de uma referência direta para a corrotina. Essa função, quando chamada, automaticamente retoma a execução da corrotina, eliminando a necessidade de utilizar `coroutine.resume` explicitamente.

O exemplo a seguir reescreve o código anterior, eliminando a necessidade de utilizar `coroutine.create` e `coroutine.resume`. No entanto, as chamadas para verificar o status da corrotina foram removidas, pois `coroutine.status` exige uma referência de corrotina, enquanto `coroutine.wrap` retorna uma função.

```
co = coroutine.wrap(corrotina)

print(co()) -- "data1"
print(co()) -- "data2"
print(co()) -- "data3"
```

2.2 **Fluxo de Dados Unidirecional**

No fluxo de dados unidirecional, os valores são produzidos por uma função geradora ou corrotina e consumidos à medida que se tornam disponíveis ou sob demanda. Nesse modelo, o consumidor não influencia diretamente a produção dos dados, que ocorre de maneira independente.

O código a seguir demonstra um exemplo de produção de valores sob demanda de duas funções geradoras como produtoras. A função `producer` produz as strings "data1"e "data3", enquanto `producer_2` produz a string "data2". Para consumir os valores gerados pelas funções, é utilizado o método `next()`. O método next() retorna um objeto com duas proriedades:

- `value`: O valor produzido pelo `yield` na função geradora.
- `done`: Um valor booleano que indica se o gerador concluiu sua execução. Quando `done` é `true`, significa que o gerador não tem mais valores a produzir.

```
function* producer() {
    yield "data1";
    yield "data3";
}

function* producer_2() {
    yield "data2";
}

const prod = producer();
const prod2 = producer_2();

console.log(prod.next().value); // "data1"
console.log(prod2.next().value); // "data2"
console.log(prod.next().value); // "data3"
```

Portanto, o `next()` permite que os dados sejam consumidos sob demanda e de forma controlada, com alternância entre as funções geradoras retornando o valor gerado a partir da propriedade `value`.

Por outro lado, é possível consumir os valores gerados utilizando o `for...of`, que funciona como um iterador, chamando automaticamente o método `next()` de forma implícita até que o gerador seja finalizado. Isso permite iterar sobre os valores produzidos pela função geradora.

```
for (const data of prod) console.log(data);
```

Em Lua, com corrotinas, a implementação do código acima é semelhante, onde o valores produzidos são retornadas através de `coroutine.yield`.

```
function producer()
    coroutine.yield("data1")
    coroutine.yield("data3")
end

function producer_2()
    coroutine.yield("data2")
end

co1 = coroutine.create(producer)
co2 = coroutine.create(producer_2)

status, value = coroutine.resume(co1)  -- "data1"
print(value)

status, value = coroutine.resume(co2)  -- "data2"
print(value)

status, value = coroutine.resume(co1)  -- "data3"
print(value)
```

Em Lua, o método `coroutine.resume` retorna múltiplos valores. O primeiro valor indica se a execução da corrotina foi bem-sucedida, enquanto os valores subsequentes correspondem ao que foi produzido pela corrotina ou mensagens de erro, caso tenham ocorrido. No exemplo acima, a variável status armazena o `status` do corrotina e a variável `value` corresponde ao dado gerado pelo `coroutine.yield`.

Há duas abordagens diferentes para iteração sobre valores produzidos por uma corrotina. A primeira abordagem usando `coroutine.resume`, exige chamadas maunais e controle explícito do estado da corrotina. Como `coroutine.resume` retorna um valor por vez, é necessário continuar chamando a função até que a corrotina termine sua execução. Para um loop de iteração, pode-se verificar o estado da corrotina com `coroutine.status` antes de continuar a execução.

```
co = coroutine.create(producer)
```

```
while coroutine.status(co) ~= "dead" do
    local status, value = coroutine.resume(co)
    if status then
        print(value)
    end
end
```

A segunda abordagem, usando `coroutine.wrap` simplifica esse processo ao permitir chamadas diretas à função retornada, sem precisar gerenciar `coroutine.status` ou lidar com múltiplos valores de retorno. Essa abordagem é semelhante à iteração em geradores, com onde o loop `for...in` simula sucessivas chamadas implícitas da função `croutine.resume`.

```
co1 = coroutine.wrap(producer)

for data in co1 do
    print(data)
end
```

2.3 **Fluxo de Dados Bidirecional**

Embora os geradores sejam, por essência, mecanismos unidirecionais de produção de dados, geradores em JavaScript permite que valores sejam enviados de volta para o gerador por meio do método `next(value)`. Essa funcionalidade adiciona um grau de interação ao fluxo de dados, permitindo que o gerador receba informações externas e ajuste seu comportamento com base nelas. No entanto, é importante destacar que essa interação ainda ocorre dentro de um ciclo bem definido:

- O gerador produz um valor com `yield`.
- A execução do gerador pausa até que `next(value)` seja chamado.
- O valor pasado para `next(value)` se torna o resultado da expressão `yield`, permitindo que o gerador reaja a ele.

A implementação a seguir ilustra esse cenário:

```
function* transformSequence(a) {
```

```
    let b = yield a * 3;
    return b - 1;
}

const co = transformSequence(10);
```

Quando a função geradora é declarada com argumentos de entrada, por exemplo `const co = transformSequence(10)`, os argumentos são passados após a ativação da função geradora, através da declaração `let c = co.next().value`. Com isso, o gerador inicia a com o valor 10 e executa até yield a * 3, quando sua execução é suspensa e o valor do yield ao chamador. Dessa forma, o valor 30 (a * 3) é recebida pela atribuição de `let c = co.next().value`, onde value possui o valor retornado pelo `yield`. Quando o gerador é ativado novamente com `let d = co.next(c + 2)`, o argumento de next é recebido pela função `yield`, logo a variável local `b` recebe 32 (c + 2).

Por fim, ao final da execução do gerador, quando a instrução `return b - 1` é atingida, o gerador retorna 31 (`b - 1`). Diferente do `yield`, que retorna um objeto contendo `value`, o `return` finaliza o gerador e retorna diretamente o resultado 31 (b - 1), sem a necessidade de acessar a propriedade `value`. Assim, a atribuição `let d = co.next(c + 2)` recebe diretamente 31, sem precisar utilizar `.value`.

Com corrotinas, a sintaxe do corpo da função transformSequence e o comportamento é semelhante. A criação da corrotina é feita por meio de `coroutine.create(transformSequen` , que recebe a função como argumento e retorna a referência da corrotina.

Assim como acontece em geradores com `const co = transformSequence(10)`, que apenas define o gerador sem ativá-lo, a corrotina não executa.

```
function transformSequence(a)
    print("Created")
    local b = coroutine.yield(a * 3)
    return b - 1
end

co = coroutine.create(transformSequence)
```

A declaração `success, c = coroutine.resume(co, 10)` ativa a corrotina criada, iniciando sua execução e passando 10 como argumento para a corrotina `transformSequence`. Quando a execução encontra `coroutine.yield(a * 3)`, a corrotina é suspensa, retornando o valor 30 (a *3) ao chamador. A função `coroutine.resume()` retorna dois valores:

um sinal booleano e o valor gerado pelo yield. O booleano true indica que a execução foi bem-sucedida e que a corrotina ainda pode ser retomada.

Quando a corrotina é reativa com a declaração `success, d = coroutine.resume(co, c + 2)`, o valor 32 (c + 2) é passado como argumento para a variável local b da corrotina. Em seguida, ao encontrar o return, assim como nos geradores, o valor 31 (b - 1) é retornado e a corrotina é finalizada.

Outra forma de implementar esse exemplo com corrotinas é utilizar o `coroutine.wrap`. Essa função simplifica a manipulação das corrotinas ao encapsular a criação com e execução dentro de uma única função. O exemplo abaixo ilustra a refatoração que ao invés de usar coroutine.create e coroutine.resume, utiliza apenas coroutine.wrap.

```
co = coroutine.wrap(function(a)
    local b = coroutine.yield(a * 3)
    return b - 1
end)
c = co(10)
d = co(b + 1)
```

Dessa forma, uma função anônima é passada diretamente para `coroutine.wrap` que retorna uma função que pode ser chamada normalmente, sem precisar utilizar `coroutine.create` e `coroutine.resume`. Essa abordagem simplifica a sintaxe e evita a necessidade de lidar manualmente com a verificação de status da corrotina, como ocorre ao usar `coroutine.resume`, tornando o código mais direto e legível.

2.4 **Chamadas Aninhadas e Controle de Execução**

CONCLUSÃO

REFERÊNCIAS

MARLIN, C. D. Coroutines: a programming methodology, a language design and an implementation. *Springer Science Business Media*, 1980.

MOURA, A. L. de; RODRIGUEZ, N.; IERUSALIMSCHY, R. Revisiting coroutines. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, v. 31, p. 1–31, 2009.

RAJANI, V. et al. Information flow control for event handling and the dom in web browsers. *2015 IEEE 28th Computer Security Foundations Symposium*, p. 366–379, 2015.