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. 32 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. 32 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, sem bloquear o programa principal, 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. Em Lua, por exemplo, corrotinas são verdadeiras abstrações de controle, com suporte à suspensão e retomada da execução por meio de funções distintas (`coroutine.yield` e `coroutine.resume`), permitindo que a execução seja interrompida até mesmo dentro de funções aninhadas devido à sua implementação *stackful*. Já em linguagens como JavaScript e Python, o que se tem são estruturas mais restritas conhecidas como geradores, que compartilham algumas características com corrotinas — como a preservação do estado entre execuções —, mas impõem limitações importantes: a suspensão apenas ocorre diretamente no corpo da função geradora. Essas restrições impedem, por exemplo, a suspensão da execução a partir de funções auxiliares, caracterizando os geradores como formas limitadas de corrotinas, também chamadas de semi-corrotinas.

Apesar dessas distinções, tanto as corrotinas de Lua quanto os geradores de linguagens como Python e JavaScript compartilham uma característica comum: são mecanismos de controle assimétricos. Isso significa que a função responsável por suspender a execução é diferente daquela que a retoma — no caso de Lua, por exemplo, usam-se explicitamente `coroutine.yield` e `coroutine.resume`, enquanto em Python a suspensão ocorre com `yield` e a retomada com `next()`. Em ambos os casos, o fluxo de controle é unidirecional: a execução sempre retorna ao ponto de onde foi chamada, ao contrário do que ocorre em abordagens simétricas, nas quais qualquer corrotina pode transferir o controle diretamente para outra.

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 entrada e saída de dados, onde o tempo de resposta varia significativamente. Ao permitir a pausa na execução de uma tarefa durante a espera pela 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 no uso dos recursos disponíveis. Com o uso de corrotinas e geradores, o fluxo de dados passa a ser tratado de maneira mais natural e eficiente, e a execução ocorre de forma assíncrona e cooperativa, sem a necessidade de alternar forçadamente entre diferentes unidades de processamento, como callbacks ou sistemas de agendamento preemptivo.

Abordagens assíncronas, como Async/Await do Javascript, por exemplo, são implementações derivadas de corrotinas (RAUSCHMAYER, 2015), onde o fluxo de execução é interrompido e retomado conforme necessário, sem bloquear a execução do restante do programa. Corrotinas, no entanto, possuem uma vantagem distinta: oferecem maior controle sobre o fluxo de execução, no qual os desenvolvedores definem explicitamente pontos de suspensão e retomada.

Dessa forma, este trabalho busca compreender o papel das corrotinas no contexto das linguagens de programação estruturadas, por meio de um estudo comparativo que destaca seus potenciais e limitações, com foco nos geradores em JavaScript e nas corrotinas em Lua. Para complementar, são apresentados cenários práticos com fluxo de dados, a fim de mostrar como as corrotinas oferecem uma maneira mais clara, modular e eficiente de controlar a execução.

Organização da Dissertação

1 REVISÃO DA LITERATURA

2 OPERAÇÕES FUNDAMENTAIS E CARACTERÍSTICAS

Esta seção apresenta uma análise acerca das operações fundamentais relacionadas ao uso de corrotinas e geradores, com foco na criação, suspensão e retomada da execução. Também discute o papel dos iteradores, os mecanismos de entrada e saída de dados e as limitações impostas por cada abordagem. Ao destacar as nuances de cada abordagem, busca-se evidenciar como cada implementação lida com o fluxo de controle e com a troca de informações.

2.1 Operações Básicas

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* generator() {
    console.log("Init");
    yield;
    console.log("Resume");
}

const gen = generator();
gen.next();
gen.next();
```

No exemplo apresentado, a função `generator` é declarada como uma função geradora por meio da palavra-chave `function*`, o que permite o uso do operador `yield` para pausar e retomar sua execução. Ao executar `const gen = generator()`, a função não é imediatamente executada; em vez disso, retorna um objeto gerador, armazenado na variável `gen`. Esse comportamento é possível porque funções geradoras são *First-Class Objects*, ou seja, podem ser atribuídas a variáveis, armazenadas em estruturas de dados, passadas como argumentos ou retornadas de outras funções.

A execução da função se inicia somente quando `gen.next()` é chamado. Nesse momento, a linha `console.log("Init")` é executada, imprimindo "Init" no console, e a execução é imediatamente suspensa no ponto em que o `yield` aparece. Ao chamar `gen.next()` novamente, o gerador retoma sua execução exatamente de onde parou, passando para a próxima instrução, que imprime "Resume" no console.

Em contrapartida, 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 `coroutine.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 `coroutine`. 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 coroutine()
    print("Init")
    coroutine.yield()
    print("Resume")
end

co = coroutine.create(coroutine)

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 Produção de Dados

Geradores e corrotinas permitem a produção de dados sob demanda, emitindo valores de forma controlada e progressiva. Cada chamada ativa a função até um ponto de suspensão, onde um valor é produzido e retornado ao consumidor. Esse modelo torna possível gerar sequências de dados de maneira mais eficiente e estruturada, especialmente em fluxos onde o ritmo da produção não depende diretamente do consumo.

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 **Consumo de Dados**

Apesar de serem comumente associados à produção de valores, geradores também permitem receber dados externos durante sua execução. No JavaScript, com geradores, isso é feito por meio do método `next(value)`, que envia um valor de volta ao ponto em que o gerador foi suspenso. Essa funcionalidade transforma o gerador em um mecanismo bidirecional, permitindo que ele ajuste seu comportamento com base em valores recebidos dinamicamente. O processo segue um ciclo bem definido:

- O gerador emite um valor por meio da instrução `yield`.
- A execução da função pausa até a próxima chamada de `next(value)`.
- O valor fornecido para `next(value)` assume o papel de resultado da expressão `yield`, o que permite ao gerador adaptar seu comportamento com base nesse valor.

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 são semelhantes. A criação da corrotina é feita por meio de `coroutine.create(transformSequ` , 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 **Delegação de Execução**

Tanto em geradores quanto em corrotinas, a transferência de controle ocorre de forma assimétrica: uma rotina suspende sua execução em um determinado ponto (por meio de operadores como yield ou instruções como coroutine.yield), e a retomada sempre retorna ao chamador original, que detém o controle externo da execução. Essa característica implica que, para que uma rotina interaja com outra, é necessário algum tipo de mediação — seja repassando valores manualmente entre geradores, como em JavaScript, ou realizando chamadas explícitas de retomada entre corrotinas, como em Lua. No exemplo a seguir, observa-se como a delegação de execução entre duas funções geradoras é realizada:

```
function* callee() {
     while (true) {
```

```
        console.log('callee: ' + (yield));
    }
}

function* caller() {
    const co_callee = callee();
    co_callee.next();

    while (true) {
        const input = yield;
        co_callee.next(input);
    }
}

const co_caller = caller();
co_caller.next();
co_caller.next('a');
co_caller.next('b');
```

A função `caller` é encapsulada como uma corrotina por meio da chamada à função geradora `caller()`, cuja execução se inicia com `co_caller.next()`. Essa chamada avança a execução de `caller` até a primeira expressão `yield`, onde ela é automaticamente suspensa, retornando o controle ao escopo principal. Em seguida, ao chamar `co_caller.next('a')`, o valor `'a'` é passado para a variável `input` dentro de `caller`, que, por sua vez, o encaminha para a geradora `callee` através de `co_callee.next(input)`.

A função `callee`, que já havia sido ativada anteriormente com uma chamada inicial `co_callee.next()`, encontra-se suspensa no primeiro `yield`, pronta para receber um valor. Ao ser retomada com `input`, o valor `'a'` é atribuído à expressão `(yield)`, o que resulta na execução de `console.log("callee: a")`.

Esse mesmo fluxo se repete com a chamada `co_caller.next('b')`, na qual o valor `'b'` é novamente encaminhado a `callee`, que imprime `"callee: b"` no console. Nesse modelo, a função `caller` atua como um intermediário entre o código externo e a função `callee`, coordenando a troca de valores de forma manual.

Essa forma de delegação, embora viável, exige o repasse explícito de valores entre os geradores. Com o ECMAScript 6, foi introduzido o operador `yield*`, que possibilita a delegação automática da iteração para outro gerador, tornando o fluxo mais direto e legível. O exemplo anterior pode ser reescrito utilizando essa construção:

```
function* caller(){
    while(true){
        yield* callee();
    }
}
```

Com o uso de `yield* callee()`, a função `caller` repassa o controle diretamente para `callee`. Dessa forma, qualquer valor enviado a `caller` por meio de `next()` é automaticamente transmitido para `callee`, eliminando a necessidade de intermediários. A execução permanece delegada até que `callee` seja finalizada ou pausada.

Em contraste, Lua não requer um operador especial como `yield*` para esse tipo de delegação. Por ser baseada em corrotinas stackful, a linguagem permite transferir o controle de execução diretamente entre corrotinas usando `coroutine.resume`. Cada corrotina mantém sua própria pilha de chamadas, o que torna possível suspender e retomar a execução de forma transparente, preservando seu contexto. Assim, a comunicação entre corrotinas se dá de forma direta: argumentos são enviados por `resume` e valores são recuperados via `yield`, sem a necessidade de abstrações adicionais. A implementação equivalente em Lua mantém estrutura semelhante à versão explícita em JavaScript, mas a troca de controle entre corrotinas acontece de maneira mais natural, dispensando operadores como `yield*`.

```
function callee()
    while true do
        print("callee: " .. coroutine.yield())
    end
end

function caller()
    local co_callee = coroutine.create(callee)
    coroutine.resume(co_callee)  -- inicia callee e pausa no primeiro \\ yi

    while true do
        local input = coroutine.yield()
        coroutine.resume(co_callee, input)
    end
end

-- Cria a corrotina chamadora
local co_caller = coroutine.create(caller)
```

```
-- Inicia a execução de caller
coroutine.resume(co_caller)       -- avança até o yield da caller

coroutine.resume(co_caller, 'a')
coroutine.resume(co_caller, 'b')
```

A função `caller` é encapsulada como uma corrotina por meio da chamada `coroutine.create(caller)`, onde sua execução começa através da declaração `coroutine.resume(co_caller)`. Com isso, a execução avança até o `yield` interno de `caller`, no que resulta na pausa de sua execução e o controle retorna ao programa principal. Em seguida, ao chamar `coroutine.resume(co_caller, 'a')`, o valor `'a'` é passado para a variável `input` e, então, encaminhado para a corrotina `callee` por meio de `coroutine.resume(co_callee, input)`.

A corrotina `callee`, que já foi iniciada e pausada em seu primeiro `yield`, retoma sua execução e imprime `"callee: a"` no console. Diferente do exemplo em JavaScript, onde a delegação é feita por `yield*`, em Lua essa delegação ocorre de forma explícita dentro da função `caller`. Como `callee` precisa continuar ativa após cada retomada, utiliza-se um laço `while true`, o que evita que sua execução termine. Assim, a chamada `coroutine.resume(co_caller, 'b')` também alcança `callee`, que recebe o valor `'b'` e imprime `"callee: b"`.

A delegação de execução em JavaScript e Lua ilustra abordagens distintas para composição de rotinas cooperativas. Enquanto o ECMAScript introduz o operador `yield*` como uma forma de simplificar a transmissão de controle entre geradores, Lua não necessita de tal mecanismo devido ao modelo *stackful* de corrotinas, que permite a transferência direta de controle por meio de `coroutine.resume`, preservando o contexto de execução e possibilitando chamadas aninhadas sem a necessidade de abstrações adicionais.

2.5 **Iteradores**

Além de permitir a delegação de execução entre funções geradoras, como discutido na seção anterior, o operador `yield*` em JavaScript também serve como um mecanismo iterador. Quando usado dessa forma, ele emite cada valor individualmente, como se fossem produzidos por chamadas sucessivas a `yield`. Esse comportamento possibilita combinar múltiplos geradores de maneira mais direta e legível, reduzindo a necessidade de estruturas de controle adicionais como o loop `for...of`. O exemplo a seguir demonstra esse uso, em que dois geradores produzem números inteiros e um terceiro os integra por meio de `yield*`, resultando em um iterador composto que unifica os valores dos dois geradores.

```
function* numbers1to3() {
    yield 1;
    yield 2;
    yield 3;
  }

  function* numbers4to6() {
    yield 4;
    yield 5;
    yield 6;
  }

  function* composableIterator() {
    yield* numbers1to3();
    yield* numbers4to6();
  }

  const resultado = [...composableIterator()];
  console.log(resultado);
```

A função `composableIterator` é definida como uma função geradora através da declaração `function*`, e sua execução é iniciada implicitamente ao se aplicar o operador de espalhamento `[...]` sobre sua invocação. Esse operador consome todos os valores produzidos pela iteração do gerador, que avança sua execução até a conclusão.

No corpo de `composableIterator`, a delegação de iteração é realizada por `yield* numbers1to3()`, o que repassa o controle ao gerador `numbers1to3`. Esse gerador, por sua vez, produz os valores `1`, `2` e `3`, que são emitidos como se fossem diretamente gerados por `composableIterator`. Após a conclusão de `numbers1to3`, o controle é novamente delegado a `numbers4to6` por meio de outra chamada a `yield*`, a qual produz os valores `4`, `5` e `6` da mesma maneira.

Ao final, o operador `[...]` coleta todos os valores produzidos em ordem e os armazena no array `resultado`, resultando na saída `[1, 2, 3, 4, 5, 6]`.

Diferentemente do JavaScript, Lua não possui um operador equivalente ao `yield*` que permita iteração automática sob valores produzidos por outra corrotina, logo a composição de iteradores em Lua precisa ser realizada manualmente, através de laços explícitos para repassar os valores produzidos. No exemplo equivalente em Lua, utiliza-se a função `coroutine.wrap` para encapsular a corrotina composta como uma função iterável, permitindo retornar os valores de forma contínua.

```
function numbers1to3()
    return coroutine.wrap(function()
        coroutine.yield(1)
        coroutine.yield(2)
        coroutine.yield(3)
    end)
end

function numbers4to6()
    return coroutine.wrap(function()
        coroutine.yield(4)
        coroutine.yield(5)
        coroutine.yield(6)
    end)
end

function composableIterator()
    return coroutine.wrap(function()
        for n in numbers1to3() do
            coroutine.yield(n)
        end
        for n in numbers4to6() do
            coroutine.yield(n)
        end
    end)
end

local resultado = {}
for valor in composableIterator() do
    table.insert(resultado, valor)
end

for i, v in ipairs(resultado) do
    print(v)
end
```

A função `composableIterator` é construída como uma corrotina encapsulada por `coroutine.wrap`, o que permite tratá-la como uma função iteradora. Esse encapsula-

mento converte a corrotina em uma função que, a cada chamada subsequente, retoma sua execução a partir do ponto onde foi anteriormente suspensa, retornando sucessivamente os valores produzidos por meio de `coroutine.yield`. Quando essa função é utilizada em um laço `for`, a chamada e retomada da corrotina ocorrem de forma implícita, o que permite seu uso como um iterador convencional.

No corpo da corrotina, a composição da iteração é feita de forma manual. Os iteradores `numbers1to3` e `numbers4to6` também são definidos como corrotinas envoltas em `coroutine.wrap`, retornando funções iteradoras que produzem valores de 1 a 3 e de 4 a 6, respectivamente. Cada um desses iteradores é percorrido por um laço `for`, e seus valores são explicitamente repassados à corrotina principal por meio de chamadas a `coroutine.yield`. Assim, embora Lua não ofereça um operador como `yield*`, que delegue automaticamente a execução e a produção de valores a outro gerador, essa delegação é simulada por meio do encadeamento controlado de chamadas a iteradores dentro de laços de repetição.

Com essa construção, os valores produzidos por `numbers1to3` e `numbers4to6` são emitidos em sequência como se fossem gerados diretamente pela corrotina principal. Ao final, o laço externo `for`, que percorre `composableIterator`, coleta todos esses valores e os armazena na tabela `resultado`, produzindo a sequência `1`, `2`, `3`, `4`, `5` e `6`.

Essa comparação evidencia como o operador `yield*` em JavaScript atua não apenas como um mecanismo de delegação de controle entre geradores, mas também como um iterador que propaga valores de forma contínua e transparente. Ao integrar os valores emitidos por outros geradores, ele transforma a função geradora em um produtor composto, capaz de emitir uma sequência unificada de elementos sem necessidade de laços adicionais ou controle explícito da iteração. Assim, o `yield*` reforça uma das principais características conceituais dos geradores: sua função como produtores de fluxos de dados, construídos sob demanda, com controle refinado da emissão de cada valor.

2.6 **Limitações**

Embora funções geradoras em JavaScript permitam a suspensão e retomada de execução por meio do operador `yield`, essa suspensão é restrita ao escopo da própria função geradora, ou seja, apenas chamadas diretas a `yield` dentro do corpo de uma `function*` são válidas. Tentativas de utilizar `yield` em funções auxiliares ou `callbacks`, como os passados para métodos como `forEach` ou `map`, resultam em erro, pois essas funções não são geradoras e não compartilham o mesmo contexto de execução suspensível.

(RAUSCHMAYER, 2015) relaciona essa limitação ao fato de que geradores são considerados *shallow coroutines*: eles apenas suspendem a ativação atual da função, sem propagar a suspensão para o restante da pilha de chamadas. Ainda de acordo com o autor,

essa limitação traz benefícios importantes: por restringirem a suspensão a um único nível, os geradores se tornam compatíveis com o modelo de multitarefa cooperativa baseado em *event loop*, utilizado pelos navegadores, na qual uma tarefa é executada até o fim antes de outra iniciar.

Essa limitação se manifesta em tentativas comuns de uso, como ao empregar `yield` dentro de funções de `callback`. Por exemplo, ao iterar sobre um array com `forEach`, pode-se imaginar que seria possível utilizar `yield` para pausar a execução. No entanto, isso resulta em erro de sintaxe, pois o `callback` passado para `forEach` não é uma função geradora e, portanto, não compartilha o contexto suspensível da função principal:

```
function* gerador() {
    [1, 2, 3].forEach((n) => {
        yield n; // SyntaxError
    });
}
```

Esse comportamento reforça a ideia de que os geradores operam apenas sobre sua própria ativação, sem possibilidade de propagar a suspensão para outras funções na pilha. Como consequência, é necessário reescrever a lógica usando estruturas compatíveis com `yield`, como o `for...of`, que permite suspender a execução diretamente no corpo da função geradora:

```
function* gerador() {
    for (const n of [1, 2, 3]) {
        yield n;
    }
}
```

Um exemplo análogo ocorre em Lua, embora por outro motivo. Segundo os autores (MOURA; RODRIGUEZ; IERUSALIMSCHY, 2004), a linguagem foi projetada, desde sua origem, para funcionar como uma linguagem de extensão leve, facilmente integrada a programas escritos em C, C++ e outras linguagens convencionais. Para isso, a linguagem oferece uma biblioteca de funções em C que, junto ao interpretador de Lua, permite que o programa hospedeiro se comunique com o ambiente Lua por meio de chamadas intercaladas entre Lua e C. Nesse contexto, como as funções em C não preservam o estado da pilha de execução, sua suspensão se torna inviável. Por esse motivo, Lua impõe a restrição de que uma corrotina não pode realizar yield enquanto houver uma função C ativa na pilha.

Essa limitação pode ser observada, por exemplo, ao tentar usar yield dentro de uma função de callback passada para table.sort, que é implementada em C. O trecho abaixo resultaria no erro *attempt to yield across a C-call boundary*.

```
co = coroutine.create(function()
  table.sort({3, 2, 1}, function(a, b)
    coroutine.yield(a < b)
    return a < b
  end)
end)

print(coroutine.resume(co))
```

Entretanto, diferentemente de JavaScript, em Lua é possível utilizar `yield` dentro de funções auxiliares escritas em Lua, desde que essas funções façam parte da pilha da corrotina ativa. Isso ocorre porque o sistema de corrotinas de Lua permite a suspensão em qualquer ponto da pilha de chamadas enquanto a execução permanecer no lado da linguagem Lua. Ou seja, mesmo que o `yield` esteja em uma função chamada indiretamente pela função principal da corrotina, a suspensão ainda será bem-sucedida, desde que nenhuma função C esteja intermediando a chamada.

Assim, tanto em Lua quanto em JavaScript, a possibilidade de suspensão está diretamente atrelada ao ambiente de execução no qual o `yield` é chamado — qualquer rompimento dessa continuidade compromete o mecanismo.

3 ESTUDOS DE CASO

3.1 Leitura de Arquivo

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. Coroutines in lua. *Journal of Universal Computer Science*, v. 10, n. 7, p. 910–925, 2004. ISSN 0021-9991.

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

RAUSCHMAYER, D. A. Exploring es6: Upgrade to the next version of javascript. *Learnpub*, 2015.