PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO DE JANEIRO

Luagrapheme

Uma biblioteca Lua para segmentação de texto Unicode

Gabriel Vianna Soldani

PROJETO FINAL DE GRADUAÇÃO

CENTRO TÉCNICO CIENTÍFICO – CTC

DEPARTAMENTO DE INFORMÁTICA

Curso de Graduação em Ciência da Computação

Rio de Janeiro, novembro de 2025.

Gabriel Vianna Soldani

Luagrapheme

Uma biblioteca Lua para segmentação de texto Unicode

Relatório de Projeto Final, apresentado ao programa de Graduação do Departamento de Informática da PUC-Rio como requisito parcial para a obtenção do título de Bacharel em Ciência da Computação.

Orientadores: Roberto Ierusalimschy
Luís Fernando Teixeira Bicalho

Rio de Janeiro
novembro de 2025.

Agradecimentos

Aos orientadores Roberto Ierusalimschy e Luís Fernando Bicalho, agradeço o estímulo, direcionamento, parceria e confiança ao longo do desenvolvimento deste trabalho.

À banca examinadora, pelo tempo dedicado à leitura e pelas contribuições que ajudaram a aprimorar o trabalho.

Ao Departamento de Informática e a todos os seus professores, pela contribuição para a minha formação, pelo ambiente de aprendizado e pela dedicação ao ensino.

À Vice-Reitoria Comunitária, na pessoa do professor Augusto Sampaio (*in memoriam*), cuja generosidade e compromisso com o acesso à educação tornaram possível a bolsa que me permitiu estudar nesta universidade.

À minha família, pelo apoio incondicional, pela paciência e por me ensinarem o valor da educação e da persistência.

À Luciana, minha psicóloga, pelo acolhimento e pela ajuda fundamental para manter o equilíbrio e a clareza ao longo do caminho.

À Marina, minha companheira, pelo carinho, pela paciência e por me acompanhar com amor e leveza em todas as etapas desta jornada.

Resumo

Vianna Soldani, Gabriel. Ierusalimschy, Roberto; Teixeira Bicalho, Luís Fernando. Luagrapheme: Uma biblioteca Lua para segmentação de texto Unicode. Rio de Janeiro, 2025. 47 p. Relatório de Projeto Final II – Departamento de Informática. Pontifícia Universidade Católica do Rio de Janeiro.

luagrapheme é uma biblioteca para segmentação de textos Unicode na linguagem Lua. Seu objetivo é identificar com precisão unidades textuais como *grapheme clusters* (caracteres visuais percebidos pelos usuários), palavras, frases e oportunidades de quebras de linha. O projeto consiste em *bindings* para a *libgrapheme*, biblioteca minimalista em C que implementa fielmente os algoritmos definidos pelo padrão Unicode. O *luagrapheme* oferece integrações com a biblioteca *LPeg*, permitindo a escrita e reconhecimento de padrões e gramáticas baseadas em Parsing Expression Grammars (PEGs) que incorporam os algoritmos de segmentação do Unicode. A metodologia combinou desenvolvimento em Lua e C, com testes automatizados e integração contínua. O resultado é uma biblioteca pequena, previsível e portátil, que amplia o suporte Unicode no ecossistema Lua e torna o processamento de texto multilíngue mais acessível a desenvolvedores.

Palavras-chave

Unicode; Lua; LPeg; Parsing Expression Grammars; *grapheme clusters*; *case folding*.

Abstract

Vianna Soldani, Gabriel. Ierusalimschy, Roberto; Teixeira Bicalho, Luís Fernando. Luagrapheme: A Lua library for Unicode text segmentation. Rio de Janeiro, 2025. 47 p. Undergraduate Thesis II – Departamento de Informática. Pontifícia Universidade Católica do Rio de Janeiro.

luagrapheme is a library for Unicode text segmentation in the Lua programming language. Its goal is to accurately identify textual units such as grapheme clusters (the visual characters perceived by users), words, sentences, and line break opportunities. The project consists of bindings for *libgrapheme*, a minimalist C library that faithfully implements the algorithms defined by the Unicode Standard. *luagrapheme* provides integration with the *LPeg* library, enabling the definition and recognition of patterns and grammars based on Parsing Expression Grammars (PEGs) that incorporate Unicode segmentation algorithms. The methodology combined development in Lua and C, supported by automated testing and continuous integration. The result is a small, predictable, and portable library that enhances Unicode support in the Lua ecosystem and makes multilingual text processing more accessible to developers.

Keywords

Unicode; Lua; LPeg; Parsing Expression Grammars; *grapheme clusters*; *case folding*.

Lista de figuras

Sumário

Introdução

Historicamente, o texto em computadores foi inicialmente representado por meio do padrão ASCII, uma codificação simples e suficiente para a língua inglesa. Logo, porém, surgiram diversas extensões e alternativas, cada uma tentando acomodar idiomas adicionais, mas de forma fragmentada e incompatível entre si. O Unicode consolidou essa diversidade em um padrão unificado, capaz de representar todos os sistemas de escrita conhecidos do mundo (The Unicode Consortium, 2023c). Essa unificação, no entanto, trouxe consigo um novo desafio: segmentar corretamente o texto — reconhecer os limites de unidades de texto significativas, como caracteres, palavras e frases (The Unicode Consortium, 2023b).

No padrão Unicode, existe o conceito de *code point* – uma abstração destinada a representar cada caractere único dentro do vasto repertório de sistemas de escrita. Em português, e em muitos outros idiomas que utilizam o alfabeto latino, a correspondência entre o que consideramos um caractere e o respectivo *code point* Unicode é, na maioria das vezes, direta. Já em outros idiomas e sistemas de escrita, um único caractere do ponto de vista do usuário — chamado pelo Unicode de *grapheme cluster* — pode resultar da combinação de uma sequência de *code points*. Essa característica expõe uma camada adicional de complexidade no processamento de texto. A partir de extensas tabelas de propriedades, o Unicode Annex #29 (The Unicode Consortium, 2023b) define algoritmos para segmentar de forma correta sequências de *code points* em *grapheme clusters*, palavras, frases e oportunidades de quebras de linha.

A segmentação correta de texto é fundamental para garantir que aplicações se comuniquem efetivamente com usuários globais, respeitando a diversidade linguística e cultural. No layout de texto, por exemplo, é essencial para determinar pontos adequados para quebras de linha. Em contextos em que a entrada de texto é limitada em tamanho, a contagem precisa de caracteres assegura uma boa experiência de uso. Na edição de texto, a capacidade de navegar, selecionar e manipular texto baseado em unidades significativas - como caracteres, palavras, frases ou linhas - depende diretamente de uma segmentação de texto intuitiva e sensível às particularidades do idioma.

O *luagrapheme* surge da constatação, a partir de notas da comunidade (Lua Users, 2018) e dos trabalhos relacionados (seção 2), de que, no ecossistema Lua, faltava uma biblioteca pequena e objetiva para cobrir as necessidades básicas de

segmentação de texto Unicode. O projeto não se propõe a reproduzir o escopo amplo da International Components for Unicode (ICU), desenvolvida pelo próprio Unicode Consortium (1999), mas a oferecer uma alternativa leve que ofereça os algoritmos de *case folding* e de segmentação em *grapheme clusters*, palavras, frases e oportunidades de quebras de linha, através de *bindings* para a *libgrapheme*, uma biblioteca em C já estabelecida para este propósito. O desenvolvimento priorizou práticas de qualidade de software, como Test-Driven Development (TDD) e análise estática, e também a integração com ferramentas do ecossistema Lua, como o *LPeg* — uma biblioteca de análise sintática amplamente utilizada (LuaRocks, 2014) para o reconhecimento de padrões.

Como resultado, o *luagrapheme* disponibiliza uma API idiomática e previsível, compatível com múltiplas versões de Lua, e com tamanho e complexidade significativamente menor em relação à ICU, permitindo a criação de aplicações multilíngues corretas e portáveis. Sua implementação representa uma contribuição prática ao ecossistema Lua, fortalecendo as bases para o desenvolvimento de aplicações compatíveis com uma ampla variedade de idiomas e sistemas de escrita através do padrão Unicode.

Este relatório detalha o desenvolvimento do *luagrapheme*, explorando desde a concepção inicial até a implementação e avaliação dos resultados. O texto está estruturado nas seguintes seções: Base teórica, que oferece um panorama sobre o Unicode necessário para o entendimento da proposta; Situação atual e trabalhos relacionados, que discute como linguagens, bibliotecas e ferramentas tratam texto Unicode, em especial no ecossistema Lua, situando o *luagrapheme* nesse contexto; Projeto e implementação, que descreve as decisões técnicas, detalha os requisitos funcionais e não-funcionais, exemplifica as funcionalidades da biblioteca e traz uma referência da API; Metodologia, que detalha a abordagem de desenvolvimento adotada; e, por fim, Considerações finais, onde são discutidas as contribuições do projeto, aprendizados obtidos e possibilidades futuras.

1 Base teórica

Para detalharmos a proposta do *luagrapheme*, é essencial compreender alguns conceitos padronizados pelo Unicode. Este entendimento é crucial para reconhecer a complexidade inerente à representação e processamento de textos em uma diversidade global de idiomas e sistemas de escrita — a mesma complexidade que o projeto busca abordar e simplificar.

O Unicode é uma peça fundamental para a comunicação global na era digital, estabelecendo-se como o padrão universal de representação de texto em softwares e na Internet. Desenvolvido pelo Unicode Consortium, esse padrão desempenha um papel crucial ao possibilitar a troca e o processamento de textos em qualquer idioma do mundo, assegurando assim a interoperabilidade de dados entre sistemas e plataformas distintas.

Esse padrão possibilita a representação precisa de todos os caracteres, símbolos e ideogramas utilizados nos mais variados idiomas e sistemas de escrita do mundo. São mais de um milhão de elementos, e está em constante atualização. Com a introdução do Unicode, a indústria superou um cenário de fragmentação, com várias representações de texto limitadas e incompatíveis entre si, adotando um paradigma de estabilidade de dados e redução de custos de desenvolvimento (The Unicode Consortium, 2023c).

Notavelmente, o Unicode foi adotado pelo World Wide Web Consortium (W3C) (2005) como o padrão para documentos HTML, viabilizando a criação de websites multilíngues. Atualmente, estima-se que 98,1% das páginas da Web utilizam o UTF-8 (W3Techs, 2024), uma das *character encodings* definida pelo Unicode, evidenciando seu papel fundamental na Internet.

O Abstract Character Repertoire é o conjunto de todos os caracteres abstratos que o Unicode se propõe a representar digitalmente. Estes caracteres são definidos pela sua identidade – vinculada ao sistema de escrita a que pertencem e suas propriedades – e não pela aparência visual. É importante diferenciar os caracteres abstratos e os glifos: os glifos são as representações gráficas concretas de uma sequência de caracteres e não fazem parte do escopo do Unicode. Um mesmo caractere pode assumir formas diferentes conforme o contexto, já que a escolha de glifos depende da fonte e do suporte tipográfico do sistema operacional utilizado (The Unicode Consortium, 2022).

O Unicode estabelece um mapeamento direto entre esse repertório de caracteres e um conjunto de inteiros não negativos, chamados *code points*,

normalmente representados em hexadecimal. Cada *code point* identifica de forma inequívoca um caractere abstrato e recebe também um nome único (The Unicode Consortium, 2023c). Por exemplo, o caractere “a”, associado ao número 97, é designado como `U+0061 LATIN SMALL LETTER A`.

Para que sequências de *code points* se tornem utilizáveis em softwares na prática, eles precisam ser codificados em sequências de *bytes*. Essa é a essência de uma *character encoding*, um padrão que determina como essa transformação é feita (The Unicode Consortium, 2022).

Dentre as várias *encodings* definidas pelo Unicode, o UTF-8 destaca-se de maneira singular. Criada por Ken Thompson para o sistema operacional *Plan 9*, o UTF-8 possui três importantes características. A primeira é sua compatibilidade retroativa com o ASCII (a principal *character encoding* para textos em inglês antes do Unicode se estabelecer), garantindo que textos em ASCII sejam automaticamente textos válidos em UTF-8 sem necessidade de alteração (Yergeau, 2003). Isso assegura uma transição suave de sistemas legados, mantendo a integridade dos dados. A segunda característica é ser uma codificação de comprimento variável, permitindo que diferentes *code* points sejam representados por diferentes quantidades de *bytes*. Assim, o UTF-8 é adaptável à expansão contínua do repertório de caracteres do Unicode. Por último, ele é auto-sincronizável (do inglês *self-synchronizing*): tanto os *bytes* iniciais e os *bytes* subsequentes de um *code point* possuem padrões de bits distintos, o que permite que decodificadores identifiquem os limites dos *code points* mesmo se começarem a leitura no meio de uma sequência (Yergeau, 2003). Essas virtudes justificam a dominância do UTF-8 na era da Internet.

Por fim, o Unicode define um conjunto de propriedades para cada *code point*, padronizando uma variedade de algoritmos que empregam essas propriedades para fins como transformação de maiúsculas e minúsculas, determinação da direção do texto, ordenação de caracteres e identificação dos limites de segmentos textuais, como palavras e frases (The Unicode Consortium, 2023c, 2023b). Os caracteres, suas propriedades e dados relacionados compõem o Unicode Character Database (UCD).

Dentro do universo do Unicode, uma complexidade notável reside na diferença entre o que o usuário percebe como um “caractere” — a unidade básica de um sistema de escrita — e um caractere abstrato do repertório do Unicode (The Unicode Consortium, 2023b). Tomemos, por exemplo, a sequência “👩‍⚕️”: apesar de ser um único caractere do ponto de vista do usuário, é codificada como uma sequência de três *code points* distintos, `U+1F469 (WOMAN)`, `U+200D (ZERO-WIDTH`

JOINER) e U+1F680 (ROCKET). O conceito que melhor corresponde à percepção do usuário é chamado de *grapheme cluster*, e seus limites dentro de uma sequência de *code points* podem ser determinados a partir das tabelas de propriedades do UCD.

Para os usuários, a maneira como o texto é representado internamente pelo software é geralmente irrelevante. Assim, as aplicações interativas que visam proporcionar uma experiência de usuário excepcional devem atender às suas expectativas sobre o que constitui um caractere. Isso implica na habilidade de identificar e manipular *grapheme clusters*, tratando-os como a menor unidade de texto em processos como a seleção, formatação e navegação textual — por exemplo, ao usar as setas do teclado — assim como na definição de limites em campos de formulários.

2 Situação atual e trabalhos relacionados

A partir dos conceitos do Unicode apresentados na seção anterior, esta seção discute como diferentes linguagens, bibliotecas e ferramentas lidam com texto Unicode na prática, com foco no ecossistema Lua e em soluções que oferecem segmentação de texto e *case folding*. São descritos trabalhos relacionados que já endereçam, de forma parcial ou completa, os problemas abordados neste projeto, destacando seus pontos fortes e limitações, situando o *luagrapheme* nesse panorama como mais uma alternativa dentro desse conjunto de soluções.

2.1 Suporte ao Unicode em linguagens de programação

Enquanto as linguagens de programação oferecem diversos mecanismos para manipulação de texto, poucas contemplam *grapheme clusters* em suas abstrações fundamentais e bibliotecas padrão. Python, por exemplo, define *strings* como “sequências imutáveis de Unicode *code points*”, caracterizando a iteração e o cálculo de tamanho em termos desses *code points* (Python Software Foundation, 2024). Similarmente, Lua “assume *character encodings* de um *byte*” (Ierusalimschy; Figueiredo; Celes, 2023), com operações de *string* baseadas no número de *bytes*. Essas abordagens, focadas na simplicidade e na compatibilidade com sistemas mais antigos, podem não preparar desenvolvedores para as sutilezas da segmentação de texto em idiomas variados. A linguagem Swift destaca-se ao incorporar *grapheme clusters* como unidades fundamentais para o manuseio de strings (Apple Inc., 2024), oferecendo uma abordagem mais alinhada às necessidades globais do processamento de texto contemporâneo para interfaces de usuário.

Isso sugere que muitos desenvolvedores, ao se basearem na documentação e exemplos predominantes, podem não se deparar com o conceito de *grapheme clusters* durante seu aprendizado. Essa situação pode implicar que o suporte à diversidade linguística do texto contemporâneo seja postergado ou ignorado nos projetos de software, especialmente porque o alfabeto latino, com sua correspondência direta entre *grapheme clusters*, *code points* e *bytes* UTF-8, tende a mascarar essas complexidades.

2.2 Suporte ao Unicode no ecossistema Lua

No ecossistema Lua, essas questões aparecem de forma especialmente evidente. Por um lado, o interpretador trata strings como sequências genéricas de *bytes* (Ierusalimschy; Figueiredo; Celes, 2023), o que facilita a integração com sistemas legados e com protocolos que usam UTF-8. Por outro, essa escolha transfere para bibliotecas externas a responsabilidade de implementar algoritmos baseados nas propriedades do Unicode — como segmentação de texto, normalização ou *case folding*.

Até a versão 5.2, Lua não oferecia suporte nativo a nenhum aspecto específico do Unicode além da capacidade de armazenar *bytes* arbitrários em *strings*. Desde então, surgiram diferentes trabalhos relacionados que procuram preencher essa lacuna em direções complementares: a biblioteca padrão `utf8` incluída no núcleo da linguagem, *bindings* para a ICU e reimplementações dos algoritmos do Unicode. As subseções a seguir descrevem essas soluções e situam o *luagrapheme* em relação a elas.

2.2.1 A biblioteca padrão *utf8*

A partir da versão 5.3, Lua passou a incluir com o interpretador a biblioteca *utf8*, que converte sequências de *bytes* em UTF-8 em sequências de *code points*, e vice-versa. Além disso, o interpretador passou também a reconhecer sequências de escape UTF-8 em *strings*: por meio da sintaxe `\u{XXX}`, em que `XXX` é uma sequência de dígitos hexadecimais que representam um *code point*, o interpretador insere a representação em UTF-8 correspondente.

Apesar disso, a biblioteca não possui implementações de algoritmos que dependem das propriedades de cada *code point* no Unicode Character Database (UCD). Roberto Ierusalimschy afirma, no livro “Programming in Lua”, que espera que uma biblioteca externa atenda essa necessidade:

> “Infelizmente, não há muito mais que o Lua possa oferecer. O Unicode tem peculiaridades demais. É praticamente impossível abstrair quase qualquer conceito das línguas específicas. (…) Por causa dessa complexidade, o suporte completo ao Unicode exige tabelas enormes, que são incompatíveis com o tamanho reduzido do Lua. Portanto, para qualquer coisa mais sofisticada, a melhor abordagem é usar uma biblioteca externa.” (Ierusalimschy, 2016, p. 31, tradução própria)

Implementar algoritmos capazes de manejar essas tarefas com precisão demanda um entendimento profundo do padrão Unicode, a incorporação das

propriedades relevantes do Unicode Character Database (UCD) na aplicação, a manutenção de uma suíte de testes abrangente e a atualização contínua da implementação para acompanhar as evoluções do padrão e do UCD. Diante dessa complexidade, a abstração dessas funcionalidades em uma biblioteca de apoio se torna não apenas prática, mas essencial.

2.2.2 International Components for Unicode (ICU)

Uma das possibilidades para oferecer funcionalidades do Unicode em Lua é o desenvolvimento de *bindings* para a International Components for Unicode (ICU). Essa biblioteca, desenvolvida pelo próprio *Unicode Consortium*, implementa de forma abrangente tudo que o padrão descreve. Porém, essa abrangência vem acompanhada de uma curva de aprendizado elevada e complexidade substancial de implementação. Ocupando cerca de 38 MB, essa alternativa pode ser inviável em ambientes computacionais com limitações de memória e armazenamento, como sistemas embarcados. Além disso, ela requer um compilador C++ e uma versão do GNU *make* (The Unicode Consortium, 1999), que podem não estar disponíveis nestes ambientes.

No ecossistema Lua, o projeto *ICU4Lua* (Duncanc, 2009) se propõe a oferecer essas *bindings*. Entretanto, ele não cobre todo o escopo da ICU, limitando-se a conversões entre *character encodings*, normalização e ordenação de *strings*. Em particular, não oferece os algoritmos de segmentação de texto em *grapheme clusters*, palavras, frases e oportunidades de quebras de linha.

2.2.3 *lua-uni-algos*

Uma segunda abordagem consiste na implementação dos algoritmos e tabelas de propriedades do Unicode diretamente em Lua, como é o caso da *lua-uni-algos* (Krüger, 2020), mantida pelo projeto *LaTeX* e voltada para o uso no *LuaTeX*. Esta biblioteca implementa segmentação de texto em *grapheme clusters* e palavras, além de *case folding* e normalização.

Esta estratégia, apesar de oferecer a vantagem de evitar dependências externas, compromete o desempenho e a eficiência em uso de memória devido ao *overhead* intrínseco à máquina virtual Lua e seus mecanismos de gerenciamento de memória. Além disso, por ser uma nova implementação dos algoritmos do Unicode, deve vir acompanhada de uma extensa suíte de testes que valide seu comportamento, o que não ocorre na *lua-uni-algos*.

2.2.4 *lua-utf8*

Outra abordagem possível consiste em desenvolver uma biblioteca em C voltada especificamente ao ecossistema Lua que implemente diretamente os algoritmos do Unicode, como ocorre na *lua-utf8*.

Inicialmente, essa biblioteca se limitava a implementar operações de conversão entre UTF-8 e *code points* para versões anteriores ao Lua 5.3 (Wang, 2019). Durante o desenvolvimento deste trabalho, a *lua-utf8* passou a oferecer também a funcionalidade de segmentar *strings* em *grapheme clusters*, fornecendo uma solução parcial para o problema que o *luagrapheme* se propõe a resolver. Porém, a biblioteca não cobre outras unidades de segmentação — palavras, frases e oportunidades de quebras de linha —, não oferece integração com *LPeg* e não se apoia em implementações estabelecidas e testadas dos algoritmos.

2.3 *libgrapheme*

O *luagrapheme* adota uma via distinta das abordagens mencionadas, oferecendo suporte aos algoritmos do Unicode através de *bindings* para a *libgrapheme.* Desenvolvida por Laslo Hunhold (2021), *libgrapheme* é uma biblioteca em C para segmentação de texto Unicode que se destaca por seu compromisso com simplicidade, eficiência e minimalismo, atuando em conformidade com o padrão Unicode.

Para a compilação, a *libgrapheme* requer apenas um compilador ISO C99 e o POSIX *make*, amplamente disponíveis em ambientes computacionais *Unix*. O binário resultante da compilação possui cerca de 300 KB. Este tamanho reduzido é fundamental para permitir sua “linkagem” estática em sistemas onde o espaço é uma preocupação, diferenciando-se significativamente da ICU, cujo tamanho limita sua aplicabilidade.

Além disso, uma característica notável da *libgrapheme* é não possuir nenhuma dependência em tempo de execução, nem mesmo a biblioteca padrão do C, facilitando sua distribuição e o uso em ambientes computacionais restritos. A *libgrapheme* é distribuída sob a licença ISC, permitindo uso, modificação e compartilhamento livre, e é acompanhada por uma suíte extensiva de testes que verificam sua conformidade com o padrão Unicode (Hunhold, 2021).

Ao escolher a *libgrapheme*, o *luagrapheme* busca fornecer uma base sólida e testada para a manipulação de texto conforme as especificações do Unicode, ao mesmo tempo que oferece uma interface amigável para programadores Lua, facilitando a integração com outros componentes do ecossistema, como o *LPeg*.

2.4 *LPeg*

O *LPeg* é uma biblioteca de *pattern matching* para Lua construída sobre Parsing Expression Grammars (PEGs), um formalismo que oferece uma base rigorosa para o reconhecimento de linguagens. A proposta surge da constatação de que as expressões regulares tradicionais são limitadas: muitas estruturas de interesse não podem ser descritas por elas, e a solução encontrada em diversas linguagens foi acumular extensões *ad hoc*, como *greedy* e *non-greedy repetition*, *lookahead* ou *longest match*. Essas extensões, porém, não têm fundamentação formal clara e levam a implementações de difícil previsibilidade. O *LPeg* evita essa fragmentação ao adotar diretamente o formalismo das PEGs, que já contempla esses mecanismos (Ierusalimschy, 2009).

Dessa forma, o *LPeg* combina a praticidade de uma biblioteca de padrões com a expressividade das PEGs. Ele é capaz de lidar com estruturas que vão além do escopo regular, como parênteses balanceados ou comentários aninhados, ao mesmo tempo em que oferece um modelo de desempenho transparente, permitindo ao desenvolvedor entender e prever o custo dos padrões que escreve. Ao invés de múltiplos “remendos” sobre *regexes*, o *LPeg* entrega uma ferramenta única e coesa, bem integrada ao ecossistema Lua, que se tornou uma escolha de referência para processamento de texto, alcançando mais de cinco milhões de *downloads* (LuaRocks, 2014).

Dada a popularidade do *LPeg* no ecossistema Lua, é desejável que uma biblioteca de processamento de texto Unicode como o *luagrapheme* integre-se de maneira harmoniosa com *LPeg*. Tal integração amplia as capacidades de *LPeg* para o domínio da manipulação de texto Unicode, lidando com padrões que envolvem segmentação de *grapheme clusters*, palavras, frases e linhas. Isso permite que desenvolvedores aproveitem o eficiente motor de correspondência de padrões de *LPeg* para um leque mais extenso de usos, como demonstrado em 3.3.4.

3 Projeto e implementação

O *luagrapheme* assume como contrato trabalhar com *strings* em UTF-8, a codificação predominante na Web e a mais comum no ecossistema Lua. Esse recorte simplifica o escopo da biblioteca: cabe à aplicação converter textos provenientes de outras codificações antes de passá-los à API.

O design da API buscou acompanhar o estilo de programação da linguagem. Sempre que apropriado, as operações de segmentação são expostas como iteradores compatíveis com *generic for loops*, permitindo consumir os resultados sob demanda, evitando alocações desnecessárias. Além disso, as funções retornam índices em *bytes*, prontos para uso com `string.sub` e funções similares. Com isso, a biblioteca se complementa à biblioteca padrão de Lua em vez de tentar substitui-la.

O escopo funcional foi mantido deliberadamente enxuto: segmentação de *grapheme clusters*, palavras, frases e oportunidades de quebras de linha, além de *case folding* e integração com *LPeg*. Funcionalidades como normalização e ordenação estão fora da proposta.

Desde o início, o projeto teve como objetivo a ampla compatibilidade: o *luagrapheme* funciona em Lua 5.1, 5.2, 5.3, 5.4 e *LuaJIT*. Para isso, o código em C utiliza a API do Lua 5.3 e conta com o módulo *compat53* (Janda et al., 2015), incluído no próprio repositório, que atua como um adaptador para as versões anteriores.

Quanto a plataformas, a biblioteca deve compilar em qualquer sistema que ofereça um compilador ISO C99, o POSIX *make* e um interpretador Lua funcional. Na prática, foi testada no macOS (arm64) e no Linux (x86_64).

Em tempo de execução, a única dependência é o próprio Lua, já que a *libgrapheme* é “linkada” estaticamente durante a compilação. Para instalação, a biblioteca pode ser compilada via *make* ou por meio do LuaRocks, sistema de distribuição amplamente utilizado no ecossistema Lua, dispensando configuração de *paths* e *flags* de compilação.

3.1 Requisitos funcionais

As funcionalidades da biblioteca foram formalizadas por meio de requisitos funcionais, identificados pelos códigos RF-01 a RF-08. Esses requisitos sintetizam o comportamento da biblioteca do ponto de vista do usuário e serão referenciados nas seções seguintes.

RF-01. Segmentar *strings* UTF-8 em *grapheme clusters* de acordo com Unicode Standard Annex #29.

RF-02. Segmentar *strings* UTF-8 em palavras de acordo com Unicode Standard Annex #29.

RF-03. Segmentar *strings UTF-8* em frases de acordo com Unicode Standard Annex #29.

RF-04. Segmentar *strings* UTF-8 em oportunidades de quebras de linha de acordo com o algoritmo do Unicode Standard Annex #14.

RF-05. Para cada tipo de unidade textual, disponibilizar uma função que crie um iterador compatível com `generic for loops`, retornando pares de índices `i, j` em *bytes* correspondentes aos limites da unidade textual.

RF-06. Para cada tipo de unidade textual, disponibilizar uma função que, dada uma *string* e um índice em *bytes*, retorne o índice em *bytes* da próxima unidade textual.

RF-07. Em um submódulo opcional, para cada tipo de unidade textual, disponibilizar um padrão *LPeg* que corresponda a exatamente uma ocorrência da unidade textual.

RF-08. Oferecer operações de *case folding* (*upper*, *lower* e *title*) a partir de propriedades do UCD, independente de *locale*.

3.2 Requisitos não funcionais

De forma complementar, foram formalizados os requisitos não funcionais, identificados pelos códigos RNF-01 a RNF-11. Esses requisitos orientam as decisões de implementação e metodologia e também serão referenciados nas seções seguintes.

RNF-01. Compatibilidade Lua: A biblioteca deve ser compatível com Lua 5.1, 5.2, 5.3, 5.4 e LuaJIT, preservando a mesma API nessas versões.

RNF-02. Ambiente de compilação: A biblioteca deve ser compilável com um compilador ISO C99 e POSIX make.

RNF-03. Ambiente de execução: Em tempo de execução, a biblioteca não deve depender de componentes externos além do interpretador Lua, incorporando a *libgrapheme* por linkagem estática.

RNF-04. *LPeg* opcional: A biblioteca deve poder ser usada sem *LPeg*, se *LPeg* não estiver disponível em tempo de execução.

RNF-05. Tamanho reduzido: O artefato resultante da compilação deve manter tamanho binário reduzido, adequado ao uso em ambientes embarcados.
RNF-06. Determinismo: As funcionalidades são determinísticas, retornando resultados idênticos para parâmetros idênticos, de forma consistente entre plataformas e independente de estados globais.
RNF-07. Eficiência: As funcionalidades expostas em Lua devem ter a mesma complexidade assintótica de tempo da funcionalidade equivalente na *libgrapheme* em C, com *overhead* constante.
RNF-08. Segurança de memória: O código em C não deve conter comportamentos indefinidos e erros de memória.
RNF-09. Análise estática e formatação: Devem ser utilizadas ferramentas de análise estática para verificar erros comuns de programação em C e Lua, e ferramentas de formatação para garantir consistência em indentação e nomenclatura.
RNF-10. Testes automatizados: Todas as funcionalidades devem ser testadas através de testes automatizados, validando casos de borda. Os testes devem testar a camada de *bindings* e não os algoritmos do Unicode em si, já testados pela *libgrapheme.*
RNF-11. Integração contínua: Alterações no código-fonte da biblioteca só devem ser integradas após a execução bem-sucedida dos fluxos de análise estática e dos testes automatizados.

3.3 Exemplos de uso

Esta seção apresenta as funcionalidades da biblioteca por meio de exemplos de uso — o caminho mais natural para compreender como a biblioteca se encaixa no ecossistema Lua.

3.3.1 Segmentação por grapheme clusters (RF-01 e RF-05)

Suponha que uma aplicação precise dividir em “caracteres” a *string* "Hello, 세상" (“mundo”, em coreano, no alfabeto Hangul). Uma primeira tentativa pode ser usar a biblioteca `utf8`, inclusa com o interpretador Lua, iterando por ocorrências de `utf8.charpattern`, um padrão de bytes que corresponde a um *code point*:

Figura 1 – Divisão de uma *string* em Hangul em *code points*

```
utf8 = require("utf8")

function split_codepoints(text)
  local t = {}
  for c in text:gmatch(utf8.charpattern) do
    t[#t+1] = c
  end
  return table.concat(t,"|")
end

print(split_codepoints("Hello, 세상"))

---> H|e|l|l|o|,| |ᄉ|ᅦ|ᄉ|ᅡ|ᆼ
```

No Hangul, o que aparece como um caractere é, na verdade, uma sílaba (por exemplo, “세” ou “상”). Cada sílaba é composta por *jamos* (consoantes e vogais como “ᄉ”, “ ᅦ”, e “ᆼ”), e cada *jamo* pode ter seu próprio *code point* no Unicode. Assim, iterar por *code points* não resulta nas sílabas, mas sim nos *jamos* que as compõem. A solução é segmentar o texto em *grapheme clusters*.

Para isso, a biblioteca oferece a função `graphemes`, que aceita uma `string` e retorna uma função iteradora que, a cada invocação, retorna pares de inteiros `i` e `j` que representam o índice inicial e o índice final, em *bytes*, que delimitam um *grapheme cluster*. Assim, cada passo da iteração corresponde a um caractere no sentido percebido pelo usuário — neste exemplo, as duas sílabas “세” e “상” ao invés dos cinco *jamos* que as compõem:

Figura 2 – Segmentação de uma *string* em Hangul em *grapheme clusters*

```
graphemes = require("luagrapheme").graphemes

function split_graphemes(text)
  local t = {}
  for i, j in graphemes(text) do
    t[#t+1] = text:sub(i, j)
  end
  return table.concat(t, "|")
end

print(split_graphemes("Hello, 세상"))

--> H|e|l|l|o|,| |세|상
```

De forma similar, esse problema se manifesta ao segmentar *strings* com emojis.

Figura 3 – Divisão de *emojis* em *code points*

```
print(split_codepoints("👩‍🚀🇧🇷"))
--> 👩|‍|🚀|🇧|🇷
```

Os emojis podem ser compostos por múltiplos *code points* que, juntos, fazem parte de um único *grapheme cluster*. Como o exemplo sugere, o emoji "👩‍🚀" (*woman astronaut*) é composto por 3 *code points*: `U+1F469 WOMAN` ("👩"), `U+200D ZERO WIDTH JOINER` e `U+1F680 ROCKET` ("🚀"). Já a bandeira do Brasil "🇧🇷" resulta da combinação dos *code points* `U+1F1E7 REGIONAL INDICATOR SYMBOL LETTER B` e `U+1F1F7 REGIONAL INDICATOR SYMBOL LETTER R`.

A biblioteca trata cada um dos emojis como *grapheme clusters* únicos, refletindo o que o usuário percebe visualmente como um caractere:

Figura 4 – Divisão de *emojis* em *grapheme clusters*

```
print(split_graphemes("👩‍🚀🇧🇷"))
--> 👩‍🚀|🇧🇷
```

3.3.2 *Case folding* (RF-08)

Case folding é o processo de transformar letras maiúsculas em minúsculas (ou vice-versa). Em Lua, essa operação é oferecida pelas funções `string.lower`

e `string.upper`, cuja lógica depende da *locale* atual — configurada com `os.setlocale`. A *locale* é um estado global do programa (Ierusalimschy; Figueiredo; Celes, 2023), e portanto, afeta as funções de processamento de texto da biblioteca padrão C da aplicação *host* e vale para todas as *threads* Lua. Além disso, a lista de *locales* disponíveis varia de acordo com o sistema operacional, normalmente representando um idioma específico associado a um *character encoding* (por exemplo, `"pt_BR.UTF-8"`) (IEEE; The Open Group, 2024).

Esse mecanismo é útil quando o texto está em um único idioma conhecido. Por exemplo, para formatar números ou datas segundo as convenções locais do usuário. Porém, para fazer *case folding* em textos multilíngues ou em *strings* cujo idioma não é conhecido, a abordagem por *locale* não é confiável.

O *luagrapheme* resolve esse problema oferecendo *case folding* de acordo com as propriedades do Unicode Character Database (UCD), que define um mapeamento universal e consistente entre maiúsculas e minúsculas (The Unicode Consortium, 2023c). Assim, é possível aplicar `lower`, `upper` e até `title` a qualquer *string* UTF-8, independentemente do idioma, do sistema operacional ou de um estado global da aplicação *host*:

Figura 5 – Comparação de funções de *case folding*

```
luagrapheme = require("luagrapheme")

input = "HeLLo, мИр!"

print(string.lower(input))
--> hello, □□□!
print(string.upper(input))
--> HELLO, мИр!

print(luagrapheme.lower(input))
--> hello, мир!
print(luagrapheme.upper(input))
--> HELLO, МИР!
print(luagrapheme.title(input))
--> Hello, Мир!
```

Na Figura 5, usando a *locale* `"en_US.UTF-8"` no macOS 26, as funções padrão de Lua produzem resultados incorretos: `string.lower` retorna uma sequência ilegível, enquanto `string.upper` simplesmente não converte os caracteres cirílicos. Na *locale* `"C"`, a única cuja disponibilidade e comportamento são garantidos pela linguagem C, por exemplo, não retorna caracteres ilegíveis,

mas também não ocorre a conversão dos caracteres cirílicos. Por outro lado, o *luagrapheme* garante resultados previsíveis e corretos em qualquer ambiente computacional.

3.3.3 Segmentação por outras unidades textuais (RF-02, RF-03 e RF-04)

Além dos *grapheme clusters*, o *luagrapheme* permite a segmentação por outros algoritmos do padrão Unicode: palavras, frases e oportunidades de quebras de linha. Esses algoritmos estão definidos em UAX #29 (2023b) e UAX #14 (2023a).

Segmentar em palavras e frases é útil em editores de texto, que oferecem comandos para mover o cursor, selecionar ou apagar blocos inteiros de palavras ou frases de uma só vez. Também é indispensável em sistemas de busca, contagem de palavras e preparação de texto para processamento de linguagem natural.

Já a segmentação por oportunidades de quebras de linha é indispensável em *layout*: quando uma aplicação precisa ajustar o texto ao espaço disponível, deve saber exatamente onde pode inserir quebras de linha sem comprometer a legibilidade. O algoritmo do Unicode fornece esses pontos de forma padronizada e consistente entre idiomas.

Assim como a função `graphemes`, as funções `words`, `sentences` e `lines` do *luagrapheme* retornam iteradores que percorrem a *string* e retornam pares de índices `i` e `j`, em *bytes*, que delimitam cada segmento. Para ilustrar, podemos definir uma função utilitária que recebe uma dessas funções de segmentação e uma *string* de entrada, imprimindo cada segmento com seus limites em *bytes* e seu conteúdo:

Figura 6 – Impressão de limites de segmentos em *bytes*

```
luagrapheme = require("luagrapheme")

words = luagrapheme.words
sentences = luagrapheme.sentences
lines = luagrapheme.lines

function print_segments(fn, s)
  for i, j in fn(s) do
    print(i, j, '"' .. s:sub(i, j) .. '"')
  end
end
```

Podemos agora aplicar os diferentes algoritmos de segmentação sobre a mesma *string* e comparar a saída:

Figura 7 – Exemplo de segmentação em frases

```
input = "Lua 5.4 foi lançada em 19/09/2025. É uma " ..
        "linguagem simples e poderosa"

print_segments(sentences, input)
--> 1   36  "Lua 5.4 foi lançada em 19/09/2025. "
--> 37  71  "É uma linguagem simples e poderosa"
```

Note que o ponto entre “5” e “4” não foi considerado o término de uma frase, mas o ponto final após a data foi.

Figura 8 – Exemplo de segmentação em palavras

```
print_segments(words, input)
--> 1   3   "Lua"
--> 4   4   " "
--> 5   7   "5.4"
--> 8   8   " "
--> 9   11  "foi"
--> 12  12  " "
--> 13  20  "lançada"
--> 21  21  " "
--> 22  23  "em"
--> 24  24  " "
--> 25  26  "19"
--> 27  27  "/"
--> 28  29  "09"
--> 30  30  "/"
--> 31  34  "2025"
--> 35  35  "."
--> 36  36  " "
--> 37  38  "É"
--> 39  39  " "
--> 40  42  "uma"
--> 43  43  " "
--> 44  52  "linguagem"
--> 53  53  " "
--> 54  60  "simples"
--> 61  61  " "
--> 62  62  "e"
--> 63  63  " "
--> 64  71  "poderosa"
```

A segmentação por palavras identifica corretamente cada unidade, tratando os espaços, a pontuação e os separadores de data como segmentos independentes.

Figura 9 – Exemplo de segmentação em oportunidades de quebras de linha

```
print_segments(lines, input)
--> 1   4   "Lua "
--> 5   8   "5.4 "
--> 9   12  "foi "
--> 13  21  "lançada "
--> 22  24  "em "
--> 25  36  "19/09/2025. "
--> 37  39  "É "
--> 40  43  "uma "
--> 44  53  "linguagem "
--> 54  61  "simples "
--> 62  63  "e "
--> 64  71  "poderosa"
```

Diferentemente da segmentação por palavras, a segmentação por oportunidade de quebras de linha não divide os componentes da data (separados por “/”), e não permite quebras em posições que poderiam gerar linhas compostas apenas por espaços.

3.3.4 Segmentação de texto em gramáticas e padrões *LPeg* (RF-07)

Além da iteração direta, a biblioteca disponibiliza os algoritmos de segmentação de texto como padrões *LPeg*, o que permite escrever gramáticas que entendem as fronteiras de *grapheme clusters*, palavras, frases e oportunidades de quebras de linha. Os padrões `G()`, `W()`, `L()` e `S()` correspondem a exatamente um *grapheme cluster*, palavra, oportunidade de quebras de linha e frase, respectivamente.

Para ilustrar, vamos criar uma linguagem markup minimalista em *plain text* que permite a escrita de listas e parágrafos, onde cada item de lista pode ter um *bullet* arbitrário. A figura a seguir mostra um exemplo de entrada nesta linguagem:

Figura 10 – Exemplo de entrada para a linguagem de *markup*

```
Olá!

🤦 Este é um item de lista;
✅ e este é outro item de lista.

✅ Este faz parte de uma lista diferente.

Isso é um parágrafo.
Um parágrafo pode ser composto por múltiplas linhas.

Uma quebra dupla de linha inicia um novo parágrafo.
```

Com *LPeg*, podemos escrever uma gramática que reconhece listas e parágrafos e constrói a seguinte árvore de sintaxe abstrata (AST, do inglês *Abstract Syntax Tree*):

Figura 11 – Árvore de sintaxe abstrata (AST) para a entrada de exemplo

```
{ { type = "paragraph", "Olá!" },
  { type = "list",
    { bullet = "🤦",
      text = "Este é um item de lista;" },
    { bullet = "✅",
      text = "e este é outro item de lista." } },
  { type = "list",
    { bullet = "✅",
      text = "Este faz parte de uma lista diferente." } },
  { type = "paragraph",
    "Isso é um parágrafo.",
    "Um parágrafo pode ser composto por múltiplas linhas." },
  { type = "paragraph",
    "Uma quebra dupla de linha inicia um novo parágrafo." } }
```

Figura 12 – Gramática para reconhecimento de listas e parágrafos

```
local lglpeg = require("luagrapheme.lpeg")
local lpeg = require("lpeg")
local P, V, C, Ct = lpeg.P, lpeg.V, lpeg.C, lpeg.Ct
local G = lglpeg.G

local EOL = P("\n") + P("\r\n")
local Space = P(" ")

local Document = V("Document")
local Block = V("Block")
local ListBlock = V("ListBlock")
local ListItem = V("ListItem")
local ParagraphBlock = V("ParagraphBlock")
local ParagraphLine = V("ParagraphLine")

local grammar = P({
   Document,
   Document = Ct(Block ^ 0) * -1,
   Block = (ListBlock + ParagraphBlock) * (EOL ^ 0),
   ListBlock = (ListItem ^ 1) / function(...)
      return { type = "list", ... }
   end,
   ListItem = (C(G()) * Space * C((1 - EOL) ^ 1)) /
function(bullet, text)
      return { bullet = bullet, text = text }
   end * EOL,
   ParagraphBlock = ((ParagraphLine ^ 1) / function(...)
      return { type = "paragraph", ... }
   end),
   ParagraphLine = C((1 - EOL) ^ 1) * EOL,
})

local ast = grammar:match(io.stdin:read("*a"))
```

Note que a regra `ListItem` usa o padrão `G()` oferecido pela *luagrapheme*. Com ele, a gramática consegue reconhecer listas cujos *bullets* sejam qualquer *grapheme cluster* — desde o *bullet* tradicional (•) até emojis compostos por vários *code points*. Sem esse recurso, a linguagem teria três alternativas indesejáveis:

- Corresponder apenas um *code point*, limitando os *bullets* possíveis;
- Aceitar sequências arbitrárias de *code points*, o que incorretamente reconheceria parágrafos como listas (no exemplo anterior, “Isso” e “Um” seriam reconhecidos como *bullets*);

- Introduzir marcadores ou separadores que definam o início da lista, como ocorre com HTML.

Ao incorporar o algoritmo de segmentação de *grapheme clusters*, esta gramática consegue descrever mais elementos sintáticos sem a necessidade de separadores artificiais no texto.

O próximo passo é usar a AST resultante para gerar uma saída útil — por exemplo, renderizar HTML e CSS para publicação na Web.

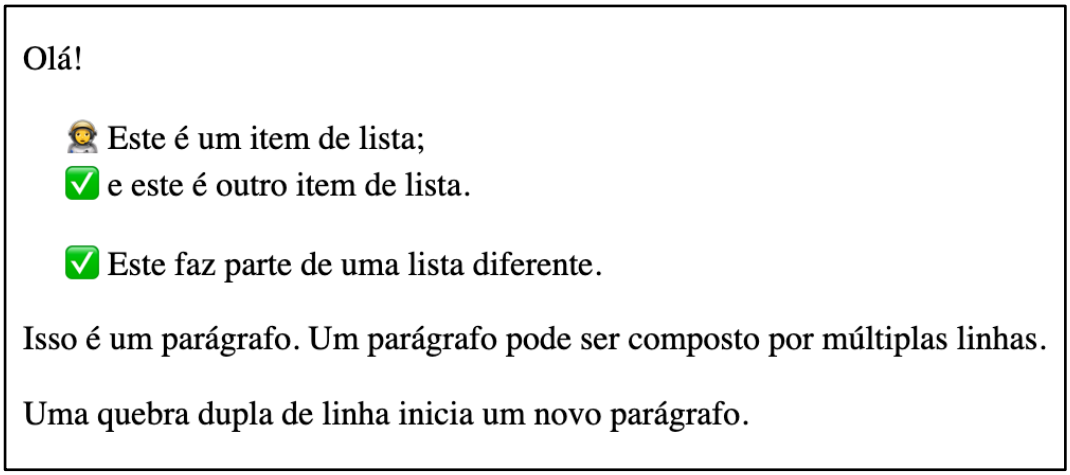
Figura 13 – Geração de HTML a partir da AST

```
local function render_html(ast, file)
   file:write("<!DOCTYPE html>\n")
   file:write("<html>\n")
   file:write("<head>\n")
   file:write("  <meta charset='utf-8'>\n")
   file:write("  <style>\n")
   file:write("    li[data-bullet]::marker { content:
attr(data-bullet) '\\a0'; }\n")
   file:write("  </style>\n")
   file:write("</head>\n")
   file:write("<body>\n")

   for _, block in ipairs(ast) do
      if block.type == "list" then
         file:write("<ul>\n")
         for _, item in ipairs(block) do
            file:write('  <li data-bullet="', item.bullet,
'">', item.text, "</li>\n")
         end
         file:write("</ul>\n")
      elseif block.type == "paragraph" then
         file:write("<p>\n")
         for _, line in ipairs(block) do
            file:write("  ", line, "\n")
         end
         file:write("</p>\n")
      else
         error("Unknown block type: " .. tostring(block.type))
      end
   end

   file:write("</body>\n")
   file:write("</html>\n")
end
```

A função `render_html` percorre a AST e escreve o arquivo HTML correspondente. Blocos do tipo `list` são convertidos em listas não ordenadas `<ul>` cujos itens são renderizados como elementos `<li>` com o atributo `data-bullet` preenchido. Blocos do tipo `paragraph` são transformados em elementos `<p>`, preservando o agrupamento de parágrafos do texto original. A folha de estilos substitui o *bullet* de cada elemento `<li>` pelo conteúdo do atributo `data-bullet` seguido de um espaço não separável (`U+00A0`). O cabeçalho inclui a declaração `<meta charset='utf-8'>`, que instrui o navegador a interpretar o documento com esta *character encoding*.

A página resultante é fiel à estrutura da entrada de exemplo: *bullets* personalizados aparecem corretamente e parágrafos são mantidos como blocos de texto separados.

Figura 14 – Captura de tela de um navegador Web exibindo a página gerada



3.4 Diagramas

Para descrever a organização de módulos da API, foi elaborado um diagrama de classes (Figura 15) que apresenta os módulos públicos `luagrapheme` e `luagrapheme.lpeg`. O núcleo da biblioteca expõe quatro segmentadores (`graphemes`, `words`, `sentences` e `lines`) que compartilham a interface `Segmenter`, oferecendo duas operações: a chamada como função (`__call`), que cria um iterador (`SegmentIterator`) e a função index_after, que retorna o índice em *bytes* do próximo segmento após um índice inicial. O diagrama explicita ainda que a integração com *LPeg* (e por consequência a dependência externa no módulo `lpeg`) reside em um submódulo separado e opcional (RNF-04).

Figura 15 – Diagrama de classes

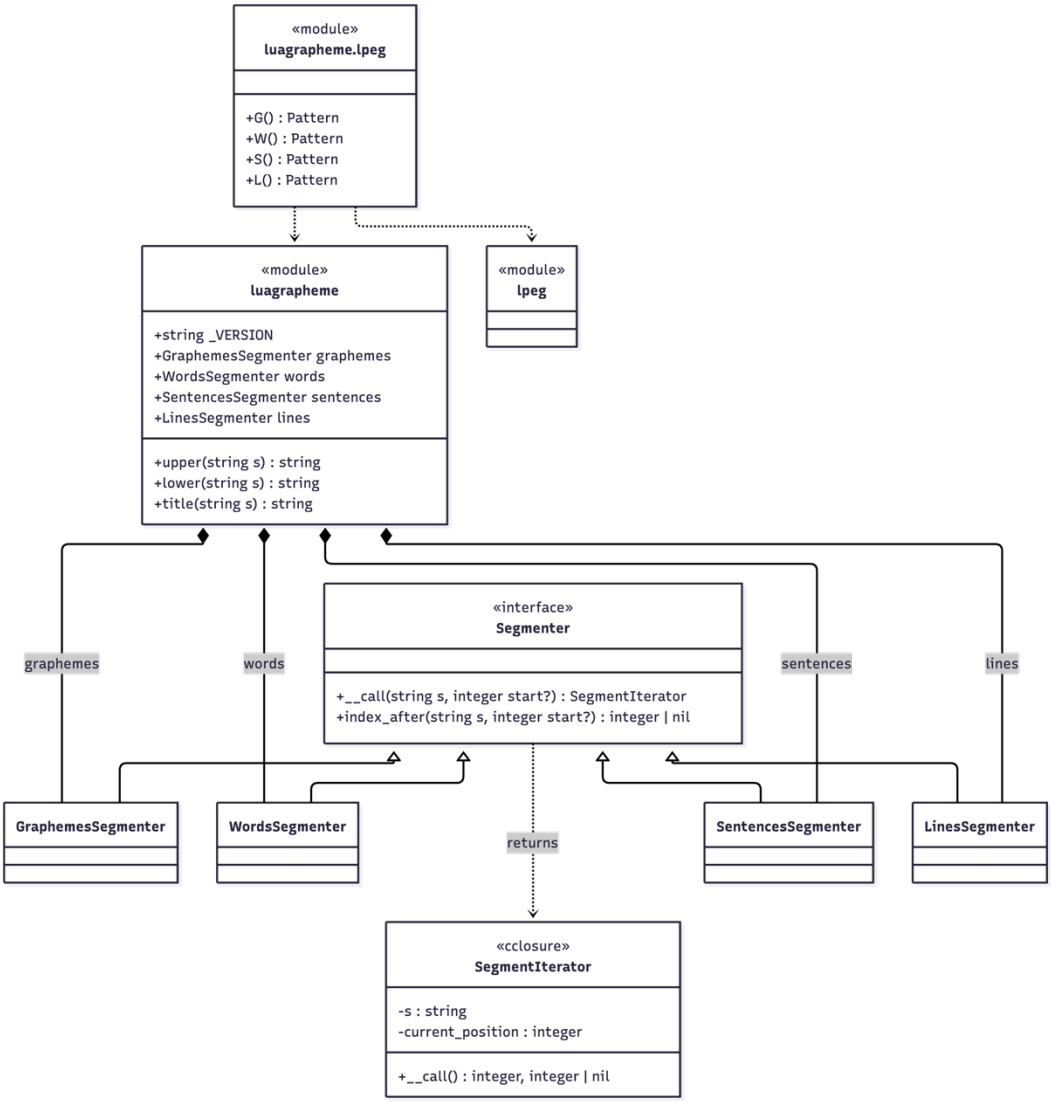
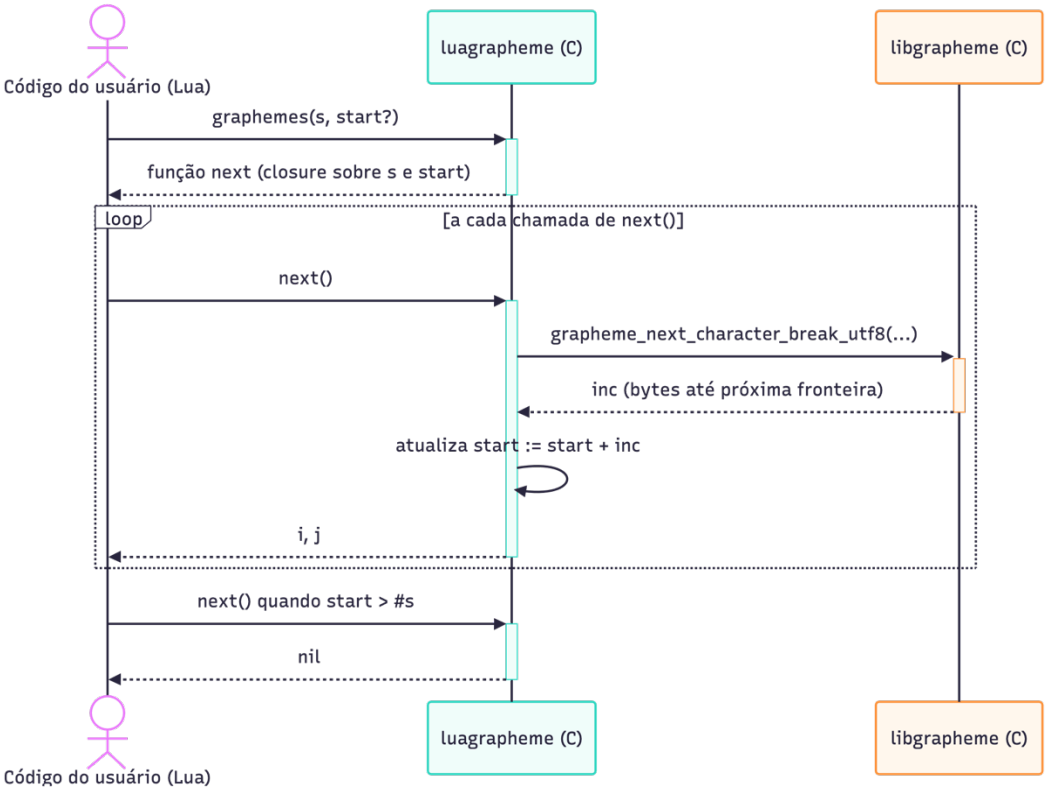
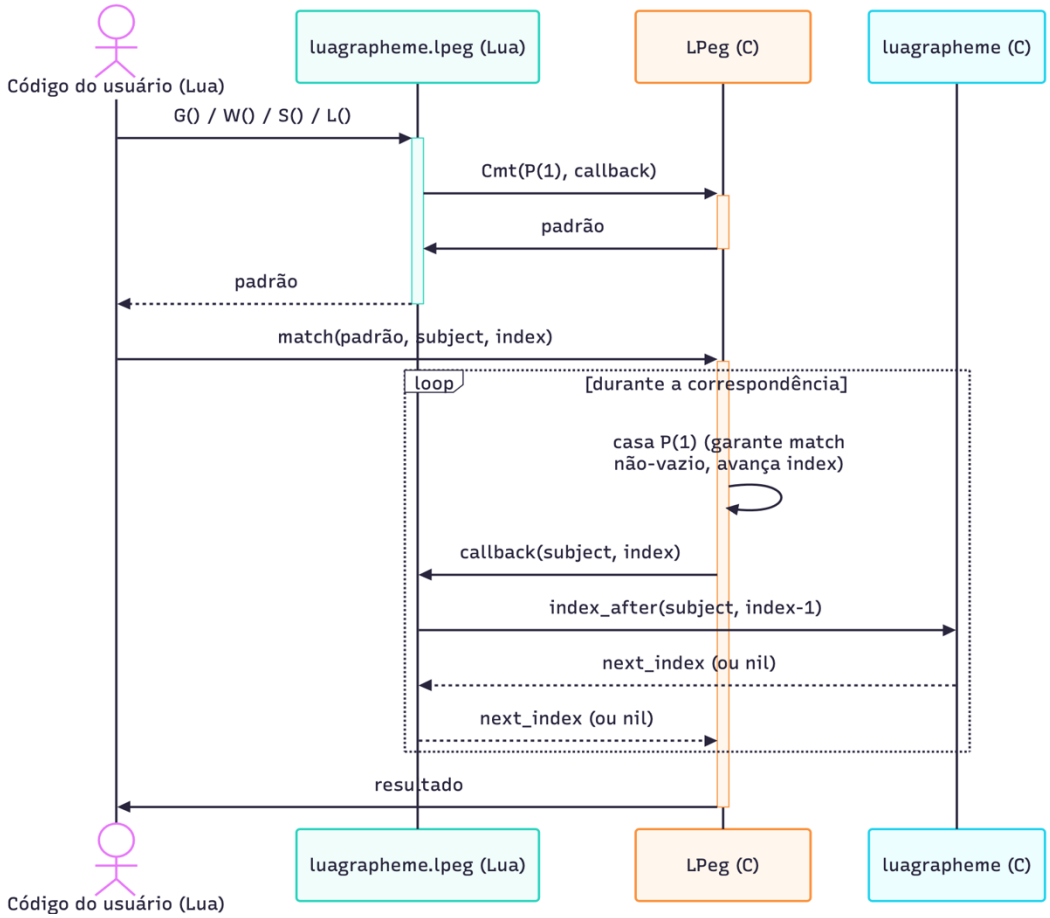


Figura 16 – Diagrama de sequência dos iteradores



A Figura 16 detalha, por meio de um diagrama de sequência, a execução de um iterador retornado pelos segmentadores (RF-05). A chamada inicial a `graphemes(s, start)` (ou aos demais segmentadores) retorna uma função iteradora que mantém internamente uma referência à *string* e a posição atual. A cada chamada subsequente a esta função, a *libgrapheme* é utilizada para determinar o início do próximo segmento, a posição atual é atualizada e são retornados os índices inicial e final do segmento atual, em *bytes*.

Figura 17 – Diagrama de sequência dos padrões *LPeg*



Por fim, a Figura 17 descreve a construção e correspondência dos padrões *LPeg* oferecidos pelo submódulo `luagrapheme.lpeg` (RF-07). As funções `G`, `W`, `S` e `L` constrem um padrão do tipo *match-time capture* através da função `Cmt`, um mecanismo que faz com que *LPeg* chame uma função arbitrária (`callback`) que determina onde ocorre a próxima correspondência (Ierusalimschy, 2025). A composição com o padrão `P(1)` garante para *LPeg* que esses padrões precisam de pelo menos um *byte* para serem correspondidos. Sem essa composição, *LPeg* impede que o padrão seja usado em repetições (por exemplo, `G() ^ N`, que corresponde a pelo menos `N` repetições de `G()`), uma vez que `callback` poderia retornar uma correspondência com a *string* vazia, e repetições não seriam bem definidas.

Durante a correspondência, *LPeg* consome um *byte* e invoca `callback`, que delega a correspondência à função `index_after` correspondente no módulo `luagrapheme` (RF-06), que por sua vez utiliza a *libgrapheme* para determinar a posição da próxima unidade textual.

3.5 Referência da API

Esta seção traz uma referência das funções oferecidas pela biblioteca *luagrapheme*, seus parâmetros e valores retornados.

3.5.1 `graphemes(s, start)`

Cria um iterador que retorna os índices iniciais e finais, em *bytes*, de cada *grapheme cluster* na *string* informada (RF-01 e RF-05).

Parâmetros

- `s` (`string`): A *string* a ser percorrida.
- `start` (`integer`, opcional). O índice inicial em `s`. O valor padrão é `1`.

Retorno

Uma função que retorna índices `i` e `j`, em *bytes*, de forma que `s:sub(i, j)` corresponde a um *grapheme cluster*. A cada chamada a iteração avança para o próximo *grapheme cluster*. Retorna `nil` se não houver mais *grapheme clusters* para avançar. Adequada para uso em *generic for statements*.

3.5.2 `words(s, start)`

Cria um iterador que retorna os índices iniciais e finais, em *bytes*, de cada palavra na *string* informada (RF-02 e RF-05).

Parâmetros

- `s` (`string`): A *string* a ser percorrida.
- `start` (`integer`, opcional). O índice inicial em `s`. O valor padrão é `1`.

Retorno

Uma função que retorna índices `i` e `j`, em *bytes*, de forma que `s:sub(i, j)` corresponde a uma palavra. A cada chamada a iteração avança para a próxima palavra. Retorna `nil` se não houver mais palavras para avançar. Adequada para uso em *generic for statements*.

3.5.3 `sentences(s, start)`

Cria um iterador que retorna os índices iniciais e finais, em *bytes*, de cada frase na *string* informada (RF-03 e RF-05).

Parâmetros

- `s` (`string`): A *string* a ser percorrida.
- `start` (`integer`, opcional). O índice inicial em `s`. O valor padrão é `1`.

Retorno

Uma função que retorna índices `i` e `j`, em *bytes*, de forma que `s:sub(i, j)` corresponde a uma frase. A cada chamada a iteração avança para a próxima frase. Retorna `nil` se não houver mais frases para avançar. Adequada para uso em *generic for statements*.

3.5.4 `lines(s, start)`

Cria um iterador que retorna os índices iniciais e finais, em *bytes*, de cada oportunidade de quebra de linha na *string* informada (RF-04 e RF-05).

Parâmetros

- `s` (`string`): A *string* a ser percorrida.
- `start` (`integer`, opcional). O índice inicial em `s`. O valor padrão é `1`.

Retorno

Uma função que retorna índices `i` e `j`, em *bytes*, de forma que `s:sub(i, j)` corresponde a uma linha. A cada chamada a iteração avança para a próxima linha. Retorna `nil` se não houver mais linhas para avançar. Adequada para uso em *generic for statements*.

3.5.5 `graphemes.index_after(s, start)`

Retorna o índice do primeiro *grapheme cluster* estritamente depois do índice `start` na *string* `s` (RF-01 e RF-06).

Parâmetros

- `s` (`string`): A *string* na qual será feita a busca.
- `start` (`integer`, opcional): O índice inicial em `s`, em *bytes*. Deve satisfazer `start >= 1` e `start <= #s + 1`. O valor padrão é `1`.

Retorno

Um inteiro que representa o índice do primeiro *grapheme cluster* estritamente após `start`, em bytes. Retorna `#s + 1` se o final da *string* for alcançado. Se `start` for exatamente `#s + 1`, retorna `nil`.

3.5.6 `words.index_after(s, start)`

Retorna o índice da primeira palavra estritamente depois do índice `start` na *string* `s` (RF-02 e RF-06).

Parâmetros

- `s` (`string`): A *string* na qual será feita a busca.
- `start` (`integer`, opcional): O índice inicial em `s`, em *bytes*. Deve satisfazer `start >= 1` e `start <= #s + 1`. O valor padrão é 1.

Retorno

Um inteiro que representa o índice da primeira palavra estritamente após `start`, em *bytes*. Retorna `#s + 1` se o final da *string* for alcançado. Se `start` for exatamente `#s + 1`, retorna `nil`.

3.5.7 `sentences.index_after(s, start)`

Retorna o índice da primeira frase estritamente depois do índice `start` na *string* `s` (RF-03 e RF-06).

Parâmetros

- `s` (`string`): A *string* na qual será feita a busca.
- `start` (`integer`, opcional): O índice inicial em `s`, em *bytes*. Deve satisfazer `start >= 1` e `start <= #s + 1`. O valor padrão é 1.

Retorno

Um inteiro que representa o índice da primeira frase estritamente após `start`, em *bytes*. Retorna `#s + 1` se o final da *string* for alcançado. Se `start` for exatamente `#s + 1`, retorna `nil`.

3.5.8 `lines.index_after(s, start)`

Retorna o índice da primeira oportunidade de quebra de linha estritamente depois do índice `start` na *string* `s` (RF-04 e RF-06).

Parâmetros

- `s` (`string`): A *string* na qual será feita a busca.
- `start` (`integer`, opcional): O índice inicial em `s`, em *bytes*. Deve satisfazer `start >= 1` e `start <= #s + 1`. O valor padrão é 1.

Retorno

Um inteiro que representa o índice da primeira oportunidade de quebra de linha estritamente após `start`, em *bytes*. Retorna `#s + 1` se o final da *string* for alcançado. Se `start` for exatamente `#s + 1`, retorna `nil`.

3.5.9 `upper(s)`

Converte a *string* informada para letras maiúsculas (RF-08).

Parâmetros

- `s` (`string`): A *string* a ser convertida.

Retorno

Uma nova *string* em que todos os *grapheme clusters* de `s` foram convertidos para letras maiúsculas.

3.5.10 `lower(s)`

Converte a *string* informada para letras minúsculas (RF-08).

Parâmetros

- `s` (`string`): A *string* a ser convertida.

Retorno

Uma nova *string* em que todos os *grapheme clusters* de `s` foram convertidos para letras minúsculas.

3.5.11 `title(s)`

Converte a string informada para *title case* (RF-08).

Parâmetros

- `s` (`string`): A *string* a ser convertida.

Retorno

Uma nova *string* em que o primeiro *grapheme cluster* de cada palavra de `s` foi convertido para maiúsculas, e os restantes foram convertidos para minúsculas.

4 Metodologia

O desenvolvimento do *luagrapheme* combinou pesquisa bibliográfica, implementação em C e Lua e práticas modernas de engenharia de software. O projeto teve como referências principais os textos normativos do Unicode, a documentação da biblioteca *libgrapheme*, o manual de referência da linguagem Lua e o livro Programming in Lua (Ierusalimschy, 2016). Esse conjunto serviu como base teórica e prática para guiar decisões técnicas e alinhar o projeto às melhores práticas do ecossistema.

A implementação seguiu duas frentes complementares. A camada de *bindings*, escrita em C, é responsável por integrar a *libgrapheme* ao ecossistema Lua, expondo uma API idiomática. Em paralelo, as funcionalidades eram verificadas por meio de testes automatizados escritos em Lua (RNF-10).

O desenvolvimento seguiu a abordagem de Test-Driven Development (TDD) proposta por Beck (2003), utilizando ciclos de três etapas: primeiro, escreve-se um teste que falha e estabelece o comportamento esperado (*red*); em seguida, implementa-se o código mínimo necessário para fazê-lo passar (*green*); e, por fim, refatora-se o código para aprimorar sua estrutura garantindo o mesmo comportamento (*refactor*).

A suíte de testes foi construída com a biblioteca *busted,* amplamente utilizada no ecossistema Lua. O objetivo não foi revalidar os algoritmos do Unicode implementados pela *libgrapheme* — que já dispõe de sua própria suíte de testes — mas sim garantir a interoperabilidade entre Lua e C na camada de *bindings*.

4.1 Controle de qualidade

O projeto adotou múltiplas estratégias para assegurar a qualidade do código. A primeira foi a análise estática (RNF-09), realizada com três *linters*:

- *Luacheck*: identifica erros de sintaxe, uso de variáveis globais não definidas, acesso a variáveis não inicializadas, código não alcançável (Melnichenko, 2018);
- *StyLua*: garante a aderência do código em Lua a padrões de estilo definidos pelo projeto, incluindo indentação consistente, limites de colunas para legibilidade e convenções de nomenclatura (JohnnyMorganz, 2023);
- *Clang-format*: similar ao *StyLua*, mas para o código em C (The Clang Team, 2025a).

Além disso, o código em C foi compilado com *flags* conservadoras (The Clang Team, 2025b):

- `-std=c99`: segue rigorosamente o padrão ISO C99, sem extensões;
- `-pedantic`: emite todos os avisos exigidos pelo padrão ISO C;
- `-Wall`: ativa diversos avisos, como variáveis e funções não utilizadas e acessos a variáveis não inicializadas;
- `-Wextra`: habilita ainda mais avisos, como estruturas com campos não inicializados explicitamente;
- `-Werror`: trata qualquer aviso como um erro de compilação.

Por fim, a análise dinâmica contou com testes unitários automatizados e com o uso de ferramentas de instrumentação, em especial o *Address Sanitizer* (ASan) e o *Undefined Behavior Sanitizer* (UBSan). Ambos fazem parte dos compiladores GCC e Clang e inserem verificações em tempo de execução para detectar falhas de memória e comportamentos indefinidos (RNF-08).

O ASan identifica problemas como acessos fora dos limites em *heap*, *stack* ou objetos globais, além de erros de *use-after-free* (Serebryany et al., 2012). Já o UBSan cobre uma gama de situações de comportamentos indefinidos na linguagem C, como *overflows* de inteiros com sinal, acessos a *arrays* fora dos limites conhecidos em tempo de compilação ou *bit shifts* inválidos (The Clang Team, 2024). Ao encontrar tais condições, essas ferramentas emitem diagnósticos detalhados e interrompem a execução, permitindo que os erros sejam corrigidos ainda na fase de testes.

A combinação de análise estática, testes automatizados e *sanitizers* forneceu múltiplas camadas de segurança contra falhas, reforçando a robustez da biblioteca.

4.2 Controle de versão

A gestão do código-fonte do projeto e da documentação foi realizada por meio do Git, com repositório hospedado no GitHub. Esta escolha refletiu não apenas uma aderência às melhores práticas da indústria, mas também uma estratégia para garantir rastreabilidade e histórico completo das modificações no código-fonte (Chacon; Straub, 2025), além de proporcionar maior alcance à biblioteca e incentivar a contribuição da comunidade de software livre.

O fluxo de trabalho adotado seguiu práticas de integração contínua, em que as alterações são integradas frequentemente a uma única *branch* principal

(Duvall; Matyas; Glover, 2013). Cada *branch* adicional teve escopo restrito, representando uma funcionalidade ou melhoria específica, e era rapidamente integrado à *branch* principal após validação. O histórico de *commits* foi mantido cuidadosamente, com mensagens explicativas e padronizadas, favorecendo a compreensão do escopo e da motivação de cada mudança. Esse fluxo contribuiu para a organização do projeto ao longo do tempo, o desenvolvimento paralelo de melhorias, e a documentação das decisões tomadas.

4.3 Integração contínua

A integração contínua (CI) é uma prática de engenharia de software que automatiza a construção, os testes e a validação de um sistema a cada alteração no código-fonte. Seu principal objetivo é detectar falhas precocemente, mantendo o sistema funcional mesmo diante de mudanças frequentes (Duvall; Matyas; Glover, 2013).

No projeto *luagrapheme*, a CI foi implementada com o GitHub Actions, serviço nativo da plataforma que permite definir fluxos de trabalho executados automaticamente em servidores do GitHub. Os *merges* na *branch* principal só ocorrem após a execução bem-sucedida de todos os fluxos configurados (RNF-11).

Com o objetivo de garantir o correto funcionamento da biblioteca, foram definidos três fluxos de trabalho:

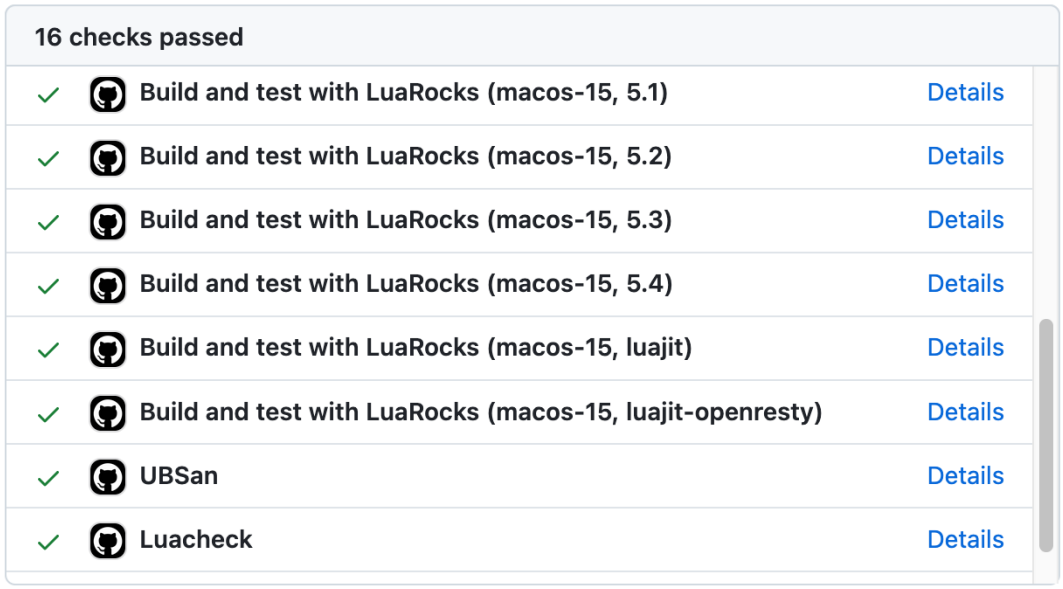
- *Lint*: análise estática do código por meio das ferramentas *StyLua*, *Clang-format* e *Luacheck*, conforme detalhado anteriormente.
- *Build and test*: compilação da biblioteca e execução dos testes automatizados.
- *Build and test with sanitizers*: compilação instrumentada com *ASan* e *UBSan*, seguida da execução de testes automatizados.

Como a biblioteca é compilada para código nativo e foi projetada para funcionar com uma variedade de versões do interpretador Lua (RNF-01 e RNF-02), o fluxo *Build and test* é executado em várias combinações de sistemas operacionais e interpretadores Lua disponíveis no GitHub Actions: macOS (arm64) e Ubuntu 24.04 (x86_64), e as versões Lua 5.1, 5.2, 5.3, 5.4, *LuaJIT* e a distribuição do *LuaJIT* usada pelo *OpenResty*.

Assim, nenhuma modificação é integrada à *branch* principal sem passar por todas as verificações (Figura 18), garantindo estabilidade e portabilidade da

biblioteca. Como os testes possuem a mesma entrada e saída esperada para todas as plataformas e versões, essa etapa também ajuda a assegurar o comportamento determinístico da biblioteca (RNF-06).

Figura 18 – Verificações realizadas no processo de integração contínua



5 Considerações finais

O *luagrapheme* atingiu o propósito que motivou seu desenvolvimento: disponibilizar ao ecossistema Lua uma biblioteca pequena, precisa e previsível para segmentação de texto Unicode e *case folding*, com integração direta ao *LPeg*. Ao adotar UTF-8 como contrato de entrada, expor operações de segmentação por meio de iteradores (RF-05) e devolver índices em *bytes* prontos para uso com `string.sub` e funções similares, o projeto privilegiou um desenho de API que se encaixa no modo como se programa em Lua, sem impor estruturas próprias nem "tomar posse" dos dados das aplicações. Esse recorte deliberado permitiu construir uma solução enxuta que resolve problemas comuns do dia a dia sem o custo e a complexidade de uma suíte Unicode completa como a ICU.

A opção por bindings em C para a *libgrapheme* mostrou-se acertada tanto técnica quanto pragmaticamente. A biblioteca final, incluindo a *libgrapheme* linkada estaticamente (RNF-03), pesa cerca de 380 KB, em claro contraste com os ~30 MB da ICU; essa diferença viabiliza usos em ambientes computacionais mais restritos, facilita a distribuição e reduz a superfície de manutenção (RNF-05). Em paralelo, a integração com *LPeg* (RF-07) ampliou as possibilidades de uso: gramáticas escritas em Lua podem apoiar-se diretamente nos algoritmos de segmentação do Unicode, sem reimplementações, incorporando suas regras de forma transparente em linguagens de domínio específico ou *parsers* personalizados.

Do ponto de vista de engenharia, o projeto conciliou portabilidade e ergonomia. A compatibilidade com Lua 5.1, 5.2, 5.3, 5.4 e *LuaJIT* (RNF-01) foi alcançada sem comprometer a clareza do código ao concentrar as diferenças na *compat53*, incluída no repositório. A biblioteca compila em qualquer sistema que ofereça um compilador ISO C99, o POSIX *make* e um interpretador Lua funcional (RNF-02), tendo sido validada, na prática, no macOS (arm64) e no Linux (x86_64). Em termos de qualidade, o ciclo de desenvolvimento combinou TDD, análise estática e dinâmica e um processo de integração contínua, o que resultou em um código mais robusto e num processo de evolução mais confiável (RNF-08, RNF-09, RNF-10 e RNF-11).

Os exemplos apresentados demonstram que a biblioteca responde a necessidades concretas: percorrer *grapheme clusters* (incluindo emojis e sistemas de escrita como o Hangul) sem "quebrar" a unidade percebida pelo usuário como um caractere (RF-01); segmentar palavras, frases e oportunidades

de quebras de linha segundo os anexos normativos do Unicode (RF-02, RF-03 e RF-04); e realizar *case folding* coerente e independente de *locale* (RF-08 e RNF-06). Esses casos de uso, embora simples, são representativos do tipo de tarefa que aparece em editores, interfaces interativas, sistemas de busca e pipelines de processamento de linguagem natural, e sustentam a tese de que é possível oferecer suporte Unicode significativo em Lua sem abrir mão da leveza e simplicidade.

Há espaço natural para continuidade. Os algoritmos de normalização Unicode (NFC, NFD, NFKC, NFKD) surgem como extensão de alto valor para cenários que exigem comparações de *strings*; a ampliação das plataformas suportadas — inclusive documentação e *tooling* para Windows — pode aumentar o alcance do projeto. Esses desdobramentos preservam a filosofia do projeto ao mesmo tempo em que atendem novas demandas quando elas de fato se apresentarem.

Do ponto de vista formativo, o *luagrapheme* sintetiza competências centrais da minha formação como Bacharel em Ciência da Computação na engenharia de software: compreender padrões e bibliografias técnicas, projetar APIs coerentes com o ambiente-alvo, justificar escolhas de implementação, testar e medir antes de integrar, e sustentar a evolução com ferramentas e práticas que reduzem incerteza. O trabalho materializa a passagem da teoria à prática — do texto normativo do Unicode a uma biblioteca concreta, pequena e útil — e reforça a convicção de que boas soluções nascem do equilíbrio entre rigor técnico, simplicidade e cuidado com a experiência de quem vai usá-las.

Referências

APPLE INC. **Character**. Disponível em: <https://developer.apple.com/documentation/swift/character>. Acesso em: 26 jan. 2024.

BECK, Kent. **Test-driven development: by example**. 1. ed. Boston: Addison-Wesley, 2003.

CHACON, Scott; STRAUB, Ben. **Pro Git**. 2. ed. *[S.l.]*: Apress, 2025.

DUNCANC. **ICU4Lua**, 30 abr. 2009. Disponível em: <https://github.com/duncanc/icu4lua>. Acesso em: 19 nov. 2025

DUVALL, Paul M.; MATYAS, Steve; GLOVER, Andrew. **Continuous integration: improving software quality and reducing risk**. 8. print ed. Upper Saddle River, NJ: Addison-Wesley, 2013.

HUNHOLD, Laslo. **libgrapheme**, 22 dez. 2021. Disponível em: <https://libs.suckless.org/libgrapheme/>. Acesso em: 26 jan. 2024

IEEE; THE OPEN GROUP. **Portable Operating System Interface (POSIX™) Base Specifications, Issue 8**. *[S.l.]*: IEEE, 2024.

IERUSALIMSCHY, Roberto. A text pattern-matching tool based on Parsing Expression Grammars. **Software: Practice and Experience**, v. 39, n. 3, p. 221–258, 10 mar. 2009.

IERUSALIMSCHY, Roberto. **Programming in Lua**. 4. ed. Rio de Janeiro: Lua.org, 2016.

IERUSALIMSCHY, Roberto. **LPeg - Parsing Expression Grammars For Lua**. Disponível em: <https://www.inf.puc-rio.br/~roberto/lpeg/>. Acesso em: 24 nov. 2025.

IERUSALIMSCHY, Roberto; FIGUEIREDO, Luiz Henrique de; CELES, Waldemar. **Lua 5.4 Reference Manual**. Disponível em: <https://www.lua.org/manual/5.4/manual.html>. Acesso em: 26 jan. 2024.

JANDA, Philipp *et al.* **lunarmodules/lua-compat-5.3**, 22 jan. 2015. Disponível em: <https://github.com/lunarmodules/lua-compat-5.3>. Acesso em: 22 set. 2025

JOHNNYMORGANZ. **StyLua: A Lua code formatter**, 12 nov. 2023. Disponível em: <https://github.com/JohnnyMorganz/StyLua>. Acesso em: 11 nov. 2025

KRÜGER, Marcel. **Unicode algorithms for LuaTEX**, 13 mar. 2020. Disponível em: <https://ctan.math.washington.edu/tex-archive/macros/luatex/generic/lua-uni-algos/lua-uni-algos.pdf>. Acesso em: 19 set. 2025

LUA USERS. **lua-users wiki: Lua Unicode**. Disponível em: <http://lua-users.org/wiki/LuaUnicode>. Acesso em: 11 nov. 2025.

LUAROCKS. **LPeg**. Disponível em: <https://luarocks.org/modules/gvvaughan/lpeg>. Acesso em: 11 nov. 2025.

MELNICHENKO, Peter. **Luacheck: A tool for linting and static analysis of Lua code**. , 2018. Disponível em: <https://github.com/mpeterv/luacheck>. Acesso em: 11 nov. 2025

PYTHON SOFTWARE FOUNDATION. **Built-in Types**. Disponível em: <https://docs.python.org/3.12/library/stdtypes.html>. Acesso em: 26 jan. 2024.

SEREBRYANY, K. *et al.* {AddressSanitizer}: A fast address sanity checker. *In*: 2012 USENIX ANNUAL TECHNICAL CONFERENCE (USENIX ATC 12), 2012. p. 309–318. Disponível em: https://www.usenix.org/system/files/conference/atc12/atc12-final39.pdf. Acesso em: 20 set. 2025.

THE CLANG TEAM. **UndefinedBehaviorSanitizer**. Disponível em: <https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html#ubsan-checks>. Acesso em: 27 jan. 2024.

THE CLANG TEAM. **ClangFormat documentation**. Disponível em: <https://clang.llvm.org/docs/ClangFormat.html>. Acesso em: 11 nov. 2025a.

THE CLANG TEAM. **Clang Compiler User's Manual**. Disponível em: <https://clang.llvm.org/docs/UsersManual.html>. Acesso em: 11 nov. 2025b.

THE UNICODE CONSORTIUM. **International Components for Unicode**. , 1999. Disponível em: <https://icu.unicode.org/>. Acesso em: 13 fev. 2024

THE UNICODE CONSORTIUM. **Unicode Technical Report #17: Unicode Character Encoding Model, Revision 9**. Mountain View, CA: The Unicode Consortium, 2022.

THE UNICODE CONSORTIUM. **Unicode Standard Annex #14: Unicode Line Breaking Algorithm, Version 15.1.0**. South San Francisco, CA: The Unicode Consortium, 2023a.

THE UNICODE CONSORTIUM. **Unicode Standard Annex #29: Unicode Text Segmentation, Version 15.1.0**. South San Francisco, CA: The Unicode Consortium, 2023b.

THE UNICODE CONSORTIUM. **The Unicode Standard, Version 15.1.0**. South San Francisco, CA: The Unicode Consortium, 2023c.

W3TECHS. **Usage Survey of Character Encodings broken down by Ranking**. Disponível em: <https://w3techs.com/technologies/cross/character_encoding/ranking>. Acesso em: 26 jan. 2024.

WANG, Xavier. **lua-utf8**, 2019. Disponível em: <https://github.com/starwing/luautf8>. Acesso em: 19 nov. 2025

WORLD WIDE WEB CONSORTIUM. **Character Model for the World Wide Web 1.0: Fundamentals**. *[S.l.]*: World Wide Web Consortium, 2005.

YERGEAU, François. **UTF-8, a transformation format of ISO 10646**. *[S.l.]*: Internet Engineering Task Force, nov. 2003. Disponível em: <https://datatracker.ietf.org/doc/rfc3629>. Acesso em: 26 jan. 2024.