CENTRO UNIVERSITÁRIO DINÂMICA DAS CATARATAS
FRANCISCO SEBASTIANY JUNIOR

DESIGN E IMPLEMENTAÇÃO DE UM PROTÓTIPO PARA UMA LINGUAGEM DE PROGRAMAÇÃO ORIENTADA AO ENTITY COMPONENT SYSTEM

FOZ DO IGUAÇU
2025

FRANCISCO SEBASTIANY JUNIOR

DESIGN E IMPLEMENTAÇÃO DE UM PROTÓTIPO PARA UMA LINGUAGEM DE PROGRAMAÇÃO ORIENTADA AO ENTITY COMPONENT SYSTEM

Trabalho de Conclusão de Curso apresentado como requisito obrigatório para obtenção do título de Bacharel em Ciência da Computação do Centro Universitário Dinâmica das Cataratas.

Orientador: Prof. Me. Luciano S. Cardoso

FOZ DO IGUAÇU

2025

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE CÓDIGOS

LISTA DE ABREVIATURAS

AST	*Abstract Syntax Tree* - Árvore Sintática Abstrata
ECS	*Entity Component System*[1]
LSP	*Language Server Protocol* - Protocolo de Servidor de Linguagem
MVC	*Model-View-Controller*[1]

[1] Tradução não usual; o termo em inglês é predominante na literatura técnica.

SUMÁRIO

1 INTRODUÇÃO

Nos últimos anos, as empresas têm adotado novas abordagens para o desenvolvimento de software, especialmente as metodologias ágeis. Sabendo que tais metodologias favorecem a flexibilidade ao invés de planejamento rígido (Agile Alliance, 2001), vem a necessidade de um modelo arquitetural que permite uma adaptação rápida a novas demandas. Um dos modelos que atendem esses requisitos é o padrão arquitetural *Entity Component System* (ECS).

O padrão ECS surgiu na área de desenvolvimento de jogos, com um dos fatores sendo a alta necessidade de adaptação rápida na indústria. Por mais que ECS continue sendo majoritariamente aplicado em jogos, sua utilidade expande para qualquer aplicação que dependa fortemente de rápida iteração de desenvolvimento, flexibilidade ou desempenho (WILLIS, 2021).

Frequentemente, o padrão ECS é abstraído em forma de biblioteca, em uma determinada linguagem de programação. Isso é devido ao fato de que a implementação do padrão, principalmente de forma eficiente, é composta de vários detalhes técnicos, como a organização dos dados na memória (Mertens, 2024). Muitas vezes, tais detalhes são irrelevantes para o desenvolvedor, e por isso eles são ocultados pela interface da biblioteca.

Dado isso, o projeto visa abstrair o padrão ECS através de um método diferente — um protótipo para uma linguagem de programação orientada a ECS. Será abordado desde as escolhas de *design* da linguagem até a implementação do interpretador, por fim examinando, de forma qualitativa, se foi possível alcançar os objetivos determinados.

1.1 PERGUNTA PROBLEMA

Como projetar e desenvolver um protótipo de interpretador para uma linguagem de programação orientada a ECS, aproveitando suas vantagens nativas para uma melhor abstração do padrão?

1.2 OBJETIVOS

1.2.1 Objetivo Geral

Investigar como o *design* e a implementação de um protótipo de interpretador para uma linguagem de programação orientada a ECS podem ser efetuados.

1.2.2 Objetivos Específicos

- Definir os requisitos e princípios de *design* da linguagem;
- Implementar um protótipo de interpretador funcional;
- Avaliar o impacto e a viabilidade do protótipo.

1.3 JUSTIFICATIVA

Com o crescimento da adoção de metodologias ágeis pelas empresas, vem a demanda por arquiteturas que promovam flexibilidade no desenvolvimento de software (WILLIS, 2021). O padrão ECS tem se destacado por atender tais demandas, especialmente nas áreas de desenvolvimento de jogos e simulações.

A maioria das abstrações feitas sobre ECS estão no formato de bibliotecas específicas para determinadas linguagens de programação, limitando a expressividade do desenvolvedor no processo de abstração.

Dado isso, este trabalho propõe uma rota alternativa: a criação do *design* e implementação de uma linguagem de programação orientada a ECS. Com a capacidade de moldar a linguagem diante dos requisitos específicos do padrão ECS, essa rota propõe investigar e implementar abstrações que são difíceis ou até mesmo impossíveis de serem aplicadas em bibliotecas.

2 FUNDAMENTAÇÃO TEÓRICA

A seguir são definidos os fundamentos e tecnologias usadas ao decorrer do projeto. O foco está no padrão arquitetural *Entity Component System* e no interpretador *tree-walking*, que são os principais pilares do projeto.

2.1 FUNDAMENTOS

2.1.1 Padrão de Design de Software

Um padrão de *design* de software (do inglês, *software design pattern*) é uma solução reutilizável para um determinado problema recorrente no design de software. Esses padrões são descrições gerais de como resolver tais problemas, e não implementações específicas.

De acordo com Freeman, Robson e Bates (2004), baseado em Gamma et al. (1994), os padrões podem ser classificados em três categorias:

- Padrões Criacionais: tratam da criação de objetos, focando em como instanciá-los de forma a resolver problemas específicos. Abrange padrões como *Singleton*, *Builder* e *Prototype*;
- Padrões Estruturais: tratam da composição de objetos em uma estrutura maior, além de como eles interagem entre si. Abrange padrões como *Adapter*, *Decorator* e *Facade*;
- Padrões Comportamentais: tratam de algoritmos e a atribuição de responsabilidades entre objetos. Abrange padrões como *Observer*, *Strategy* e *Visitor*.

É importante notar como o autor explica os vários padrões usando o paradigma de programação orientado a objetos — por mais que esse seja o caso, nem todos os padrões de *design* são exclusivos a esse paradigma. Um exemplo é o padrão *Observer*, cujo conceito é de grande importância para o paradigma de programação reativa (do inglês, *reactive programming*) (MICROSOFT, 2009).

2.1.2 Padrão de Arquitetura de Software

Um padrão de arquitetura de software (do inglês, *software architecture pattern*), assim como um padrão de *design* de software, é uma solução reutilizável para um problema recorrente, só que dessa vez na arquitetura de software. Enquanto um padrão de *design* foca em resolver problemas mais específicos no código, um padrão de arquitetura foca em resolver problemas mais amplos, como a organização de toda a aplicação e a interação entre seus diversos componentes (RICHARDS; FORD, 2020).

Pode-se dizer que um dos maiores exemplos de padrão arquitetural é o *Model-View-Controller* (MVC), que separa a aplicação em três componentes distintos a fim de isolar as várias responsabilidades da aplicação. Porém, neste projeto, será utilizado outro padrão arquitetural: o *Entity Component System*.

2.1.3 Paradigma de Programação

Um paradigma de programação pode ser definido como um conjunto de princípios que orientam o desenvolvimento de software, e que, consequentemente, geram um software estruturado de uma maneira específica ao paradigma (IONOS, 2020).

Os paradigmas são organizados em categorias, podendo ser vistos como uma hierarquia, assim como mostra a Figura 1:

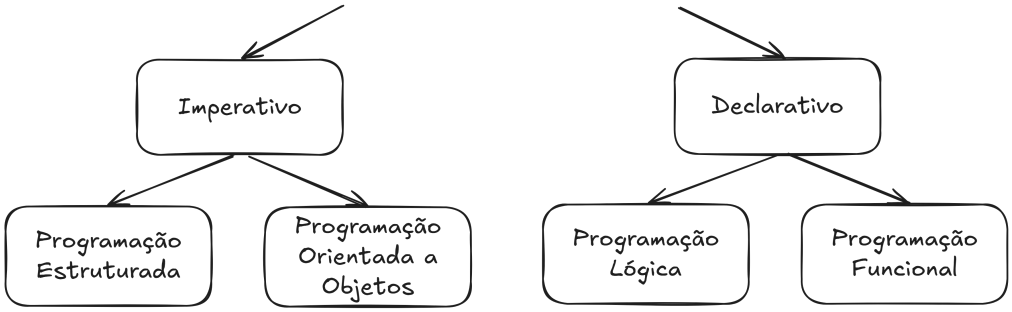

FIGURA 1 – Paradigmas de programação organizados hierarquicamente.
FONTE: Adaptado de Liyan (2023).

Com base na Figura 1, observa-se a relevância dos paradigmas imperativo e declarativo, já que todos os outros paradigmas derivam, em maior ou menor grau, de um desses dois. Por mais que paradigmas mais específicos possam tomar rumos inesperados, eles ainda terão algum grau de herança dos paradigmas imperativo ou declarativo.

Dada a abrangência dos paradigmas imperativo e declarativo, o conhecimento de ambos já se torna suficiente para que o leitor entenda a essência da maioria dos outros paradigmas. Assim, a seguir, eles serão abordados:

- Paradigma imperativo: descreve a aplicação em termos de instruções que alteram o estado do programa, linha por linha. O foco está *em como* fazer algo, geralmente oferecendo maior controle e menor abstração. Exemplos de linguagens imperativas incluem C, Java e Python;
- Paradigma declarativo: descreve a aplicação em termos de declarações que expressam *o que* o programa deve fazer, e não *em como* fazer. Esta abordagem costuma ser mais abstrata, porém com menos controle. Exemplos de linguagens declarativas incluem SQL, Haskell e Lisp.

2.1.3.1 O Entity Component System pode ser considerado um paradigma?

O padrão *Entity Component System* (ECS) será extremamente crucial em todas as fases do projeto, por isso, é importante entender em que definição ele se encaixa.

Com base na definição de paradigma dada pelo dicionário Merriam-Webster:

> Uma estrutura filosófica e teórica de uma escola ou disciplina científica dentro da qual teorias, leis e generalizações e os experimentos realizados em apoio a elas são formulados (MERRIAM-WEBSTER, s.d., tradução nossa).

O autor da biblioteca de ECS Flecs, Mertens (2021), defende que o ECS não é um paradigma:

> Isso não corresponde exatamente ao ECS. Ter pensamentos profundos sobre o ECS não é o mesmo que ter uma estrutura filosófica. Tutoriais, documentação e postagens em blogs não equivalem a "teorias, leis e generalizações". É seguro dizer que a base teórica para o ECS é, na melhor das hipóteses, instável (MERTENS, 2021, tradução nossa).

Seguindo a conclusão de Mertens (2021), o ECS não será tratado como um paradigma ao decorrer do projeto, mas sim apenas como um padrão de arquitetura.

2.1.4 Entity Component System

Entity Component System (ECS) é um padrão arquitetural baseado no *design* orientado a dados. Ele surgiu na área de desenvolvimento de jogos, onde há uma grande necessidade de otimização

e atualizações frequentes no código. Com o passar do tempo, o padrão ECS começou a ser utilizado em outras áreas, como em simulações físicas (WILLIS, 2021).

O padrão consiste na separação de dado e lógica de tal forma que as vários entidades da aplicação possam ser compostas de dados reutilizáveis e independentes (MERTENS, 2019a), com as funções sendo direcionadas aos dados, e não às entidades em si. Devido ao desacoplamento gerado por essa separação, o padrão ECS garante alta flexibilidade e modularidade, além do aumento de desempenho gerado pela melhor distribuição de dados na memória (MERTENS, 2024).

Neste projeto, o padrão ECS será um dos principais fundamentos para o *design* e implementação da linguagem de programação, já que o intuito dela será abstrair ele.

2.1.4.1 Os Três Elementos Fundamentais do ECS

Pode-se dizer que o ECS é separado em três elementos fundamentais: entidades, componentes e sistemas (MERTENS, 2019a). Cada um desses elementos desempenha um papel específico na aplicação:

- Entidades: identificadores únicos que representam os vários conceitos de uma aplicação. Sozinhas, as entidades não contêm dados nem funcionalidade;
- Componentes: estruturas de dados que armazenam informações específicas. Uma entidade pode ter múltiplos componentes diferentes, definindo suas características;
- Sistemas: funções responsáveis por processar sobre entidades com um determinado conjunto de componentes — processo denominado *querying*.

Como ilustrado na Figura 2, o estado da aplicação é dado por um conjunto de entidades, cada uma com seus respectivos componentes. Os sistemas são responsáveis pela transformação do estado da aplicação, processando as entidades que possuem os componentes necessários para a execução do sistema.

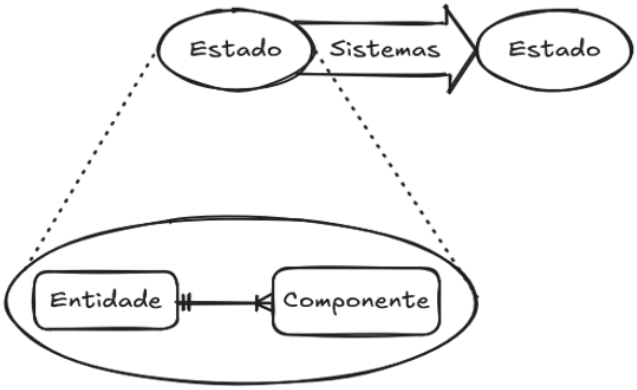

FIGURA 2 – Relação entre entidades, componentes e sistemas.
FONTE: Elaboração própria.

Em termos de código, o padrão ECS pode ser representado sem nenhum construto especializado, mapeando entidades para números únicos, componentes para *structs* e sistemas para funções, como ilustrado no Código 1.

CÓDIGO 1 – Implementação simplificada de um padrão ECS incompleto.

```
// Componentes podem ser representados através de simples structs.
struct Position { x: f32, y: f32 }

struct Velocity { dx: f32, dy: f32 }

// Sistemas podem ser representados como funções que processam todas as
// entidades e seus respectivos componentes.
fn apply_velocity(
  entities: &[usize],
  positions: &mut [Position],
  velocities: &[Velocity]
) {
  for &entity in entities {
    positions[entity].x += velocities[entity].dx;
    positions[entity].y += velocities[entity].dy;
  }
}

fn main() {
  // Entidades podem ser representadas como números únicos.
  let entities = [0, 1];

  // Cada componente é armazenado em um vetor separado, onde o índice
  // representa a entidade e o valor representa seu componente.
  let mut positions = [
    Position { x: 0.0, y: 0.0 }, // Entidade 0.
    Position { x: 1.0, y: 1.0 }, // Entidade 1.
  ];

  let velocities = [
    Velocity { dx: 1.0, dy: 1.0 }, // Entidade 0.
    Velocity { dx: 2.0, dy: 2.0 }, // Entidade 1.
  ];

  // No padrão ECS, é muito comum que os sistemas sejam executados ↩
    repetidamente.
  loop {
    apply_velocity(&entities, &mut positions, &velocities);
  }
}
```

FONTE: Elaboração própria.

É importante ressaltar que o Código 1, por mais que seja funcional e siga o *design* orientado a dados, ainda é uma simplificação da implementação de um padrão ECS incompleto. Na prática, o armazenamento dos dados é feito através de estruturas de dados mais complexas (MERTENS, 2024), que permitem que entidades escolham quais componentes possuem, que sistemas sejam executados automaticamente, além de outras funcionalidades principais do padrão ECS.

Fora a definição de ECS e seus três elementos fundamentais, o padrão ainda peca pela falta de formalização — quais são as práticas recomendadas ao usar ECS? Como os sistemas são executados? E se apenas entidades, componentes e sistemas não forem suficientes para resolver meu problema? Essas perguntas não possuem respostas definitivas, porém, diferentes autores e implementações abordam o padrão do seu jeito. A seguir, são apresentados alguns conceitos herdados de tais autores e implementações.

2.1.4.2 Agendador

O agendador é um construto com a finalidade de executar todos os sistemas da aplicação, podendo determinar a ordem e frequência de execução de forma declarativa, resolvendo dependência entre sistemas e tornando o ciclo de atualização da aplicação mais previsível (Bevy Foundation, 2020). Pode-se dizer que, dentre os conceitos mais experimentais, o agendador é o mais próximo de uma formalização.

A Figura 3 ilustra o agendador executando os sistemas de forma cíclica e sequencial, onde cada sistema é executado uma vez por ciclo. O agendador pode ser configurado para executar sistemas em diferentes momentos do ciclo, como antes ou depois de outros sistemas.

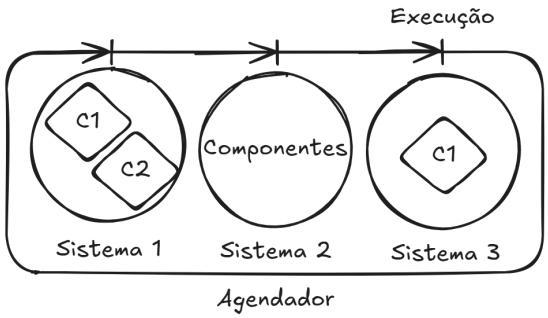

FIGURA 3 – Agendador executando os sistemas de forma cíclica e sequencial.

FONTE: Elaboração própria.

2.1.4.3 Relacionamento de Entidades

Independente da aplicação, é muito comum a necessidade de relacionar diferentes conceitos entre si. Exemplo disso são os sistemas de arquivos, onde pastas podem conter arquivos, uma relação pai-filho.

Relacionamento de entidades (do inglês, *entity relationship*) é um conceito que supre essa necessidade, permitindo que entidades se relacionem entre si. O autor da biblioteca Flecs, Mertens (2019b), explica o conceito fazendo uma paralela com o simples ato de adicionar um componente a uma entidade, como ilustrado na Figura 4.

FIGURA 4 – Representação do relacionamento entre o Sol, a Terra e a Lua.

FONTE: Adaptado de Mertens (2022).

Do mesmo jeito que se adiciona um único componente a uma entidade, como mostra a Figura 4, pode-se criar um relacionamento entre duas entidades adicionando uma tupla componente-entidade, onde o componente dita o tipo de relação.

Como exemplo, a Figura 5 ilustra o relacionamento entre o Sol, a Terra e a Lua, onde a Terra é filha do Sol e a Lua é filha da Terra.

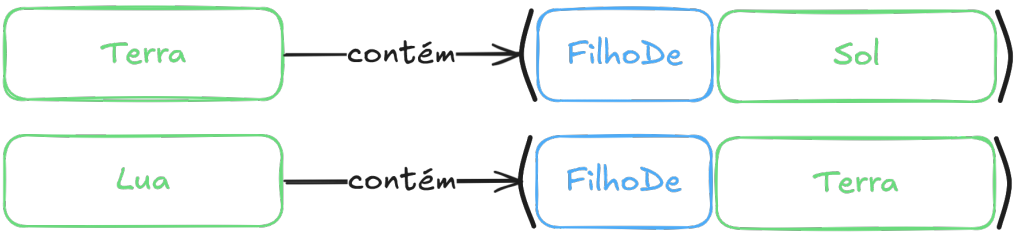

FIGURA 5 – Representação do relacionamento entre o Sol, a Terra e a Lua.

FONTE: Adaptado de Mertens (2022).

2.1.5 Interpretador *Tree-Walking*

Um interpretador é um programa que executa diretamente o código fonte de uma linguagem de programação, linha por linha. No caso deste trabalho, será utilizada a variante *tree-walking*, que simplifica o método de execução do interpretador ao custo de desempenho (NYSTROM, 2021). Esta variante se adequa ao objetivo do trabalho, que visa a implementação de um protótipo simples, sem preocupações com desempenho.

A Figura 6 ilustra uma montanha fazendo analogia a alguns dos principais caminhos que a implementação de uma linguagem de programação pode seguir. Como será implementado um interpretador *tree-walking* neste trabalho, o caminho pela montanha será o da fase de análise léxica, análise sintática, análise semântica e, por fim, evitando a descida pela montanha, a interpretação direta da representação intermediária gerada pela análise semântica.

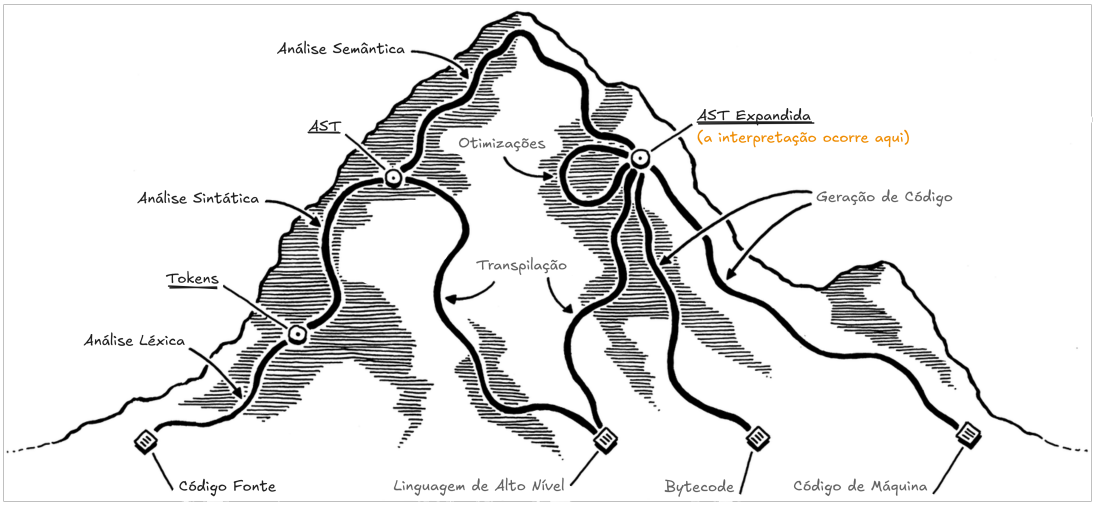

FIGURA 6 – Mapa do território de um interpretador.
FONTE: (NYSTROM, 2021).

A seguir, serão definidas em detalhe cada uma das quatro principais fases de um interpretador.

2.1.5.1 Análise Léxica

A análise léxica, também conhecida como *lexer*, é a primeira fase do processo de interpretação. Nesta fase, o código fonte é lido e dividido em unidades chamadas *tokens*. Cada *token* representa uma palavra ou símbolo da linguagem de programação, como palavras-chave e operadores. Além disso, cada *token* costuma conter informações adicionais a respeito da palavra, como sua localização ou valor agregado, caso ela possua algum (NYSTROM, 2021).

A Figura 7 ilustra o código fonte `8 + 4 / 2` sendo mapeado para uma lista de *tokens*, onde cada um representa sua determinada palavra do código fonte.

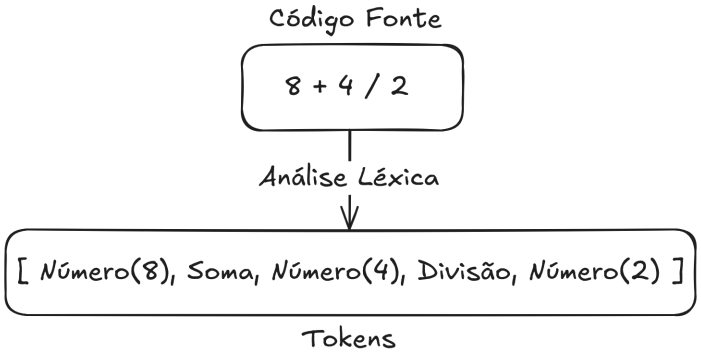

FIGURA 7 – Código fonte sendo mapeado para uma lista de tokens pela análise léxica.
FONTE: Elaboração própria com base em Nystrom (2021).

2.1.5.2 Análise Sintática

A análise sintática, também conhecida como *parser*, é a segunda fase do processo de interpretação. Nesta fase, os *tokens* gerados na fase de análise léxica são organizados em uma estrutura hierárquica chamada árvore sintática abstrata (do inglês, *abstract syntax tree* — AST). A AST representa a estrutura do código fonte de acordo com as regras gramaticais da linguagem de programação (NYSTROM, 2021).

A Figura 8 ilustra a lista de *tokens* sendo mapeada para uma AST, assim gerando a ordem de procedência dos operadores.

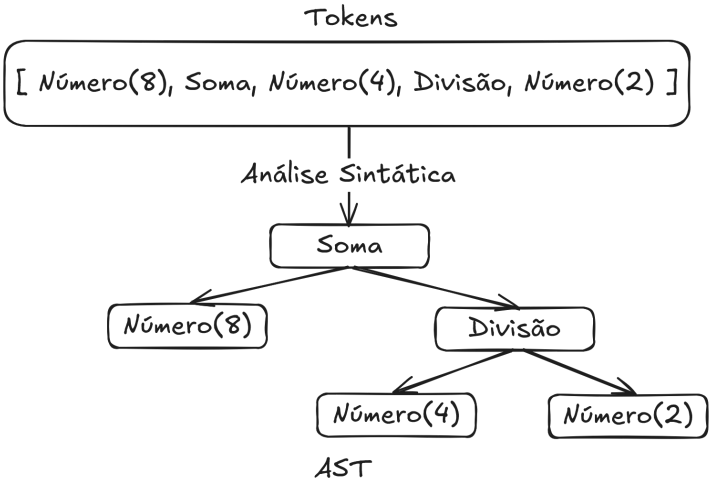

FIGURA 8 – Tokens sendo organizados em uma AST pela análise sintática.
FONTE: Elaboração própria com base em Nystrom (2021).

2.1.5.3 Análise Semântica

A análise semântica é uma fase opcional[1] do processo de interpretação, que visa verificar a consistência e validade do código fonte, além de anotar a AST com informações adicionais, como o tipo de cada expressão (NYSTROM, 2021).

Durante esta fase, o interpretador analisa a AST gerada na fase de análise sintática para garantir que as operações e expressões estejam corretas de acordo com as regras da linguagem (NYSTROM, 2021). Por exemplo, pode verificar se variáveis foram declaradas antes de serem usadas ou se os tipos de dados são compatíveis.

[1] Neste trabalho, a análise semântica será implementada.

A Figura 9 ilustra a AST sendo anotada com informações adicionais, como o tipo de cada expressão, resultando em uma AST com mais conhecimento sobre o código fonte.

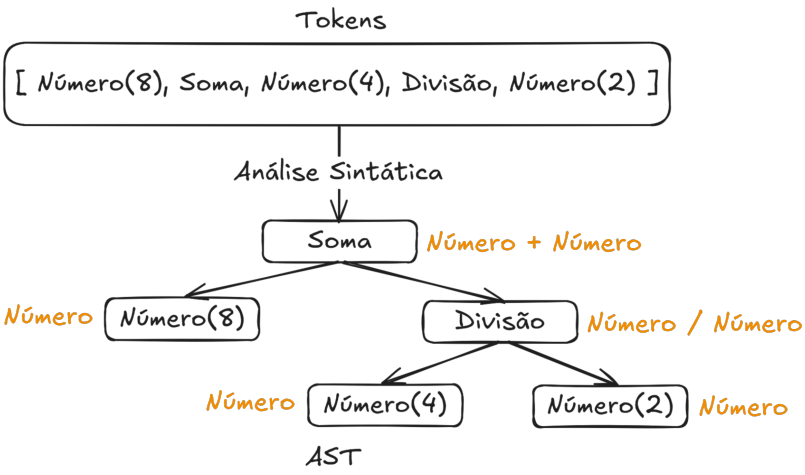

FIGURA 9 – AST sendo anotada com informações adicionais pela análise semântica.
FONTE: Elaboração própria com base em Nystrom (2021).

2.1.5.4 Interpretação

A interpretação é a última fase do processo de interpretação. Nesta fase, a AST gerada na fase de análise sintática e modificada pela análise semântica é percorrida e executada. Durante esta fase, o interpretador avalia expressões, atualiza o estado do programa e executa funções (NYSTROM, 2021).

A Figura 10 ilustra o processo de interpretação, onde a AST é percorrida e executada pelo interpretador, resultando no número 10.

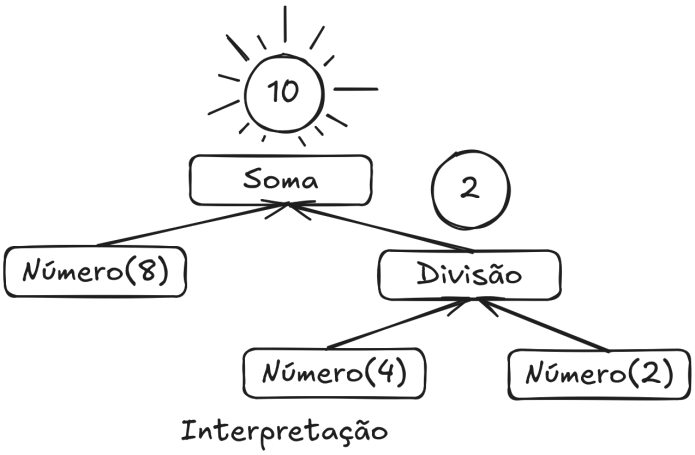

FIGURA 10 – AST sendo percorrida e executada pela fase de interpretação.
FONTE: Elaboração própria com base em Nystrom (2021).

É importante ressaltar que é na interpretação que o tempo de execução (do inglês, *runtime*) do programa é determinado, pois é quando as expressões são avaliadas e os resultados são produzidos. Em contrapartida, a análise léxica, sintática e semântica são fases de preparação, caracterizando o tempo de compilação (do inglês, *compile time*) do programa.

2.2 TECNOLOGIAS

2.2.1 Linguagem de Programação Rust

A linguagem de programação utilizada para o desenvolvimento do interpretador será Rust. A motivação por trás da escolha se dá pelos seguintes fatos:

- Possui um sistema de tipagem forte — o uso de enum e match é especialmente útil na definição dos tokens e na construção da AST (KLABNIK; NICHOLS, 2022);
- O tratamento de erros é explícito, indicando com clareza quais partes do código precisam ser tratadas adequadamente. Todas as fases de um interpretador estão sujeitas a erros, e por isso, tratá-los do jeito mais claro possível é benéfico para o estudo do código (KLABNIK; NICHOLS, 2022);
- Possui alto desempenho, muitas vezes comparado ao de C. Desempenho é importante não só para ECS em si, mas também para qualquer interpretador, minimizando o tempo que o desenvolvedor espera pela execução de seu código (KLABNIK; NICHOLS, 2022).

2.2.2 Biblioteca Logos

Logos é uma biblioteca de análise léxica para Rust. Ela consiste na definição de *tokens* através de *macros* e expressões regulares, tornando o código extremamente conciso.

O Código 2 ilustra a definição dos *tokens* de uma calculadora simples, onde cada *token* é definido através de uma expressão regular. A biblioteca automaticamente gera o analisador léxico, que pode ser usado para analisar uma string e retornar os *tokens* correspondentes.

CÓDIGO 2 – Análise léxica para uma calculadora usando a biblioteca Logos.

```
use logos::Logos;

#[derive(Logos)]
#[logos(skip r"[ \t\n]+")]
enum Token {
  #[token("+")]
  Plus,

  #[token("-")]
  Minus,

  #[token("*")]
  Multiply,

  #[token("/")]
  Divide,

  #[token("(")]
  LParen,

  #[token(")")]
  RParen,

  #[regex("[0-9]+", |lex| lex.slice().parse::<isize>().unwrap())]
  Integer(isize),
}

fn main() {
  let mut lexer = Token::lexer("1 + 2 * (3 - 4)");

  while let Some(token) = lexer.next() {
    println!("{:?}", token);
  }
}
```

TABELA 1 – Teste de desempenho da biblioteca Logos.

Teste	**Benchmark**
Identificadores	647 ns/iter (+/- 27) = 1204 MB/s
Palavras-chave, operadores e pontuações	2,054 ns/iter (+/- 78) = 1037 MB/s
Strings	553 ns/iter (+/- 34) = 1575 MB/s

FONTE: Adaptado de Hirsz (2018).

FONTE: Adaptado de Hirsz (2018).

Além da simplicidade na definição dos *tokens*, o analisador léxico gerado é extremamente rápido, como mostra o teste de desempenho do repositório oficial da biblioteca na Tabela 1.

Por fim, o uso da biblioteca Logos estará na implementação de toda a análise léxica, evitando que tempo seja gasto na análise manual de cada *token*. A motivação para a escolha da biblioteca se deve à sua simplicidade e maturidade no ecossistema Rust, assim minimizando o tempo de desenvolvimento e garantindo maior estabilidade.

2.2.3 Biblioteca Chumsky

Chumsky é uma biblioteca de análise sintática para Rust. Ela é baseada no conceito de *parser combinators*[2], e permite que a definição de *parsers* seja feita de forma declarativa. Seu escopo abrange tanto gramáticas livres de contexto quanto gramáticas sensíveis ao contexto.

Ao usar a biblioteca para construir um *parser*, nota-se a influência do paradigma funcional. Por mais que seja um paradigma mais incomum, seu uso na biblioteca torna o processo de construção do *parser* muito parecido com a construção de uma gramática formal, como demonstrado pelos comentários no Código 3.

CÓDIGO 3 – *Parser* para uma gramática de expressões aritméticas simples usando a biblioteca Chumsky.

```
use chumsky::prelude::*;

// Nodos da AST, cuja estrutura é representada por expressões recursivas.
enum Expr<'a> {
  Int(f64),
  Neg(Box<Expr<'a>>),
  Add(Box<Expr<'a>>, Box<Expr<'a>>),
  Sub(Box<Expr<'a>>, Box<Expr<'a>>),
  Mul(Box<Expr<'a>>, Box<Expr<'a>>),
  Div(Box<Expr<'a>>, Box<Expr<'a>>),
}

// Parser final composto por parsers menores (int, unary, product e sum), ou ↩
    seja, um parser combinator.
fn parser<'a>() -> impl Parser<'a, &'a str, Expr<'a>> {
  let op = |c| just(c).padded()

  // int -> regex([0-9]+)
  let int = text::int(10).map(|s: &str| Expr::Int(s.parse().unwrap()))

  // unary -> atom | '-' unary
  let unary = op('-')
    .repeated()
    .foldr(int, |_op, rhs| Expr::Neg(Box::new(rhs)));

  // product -> unary ( '*' unary | '/' unary )*
  let product = unary.foldl(
    choice((
      op('*').to(Expr::Mul as fn(_, _) -> _),
```

[2] Um *parser combinator* consiste na combinação de parsers mais simples para criar parsers mais complexos, assim como é de costume compor uma função maior de funções menores.

```
29        op('/').to(Expr::Div as fn(_, _) -> _),
30      ))
31      .then(unary)
32      .repeated(),
33      |lhs, (op, rhs)| op(Box::new(lhs), Box::new(rhs)),
34    );
35
36    // sum -> product ( '+' product | '-' product )*
37    let sum = product.foldl(
38      choice((
39        op('+').to(Expr::Add as fn(_, _) -> _),
40        op('-').to(Expr::Sub as fn(_, _) -> _),
41      ))
42      .then(product)
43      .repeated(),
44      |lhs, (op, rhs)| op(Box::new(lhs), Box::new(rhs)),
45    )
46
47    sum
48  }
49
50  fn main() {
51    // (+)
52    // |- (*)
53    // |  |- (-)
54    // |  |  \- 2
55    // |  \- 3
56    // \- 5
57    parser().parse("-2 * 3 + 5");
58  }
```

FONTE: Adaptado de Hirsz (2018).

De acordo com a classificação do teste de desempenho da biblioteca e seus competidores localizada na Tabela 2, Chumsky tem a capacidade de ser a biblioteca de análise sintática mais rápida para Rust.

TABELA 2 – Classificação do teste de desempenho da biblioteca Chumsky e competidores.

Classificação	**Biblioteca**	**Tempo de Execução**
1	chumsky (check-only)	140.77 µs
2	winnow	178.91 µs
3	chumsky	210.43 µs
4	sn	237.94 µs
5	serde_json	477.41 µs
6	nom	526.52 µs
7	pest	1.9706 ms
8	pom	13.730 ms

FONTE: Adaptado de Barretto (2021).

Por fim, o uso da biblioteca Chumsky estará na implementação da análise sintática, e será utilizada em conjunto com a biblioteca Logos na implementação do interpretador como um todo. Assim como foi o caso com Logos, a escolha de Chumsky se deve à sua maturidade.

3 METODOLOGIA

Este capítulo apresenta os procedimentos e métodos adotados para a realização do projeto. Aqui serão detalhados o tipo e a abordagem da pesquisa, o contexto no qual o estudo se encontra, a população e a amostra analisadas, e por fim as técnicas de coleta e análise de dados utilizadas.

3.1 TIPO E ABORDAGEM DE PESQUISA

Este projeto consiste de uma pesquisa aplicada e exploratória, visando compreender melhor o tema de linguagem de programação baseada em ECS, que ainda é pouco abordado, e assim prover um protótipo com aplicações práticas como base para pesquisas futuras.

O projeto busca investigar o *design* e a implementação da linguagem com uma abordagem focada em aspectos qualitativos, como usabilidade, modularidade e segurança, descartando qualquer aspecto quantitativo, como o desempenho. Essa abordagem será aplicada no estudo e análise de todas as etapas do projeto, desde a definição do *design* da linguagem até a implementação do protótipo de interpretador.

3.2 CONTEXTO

O projeto se encontra no contexto dado pela intersecção da área de desenvolvimento de linguagens de programação e do padrão arquitetural *Entity Component System* (ECS). O ECS guia o *design* e a implementação da linguagem de programação proposta de forma a facilitar o uso do padrão.

3.3 DELIMITAÇÃO DO ESTUDO

Com a finalidade de manter o foco do projeto em seus objetivos centrais, serão estabelecidas algumas delimitações. O escopo do *design* e da implementação do projeto estará estritamente concentrado na linguagem de programação e em sua biblioteca padrão (do inglês, *standard library*).

Dessa forma, o projeto não abordará aspectos relacionados ao ecosistema de desenvolvimento, como *syntax highlighting* e *Language Server Protocol* (LSP).

3.4 POPULAÇÃO E AMOSTRA

A população de referência para o estudo são linguagens de programação e bibliotecas que implementam ou dão suporte ao padrão ECS. A amostra escolhida inclui a linguagem Rust e as bibliotecas de ECS Flecs e Bevy, por serem extremamente relevantes no contexto de ECS e por serem inovadoras em suas abordagens.

3.5 TÉCNICAS DE COLETA DE DADOS

Para a coleta de dados, foram utilizadas as técnicas de pesquisa bibliográfica, documental e experimental, conforme descrito a seguir:

- Pesquisa bibliográfica: análise de livros, artigos e publicações acadêmicas sobre linguagens de programação, ECS e *design* de software;
- Pesquisa documental: análise de documentação, repositórios de código-fonte e exemplos de uso de linguagens e bibliotecas que implementam o padrão ECS;

- Pesquisa experimental: definição do *design* da linguagem de forma iterativa com base na análise qualitativa dos resultados.

3.6 TÉCNICAS DE ANÁLISE DE DADOS

A análise dos dados coletados será realizada de forma qualitativa, por meio das seguintes técnicas de análise:

- Análise documental e bibliográfica: análise e interpretação dos conceitos, padrões e práticas encontrados nas fontes pesquisadas.
- Análise comparativa: avaliação das características do protótipo em comparação com linguagens e bibliotecas selecionadas na amostra, destacando pontos fortes e limitações.

3.7 PROCEDIMENTOS METODOLÓGICOS

O procedimento metodológico adotado para o desenvolvimento do projeto será dividido nas quatro etapas a seguir, onde cada uma depende da conclusão da anterior:

- Definição do *design* da linguagem: nesta etapa, serão definidos os conceitos fundamentais da linguagem, como o que ela propõe e não propõe fazer, além de sua sintaxe, semântica e paradigmas de programação, tudo com um foco na abstração do padrão ECS.
- Avaliação do *design* da linguagem: nesta etapa, serão avaliados os conceitos definidos na etapa anterior, buscando analisar critérios de qualidade de software, como usabilidade, expressividade, acoplamento, entre outros. As avaliações serão feitas de forma qualitativa, comparando com as linguagens e bibliotecas selecionadas na amostra.
- Implementação do protótipo de interpretador: nesta etapa, as definições do *design* da linguagem serão implementadas em um protótipo de interpretador, que será desenvolvido em Rust. É importante ressaltar que será permitido que alguns itens menos importantes propostos pelo design não sejam implementados, a fim de tornar esta etapa viável dentro do tempo disponível para o projeto.
- Avaliação e validação do protótipo: nesta etapa, será avaliado todo o processo da implementação do protótipo, buscando analisar dificuldades e limitações encontradas. Além disso, será analisado o quanto o protótipo atende ao design da linguagem previamente definido.

REFERÊNCIAS

Agile Alliance. *The Agile Manifesto*. 2001. Disponível em: <https://www.agilealliance.org/agile101/the-agile-manifesto/>.

BARRETTO, J. *Chumsky*. 2021. Disponível em: <https://github.com/zesterer/chumsky>.

Bevy Foundation. *Bevy: A refreshingly simple data-driven game engine built in Rust*. 2020. Disponível em: <https://github.com/bevyengine/bevy>.

FREEMAN, E.; ROBSON, E.; BATES, B. *Head First Design Patterns: A Brain-Friendly Guide*. Sebastopol, California: O'Reilly Media, 2004.

GAMMA, E. et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. 1st. ed. Boston, Massachusetts: Addison-Wesley, 1994.

HIRSZ, M. *Logos*. 2018. Disponível em: <https://github.com/maciejhirsz/logos>.

IONOS. *Programming paradigms: What are the principles of programming?* 2020. Disponível em: <https://www.ionos.com/digitalguide/websites/web-development/programming-paradigms/>.

KLABNIK, S.; NICHOLS, C. *The Rust Programming Language*. 2nd. ed. San Francisco, California: No Starch Press, 2022.

LIYAN, A. *What is a Programming Paradigm?* 2023. Disponível em: <https://medium.com/@Ariobarxan/what-is-a-programming-paradigm-ec6c5879952b>.

MERRIAM-WEBSTER, I. *Merriam-Webster*. s.d. Disponível em: <https://www.merriam-webster.com/>.

MERTENS, S. *Entity Component System FAQ*. 2019. Disponível em: <https://github.com/SanderMertens/ecs-faq>.

MERTENS, S. *Flecs: Fast Entity Component System*. 2019. Disponível em: <https://github.com/SanderMertens/flecs>.

MERTENS, S. *ECS: From Tool to Paradigm*. 2021. Disponível em: <https://ajmmertens.medium.com/ecs-from-tool-to-paradigm-350587cdf216>.

MERTENS, S. *Building Games in ECS with Entity Relationships*. 2022. Disponível em: <https://ajmmertens.medium.com/building-games-in-ecs-with-entity-relationships-657275ba2c6c>.

MERTENS, S. *ECS Storage in Pictures*. 2024. Disponível em: <https://ajmmertens.medium.com/building-an-ecs-storage-in-pictures-642b8bfd6e04>.

MICROSOFT. *RxJS: Reactive Extensions for JavaScript*. 2009. Disponível em: <https://reactivex.io/>.

NYSTROM, R. *Crafting Interpreters*. [S.l.]: Genever Benning, 2021.

RICHARDS, M.; FORD, N. *Fundamentals of Software Architecture: An Engineering Approach*. 1st. ed. Sebastopol, California: O'Reilly Media, 2020.

WILLIS, C. A. *Two Published Flight Dynamics Models Rewritten in Rust and Structures as an ECS*. Dissertação (MS Thesis) — Air Force Institute of Technology, Wright-Patterson Air Force Base, Ohio, 2021.