UNIVERSIDADE DO ESTADO DO RIO GRANDE DO NORTE
FACULDADE DE CIÊNCIAS NATURAIS E EXATAS - FANAT
CIÊNCIA DA COMPUTAÇÃO

ERICA KATHLEN DE ANDRADE DOS SANTOS
IGOR DIAS ANISIO
JOÃO MARCELO NUNES DE SOUZA
JOÃO GUILHERME BEZERRA SANTOS
RAFAEL ALEXANDER DE SOUZA BRITO

DOCUMENTAÇÃO DO TRABALHO FINAL: TRANSPILADOR JAVASCRIPT PARA RUST

Mossoró/RN
2025

Sumário

1. Introdução

Este trabalho apresenta o desenvolvimento de um transpilador *source-to-source* capaz de converter código JavaScript (subconjunto ES6) para a linguagem Rust. O projeto surge da necessidade de unir a flexibilidade e popularidade do JavaScript com a segurança de memória e a performance de linguagens de sistema compiladas.

Embora o JavaScript seja a linguagem dominante na Web, sua execução fora do navegador exige ambientes interpretadores robustos, como o Node.js, o que cria dependências complexas e dificulta a distribuição de softwares leves. A justificativa central para este trabalho reside na capacidade de gerar **binários nativos e autocontidos** (*single binaries*). Ao transpilar para Rust, o código resultante pode ser compilado para qualquer sistema operacional sem exigir que o usuário final possua um ambiente JavaScript instalado.

Além da portabilidade, o projeto aborda a **segurança e robustez**. Através de um *Runtime Environment* customizado, o transpilador emula o comportamento dinâmico do JavaScript dentro das regras estritas de *Ownership* do Rust. Isso permite que desenvolvedores criem ferramentas de sistema e utilitários CLI (*Command Line Interface*) aproveitando a ergonomia de escrita do JavaScript, mas herdando a eficiência de recursos e a proteção contra erros de memória garantidas pelo compilador Rust.

2. Linguagem de Origem: JavaScript

O JavaScript, formalmente padronizado como ECMAScript (ECMA-262), é uma linguagem de programação de alto nível, interpretada e multiparadigma. Originalmente concebida em 1995 para adicionar interatividade a páginas web no lado do cliente, a linguagem evoluiu para se tornar uma tecnologia onipresente, executando em servidores (*Node.js*), dispositivos móveis e sistemas embarcados.

Para este projeto, o foco recai sobre a versão **ECMAScript 2015 (ES6)** e posteriores, que introduziram melhorias significativas de sintaxe, modularização e estruturação de escopo.

2.1. Paradigmas e Características Fundamentais

O JavaScript distingue-se por sua flexibilidade, permitindo ao desenvolvedor adotar diferentes estilos de programação dentro de uma mesma base de código. As características fundamentais que definem a linguagem e que são suportadas pelo subconjunto deste transpilador incluem:

2.1.1. Tipagem Dinâmica e Fraca

O sistema de tipos do JavaScript é dinâmico, o que significa que as variáveis não estão vinculadas a um tipo de dado específico em tempo de compilação. Uma variável declarada com *let* pode ser inicializada com um valor numérico e, subsequentemente, receber uma *string* ou um objeto. Além disso, a linguagem é fracamente tipada, realizando coerção implícita de valores durante operações (ex: converter automaticamente números para strings em concatenações), o que confere grande expressividade sintática, embora reduza a previsibilidade estática.

2.1.2. Funções de Primeira Classe (*First-Class Functions*)

No JavaScript, funções são tratadas como "cidadãos de primeira classe". Isso significa que elas são, essencialmente, objetos que podem ser:

- Atribuídos a variáveis;
- Passados como argumentos para outras funções (*callbacks*);
- Retornados como valores de outras funções. Essa característica é a base para o suporte ao paradigma funcional e para a implementação de *Closures* — a capacidade de uma função "lembrar" e acessar o escopo léxico onde foi criada, mesmo quando executada fora dele.

2.1.3. Orientação a Objetos Baseada em Protótipos

Diferente da orientação a objetos clássica baseada em Classes (como em Java ou C++), o JavaScript utiliza um modelo prototípico. Objetos são coleções dinâmicas de pares

chave-valor (estruturalmente semelhantes a dicionários ou *HashMaps*). Propriedades e métodos podem ser adicionados ou removidos de objetos em tempo de execução, oferecendo uma maleabilidade estrutural que não existe em linguagens estáticas tradicionais.

2.2. A Sintaxe Moderna (ES6+)

A escolha do padrão ES6 para a entrada do transpilador deve-se à introdução de recursos que tornam o código mais legível e seguro em comparação às versões anteriores:

- **Escopo de Bloco:** A introdução de *let* e *const* mitiga problemas clássicos de vazamento de escopo causados pelo antigo var.
- **Arrow Functions:** A sintaxe concisa *() => {}* facilita a escrita de expressões funcionais e lambdas.
- **Template Literals:** O uso de crases permite interpolação direta de expressões em strings, eliminando a complexidade de concatenações manuais.

3. Linguagem de Destino: Rust

A linguagem escolhida como alvo da transpilação é o Rust (versão Edition 2021). Rust é uma linguagem de programação de sistemas multiparadigma, desenvolvida originalmente pela Mozilla Research, que tem como pilares centrais a segurança de memória, o controle de concorrência e a alta performance. Ela foi projetada para preencher a lacuna entre linguagens de alto nível (que oferecem segurança mas têm custo de performance) e linguagens de baixo nível como C++ (que oferecem controle total mas são propensas a erros de memória).

3.1. O Modelo de Memória e Ownership

A característica mais distintiva do Rust é o seu modelo de gerenciamento de memória, que dispensa o uso de um Coletor de Lixo e também do gerenciamento manual (como *malloc/free*). Em vez disso, o Rust introduz o conceito de **Posse** rastreado em tempo de compilação.

O sistema baseia-se em três regras estritas verificadas pelo compilador:

1. Cada valor na memória possui uma variável que é sua "dona" (*owner*).
2. Só pode haver um dono por vez.
3. Quando a variável dona sai de escopo, o valor é imediatamente desalocado.

Para permitir o uso de dados sem transferir a posse, o Rust utiliza o conceito de **Empréstimo** (*Borrowing*). O mecanismo de *Borrow Checker* impõe que, para um dado

recurso, pode existir um número ilimitado de referências imutáveis (leitura), ou exatamente uma referência mutável (escrita), mas nunca ambas simultaneamente. Isso elimina, por definição, condições de corrida (*data races*) em tempo de compilação.

3.2. Sistema de Tipos e Tipos Algébricos de Dados

Rust possui um sistema de tipos estático e forte, garantindo que operações inválidas sejam detectadas antes da execução. No entanto, sua expressividade é ampliada pelo suporte a **Tipos Algébricos de Dados** (ADTs).

Os *Enums* em Rust são superconjuntos de enums tradicionais (encontrados em C ou Java). Eles permitem que cada variante do enumerador carregue dados de tipos e estruturas diferentes. Juntamente com o recurso de **Pattern Matching** (*match*), isso permite que o compilador force o programador a tratar explicitamente todas as possibilidades de estado de um dado, evitando erros comuns como *Null Pointer Exceptions* (o Rust não possui *null* nativo, utilizando o enum *Option<T>* para representar ausência de valor).

3.3. Ponteiros Inteligentes e Mutabilidade Interior

Embora o modelo de *Ownership* seja rígido, a biblioteca padrão do Rust oferece estruturas para flexibilizar essas regras de forma segura quando necessário, conhecidas como *Smart Pointers*:

- ***Box<T>*:** Permite alocar dados na *Heap* em vez da *Stack*, mantendo a semântica de posse única.
- ***Rc<T>* (Reference Counting):** Permite que um dado tenha múltiplos donos. O dado é mantido na memória enquanto houver pelo menos uma referência ativa a ele.
- ***RefCell<T>* (Interior Mutability):** Permite alterar dados mesmo quando existem referências imutáveis para ele. O *RefCell* move a verificação das regras de empréstimo do tempo de compilação para o tempo de execução, causando um *panic* (erro controlado) se as regras forem violadas, em vez de um erro de compilação.

3.4. Compilação e Infraestrutura

O compilador do Rust utiliza o framework LLVM (*Low Level Virtual Machine*) como backend. Isso significa que o código Rust passa por otimizações agressivas de nível de máquina, resultando em binários nativos que competem diretamente em performance com C e C++, sem a sobrecarga de uma máquina virtual ou interpretador acoplado.

4. Transpilador

4.1. Desafios de Implementação e Soluções Adotadas

A principal dificuldade no desenvolvimento de um transpilador *JavaScript-to-Rust* reside na incompatibilidade fundamental entre os paradigmas das duas linguagens. O JavaScript opera sob premissas de flexibilidade extrema, enquanto o Rust impõe rigorosas verificações de segurança. Esta seção detalha os obstáculos semânticos encontrados e as soluções arquiteturais desenvolvidas no *Runtime* para superá-los.

O Rust exige que o tamanho e o tipo de cada variável sejam conhecidos em tempo de compilação. O JavaScript permite que uma variável mude de tipo arbitrariamente durante a execução. Uma tradução direta de *let x* (JS) para uma variável primitiva Rust é impossível. Para contornar a rigidez estática, foi implementado um *Enum* abrangente que atua como um container polimórfico. Todas as variáveis no código transpilado são, na verdade, do mesmo tipo Rust (*JsValue*), mas carregam variantes de dados diferentes internamente.

O *Runtime* verifica a variante ativa (via *Pattern Matching*) apenas no momento da operação, emulando o comportamento dinâmico:

```rust
pub enum JsValue {
    Undefined,
    Null,
    Number(f64),
    String(String),
    Boolean(bool),
    Array(Vec<JsVar>),
    Object(HashMap<String, JsVar>),
    Function(JsFunc),
}
```

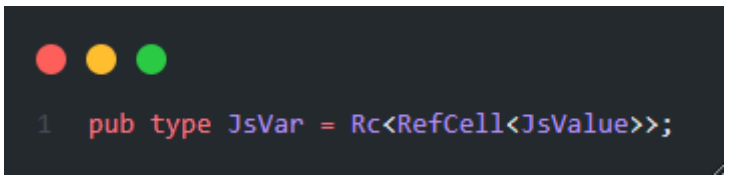
Além disso, em JavaScript, objetos e arrays são passados por referência e são mutáveis por padrão. Múltiplas partes do código podem apontar para o mesmo objeto e modificá-lo. O Rust aplica o conceito de *Ownership* (Posse Única) e proíbe "referências mutáveis compartilhadas" (aliasing + mutability) para evitar *data races*. Tentar traduzir *const b = a;* diretamente resultaria em um erro de compilação de "valor movido" (*move semantics*). A solução adotada foi envolver o enum *JsValue* em duas camadas de abstração da biblioteca padrão do Rust:

1. **Rc (Reference Counting):** Resolve o problema da posse compartilhada. Permite que variáveis como *a* e *b* apontem para o mesmo dado na *Heap*. O dado só é destruído

quando o contador de referências chega a zero (simulando um Garbage Collector local).

2. ***RefCell* (Interior Mutability):** Resolve o problema da mutabilidade. Permite alterar o valor contido dentro do *Rc* (que é imutável por natureza) movendo a checagem de segurança do tempo de compilação para o tempo de execução.

A definição final do tipo de variável no transpilador tornou-se:



Outro problema enfrentado é que o operador "+" em JavaScript é sobrecarregado e complexo. *10 + 20* é soma (*30*), mas *10 + "20"* é concatenação (*"1020"*). O Rust não permite somar tipos distintos (*i32* com *String*) nativamente e lançaria um erro de tipo.

Em vez de traduzir operadores aritméticos para seus equivalentes nativos no Rust, o transpilador os converte para chamadas de função do *Runtime*. A função *rt::add(a, b)*, por exemplo, implementa manualmente a lógica de coerção do padrão ECMAScript:

1. Recebe dois *JsVar* genéricos.
2. Extrai seus valores internos (borrow).
3. Verifica os tipos:
    - Se ambos forem números → Realiza soma aritmética.
    - Se um deles for string → Converte o outro para string e concatena.
    - Outros casos → Retorna *NaN* ou *undefined*.

Em relação aos condicionais, o JavaScript avalia qualquer valor em contextos booleanos. *if ("texto")* é verdadeiro; *if (0)* é falso. Já o Rust exige estritamente um tipo *bool (true/false)* em condicionais. Sendo assim, foi implementada uma função auxiliar *rt::is_truthy(&JsVar) -> bool* que é injetada em todas as condições (*if, while*). Ela examina o conteúdo da variável e aplica as regras do JS:

- *0, "", null, undefined, NaN* → Retornam *false* (Rust).
- Qualquer outro valor → Retorna *true* (Rust).

Desta forma, o comando JS *if (x)* é transpilado para *if rt::is_truthy(&x)*.

Por fim, funções em JavaScript capturam variáveis do escopo pai implicitamente. O Rust exige que a captura seja explícita e segura quanto ao tempo de vida da memória. Se uma função retornar outra função que usa uma variável local, essa variável seria desalocada da *Stack* ao fim da função pai, causando erro.

Para resolver isso, o transpilador gera *closures* Rust utilizando a palavra-chave *move*. Como todas as variáveis são *Rc* (ponteiros contados), o comando *move* move apenas o ponteiro (clone barato), e não o dado profundo. Isso garante que, enquanto a função filha existir, o dado referenciado por ela permanecerá vivo na memória, replicando com fidelidade o comportamento de escopo léxico do JavaScript.

4.2 Análise Léxica e Tokens Suportados

A primeira etapa do processo de transpilação é a Análise Léxica, responsável por converter a sequência bruta de caracteres do código fonte em uma sequência de tokens significativos.

Neste projeto, a tokenização e a construção da Árvore de Sintaxe Abstrata (AST) são delegadas à biblioteca *@babel/parser*. Esta decisão de projeto garante que a interpretação da sintaxe siga rigorosamente a especificação ECMAScript padrão da indústria. O transpilador, atuando como um *Visitor* sobre a AST, reconhece e processa um subconjunto específico de tokens, categorizados abaixo.

4.2.1. Palavras Reservadas (Keywords)

O transpilador identifica palavras-chave que definem a estrutura e o fluxo do programa. Elas são mapeadas diretamente para estruturas de controle equivalentes no *runtime* Rust ou para blocos de lógica nativa.

Categoria	**Tokens Suportados**	**Descrição no Contexto do Transpilador**
Declaração	*let, const*	Declaração de variáveis com escopo de bloco. O antigo *var* é intencionalmente não suportado para evitar *hoisting*.
Funções	*function, return*	Definição de sub-rotinas e retorno de valores encapsulados em *JsValue*.

Condicional	*if, else*	Estruturas de decisão baseadas na avaliação de "Verdadeiro/Falso" (*is_truthy*).
Repetição	*while, for*	Loops de execução. O loop for é desaçucarado em um *while* equivalente.
Exceções	*try, catch, throw*	Blocos de tratamento de erro. O *throw* é convertido para um *panic!* controlado no Rust.
Outros	*new*	Instanciação de objetos (tratada internamente como chamada de função construtora).

4.2.2. Literais e Tipos de Dados

Os literais representam valores fixos inseridos diretamente no código fonte. O transpilador reconhece os seguintes formatos e os converte para o construtor apropriado no Enum *JsValue*:

- **Literais Numéricos (*NumericLiteral*):**
  - Inteiros: *42, 100, 0*.
  - Ponto Flutuante: *3.14, 0.5, .99*.
  - *Tradução:* Todos são normalizados para o tipo *f64* (float de 64 bits) no Rust.
- **Literais de Texto (*StringLiteral*):**
  - Aspas duplas: *"Exemplo"*.
  - Aspas simples: *'Exemplo'*.
  - *Tradução:* Instanciam variantes *JsValue::String*.
- **Literais de Modelo (*TemplateLiteral*):**
  - Sintaxe: *`Valor: ${x}`*.
  - *Tradução:* Reconhecidos como strings interpoladas, convertidos para a macro *format!* do Rust, permitindo a inserção dinâmica de variáveis.
- **Literais Booleanos (*BooleanLiteral*):**
  - Tokens: *true, false*.
- **Literais Estruturais:**
  - Arrays: *[1, 2, "a"]* (Cria um *Vec<JsVar>*).

    - Objetos: *{ chave: "valor" }* (Cria um *HashMap<String, JsVar>*).
- **Nulos e Indefinidos:**
    - *null*: Tratado explicitamente.
    - *undefined*: Gerado implicitamente na ausência de retorno ou valor.

4.2.3. Símbolos e Operadores

O analisador léxico suporta os operadores padrão da aritmética e lógica do JavaScript. É importante notar que a semântica destes operadores é reimplementada no *runtime* para suportar a coerção de tipos.

- **Aritméticos:** + (soma/concatenação), -, *, /.
- **Atribuição:** =, +=, -=.
- **Incremento/Decremento:** ++, -- (Unários).
- **Comparação:**
    - Igualdade: ==, === (Nesta implementação, a igualdade estrita é tratada como igualdade de valor).
    - Desigualdade: *!=, !==*.
    - Relacionais: >, <, >=, <=.
- **Lógicos:** *&&* (E lógico), || (OU lógico), *!* (Negação).
- **Pontuação:** *{ }* (Escopo), *( )* (Agrupamento/Chamada), *[ ]* (Acesso a membro), , (Separador), *;* (Terminador), => (Arrow Functions), . (Acesso a propriedade).

4.2.4. Identificadores Intrínsecos (Built-ins)

Embora tecnicamente sejam identificadores e não palavras reservadas na gramática do JS, o transpilador reconhece tokens específicos que acionam funcionalidades do sistema operacional ou da biblioteca padrão:

- ***console***: Objeto global para I/O (*.log, .error*).
- ***prompt***: Função para captura de entrada do usuário (*Stdin*).
- ***Promise***: Construtor para simulação de assincronismo.
- ***setTimeout***: Função de temporização.

Qualquer token que não se enquadre nestas categorias ou que represente uma funcionalidade não implementada (como *switch* ou *class*) é ignorado ou resulta em um valor *undefined* seguro no código de destino.

4.3 Gramática e Reconhecimento Sintático

A etapa de análise sintática define as regras de composição dos tokens para formar estruturas válidas na linguagem. O transpilador adota uma abordagem de **Descida Recursiva** (*Recursive Descent*), percorrendo a Árvore de Sintaxe Abstrata (AST) da raiz até as folhas.

A gramática implementada é um subconjunto **Livre de Contexto** (*Context-Free*) da especificação ECMAScript. Abaixo, apresenta-se a formalização desta gramática utilizando a notação EBNF (*Extended Backus-Naur Form*) simplificada para fins de legibilidade.

4.3.1. Estrutura Global e Declarações

Um programa é definido como uma sequência de declarações (*Statements*). O parser distingue claramente entre declarações (que executam ações) e expressões (que avaliam valores).

```
Program         ::= Statement*

Statement       ::= VariableDecl
                  | FunctionDecl
                  | IfStatement
                  | IterationStmt
                  | ReturnStmt
                  | BlockStmt
                  | TryStmt
                  | ExpressionStmt

BlockStmt       ::= "{" Statement* "}"
```

4.3.2. Declarações de Variáveis e Funções

O reconhecimento de variáveis foca estritamente no escopo de bloco (*let, const*), ignorando o içamento (*hoisting*) do antigo *var*. Funções são tratadas como definições de alto nível ou expressões anônimas.

```
VariableDecl    ::= ("let" | "const") Identifier ["=" Expression] ";"

FunctionDecl    ::= "function" Identifier "(" ParamList ")" BlockStmt

ParamList       ::= [Identifier {"," Identifier}]
```

4.3.3. Estruturas de Controle de Fluxo

As estruturas de controle seguem o padrão imperativo clássico. Note que o loop *for* é semanticamente tratado como uma especialização de um *while* com inicialização e incremento prévios.

```
IfStatement     ::= "if" "(" Expression ")" Statement ["else" Statement]

IterationStmt   ::= WhileLoop | ForLoop

WhileLoop       ::= "while" "(" Expression ")" Statement

ForLoop         ::= "for" "(" [VarDecl | ExprStmt] ";" [Expression] ";" [Expression] ")" Statement

ReturnStmt      ::= "return" [Expression] ";"
```

4.3.4. Expressões e Precedência de Operadores

A gramática de expressões é estruturada hierarquicamente para garantir a precedência correta dos operadores (ex: multiplicação ocorre antes da adição). A descida recursiva avalia da menor precedência (atribuição) para a maior (fator primário).

```
Expression      ::= AssignmentExpr

(* Precedência 1: Atribuição (Right-associative) *)
AssignmentExpr  ::= LogicalOrExpr [("=" | "+=" | "-=") AssignmentExpr]

(* Precedência 2: Lógica Booleana *)
LogicalOrExpr   ::= LogicalAndExpr {"||" LogicalAndExpr}
LogicalAndExpr  ::= EqualityExpr   {"&&" EqualityExpr}

(* Precedência 3: Comparação *)
EqualityExpr    ::= RelationalExpr {("==" | "!=" | "===" | "!==") RelationalExpr}
RelationalExpr  ::= AdditiveExpr   {("<" | ">" | "<=" | ">=") AdditiveExpr}

(* Precedência 4: Aritmética *)
AdditiveExpr    ::= MultiplicativeExpr {("+" | "-") MultiplicativeExpr}
MultiplicativeExpr ::= UnaryExpr {("*" | "/") UnaryExpr}

(* Precedência 5: Unários e Prefixos *)
UnaryExpr       ::= ("!" | "-" | "typeof") UnaryExpr
                  | UpdateExpr

UpdateExpr      ::= MemberExpr ["++" | "--"]
```

```
(* Precedência 6: Acesso a Membros e Chamadas *)
MemberExpr      ::= PrimaryExpr {
                      ("[" Expression "]") |
                      ("." Identifier) |
                      ("(" ArgList ")")
                    }

(* Precedência 7: Termos Primários (Folhas da Árvore) *)
PrimaryExpr     ::= NumericLiteral
                  | StringLiteral
                  | BooleanLiteral
                  | TemplateLiteral
                  | Identifier
                  | ArrayLiteral
                  | ObjectLiteral
                  | ArrowFunction
                  | "(" Expression ")"

(* Estruturas Literais Complexas *)
ArrayLiteral    ::= "[" [Expression {"," Expression}] "]"
ObjectLiteral   ::= "{" [PropAssignment {"," PropAssignment}] "}"
ArrowFunction   ::= "(" ParamList ")" "=>" (BlockStmt | Expression)
```

4.3.5. Observações sobre a Implementação Gramatical

1. **Associatividade:** A gramática reflete a associatividade dos operadores. Atribuições são associativas à direita ($a = b = c$), enquanto operações aritméticas são associativas à esquerda ($a + b + c$).
2. **Desaçucaramento Sintático (*Syntactic Desugaring*):** Algumas construções gramaticais do JavaScript são convertidas para formas mais simples no Rust durante a visitação. Por exemplo, *Template Literals* (strings com crase) não possuem um tipo primitivo correspondente direto, sendo convertidas gramaticalmente para chamadas de macro *format!*.
3. **Tratamento de Ponto e Vírgula:** Embora o JavaScript permita a inserção automática de ponto e vírgula (ASI), este transpilador assume uma estrutura onde as declarações são delimitadas explicitamente ou pelo fim do nó na AST, gerando o código Rust correspondente com terminadores explícitos ; para garantir a compilação.

5. Conclusão

O desenvolvimento deste transpilador *source-to-source* demonstrou a viabilidade técnica e prática de converter uma linguagem de tipagem dinâmica e alto nível, como o JavaScript (ES6), para uma linguagem de sistemas estática e segura, como o Rust.

A principal contribuição deste trabalho não reside apenas na tradução sintática dos comandos, mas na arquitetura do *Runtime Environment* desenvolvido. A implementação da estrutura *JsValue*, combinada com o uso estratégico de ponteiros inteligentes (*Rc* e *RefCell*), provou ser uma solução eficaz para superar o "abismo de impedância" entre os paradigmas. Foi possível emular o comportamento de *Garbage Collection* e tipagem fraca do JavaScript dentro das regras estritas de *Ownership* e *Borrowing* do Rust, sem sacrificar a segurança de memória que torna a linguagem de destino tão valiosa.

O resultado final é uma ferramenta que confere aos scripts JavaScript características desejáveis de engenharia de software moderna: a geração de binários nativos autocontidos, portabilidade entre sistemas operacionais sem dependências externas (como o Node.js) e a robustez de um sistema de tipos compilado. Do ponto de vista acadêmico, o projeto serviu como uma aplicação prática dos conceitos de Análise Léxica, Sintática e Semântica, validando o uso de Árvores de Sintaxe Abstrata (AST) e do padrão *Visitor* na construção de compiladores.

Todo o código fonte desenvolvido neste projeto, incluindo o transpilador, a biblioteca de *runtime*, alguns exemplos de uso e um guia de instalação, estão disponíveis publicamente no repositório oficial:

Repositório: https://github.com/JMarcelooo/Compiladores