UM NOVO HORIZONTE PARA APLICAÇÕES WEB:
Desenvolvimento de um compilador C++ para WASM

Felipe N. Ribeiro[1], Judson S. Santiago[2]

RESUMO

Devido às limitações do JavaScript, a Web precisou desenvolver um novo tipo de código que suportasse executar sistemas que exigem alto desempenho. Nesse contexto, surgiu o WebAssembly, ou WASM, um novo formato de código binário para ser executado em ambientes Web. Para usar esse formato, é possível criar um código escrito na linguagem C++ e compilar para WASM usando o compilador oficial Emscripten. Porém, do ponto de vista de um programador que deseja identificar as conversões que foram feitas do programa fonte, em C++, para o programa alvo, em WASM, o código gerado tende a ser muito complexo e pouco claro. Com isso, foi desenvolvido um novo compilador de C++ para WASM, com foco em uma compilação simples e mais direta, útil para quem precisa entender as equivalências de um programa para o outro. Além disso, o compilador foi construído para gerar código que executasse mais rápido que o código equivalente em JavaScript puro, de tal modo, que comprovasse a usabilidade do WASM nos ambientes Web.

Palavras-chave: Web, JavaScript, WebAssembly, Compiladores.

1 INTRODUÇÃO

A Web atual, é construída com apenas três elemento: HTML, CSS e JavaScript. Os dois primeiros com as responsabilidades de estruturação e estilização da página, respectivamente, e o último com a execução de scripts simples. A Web se consolido como o principal ambiente de troca de informações e execução de aplicações e sistemas de todos os tipos, desde redes sociais, passando por jogos e chegando até em sistemas empresariais. Com isso, o JavaScript, que gerencia as regras de lógica de tudo isso, precisou avançar bastante para acompanhar todo esse desenvolvimento.

Hoje o JavaScript (JS) é uma das tecnologias da computação mais importantes do mundo. Segundo a pesquisa anual "*Developer Survey*" (2023) realizada pelo site *Stack Overflow*, o JS está a dez anos consecutivos como linguagem de programação mais usada. Isso nos mostra a dimensão e importância que a linguagem possui. Mas mesmo com toda a sua disseminação, será que ela está conseguindo acompanhar todas as demandas da Web?

Quando falamos de sistemas que exigem alto recurso computacional, como jogos *Triplo-A* ou softwares do tipo CAD, estamos familiarizados em vê-los sendo executados em ambientes nativos com aplicações desktops. Não vemos esses programas executando em ambientes Web porque as tecnologias não suportam demandas com alto nível de desempenho. Em outras palavras, o JS, que é a linguagem que executa os programas na Web, não tem capacidade computacional para executar tarefas de forma tão eficiente quanto os ambientes nativos, que usam linguagens como C e C++ e conseguem aproveitar de forma mais eficiente os recursos da máquina.

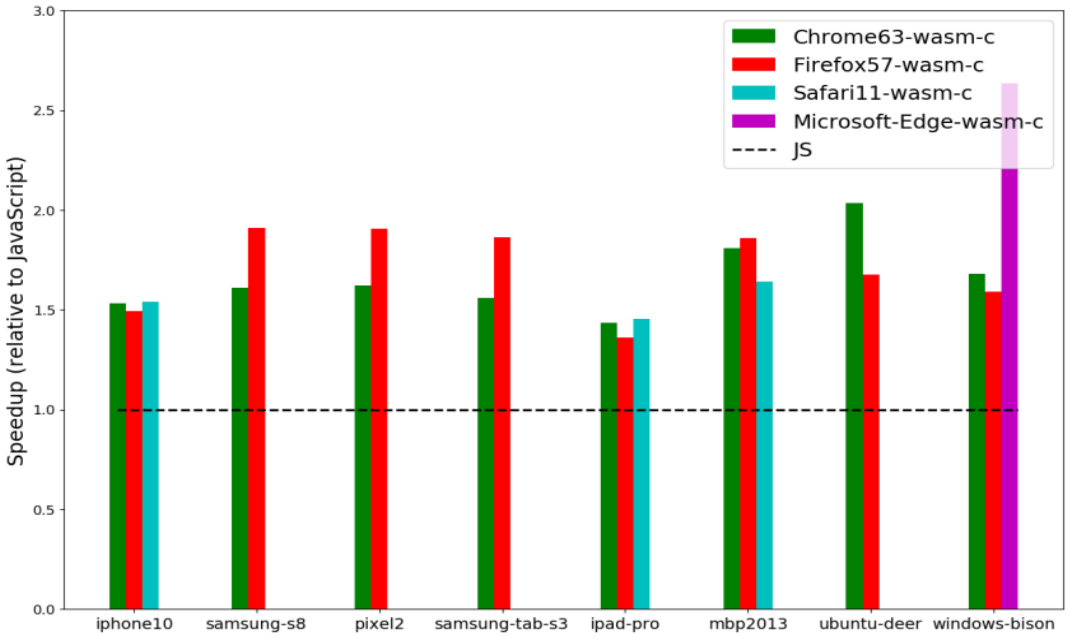
Nesse cenário, surge o ASM.js, subconjunto do JS, desenvolvido pela Mozilla em parceria com empresas de jogos, como Epic Games e Unity, segundo Working Draft (2014). Apesar do êxito de conseguir proporcionar a execução de jogos pesados usando essa tecnologia, logo se percebeu que ainda existiam problemas estruturas como não ter threads, bloqueios e outras funcionalidades de código nativos que travavam o avanço do ASM.js como solução definitiva para o problema de desempenho do JS na Web.

Em 17 de junho de 2015, Brendan Eich, o criador do JS, anunciava um projeto para desenvolvimento de um novo formato de código binário para execução de código na Web, chamado WebAssembly, abreviado para WASM. A equipe de trabalho contava com desenvolvedores do Google, Microsoft, Mozilla e Apple fundando a W3C WebAssembly Community Group, segundo WebAssembly Community Group (2023), que existe até hoje e segue com seus trabalhos. Em Elliott (2015), Brendan Eich afirma: "[...] a evolução contínua do ASM.js é WASM".

"A proposta do WASM é ser seguro, rápido, portátil e de baixo nível" (Hass, 2017). Ele foi construído com a função de executar as funcionalidades mais pesadas de um sistema, sendo acoplado ao JS e não o substituindo por completo. Além disso, ele visa auxiliar o ecossistema da Web com a execução de códigos externos, escritos em linguagens como C e C++, que podem ser compiladas para WASM e executado. Tudo isso só foi possível pela colaboração sem precedentes entre todos os fornecedores de navegadores e o grupo da comunidade online, segundo Hass.

Segundo o trabalho de Herrera, o WASM apresenta bons resultados de tempo de execução em comparação ao JS. Em um dos seus conjuntos de testes, são comparados o tempo de execução de algoritmos escritos em JS e em WASM, sendo executados em diferentes navegadores e dispositivos.

Figura 1 - Desempenho do WASM em relação ao JS em diversas plataformas.



Fonte: Herrera (2018, p. 15)

Como podemos ver na Figura 1, o WASM apresenta ganho de desempenho em relação ao JS em todos os testes. Nos indicando que o WASM pode ser uma boa esperança para a solução do problema de desempenho em sistemas pesados em ambientes Web.

1.1 Problema

Para compilação de código em WASM, existe a ferramenta Emscripten (Zakai, 2011), que compila código C/C++ para WASM, permitindo a execução na Web. A instalação da

ferramenta é simples e fácil, podendo ser usada para os mais diversos fins. O site oficial, nos evidencia isso com a passagem:

> Praticamente qualquer base de código C ou C++ portátil pode ser compilada em WebAssembly usando Emscripten, desde jogos de alto desempenho que precisam renderizar gráficos, reproduzir sons e carregar e processar arquivos, até estruturas de aplicativos como Qt. O Emscripten já foi usado para converter uma longa lista de bases de código do mundo real para WebAssembly, incluindo bases de código comerciais como o Unreal Engine 4 e o mecanismo Unity. (About emscripten, 2015)

Por padrão, o Emscripten compila o arquivo .c/.cpp gerando três arquivos, dentre eles há um ".wasm", um ".js" e um ".html". O ".wasm" conterá as funções C/C++ que foram compiladas e que será executado através do ".js", que por sua vez é chamado dentro do ".html". Para os mais curiosos que forem abrir o arquivo ".wasm" vão se deparar com um código binário não legível por humanos. Apesar disso, o WASM possui um formato textual que nos permite converter o arquivo binário, com extensão ".wasm", em um formato legível, que tem a extensão ".wat". Logo, podemos converter o ".wasm" em ".wat" e analisarmos o código compilado.

Devido a robustez do compilador, no processo de compilação são geradas várias variáveis e rotinas extras, que não estão ligadas ao código original, mas que lidam com alocação de memória, vazamentos de memória e uma série de outros problemas. Logo, mesmo com o conhecimento da sintaxe do ".wat", fica difícil o entendimento da conversão dos comandos da linguagem original para o comando binário WASM. Por exemplo, um simples código que exibi uma mensagem no console, evidenciado na Figura 2, será compilado em um arquivo ".wat" com 68.451 linhas. Achar o código equivalente não é trivial.

Figura 2 - Código C++ para exibir "*Hello World!"* no console.

```
#include <iostream>
int main() {
    std::cout << "Hello World!";
}
```

Fonte: Autoria própria.

Do ponto de vista de um desenvolvedor Web que deseja apenas executar uma função C++ isolada em um código JS, entender a compilação é algo pouco relevante. Mas para alguém que precisa escrever módulos WASM para otimização do desempenho de uma biblioteca JavaScript, é importante que ele entenda qual é a equivalência de cada comando que está sendo realizado através da compilação. Nesse cenário, o Emscripten não é a melhor solução para a tarefa.

1.2 Hipótese

Um compilador que trouxesse o código compilado de forma crua, somente com o equivalente dos comandos da linguagem original para o comando binário seria bem mais proveitoso para um desenvolvedor que precisa entender o funcionamento e a sintaxe do WASM.

1.3 Objetivo

Com o problema proposto e com a hipótese levantada, o objetivo principal desse trabalho é construir um compilador simples de código C++ para WASM, onde esse código pode ser executado em um ambiente Web e possui somente a tradução fidedigna dos comandos originais sem a adição de código extra. Além disso, o objetivo secundário é que o código gerado seja mais eficiente que um código JS normal, comprovando a eficiência e justificando o uso do WASM em aplicações reais.

2 DESENVOLVIMENTO

O coração desse trabalho foi a criação de um compilador simples que convertesse programas escritos na linguagem de programação C++ em código binário WASM. Antes de entrarmos propriamente no compilador desenvolvido, vamos entender minimamente o funcionamento de um compilador e do WASM. Em seguida, apresentamos o método científico escolhido para orientação desse trabalho. E por fim, trazemos os resultados encontrados e as discussões levantadas a partir deles.

2.1 Fundamentação Teórica

Dividimos a Fundamentação Teórica em duas partes: primeiro, abordaremos sobre compiladores de forma geral e todas as suas características; e em seguida, sobre a sintaxe do WASM e seu funcionamento com o JS.

2.1.1 Compiladores

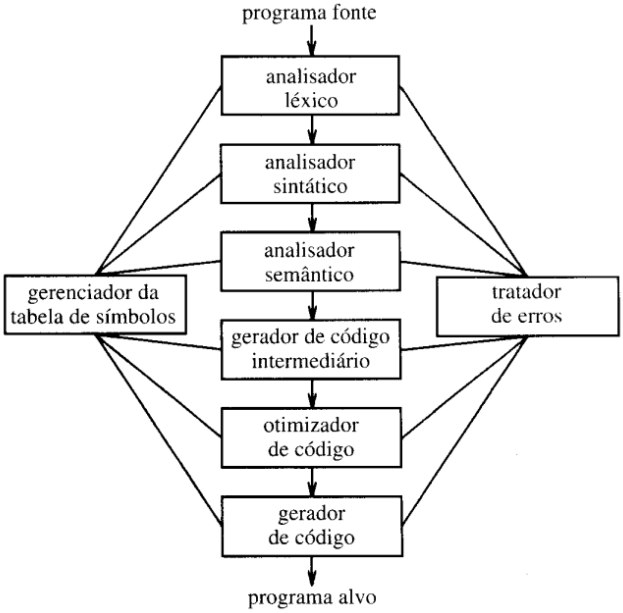
Para Cooper (2014), o compilador é um programa que traduz software escrito em uma linguagem, linguagem-fonte, para outra, linguagem-alvo. Para fazer a tradução, o programa deve entender a forma e o conteúdo da linguagem de entrada e de saída. Além disso, é necessário um esquema de mapeamento de conteúdo da linguagem-fonte para a linguagem-alvo.

O uso de compiladores na computação se dá de forma quase que onipresente, pois todo sistema/programa computacional contemporâneo usa um compilador diretamente ou indiretamente. Diante de responsabilidades tão altas, o desenvolvimento de tais componentes é um trabalho árduo e complexo. Cooper traz um comentário que colabora com nossa afirmação:

> Um bom compilador contém um microcosmo da ciência da computação. Faz uso prático de algoritmos gulosos (alocação de registradores), técnicas de busca heurística (agendamento de lista), algoritmos de grafos (eliminação de código morto), programação dinâmica (seleção de instruções), autômatos finitos e autômatos de pilha (análises léxica e sintática) e algoritmos de ponto fixo (análise de fluxo de dados). Lida com problemas, como alocação dinâmica, sincronização, nomeação, localidade, gerenciamento da hierarquia de memória e escalonamento de pipeline. Poucos sistemas de software reúnem tantos componentes complexos e diversificados. Trabalhar dentro de um compilador fornece experiência em engenharia de software, difícil de se obter com sistemas menores, menos complicados. (Cooper, 2014, p. 3)

Um compilador é composto por vários componentes com responsabilidades únicas, onde a entrada passa em um processo de pipeline, com fases bem definidas e ordenadas. Em Aho (2008), ele nomeia e estrutura os componentes como mostra a Figura 3. Na prática, alguns componentes podem ser construídos juntos mantendo as suas respectivas responsabilidades. A seguir, vamos detalhar o funcionamento dos principais componentes.

Figura 3 - Componentes de um compilador.



Fonte: Aho (2008, p. 5)

O analisador léxico, também chamado de scanner, é o primeiro componente do fluxo. Ele vai ser o único componente do compilador que interage com cada caractere do programa de entrada. Para Cooper (2014), o analisador léxico de um compilador lê a cadeia de caracteres e produz um fluxo de saída que contém palavras, cada uma rotulada com sua categoria sintática. Para realizar a agregação e classificação, ele aplica um conjunto de regras que descrevem a estrutura léxica da linguagem de programação de entrada.

O próximo passo é o analisador sintático, também conhecido como *parser*, que vai receber como entrada o programa transformado pelo *scanner*. Sua responsabilidade principal, segundo Cooper (2014), é validar se as palavras estão respeitando a sintaxe da linguagem fonte. Se o *parser* determina que elas representam um programa válido, ele constrói uma árvore sintática que organiza as palavras em uma ordem lógica. Caso contrário, ele dispara um problema com diagnóstico apropriada ao usuário. A sintaxe da linguagem fonte é expressa por uma gramática livre de contexto.

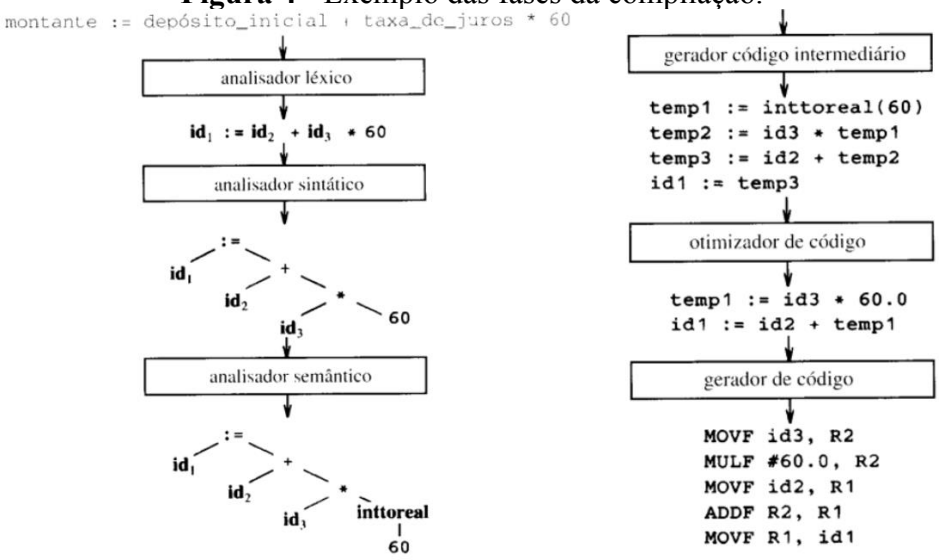
Uma gramática livre de contexto é um conjunto de regras que descrevem como formar sentenças. A notação tradicional usada por cientistas de computação para representar uma gramática livre de contexto é a Backus-Naur Form (BNF), como Cooper nos traz. Por decisão interna, adotamos a Extended BNF (EBNF) para descrever nossa gramática livre de contexto que será exposta no Método. A EBNF é uma extensão da BNF e é definida na ISO/IEC 14977:1996, em International organization for standardization (2023).

Voltando aos componentes do compilador, temos agora o analisador semântico, ou analisador sensível ao contexto. Mesmo que o programa passe com sucesso pelo *scanner* e pelo *parser*, ele ainda pode conter erros que impediriam a sua compilação. Para detectá-los, Cooper

(2014) fala que o compilador precisa olhar as posições dos itens na árvore sintática e analisar se eles fazem sentindo no ponto de vista de tipos e de concordância. Esse tipo de análise pode ser realizado junto da análise sintática ou em um passo posterior que percorra a árvore de derivação produzida pelo *parser*.

Após todas essas fases, finalizamos a análise da linguagem-fonte e vamos começar a traduzir para a linguagem-alvo. Para Aho (2008), a sequência de passos é a seguinte: primeiro, o gerador de código intermediário vai gerar uma representação intermediária explicita, que seja fácil de produzir e fácil de traduzir para a linguagem-alvo; em seguida, no otimizador de código, aplicamos otimizações na tentativa de deixar o código intermediário mais rápido em tempo de execução; e por fim, no gerador de código, que é a fase final do compilador, é gerado a linguagem-alvo, que normalmente é um código de máquina realocável ou código de montagem.

Figura 4 - Exemplo das fases da compilação.



Fonte: Aho (2008, p. 6)

Para ilustrar todos esses componentes em funcionamento, temos a Figura 4 que recebe como entrada um comando "montante := depósito_inicial + taxa_de_juros * 60" e passa fase por fase transformando o código até chegar na linguagem-alvo. Assim, podemos visualizar todos os processos que ocorrem ao decorrer de uma compilação convencional.

2.1.2 WebAssembly

Nosso objetivo nesta seção é elucidar o funcionamento do WebAssembly, tendo em vista todo o conteúdo já abordado na introdução desse trabalho, explicando como o WASM surgiu e qual é o seu propósito. Assim, inicialmente vamos entender como se estrutura o código e as formas de representá-lo, em seguida analisar a tipagem, a máquina de pilha e as principais instruções, e por fim, entender como funciona a integração do WASM com o JS. Todas as informações expostas a seguir foram retiradas da documentação oficial do WASM disponibilizada em Webassembly (2023).

O WASM tem duas representações: o formato binário, com a extensão de arquivo ".wasm", que é o código que a máquina executa; e o formato textual, com a extensão de arquivo ".wat", que é o código intermediário que pode ser lido e editado por humanos. A conversão de um formato para o outro pode ser feita através da ferramenta Wabt disponibilizada no repositório do GitHub do WebAssembly, em Webassembly/wabt (2023).

A unidade básica do WASM é um módulo, representado como uma grande expressão S. Esse tipo de representação é semelhante a estrutura de árvore, com a particularidade de ser bem plana. No formato textual, o nó da árvore é rodeado de parênteses, com o primeiro rótulo indicando o tipo de nó e em seguida os atributos do nó ou os nós filhos. Podemos ver um exemplo na Figura 5.

Figura 5 - Código WASM em formato textual.

```
(module
  (func (result i32)
    i32.const 1
    i32.const 2
    i32.add))
```

Fonte: Autoria própria (2023)

Todos os nós que representam parâmetros, constantes e variáveis possuem um tipo declarado explicitamente. Temos os tipos numéricos, tipos de referência e tipos de vetor. Nos tipos numéricos, temos: i32, para inteiros de 32 bits; i64, para inteiros de 64 bits; f32, para flutuantes de 32 bits; e f64, para flutuantes de 64 bits.

As instruções de operações numéricas são formadas pelo tipo numérico mais um ponto final mais a *label* da operação, como por exemplo: "i32.add" que representa a operação de soma de dois elementos. Para acesso de parâmetros e variáveis locais temos os comandos do tipo "local". A lista com todas as instruções pode ser encontrada na documentação da semântica do WASM, em Execution (2023).

O fluxo de execução do WASM é estruturado como uma máquina de pilha com cada instrução manipulando a cabeça da pilha, seja removendo ou adicionando valores. Por exemplo, voltando ao exemplo da Figura 5, o WASM vai adicionar os valores constantes 1 e 2 na pilha com as instruções do tipo "i32.const", após isso, a instrução "i32.add" vai remover os dois últimos valores e devolver para a pilha a soma dos valores. Ainda, como a função tem a instrução "result i32", indica que ela retorna um valor do tipo i32, logo, no final da execução da função, ela espera que a pilha tenha armazenada um valor i32 para ser retornado.

Hoje, segundo Webassembly (2023), códigos WASM podem ser executados nos cinco principais navegadores Web: Chrome, Edge, Firefox, Opera e Safari. Além dos navegadores, as ferramentas *runtime* de JavaScript Node.js e Deno também já possuem suporte.

2.2 Método

Começamos o desenvolvimento do nosso compilador tendo como base o compilador desenvolvido em C++ na disciplina de Compiladores, em Judsonss/compiladores (2023), lecionada por um dos autores desse trabalho, professor Judson Santiago. O compilador é usado como demonstração prática dos conteúdos abordados nas aulas. Ele é um compilador simples que contém as quatro primeiras fases de um compilador padrão, indo do analisador léxico ao gerador de código intermediário. Para a entrada, o compilador foi desenvolvido para receber o código com a sintaxe da linguagem C++ e retornar, como saída, uma representação intermediaria de código de três endereços. A gramática livre de contexto do compilador escrita em EBNF está representada na Figura 7.

Figura 7 - Gramática livre de contexto do compilador base.

```
program = "int", "main()",block ;
block   = "{", {decl}, {stmt}, "}" ;
decl    = type, id, [index], ";" ;

index   = "[", integer, "]" ;
```

```
rel    = ari, {comp}
;
comp   = "<", ari
       | "<=", ari
       | ">", ari
       | ">=", ari ;
```

```
type     = "int"
         | "float"
         | "bool"

zero     = "0" ;
```

```
stmt    = local, "=", bool, ";"
        | "if", "(", bool, ")", stmt
        | "while", "(", bool, ")",
stmt
        | "do", stmt, "while", "(",
bool, ")", ";"
        | block ;

local   = local, "[", bool, "]"
        | id ;

bool    = join, {lor} ;
lor     = "||", join ;

join    = equality, {land}
land    = "&&", equality ;

equality = rel, {eqdif} ;
eqdif    = "==", rel
         | "!=", rel ;
```

```
ari     = term {oper}
;
oper    = "+", term
        | "-", term ;

term    = unary
{calc} ;
calc    = "*", unary
        | "/", unary
;

unary   = "!", unary
        | "-", unary
        | factor ;

fator   = "(", bool,
")"
        | local
        | integer
        | floating
        | "true"
        | "false" ;
```

```
digit    = "0" | "1" | "2" | "3" |
"4"
         | "5" | "6" | "7" | "8" |
"9" ;
integer  = zero
         | ["-"], digit-zero, {digit}
;
floating = integer, ".", {digit} ;

letter   = "A" | "B" | "C" | "D" |
"E" | "F" | "G"
         | "H" | "I" | "J" | "K" |
"L" | "M" | "N"
         | "O" | "P" | "Q" | "R" |
"S" | "T" | "U"
         | "V" | "W" | "X" | "Y" |
"Z"
         | "a" | "b" | "c" | "d" |
"e" | "f" | "g"
         | "h" | "i" | "j" | "k" |
"l" | "m" | "n"
         | "o" | "p" | "q" | "r" |
"s" | "t" | "u"
         | "v" | "w" | "x" | "y" |
"z" ;
id       = letter, { letter | "_" |
digit } ;
```

Fonte: Autoria própria (2023)

Alteramos e adicionamos mais regras à gramática inicial para possibilitar o uso de mais recursos da linguagem C++. As funcionalidades implementadas foram:

- instrução de “cout”;
- declaração de funções;
- chamada de funções;
- retorno de funções;
- variáveis do tipo void;
- declaração e atribuição de vetores;
- atribuição de variáveis usando operadores compostos;
- instruções “if... else” e “for”;
- operador de resto da divisão “%”.

A nova gramática livre de contexto do compilador escrita em EBNF está representada na Figura 8.

Figura 8 - Gramática livre de contexto do compilador produzido.

```
function = type, id, "(", [param], ")", "{",
{stmt}, "}" ;
param    = type, id, ["[", "]"], {tail} ;
tail     = ",", param ;

stmt     = decl, ";"
         | assign, ";"
         | call, ";"
         | "cout", "<<", operator, ";"
         | "return", operator,";"
         | newScope ;

decl     = type, id, position, [list]
         | type, id, [value] ;
list     = "=", "{", [operato, {",", operator}]
"}" ;
value    = "=", operator ;

assign     = change_one, local
```

```
term        = unary, {calc} ;
calc        = ("*" | "/" | "%"), unary ;

unary       = "!", unary
            | "-", unary
            | factor ;

factor      = "(", operator, ")"
            | change_one local
            | call
            | local {change_one}
            | boolean
            | integer
            | real ;

call        = id, "(", {args}, ")" ;
args        = operator, {tailAr} ;
tailAr      = ",", operator ;
```

```
           | local, change_one
           | local, [oper], "=", operator ;
change_one = "++" | "--" ;
oper       = "+" | "-"
           | "*" | "/"
           | "%" | "&"
           | "|" ;

newScope = "{", stmt, "}"
         | "if", "(", operator, ")", stmt,
["else", stmt]
         | "while", "(", operator")", stmt
         | "do", stmt, "while", "(", operator,
")", ";"
         | "for", "(", (decl | assign), ";",
              operator, ";", assign, ")", stmt ;

operator      = join, {lor} ;
lor           = "||", join ;

join          = equality, {land} ;
land          = "&&", equality ;

equality      = rel, {eqdif} ;
eqdif         = ("==" | "!="), rel ;

rel           = ari, {comp} ;
comp          = ("<" | "<=" | ">" | ">="), ari ;

ari           = term, {oper} ;
oper          = ("+" | "-"), term ;
```

```
local       = id, {position} ;
position    = "[", operator, "]" ;

type        = "void"
            | "int"
            | "float"
            | "bool" ;

boolean     = "true" | "false" ;

zero        = "0" ;
digit       = "0" | "1" | "2" | "3" | "4"
            | "5" | "6" | "7" | "8" | "9" ;
integer     = zero
            | ["-"], digit-zero, {digit} ;
real        = integer, ".", {digit} ;

letter      = "A" | "B" | "C" | "D" | "E" | "F"
| "G"
            | "H" | "I" | "J" | "K" | "L" | "M"
| "N"
            | "O" | "P" | "Q" | "R" | "S" | "T"
| "U"
            | "V" | "W" | "X" | "Y" | "Z"
            | "a" | "b" | "c" | "d" | "e" | "f"
| "g"
            | "h" | "i" | "j" | "k" | "l" | "m"
| "n"
            | "o" | "p" | "q" | "r" | "s" | "t"
| "u"
            | "v" | "w" | "x" | "y" | "z" ;
id          = letter, { letter | "_" | digit } ;
```

Fonte: Autoria própria (2023)

O compilador contém sete componentes lógicos: analisador léxico, analisador sintático, analisador semântico, gerador de código intermediário, gerador de código, gerenciador da tabela de símbolos e tratador de erros. Esses componentes, estão todos ligados, se comunicando entre si através de vários arquivos para processar o código fonte com todas as regras implementas.

O primeiro componente que entra em contato com o programa fonte é o analisador léxico. Ele é responsável por identificar e categorizar as palavras. No compilador, os arquivos “lexer.cpp” e “lexer.h”, vão identificar as palavras criando *structs* do tipo “Token”, que possui dois membros: o *int* “tag”, que classifica o tipo da palavra, e a *string* “lexeme”, que salva os caracteres da palavra. Se a palavra for apenas um caractere, como em um operador de atribuição (“=”), o membro “tag” recebe o número da tabela ASCII do caractere. Caso não seja apenas um caractere, a palavra precisa se encaixar em uma das possibilidades definidas na gramática. De forma explicita, a lista de possibilidades é declarada no *enum* “Tag”, logo, com a palavra identificada, o membro “tag” de “Token” recebe o valor de “Tag” equivalente. O enum “Tag” está representado na Figura 9.

Figura 9 - Lista de itens do *enum* Tag.

```
enum Tag
{
     ID = 256,   // -> [:alpha:]([:alnum:]|\_)*
     INTEGER,    // -> [:digit:]+
     REAL,       // -> [:digit:]\.([:digit:]*)?
     TYPE,       // -> void|int|float|bool
     INCLUDE,    // -> include
     USING,      // -> using
     NAMESPACE,  // -> namespace
     TRUE,       // -> true
     FALSE,      // -> false
     MAIN,       // -> main
     IF,         // -> if
     RETURN,     // -> return
     ATTADD,     // -> \+\=
     ATTSUB,     // -> \-\=
     ATTMUL,     // -> \*\=
     ATTDIV,     // -> \/\=
     ATTREM,     // -> \%\=
     ATTOR,      // -> \|\=
     ATTAND,     // -> \&\=
     PLUSPLUS,   // -> \+\+
     LESSLESS,   // -> \-\-
     OR,         // -> \|\|
     AND,        // -> \&\&
     EQ,         // -> \=\=
```

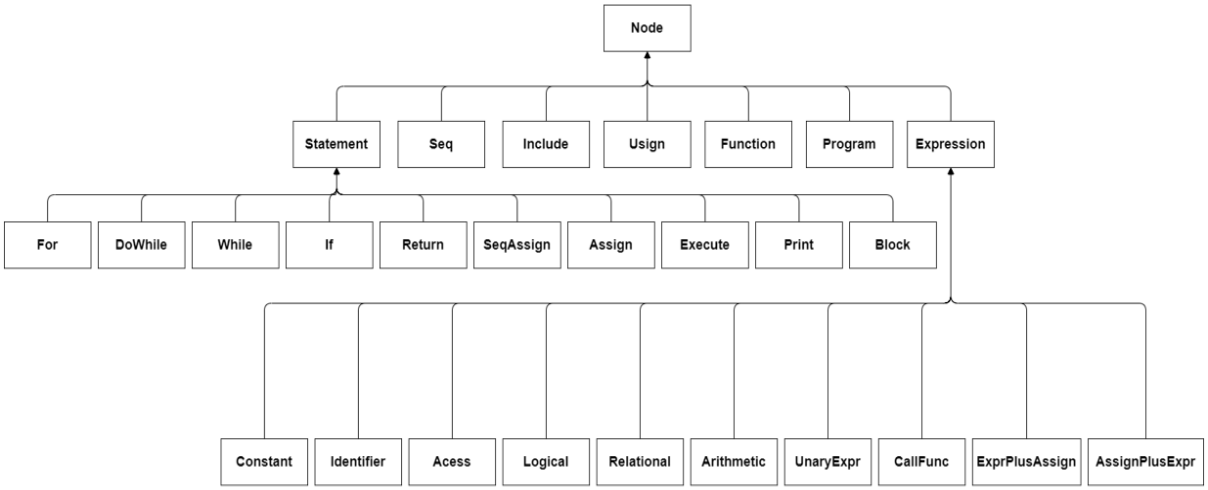
```
    ELSE,       // -> else
    WHILE,      // -> while
    DO,         // -> do
    FOR,        // -> for
    LIB,        // -> iostream
    PRINT,      // -> std\:\:cout
    NEQ,        // -> \!\=
    LTE,        // -> \<\=
    GTE,        // -> \=
    LST,        // -> \<\<
    SCOPE       // -> \:\:
};
```

Fonte: Autoria própria (2023)

Após a identificação, a palavra passa para o analisador sintático, onde é realizado a validação sintática respeitando as regras da gramática. Os arquivos que representam esse componente no compilador são os: "parser.h", "parserMain.cpp", "parserStatement.cpp" e "parserExpression.cpp". Neles, vão conter as regras da gramática traduzidas em comandos "*if... else*" e "*switch*". Para validar uma regra, que é uma linha da gramática, o analisador sintático precisa validar várias palavras dentro de um contexto específico. Se a validação ocorre corretamente, é criado um nó da árvore sintática. Se a validação não ocorre bem, o analisador sintático chama o componente tratador de erros.

Um nó da árvore sintática é representado através da *struct* "Node" e das suas subclasses, que estão representadas na Figura 10. Nele são armazenados o seu tipo sintático e as referências para os nós filhos. Se o nó for uma expressão, ele também guarda a tipagem da expressão. Com essas informações, é possível executar validações semânticas: tipagem e concordância entre nós. Todo esse trabalho é executado pelo analisador semântico, através dos arquivos "ast.cpp" e "ast.h". Onde, o analisar sintático encontra os padrões e chama o analisador semântico que faz as validações no momento da criação dos nós. Novamente, se a validação identificar um erro, nesse caso semântico e não mais na sintático, o tratador de erros também é chamado.

Figura 10 - Diagrama de classe dos tipos de nó da árvore sintática.



Fonte: Autoria própria (2023)

Além da criação da árvore, existe outra estrutura importante que o compilador usa. O controle da declaração e chamada de variáveis e funções não pode ser feito de forma simples através da árvore. Para cada validação, seria preciso percorrer toda a árvore atual para identificar se um elemento já foi declarado ou não. Em vez disso, usamos uma tabela *heap*, onde a verificação da existência de elementos é mais ágil. Essa tabela, denominada de tabela de símbolos, fica localizada dentro do gerenciador da tabela de símbolos, que é o componente responsável pelos métodos que manipulam a tabela e está implementado nos arquivos: "functable.cpp" e "functable.h", para gerenciar as funções; e "symtable.cpp" e "symtabe.h", para gerenciar as variáveis.

É importante salientar que o gerenciador da tabela de símbolos e o tratador de erros podem

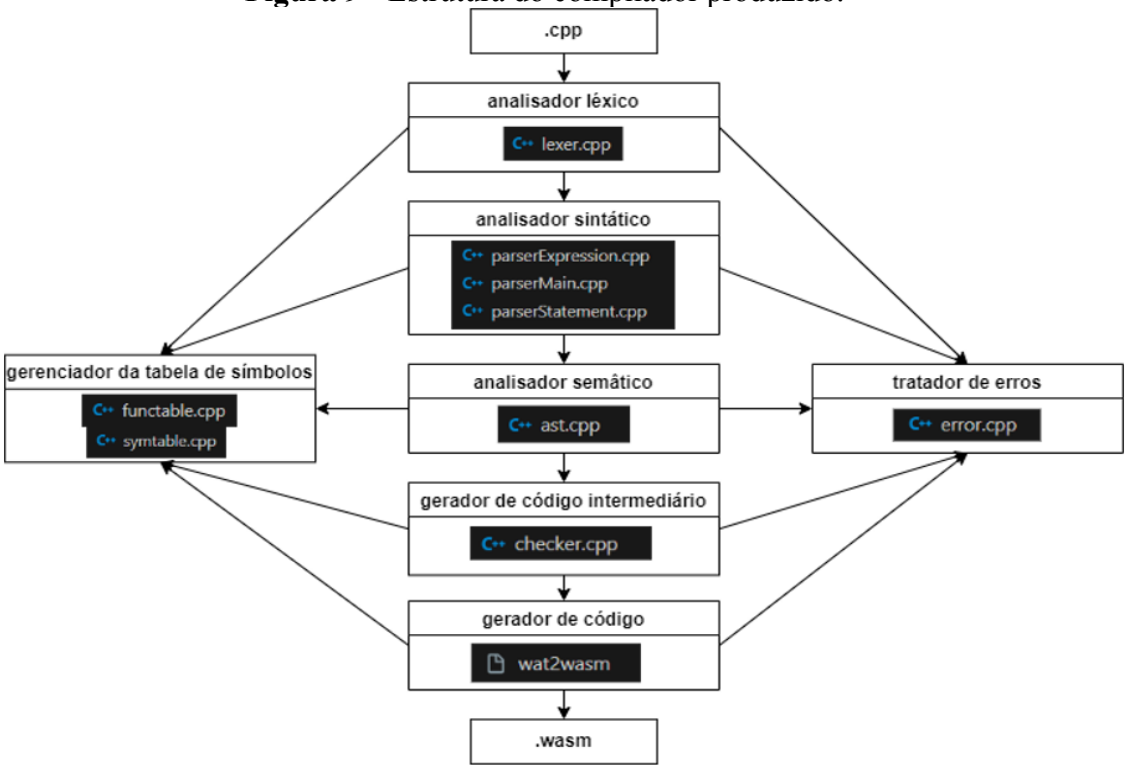
ser chamados por todos os outros componentes do compilador, tendo em vista que os seus métodos são utilizados do início ao fim da compilação. Um erro pode ocorrer em todos os componentes, assim como a necessidade de validar símbolos, sejam funções ou variáveis.

Seguindo o processo de compilação, após a geração da árvore sintática, chegamos na geração do programa alvo. Aqui, todo o programa fonte já foi processado, agora precisamos ler a árvore e gerar o programa alvo. Para isso, o gerador de código intermediário entra em ação, sendo o último componente que criamos. Sua implementação foi feita no "checker.cpp" e "checker.h". De forma simples, percorremos a árvore e identificamos o tipo de nó por meio de um *switch*, onde cada *case* possui a conversão direta de um tipo de nó para seu o código WASM equivalente. Em cada iteração, é escrito uma parte do código alvo. No final da leitura de toda a árvore, o código WASM em formato textual está pronto.

Para nossos objetivos de implementação do compilador, aqui é nosso ponto de parada. Porém, um código WASM no formato textual não pode ser executado, assim, precisamos converter o mesmo em formato binário. Desse modo, decidimos usar o executável disponibilizado pelo Wabt, Webassembly/wabt (2023), chamado de "wat2wasm", que converte de um formato para o outro. Assim, o "wat2wasm" foi integrado ao nosso compilador como o gerador de código, que na prática é o componente responsável por gerar o código final após todos os passos da compilação.

De forma holística, temos todos os componentes e suas comunicações em si representadas na Figura 9.

Figura 9 - Estrutura do compilador produzido.



Fonte: Autoria própria (2023)

Com o compilador implementado e funcional, conseguimos compilar qualquer código C++ que esteja dentro das regras da nossa gramática e gerar um código WASM que pode ser executado em um dos ambientes com suporte a WASM, sejam navegadores ou *runtimes* JS. Temos um exemplo de programa fonte e seu respectivio programa alvo na Figura 10.

Figura 10 - Exemplo de programa fonte C++ (a) e programa alvo WASM (b).

(a)

```
int sum()
{
  int a = 1;
  a = a * 2;
  return a;
}
```

(b)

```
(module
  (func $sum (result
i32)
      (local $a i32)
      (local.set $a
         (i32.const 1))
      (local.set $a
         (i32.mul
            (local.get
$a)
            (i32.const
2)))
      (return)))
```

Fonte: Autoria própria (2023)

O objetivo prinicipal desse trabalho era reduzir a complexidade do programa alvo deixando-o mais próximo do programa fonte, de modo que fosse fácil achar as equivalencias de um código para o outro. Para mensurar o alcance do objetivo, decidimos compilar um mesmo código C++ usando o compilador oficial Emscripten e o nosso compilador e compararmos os tamanhos dos arquivos .wasm gerados e a quantidade de linhas nos arquivos .wat subjacentes dos .wasm.

Para o objetivo secundário, que era gerar código WASM que fosse mais eficiente do que o respectivo código JS simples, resolvermos comparar o tempo de execução de alguns algoritmos. Implementamos os mesmos algortimos em C++ e compilamos para WASM usando nosso compilador e implementamos também em JS. Para medir o tempo de execução dos mesmos, usamos a biblioteca de benchmarking Benchmark.js, disponível em Bynens (2023). O dispositivo usado nas medições foi um Notebook Asus Aspire 315-23, processador AMD Ryzen 5 3500U 2.10 ghz, 12 GB de mémoria RAM, sistema operacional Windows 11 Home Versão 22H2. Os ambientes de execução, tanto navegadores quanto o runtime, estão descritos na Tabela 1. Os algoritmos usados estão descritos da Tabela 2.

Tabela 1 - Ambientes usados nos testes.

Ambiente	Versão
Chrome	114.0.0
Firefox	115.0
Edge	115.0.1901.183
Opera	99.0.0.0
Node.js	18.16.0

Fonte: Autoria própria (2023)

Tabela 2 - Algoritmos usados nos testes.

Algoritmo	Descrição	Tipos de instruções
Soma de vetor	Soma todos os itens de um vetor de números.	Declaração de variável, acesso de vetor, estrutura de fluxo (*for*), operação de soma, acesso de variável, atribuição de variável.
Busca binária	Busca em vetores que segue o paradigma de divisão e conquista. Ela	Declaração de variável, acesso de vetor, estrutura de

	parte do pressuposto de que o vetor está ordenado e realiza sucessivas divisões do espaço de busca comparando o elemento buscado com o elemento no meio do vetor.	fluxo (*if, for*), operação de divisão, operação de comparação, acesso de variável, atribuição de variável.
Fibonacci recursivo	Calcula o valor de uma certa posição da sequência de Fibonacci usando recursividade.	Acesso de variável, operação de comparação, estrutura de fluxo (*if*), chamada de função.

Fonte: Autoria própria (2023)

Cada algoritmo foi escolhido pensando em uma de suas especificidades na sua natureza de execução. A soma de vetor acessa os valores de uma lista e os soma interativamente. Na busca binária aplica-se um algoritmo do tipo divisão e conquista, acessando elementos específicos da lista e realizando comparações logicas. E Fibonacci, por sua vez, aplica recursividade em sua execução. Assim, os algoritmos apresentam divergência em suas construções que enriquecem o escopo de atuação dos testes.

2.2 Resultados e Discussões

Para o primeiro objetivo, realizamos o grupo de testes que avaliaram a complexidade dos códigos, medindo os tamanhos dos arquivos ".wasm" gerados e a quantidade de linhas nos arquivos ".wat". Realizamos as compilações dos códigos em C++ nos dois compiladores, o nosso compilador e o Emscripten, com os resultados expostos na Tabela 3.

Tabela 3 - Tamanhos dos arquivos compilados para cada teste.

	Nosso compilador		Emscripten	
Algoritmo	Tamanho do ".wasm"	Quantidade de linhas no ".wat"	Tamanho do ".wasm"	Quantidade de linhas no ".wat"
Soma de vetor	273 bytes	110	1.312 bytes	428
Busca binária	332 bytes	139	1.543 bytes	545
Fibonacci	88 bytes	24	1.072 bytes	346

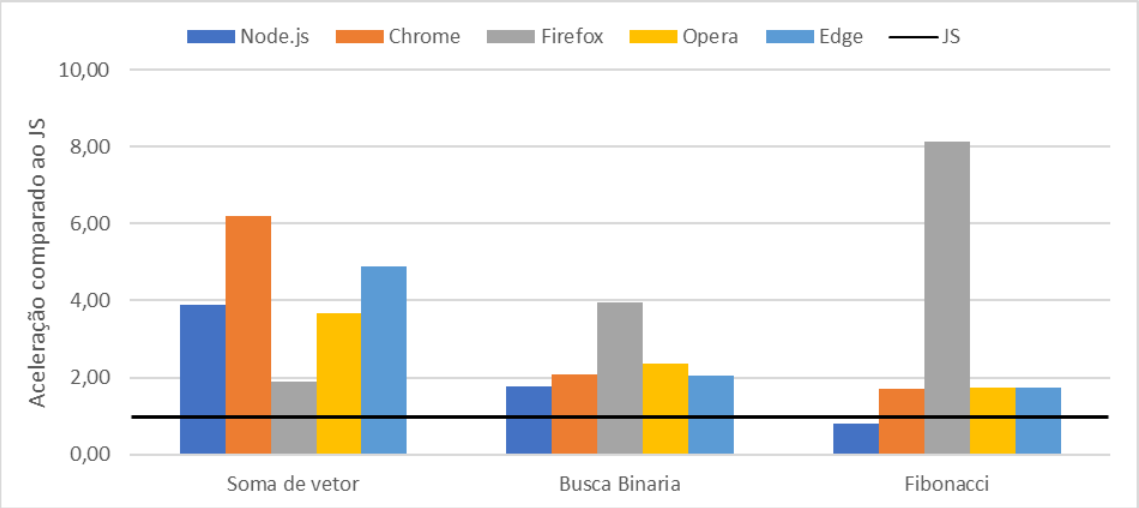
Fonte: Autoria própria (2023)

Comparando os tamanhos do grupo de arquivos gerados pelo nosso compilador e o do grupo de arquivos gerados pelo Emscripten, percebemos uma redução do tamanho dos arquivos entre 78% e 92% de um grupo para o outro. Em relação a quantidade de linhas, a redução ficou entre 74% e 93%. Comprovando que o tamanho foi reduzido, nos permitindo inferir que a complexidade foi reduzida proporcionalmente. Além disso, com a diminuição dos tamanhos dos arquivos, temos ganho em armazenamento e memória.

Para o segundo objetivo, realizamos os testes de desempenho, medindo a velocidade de execução, entre os códigos compilados por nosso compilador e os mesmos códigos escritos em JS. Usamos os mesmos algoritmos do primeiro grupo de testes, executando-os em diversos ambientes e retirando o valor médio com a ajuda do Benchmark.js. Resolvermos apresentar os dados no mesmo formato que Herrera (2018) apresenta em seu trabalho, trazendo a velocidade do JS como padrão de uma unidade de medida e o restante em função dela, isso é, a acerelação

das velocidades em relação a velocidade do JS. Os dados estão expostos na Figura 11.

Figura 11 - Resultado do desempenho dos WASM gerados pelo nosso compilador em relação aos códigos JS.



Fonte: Autoria própria (2023)

Analisando os dados, percebemos que em todas as medições, com exceção do Fibonacci sendo executado no Node.js, o WASM se saiu melhor do que o JS, com uma média de três vezes mais rápido. Comprovando que o nosso compilador consegue gerar código WASM mais eficiente em tempo de execução que o mesmo código escrito em JS puro.

3 CONSIDERAÇÕES FINAIS

Esse trabalho teve como objetivo construir um compilador de código C++ para WebAssembly de modo que o código compilado fosse o mais simples e limpo possível, de tal forma que fosse fácil o programador depurar e achar as conversões feitas de um código para o outro. Para isso, desenvolvemos um compilador funcional e eficaz na geração de código WASM, que consegue ser mais eficiente em tempo de execução do que o mesmo código JS equivalente. Assim, viabilizando o uso do WASM e do compilador no dia a dia para funções simples, que estão dentro da sintaxe aceita pelo compilador, possibilitando a otimização de código em aplicações da Web.

Como melhoria futura, vemos que o compilador ainda pode ser ampliado, aceitando toda a sintaxe oficial do C++ que nesse momento ainda não está sendo contemplada. Além disso, o compilador poderia gerar o código JS que chama o WASM, no mesmo padrão que o compilador Emscripten opera, que gera tanto o WASM quanto o JS. Dessa forma, melhoraria a experiência do programador no uso do nosso compilador, tornando a compilação de código ainda mais produtiva.

REFERÊNCIAS

2023 Developer Survey. **Stack Overflow**. Disponível em: https://survey.stackoverflow.co/2023/#section-most-popular-technologies-programming-scripting-and-markup-languages. Acesso em: 31 ago. 2023.

ABOUT EMSCRIPTEN, **Emscripten**, 2015. Disponível em:

https://emscripten.org/docs/introducing_emscripten/about_emscripten.html. Acesso em: 10 set. 2023.

AHO, Alfred V. et al. **Compiladores – Princípios, Técnicas e Ferramentas**. 2. ed. Pearson. 2008.

BYNENS, Mathias; DALTONHTTPS, John-David. **Benchmark.js v2.1.2**. Disponível em: https://benchmarkjs.com/docs/. Acesso em: 15 set. 2023.

COOPER, Keith; TORCZON, Linda. **Construindo Compiladores**. 2. ed. Elsevier, 2014.

ELLIOTT, Eric. Why we Need WebAssembly. **Medium**, 27 jun. 2015. Disponível em: https://medium.com/javascript-scene/why-we-need-webassembly-an-interview-with-brendan-eich-7fb2a60b0723. Acesso em: 8 set. 2023.

EXECUTION. **GitHub**. Disponível em: https://webassembly.github.io/spec/core/exec/index.html. Acesso em: 15 set. 2023.

HAAS, Andreas et al. **Bringing the web up to speed with WebAssembly**. In: Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation. 2017. p. 185-200.

HERRERA, David et al. **WebAssembly and JavaScript Challenge:** Numerical program performance using modern browser technologies and devices. University of McGill, Montreal: QC, Technical report SABLE-TR-2018-2, 2018.

INTERNATIONAL ORGANIZATION FOR STANDARDIZATION. **ISO/IEC 14977:1996**: Information technology — Syntactic metalanguage — Extended BNF. 2023.

JUDSONSS/COMPILADORES. **GitHub**. Disponível em: https://github.com/JudsonSS/Compiladores. Acesso em: 15 set. 2023.

WEBASSEMBLY. **MDN Web Docs**. Disponível em: https://developer.mozilla.org/en-US/docs/WebAssembly. Acesso em: 15 set. 2023.

WEBASSEMBLY COMMUNITY GROUP. **W3C**. Disponível em: https://www.w3.org/community/webassembly/. Acesso em: 6 set. 2023.

WEBASSEMBLY/WABT. **GitHub**. Disponível em: https://github.com/webassembly/wabt. Acesso em: 15 set. 2023.

WORKING DRAFT. **asm.js**, 18 ago. 2014. Disponível em: http://asmjs.org. Acesso em: 8 set. 2023.

ZAKAI, Alon. **Emscripten**: an LLVM-to-JavaScript compiler. In: Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion. 2011. p. 301-312.