UNIVERSIDADE DE CAXIAS DO SUL
ÁREA DO CONHECIMENTO DE CIÊNCIAS EXATAS E ENGENHARIAS

GUSTAVO LIMA SUSIN

WEBALGO: DESENVOLVIMENTO DE UM COMPILADOR DA LINGUAGEM C E UMA MÁQUINA VIRTUAL BASEADA EM REGISTRADORES UTILIZANDO C3E NA WEB

CAXIAS DO SUL

2024

GUSTAVO LIMA SUSIN

WEBALGO: DESENVOLVIMENTO DE UM COMPILADOR DA LINGUAGEM C E UMA MÁQUINA VIRTUAL BASEADA EM REGISTRADORES UTILIZANDO C3E NA WEB

Trabalho de Conclusão de Curso apresentado como requisito parcial à obtenção do título de Bacharel em Ciência da Computação na Área do Conhecimento de Ciências Exatas e Engenharias da Universidade de Caxias do Sul.

Orientador: Prof. Dr. Ricardo Vargas Dorneles

CAXIAS DO SUL

2024

GUSTAVO LIMA SUSIN

WEBALGO: DESENVOLVIMENTO DE UM COMPILADOR DA LINGUAGEM C E UMA MÁQUINA VIRTUAL BASEADA EM REGISTRADORES UTILIZANDO C3E NA WEB

Trabalho de Conclusão de Curso apresentado como requisito parcial à obtenção do título de Bacharel em Ciência da Computação na Área do Conhecimento de Ciências Exatas e Engenharias da Universidade de Caxias do Sul.

Aprovado(a) em 26/11/2024

BANCA EXAMINADORA

Prof. Dr. Ricardo Vargas Dorneles
Universidade de Caxias do Sul - UCS

Prof. Me. Alexandre Erasmo Krohn Nascimento
Universidade de Caxias do Sul - UCS

Prof. Dr. Leonardo Pellizzoni
Universidade de Caxias do Sul - UCS

RESUMO

A prática de exercícios de programação no início do aprendizado é essencial para os alunos, e as disciplinas introdutórias dos cursos de computação da Universidade de Caxias do Sul (UCS) reconhecem essa importância. Para aprimorar os conhecimentos em programação, a UCS disponibiliza a ferramenta WebAlgo, projetada para oferecer uma variedade de exercícios na linguagem C. O WebAlgo integra um compilador de subconjuntos específicos da linguagem C, juntamente com um interpretador que realiza as operações lógicas e aritméticas dos algoritmos desenvolvidos pelos alunos. No entanto, a atual implementação do interpretador no WebAlgo utiliza uma lista duplamente encadeada, o que dificulta a expansão e a manutenção da ferramenta, dada a complexidade do código existente. Diante desse cenário, este trabalho propõe o desenvolvimento de um novo compilador, mantendo os subconjuntos da linguagem C utilizados no WebAlgo, mas agora, projetado para a web, fazendo uso da estrutura de código intermediário no qual será utilizado o Código de Três Endereços (C3E). Essa abordagem visa simplificar e facilitar as futuras manutenções e expansões do WebAlgo. Além da implementação do novo compilador, o trabalho inclui a definição de uma máquina virtual baseada em registradores virtuais que interpretará o código intermediário gerado.

Palavras-chave: WebAlgo, Compiladores, Código de Três Endereços, Máquina Virtual

ABSTRACT

The practice of programming exercises at the beginning of learning is essential for students, and the introductory courses in computer science at the University of Caxias do Sul (UCS) recognize this importance. To enhance programming skills, UCS provides the WebAlgo tool, designed to offer a variety of exercises in the C language. WebAlgo integrates a compiler for specific subsets of the C language, along with an interpreter that performs logical and arithmetic operations for algorithms developed by students. However, the current implementation of the interpreter in WebAlgo uses a doubly linked list, making expansion and maintenance challenging due to the complexity of the existing code. In light of this scenario, this work proposes the development of a new compiler, retaining the subsets of the C language used in WebAlgo, but now, designed for the web and utilizing an intermediate code structure known as *Three-Address Code* (TAC). This approach aims to simplify and facilitate future maintenance and expansions of WebAlgo. In addition to the new compiler implementation, the work includes defining a virtual machine based on virtual registers to interpret the generated intermediate code.

Keywords: WebAlgo, Compilers, Three-Address Code, Virtual Machine

LISTA DE FIGURAS

Figura 1 – Compilador 16
Figura 2 – Fases do compilador 17
Figura 3 – C3E exemplo código 19
Figura 4 – Exemplo arquivo .wat 23
Figura 5 – Exemplo CodeMirror 25
Figura 6 – Exemplo erro de compilação 26
Figura 7 – Tabela de símbolos inicial de Algoritmo 13 41
Figura 8 – Exemplo da arquitetura de sistema e suas respectivas ferramentas 50
Figura 9 – Exemplo de identificação de tokens na liguagem C com o editor CodeMirror 51
Figura 10 – Modelo de domínio do compilador 54
Figura 11 – Mockup de tela do WebAlgo Web - execução sucedida 55
Figura 12 – Mockup de tela do WebAlgo Web - execução com erro léxico 56
Figura 13 – Mockup de tela do WebAlgo Web - histórico de variáveis 56
Figura 14 – Mockup de tela do WebAlgo Web - interrupção para aguardo de digitação do usuário devido a chamada da função *scanf* 57
Figura 15 – Mockup de tela do WebAlgo Web - depurador, breakpoint e botão passo a passo 58
Figura 16 – *Frontend* do Compilador *Web* 59
Figura 17 – *Tokens* utilizados no Compilador *Web* 61
Figura 18 – Erro sintático na gramática do comando *printf* 63
Figura 19 – Tabela de Símbolos do Algoritmo 29 67
Figura 20 – Objeto BytecodeC3E que será interpretado pela máquina virtual 70
Figura 21 – Estrutura da máquina virtual 74
Figura 22 – Feedback dos alunos ao Compilador *Web* 79
Figura 23 – Exemplo no compilador - Lê um número e escreve a sequência de Fibonacci até valor lido 86
Figura 24 – Exemplo no compilador - Realiza a leitura de 5 números inteiros e ordena-os em ordem crescente 88
Figura 25 – Exemplo no compilador - Lê um número e verifica se o número lido é primo 91
Figura 26 – Exemplo no compilador - Lê a dimensão da matriz, em seguinda realiza a leitura dos valores da matriz e ao final escreve a soma da diagonal principal 94
Figura 27 – Exemplo no compilador - Teste realizado para chamadas de função e funcionamento correto de escopo de variáveis 96
Figura 28 – Erro léxico no caractere ’’ 98
Figura 29 – Erro léxico na declaração da bibioteca *stdio.h* 99
Figura 30 – Erro sintático devido a falta do caractere ’;’ na declaração da variável somar 100

Figura 31 – Erro sintático devido a falta do comando *return* na função soma 101
Figura 32 – Erro sintático devido a falta do caractere ')' no comando *printf* 101
Figura 33 – Erro semântico devido a variável somar não estar declarada 102
Figura 34 – Erro semântico devido a variável declarada já existir uma macro com o mesmo identificador 103
Figura 35 – Erro semântico devido a variável declarada duas vezes no mesmo escopo . . 103
Figura 36 – Questionário aplicado para os alunos avaliarem o compilador web página 1 . 104
Figura 37 – Questionário aplicado para os alunos avaliarem o compilador web página 2 . 105
Figura 38 – Questionário aplicado para os alunos avaliarem o compilador web página 3 . 106
Figura 39 – Questionário aplicado para os alunos avaliarem o compilador web página 4 . 107
Figura 40 – Lista de exercício utilizada pelos alunos - Página 1 108
Figura 41 – Lista de exercício utilizada pelos alunos - Página 2 109
Figura 42 – Lista de exercício utilizada pelos alunos - Página 3 110
Figura 43 – Lista de exercício utilizada pelos alunos - Página 4 111
Figura 44 – Lista de exercício utilizada pelos alunos - Página 5 112

LISTA DE ALGORITMOS

Algoritmo 1 Exemplo de instrução condicional gerada para código de três endereços . 19
Algoritmo 2 Exemplo *while* de instrução de repetição gerada para código de três endereços 20
Algoritmo 3 Exemplo *for* de instrução de repetição gerada para código de três endereços 20
Algoritmo 4 Exemplo *do while* de instrução de repetição gerada para código de três endereços 21
Algoritmo 5 Exemplo de código em máquina virtual de pilha 22
Algoritmo 6 Exemplo de código em máquina virtual de registradores 22
Algoritmo 7 Exemplo de estrutura de atribuição gerada para código de três endereços . 37
Algoritmo 8 Exemplo de estrutura condicional genérica gerada para código de três endereços 37
Algoritmo 9 Exemplo de estrutura de repetição *while* genérica gerada para código de três endereços 38
Algoritmo 10 Exemplo de estrutura de repetição *for* genérica gerada para código de três endereços 38
Algoritmo 11 Exemplo de estrutura de repetição *do while* genérica gerada para código de três endereços 38
Algoritmo 12 Exemplo de estrutura de atribuição de vetor genérica gerada para código de três endereços 39
Algoritmo 13 Exemplo algoritmo de fibonacci em C para exemplificar a criação da tabela de símbolos do compilador 40
Algoritmo 14 Exemplo de instrução de incremento equivalente na linguagem C 44
Algoritmo 15 Exemplo de instrução de decremento equivalente na linguagem C 44
Algoritmo 16 Exemplo de instrução de E lógico equivalente na linguagem C 45
Algoritmo 17 Exemplo de instrução de OU lógico equivalente na linguagem C 45
Algoritmo 18 Exemplo de instrução de igualdade equivalente na linguagem C 46
Algoritmo 19 Exemplo de instrução de menor que equivalente na linguagem C 46
Algoritmo 20 Exemplo de instrução de menor igual que equivalente na linguagem C . . 46
Algoritmo 21 Exemplo de instrução de maior que equivalente na linguagem C 47
Algoritmo 22 Exemplo de instrução de maior igual que equivalente na linguagem C . . . 47
Algoritmo 23 Exemplo de instrução de desvio condicional equivalente na linguagem C . 47
Algoritmo 24 Exemplo de chamada de função equivalente na linguagem C 48
Algoritmo 25 Exemplo de eetorno de função equivalente na linguagem C 48
Algoritmo 26 Chamadas da função *getToken()* após verificações de *token* na análise sintática do comando *WHILE* da gramática do compilador *web* 62
Algoritmo 27 Análise sintática do comando *while* da gramática do compilador *web* . . . 63

Algoritmo 28 Análise semântica - Criação de escopos do comando *while* da gramática do compilador *web* 65
Algoritmo 29 Exemplo de código C para demonstração da tabela de símbolos 66
Algoritmo 30 Geração de código de três endereços do comando *while* da gramática do compilador *web* 68
Algoritmo 31 Código de três endereços gerado pelo compilador correspondente ao algoritmo 29 71
Algoritmo 32 Exemplo de código C - Lê um número e escreve a sequência de Fibonacci até valor lido 85
Algoritmo 33 Código de três endereços - Lê um número e escreve a sequência de Fibonacci até valor lido 86
Algoritmo 34 Exemplo de código C - Realiza a leitura de 5 números inteiros e ordena-os em ordem crescente 87
Algoritmo 35 Código de três endereços - Realiza a leitura de 5 números inteiros e ordena-os em ordem crescente 88
Algoritmo 36 Exemplo de código C - Lê um número e verifica se o número lido é primo 90
Algoritmo 37 Código de três endereços - Lê um número e verifica se o número lido é primo 92
Algoritmo 38 Exemplo de código C - Lê a dimensão da matriz, em seguinda realiza a leitura dos valores da matriz e ao final escreve a soma da diagonal principal 93
Algoritmo 39 Código de três endereços - Lê a dimensão da matriz, em seguinda realiza a leitura dos valores da matriz e ao final escreve a soma da diagonal principal 94
Algoritmo 40 Exemplo de código C - Teste realizado para chamadas de função e funcionamento correto de escopo de variáveis 96
Algoritmo 41 Código de três endereços - Teste realizado para chamadas de função e funcionamento correto de escopo de variáveis 97

LISTA DE SIGLAS

AST Árvore Sintática Abstrata

AJAX *Asynchronous JavaScript And XML*

CSS *Cascading Style Sheets*

C3E Código de Três Endereços

CLR *Common Language Runtime*

CSE *Computer Science Education*

GCC **GNU Compiler Collection**

HTML *Hypertext Markup Language*

JVM Máquina Virtual Java

VM Máquina Virtual

TAC *Three-Address Code*

TCC Trabalho de Conclusão de Curso

UCS Universidade de Caxias do Sul

SUMÁRIO

1 INTRODUÇÃO 13
1.1 PROBLEMA DE PESQUISA 14
1.2 QUESTÃO DE PESQUISA 14
1.3 OBJETIVOS DO TRABALHO 15
1.4 ESTRUTURA DO TRABALHO 15

2 REFERENCIAL TEÓRICO 16
2.1 COMPILADORES 16
2.1.1 Geração de código intermediário: código de três endereços 18
2.1.1.1 Código de três endereços - condicionais 19
2.1.1.2 Código de três endereços - repetição 20
2.2 MÁQUINAS VIRTUAIS 21
2.2.1 WebAssembly 22
2.3 TECNOLOGIAS PARA DESENVOLVIMENTO WEB 24
2.3.1 Javascript 24
2.3.2 Bibliotecas para editores de texto 24

3 TRABALHOS RELACIONADOS 26
3.1 UM SIMPLES INTERPRETADOR DE C++ DESENVOLVIDO EM JAVASCRIPT 26
3.2 *A FRIENDLY ONLINE C COMPILER TO IMPROVE PROGRAMMING SKILLS BASED ON STUDENT SELF-ASSESSMENT* 27
3.3 *CHASM* - UM SIMPLES COMPILADOR PARA LINGUAGEM WEBASSEMBLY 28

4 PROPOSTA DE SOLUÇÃO 29
4.1 SUBCONJUNTOS DE C 29
4.2 GRAMÁTICA DOS SUBCONJUNTOS DE C 31
4.3 ESTRUTURA C3E - CÓDIGO DE TRÊS ENDEREÇOS (*BYTECODE*) . . 33
4.3.1 Estrutura condicional - *if* 37
4.3.2 Estrutura de repetição - *while, for* e *do while* 37
4.3.3 Arranjos ou atribuições indexadas (vetores e matrizes) 39
4.4 CONTROLE DE VARIÁVEIS - TABELA DE SÍMBOLOS 39
4.5 ARQUITETURA DA MÁQUINA VIRTUAL 42
4.5.1 Operações artiméticas 42
4.5.1.1 Soma 43

4.5.1.2 Subtração . . . 43
4.5.1.3 Multiplicação . . . 43
4.5.1.4 Divisão . . . 43
4.5.1.5 Módulo ou resto da divisão . . . 43
4.5.1.6 Incremento . . . 44
4.5.1.7 Decremento . . . 44
4.5.2 Operações lógicas . . . 44
4.5.2.1 E lógico . . . 44
4.5.2.2 Ou lógico . . . 45
4.5.3 Operações relacionais . . . 45
4.5.3.1 Igualdade . . . 45
4.5.3.2 Menor que . . . 46
4.5.3.3 Menor igual que . . . 46
4.5.3.4 Maior que . . . 46
4.5.3.5 Maior igual que . . . 47
4.5.4 Operações de desvios . . . 47
4.5.4.1 Desvio condicional . . . 47
4.5.4.2 Desvio incondicional . . . 48
4.5.4.3 Empilhamento de parâmetros e chamada de função . . . 48
4.5.4.4 Retorno de função . . . 48
4.6 ARQUITETURA DO SISTEMA . . . 49
4.6.1 Casos de uso do sistema . . . 51
4.6.1.1 Caso de uso 1: executar código programado . . . 52
4.6.1.2 Caso de uso 2: depuração do código desenvolvido . . . 53
4.6.2 Modelo de domínio do sistema . . . 54
4.6.3 *Mockup* de tela do compilador . . . 54
4.7 DEPURADOR . . . 57

5 DESENVOLVIMENTO . . . 59
5.1 ESTRUTURA DE DESENVOLVIMENTO DA TELA DO COMPILADOR 59
5.2 ANÁLISE LÉXICA . . . 60
5.3 ANÁLISE SINTÁTICA . . . 62
5.4 ANÁLISE SEMÂNTICA . . . 64
5.5 GERAÇÃO DE CÓDIGO DE TRÊS ENDEREÇOS (C3E) . . . 68
5.6 MÁQUINA VIRTUAL BASEADA EM REGISTRADORES VIRTUAIS . . 73

6 VALIDAÇÃO E TESTES . . . 77
6.1 TEMPO DE EXECUÇÃO DO COMPILADOR WEB COMPARADO AO WEBALGO . . . 80

7 CONSIDERAÇÕES FINAIS 81
7.1 TRABALHOS FUTUROS 82

REFERÊNCIAS 83

APÊNDICE A – CASOS DE TESTE REALIZADOS NO COMPILADOR BEM-SUCEDIDOS 85

APÊNDICE B – CASOS DE TESTE REALIZADOS NO COMPILADOR COM ERRO LÉXICO 98

APÊNDICE C – CASOS DE TESTE REALIZADOS NO COMPILADOR COM ERRO SINTÁTICO 100

APÊNDICE D – CASOS DE TESTE REALIZADOS NO COMPILADOR COM ERRO SEMÂNTICO 102

APÊNDICE E – QUESTIONÁRIO APLICADO EM SALA DE AULA COM OS ALUNOS QUE TESTARAM O COMPILADOR WEB 104

ANEXO A – EXERCÍCIOS REALIZADOS DURANTE TESTES COM ALUNOS EM AMBIENTE DE SALA DE AULA 108

1 INTRODUÇÃO

A busca constante pela melhora da educação no curso de Ciência da Computação está sempre em progresso. Professores encaram desafios nos cursos introdutórios de programação para auxiliar seus alunos a desenvolverem a compreensão em lógica de programação (QIAN; LEHMAN, 2017). A comunidade de *Computer Science Education* (CSE) tem conhecimento sobre a dificuldade que alunos recém ingressos ao curso possuem no aprendizado de programação, consequentemente resultando no mau desempenho do aluno e ocasionando a sua desistência (BERGIN; REILLY, 2005).

O artigo de (JESUS; BRITO, 2009) apresenta uma revisão de literatura referente ao ensino-aprendizagem dos alunos ingressantes em disciplinas de programação e suas dificuldades. O autor menciona a importância do conhecimento das linguagens de programação através da prática em exercícios. Para isso, existem ferramentas e plataformas específicas para fortalecimento do ensino em programação, como exemplo, a Code.Org[1] (SILVA *et al.*, 2015) e o Beecrowd[2] (CRUZ *et al.*, 2022). Além destas ferramentas públicas, as instituições de ensino superior utilizam ambientes de aprendizados desenvolvidos internamente para auxiliar o aluno no aprendizado. Como exemplo, a Universidade de Caxias do Sul (UCS) utiliza uma ferramenta nomeada de WebAlgo para fortalecer a compreensão e desenvolvimento do aluno nas disciplinas introdutórias, programação um e dois, do curso de Ciência da Computação.

O WebAlgo é uma ferramenta amplamente utilizada por professores de Ciência da Computação para introduzir conceitos de programação aos alunos. Esta aplicação oferece uma série de exercícios sequenciais, condicionais e iterativos, projetados para reforçar a compreensão dos alunos e aprimorar suas habilidades em linguagem de programação C (DORNELES; JR; ADAMI, 2010). Desenvolvido pelos professores da universidade, o WebAlgo está em operação ininterrupta desde 2009. Durante esse período, foram realizadas manutenções e implementações na ferramenta, incluindo projetos de conclusão que a utilizaram como base. Um exemplo notável é a criação de um compilador e uma máquina virtual para um subconjunto da linguagem Python, um trabalho desenvolvido por (MIOTTO, 2019).

Apesar da validação da ferramenta e das melhorias implementadas ao longo desses anos, durante as discussões com o professor e orientador deste projeto, foi destacada a complexidade do código-fonte utilizado para geração do código intermediário no qual se aplica o uso de uma lista duplamente encadeada. Este detalhe resulta em desafios de escalabilidade da ferramenta, bem como na dificuldade de fornecer suporte ao sistema.

[1] <https://code.org/>
[2] <https://www.beecrowd.com.br>

1.1 PROBLEMA DE PESQUISA

O WebAlgo representa uma ferramenta educacional dedicada ao aprimoramento da lógica e à criação de soluções em exercícios de programação. Essa plataforma é empregada nas disciplinas iniciais de programação do curso de Ciência da Computação da Universidade de Caxias do Sul, com o objetivo de auxiliar os alunos no desenvolvimento de suas habilidades em programação. Para utilizar esta ferramenta, o aluno primeiro desenvolve o seu código-fonte e, em seguida, realiza a compilação do mesmo. No caso de uma falha na compilação, o programa emite um alerta de erro que identifica a linha e a posição onde ocorreu a inconsistência no código do aluno, seja por erro léxico, sintático ou semântico. No entanto, se a compilação for bem-sucedida, o aluno conclui o desenvolvimento do exercício e recebe uma nota de desempenho com base no número de instruções geradas pelo seu código. Além disso, os professores podem visualizar e acompanhar o progresso dos alunos no desenvolvimento de suas atividades.

O WebAlgo é um programa desktop desenvolvido em linguagem Java que simula a funcionalidade de um compilador/interpretador para reconhecimento das linguagens de programação Português Estruturado e C, conforme mencionado por Fernando em seu trabalho (LIMA, 2017). Devido à sua natureza como compilador, o código-fonte do WebAlgo apresenta um alto grau de complexidade, principalmente na etapa de geração de código intermediário. Nessa fase, o programa faz uso de estruturas de dados como listas duplamente encadeadas para representar árvores sintáticas. Entretanto, a complexidade na geração do código intermediário pode dificultar a depuração do código-fonte do WebAlgo, resultando em atrasos nas manutenção e atualização da ferramenta.

Em vista disso, este trabalho apresenta a reescrita do compilador e do gerador de código intermediário, substituindo a geração de uma árvore sintática através da lista duplamente encadeada pela geração de código de três endereços, também conhecido como C3E. Deste modo, foi necessário implementar também a interpretação do código intermediário para que o código-fonte do aluno possa ser executado. Para isso, foi necessário a definição de uma máquina virtual para execução deste código de três endereços.

1.2 QUESTÃO DE PESQUISA

A migração do código-fonte da geração do código intermediário no WebAlgo, que atualmente utiliza uma árvore sintática com listas duplamente encadeadas, para um código de três endereços, tem o potencial de melhorar o tempo de manutenção e a escalabilidade da aplicação em atualizações futuras?

1.3 OBJETIVOS DO TRABALHO

O objetivo deste trabalho visou desenvolver uma aplicação *web* capaz de compilar e executar subconjuntos de código da linguagem de programação C. Para isso, foi necessário o desenvolvimento do compilador, contemplando as três análises sequenciais de um compilador, geração de código de três endereços como código intermediário, no qual serve como *bytecode* e por fim o desenvolvimento de uma máquina virtual de registradores para execução desse código intermediário.

1.4 ESTRUTURA DO TRABALHO

A estrutura deste Trabalho de Conclusão de Curso (TCC) da Área da Informática da UCS inicia-se por uma introdução que apresenta previamente o escopo do trabalho realizado conteplando o problema de pesquisa, questão de pesquisa, os objetivos do trabalho, desenvolvimento do compilador *web* e máquina virtual, testes e validações realizadas e por fim a conclusão.

Na sequência, o presente trabalho está organizado da seguinte forma:

- O Capítulo 2 apresenta a pesquisa referencial teórica sobre compiladores, geração de código intermediário de três endereços, máquinas virtuais e ferramentas *frontend* para utilização na *web*.
- O Capítulo 3 apresenta três trabalhos relacionados a estes assuntos que ajudarão na construção da proposta de solução.
- O Capítulo 4 apresenta a proposta de solução deste trabalho através do desenvolvimento de uma aplicação capaz de compilar e executar subconjuntos de código da linguagem de programação C inteiramente na *web*.
- O Capítulo 5 apresenta o desenvolvimento de todas as etapas do compilador, geração de código intermediário de três endereços e o desenvolvimento da máquina virtual baseada em registradores.
- O Capítulo 6 apresenta os casos realizados para validação do uso do compilador.
- O Capítulo 7 apresenta a conclusão, onde são discutidos os principais resultados obtidos e as sugestões para futuros aprimoramentos e pesquisas no trabalho.

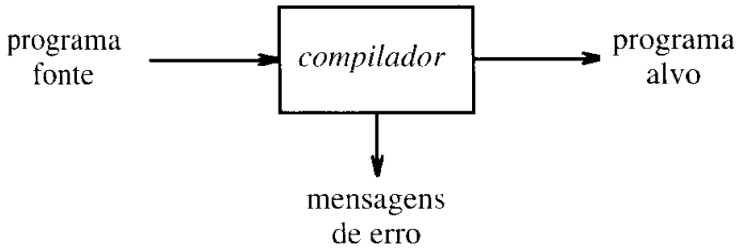
2 REFERENCIAL TEÓRICO

Este capítulo tem por objetivo apresentar as pesquisas técnicas realizadas em cima das áreas de estado pertencentes a monografia deste TCC. Para desenvolver os conteúdos apresentados a seguir, foram consultadas referências provenientes de trabalhos acadêmicos, artigos e livros relacionadas à Ciência da Computação.

2.1 COMPILADORES

Um compilador é uma aplicação que examina um código-fonte escrito em uma linguagem específica, denominada linguagem fonte, e o traduz em um programa equivalente em outra linguagem, denominada linguagem alvo. Durante este processo de tradução, é importante o compilador identificar a presença de erros no programa fonte e comunicá-los ao usuário (AHO; SETHI; ULLMAN, 1995). Na Figura 1, o fluxo de um compilador é apresentado. De forma simples, para algo complexo que é um compilador, Aho e Ulman, explicam o funcionamento de um compilador. Com base no livro Compiladores: Princípios, Técnicas e Ferramentas, conhecida também por "Livro do Dragão", o conteúdo desta seção será apresentado.

Figura 1 – Compilador

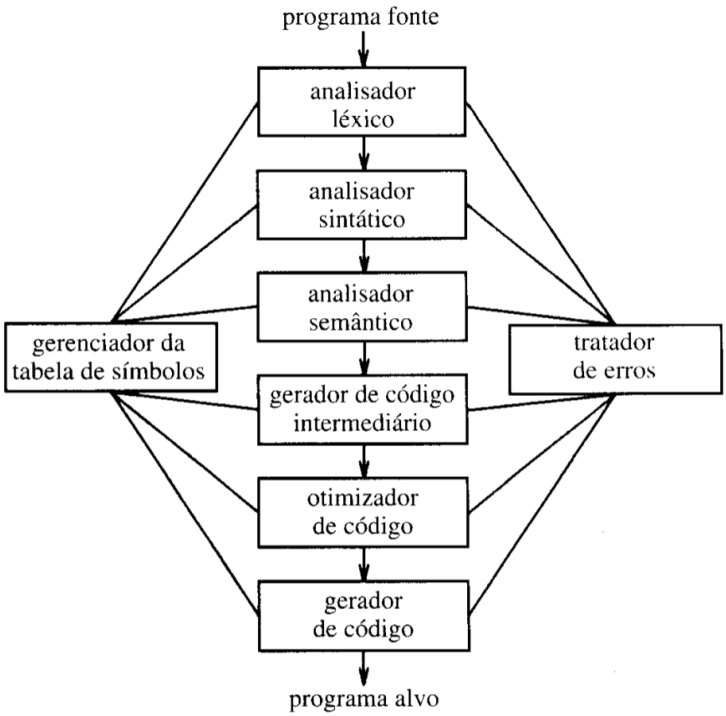


Fonte: Adaptado de (AHO; SETHI; ULLMAN, 1995).

De um ponto de vista conceitual, um compilador opera por meio de fases distintas. Cada fase tem suas próprias características e funções, nas quais transforma o código-fonte de uma representação para outra, verificando se o programa sendo processado está em conformidade com a linguagem gramatical definida. Conforme apresentado na Figura 2, o compilador detém de 6 fases sequenciais e mais duas que operam durante cada uma das seis. Para este trabalho, todas as fases foram estudadas e aplicadas exceto a otimização de código.

Na primeira etapa, o analisador léxico tem a responsabilidade de ler os caracteres presentes no código-fonte da linguagem de origem e produzir uma sequência de *tokens* que o *parser* usará para realizar a análise sintática. De maneira simplificada, o analisador léxico examina os caracteres no texto-fonte inserido pelo usuário até identificar um *token* que esteja de acordo

Figura 2 – Fases do compilador



Fonte: Adaptado de (AHO; SETHI; ULLMAN, 1995).

com a gramática da linguagem definida, ao encontrá-lo a verificação deste *token* é válida, caso contrário a análise léxica emitirá um erro léxico no programa.

Com a identificação do *token* realizada, a análise sintática pode ser executada no compilador. O analisador sintático verifica todas as sequências de *tokens* gerados no código-fonte digitado e realiza a validação dos mesmos através da sintaxe definida na gramática da linguagem fonte. É importante o relato de erros ao usuário durante a execução do analisador sintático caso os *tokens* não estejam em conformidade com a sintaxe da gramática.

Partindo para a terceira fase, a análise semântica utiliza a mesma estrutura hierárquica desenvolvida para o analisador sintático. Neste ponto, o analisador semântico tem o importante objetivo de realizar a verificação dos tipos, checando se para cada operando são atribuídos operadores correspondentes conforme as tipagens declaradas no código-fonte digitado, caso contrário erros semânticos deverão ser sinalizados ao usuário. Conversões de tipos são realizadas durante o processo de operação, em situações na qual um inteiro é atribuído a um *float*, então este inteiro necessitará de uma conversão para *float* previamente antes de ser atribuído.

As fases de análise do código-fonte finalizam após a análise semântica. Neste ponto alguns compiladores iniciam-se pela a geração de código intermediário, na sequência a otimização de código e por fim a geração do código alvo finalizando a estrutura completa de um compilador.

A geração de um código intermediário pode ser representada como um programa gerado

para uma máquina abstrata, devendo ser fácil a sua produção e tradução ao programa alvo. Esta intermediação possibilita uma variedade de formas, sendo destacada neste trabalho as duas que serão citadas, a árvore sintática utilizada no WebAlgo atual e o C3E que foi explorado em busca de uma melhora significativa na forma que hoje é implementado o código intermediário.

A otimização de código busca melhorar e simplificar o código intermediário gerado na fase anterior para que resulte em um código alvo mais performático, diminuindo seu tempo de execução pela máquina. É importante ressaltar essa fase, porém ela não será aplicada neste trabalho.

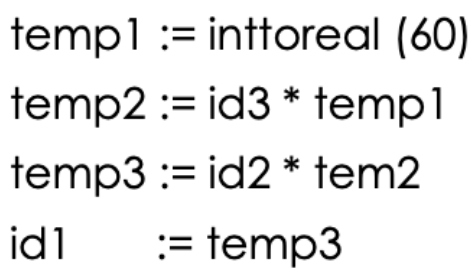
Concluindo as etapas de um compilador, a geração do código-alvo ou código de montagem traduz o código intermediário gerado em uma sequência de instruções de máquina, a ser finalmente executada por uma máquina virtual.

2.1.1 Geração de código intermediário: código de três endereços

O código de três endereços é uma forma de representação de código intermediário amplamente utilizada como uma linguagem de montagem para máquinas. O C3E consiste em uma sequência de instruções, cada uma contendo no máximo três operandos, conforme demonstrado na Figura 3. Cada instrução segue o formato, $x = y\ op\ z$ onde "x"é o identificador de atribuição, "y"e "z"são operandos, podendo conter no máximo um operador aritmético de ponto fixo ou flutuante "op", excluindo o operador de atribuição, ou, no caso de dados booleanos, um operador lógico. Outros enunciados de três endereços comumentes utilizados segundo o livro do dragão são:

- $x := op\ y \rightarrow$ onde "op"representa uma operação unária, incluindo o operador de negação, negação lógica, operadores de deslocamento e conversão (como a conversão de ponto fixo para flutuante).
- $x := y \rightarrow$ "y"é atribuído a "x".
- $goto\ L \rightarrow$ utilizado para desvios/saltos incondicionais. Neste caso o rótulo L será o próximo a ser executado.
- $if\ x\ relop\ y\ goto\ L \rightarrow$ utilizado para desvios condicionais em que se aplica um operador relacional ("relop") em "x"sobre "y"em que se a condição for verdadeira o rótulo a ser chamado é "L"e um salto será realizado, caso contrário continua-se a execução sequencial. Utilizado em laços de repetições.

Figura 3 – C3E exemplo código



Fonte: Adaptado de (AHO; SETHI; ULLMAN, 1995).

A seguir, na Seção 2.1.1.1 e Seção 2.1.1.2 serão demonstrados exemplos práticos do funcionamento do código de três endereços para estruturas de código da linguagem.

2.1.1.1 Código de três endereços - condicionais

Comandos: If, else if e else

Descrição: Os comandos condicionais no código de três endereços geram desvios condicionais e incondicionais nas intruções.

Exemplo Algoritmo 1: *if (g > 10) a = b + c else a = b + d;*

Algoritmo 1 – Exemplo de instrução condicional gerada para código de três endereços

```
1
2  # C3E Gerado
3          t1 = g > 10
4          if t1 goto LB1
5          t3 = b + d
6          a = t3
7          goto FIM
8  LB1 :   t2 = b + c
9          a  = t2
10 FIM :
```

Fonte: O Autor, 2024

No exemplo Algoritmo 1, são utilizados dois desvios, o primeiro ocorre quando o endereço t1 é avaliado como Falso, nesse caso, a execução do programa segue para o rótulo ELSE, caso contrário, a linha subsequente atribui t2 ao endereço final e, em seguida, ocorre um desvio incondicional para o rótulo FIM.

Os rótulos representam pontos específicos no código de três endereços, onde é possível efetuar saltos, sejam eles condicionais ou incondicionais, direcionados a esses pontos.

2.1.1.2 Código de três endereços - repetição

Comandos: while, for e do while

Descrição: Os comandos de repetição no código de três endereços geram desvios condicionais e incodicionais nas intruções.

Exemplo *While* Algoritmo 2: *while (i<10) if (i==5) break else a++*

Exemplo *For* Algoritmo 3: *for (i=0; i<10; i++) if (i==5) break else a += 1*

Exemplo *Do while* Algoritmo 4: *do if (i==5) break else a += 1; i++ while (i<10);*

Algoritmo 2 – Exemplo *while* de instrução de repetição gerada para código de três endereços

```
1  # C3E Gerado
2          t2 = 0
3          i  = t2
4  LACO:   t3 = i < 10
5          if t1 goto LB1
6          goto FIM
7  LB1 :   t4 = i < 5
8          if t4 goto LB2        // Verifica i == 5
9          goto FIM
10 LB2 :   t6 = a + 1
11         a = t6
12         i = i + 1             // Incrementa valor de i do laco
13         goto LACO
14 FIM :
```

Fonte: O Autor, 2024

Algoritmo 3 – Exemplo *for* de instrução de repetição gerada para código de três endereços

```
1  # C3E Gerado
2          t2 = 0
3          i  = t2
4  LACO:   t3 = i < 10
5          if t3 goto LB1        // Verificacao do laco i < 10
6          goto FIM
7  LB1 :   t4 = i < 5
8          if t4 goto LB2        // Verifica i == 5
9          goto LB3              // Chamada do break
10 LB2 :   t6 = a + 1
11         a  = t6
12 LB3 :   i = i + 1             // Incrementa valor de i do laco
13         goto LACO
14 FIM :
```

Fonte: O Autor, 2024

Algoritmo 4 – Exemplo *do while* de instrução de repetição gerada para código de três endereços

```
1  # C3E Gerado
2          t2 = 0
3          i  = t2
4  LACO:   t3 = i < 5
5          if t3 goto LB2        // Verifica i == 5
6          goto LB3              // Chamada do break
7  LB2 :   t4 = a + 1
8          a = t4
9          i = i + 1             // Incrementa valor de i do laco
10         t5 = i < 10
11         if t5 goto LACO
12 FIM :
```

Fonte: O Autor, 2024

2.2 MÁQUINAS VIRTUAIS

Embora não seja uma tecnologia nova, tendo sido desenvolvida na década de 60, a utilização de máquinas virtuais (VMs) está sendo cada vez mais explorada, devido aos avanços tecnológicos. Elas são usadas em servidores e aplicações com diversos objetivos, como garantir segurança, compatibilidade com sistemas legados e consolidar servidores. Uma máquina virtual pode ser definida como um ambiente que suporta a execução de software e pode ser classificada em três categorias: máquinas virtuais de processos, de sistemas operacionais e de sistemas (MAZIERO, 2008). No entanto, neste trabalho, não será necessário abordar nem implementar máquinas virtuais de sistemas operacionais ou sistemas. O foco será nas máquinas virtuais de processos.

As máquinas virtuais de processos, também conhecidas como máquinas virtuais de aplicação, são ambientes criados exclusivamente durante a execução de um processo de software e são destruídos após a finalização desse processo. Portanto, a cada execução, uma nova instância é criada. Esse tipo de Máquina Virtual (VM) é utilizado por desenvolvedores de linguagens de programação que precisam tornar seus programas portáteis para que possam ser compilados ou interpretados em várias arquiteturas (MAZIERO, 2008).

A Máquina Virtual Java (JVM) exemplifica os benefícios da virtualização, pois permite que os programas escritos em Java sejam executados em uma ampla gama de plataformas. Quando um código-fonte Java é compilado, ele gera um arquivo binário conhecido como *bytecode*. Esse *bytecode* é então lido e interpretado pela JVM, resultando na execução de instruções específicas para a arquitetura real da máquina. Essa arquitetura facilita a portabilidade da linguagem, pois para migrá-la para uma arquitetura diferente, basta adaptar a JVM para gerar as instruções apropriadas para essa nova máquina. Não apenas o Java, mas muitas outras linguagens também adotaram essa abordagem de desenvolver seu próprio *bytecode* para ser executado

em suas respectivas máquinas virtuais, como é o caso do C e do Python (MAZIERO, 2008).

Ao criar uma máquina virtual, há duas opções de implementação disponíveis: máquinas virtuais baseadas em pilha e máquinas virtuais baseadas em registradores. Normalmente, as máquinas virtuais baseadas em pilha são escolhidas devido à sua simplicidade de implementação. Estas executam operações lógicas e aritméticas usando uma estrutura de pilha, manipulando os valores por meio de operações de empilhar (*push*) e desempilhar (*pop*). Exemplos de máquinas virtuais bem conhecidas que utilizam essa abordagem incluem a JVM do Java, o *Common Language Runtime* (CLR) do .NET e a CPython da linguagem Python (MIOTTO, 2019). Outra máquina virtual baseada em pilha é o WebAssembly, que merece destaque na pesquisa e análise, dada a sua relevância como uma máquina virtual que pode ser executada diretamente nos principais navegadores da web, ela será apresentada na Seção 2.2.1.

Já a máquina de registradores têm a sua procura aumentada no decorrer dos anos devido a alguns pontos de performance, no qual ela se destaca em comparação a máquina de pilha. Estudos, como o realizado nos trabalhos de Gregg (GREGG *et al.*, 2005) e Shi (SHI *et al.*, 2008), detalharam essa comparação, evidenciando que a máquina de registradores gera menos instruções a partir do código-fonte em comparação à máquina de pilha, como pode ser observado nos exemplos de trechos de código Algoritmo 5 e Algoritmo 6. Essa melhoria de desempenho e a redução no número de instruções geradas foram alcançadas por meio da transcrição da máquina virtual baseada em pilha da JVM em uma máquina virtual de registradores mais simples.

Algoritmo 5 – Exemplo de código em máquina virtual de pilha

```
1 iload c  //carrega c na pilha
2 iload b  //carrega b na pilha
3 iadd     //soma c e b (c+b)
4 istore a // armazena resultado do topo da pilha em a
```

Fonte: (GREGG *et al.*, 2005)

Algoritmo 6 – Exemplo de código em máquina virtual de registradores

```
1 iadd a, b, c     //soma c e b (c+b) e armazena resultado no em a
```

Fonte: (GREGG *et al.*, 2005)

2.2.1 WebAssembly

O WebAssembly[1], também conhecido como Wasm, representa uma nova linguagem de programação de baixo nível. O formato .wasm consiste em instruções binárias destinadas a uma máquina virtual baseada em pilha, visando a execução em alta velocidade. O WebAssembly foi concebido com a finalidade específica de viabilizar a portabilidade de aplicações desenvolvidas

[1] <https://webassembly.org/>

em linguagens como C, C++ e Rust para a web. Ferramentas como o Emscripten[2] desempenham a função de transpilar o código C ou C++ para o formato .wasm, tornando possível a execução desses programas em navegadores. À medida que o tempo passa, novas linguagens de programação são gradualmente incorporadas a essas ferramentas para facilitar a migração do código original para o formato .wasm.

Grandes empresas do mercado, incluindo Google, Mozilla, Microsoft e Apple, incorporaram a máquina virtual do WebAssembly aos seus navegadores (WATT, 2018). Para executar os códigos .wasm, é essencial usar JavaScript para chamar os arquivos, devido à impossibilidade temporária de importação diretamente através do *Hypertext Markup Language* (HTML) da página, de acordo com a documentação do WebAssembly.

O arquivo .wasm, por ser binário, pode ser desafiador de ler durante processos de depuração ou análise de código. Portanto, os navegadores têm a capacidade de transformá-los em um formato de texto, conhecido como .wat, que torna a leitura e compreensão do código mais acessível. Esse arquivo .wat é baseado em uma representação de Expressão-S. A Figura 4 ilustra a sintaxe de uma Expressão-S usada para gerar o arquivo .wat do WebAssembly. Como exemplo, retirado da documentação oficial do WebAssembly, o código gerado define o método "*getAnswerPlus1*", que pode ser invocado pelo nome correspondente no código JavaScript da aplicação. Esse método chama outro procedimento chamado "*$getAnswer*", que retorna o valor de uma constante e, em seguida, soma-o a outra constante antes de retornar o valor resultante.

Figura 4 – Exemplo arquivo .wat

```
WASM

(module
  (func $getAnswer (result i32)
    i32.const 42)
  (func (export "getAnswerPlus1") (result i32)
    call $getAnswer
    i32.const 1
    i32.add))
```

Fonte: Adaptado da Documentação Oficial do WebAssembly

A investigação sobre o WebAssembly proporcionou *insights* valiosos para o trabalho, enquanto procurava-se uma ferramenta que pudesse simplificar o desenvolvimento do interpretador. No entanto, foi percebido que o uso do WebAssembly poderia restringir o compilador *web* a ser desenvolvido, caso houvesse uma expansão, e também, que neste caso haveria um passo a mais já que necessitaria uma conversão do código de três endereços para o *bytecode* do WebAssembly. Devido a essa consideração, a opção preferencial continuou sendo o desenvolvimento de uma máquina virtual.

[2] <https://emscripten.org>

2.3 TECNOLOGIAS PARA DESENVOLVIMENTO WEB

O desenvolvimento *web* é composto por várias etapas além da codificação da aplicação. Inicialmente, o sistema a ser desenvolvido é estruturado seguindo uma metodologia da engenharia de software. As definições das linguagens de programação não são enigmáticas no desenvolvimento do *frontend*. HTML, *Cascading Style Sheets* (CSS) e JavaScript são os principais componentes para a execução de uma aplicação em um navegador *web*.

Atualmente, nenhum *web designer* desenvolve uma aplicação *web* escrevendo uma imensidão de linhas de código HTML, CSS e Javascript. É recomendado utilizar as estruturas e bibliotecas desenvolvidas por empresas especializadas, e, até mesmo pela comunidade para agilizar o processo de desenvolvimento e também pela confiabilidade no funcionamento dos elementos utilizados, evitando assim manutenções desnecessárias (ALVES, 2015).

2.3.1 Javascript

JavaScript[3], comumente abreviado como JS, é a linguagem de programação predominante usada nos navegadores modernos para criar interações dinâmicas nas estruturas HTML das páginas da *web* (FLANAGAN; (FIRM); SAFARI, 2020). Qualquer alteração feita na tela do navegador após o carregamento da página requer a incorporação de *scripts* em JavaScript para torná-la interativa. Além de possibilitar modificações no *frontend*, o JavaScript permite comunicações assíncronas entre o servidor e o navegador, um conceito conhecido como *Asynchronous JavaScript And XML* (AJAX).

O Javascript é um elemento essencial para este trabalho já que toda a lógica de programação foi construída utilizando-o para possibilitar a compilação no navegador.

2.3.2 Bibliotecas para editores de texto

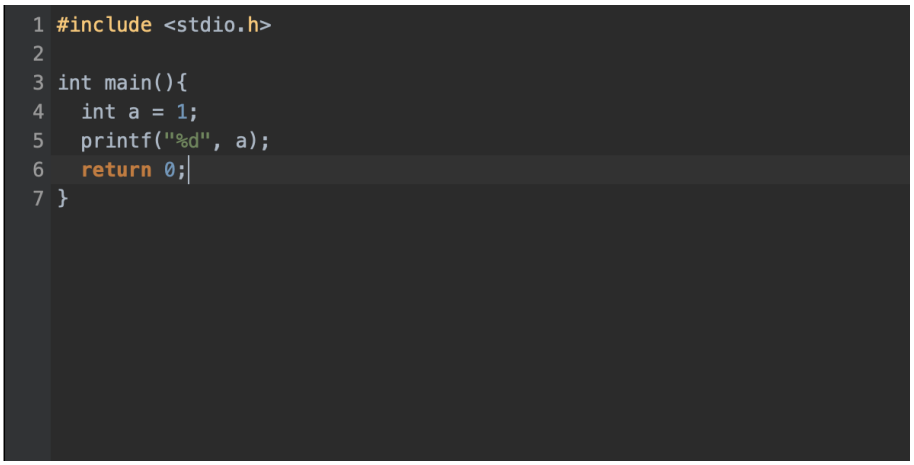
O CodeMirror[4] é um componente *web* de código aberto que oferece funcionalidades de um editor de texto para várias linguagens de programação. Esta biblioteca é capaz de atuar como um analisador completo para diversas linguagens, incluindo C, C++[5], Python, Java e outras linguagens oficiais, além de versões estendidas criadas pela comunidade, que incluem recursos adicionais e suporte para outras linguagens. Para utilizar o CodeMirror, é necessário importar seus arquivos JavaScript e CSS no modelo HTML da aplicação.

O CodeMirror é uma biblioteca robusta para edição de texto. Além dos analisadores léxicos e sintáticos integrados para verificar o código-fonte inserido pelo usuário, a biblioteca oferece recursos adicionais, como a exibição do número das linhas, a possibilidade de imple-

[3] <https://www.javascript.com/>
[4] <https://codemirror.net>
[5] <https://cplusplus.com/>

mentação de um depurador, a funcionalidade de autocompletar e a capacidade de personalizar o tema do editor, como mostrado na Figura 5.

Figura 5 – Exemplo CodeMirror



Fonte: O Autor, 2024

O ACE[6] segue a mesma linha de componentes mencionada anteriormente, semelhante ao CodeMirror. Essa extensão JavaScript oferece um editor de alta performance que pode ser usado na *web*. Além de incluir as funcionalidades do CodeMirror, o ACE se diferencia pelo amplo suporte a linguagens de programação em seu analisador léxico e *parser*.

Conforme mencionado anteriormente, a incorporação de componentes e bibliotecas criados, tanto por empresas quanto pela comunidade de desenvolvedores, desempenhará um papel crucial neste projeto, auxiliando no processo de desenvolvimento. Um exemplo é a escolha de um dos editores mencionados para a implementação do ambiente de edição de texto, o qual permitirá aos usuários escrever seu código em linguagem C.

[6] <https://ace.c9.io>

3 TRABALHOS RELACIONADOS

O capítulo sobre trabalhos relacionados abrange a base teórica que auxilia na construção deste trabalho, utilizando pesquisas prévias já realizadas. A análise dos trabalhos relacionados permite entender os desafios enfrentados pelos autores durante o desenvolvimento de suas pesquisas, bem como das ferramentas necessárias para alcançar os objetivos desejados.

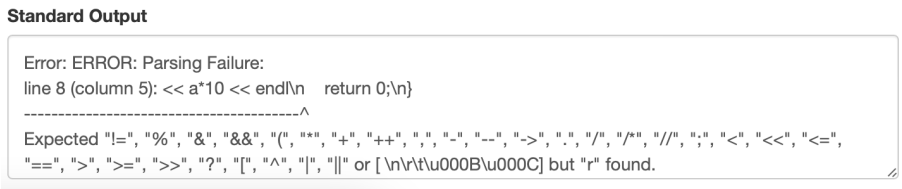
O foco dos trabalhos apresentados a seguir foi fundamental para o presente trabalho para entender quais foram seus objetivos, suas dificuldades e as ferramentas selecionadas para auxiliar no desenvolvimento da pesquisa. Os trabalhos apresentados têm como foco os assuntos do presente trabalho.

3.1 UM SIMPLES INTERPRETADOR DE C++ DESENVOLVIDO EM JAVASCRIPT

O projeto de HAO (HAO, 2021) teve como objetivo principal o desenvolvimento de um interpretador de C++ que pudesse ser usado diretamente em navegadores modernos, sem a necessidade de se comunicar com um servidor para compilar e executar o código inserido pelo programador. Este projeto, chamado JSCPP e disponível no GitHub[1], foi concebido principalmente para fins educacionais e pode ser utilizado como uma biblioteca JavaScript.

Por ser um interpretador, ele não gera código para uma máquina virtual, no entanto, é uma biblioteca completa que abrange todas as fases de um compilador, incluindo análise léxica, sintática e semântica. Isso significa que ele é capaz de identificar erros de compilação caso o código não esteja em conformidade com a gramática da linguagem. A Figura mostra um exemplo de erro sintático, onde falta um ponto e vírgula no final de uma linha. As imagens utilizadas foram retiradas de um emulador fornecido pelo autor no GitHub.io para testar o interpretador.

Figura 6 – Exemplo erro de compilação



Fonte: Imagem retirada do GitHub.Io e adaptada pelo Autor, 2024

Embora a documentação disponível não detalhe completamente o processo de desenvolvimento do interpretador, é possível entender a sua estrutura e funcionalidades por meio

[1] <https://github.com/felixhao28/JSCPP>

do material fornecido pelo autor e pela análise do código-fonte. Algumas das funcionalidades implementadas incluem operadores, variáveis, vetores, ponteiros, estruturas condicionais, estruturas de repetição, *switch*, *case*, funções e outras. Além disso, o interpretador permite a depuração do código inserido. Ele utiliza uma Árvore Sintática Abstrata (AST) em sua estrutura, o que possibilita a inserção de pontos de parada para análise do código durante a execução.

Além das funcionalidades essenciais que um compilador deve incluir, o JSCPP permite que os desenvolvedores que utilizam essa ferramenta configurem várias opções de execução. Isso inclui a definição de limites para tipos de variáveis, a permissão, aviso, ou proibição de erros de *unsigned overflow*, a definição do tempo máximo de execução do código durante a compilação e outras personalizações.

3.2 *A FRIENDLY ONLINE C COMPILER TO IMPROVE PROGRAMMING SKILLS BASED ON STUDENT SELF-ASSESSMENT*

O segundo trabalho apresentado difere significativamente do projeto anterior, embora ambos tenham suas particularidades. Enquanto o trabalho anterior se concentrava em um interpretador, este segundo trabalho trata-se de um compilador, o que representa uma mudança em relação ao objetivo do presente trabalho. No entanto, é importante destacar as tecnologias utilizadas pelos autores e uma possível abordagem de desenvolvimento na qual a compilação final ocorre no servidor.

O trabalho realizado por Cedazo e Cena Cedazo, Cena e Al-Hadithi (2015) teve como foco a criação de um ambiente de aprendizado voltado para estudantes de programação, permitindo o desenvolvimento de exercícios de programação online, especialmente na linguagem C. Semelhante ao ambiente educacional da UCS, a solução proposta pelos autores permite que os alunos escrevam, editem, compilem e executem programas em linguagem C. A aplicação é dividida em duas visões: uma destinada aos alunos, que podem desenvolver os exercícios propostos, e outra para os professores, que podem acompanhar o desempenho dos estudantes.

Devido à aplicação ser fundamentada para *web*, foram selecionadas tecnologias específicas. O JavaScript foi escolhido para desenvolver as primeiras etapas do compilador, enquanto a biblioteca/ferramenta CodeMirror foi utilizada para criar o editor de texto e também o *framework* ExtJS que permite um melhor desenvolvimento devido à lógica utilizada para organização de código Javascript. Para gerenciar as comunicações entre o cliente e o servidor, os autores optaram pelo *framework* ExpressJS[2] e a compilação do código C foi realizada no servidor utilizando o compilador GCC[3].

É importante notar que, neste trabalho, os autores não se concentraram no desenvol-

[2] ExpressJS: *Framework* de Node.JS para possibilitar o gerenciamento de *requests*, *responses* e controles de sessão da aplicação.

[3] **GNU Compiler Collection** (GCC): Conjunto de compiladores de linguagens de programação como C e C++.

vimento completo do compilador. O texto enfatiza que a interface do usuário (*frontend*) foi projetada apenas para a edição do código-fonte, e após a conclusão, o código foi enviado ao servidor para compilação direta utilizando o GCC. Essa abordagem foi mencionada como uma alternativa ao interpretador, mas o objetivo do presente trabalho permaneceu na realização da compilação através do *frontend*.

3.3 *CHASM* - UM SIMPLES COMPILADOR PARA LINGUAGEM WEBASSEMBLY

O trabalho de Eberhardt Eberhardt (2019), embora consista em um compilador simples para uma linguagem por ele criada, compartilha de maneira semelhante o propósito do presente trabalho, que é compilar e executar código-fonte diretamente no navegador. O seu objetivo era desenvolver um simples compilador e gerar código Wasm para máquina virutal do WebAssembly, permitindo assim a execução direta no navegador.

Para demonstrar essa funcionalidade, Eberhardt desenvolveu todas as etapas de um compilador, começando pela análise léxica, passando pela análise sintática e semântica. A estrutura do seu compilador gera uma AST, que é posteriormente percorrida por uma função responsável pela geração de código binário com base nos *opcodes* do WebAssembly. A geração do código binário ocorria através da empilhamento das instruções binárias correspondentes aos *opcodes* do WebAssembly, resultando em uma pilha que continha as instruções binárias do programa, prontas para serem executadas pela máquina virtual Wasm no navegador.

Uma demonstração dessa funcionalidade está disponível no repositório do projeto no GitHub, permitindo a simulação das compilações e a visualização da manipulação de arquivos com extensão ".wat"que são instanciados durante a execução do programa no armazenamento do navegador.

Embora o foco principal da apresentação desse trabalho seja a geração de código WebAssembly e sua execução dinâmica no navegador, ao analisar os códigos-fonte do projeto, é possível observar que o autor utilizou a biblioteca JavaScript CodeMirror para implementar seu editor de texto.

4 PROPOSTA DE SOLUÇÃO

Este capítulo tem como objetivo principal apresentar a proposta de solução deste trabalho. Com base no objetivo central, que consistiu no desenvolvimento de um compilador *web* para a linguagem C, utilizando o C3E como código intermediário e realizando todo o processo de compilação diretamente no navegador. Essa proposta foi construída com base na fundamentação teórica apresentada no Capítulo 2 e nos estudos relacionados abordados no Capítulo 3.

Um dos maiores desafios encontrados na organização da estrutura do compilador foi, sem dúvida, a definição da máquina virtual. A pesquisa realizada sobre as possíveis ferramentas a serem empregadas, juntamente com a análise dos trabalhos relacionados e as pesquisas sobre máquinas virtuais baseadas em pilha e registradores, desempenhou um papel fundamental na tomada de decisão para estruturar essa parte do compilador.

Os temas a serem abordados a seguir têm como foco a construção do compilador e do interpretador, nos quais uma máquina virtual foi empregada para interpretar o código de três endereços. Neste capítulo, serão abordadas as seguintes etapas: a definição dos subconjuntos de C presentes no WebAlgo atual, juntamente com o desenvolvimento da gramática da linguagem C, que foi utilizada para validar a análise sintática do código. É discutida a geração de código intermediário por meio do código de três endereços, detalhando todas as instruções utilizadas neste trabalho. É explorado, o funcionamento da tabela de símbolos, no qual valida o uso correto de variáveis e a conversão apropriada de tipos de variáveis. A definição da máquina virtual e o mapeamento de todas as instruções de C3E para a máquina possibilitando assim, a interpretação eficiente desse código intermediário (*bytecode*).

No contexto das definições de engenharia de software, são apresentados dois casos de uso do compilador, junto com as tecnologias e ferramentas utilizadas no desenvolvimento do *frontend* da aplicação, compilador e máquina virtual. Concluindo, é abordado a última funcionalidade necessária para a implementação do compilador do WebAlgo neste trabalho que é o funcionamento do depurador do compilador.

4.1 SUBCONJUNTOS DE C

A linguagem C oferece uma ampla gama de recursos em sua gramática. No entanto, de acordo com os objetivos do presente trabalho, que se concentra no desenvolvimento de um compilador e depurador a ser executado na *web* para para uma possível migração do WebAlgo para *web*, os subconjuntos da linguagem C incluídos no compilador *web* devem corresponder exatamente aos subconjuntos presentes no WebAlgo.

Para identificar as estruturas presentes no WebAlgo, foi realizada uma consulta ao pro-

fessor e orientador deste Trabalho de Conclusão de Curso, Ricardo Vargas Dorneles, que forneceu orientações sobre os tópicos a serem implementados. Esses tópicos incluem:

- Variáveis: int, *float*, *double* e char;
- Matrizes;
- Vetores;
- Operadores aritméticos: adição, subtração, multiplicação, divisão, módulo (+, -, *, /, %);
- Operadores de atribuição: atribuição, atribuição por adição, atribuição por subtração, atribuição por multiplicação, atribuição por divisão e atribuição por módulo (=, +=, -=, *=, /=, %=);
- Operadores pós-fixos (++ e –);
- Operadores pré-fixos/unários (++, –. +, - e !);
- Operadores lógicos e relacionais: "igual a", "diferente de", E lógico, OU lógico, "maior que", "menor que", "maior igual que"e "menor igual que"(==, !=, &&, ||, >, <, >= e <=);
- Estruturas de condição (*if* com e sem *else*);
- Estrturas de repetição (*while*, *do while* e *for*);
- Desvio incondicional (interrupções) (*break*, *continue*);
- Chamadas de função;
- Importação de biblioteca (<*stdio.h*>);
- Entrada dados (*scanf*);
- Saída dados (*printf*).

A inclusão da biblioteca "*stdio.h*"abrangerá as chamadas para entrada (*scanf*) e saída de dados (*printf*), uma vez que esse é o método utilizado pela versão desktop do WebAlgo. É importante destacar que a adição de novas funcionalidades poderá ser realizada em futuros projetos, uma vez que a substituição do código intermediário de uma árvore sintática (WebAlgo) para C3E (compilador *web*) simplifica a expansão do compilador.

4.2 GRAMÁTICA DOS SUBCONJUNTOS DE C

A definição da gramática de uma linguagem é fundamental para a implementação da etapa de análise sintática do compilador. A partir dos *tokens* gerados pela gramática, é possível realizar uma análise para verificar se eles estão em conformidade com a estrutura da linguagem. A implementação da gramática da linguagem C no compilador *web* adota uma abordagem de gramática descendente recursiva ou LL(1), na qual o analisador (*parser*) examina a entrada da direita para a esquerda, analisando um *token* por vez e determinando a produção apropriada a ser utilizada (AHO; SETHI; ULLMAN, 1995). A gramática implementada será apresentada a seguir. Para a construção das produções, foi utilizado como referência o ANSI C Grammar.[1].

- *programa*: **listaDec2**
- *listaDec2*: **dec2 listaDec2**
- *dec2*: **declaracao | decFunc | decLibDefine**
- *decLibDefine*: DEFINE ID CONSTANTE | INCLUDE STDIO.H | INCLUDE MATH.H
- *decFunc*: **tipo nomeFunc** ’(’ **listaParam** ’)’ **corpoFunc**
- *corpoFunc*: ’’ ’’ | ’’ **listaInstr** ’’
- *listaDec*: **declaracao listaDecRestante**
- *listaDecRestante*: **declaracao listaDecRestante** | E
- *declaracao*: **tipo listaDecInicial** ’;’
- *listaDecInicial*: **decInicial listaDecInicialRestante**
- *listaDecInicialRestante*: ’,’ **decInicial listaDecInicialRestante** | E
- *decInicial*: **dec** ’=’ **expressAtrib | dec**
- *dec*: ID **decRestante**
- *decRestante*: ’[’ **expressCondic** ’]’ | E
- *listaInstr*: **instr listaInstrRestante**
- *listaInstrRestante*: **instr listaInstrRestante**
- *instr*: **instrCondicional | instrExpress | instrIteracao | instrSalto | listaDec | instrEscrita | instrLeitura | corpoFunc**
- *instrEscrita*: PRINTF ’(’ STRINGSTDIO **expressaoRestantePrintf** ’)’ ’;’

[1] <https://www.lysator.liu.se/c/ANSI-C-grammar-y.html>

- *instrLeitura*: SCANF '(' STRINGSTDIO **leituraRestante** ')' ';'
- *leituraRestante*: ',' '&' **expressPos leituraRestante** | E
- *instrSalto*: CONTINUE ';' | BREAK ';' | RETURN ';' | RETURN **expressao** ';'
- *instrIteracao*: WHILE '(' **expressao** ')' **instr** | DO **instr** WHILE '(' **expressao** ')' ';' | FOR '(' **instrExpress instrExpress expressao** ')' **instr**
- *instrCondicional*: IF '(' **expressao** ')' **instr** | IF '(' **expressao** ')' **instr** ELSE **instr**
- *instrExpress*: ';' | '[' **expressao** ';'
- *expressao*: **expressaoAtrib expressaoRestante**
- *expressaoRestante*: ',' **expressaoAtrib expressaoRestante** | E
- *expressaoRestantePrintf*: ',' **expressCondicPrintf expressaoRestantePrintf** | E
- *expressCondicPrintf*: **expressCondic**
- *expressAtrib*: **expressUnaria operadorAtrib expressAtrib** | **expressCondic**
- *expressCondic*: **expressLogicOr**
- *expressLogicOr*: **expressLogicAnd expressLogicOrRestante**
- *expressLogicOrRestante*: '||' **expressLogicAnd expressLogicOrRestante** | E
- *expressLogicAnd*: **expressIgual expressLogicAndRestante**
- *expressLogicAndRestante*: '' **expressIgual expressLogicAndRestante** | E
- *expressIgual*: **expressRelacional expressIgualRestante**
- *expressIgualRestante*: '==' **expressRelacional expressIgualRestante** | '!=' **expressRelacional expressIgualRestante** | E
- *expressRelacional*: **expressAdd expressRelacionalRestante**
- *expressRelacionalRestante*: '<' **expressAdd expressRelacionalRestante** | '>' **expressAdd expressRelacionalRestante** | '<=' **expressAdd expressRelacionalRestante** | '>=' **expressAdd expressRelacionalRestante** | E
- *expressAdd*: **expressMultipl expressAddRestante**
- *expressAddRestante*: '+' **expressMultipl expressAddRestante** | '-' **expressMultipl expressAddRestante** | '!' **expressMultipl expressAddRestante** | E
- *expressMultipl*: **expressUnaria expressMultiplRestante**

- *expressMultiplRestante*: '/' **expressUnaria expressMultiplRestante** | '*' **expressUnaria expressMultiplRestante** | '%' **expressUnaria expressMultiplRestante** | E
- *expressUnaria*: **expressPos** | '++' **expressUnaria** | '–' **expressUnaria** | **operadorUnario expressUnaria**
- *expressPos*: **expressaoPrima expressaoPosRestante**
- *expressaoPrima*: ID | CONSTANTE | STRING | '(' **expressao** ')'
- *expressaoPosRestante*: '[' **expressao** ']' **expressaoPosRestante** | '(' ')' | '(' **expressao** ')' | '++' **expressaoPosRestante** || '–' **expressaoPosRestante** | E
- *listaParam*: **param listaParamRestante** | E
- *listaParamRestante*: **param listaParamRestante** | E
- *param*: **tipo** ID **decRestante**
- *nomeFunc*: ID
- *tipo*: INT | VOID | FLOAT
- *operadorAtrib*: '=' | '*=' | '/=' | '%=' | '+=' | '-='
- *operadorUnario*: '+' | '-' | '!'

As definições gramaticais são indicadas em expressões regulares, os nomes em itálico à esquerda das produções representam definições e símbolos não-terminais. Por outro lado, os símbolos em negrito à direita das produções também são símbolos não-terminais. Nomes entre apóstrofos denotam símbolos terminais, enquanto nomes em caixa alta representam *tokens* exclusivos utilizados na linguagem C. O caractere *pipe* nas produções separa expressões pertinentes ao mesmo não-terminal.

4.3 ESTRUTURA C3E - CÓDIGO DE TRÊS ENDEREÇOS (*BYTECODE*)

O código de três endereços, previamente apresentado no Capítulo 2, desempenha um papel fundamental neste trabalho, uma vez que a substituição da árvore sintática pelo C3E traga benefícios significativos em termos de manutenção e escalabilidade do compilador em cenários futuros. O *bytecode*, também previamente relatado no mesmo Capítulo é responsável pela interpretação do seu código pela máquina virtual de registradores.

Em uma estrutura de compilador convencional, o código intermediário é responsável por aproximar o programa em processo de compilação o máximo possível de uma linguagem de máquina. Geralmente, após a geração desse código intermediário, ocorre a geração de um

código intermediário de máquina, frequentemente chamado de *bytecode*, que é interpretado pela máquina virtual. No entanto, é importante ressaltar que, neste trabalho, não haverá a geração de um *bytecode* específico para a interpretação pela máquina virtual. O código a ser interpretado pela máquina virtual consistirá na representação linear do código de três endereços (C3E) gerado durante o processo de compilação.

A implementação do código de três endereços abrangerá a geração de uma variedade de instruções, incluindo atribuições, desvios condicionais e incondicionais, estruturas de repetição, bem como expressões que envolvem operadores aritméticos, relacionais, lógicos, vetores e matrizes. Para cada tipo de instrução, foi invocado um método específico responsável por gerar o código de três endereços correspondente. No caso de expressões do tipo ($x = y\ op\ z$), foi necessário utilizar duas funções distintas: uma para o endereço de atribuição (x) e outra para os endereços de operação ($y\ op\ z$), assim gerando uma variável temporária para atribuir o resultado a operação, primeiramente.

É importante ressaltar que neste trabalho não está prevista a implementação de métodos de otimização de código durante a geração do código de três endereços (C3E). Portanto, todas as expressões, inclusive aquelas com dois operandos, resultarão na criação de duas instruções de C3E, como exemplificado no Algoritmo 7. Na sequência, será exemplificada como funciona a geração do C3E para as estruturas de comando do subconjunto da linguagem C definidos. Nas instruções de C3E, T1 representa uma variável temporária.

- Instrução $x = y\ op\ z$. Atribuição com operação aritmética ou lógica, definido como op na instrução. Neste caso op poderá implementar as seguintes operações:
  - Atribuição por adição (x += y):
    T1 = x + y
    x = T1
  - Atribuição por subtração (x -= y):
    T1 = x - y
    x = T1
  - Atribuição por multiplicação (x *= y):
    T1 = x * y
    x = T1
  - Atribuição por divisão (x /= y):
    T1 = x / y
    x = T1
  - Atribuição por módulo (x %= y):
    T1 = x % y
    x = T1

  - Adição ($x = y + z$) implementa:
    T1 = $y + z$
    x = T1
  - Subtração ($x = y - z$) implementa:
    T1 = $y - z$
    x = T1
  - Multiplicação ($x = y * z$) implementa:
    T1 = $y * z$
    x = T1
  - Divisão ($x = y / z$) implementa:
    T1 = y / z
    x = T1
  - Módulo ($x = y \% z$) implementa:
    T1 = $y \% z$
    x = T1
  - E lógico ($x \&\& y$) implementa:
    T1 = $x \&\& y$
  - Ou lógico ($x \mid\mid y$) implementa:
    T1 = $x \mid\mid y$
  - Igual a ($x = y$) implementa:
    T1 = $x = y$
  - Menor que ou menor igual que ($x < y \mid\mid x <= y$) implementa:
    T1 = $x < y$ || T1 = $x <= y$
  - Maior que ou Maior igual que ($x > y \mid\mid x >= y$) implementa:
    T1 = $x > y$ || T1 = $x >= y$

- Instrução unária $x = op\ y$. Neste caso *op* é definido como uma operação unária, podendo implementar as seguintes operações:
  - Incremento pós-fixo ($x{+}{+}$) implementa:
    T1 = $x + 1$
    x = T1
  - Decremento pós-fixo ($x{-}{-}$) implementa:
    T1 = $x - 1$
    x = T1
  - Incremento pré-fixo ($+{+}x$) implementa:
    T1 = $x + 1$
    x = T1

  - Decremento pré-fixo (*–x*) implementa:
    T1 = *x - 1*
    x = T1
  - Operador unário positivo (*+x*) implementa:
    x = *+x*
  - Operador unário negativo (*-x*) implementa:
    x = *-x*
  - Operador unário não lógico (*!x*) implementa:
    x = *!x*

- Atribuição ou cópia implementa *x = y*.
- Desvio condicional implementa *if temporário goto L*. Está instrução implicará na utilização de uma variável temporária no qual executará a operação relacional (=, >, >=, < e <=). Em caso da afirmação ser verdadeira o salto é realizado para o *label*, caso contrário continua a execução sequencial das instruções abaixo da condição.
- Desvio incondicional realiza salto absoluto. Implementa *goto* LB2, onde, sem nenhuma condição, a execução do programa é direcionada para o *label*
- Chamada de função (printf("%d", a)) implementa *param a | goto printf("%d",a)*[2]
- Retorno de função implementa *return* podendo ser opcional.
- Vetores e matrizes representam instruções x = y[i] ou x = y[i][j] ou também na forma no qual o arranjo é do lado esquerdo da atribuição x[i] = y ou x[i][j] = y.
- Os rótulos ou *labels* são utilizados para direcionar os saltos condicionais ou incondicionais durante a execução do código de três endereços das estruturas condicionais (*if*, com e sem *else*), de repetição (*while*, *do while* e *for*) e comandos *break* e *continue*.
- A conversão de tipos surge em situações em que um operando em uma operação aritmética envolve variáveis ou constantes de tipos distintos, como, por exemplo, entre inteiros e *floats*. Nesses casos, é necessário converter o valor inteiro para *float*, conforme a precedência, antes de realizar a operação. Em código de três endereços, uma operação como *x = (int) y + (float) z* seria representada em Código de Três Endereços (C3E) da seguinte maneira:
  T1 = (*inttofloat*) y
  T2 = T1 + z
  x = T2

[2] As chamadas de função tem seus parâmetros instanciados (param <parâmetro>) logo antes da chamada da função (call), em seguida é indicado o número de parâmetros que ela compõe. No caso do exemplo do printf foi passado somente o parâmetro a por isso o número um na instrução call.

As estruturas condicionais e de repetição, no qual geram saltos condicionais e incondicionais foram detalhadas na Seção 4.3.1 e na Seção 4.3.2. Nestes exemplos as "extensões".*place* e *.cod* correspondem as seguintes informações: *.place* é a variável que receberá o valor da instrução e *.cod* recebe a sequência do C3E da instrução. Na Seção 4.3.3

Algoritmo 7 – Exemplo de estrutura de atribuição gerada para código de três endereços

```
1  # Expressao
2  # a = b + c
3
4  # C3E Gerado
5  t1 = b + c
6  a  = t1
```

Fonte: O Autor, 2024

4.3.1 Estrutura condicional - *if*

Comando: IF (E) COMANDO1 ELSE COMANDO 2 (Algoritmo 8)

Descrição: Os comandos condicionais no código de três endereços geram desvios condicionais e incondicionais nas intruções.

A instrução E representa a expressão condicional e a instrução COMANDO representa os código executados dentro dos laços de repetição.

Algoritmo 8 – Exemplo de estrutura condicional genérica gerada para código de três endereços

```
1  # Expressao
2  # IF (E) COMANDO1 ELSE COMANDO 2
3
4  # C3E Gerado
5        E.cod                    // Verifica a condicao (ex: A < B)
6        if E.place goto ELSE     // Verifica se E.place e verdadeiro ou falso
7                                    e realiza salto conforme resultado
8        COMANDO1.cod             // COMANDO1 caso E.place seja verdadeira
9        goto LB1                 // Desvio incondicional para o LB1
10 ELSE: COMANDO2.cod             // COMANDO2 caso E.place seja falso
11 LB1 :
```

Fonte: O Autor, 2024

4.3.2 Estrutura de repetição - *while*, *for* e *do while*

Comando *While*: *WHILE (E) COMANDO* (Algoritmo 9)

Comando *For*: *FOR (E; E; E) COMANDO* (Algoritmo 10)

Comando *Do While*: *DO COMANDO WHILE (E);* (Algoritmo 10)

Descrição: Os comandos de repetição *WHILE, FOR* e *DO WHILE* no código de três endereços geram desvios condicionais e incondicionais nas intruções.

A instrução E representa a expressão condicional e a instrução COMANDO representa os código executados dentro dos laços de repetição.

Algoritmo 9 – Exemplo de estrutura de repetição *while* genérica gerada para código de três endereços

```
1  # Expressao
2  # WHILE (E) COMANDO
3
4  # C3E Gerado
5  LB1:  E.cod                    // Verifica a condicao (ex: A < B)
6        if E.place goto LB2      // Verifica se E.place e verdadeiro ou falso
7                                    e realiza salto conforme resultado
8        COMANDO.cod              // COMANDO caso E.place seja falso
9        goto LB1                 // Desvio incondicional para o LB1
10 LB2:
```

Fonte: O Autor, 2024

Algoritmo 10 – Exemplo de estrutura de repetição *for* genérica gerada para código de três endereços

```
1  # Expressao
2  # FOR (E; E; E) COMANDO
3
4  # C3E Gerado
5  LB1:  E.cod                    // Verifica a condicao
6        if E.place goto LB2      // Verifica se E.place e verdadeiro ou falso
7                                    e realiza salto conforme resultado
8        COMANDO.cod              // COMANDO caso E.place seja falso
9        goto LB1                 // Desvio incondicional para o LB1
10 LB2:
```

Fonte: O Autor, 2024

Algoritmo 11 – Exemplo de estrutura de repetição *do while* genérica gerada para código de três endereços

```
1  # Expressao
2  # DO COMANDO WHILE (E);
3
4  # C3E Gerado
5  LB1:  COMANDO.cod              // COMANDO e executado sempre pelo menos 1 vez
```

```
6       E.cod                       // Verifica a condicao
7       if E.place goto LB2         // Verifica se E.place e verdadeiro ou falso
8                                      e realiza salto conforme resultado
9       goto LB1                    // Desvio incondicional para o LB1
10  LB2:
```

Fonte: O Autor, 2024

4.3.3 Arranjos ou atribuições indexadas (vetores e matrizes)

Comando: *x[i] = y* (Algoritmo 12)

Descrição: Os valores indexados dentro dos vetores serão armazenados em uma lista, na coluna valor da talela de símbolos, no qual será descrita na Seção 4.4, exatamente na mesma posição no qual foi inserida.

A instrução E representa a expressão condicional e a instrução COMANDO representa os código executados dentro dos laços de repetição.

Algoritmo 12 – Exemplo de estrutura de atribuição de vetor genérica gerada para código de três endereços

```
1   # Expressao
2   # FOR (i=0;i<10;i++){
3       x[i] = y;
4   }
5
6   # C3E Gerado
7         i = 0;
8   LB1 : E.cod                       // Verifica a condicao (ex: A < B)
9         if E.place goto ELSE        // Verifica se E.place e verdadeiro ou falso
10                                       e realiza salto conforme resultado
11        x[i] = y (COMANDO1.cod)     // COMANDO1 caso E.place seja verdadeira
12        T1 = i + 1                  // Incrementa I
13        i = T1
14        goto LB1                    // Desvio incondicional para o LB1
15  ELSE:                             // Finaliza caso E.place seja falso
```

Fonte: O Autor, 2024

4.4 CONTROLE DE VARIÁVEIS - TABELA DE SÍMBOLOS

A estrutura de dados da tabela de símbolos no compilador mantém informações essenciais para o funcionamento correto do código a ser executado. Essa tabela tem a função de gerenciar variáveis e funções, atribuindo-lhes escopos específicos durante a execução do programa. Criada durante a fase de análise sintática do compilador, a tabela de símbolos é fundamental para o processo de execução do código na máquina alvo (AHO; SETHI; ULLMAN, 1995).

Para este trabalho, a tabela de símbolos é responsável pelo armazenamento dos identificadores das variáveis e das funções. Junto destes, haverá informações de tipo de variável (inteiro, *float*, *double* ou *char*), valor das mesmas, apontamento para tabela de símbolos aninhada acima, caso contrário valor será nulo. Conforme os subconjuntos de C definidos, apresentarem estruturas condicionais e de repetição, e também as estruturas da gramática definida possuirem blocos aninhados, o código a ser análisado pelo compilador desencadeará na criação de mais tabelas de símbolos relacionado com a quatidade de escopos presente.

Os escopos na linguagem C são delimitados por meio dos *tokens* '{' (abre chaves), que sinalizam o início de um novo escopo, e '}' (fecha chaves), que denotam o término do escopo. A quantidade de escopos pode variar, uma vez que existem situações em que um escopo não introduz ou modifica variáveis, tornando desnecessária a criação de uma nova tabela de símbolos. Em situações que ocorrer a necessidade da criação de uma tabela de símbolos devido a presença de um novo escopo, esta nova tabela possuirá um ponteiro associado para a tabela de símbolos acima. A seguir, o código em linguagem C, conforme mostrado no Algoritmo 13, é seguido pela exibição dos resultados das tabelas de símbolos geradas na Figura 7.

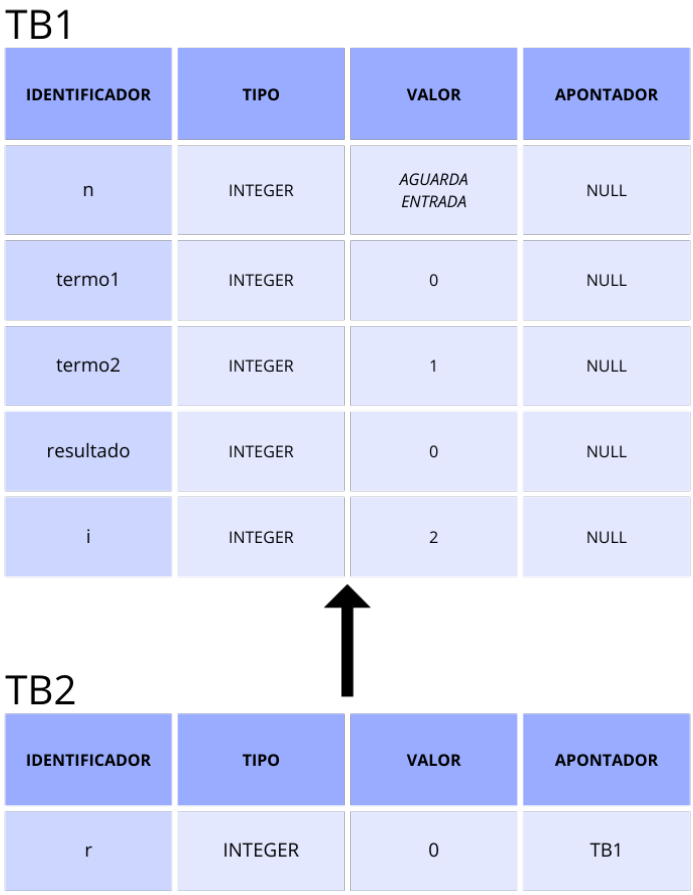
Algoritmo 13 – Exemplo algoritmo de fibonacci em C para exemplificar a criação da tabela de símbolos do compilador

```
1  // Exercicio usado como um exemplo e desafio para os estudantes no ambiente
2  // do WebAlgo atual. Este exemplo retorna o valor do n-esimo numero de
3  // Fibonacci
4
5
6  #include <stdio.h>
7
8  int main() {
9      int n;
10
11     scanf("%d", &n);
12     if (n <= 1) {
13         return n;
14     }
15
16     int termo1 = 0;
17     int termo2 = 1;
18     int resultado = 0;
19
20     for (int i = 2; i <= n; i++) {
21         resultado = termo1 + termo2;
22         termo1 = termo2;
23         termo2 = resultado;
24         int r = 0;  // Criada declaracao de variavel para demonstracao de
25                     // criacao de novo escopo;
26     }
```

```
27      printf("%d", resultado);
28      return 0;
29  }
```

Fonte: O Autor, 2024

Figura 7 – Tabela de símbolos inicial de Algoritmo 13



Fonte: O Autor, 2024

A Figura 7 resulta na criação de duas tabelas de símbolos (TB1 e TB2) devido à declaração da variável "r", que não está em uso. Sua presença no código tem o propósito de ilustrar a criação de uma nova tabela de símbolos (TB2), que compartilha um apontador com a tabela de símbolos criada anteriormente (TB1). Importante destacar que neste exemplo, caso a variável "r"fosse utilizada ou impressa na tela através do comando *printf* fora do escopo do laço de repetição, o compilador não permitiria a compilação e execução do código, resultando em um erro semântico de variável não declarada.

4.5 ARQUITETURA DA MÁQUINA VIRTUAL

A decisão sobre a escolha de uma máquina virtual envolveu várias possibilidades a serem consideradas. A primeira alternativa seria a utilização de uma máquina virtual no lado do servidor da aplicação, no entanto, essa opção não se alinhava com o objetivo central do trabalho, que era desenvolver uma compilação totalmente executada no lado do cliente. A segunda opção consistia em criar uma máquina virtual utilizando as estruturas da linguagem JavaScript, enquanto a terceira opção era buscar alguma ferramenta disponível que pudesse ser aproveitada para esse propósito.

Nesse contexto, a escolha recaiu sobre a segunda opção. Portanto, uma máquina virtual baseada em registradores foi desenvolvida utilizando os recursos disponíveis na linguagem Javascript para interpretar o código de três endereços, no qual, será utilizado como bytecode, tendo a sua geração durante a compilação do código de entrada fornecido pelo usuário. A escolha por desenvolver uma máquina virtual baseada em registradores se fundamenta através das análises realizadas no Capítulo 2, que destacam a máquina virtual baseada em registradores devido ao seu desempenho superior na execução e à geração de um menor número de instruções. A utilização do código de três endereços, como código intermediário do compilador também foi levado em consideração para a escolha da máquina, já que esta é uma forma de código intermediário utilizado para máquinas de registradores (AHO; SETHI; ULLMAN, 1995). O desenvolvimento da máquina virtual permite também a adição de novas implementações em próximos projetos como exemplo, a escalabilidade da aplicação para suportar outras linguagens ou adição de novas estruturas da linguagem C.

Para desenvolvimento da máquina virtual, foi examinada a viabilidade de empregar registradores de hardware para conduzir as operações durante a execução. Contudo, devido à escolha da linguagem JavaScript, interpretada pelos navegadores de internet, torna-se inviável utilizar registradores de hardware. Portanto, para a implementação da máquina virtual baseada em registradores, foi empregado os conceitos de uma máquina de registradores, simulando seu funcionamento por meio de registradores virtuais. Essa máquina virtual realizará a interpretação do código de três endereços, aderindo aos princípios de uma arquitetura de registradores.

Os *opcodes* destinados à interpretação pela máquina virtual consistirão diretamente nos códigos de três endereços, conforme exemplificado e com instruções detalhadas na Seção 4.3. Nessa mesma seção, serão exemplificadas e detalhadas as operações que a máquina virtual executa, proporcionando um mapeamento abrangente de todas as instruções C3E e uma explicação detalhada do processo de interpretação.

4.5.1 Operações artiméticas

As operações artiméticas que serão interpretadas pela máquina virtual são:

- Operação de soma;
- Operação de subtração;
- Operação de multiplicação;
- Operação de divisão;
- Operação de módulo;
- Operação de incremento e;
- Operação de decremento.

4.5.1.1 Soma

Instrução C3E: *(op1 = op2 + op3).*

Descrição: realiza a operação artimética de soma entre os operandos *op2* e *op3* e armazena o resultado em *op1*.

4.5.1.2 Subtração

Instrução C3E: *(op1 = op2 - op3).*

Descrição: realiza a operação artimética de subração entre os operandos *op2* e *op3* e armazena o resultado em *op1*.

4.5.1.3 Multiplicação

Instrução C3E: *(op1 = op2 * op3).*

Descrição: realiza a operação artimética de multiplicação entre os operandos *op2* e *op3* e armazena o resultado em *op1*.

4.5.1.4 Divisão

Instrução C3E: *(op1 = op2 / op3).*

Descrição: realiza a operação artimética de divisão entre os operandos *op2* e *op3* e armazena o resultado em *op1*.

4.5.1.5 Módulo ou resto da divisão

Instrução C3E: *(op1 = op2 % op3).*

Descrição: realiza a operação artimética de módulo entre os operandos *op2* e *op3* e armazena o resultado em *op1*.

4.5.1.6 Incremento

Instrução C3E: *(op1 = op2 + 1).*

Descrição: realiza a operação artimética somar uma unidade ao operando *op2* e armazena o resultado em *op1*.

Algoritmo 14 – Exemplo de instrução de incremento equivalente na linguagem C

```
1 // Situacao 1 - pos-fixa
2 int a = 0;
3 a++;
4
5 // Situacao 2 - pre-fixa
6 int a = 0;
7 ++a;
```

Fonte: O Autor, 2024

4.5.1.7 Decremento

Instrução C3E: *(op1 = op2 - 1).*

Descrição: realiza a operação artimética subtrair uma unidade ao operando *op2* e armazena o resultado em *op1*.

Algoritmo 15 – Exemplo de instrução de decremento equivalente na linguagem C

```
1 // Situacao 1 - pos-fixa
2 int a = 0;
3 a--;
4
5 // Situacao 2 - pre-fixa
6 int a = 0;
7 --a;
```

Fonte: O Autor, 2024

4.5.2 Operações lógicas

As operações lógicas que serão interpretadas pela máquina virtual são:

- Operação de E lógico; e
- Operação de Ou lógico.

4.5.2.1 E lógico

Instrução C3E: *(op1 = op2 && op3).*

Descrição: realiza a operação de análise lógica '&&' entre op2 e op3 e armazena o valor lógico resultante da operação em op1, sendo os valores possíveis *true* ou *false*

Algoritmo 16 – Exemplo de instrução de E lógico equivalente na linguagem C

```
1 // Situacao 1 - pos-fixa
2 if (A && B){
3     COMANDOS
4 }
```

Fonte: O Autor, 2024

4.5.2.2 Ou lógico

Instrução C3E: *(op1 = op2 || op3).*

Descrição: realiza a operação de análise lógica '||' entre op2 e op3 e armazena o valor lógico resultante da operação em op1, sendo os valores possíveis *true* ou *false*

Algoritmo 17 – Exemplo de instrução de OU lógico equivalente na linguagem C

```
1 // Situacao 1 - pos-fixa
2 if (A || B){
3     COMANDOS
4 }
```

Fonte: O Autor, 2024

4.5.3 Operações relacionais

As operações relacionais que serão interpretadas pela máquina virtual são:

- Operação de igualdade;
- Operação de menor que;
- Operação de menor igual que;
- Operação de maior que; e
- Operação de maior igual que.

4.5.3.1 Igualdade

Instrução C3E: *(op1 = op2 == op3).*

Descrição: realiza a operação de análise relacional de op2 é igual a op3 e armazena o valor lógico resultante da operação em op1, sendo os valores possíveis *true* ou *false*

Algoritmo 18 – Exemplo de instrução de igualdade equivalente na linguagem C

```
1 // Situacao 1 - pos-fixa
2 if (A == B){
3     COMANDOS
4 }
```

Fonte: O Autor, 2024

4.5.3.2 Menor que

Instrução C3E: *(op1 = op2 < op3).*

Descrição: realiza a operação de análise relacional de op2 é menor que op3 e armazena o valor lógico resultante da operação em op1, sendo os valores possíveis *true* ou *false*

Algoritmo 19 – Exemplo de instrução de menor que equivalente na linguagem C

```
1 // Situacao 1 - pos-fixa
2 if (A < B){
3     COMANDOS
4 }
```

Fonte: O Autor, 2024

4.5.3.3 Menor igual que

Instrução C3E: *(op1 = op2 <= op3).*

Descrição: realiza a operação de análise relacional de op2 é menor igual que op3 e armazena o valor lógico resultante da operação em op1, sendo os valores possíveis *true* ou *false*

Algoritmo 20 – Exemplo de instrução de menor igual que equivalente na linguagem C

```
1 // Situacao 1 - pos-fixa
2 if (A <= B){
3     COMANDOS
4 }
```

Fonte: O Autor, 2024

4.5.3.4 Maior que

Instrução C3E: *(op1 = op2 > op3).*

Descrição: realiza a operação de análise relacional de op2 é maior que op3 e armazena o valor lógico resultante da operação em op1, sendo os valores possíveis *true* ou *false*

Algoritmo 21 – Exemplo de instrução de maior que equivalente na linguagem C

```
1 // Situacao 1 - pos-fixa
2 if (A > B){
3     COMANDOS
4 }
```

Fonte: O Autor, 2024

4.5.3.5 Maior igual que

Instrução C3E: *(op1 = op2 >= op3).*

Descrição: realiza a operação de análise relacional de op2 é maior igual que op3 e armazena o valor lógico resultante da operação em op1, sendo os valores possíveis *true* ou *false*

Algoritmo 22 – Exemplo de instrução de maior igual que equivalente na linguagem C

```
1 if (A >= B){
2     COMANDOS
3 }
```

Fonte: O Autor, 2024

4.5.4 Operações de desvios

As operações lógicas que serão interpretadas pela máquina virtual são:

- Operação de desvio condicional;
- Operação de desvio incondicional;
- Operação de chamada de função; e
- Operação de retorno de função.

4.5.4.1 Desvio condicional

Instrução C3E: *(ifFalse op1 goto LABEL).*

Descrição: realiza a operação de análise relacional de op2 é maior igual que op3 e armazena o valor lógico resultante da operação em op1, sendo os valores possíveis *true* ou *false*

Algoritmo 23 – Exemplo de instrução de desvio condicional equivalente na linguagem C

```
1 // Expressao if gera desvio condicional
2 if (A >= B){
3     COMANDOS
4 }
5
```

```
6  // Expressao while gera desvio condicional
7  while (A >= B){
8      COMANDOS
9  }
10
11 // Expressao for gera desvio condicional
12 for (; A >= B; ){
13     COMANDOS
14 }
```

Fonte: O Autor, 2024

4.5.4.2 Desvio incondicional

Instrução C3E: *(goto LABEL)*.

Descrição: realiza salto sem condição para posição do *LABEL* indicado. Utilizado para repetir comandos dentro de laços de repetição.

4.5.4.3 Empilhamento de parâmetros e chamada de função

Instrução C3E:

param a

call LABEL

Descrição: a chamada de função realiza um salto sem condição para posição do *LABEL* indicado e juntamente passa os parâmetros que foram previamente instanciados, empilhados na lista e que serão utilizados na função.

Algoritmo 24 – Exemplo de chamada de função equivalente na linguagem C

```
1 printf("%d", a);
```

Fonte: O Autor, 2024

4.5.4.4 Retorno de função

Instrução C3E:(*RET <parâmetro>*)

Descrição: realiza retorno da função para o seguinte ponto após a chamada dela no código de três endereços podendo retornar um parâmetro para o escopo anterior.

Algoritmo 25 – Exemplo de eetorno de função equivalente na linguagem C

```
1 return 0;
```

Fonte: O Autor, 2024

A chamada da função *printf* resultará em uma saída exibida no console para o usuário que estiver executando o código em C. Por outro lado, a chamada da função *scanf* causará uma interrupção na interpretação do código de três endereços pela máquina virtual, permitindo que o usuário insira um valor a ser utilizado na execução. A retomada da execução só ocorrerá após o usuário fornecer todos os valores solicitados pela função *scanf*.

4.6 ARQUITETURA DO SISTEMA

A seção sobre a arquitetura do sistema tem como objetivo revisar o desenvolvimento da proposta de solução para a aplicação do compilador *web*. Foi introduzida a estrutura básica do *frontend*, a qual será detalhada para permitir a edição do código-fonte pelo aluno. Isso inclui também um console para exibir as saídas de dados do programa por meio das funções *printf*, presentes nos códigos compilados e interpretados pela máquina virtual. Após, serão detalhadas as ferramentas e a linguagem de programação que foram utilizadas no desenvolvimento da aplicação. Dois casos de uso e o digrama de domínio da aplicação serão apresentados, nos quais o aluno desempenha o agente dos casos, o primeiro, caso de execução do código e o segundo caso de depuração do código. Ao final da seção, de maneira simplificada, serão recapitulados todos os passos envolvidos na criação do compilador e da máquina virtual, com a finalidade de consolidar integralmente a proposta nesta seção.

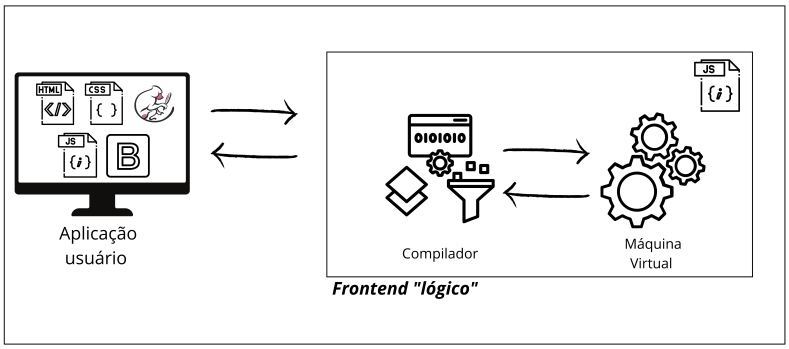
Pelo fato de a aplicação do compilador concentrar-se totalmente no navegador, a linguagem principal adotada é o JavaScript, conforme ilustrado na Figura 8, que apresenta a estrutura da aplicação. O JavaScript desempenha um papel integral na manipulação das interações na tela do aluno, sendo destacado como "*Aplicação usuário*". Ele é utilizado para o desenvolvimento completo do compilador, abrangendo desde a verificação de tokens na análise léxica até a validação da gramática, definida na Seção 4.2, por meio do analisador sintático (*parser*). Além disso, o JavaScript é responsável pela verificação de tipos utilizando a tabela de símbolos, que armazena as declarações das variáveis (estruturada como um objeto em JavaScript), e pela geração dos bytecodes (C3E) destinados à interpretação pela máquina virtual.

A máquina virtual em si, baseada em registradores, também é implementada utilizando a linguagem JavaScript. Essa fase de compilação, geração de código e máquina virtual é referida como "*Frontend lógico*", enquanto a integração disso com a "*Aplicação usuário*"é representada pelo termo "*Frontend*".

Além disso, devido à natureza do ambiente de execução no navegador, foi necessário utilizar a estrutura de um *worker* em JavaScript. O *worker* é um mecanismo de JavaScript que permite a execução de tarefas em segundo plano, fora da *thread* principal de execução do navegador. Isso é fundamental, pois, em casos de laços infinitos ou execuções demoradas, o *worker* possibilita o cancelamento da execução do compilador, evitando que o navegador trave. Sem o uso de *workers*, um laço infinito poderia resultar em um bloqueio do navegador, exigindo o

encerramento manual da aplicação e a necessidade de reabri-la. Assim, o *worker* garante a continuidade da operação do navegador e a capacidade de interromper tarefas quando necessário, mantendo a interface do usuário responsiva e funcional durante o processo de compilação.

Figura 8 – Exemplo da arquitetura de sistema e suas respectivas ferramentas



Fonte: O Autor, 2024

Para a "*Aplicação Usuário*", as ferramentas empregadas no desenvolvimento da interface inclui o JavaScript para a manipulação dinâmica dos elementos na tela e para realizar chamadas de compilação e execução do código inserido pelo usuário. O HTML é utilizado para criar os principais elementos visuais da página, destacando-se o editor de texto que é descrito ainda nessa seção, no qual oferecerá a possibilidade de implementar *breakpoints* para depuração diretamente no editor. Adicionalmente, foram incluídas duas "caixas de texto"não editáveis destinada a exibir as informações da variáveis pós execução ou durante a execução passo a passo e também informações essenciais para o aluno (*console*), como a execução bem-sucedida do seu código, juntamente com os dados de saída retornados pelo interpretador. Em caso de erros de compilação, a caixa de texto exibirá detalhadamente o tipo de erro (léxico, sintático ou semântico - declaração de variáveis), incluindo a linha e posição onde ocorreram. Por fim, foi incorporado um botão para iniciar a execução do código e outro para pular, para o caso de estar sendo realizada uma depuração.

O CSS é utilizado para o controle, posicionamento e personalização dos elementos HTML na tela. Visando otimizar o desenvolvimento, o Bootstrap[3] foi empregado para agilizar a criação dos elementos utilizados na aplicação.

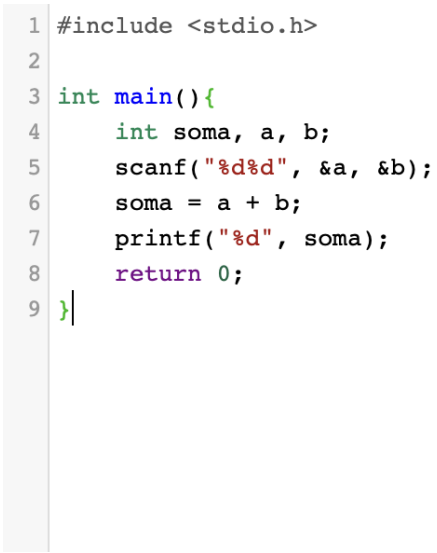
Uma outra ferramenta apresentada na imagem é o CodeMirror [4], conforme discutido na Seção 2.3. O CodeMirror é um editor de texto "importável"no código HTML, utilizado para a

[3] O Bootstrap tem como objetivo acelerar o desenvolvimento de diversos elementos, como botões, alertas, campos editáveis, e outros, proporcionando esses elementos prontos para uso. Essa biblioteca visa eliminar a necessidade do desenvolvedor programar e personalizar esses elementos individualmente, permitindo a entrega de componentes visuais prontos e esteticamente agradáveis na interface. Para este trabalho será utilizada a versão 5.2, última versão disponível na data da realização do presente trabalho.

[4] Para este trabalho foi utilizada a versão 5.63.1.

digitação de código C pelo aluno. A escolha dessa ferramenta baseou-se em sua documentação completa e detalhada, além da extensa comunidade de desenvolvedores que a utilizam em suas aplicações. No desenvolvimento da aplicação do compilador na *web*, o CodeMirror oferece eficiência significativa, uma vez que seu editor de texto contempla todas as funcionalidades planejadas para o desenvolvimento do compilador. Isso inclui diferenciação de cor no editor de texto para tokens da linguagem C (Figura 9) e também funcionalidades gráficas para navegar pelas linhas e facilitar a depuração do código inserido pelo aluno através da visualização.

Figura 9 – Exemplo de identificação de tokens na liguagem C com o editor CodeMirror



Fonte: O Autor, 2024

4.6.1 Casos de uso do sistema

O detalhamento dos casos de uso do sistema durante a fase de planejamento do desenvolvimento visa simplificar a identificação de todos os requisitos essenciais. Isso contribui para que o processo de desenvolvimento seja planejado de forma a se aproximar o máximo possível da versão final do sistema, prevenindo surpresas indesejadas ao longo do desenvolvimento. A definição dos casos de uso devem ser simplificadas, para fácil compreensão mas suficientemente detalhadas (LARMAN, 2007).

Neste projeto, os casos de uso seguiram duas instâncias em que o autor é o aluno. O primeiro caso diz respeito à execução direta do código programado pelo aluno, enquanto o segundo envolve a execução do código programado pelo aluno em conjunto com a execução passo a passo do depurador. É fundamental ressaltar que os casos de uso apresentados concentram-se na execução do código C inserido pelo aluno. Isso implica verificar se o compilador interpretou corretamente o código, resultando na obtenção do retorno esperado.

4.6.1.1 Caso de uso 1: executar código programado

O propósito deste caso de uso é a execução do código C elaborado pelo aluno, visando a compilação e execução bem-sucedida, assim obtendo o resultado correto ao final do processo.

Ator principal: Aluno

Interessado e interesses:

- Aluno: desenvolver código C através do editor de texto presente na aplicação com o objetivo de que o código desenvolvido execute corretamente retornando o resultado esperado.

Pré-condições: Digitar o código C respeitando as regras léxicas (*tokens* do subconjunto definido), sintática (gramática do subconjunto definido) e semânticas (validar os tipos de variáveis).

Pós-condições: O código foi compilado e executado da maneira correta, sem erros e, por fim, retornando o resultado "correto"ou "esperado"de acordo com o código do aluno.

Fluxo Básico:

1. Aluno digita o código C respeitando as regras léxicas, sintáticas e semânticas.
2. Aluno inicia a compilação e execução do seu código.
3. O compilador realiza a análise léxica, sintática e semântica e, em paralelo, a análise realiza a geração do código de três endereços.
4. Máquina virtual interpretará o código de três endereços gerado pelo compilador, a fim de executar as operações lógicas e aritméticas, para que o resultado seja obtido, e realizar as determinadas entradas (*scanf*) e saídas (*printf*) de dados conforme estiverem presentes no código.
5. Compilação e execução realizadas com sucesso, sendo exibido no console da aplicação a mensagem de sucesso da compilação e execução e os resultados obtidos caso o programa executado tenha comandos de saída de dados (*printf*).

Fluxos Alternativos:

- Aparição de erro léxico durante a compilação:
    1. Compilação será interrompida.
    2. Erro léxico será exibido para o aluno com a linha e posição que ocorreu.
- Apararição de erro sintático durante a compilação:

1. Compilação será interrompida.
2. Erro sintático será exibido para o aluno com a linha e posição que ocorreu.

- Apararição de erro semântico durante a compilação:

1. Compilação será interrompida.
2. Erro semântico será exibido para o aluno com a linha e posição que ocorreu.

4.6.1.2 Caso de uso 2: depuração do código desenvolvido

O propósito deste caso de uso serve como uma extensão do caso de uso 1 apresentado na Seção 4.6.1.1. Nesta ocasião o aluno realizará também a execução do seu código C digitado, porém esta execução ocorrerá de forma passo a passo, ou seja, seu código será depurado para que possa ser acompanhada as execuções das instruções. Esta funcionalidade é utilizada em situações que possa estar divergindo o resultado do código por alguma instrução incorreta ou deslocada, logo a depuração torna-se um facilitador para o aluno encontrar o problema.

Ator principal: Aluno

Interessado e interesses:

- Aluno: depurar o código que digitou, seja para corrigir resultados incorretos ou para compreender o funcionamento do código que desenvolveu.

Pré-condições: Digitar o código C respeitando as regras léxicas (*tokens* do subconjunto definido), sintática (gramática do subconjunto definido) e semânticas (validar os tipos de variáveis) e inserir *breakpoint* para possibilitar a parada do depurador.

Pós-condições: O código foi compilado, depurado e executado da maneira correta no qual foi digitado, sem erros e por fim o aluno consegue identificar o problema do seu código através da depuração passo a passo.

Fluxo Básico:

1. Aluno digita o código C respeitando as regras léxicas, sintáticas e semânticas.
2. Aluno sinaliza no editor de texto a linha que deseja inserir o *breakpoint* para realizar a parada.
3. O aluno inicia a compilação do seu código.
4. O compilador realiza a análise léxica, sintática e semântica e, em paralelo, a análise realiza a geração do código de três endereços.

5. A Máquina virtual interpretará o código de três endereços gerado pelo compilador, executando operações lógicas e aritméticas até o ponto em que o *breakpoint* foi definido. A partir desse ponto, o programa só continuará sua execução mediante a intervenção do aluno.

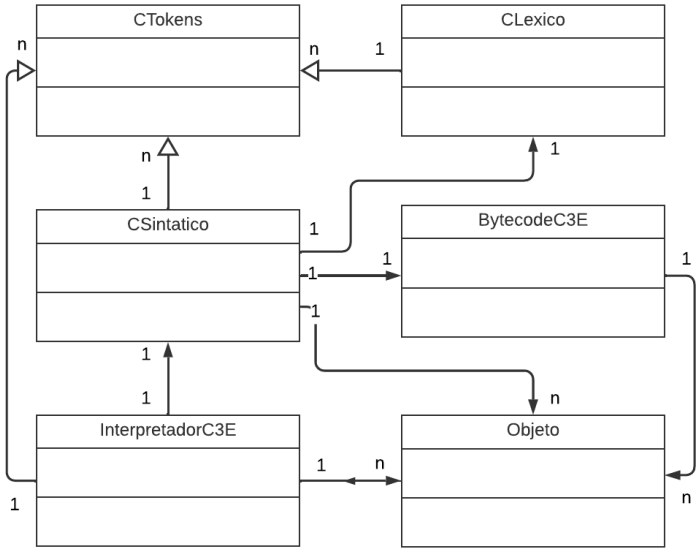
6. O aluno percorrerá o código até o término da execução, identificando e solucionando os problemas presentes em seu código.

7. Compilação, depuração e execução realizadas com sucesso, sendo exibido no console da aplicação a mensagem de sucesso da compilação e execução e os resultados obtidos caso o programa executado tenha comandos de saída de dados (*printf*).

4.6.2 Modelo de domínio do sistema

O modelo de domínio apresentado na Figura 10 detalha o funcionamento do compilador através das classes exibidas.

Figura 10 – Modelo de domínio do compilador



Fonte: O Autor, 2024

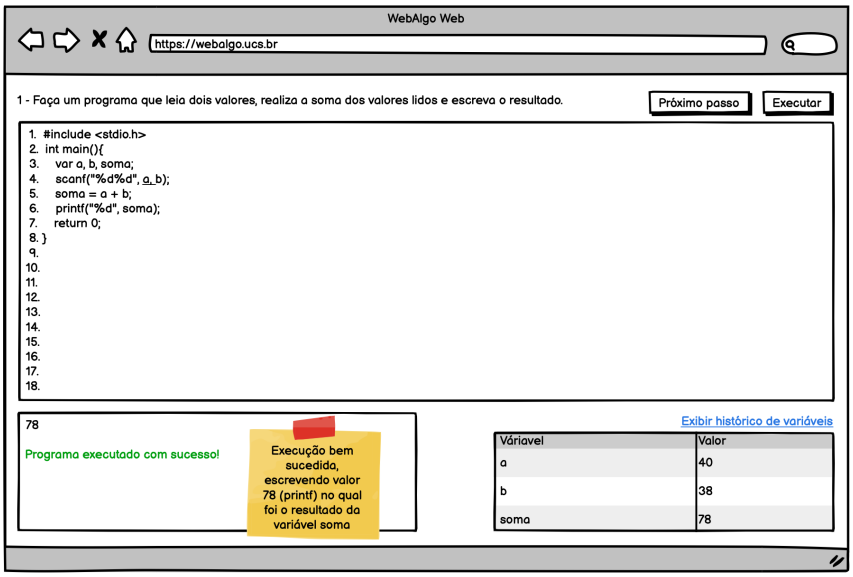
4.6.3 *Mockup* de tela do compilador

Com a definição do trabalho restrita à compilação e validação do código C inserido pelo aluno, as configurações da interface do compilador WebAlgo na *web* foram simplificadas para incluir apenas os elementos essenciais para o seu funcionamento. Portanto, não serão abordadas em detalhes as estruturas completas presentes na versão atual do WebAlgo, tais como login

de usuário, carregamento de exercícios do banco de dados do WebAlgo, salvamento de códigos desenvolvidos e exibição do ranking de algoritmos "mais performáticos"de usuários por exercício.

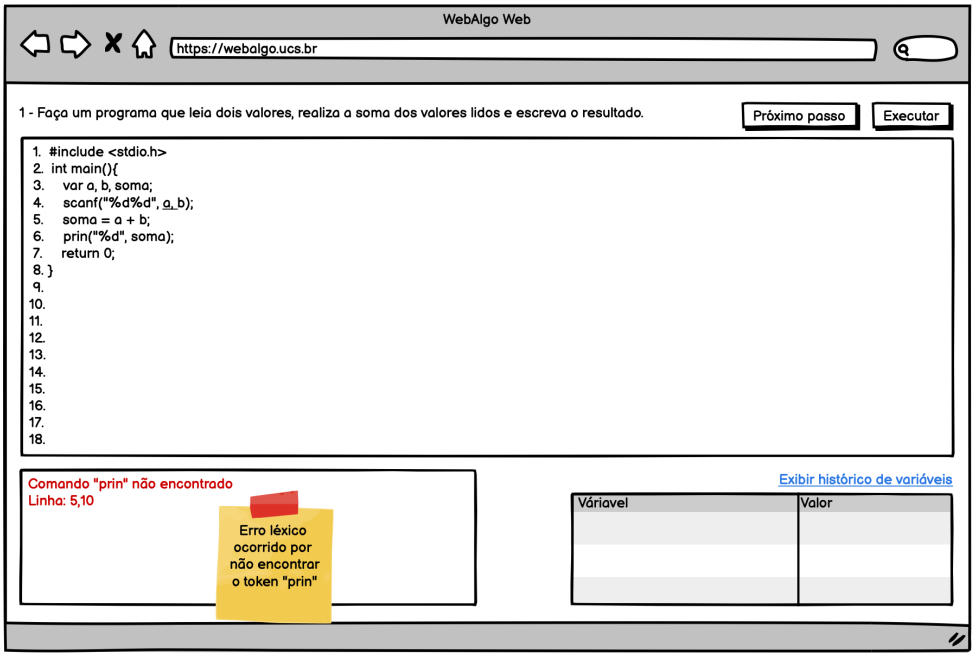
Essas funcionalidades estarão disponíveis para implementação em trabalhos futuros relacionados a este projeto. Dessa forma, as estruturas semelhantes à versão atual do WebAlgo que foram implementadas incluem: um editor de texto, um console para exibição de saídas do código (*printf*) e mensagens de sucesso ou erro, sendo que em caso de erro, é indicada a linha e a posição correspondentes, apresentados nas ilustrações Figura 11 e Figura 12. Além disso, há uma tabela de variáveis que permite visualizar o histórico de vida das variáveis ao longo da execução do código desenvolvido, Figura 13. Por fim, a entrada de dados é realizada por meio do console que guarda a digitação de algum valor pelo usuário, Figura 14.

Figura 11 – Mockup de tela do WebAlgo Web - execução sucedida



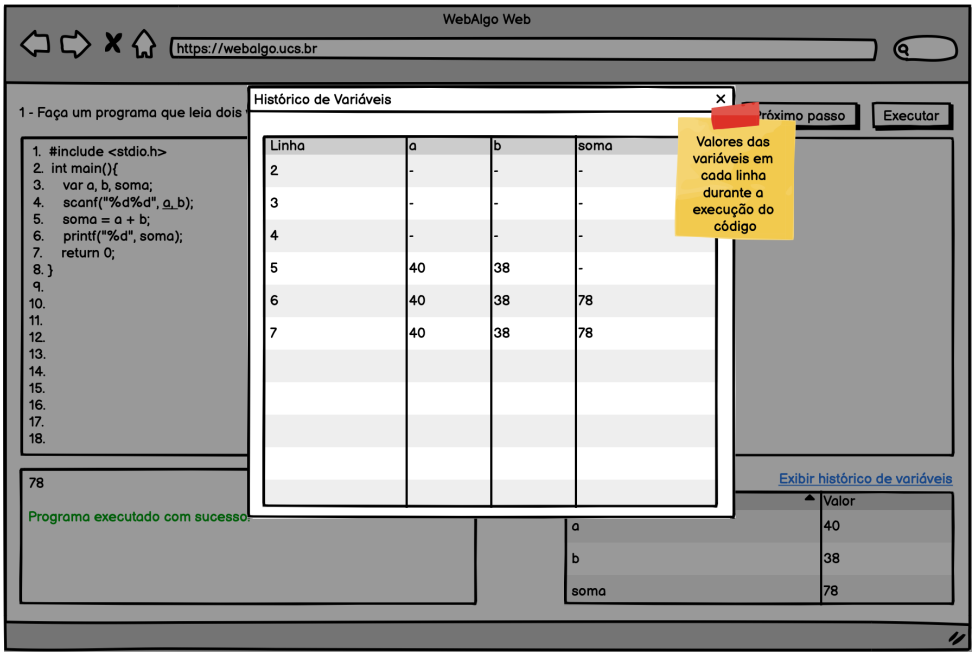
Fonte: O Autor, 2024

Figura 12 – Mockup de tela do WebAlgo Web - execução com erro léxico



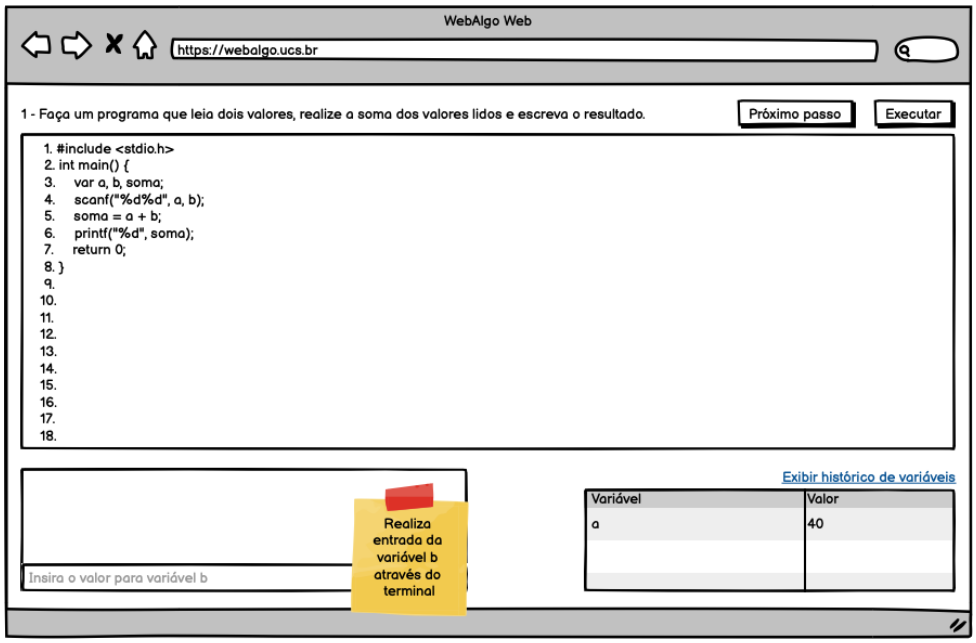
Fonte: O Autor, 2024

Figura 13 – Mockup de tela do WebAlgo Web - histórico de variáveis



Fonte: O Autor, 2024

Figura 14 – Mockup de tela do WebAlgo Web - interrupção para aguardo de digitação do usuário devido a chamada da função *scanf*



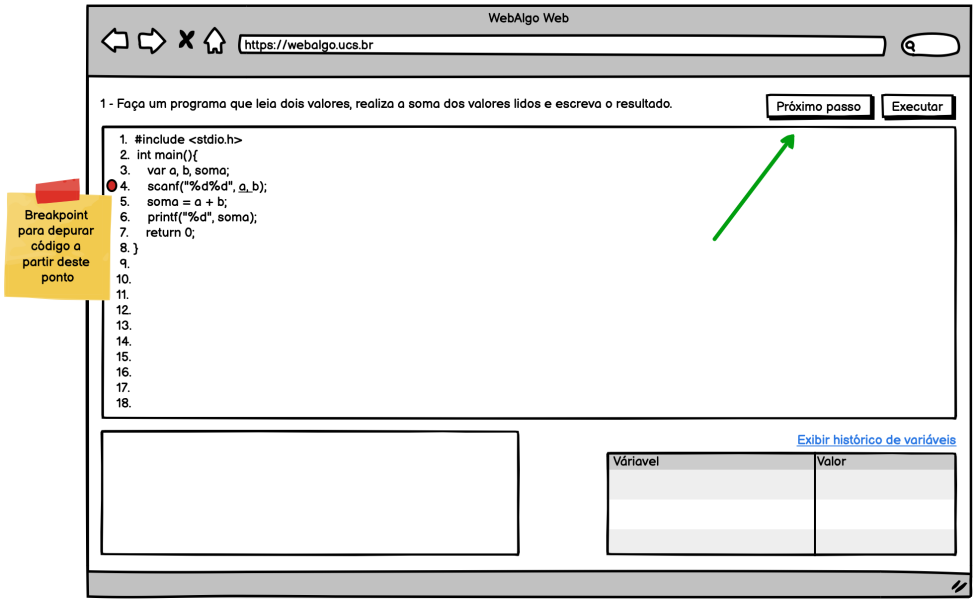
Fonte: O Autor, 2024

4.7 DEPURADOR

O depurador tem o objetivo de realizar a execução do código-fonte digitado pelo aluno passo a passo. Ele atua como uma ferramenta facilitadora para analisar o funcionamento do código desenvolvido, permitindo a verificação passo a passo do comportamento das variáveis. Em outras palavras, ele examina os valores que as variáveis estão adquirindo ao longo da execução do código, frequentemente utilizado para resolver problemas relacionados a algoritmos.

Para o compilador *web*, o depurador não será diferente. Conforme descrito no caso de uso 2 da Seção 4.6.1, para que o aluno consiga depurar o código o requisito para que isso aconteça é sinalizar no editor de texto, através da marcação de um *breakpoint* na linha desejada. Assim, ao iniciar a execução do código o interpretador realiza a execução até o ponto de parada e para continuar a execução a partir deste ponto somente com o prosseguimento do aluno através do clique no botão próximo passo, ilustrado na Figura 15. A partir do clique no botão executar, inicia a execução do código e o mesmo botão se transformará em uma opção de cancelar a execução para possibilitar a parada da execução em situações de loop infinito.

Figura 15 – Mockup de tela do WebAlgo Web - depurador, breakpoint e botão passo a passo



Fonte: O Autor, 2024

Conforme a organização do modelo de domínio apresentado na Seção 4.6.2, para que o código-fonte seja depurado é necessário que antes ele seja compilado com sucesso, sem nenhum erro léxico, sintático ou semântico e que o código de três endereços também já esteja gerado, ou seja, a depuração é realizada na etapa de interpretação do C3E pela máquina virtual.

Para que seja possível realizar as paradas durante a execução do código, a cada linha de código digitado pelo aluno será convertida para C3E, no qual, é atribuído a classe *Object* dentro de um *ArrayList*. Em termos simplificados, cada linha digitada resultará em uma posição dentro do vetor. Essa organização visa simplificar não apenas a depuração, mas também possibilitar a execução dos saltos para os rótulos (*labels*) do C3E através das posições no vetor.

Para que o interpretador entenda que é necessário realizar a espera de um comando do usuário, esta interrupção é identificada através de um dado dentro do *Object*, no qual possui o trecho de C3E que será executado. Neste caso, o interpretador ficará aguardando, através de um observador que "escuta"o evento de clique no botão "próximo passo" assim prosseguindo a depuração.

É válido ressaltar que, neste contexto, o depurador opera da seguinte maneira: ao estabelecer um *breakpoint* para interromper a execução do código para depuração, o aluno deve seguir os passos um a um até a conclusão da execução pelo interpretador.

5 DESENVOLVIMENTO

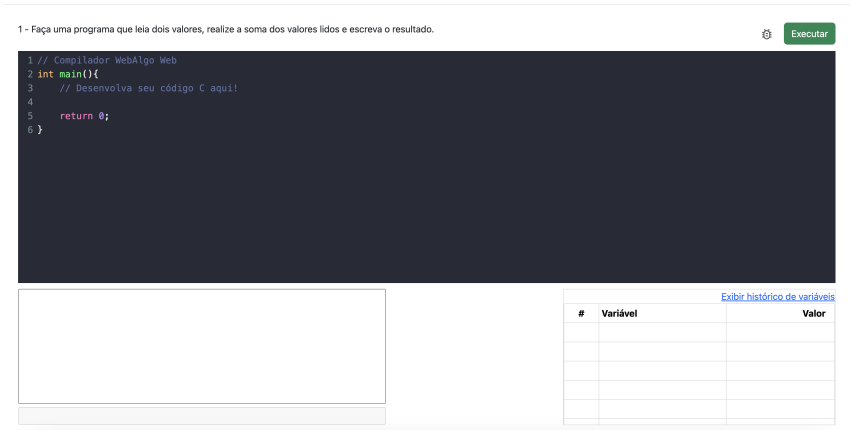
Com base nos assuntos discutidos nos capítulos anteriores e na solução proposta, as funcionalidades foram desenvolvidas de maneira totalmente nova, sem reutilizar qualquer estrutura existente do WebAlgo. Durante o desenvolvimento, foram necessárias algumas alterações pontuais, sem ocorrer mudanças significativas. Portanto, pode-se concluir que o planejamento prévio, apresentado na proposta, facilitou o desenvolvimento do compilador *web*.

Este capítulo apresentará o percurso do desenvolvimento do compilador *web*, começando pelas ferramentas utilizadas no desenvolvimento, detalhando as etapas de análise léxica, análise sintática, análise semântica, geração de código intermediário, desenvolvimento da máquina virtual baseada em registradores virtuais e concluindo com o depurador do compilador.

5.1 ESTRUTURA DE DESENVOLVIMENTO DA TELA DO COMPILADOR

O desenvolvimento do compilador iniciou com a preparação do ambiente de desenvolvimento. Foi escolhido um sistema de controle de versionamento para facilitar o acompanhamento do progresso e garantir a segurança dos códigos desenvolvidos. Optou-se pelo GitHub, onde o trabalho está disponível para acesso[1] público. Com a estrutura do controle de versionamento definida, iniciou-se o desenvolvimento do compilador, replicando os *layouts* de tela apresentados na proposta. A Figura 16 apresenta o design desenvolvido no compilador *web*, baseado no *mockup* proposto para o frontend do compilador.

Figura 16 – *Frontend* do Compilador *Web*



Fonte: O Autor, 2024

Para o desenvolvimento desta tela, foram utilizadas as seguintes estruturas de *frontend*:

[1] https://github.com/glsusin/C-Compiler

HTML, CSS e JavaScript, sem a presença de um *framework* específico para *frontend*. Para a personalização dos elementos na tela e com o objetivo de acelerar o desenvolvimento, foram empregadas algumas bibliotecas: Bootstrap (versão 5.3.3) para agilizar o design dos elementos de botões, modais e tabelas, jQuery (versão 3.6.0) para facilitar a interação do JavaScript com os elementos HTML na tela, e o CodeMirror (versão 5.63.1), no qual é responsável pelo editor de texto para desenvolvimento do código em C. Foram utilizadas quatro extensões da biblioteca CodeMirror: a extensão básica do editor de texto, a extensão do tema Drácula, a extensão para destacar os *tokens* da linguagem C, e a extensão *autocomplete*, que permite a digitação dos principais *tokens* da liguagem C para acelerar a digitação do usuário.

Na Figura 16, é possível observar os elementos essenciais propostos neste trabalho. No canto superior direito, estão os botões "Executar"e, ao lado, o botão com o ícone de um inseto, que indica o botão de depuração. No canto superior esquerdo, há uma representação de como seria uma pergunta proposta ao aluno no compilador *web*. Abaixo, centralizado no meio da tela, está o editor de texto configurado para a linguagem C. No canto inferior direito, encontra-se a tabela de variáveis carregada durante a execução do código intermediário de três endereços pela máquina virtual, que será detalhada mais adiante neste capítulo. Logo acima da tabela, há um *link* HTML para listar o histórico das variáveis durante a execução do programa. Por fim, no canto inferior esquerdo, está o *console*, onde serão listadas todas as saídas do programa (*printf* ou alertas de erro de compilação), com um campo abaixo para a inserção de dados quando solicitado pelo compilador através do comando de leitura (*scanf*).

5.2 ANÁLISE LÉXICA

A análise léxica, cujo objetivo é identificar os *tokens* da linguagem a partir dos caracteres do código a ser compilado, é a primeira etapa do compilador e foi a primeira a ser desenvolvida. No compilador *web*, a análise léxica é composta por duas funções essenciais. A função ***getToken()*** é responsável pela identificação de um dos 57 *tokens*, apresentados na Figura 17, presentes no compilador a partir de um ou mais caracteres. A segunda função, ***proxC()***, é chamada internamente pela ***getToken()*** e é responsável por percorrer caractere a caractere do código em compilação até que um *token* seja identificado. É importante destacar que a função ***proxC()*** também identifica a linha(armazenado na variável global *count_line*) e a posição na linha do compilador(armazenado na variável global *count_column*), informações que serão utilizadas para mensagens de erros léxicos, sintáticos e semânticos, além de serem essenciais para a geração de código de três endereços, possibilitando assim, a marcação das linhas no editor de texto durante a depuração.

Figura 17 – *Tokens* utilizados no Compilador *Web*

Tokens	Caractere	Tokens	Caractere
TKId	IDENTIFICADOR	TKDuploMenos	--
TKVoid	void	TKMenosIgual	-=
TKInt	int	TKMenos	-
TKFloat	float	TKMultIgual	*=
TKDouble	double	TKMult	*=
TKIf	if	TKDivIgual	/=
TKElse	else	TKDiv	/
TKDo	do	TKRestoIgual	%=
TKWhile	while	TKResto	%
TKFor	for	TKCompare	==
TKBreak	break	TKIgual	=
TKReturn	return	TKDiferent	!=
TKCteDouble	CONSTANTE	TKLogicalNot	!
TKCteInt	CONSTANTE	TKMenorIgual	<=
TKVirgula	,	TKMenor	<=
TKPonto	.	TKMaiorIgual	>=
TKDoisPontos	:	TKMaior	>=
TKPontoEVirgula	;	TKLogicalAnd	&&
TKAbreParenteses	(	TKLogicalOr	||
TKFechaParenteses	)	TKContinue	continue
TKAbreColchete	[	TKPrintf	printf
TKFechaColchete	]	TKScanf	scanf
TKAbreChaves	{	TKStringStdio	"QUALQUER CARACTERE"
TKFechaChaves	}	TKEnderecoVariavel	&
TKDuploMais	++	TKDefine	#define
TKMaisIgual	+=	TKStdioh	<stdio.h>
TKMais	+	TKInclude	#include
TKMathh	<math.h>	TKChar	char

Fonte: O Autor, 2024

No fluxo de execução do compilador, a análise léxica é chamada sequencialmente após o reconhecimento de cada *token* durante a análise sintática. No Algoritmo 26, é apresentado um exemplo do código sintático do compilador para o comando ***WHILE***, no qual a função ***getToken()*** é chamada após as verificações dos *tokens* nas linhas 3, 5 e 8 do algoritmo. Este exemplo foi modificado para destacar apenas as chamadas da função ***getToken()***, não incluindo a geração de código intermediário.

Se a função ***getToken()*** não encontrar um *token* válido durante a sua execução na análise léxica, uma exceção será executada e um erro léxico será emitido. Um exemplo de erro léxico não contemplado pela gramática da linguagem C é a utilização do símbolo "@"ou o comentário "//"utilizado em C++. Caso o analisador léxico encontre esses caracteres fora de um comentário ou de uma *string*, o compilador irá gerar uma exceção, interrompendo a compilação e emitindo um erro léxico no console do compilador *web*. Para o compilador *web*, o usuário é informado sobre o caractere não encontrado e também a linha e coluna no qual ocorreu o erro léxico.

Algoritmo 26 – Chamadas da função *getToken()* após verificações de *token* na análise sintática do comando *WHILE* da gramática do compilador *web*

```
funtion InstrIteracao(){
    // Verifica Token
    if (globalVarC.tk === globalVarC.TKs["TKWhile"]){
        getToken();  // Busca proximo token
        // Verifica Token
        if (globalVarC.tk === globalVarC.TKs["TKAbreParenteses"]){
            getToken();  // Busca proximo token
            if (Expressao()){
                // Verifica Token
                if (globalVarC.tk === globalVarC.TKs["TKFechaParenteses"]){
                    getToken();  // Busca proximo token
                    if (Instr()){
                        return true;
                    }
                } return false;
            } return false;
        } return false;
    } return false;
}
```

Fonte: O Autor, 2024

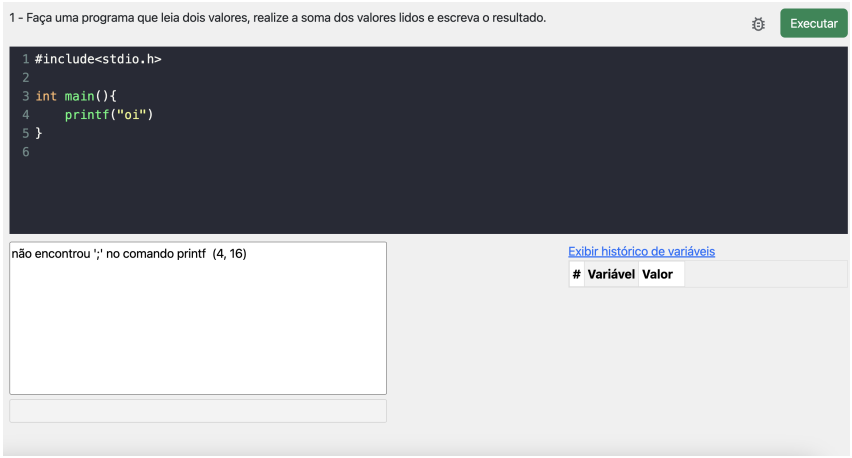
5.3 ANÁLISE SINTÁTICA

O analisador sintático do compilador *web* foi construído de acordo com a gramática apresentada na Seção 4.2. A análise sintática verifica se a estrutura do código em compilação está em conformidade com a gramática definida, validando os *tokens* conforme especificados na gramática e através de chamadas recursivas, se necessário. No compilador *web*, as análises léxica e semântica, bem como a geração de código de três endereços, são realizadas durante a execução do analisador sintático. Assim, a função ***Programa()***, que executa a análise sintática, é a função principal de execução do compilador e instrução inicial da gramática.

A análise sintática é a engrenagem principal de um compilador, pois verifica a conformidade do programa com a gramática, garantindo sua execução correta. Além disso, em casos de erros sintáticos, o compilador identifica-os corretamente e apresenta-os ao usuário, facilitando a correção do código digitado no editor de texto. Os erros sintáticos são alertados quando o compilador, durante a análise de *tokens*, não encontra um *token* conforme esperado pela gramática. Nestes casos, a compilação deve retornar um erro indicando o *token* ausente. No compilador *web*, o usuário é informado sobre o caractere ou *token* ausente, junto com a linha e coluna onde ocorreu o erro sintático, especificando também se o *token* está dentro de alguma estrutura específica da gramática. A Figura 18 ilustra um erro sintático devido à falta de ponto-e-vírgula no

comando *printf*, explicitando o erro na linha 4 e coluna 16 do editor de texto.

Figura 18 – Erro sintático na gramática do comando *printf*



Fonte: O Autor, 2024

No Algoritmo 27 apresentado, foram implementados o tratamento das mensagens de erro para alertar o usuário sobre a ausência de caracteres na gramática, resultando em erros sintáticos. No compilador as mensagens de erro ficam armazenadas no objeto *dic_control*, na chave *msg_erro* como apresentado no Algoritmo. Em caso de erro sintático o valor armazenado nesta chave do objeto é exibido no console do compilador.

Algoritmo 27 – Análise sintática do comando *while* da gramática do compilador *web*

```
funtion InstrIteracao(){
    if (globalVarC.tk === globalVarC.TKs["TKWhile"]){
        getToken();
        if (globalVarC.tk === globalVarC.TKs["TKAbreParenteses"]){
            getToken();
            if (Expressao()){
                if (globalVarC.tk === globalVarC.TKs["TKFechaParenteses"]){
                    getToken();
                    if (Instr()){
                        return true;
                    } return false
                } else {
                    globalVarC.dic_control["msg_erro"] = "Nao encontrou o \
caractere ')' no comando WHILE" + " (" + globalVarC.count_line + ", " + \
globalVarC.count_column + "\n";
                    return false;
                }
            } return false
        } else {
            globalVarC.dic_control["msg_erro"] = "Nao encontrou o \
caractere '(' no comando WHILE" + " (" + globalVarC.count_line + ", " + \
globalVarC.count_column + "\n";
```

```
23                  return false;
24              }
25          } return false;
26      }
```

Fonte: O Autor, 2024

Outra funcionalidade necessária e implementada para a análise sintática da gramática foi o *backtracking*[2] em dois pontos específicos. O *backtracking* foi utilizado nas instruções *dec2* e *expressAtrib*. Na instrução *dec2*, o *backtracking* foi necessário devido à verificação de declarações de variáveis globais e funções, que iniciam com a mesma sintaxe. No entanto, diferem-se pela presença do *token* ***TKAbreParenteses*** ("(") nas declarações de funções, enquanto nas declarações de variáveis globais, após o identificador, pode-se encontrar um *token* ***TKIgual*** ("=") para atribuição de valor, um *token* ***TKAbreColchetes*** ("[") para declaração de arranjo, um *token* ***TKVirgula*** (",") para lista de variáveis, ou um *token* ***TKPontoEVirgula*** (";") para finalizar a declaração. A instrução *expressAtrib* utiliza *backtracking* para expressões de atribuição, onde a ausência de operadores de atribuição (*operadorAtrib*) pode direcionar a análise para uma expressão condicional (*expressCondic*)[3].

5.4 ANÁLISE SEMÂNTICA

O analisador semântico desenvolvido para o compilador tem como objetivo inserir e verificar as variáveis utilizadas no programa durante a compilação, assegurando que todas as variáveis estejam declaradas e utilizadas corretamente em seus respectivos escopos. Esta etapa da compilação inclui verificações na gramática sempre que o *token TKId* (identificador de variáveis) é chamado.

De maneira geral, a análise semântica do compilador instancia as variáveis em uma tabela de símbolos no momento de sua declaração, associando-as ao seu escopo específico com as informações necessárias. Essas informações incluem o tipo da variável (inteiro, *float* ou *double*), se a variável é uma função, se é uma variável macro[4], normal, vetor ou matriz, e sua dimensão no caso de ser um vetor ou matriz. As variáveis declaradas no código em compilação são armazenadas em uma variável global chamada *tabela_de_simbolos*, no qual é um vetor onde cada posição corresponde a um escopo específico.

A definição de escopos gerencia o limite de uso das variáveis declaradas. O compilador,

[2] O *backtracking* é uma técnica que cria um ponto de salvamento dos dados de um programa, permitindo que, em caso de um retorno falso ou exceção, o programa possa retornar a esse ponto e restabelecer os dados exatamente como estavam no momento do salvamento, independentemente de quaisquer modificações ocorridas entre o salvamento e o restabelecimento.

[3] Instruções *dec2* e *expressAtrib* na gramática
dec2: ***declaracao*** | ***decFunc*** | ***decLibDefine***
expressAtrib: ***expressUnaria operadorAtrib expressAtrib*** | ***expressCondic***

[4] Variáveis macros são aquelas definidas com #define na linguagem C.

baseado no WebAlgo atual que segue a linguagem C, mantém o mesmo padrão de definição de escopos. No compilador desenvolvido, os escopos são segmentados em: escopo global (o primeiro a ser definido), escopos de funções (cada função definida possui seu próprio escopo de variáveis) e escopos em estruturas condicionais (*if* e *else*) e laços de repetição, que inicializam um novo escopo. Este controle é realizado em conjunto com a análise sintática do compilador, que identifica os *tokens* indicando esses pontos para alterações de escopo.

Na alteração dos escopos, além do controle de incremento de escopos, é necessário retornar ao escopo anterior quando uma função ou comando *if*/*else*, ou de repetição, ou retorno de função que gera um novo escopo termina. Para isso, o escopo pai do escopo atual é armazenado, permitindo um retorno correto através da hierarquia, continuando a análise sintática.

No Algoritmo 28 foi adicionado ao código do comando *WHILE* o controle de atualização de escopos, utilizando as variáveis globais *index_escopo* e *index_escopo_pai* antes de iniciar e após o retorno dos comandos internos do laço de repetição.

Algoritmo 28 – Análise semântica - Criação de escopos do comando *while* da gramática do compilador *web*

```
1  funtion InstrIteracao(){
2      if (globalVarC.tk === globalVarC.TKs["TKWhile"]){
3          getToken();
4          if (globalVarC.tk === globalVarC.TKs["TKAbreParenteses"]){
5              getToken();
6              if (Expressao()){
7                  if (globalVarC.tk === globalVarC.TKs["TKFechaParenteses"]){
8                      getToken();
9                      globalVarC.index_escopo_pai = globalVarC.index_escopo;
10                     globalVarC.index_escopo = \
11                     globalVarC.tabela_de_simbolos.length;
12                     if (Instr()){
13                         globalVarC.index_escopo = \
14                         globalVarC.index_escopo_pai;
15                         globalVarC.index_escopo_pai = \
16                         globalVarC.tabela_de_simbolos[index_escopo]
17                         ['escopo_pai'];
18                         return true;
19                     } return false
20                 } else {
21                     globalVarC.dic_control["msg_erro"] = "Nao␣encontrou␣o␣\
22 caractere␣')'␣no␣comando␣WHILE" + "␣(" + globalVarC.count_line + ",␣" + \
23 globalVarC.count_column + "\n";
24                     return false;
25                 }
26             } return false
27         } else {
28             globalVarC.dic_control["msg_erro"] = "Nao␣encontrou␣o␣\
```

```
29 caractere ')' no comando WHILE" + " (" + globalVarC.count_line + ", " + \
30 globalVarC.count_column + "\n";
31             return false;
32         }
33     } return false;
34 }
```

Fonte: O Autor, 2024

O Algoritmo 29 ilustrado apresenta um código C que realiza a leitura do tamanho de uma matriz, após lê os elementos que a comporão e em seguida realiza a soma da diagonal principal. O objtivo deste exemplo serve para exemplificar como a variável global *tabela_de_simbolos* estará durante a execução deste código no compilador desenvolvido.

Algoritmo 29 – Exemplo de código C para demonstração da tabela de símbolos

```
1  // Soma da Diagonal
2  // ESCOPO 0 - ESCOPO GLOBAL E DE DECLARACAO DE FUNCAO
3  #include <stdio.h>
4  int main() {
5      # ESCOPO 1 - VARIAVEIS DECLARADAS DENTRO DA FUNCAO
6      int N, i, j;
7
8      printf("Digite o tamanho da matriz NxN ");
9      scanf("%d", &N);
10
11     int matriz[N][N];
12
13     printf("Digite os elementos da matriz\n");
14     for (i = 0; i < N; i++)
15         // ESCOPO 2 - SEM VARIAVEIS DECLARADAS
16         for (j = 0; j < N; j++)
17             // ESCOPO 3 - SEM VARIAVEIS DECLARADAS
18             scanf("%d", &matriz[i][j]);
19
20
21     int soma_diagonal = 0;
22     for (i = 0; i < N; i++)
23         // ESCOPO 4 - SEM VARIAVEIS DECLARADAS
24         soma_diagonal += matriz[i][i];
25
26     printf("A soma da diagonal principal eh %d\n", soma_diagonal);
27 }
```

Fonte: O Autor, 2024

Este código de soma da diagonal resultará na criação de cinco escopos distintos. O primeiro escopo (posição zero do vetor *tabela_de_simbolos*) é utilizado para variáveis globais,

macros e funções, incluindo a função *main*. O segundo escopo corresponde à função *main* em si, incluindo todas as variáveis declaradas dentro da função *main* que não estejam sendo declaradas dentro de outros escopos. Estas variáveis são *N*, *i*, *j*, *matriz[N][N]* e *soma_diagonal*.

Os próximos escopos serão criados pelos laços de repetição, mas não armazenarão nenhuma variável declarada, pois os laços não possuem declarações internas. No último laço de repetição do Algoritmo 29, onde a variável *soma_diagonal* é incrementada pela *matriz[i][i]*, o *index_escopo* corresponde a 4. Nesse ponto, essas variáveis não serão encontradas diretamente no escopo atual.

Para resolver isso, a função *verifica_variavel_declarada_em_escopos* é utilizada no compilador. Esta função procura as variáveis em escopos hierarquicamente ligados ao escopo atual. No exemplo, o escopo do laço de repetição tem como escopo pai o escopo 1, que por sua vez tem como escopo pai o escopo 0. A função percorre toda a hierarquia de escopos, do mais recente ao mais antigo, para encontrar a declaração das variáveis. Caso contrário, um erro será gerado indicando que a variável não está declarada no mesmo escopo. A representação da tabela de símbolos deste exemplo é ilustrada na Algoritmo 29.

Figura 19 – Tabela de Símbolos do Algoritmo 29

ESCOPO: 0					
ESCOPO PAI: 0					
Variavel	**Tipo**	**Matriz ou Vetor**	**Dimensão**	**Define**	**É função**
main	int		0	Falso	Verdadeiro
ESCOPO: 1					
ESCOPO PAI: 0					
Variavel	**Tipo**	**Matriz ou Vetor**	**Dimensão**	**Define**	**É função**
N	int		0	Falso	Falso
i	int		0	Falso	Falso
j	int		0	Falso	Falso
matriz	int	Matriz	2	Falso	Falso
soma_diagonal	int		0	Falso	Falso
ESCOPO: 2					
ESCOPO PAI: 1					
Sem Declaração de Variável					
ESCOPO: 3					
ESCOPO PAI: 2					
Sem Declaração de Variável					
ESCOPO: 4					
ESCOPO PAI: 1					
Sem Declaração de Variável					

Fonte: O Autor, 2024

5.5 GERAÇÃO DE CÓDIGO DE TRÊS ENDEREÇOS (C3E)

Após detalhar as análises e concluir as verificações léxicas, sintáticas e semânticas, a fase seguinte no compilador é a geração de código intermediário. No caso, o compilador gera código de três endereços para permitir a execução pela máquina virtual. A geração desse código intermediário ocorre durante a execução da análise sintática e envolve a criação de instruções em pontos específicos do código, principalmente em expressões com terminais, tais como identificadores, constantes, atribuições, operadores aritméticos, lógicos e relacionais, comandos de leitura e escrita, comandos condicionais e de iteração, e comandos de controle de fluxo como *break*, *continue* e *return*. Além disso, o código intermediário inclui instruções para indicar o início de funções e mudanças de escopo, garantindo que as variáveis sejam manipuladas da mesma forma que na análise semântica durante a execução na máquina virtual.

O Algoritmo 30 ilustra a inserção da geração de código de três endereços no comando *WHILE*, apresentando as verificações necessárias para a continuação do laço de repetição conforme os parâmetros da expressão. No caso do comando *WHILE*, o código de três endereços é gerado para manipular os saltos necessários, conforme a condição estabelecida no laço. O conteúdo interno do laço é gerado pela expressão *Instr()* da gramática da *InstrIteracao()*.

Algoritmo 30 – Geração de código de três endereços do comando *while* da gramática do compilador *web*

```
1  funtion InstrIteracao(){
2      if (globalVarC.tk === globalVarC.TKs["TKWhile"]){
3          let labelInicio = newLabel();
4          geraInstrucao('', '', '', labelInicio, globalVarC.count_line, false,
5          false, true);
6          getToken();
7          if (globalVarC.tk === globalVarC.TKs["TKAbreParenteses"]){
8              getToken();
9              let result = Expressao();
10             if (result){
11                 let labelFim = newLabel();
12                 geraInstrucao('goto', result, labelFim, 'ifFalse',
13                 globalVarC.count_line, true);
14                 if (globalVarC.tk === globalVarC.TKs["TKFechaParenteses"]){
15                     getToken();
16                     globalVarC.index_escopo_pai = globalVarC.index_escopo;
17                     globalVarC.index_escopo = \
18                     globalVarC.tabela_de_simbolos.length;
19                     geraInstrucao('', '', '', '#' + index_escopo,
20                     globalVarC.count_line, false, false, true, false, true);
21                     if (Instr()){
22                         geraInstrucao('', labelInicio, '', 'goto',
23                         globalVarC.count_line, true);
24                         geraInstrucao('', '', '', labelFim,
```

```
25                              globalVarC.count_line, false, false, true);
26                              globalVarC.index_escopo = \
27                              globalVarC.index_escopo_pai;
28                              globalVarC.index_escopo_pai = \
29                              globalVarC.tabela_de_simbolos[index_escopo]
30                              ['escopo_pai'];
31                              return true;
32                          } return false
33                      } else {
34                          globalVarC.dic_control["msg_erro"] = "Nao encontrou o \
35 caractere ')' no comando WHILE" + " (" + globalVarC.count_line + ", " + \
36 globalVarC.count_column + "\n";
37                          return false;
38                      }
39                  } return false
40              } else {
41                  globalVarC.dic_control["msg_erro"] = "Nao encontrou o \
42 caractere '(' no comando WHILE" + " (" + globalVarC.count_line + \
43 ", " + count_column + "\n";
44                  return false;
45              }
46          } return false;
47 }
```

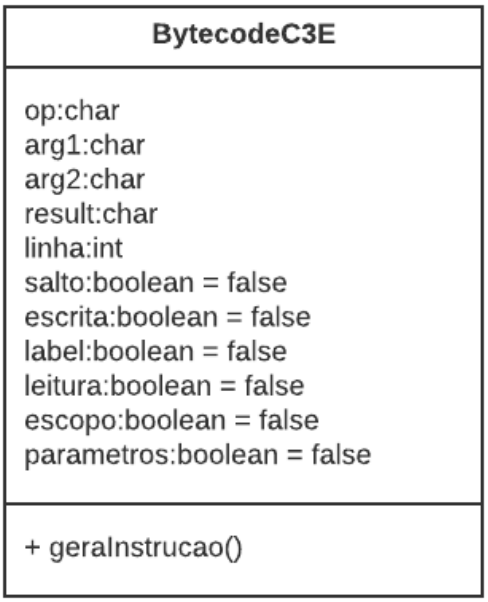
Fonte: O Autor, 2024

A geração de código intermediário foi a etapa de desenvolvimento que mais modificou a estrutura lógica da análise sintática. Isso ocorreu não apenas pela inserção das funções que criam os códigos intermediários, mas também pela alteração na estrutura do analisador sintático. A partir deste ponto, em algumas expressões da gramática, o analisador não retorna mais apenas verdadeiro ou falso, mas também variáveis, variáveis temporárias[5] ou constantes, para compor a geração do código intermediário.

No Algoritmo 30, a função *Expressao()* pode retornar *true* ou uma variável a ser utilizada na geração de código. Além disso, no caso do comando *WHILE*, onde ocorrem saltos, é possível ver a inicialização de *labels* do código intermediário através da função *newLabel()*. Por fim, a função *geraInstrucao()* é responsável por armazenar as instruções do código de três endereços em um objeto, que é incrementado em um vetor para manter as instruções ordenadas. O objeto adicionado ao vetor em toda a chamada de função *geraInstrucao* pode ser observado na Figura 20

[5] As variáveis temporárias no C3E desenvolvido são simbolizadas pelo caractere "@"seguido da letra "*t*", por exemplo: *@t1*, *@t2*, *@t3*... *@t20* designam variáveis temporárias.

Figura 20 – Objeto BytecodeC3E que será interpretado pela máquina virtual



Fonte: O Autor, 2024

A partir da lista do objetos *BytecodeC3E* a máquina virtual realizará a execução do código de três endereços. Além dos elementos op, arg1, arg2 e result que compõem a estrutura do código de três endereços outros elementos foram inseridos para implementar a execução da maquina virtual.

- Elemento ***op***: operador aritmético, lógico ou relacional (’+’, ’-’, ’/’, ’*’, ’%’, ’<’, ’>’, ’<=’, ’>=’, ’==’, ’!=’, ’&&’, ’||’, ’sqrt’ e ’abs’).
- Elemento ***arg1***: variável, variável temporária ou constante. Para chamada inicial da função *main*, *arg1* irá compor com o valor *$main*. Para retorno de função arg1 irá compor com o valor a ser retornado, exemplo instrução de três endereços *return 0*, logo arg1 terá valor 0.
- Elemento ***arg2***: variável ou constante ou *label*, exemplo instrução de três endereços *ifFalse @t30 goto L1*, logo *arg2* terá valor *L1*.
- Elemento ***result***: variável ou variável temporária, *ifFalse* para desvios condicionais, labels e mudanças de escopos.
- Elemento ***linha***: indica a linha do editor que aquela instrução pertence para fins de depuração, facilitando a exibição da linha que está em depuração no editor de texto.
- Elemento ***salto***: indica que instrução poderá realizar saldo condicional ou incondicional.

- Elemento ***escrita***: indica que instrução gerará uma saída no *console* do compilador.
- Elemento ***label***: indica que instrução é um *label*, sendo direcionado para o mesmo durante os saltos e não necessitando nenhum execução em cima da instrução.
- Elemento ***leitura***: indica que instrução gerará uma interrupção aguardando entrada de dados pelo usuário.
- Elemento ***escopo***: indica alteração de mudanças de escopo na máquina virtual
- Elemento ***parametros***: indica se chamada de função possuí parâmetros a serem passados pra função.

No Algoritmo 29, o código de três endereços gerado pelo compilador *web* correspondente pode ser observado no Algoritmo 31. Este código de três endereços apresenta cinco instruções que não estavam previstas antes do desenvolvimento. A primeira corresponde à linha do editor de texto. Cada instrução começa com um número entre colchetes no início, indicando a linha do editor para facilitar a visualização de onde o usuário está depurando durante a execução.

A primeira instrução é a linha inicial do código de três endereços, com o número 5 indicando a quantidade de escopos que o C3E a ser executado possui. A segunda instrução, na segunda linha do código de três endereços, define os escopos. Qualquer comando que comece com "#"seguido de um número inteiro positivo representa o escopo que a máquina virtual deve acessar.

A terceira instrução, na terceira linha do C3E, é um salto incondicional para a função *$main*, garantindo que a execução da máquina virtual comece na função inicial. Por último, na linha 4, há a instrução que indica o início da função *$main*.

Algoritmo 31 – Código de três endereços gerado pelo compilador correspondente ao algoritmo 29

```
1  [23] 5
2  [23] #0              //escopo=true | label=true
3  [23] goto $main      //salto=true
4  [3] $main            //label=true
5  [3] #1               //escopo=true | label=true
6  [4] N
7  [4] i
8  [4] j
9  [7] printf("Digite␣o␣tamanho␣da␣matriz␣NxN␣")   //escrita=true
10 [8] scanf("%d",N)   //leitura=true
11 [9] matriz[N][N]
12 [12] printf("Digite␣os␣elementos␣da␣matriz\n") //escrita=true
13 [12] i = 0
14 [12] L0              //label=true
```

```
15 [12] @t30 = i < N
16 [12] ifFalse @t30 goto L1  //salto=true
17 [12] goto L2        //salto=true
18 [12] L3             //label=true
19 [12] #1             //escopo=true | label=true
20 [12] i = i + 1
21 [12] goto L0        //salto=true
22 [13] L2             //label=true
23 [13] #2             //escopo=true | label=true
24 [13] j = 0
25 [13] L4             //label=true
26 [13] @t56 = j < N
27 [13] ifFalse @t56 goto L5  //salto=true
28 [13] goto L6        //salto=true
29 [13] L7             //label=true
30 [13] #2             //escopo=true | label=true
31 [13] j = j + 1
32 [13] goto L4        //salto=true
33 [14] L6             //label=true
34 [14] #3             //escopo=true | label=true
35 [16] scanf("%d",matriz[i][j])  //leitura=true
36 [17] goto L7        //salto=true
37 [17] L5             //label=true
38 [17] #2             //escopo=true | label=true
39 [17] goto L3        //salto=true
40 [17] L1             //label=true
41 [17] #1             //escopo=true | label=true
42 [17] soma_diagonal
43 [17] soma_diagonal = 0
44 [18] i = 0
45 [18] L8             //label=true
46 [18] @t105 = i < N
47 [18] ifFalse @t105 goto L9  //salto=true
48 [18] goto L10       //salto=true
49 [18] L11            //label=true
50 [18] #1             //escopo=true | label=true
51 [18] i = i + 1
52 [18] goto L8        //salto=true
53 [19] L10            //label=true
54 [19] #4             //escopo=true | label=true
55 [19] soma_diagonal = soma_diagonal + matriz[i][i]
56 [21] goto L11       //salto=true
57 [21] L9             //label=true
58 [21] #1             //escopo=true | label=true
59 //escrita=true
60 [21] printf("A␣soma␣da␣diagonal␣principal␣eh␣%d\n",soma_diagonal)
61 [22] #1             //escopo=true | label=true
```

```
62 [22] return 0        //salto=true
```

Fonte: O Autor, 2024

5.6 MÁQUINA VIRTUAL BASEADA EM REGISTRADORES VIRTUAIS

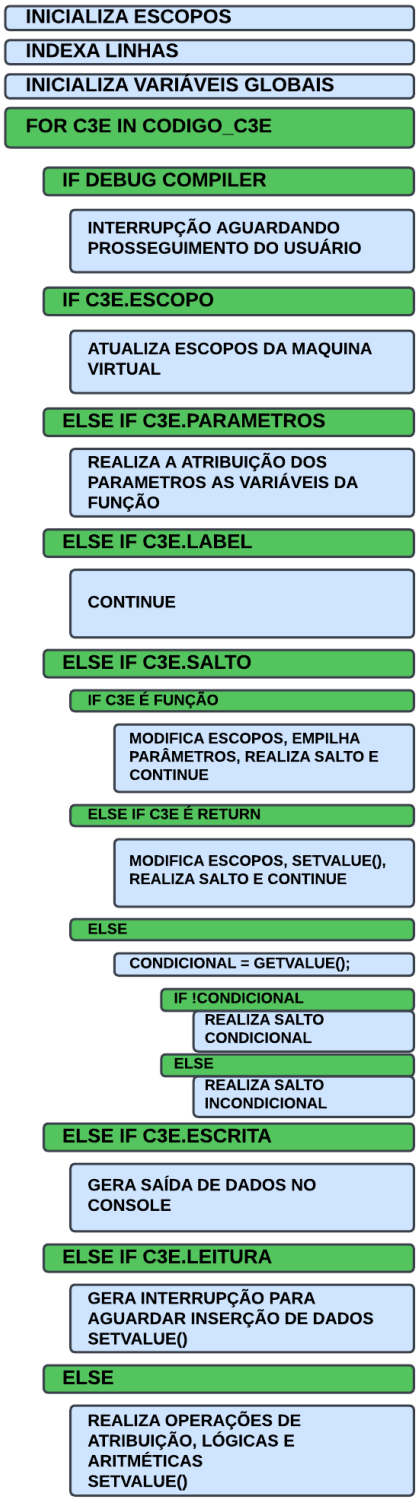
O desenvolvimento da máquina virtual foi realizado de forma completamente independente do compilador, tratando-a como um programa distinto. Essa abordagem visou garantir que a máquina virtual pudesse simplesmente receber e executar o código de três endereços (C3E) gerado pelo compilador, sem dependências adicionais. A criação de um compilador que utiliza uma máquina virtual oferece uma grande flexibilidade para expansão, pois basta gerar o C3E conforme os requisitos da máquina virtual para que o programa seja executado corretamente.

O compilador *web* foi projetado com a máquina virtual baseada em registradores virtuais. Após a conclusão da análise sintática, que inclui as análises léxica e semântica e a geração do código de três endereços, esse código intermediário é retornado pela função do compilador. Em seguida, a função da máquina virtual é chamada, recebendo a lista de objetos ***BytecodeC3E***. A máquina virtual então percorre cada linha dessa lista, executando o programa conforme as instruções lidas.

Para detalhar o funcionamento da máquina virtual, é importante destacar que, sendo ela baseada em registradores, o armazenamento e a manipulação dos valores das variáveis do código de três endereços ocorrem em registradores virtuais. Esses registradores são armazenados em uma lista de objetos, onde cada posição da lista corresponde ao escopo em que o registrador foi declarado. O gerenciamento de escopos na máquina virtual segue o mesmo princípio descrito na Seção 5.4, garantindo que cada variável seja acessada e manipulada de acordo com seu escopo específico.

A Figura 21 ilustra de forma sintética o funcionamento da máquina virtual.

Figura 21 – Estrutura da máquina virtual



Fonte: O Autor, 2024

A seguir, será detalhado o passo a passo de cada funcionalidade presente na Figura 21.

- Inicialização dos Escopos: a partir da primeira linha do código de três endereços, que indica a quantidade de escopos presentes no código, a lista de objetos que formarão os re-

gistradores é inicializada. Cada posição desta lista corresponderá a um escopo específico.

- Indexação das Linhas: esta etapa visa armazenar todos os labels do programa em um objeto. Durante a execução do código de três endereços, essa indexação permite que a máquina virtual realize saltos para os labels e funções específicas conforme determinado pela execução, conforme o label armazenado no objeto.
- Incialização de Variáveis Globais: esta etapa inicializa todas as variáveis e funções declaradas no escopo 0 (escopo global) do código de três endereços.

Com as inicializações realizadas, o código de três endereços poderá ser executado a partir do laço de repetição ilustrado na imagem. A partir deste ponto, os elementos booleanos presentes no objeto *BytecodeC3E* facilitarão a execução dos testes, eliminando a necessidade de uma compilação adicional para verificar a sintaxe das instruções.

- *Debug Compiler*: esta opção é utilizada quando o usuário executa o programa em modo de depuração. Ao iniciar a execução do código em modo de depuração, a *flag debug_compiler* é ativada. A cada linha do código de três endereços, a máquina virtual pausa e aguarda a interação do usuário com o botão "próximo passo"para continuar a execução. Essa pausa é necessária porque a função da máquina virtual é assíncrona. Assim, o comando *await* é usado junto com uma *Promise*, ativada por eventos, para indicar à máquina virtual quando prosseguir com a execução.
- Escopo (C3E == #1): esta condição é realizada com o objetivo de alterar os escopos durante a execução do C3E. Em vários pontos do código intermediário podem haver mudanças de escopo, seja por entrada em função, estrutura condicional, estrutura de repetição ou retorno de função.
- Parâmetros (C3E == a,b): esta condição é executada para carregar os parâmetros enviados à função nas determinadas variáveis.
- Label (C3E == L1): esta condição apenas continua a execução do código de três endereços para a próxima instrução.
- Salto (goto $main | return 0| ifFalse @t1 goto L2): Os saltos na máquina virtual podem ocorrer em três situações:
  - Chamada de função: Quando a instrução é uma chamada de função, os parâmetros, se houver, são empilhados junto com as informações atuais da máquina virtual, como escopo, escopo pai, índice do laço de repetição e o identificador da função sendo chamada. Em seguida, o índice do laço é direcionado para o índice da função, utilizando o objeto criado na indexação das linhas.

  - Retorno de função: No retorno de uma função, essas informações adicionais são desempilhadas para que o ponto de execução do código de três endereços retorne para o ponto imediatamente após a chamada da função. Isso permite recuperar o índice do laço de repetição da execução do C3E, os escopos e o identificador da função, que terá o valor retornado atribuído a ela.
  - Saltos condicionais: Estes dependem do resultado da expressão. Se a expressão for falsa, o salto é realizado; caso contrário, o índice do laço de repetição não é alterado, e a execução sequencial continua.

- Escrita (C3E == printf("O resultado da soma é: %d", soma): esta condição resultará em uma saída de dados no console do compilador.

- Leitura (C3E == scanf("%d%d", a, b)): está condição resultará em uma interrupção da execução utilizando o comando *await* e uma *Promise* para liberação da execução, no instante que o usuário inserir algum valor numérico e teclar enter. No momento da leitura o valor lido é atribuído ao registrador da variável "a"conforme o escopo atual.

- Atribuições e Resolução de Operações (C3E == a = 1 + 2): por fim, na ausência de indicadores com a *flag* verdadeira no objeto *BytecodeC3E*, a última condição realizará operações de atribuição, aritmética, lógica e relacional, atribuindo os resultados ao registrador resultante dessas operações.

6 VALIDAÇÃO E TESTES

Para assegurar as funcionalidades do compilador e da máquina virtual, foi realizada uma série de testes. Esses testes visaram corrigir falhas encontradas durante a compilação do programa e a execução pela máquina virtual. O objetivo principal foi verificar a consistência dos dados que o compilador forneceria ao usuário, seja através de um programa executado com sucesso, retornando os valores esperados, ou de um programa que retornou erros léxicos, sintáticos ou semânticos. Além de testes de sucesso e erro, também foram realizadas simulações mais complexas para identificar erros em programas que, embora estivessem executando corretamente, deveriam gerar alertas de erros sintáticos. Esses problemas foram corrigidos e os testes foram repetidos.

Embora o compilador *web* precise apenas de um navegador para ser utilizado, ele foi submetido a diversos testes em diferentes sistemas operacionais. Foram utilizados macOS Somma, Windows 7 e Linux Mint 21. Os testes foram realizados nos navegadores Google Chrome (v126.0.6478.61 arm) e Firefox (versão 127.0). O objetivo do trabalho não foi comparar diferentes versões de navegadores ou sistemas operacionais em termos de desempenho do compilador, portanto, não foram realizadas análises de tempo de execução. No entanto, visualmente, não houve diferença notável no tempo de execução entre os navegadores e sistemas operacionais testados.

Os testes de execução do programa foram considerados bem-sucedidos quando o programa compilado não apresentou erros de compilação, gerou corretamente o código de três endereços e foi executado com sucesso pela máquina virtual. Além de testar programas que compilaram e executaram corretamente, também foram realizados testes em programas projetados para gerar erros léxicos, devido à presença de caracteres não permitidos na linguagem, erros sintáticos, pela falta de identificação de tokens específicos em determinados pontos do código, e erros semânticos, como a falta de declaração de variáveis ou variáveis já declaradas no mesmo escopo. Para a análise semântica, foram realizados testes adicionais sobre a utilização correta das variáveis em diferentes escopos. Alguns casos de testes realizados foram acrescentados nos apêndices deste trabalho.

Os casos de teste foram baseados na gramática proposta na Seção 4.2, que inclui estruturas condicionais *if* e *else*, laços de repetição *do while*, *while* e *for*, declarações de funções, operações aritméticas, lógicas e relacionais, manipulação de variáveis locais e globais, além do uso da biblioteca <*stdio.h*> e <*math.h*> utilizando as funções *power*, *abs* e *sqrt*.

Para uma validação mais rigorosa do compilador *web* desenvolvido, foram realizados testes em sala de aula[1], conforme proposto na proposta de solução. Esses testes envolveram alunos da disciplina de Programação I dos cursos de exatas da UCS. Ao todo, 20 alunos participaram, realizando exercícios focados no uso de vetores, abordando conteúdos como laços de repetição, estruturas condicionais e sequenciais, operações de entrada e saída de dados, bem como o uso de funções da biblioteca <*math.h*>, de acordo com o conteúdo programático da disciplina.

Para permitir a execução desses testes, foi realizado um deploy na aplicação Vercel[2], o que garantiu a hospedagem e execução da aplicação de forma otimizada e com a versão mais recente do código. Dessa forma, os alunos puderam acessar o compilador diretamente via navegador, sem a necessidade de instalação local, proporcionando uma experiência prática muito similar à de uma versão finalizada do sistema. A interface e a funcionalidade da aplicação, acessada pelo navegador, permitiram que os testes fossem realizados de maneira eficiente e sem contratempos, garantindo que o ambiente de desenvolvimento fosse o mais próximo possível de uma versão de produção. Os exercícios aplicados em sala de aula pode ser visualizados no Anexo A.

Alguns erros, no entanto, foram identificados durante os testes: um problema de execução relacionado à função *abs* da biblioteca <*math.h*>, que não estava operando corretamente, alertas de erro sintático que, em algumas situações, sinalizavam incorretamente uma mensagem de erro ao invés de outra e a utilização operações pré e pós fixado em um vetor ou uma matriz que não estava funcionando corretamente. Todos esses problemas foram devidamente corrigidos após os testes realizados com os alunos.

Para avaliar o desempenho, a funcionalidade e a usabilidade do compilador pelos alunos, foi aplicado um questionário no Google Forms após o uso do mesmo. O questionário consistiu em 14 perguntas, sendo 11 baseadas em uma escala linear de avaliação, com notas de 1 a 5, onde 1 representava a pior nota e 5, a melhor. As outras 3 perguntas eram discursivas, permitindo que os alunos deixassem comentários sobre o uso, sugestões de melhorias e um feedback geral sobre o compilador. A pesquisa foi realizada de forma anônima e pode ser visualizado no Apêndice E.

O questionário contou com a participação de 14 alunos, entre os 20 participantes, dos cursos de Ciência da Computação, Análise e Desenvolvimento de Sistemas, Engenharia de Software, Engenharia da Computação e Engenharia Elétrica. Os resultados demonstraram uma avaliação positiva do compilador.

Quanto à facilidade de compreensão no primeiro impacto, 85,7% dos alunos classificaram como "muito fácil"e 14,3% como "fácil". Sobre a utilização do compilador para resolver os 17 exercícios de vetores, 78,6% o consideraram "muito fácil de usar"e 21,4% "fácil de usar".

[1] Versão do *git* utilizada para testes com os alunos: 23/10/24, 18:27

[2] A Vercel é uma plataforma de hospedagem e deploy que possibilita de forma simplificada, o upload de aplicações *web* sem a necessidade da criação de uma infraestrutura de servidor.

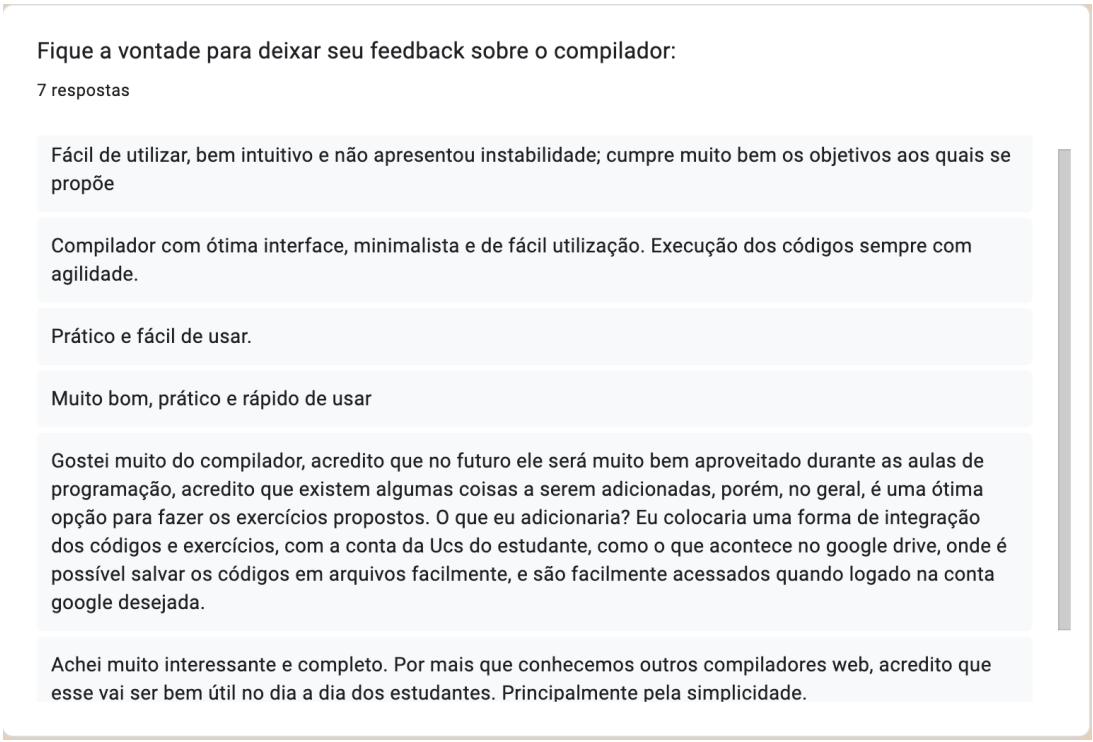
No quesito desempenho, durante a execução dos exercícios, 85,7% avaliaram como "muito rápido"e 14,3% como "rápido".

A interface do compilador também foi analisada: 57,1% classificaram como "muito intuitiva", 28,6% como "intuitiva"e 14,3% como "regular". Em relação às mensagens de erros exibidas, 50% as consideraram "muito objetivas", 21,4% "objetivas"e 28,6% "regulares". Sobre travamentos, 85,7% afirmaram que o compilador não travou nenhuma vez, enquanto 14,3% relataram algum travamento durante a execução dos exercícios.

Ao avaliar o desempenho em códigos mais longos, 64,3% classificaram como "muito rápido", 21,4% como "rápido"e 14,3% como "normal". Já o depurador do compilador foi considerado "muito bom"por 64,3% dos alunos, "bom"por 28,6% e "regular"por 7,1%.

Por fim, sobre a recomendação do compilador *web* para o aprendizado da linguagem C, 92,9% dos participantes indicaram que o recomendariam. Além das avaliações, o questionário incluiu uma solicitação de feedback dos alunos, resultando em sete comentários que podem ser visualizados na Figura 22

Figura 22 – Feedback dos alunos ao Compilador *Web*



Fonte: O Autor, 2024

6.1 TEMPO DE EXECUÇÃO DO COMPILADOR WEB COMPARADO AO WEBALGO

Embora o objetivo principal não fosse comparar o tempo de execução entre os dois compiladores, foi realizada uma análise do desempenho de ambos em dois algoritmos que exigem um processamento mais intenso. O primeiro algoritmo consistia em verificar quais números entre 0 e 1000 são primos. O tempo de execução no compilador *web* foi de 02,56 segundos, enquanto no compilador WebAlgo, o tempo foi de 0,76 segundos. Isso resultou em um tempo de execução 336,84% maior no compilador *web*. O segundo algoritmo testado gerava os primeiros 50 números de Fibonacci, e o tempo de execução foi de 01,01 segundo no compilador *web*, enquanto no WebAlgo o tempo foi de apenas 0,12 segundos, o que representa um aumento de 841% no tempo de execução no compilador *web*.

Os testes foram realizados em um sistema operacional Windows 10 Pro, utilizando um processador I7 9700k e 16GB de RAM. O compilador *web* foi executado no navegador Microsoft Edge, versão 131.0.2903.70. Para garantir a consistência dos testes, após a execução do primeiro teste no WebAlgo, o computador foi reiniciado antes de iniciar o próximo teste, para assegurar que as condições de processamento fossem as mesmas para ambos os testes. Esses resultados indicam uma diferença significativa no desempenho, com o compilador WebAlgo apresentando um tempo de execução consideravelmente menor em ambos os casos analisados.

7 CONSIDERAÇÕES FINAIS

Como relatado por Aho, Sethi e Ullman no livro Compiladores: Princípios, Técnicas e Ferramentas (AHO; SETHI; ULLMAN, 1995), o desenvolvimento de um compilador não é uma tarefa simples e exige um tempo considerável para ser executado. Esse processo envolve uma série de desafios técnicos e conceituais que demandam atenção cuidadosa em cada uma das suas fases. No caso do compilador *web* desenvolvido neste trabalho, o cenário não foi diferente. Durante todas as etapas de seu desenvolvimento, desde a criação das fases do compilador até a implementação da máquina virtual baseada em registradores virtuais, diversos obstáculos foram encontrados. Esses desafios não apenas exigiram ajustes e soluções técnicas, mas também proporcionaram um aprendizado contínuo sobre a construção de sistemas complexos e a necessidade de adaptação frente a imprevistos. Em muitos momentos, foi necessário revisar códigos desenvolvidos e realizar novas baterias de testes repetitivas para garantir a consistência do compilador em diferentes cenários de uso.

Durante a construção do compilador *web*, foi necessário realizar algumas alterações na proposta inicial. A primeira modificação ocorreu na gramática proposta, pois, durante o desenvolvimento, identificou-se situações nas produções que demandaram ajustes. Além disso, as produções relacionadas à verificação dos comandos das bibliotecas *math.h* e *stdio.h* não estavam contempladas, o que também exigiu alterações. Outra mudança significativa foi a implementação de um *worker* no compilador. Inicialmente, não foi dada a devida atenção a cenários em que o compilador poderia entrar em loop infinito ou travar devido a problemas no código produzido. Para resolver isso, o *worker* foi implementado, permitindo que a execução ocorra em uma *thread* separada da principal do navegador, evitando o travamento completo do navegador em caso de travamento na execução do compilador. Apesar dessas duas adaptações, foi possível seguir a proposta apresentada no Capítulo 4.

A utilização do compilador *web* pelos alunos, juntamente com o questionário respondido, permitiu concluir que a ferramenta está pronta para ser utilizada fora do ambiente de desenvolvimento local. A experiência reportada pelos alunos, juntamente com a baixa incidência de erros, concluí-se que o compilador *web* é abrangente o suficiente para ser utilizado em ambientes de ensino e práticas de programação, confirmando sua eficácia e estabilidade além do contexto de desenvolvimento.

O objetivo do trabalho foi plenamente alcançado, demonstrando que a utilização de uma máquina virtual baseada em registradores virtuais não apenas atendeu às expectativas, mas também abriu diversas possibilidades para a expansão do compilador. Essa abordagem permite a adição de outros compiladores para suportar novas linguagens, ampliando o aprendizado dos alunos para além da linguagem C. Para isso, basta desenvolver o compilador da linguagem desejada e implementar a geração do código de três endereços conforme o formato descrito da

máquina virtual.

7.1 TRABALHOS FUTUROS

Os próximos passos deste trabalho envolvem iniciar a análise para integrar o atual banco de dados do WebAlgo com o compilador *web* desenvolvido. Para isso, é crucial definir as tecnologias de *backend* que serão utilizadas para facilitar essa conexão. Além disso, é necessário realizar melhorias no layout do compilador para suportar funcionalidades essenciais como gerenciamento de login, gestão de perguntas, carregamento de perguntas previamente resolvidas pelos usuários e exibição do ranqueamento dos usuários em cada exercício resolvido.

Além da conexão com o banco de dados do WebAlgo, o desenvolvimento da máquina virtual neste trabalho abre novas possibilidades de expansão para outras linguagens de programação. Ao desenvolver o compilador para uma nova linguagem e gerar o código de três endereços suportado pela máquina virtual, a execução do código será realizada com êxito, ampliando o escopo e a utilidade do compilador para mais linguagens.

REFERÊNCIAS

AHO, A. V.; SETHI, R.; ULLMAN, J. D. Compiladores: Princıpios, técnicas e ferramentas. **LTC, Rio de Janeiro, Brasil**, p. 219–276, 1995.

ALVES, W. P. **Projetos de Sistemas Web: Conceitos, Estruturas, Criação de Banco de Dados e Ferramentas de Desenvolvimento**. [S.l.]: Saraiva Educação SA, 2015.

BERGIN, S.; REILLY, R. The influence of motivation and comfort-level on learning to program. Psychology of Programming Interest Group, 2005.

CEDAZO, R.; CENA, C. E. G.; AL-HADITHI, B. M. A friendly online c compiler to improve programming skills based on student self-assessment. **Computer Applications in Engineering Education**, Wiley Online Library, v. 23, n. 6, p. 887–896, 2015.

CRUZ, A. K. B. S. da *et al.* Utilizaçao da plataforma beecrowd de maratona de programaçao como estratégia para o ensino de algoritmos. In: SBC. **Anais Estendidos do XXI Simpósio Brasileiro de Jogos e Entretenimento Digital**. [S.l.], 2022. p. 754–764.

DORNELES, R. V.; JR, D. P.; ADAMI, A. G. Algoweb: a web-based environment for learning introductory programming. In: IEEE. **2010 10th IEEE International Conference on Advanced Learning Technologies**. [S.l.], 2010. p. 83–85.

EBERHARDT, C. **A simple compile-to-WebAssembly language**. 2019. Disponível em: <https://github.com/ColinEberhardt/chasm>. Acesso em: 20 out. 2023.

FLANAGAN, D.; (FIRM), O. for H. E.; SAFARI, a. O. M. C. **JavaScript: The Definitive Guide, 7th Edition**. [S.l.]: O'Reilly Media, Incorporated, 2020. ISBN 9781491952016.

GREGG, D. *et al.* The case for virtual register machines. **Science of Computer Programming**, v. 57, n. 3, p. 319–338, 2005. ISSN 0167-6423. Advances in Interpreters, Virtual Machines and Emulators. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0167642305000389>.

HAO, F. **A simple C++ interpreter written in JavaScript**. 2021. Disponível em: <https://github.com/felixhao28/JSCPP>. Acesso em: 03 out. 2023.

JESUS, A. D.; BRITO, G. S. Concepção de ensino-aprendizagem de algoritmos e programação de computadores: a prática docente. **Varia Scientia**, v. 9, n. 16, p. 149–158, 2009.

LARMAN, C. **Utilizando UML e padrões**. [S.l.]: Bookman Editora, 2007.

LIMA, F. P. d. Implementação de gerência de memória e tipos estruturados da linguagem c no webalgo. 2017. Universidade de Caxias do Sul.

LIMA, M. d. F. W. d. P.; OLIVEIRA, L. F. d. **Lista de exercícios de Programação I**. 2024. Material não publicado.

MAZIERO, C. A. Sistemas operacionais ix-máquinas virtuais. **Sistemas Operacionais**, 2008. Disponível em: <https://docplayer.com.br/222840377-Sistemas-operacionais-ix-maquinas-virtuais.html#google_vignette>. Acesso em: 19 set. 2023.

MIOTTO, F. Desenvolvimento de um compilador e uma máquina virtual de python para o ambiente webalgo. 2019. Universidade de Caxias do Sul. Disponível em: <https://repositorio.ucs.br/xmlui/handle/11338/6308>.

QIAN, Y.; LEHMAN, J. Students' misconceptions and other difficulties in introductory programming: A literature review. **ACM Trans. Comput. Educ.**, Association for Computing Machinery, New York, NY, USA, v. 18, n. 1, oct 2017. Disponível em: <https://doi.org/10.1145/3077618>.

SHI, Y. *et al.* Virtual machine showdown: Stack versus registers. **ACM Transactions on Architecture and Code Optimization (TACO)**, ACM New York, NY, USA, v. 4, n. 4, p. 1–36, 2008.

SILVA, T. R. da *et al.* Ensino-aprendizagem de programação: uma revisão sistemática da literatura. **Revista Brasileira de Informática na Educação**, v. 23, n. 01, p. 182, 2015.

WATT, C. Mechanising and verifying the webassembly specification. In: **Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs**. New York, NY, USA: Association for Computing Machinery, 2018. (CPP 2018), p. 53–65. ISBN 9781450355865. Disponível em: <https://doi.org/10.1145/3167082>.

APÊNDICE A – CASOS DE TESTE REALIZADOS NO COMPILADOR BEM-SUCEDIDOS

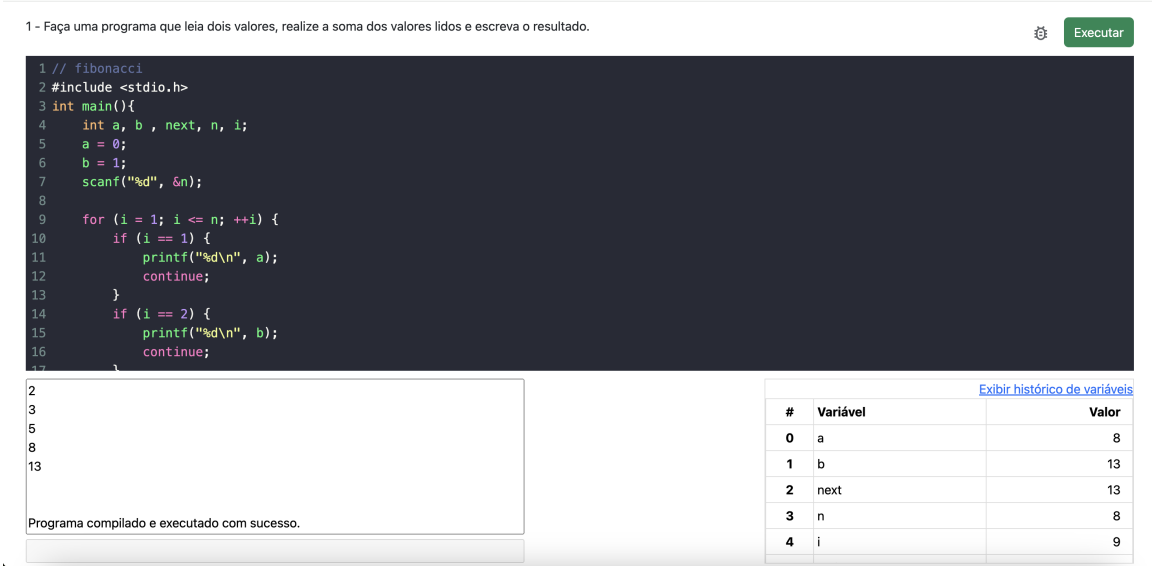
Este apêndice apresenta algoritmos compilados e executados no compilador *web* desenvolvido, demonstrando a funcionalidade do compilador na geração de código de três endereços e na execução do código intermediário pela máquina virtual. Cada estrutura essencial é exemplificada, incluindo estruturas condicionais, laços de repetição, declarações de funções, operações aritméticas, lógicas e relacionais, manipulação de variáveis locais e globais, além do uso da biblioteca <*stdio.h*>.

Algoritmo 32 – Exemplo de código C - Lê um número e escreve a sequência de Fibonacci até valor lido

```c
// fibonacci
#include <stdio.h>
int main(){
    int a, b , next, n, i;
    a = 0;
    b = 1;
    scanf("%d", &n);

    for (i = 1; i <= n; ++i) {
        if (i == 1) {
            printf("%d\n", a);
            continue;
        }
        if (i == 2) {
            printf("%d\n", b);
            continue;
        }
        next = a + b;
        a = b;
        b = next;

        printf("%d\n", next);
    }
    return 0;
}
```

A Figura 23 ilustra a compilação bem-sucedida pelo compilador. Neste exemplo o valor lido foi 8. Logo a máquina virtual executou o programa até o 8 número de Fibonacci.

Figura 23 – Exemplo no compilador - Lê um número e escreve a sequência de Fibonacci até valor lido



Fonte: O Autor, 2024

Algoritmo 33 – Código de três endereços - Lê um número e escreve a sequência de Fibonacci até valor lido

```
1  [25] 5
2  [25] #0
3  [25] goto $main
4  [3] $main
5  [3] #1
6  [4] a
7  [4] b
8  [4] next
9  [4] n
10 [4] i
11 [5] a = 0
12 [6] b = 1
13 [8] scanf("%d",n)
14 [9] i = 1
15 [9] L0
16 [9] @t27 = i <= n
17 [9] ifFalse @t27 goto L1
18 [9] goto L2
19 [9] L3
20 [9] #1
21 [9] i = i + 1
22 [9] goto L0
23 [9] L2
24 [9] #2
25 [10] @t47 = i == 1
```

```
[10] ifFalse @t47 goto L4
[10] #3
[11] printf("%d\n",a)
[12] goto L3
[14] L4
[14] #2
[14] @t69 = i == 2
[14] ifFalse @t69 goto L5
[14] #4
[15] printf("%d\n",b)
[16] goto L3
[18] L5
[18] #2
[18] @t89 = a + b
[18] next = @t89
[19] a = b
[20] b = next
[22] printf("%d\n",next)
[24] goto L3
[24] L1
[24] #1
[24] return 0
```

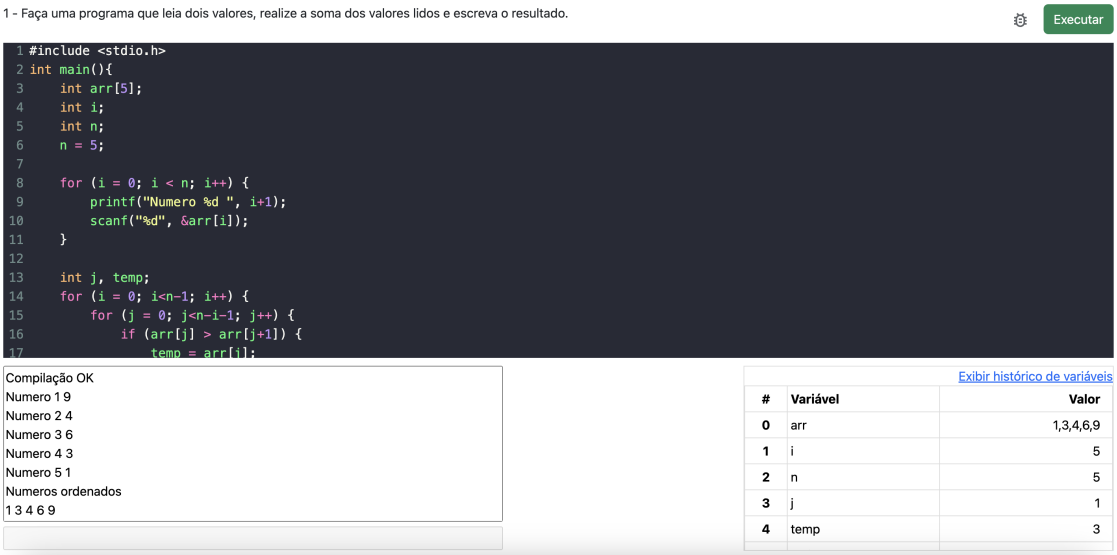
Algoritmo 34 – Exemplo de código C - Realiza a leitura de 5 números inteiros e ordena-os em ordem crescente

```c
#include <stdio.h>
int main(){
    int arr[5];
    int i;
    int n;
    n = 5;

    for (i = 0; i < n; i++) {
        printf("Numero %d ", i+1);
        scanf("%d", &arr[i]);
    }

    int j, temp;
    for (i = 0; i<n-1; i++) {
        for (j = 0; j<n-i-1; j++) {
            if (arr[j] > arr[j+1]) {
                temp = arr[j];
                arr[j] = arr[j+1];
                arr[j+1] = temp;
            }
        }
    }
```

```
23
24      printf("Numeros ordenados\n");
25      for (i = 0; i < n; i++) {
26          printf("%d ", arr[i]);
27      }
28      return 0;
29  }
```

A Figura 24 ilustra a compilação bem-sucedida pelo compilador. Neste exemplo os valores lidos foram 9,4,6,3 e 1. Logo a máquina virtual executou o programa e retornou os valores lidos ordenados de forma crescente.

Figura 24 – Exemplo no compilador - Realiza a leitura de 5 números inteiros e ordena-os em ordem crescente



Fonte: O Autor, 2024

Algoritmo 35 – Código de três endereços - Realiza a leitura de 5 números inteiros e ordena-os em ordem crescente

```
1   [29] 7
2   [29] #0
3   [29] goto $main
4   [2] $main
5   [2] #1
6   [3] arr[5]
7   [4] i
8   [5] n
9   [6] n = 5
10  [8] i = 0
11  [8] L0
12  [8] @t26 = i < n
13  [8] ifFalse @t26 goto L1
```

```
14 [8] goto L2
15 [8] L3
16 [8] #1
17 [8] i = i + 1
18 [8] goto L0
19 [8] L2
20 [8] #2
21 [9] @t45 = i + 1
22 [9] printf("Numero␣%d␣",@t45)
23 [10] scanf("%d",arr[i])
24 [13] goto L3
25 [13] L1
26 [13] #1
27 [13] j
28 [13] temp
29 [14] i = 0
30 [14] L4
31 [14] @t76 = n - 1
32 [14] @t73 = i < @t76
33 [14] ifFalse @t73 goto L5
34 [14] goto L6
35 [14] L7
36 [14] #1
37 [14] i = i + 1
38 [14] goto L4
39 [14] L6
40 [14] #3
41 [15] j = 0
42 [15] L8
43 [15] @t106 = n - i
44 [15] @t109 = @t106 - 1
45 [15] @t103 = j < @t109
46 [15] ifFalse @t103 goto L9
47 [15] goto L10
48 [15] L11
49 [15] #3
50 [15] j = j + 1
51 [15] goto L8
52 [15] L10
53 [15] #4
54 [16] @t146 = j + 1
55 [16] @t142 = arr[j] > arr[@t146]
56 [16] ifFalse @t142 goto L12
57 [16] #5
58 [17] temp = arr[j]
59 [18] @t202 = j + 1
60 [18] arr[j] = arr[@t202]
```

```
[19] @t218 = j + 1
[19] arr[@t218] = temp
[21] L12
[21] #4
[22] goto L11
[22] L9
[22] #3
[24] goto L7
[24] L5
[24] #1
[25] printf("Numeros ordenados\n")
[25] i = 0
[25] L13
[25] @t245 = i < n
[25] ifFalse @t245 goto L14
[25] goto L15
[25] L16
[25] #1
[25] i = i + 1
[25] goto L13
[25] L15
[25] #6
[26] printf("%d ",arr[i])
[28] goto L16
[28] L14
[28] #1
[28] return 0
```

Algoritmo 36 – Exemplo de código C - Lê um número e verifica se o número lido é primo

```c
// Identificador de n mero primo
#include <stdio.h>
int main(){
    int numero, i, flag;

    while (1) {
        flag = 1;
        // Solicita ao usuario que insira um numero
        printf("Digite um numero (digite -1 para sair): ");
        scanf("%d", &numero);

        // Verifica se o usuario digitou -1 para sair do loop
        if (numero == -1) break;

        if (numero <= 1){
            // Numeros menores ou iguais a 1 nao sao primos
                printf("%d nao e primo\n", numero);
        } else {
```

```
19              for (i = 2; i * i <= numero; i++) {
20                  if (numero % i == 0){
21                      flag = 0;
22                      break;
23                  }
24              }
25              if (flag){
26                  printf("%d e primo\n", numero);
27              } else {
28                  printf("%d n o e primo\n", numero);
29              }
30          }
31      }
32      return 0;
33  }
```

A Figura 25 ilustra a compilação bem-sucedida pelo compilador. Neste exemplo os valores lidos foram 117, 119, 123, 127 e -1 para parar a execução do programa. Logo a máquina virtual executou o programa até o valor lido -1 e escreveu no *console* que apenas o 127 é primo.

Figura 25 – Exemplo no compilador - Lê um número e verifica se o número lido é primo

Fonte: O Autor, 2024

Algoritmo 37 – Código de três endereços - Lê um número e verifica se o número lido é primo

```
1  [32] 10
2  [32] #0
3  [32] goto $main
4  [3] $main
5  [3] #1
6  [4] numero
7  [4] i
8  [4] flag
9  [6] L0
10 [6] ifFalse 1 goto L1
11 [6] #2
12 [7] flag = 1
13 [10] printf("Digite um numero (digite -1 para sair): ")
14 [12] scanf("\%d",numero)
15 [13] @t23 = numero == -1
16 [13] ifFalse @t23 goto L2
17 [13] #3
18 [13] goto L1
19 [15] L2
20 [15] #2
21 [15] @t35 = numero <= 1
22 [15] ifFalse @t35 goto L3
23 [15] #4
24 [16] printf("\%d n o e primo\n",numero)
25 [17] goto L4
26 [17] L3
27 [17] #5
28 [18] i = 2
29 [18] L5
30 [18] @t62 = i * i
31 [18] @t65 = @t62 <= numero
32 [18] ifFalse @t65 goto L6
33 [18] goto L7
34 [18] L8
35 [18] #5
36 [18] i = i + 1
37 [18] goto L5
38 [18] L7
39 [18] #6
40 [19] @t82 = numero % i
41 [19] @t86 = @t82 == 0
42 [19] ifFalse @t86 goto L9
43 [19] #7
44 [20] flag = 0
45 [21] goto L6
46 [23] L9
```

```
47 [23] #6
48 [24] goto L8
49 [24] L6
50 [24] #5
51 [24] ifFalse flag goto L10
52 [24] #8
53 [25] printf("%d e primo\n",numero)
54 [26] goto L11
55 [26] L10
56 [26] #9
57 [27] printf("%d n o e primo\n",numero)
58 [29] L11
59 [30] L4
60 [31] goto L0
61 [31] L1
62 [31] #10
63 [31] return 0
```

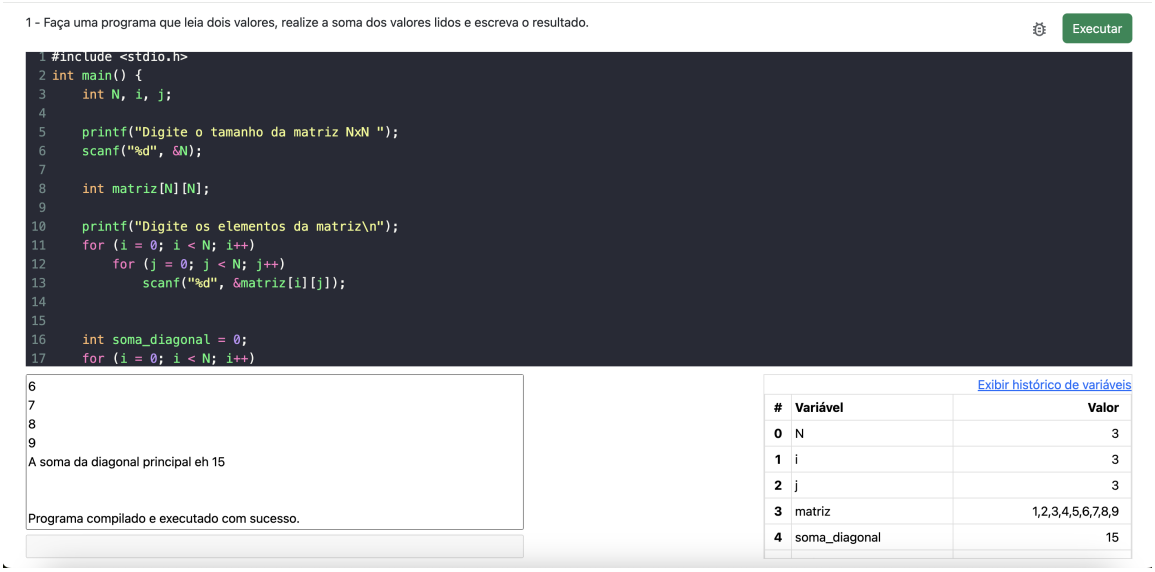
Algoritmo 38 – Exemplo de código C - Lê a dimensão da matriz, em seguinda realiza a leitura dos valores da matriz e ao final escreve a soma da diagonal principal

```
1  // Soma da Diagonal
2  #include <stdio.h>
3  int main() {
4      int N, i, j;
5
6      printf("Digite o tamanho da matriz NxN ");
7      scanf("%d", &N);
8
9      int matriz[N][N];
10
11     printf("Digite os elementos da matriz\n");
12     for (i = 0; i < N; i++)
13         for (j = 0; j < N; j++)
14             scanf("%d", &matriz[i][j]);
15
16
17     int soma_diagonal = 0;
18     for (i = 0; i < N; i++)
19         soma_diagonal += matriz[i][i];
20
21     printf("A soma da diagonal principal eh %d\n", soma_diagonal);
22     return 0;
23 }
```

A Figura 26 ilustra a compilação bem-sucedida pelo compilador. Neste exemplo o programa realiza a leitura dos elementos e insere-os em uma matriz 3x3. Após inserção realiza o

cálculo de soma da diagonal principal e escreve no *console*.

Figura 26 – Exemplo no compilador - Lê a dimensão da matriz, em seguinda realiza a leitura dos valores da matriz e ao final escreve a soma da diagonal principal



Fonte: O Autor, 2024

Algoritmo 39 – Código de três endereços - Lê a dimensão da matriz, em seguinda realiza a leitura dos valores da matriz e ao final escreve a soma da diagonal principal

```
1  [22] 5
2  [22] #0
3  [22] goto $main
4  [2] $main
5  [2] #1
6  [3] N
7  [3] i
8  [3] j
9  [6] printf("Digite␣o␣tamanho␣da␣matriz␣NxN␣")
10 [7] scanf("%d",N)
11 [8] matriz[N][N]
12 [11] printf("Digite␣os␣elementos␣da␣matriz\n")
13 [11] i = 0
14 [11] L0
15 [11] @t30 = i < N
16 [11] ifFalse @t30 goto L1
17 [11] goto L2
18 [11] L3
19 [11] #1
20 [11] i = i + 1
21 [11] goto L0
22 [12] L2
23 [12] #2
```

```
24 [12] j = 0
25 [12] L4
26 [12] @t56 = j < N
27 [12] ifFalse @t56 goto L5
28 [12] goto L6
29 [12] L7
30 [12] #2
31 [12] j = j + 1
32 [12] goto L4
33 [13] L6
34 [13] #3
35 [15] scanf("%d",matriz[i][j])
36 [16] goto L7
37 [16] L5
38 [16] #2
39 [16] goto L3
40 [16] L1
41 [16] #1
42 [16] soma_diagonal
43 [16] soma_diagonal = 0
44 [17] i = 0
45 [17] L8
46 [17] @t105 = i < N
47 [17] ifFalse @t105 goto L9
48 [17] goto L10
49 [17] L11
50 [17] #1
51 [17] i = i + 1
52 [17] goto L8
53 [18] L10
54 [18] #4
55 [18] soma_diagonal = soma_diagonal + matriz[i][i]
56 [20] goto L11
57 [20] L9
58 [20] #1
59 [20] printf("A␣soma␣da␣diagonal␣principal␣eh␣%d\n",soma_diagonal)
60 [21] #1
61 [21] return 0
```

Algoritmo 40 – Exemplo de código C - Teste realizado para chamadas de função e funcionamento correto de escopo de variáveis

```c
#include <stdio.h>
int soma(int a, int b, int c){
    return a+b+c;
}

int main(){
    int somar;
    somar = soma(soma(soma(20, 30, 70), 20, 20), 10, 20);
    printf("%d\n", somar);
    return 0;
}
```

A Figura 27 ilustra a compilação bem-sucedida pelo compilador. Neste exemplo o programa realiza a chamadas da função soma a qual espera três parâmetros em sua chamada e retornará a soma dos três parâmetros passado a ela.

Figura 27 – Exemplo no compilador - Teste realizado para chamadas de função e funcionamento correto de escopo de variáveis

Fonte: O Autor, 2024

Algoritmo 41 – Código de três endereços - Teste realizado para chamadas de função e funcionamento correto de escopo de variáveis

```
1  [11] 3
2  [11] #0
3  [11] goto $main
4  [2] $soma
5  [2] #1
6  [2] a,b,c
7  [3] @t3 = a + b
8  [3] @t6 = @t3 + c
9  [3] #1
10 [3] return @t6
11 [6] $main
12 [6] #2
13 [7] somar
14 [8] goto $soma  20,30,70
15 [8] goto $soma  soma,20,20
16 [8] goto $soma  soma,10,20
17 [8] somar = soma
18 [9] printf("%d\n",somar)
19 [10] #2
20 [10] return 0
```

APÊNDICE B – CASOS DE TESTE REALIZADOS NO COMPILADOR COM ERRO LÉXICO

Este apêndice apresenta algoritmos não compilados no compilador *web* desenvolvido devido a erros léxicos encontrados durante a compilação. Para simular erros léxicos na linguagem C basta inserir um caractere '@' no código. No caso do compilador *web*, erros léxicos que podem ocorrer serão de caracteres que não contemplam na lista de *tokens* apresentada na Seção 5.2.

Figura 28 – Erro léxico no caractere ''

```
1 #include <stdio.h>
2 int soma(int a, int b, int c){
3     return a+b+c;
4 }
5
6 int main(){
7     @;
8     int somar;
9     somar = soma(soma(soma(20, 30, 70), 20, 20), 10, 20);
10    printf("%d\n", somar);
11    return 0;
12 }
```

```
Erro léxico encontrado no caractere @ (7, 4)
```

Fonte: O Autor, 2024

Figura 29 – Erro léxico na declaração da bibioteca *stdio.h*

```c
#include <stdio.
int soma(int a, int b, int c){
    return a+b+c;
}

int main(){
    @;
    int somar;
    somar = soma(soma(soma(20, 30, 70), 20, 20), 10, 20);
    printf("%d\n", somar);
    return 0;
}
```

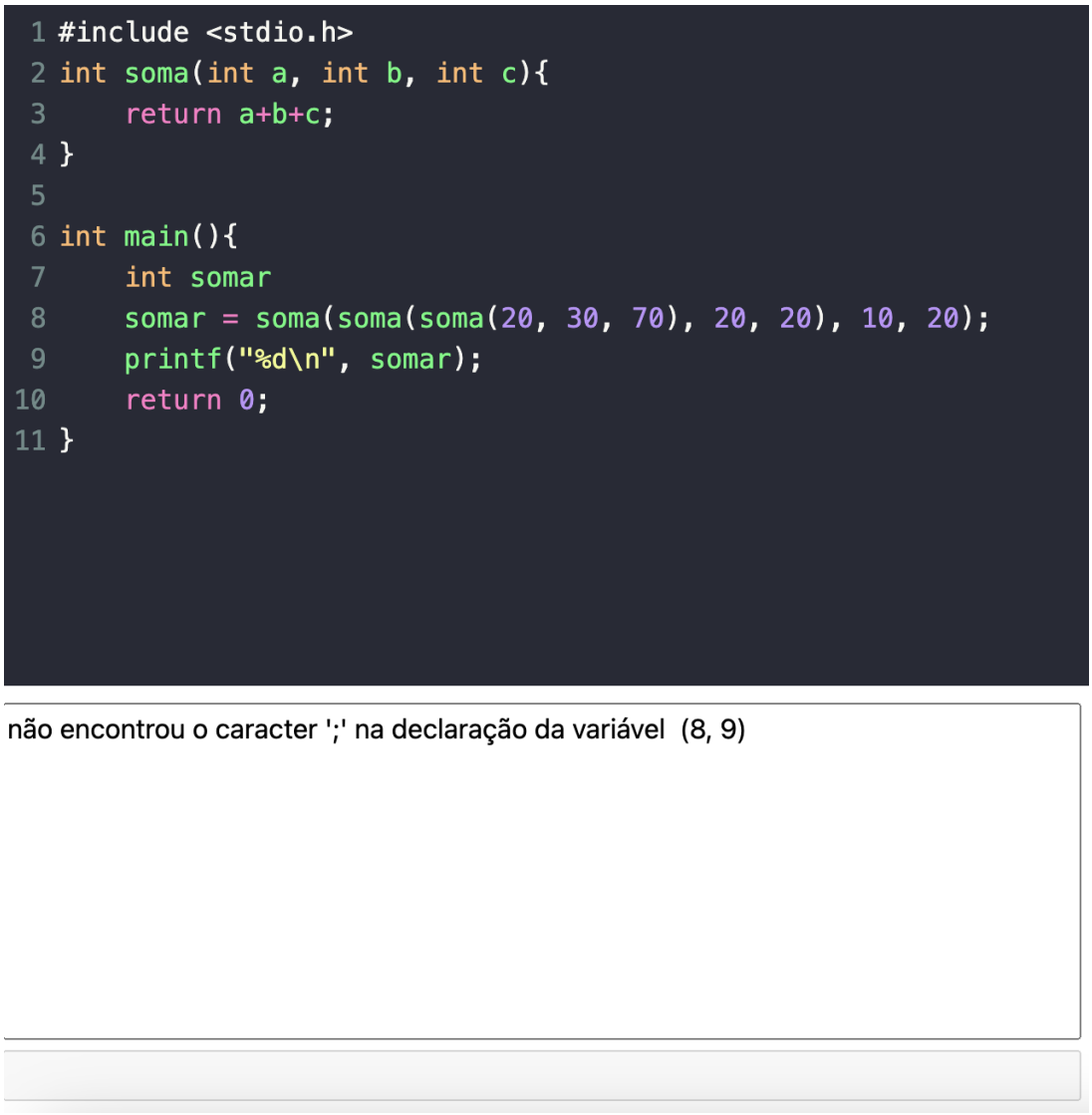
```
Erro léxico encontrado no caractere <stdio. (1, 17)
```

Fonte: O Autor, 2024

APÊNDICE C – CASOS DE TESTE REALIZADOS NO COMPILADOR COM ERRO SINTÁTICO

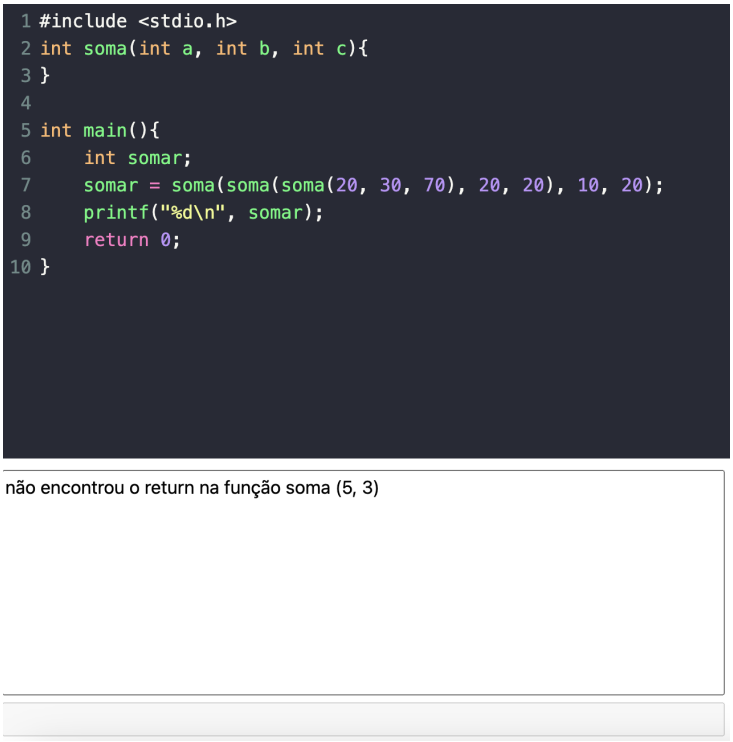
Este apêndice apresenta algoritmos não compilados no compilador *web* desenvolvido devido a erros sintáticos encontrados durante a compilação. Para simular erros sintáticos no compilador *web* basta remover um caractere esperado pela gramática ou adicionar um caractere não esperado pela gramática.

Figura 30 – Erro sintático devido a falta do caractere ';' na declaração da variável somar



Fonte: O Autor, 2024

Figura 31 – Erro sintático devido a falta do comando *return* na função soma

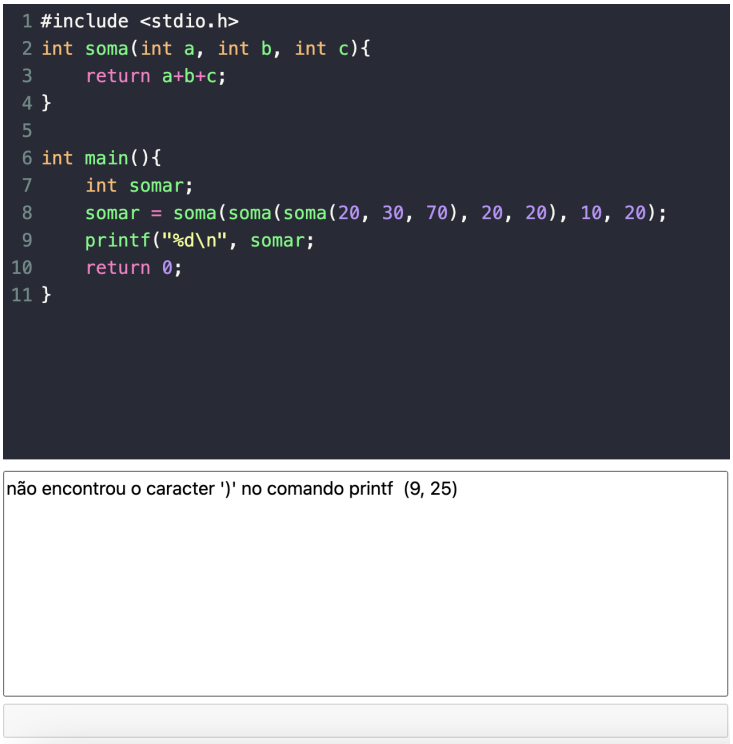


```
1 #include <stdio.h>
2 int soma(int a, int b, int c){
3 }
4
5 int main(){
6     int somar;
7     somar = soma(soma(soma(20, 30, 70), 20, 20), 10, 20);
8     printf("%d\n", somar);
9     return 0;
10 }
```

```
não encontrou o return na função soma (5, 3)
```

Fonte: O Autor, 2024

Figura 32 – Erro sintático devido a falta do caractere ')' no comando *printf*



```
1 #include <stdio.h>
2 int soma(int a, int b, int c){
3     return a+b+c;
4 }
5
6 int main(){
7     int somar;
8     somar = soma(soma(soma(20, 30, 70), 20, 20), 10, 20);
9     printf("%d\n", somar;
10    return 0;
11 }
```

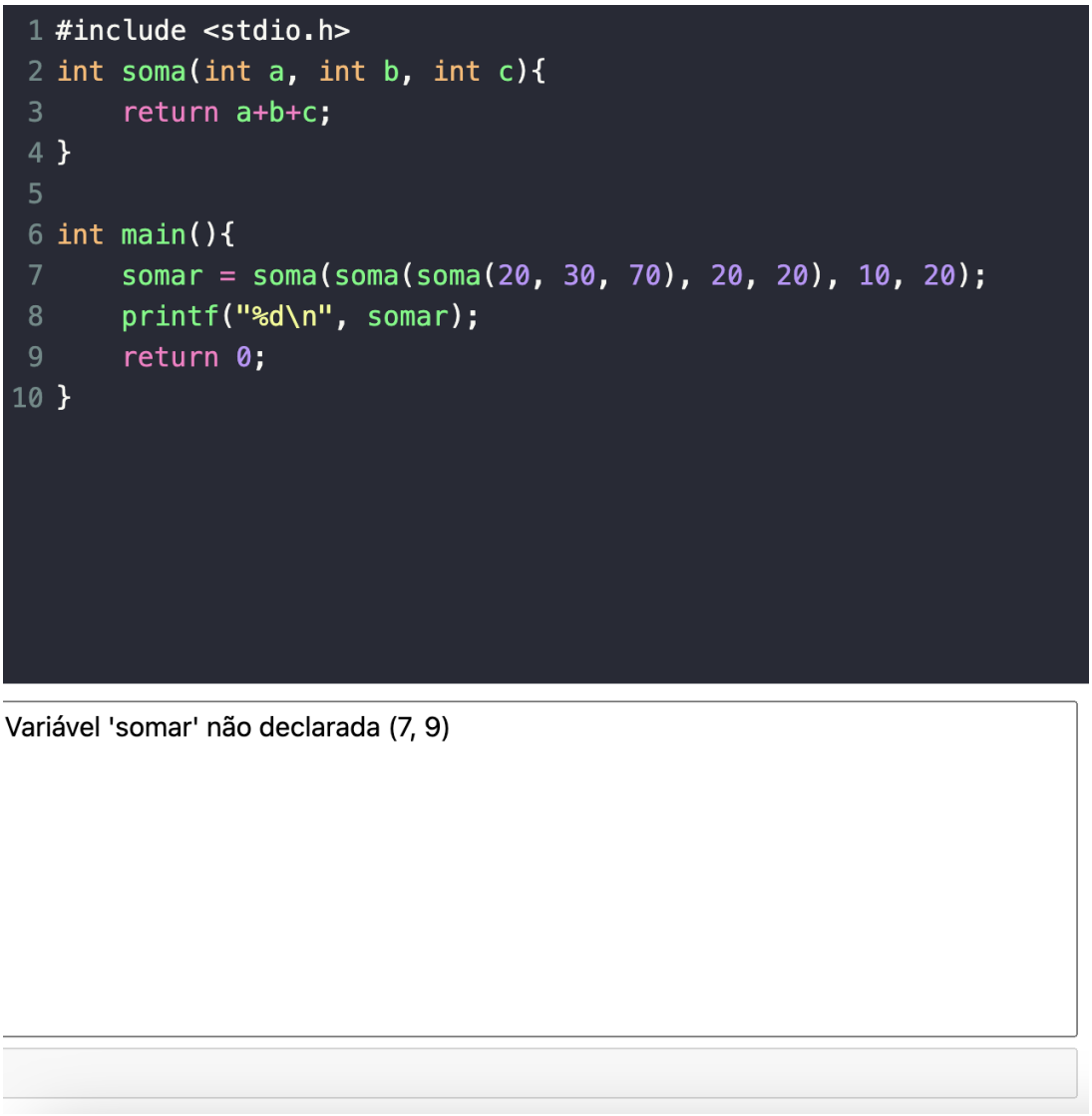
```
não encontrou o caracter ')' no comando printf  (9, 25)
```

Fonte: O Autor, 2024

APÊNDICE D – CASOS DE TESTE REALIZADOS NO COMPILADOR COM ERRO SEMÂNTICO

Este apêndice apresenta algoritmos não compilados no compilador *web* desenvolvido devido a erros semânticos encontrados durante a compilação. Para simular erros semânticos no compilador *web* basta não declarar uma variável e utiliza-lá, definir variáveis no mesmo escopo ou declarar uma variável com o mesmo nome de uma macro já declarada no #define do código.

Figura 33 – Erro semântico devido a variável somar não estar declarada



Fonte: O Autor, 2024

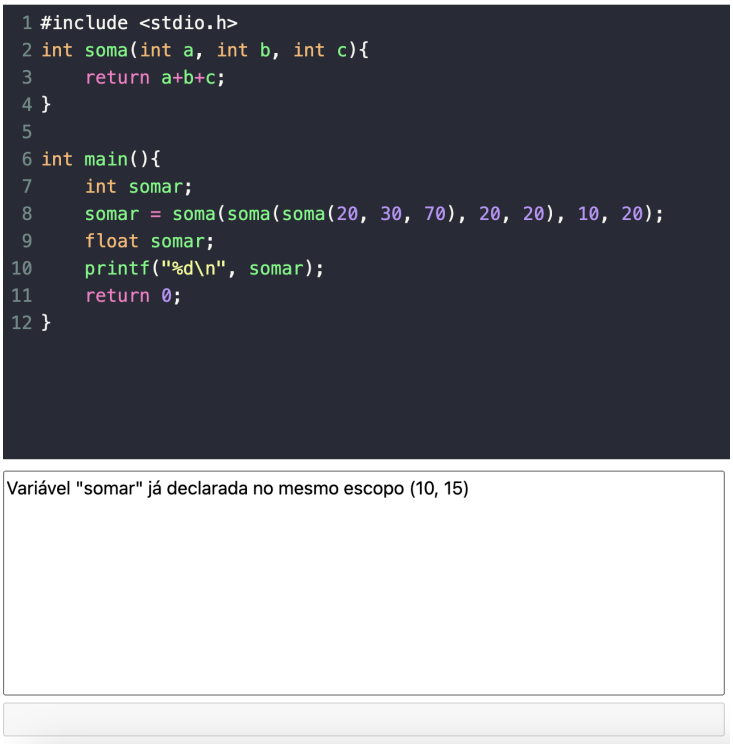
Figura 34 – Erro semântico devido a variável declarada já existir uma macro com o mesmo identificador

```c
#include <stdio.h>
#define somar 10
int soma(int a, int b, int c){
    return a+b+c;
}

int main(){
    int somar
    somar = soma(soma(soma(20, 30, 70), 20, 20), 10, 20);
    printf("%d\n", somar);
    return 0;
}
```

```
variável somar não pode ser declarada devido a macro com o mesmo nome declarada (8, 13)
```

Fonte: O Autor, 2024

Figura 35 – Erro semântico devido a variável declarada duas vezes no mesmo escopo



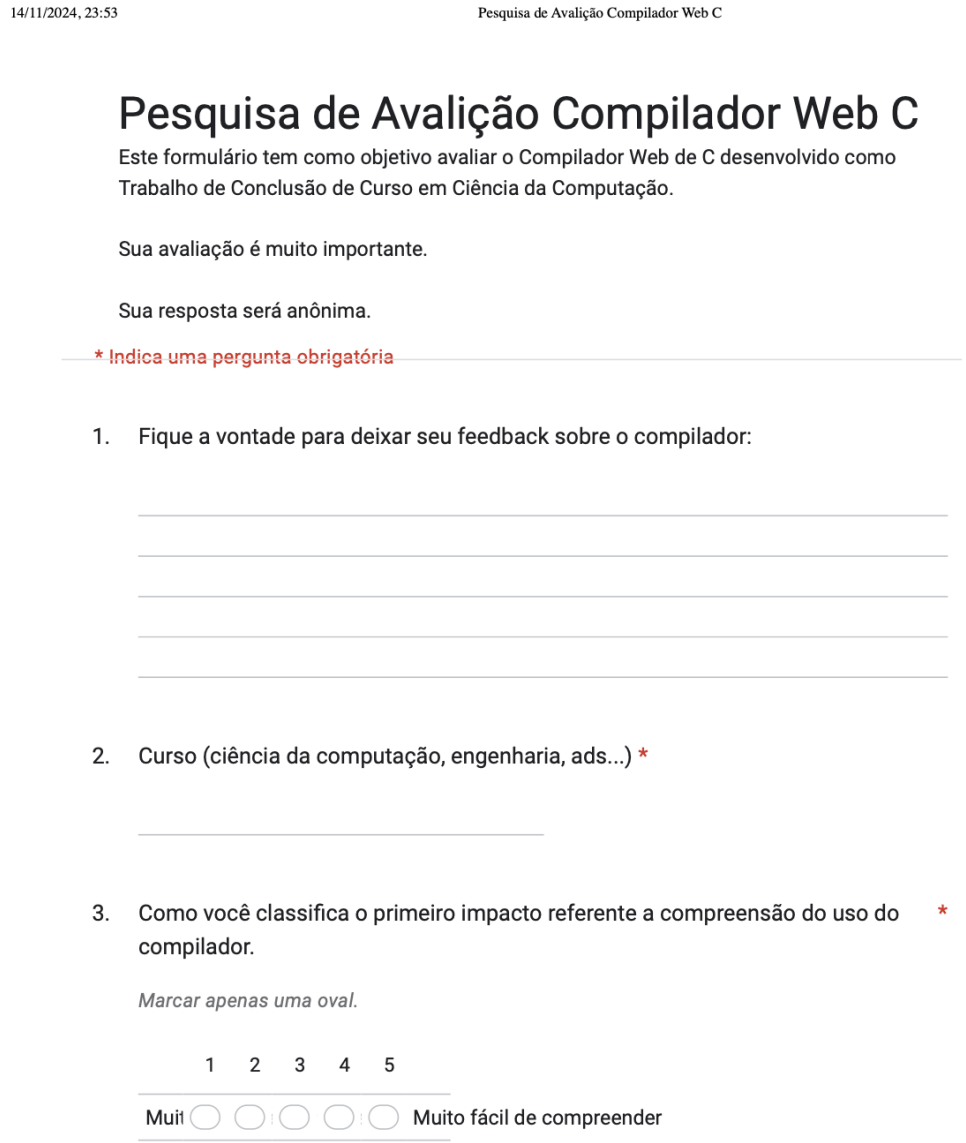
```c
#include <stdio.h>
int soma(int a, int b, int c){
    return a+b+c;
}

int main(){
    int somar;
    somar = soma(soma(soma(20, 30, 70), 20, 20), 10, 20);
    float somar;
    printf("%d\n", somar);
    return 0;
}
```

```
Variável "somar" já declarada no mesmo escopo (10, 15)
```

Fonte: O Autor, 2024

APÊNDICE E – QUESTIONÁRIO APLICADO EM SALA DE AULA COM OS ALUNOS QUE TESTARAM O COMPILADOR WEB

Figura 36 – Questionário aplicado para os alunos avaliarem o compilador web página 1



14/11/2024, 23:53 Pesquisa de Avalição Compilador Web C

Pesquisa de Avalição Compilador Web C

Este formulário tem como objetivo avaliar o Compilador Web de C desenvolvido como Trabalho de Conclusão de Curso em Ciência da Computação.

Sua avaliação é muito importante.

Sua resposta será anônima.

* Indica uma pergunta obrigatória

1. Fique a vontade para deixar seu feedback sobre o compilador:

2. Curso (ciência da computação, engenharia, ads...) *

3. Como você classifica o primeiro impacto referente a compreensão do uso do compilador. *

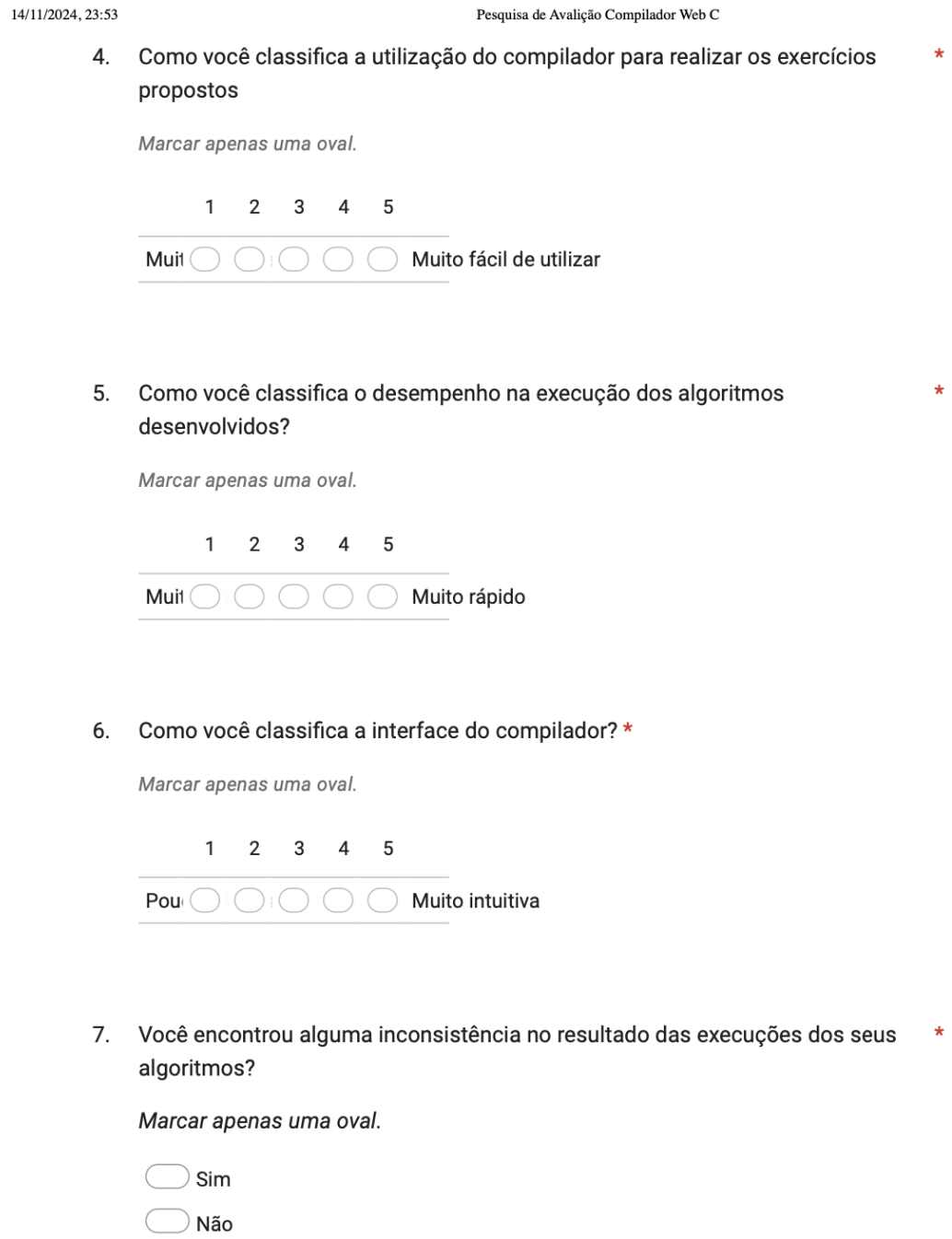
Marcar apenas uma oval.

1 2 3 4 5

Muit ◯ ◯ ◯ ◯ ◯ Muito fácil de compreender

Fonte: O Autor, 2024

Figura 37 – Questionário aplicado para os alunos avaliarem o compilador web página 2



Fonte: O Autor, 2024

Figura 38 – Questionário aplicado para os alunos avaliarem o compilador web página 3

8. Se a resposta anterior for Sim, diga qual foi a inconsistência encontrada

9. Como você classifica as mensagens de erro do compilador? *

Marcar apenas uma oval.

	1	2	3	4	5	
Pou	◯	◯	◯	◯	◯	Muito objetivas

10. O compilador apresentou travamentos frequentes? *

Marcar apenas uma oval.

	1	2	3	4	5	
Muit	◯	◯	◯	◯	◯	Não travou nenhuma vez

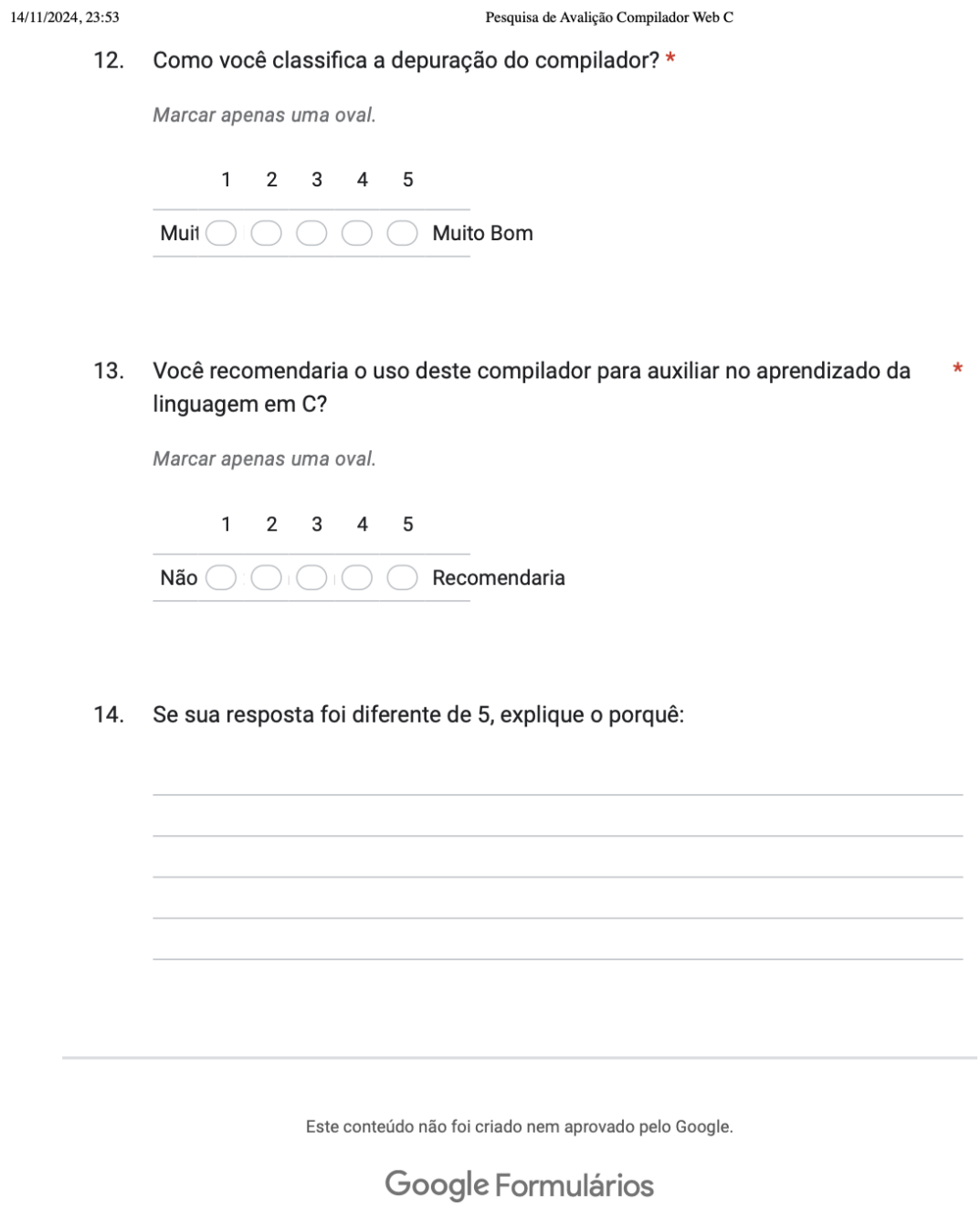
11. Como você classifica o desempenho da execução do compilador ao executar códigos mais longos? *

Marcar apenas uma oval.

	1	2	3	4	5	
Muit	◯	◯	◯	◯	◯	Muito rápido

Fonte: O Autor, 2024

Figura 39 – Questionário aplicado para os alunos avaliarem o compilador web página 4

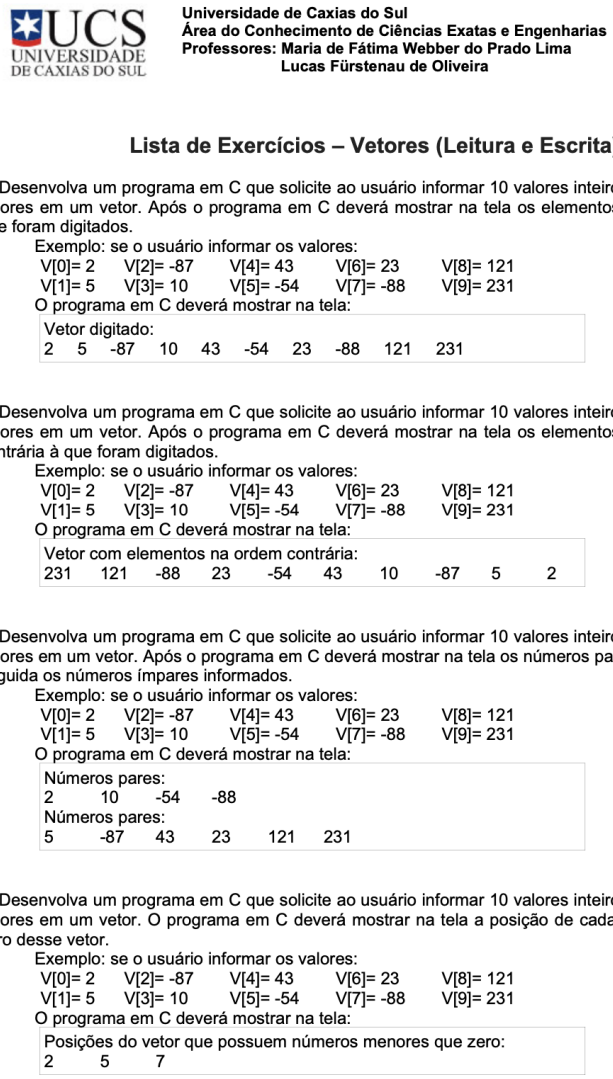


Fonte: O Autor, 2024

ANEXO A – EXERCÍCIOS REALIZADOS DURANTE TESTES COM ALUNOS EM AMBIENTE DE SALA DE AULA

Os 17 exercícios apresentados na Figura 40, Figura 41, Figura 42, Figura 43 e Figura 44 foram validados pelos alunos no compilador web desenvolvido. Esses exercícios foram elaborados pelos professores da disciplina de Programação I, Maria de Fátima Webber do Prado Lima e Lucas Fürstenau de Oliveira (LIMA; OLIVEIRA, 2024), da Universidade de Caxias do Sul.

Figura 40 – Lista de exercício utilizada pelos alunos - Página 1



UCS UNIVERSIDADE DE CAXIAS DO SUL

Universidade de Caxias do Sul
Área do Conhecimento de Ciências Exatas e Engenharias
Professores: Maria de Fátima Webber do Prado Lima
Lucas Fürstenau de Oliveira

Lista de Exercícios – Vetores (Leitura e Escrita)

1) Desenvolva um programa em C que solicite ao usuário informar 10 valores inteiros e armazene estes valores em um vetor. Após o programa em C deverá mostrar na tela os elementos do vetor na ordem que foram digitados.

Exemplo: se o usuário informar os valores:
V[0]= 2 V[2]= -87 V[4]= 43 V[6]= 23 V[8]= 121
V[1]= 5 V[3]= 10 V[5]= -54 V[7]= -88 V[9]= 231
O programa em C deverá mostrar na tela:

```
Vetor digitado:
2  5  -87  10  43  -54  23  -88  121  231
```

2) Desenvolva um programa em C que solicite ao usuário informar 10 valores inteiros e armazene estes valores em um vetor. Após o programa em C deverá mostrar na tela os elementos do vetor na ordem contrária à que foram digitados.

Exemplo: se o usuário informar os valores:
V[0]= 2 V[2]= -87 V[4]= 43 V[6]= 23 V[8]= 121
V[1]= 5 V[3]= 10 V[5]= -54 V[7]= -88 V[9]= 231
O programa em C deverá mostrar na tela:

```
Vetor com elementos na ordem contrária:
231  121  -88  23  -54  43  10  -87  5  2
```

3) Desenvolva um programa em C que solicite ao usuário informar 10 valores inteiros e armazene estes valores em um vetor. Após o programa em C deverá mostrar na tela os números pares informados e em seguida os números ímpares informados.

Exemplo: se o usuário informar os valores:
V[0]= 2 V[2]= -87 V[4]= 43 V[6]= 23 V[8]= 121
V[1]= 5 V[3]= 10 V[5]= -54 V[7]= -88 V[9]= 231
O programa em C deverá mostrar na tela:

```
Números pares:
2  10  -54  -88
Números pares:
5  -87  43  23  121  231
```

4) Desenvolva um programa em C que solicite ao usuário informar 10 valores inteiros e armazene estes valores em um vetor. O programa em C deverá mostrar na tela a posição de cada número menor que zero desse vetor.

Exemplo: se o usuário informar os valores:
V[0]= 2 V[2]= -87 V[4]= 43 V[6]= 23 V[8]= 121
V[1]= 5 V[3]= 10 V[5]= -54 V[7]= -88 V[9]= 231
O programa em C deverá mostrar na tela:

```
Posições do vetor que possuem números menores que zero:
2  5  7
```

Fonte: Adaptado de (LIMA; OLIVEIRA, 2024)

Figura 41 – Lista de exercício utilizada pelos alunos - Página 2

5) Desenvolva um programa em C que solicite ao usuário informar 10 valores inteiros e armazene estes valores em um vetor. O programa em C deverá mostrar na tela a posição de cada elemento primo desse vetor.

Exemplo: se o usuário informar os valores:

V[0]= 2 V[2]= 87 V[4]= 43 V[6]= 23 V[8]= 121
V[1]= 76 V[3]= 10 V[5]= 54 V[7]= 88 V[9]= 231

O programa em C deverá mostrar na tela:

```
Posições do vetor que possuem números primos:
0 4 6
```

6) Desenvolva um programa em C que solicite ao usuário informar 10 valores inteiros e armazene estes valores em um vetor. O programa em C deverá substituir todos os valores negativos do vetor pelo seu módulo. O programa em C deverá mostrar na tela o vetor modificado. Lembre-se que na linguagem de programa em Cs, a função a ser utilizada no cálculo do módulo é abs().

Exemplo: se o usuário informar os valores:

V[0]= 2 V[2]= -87 V[4]= 43 V[6]= 23 V[8]= 121
V[1]= 5 V[3]= 10 V[5]= -54 V[7]= -88 V[9]= 231

O programa em C deverá substituir os valores -87, -54 e -88 do vetor pelos valores 87, 54 e 88 e mostrar na tela o vetor modificado:

```
Vetor modificado:
2 5 87 10 43 54 23 88 121 231
```

7) Desenvolva um programa em C que solicite ao usuário informar 10 valores inteiros e armazene estes valores em um vetor. O programa em C deverá contar a quantidade de valores negativos. Após o programa em C deverá mostrar na tela o vetor digitado e a quantidade de valores negativos.

Exemplo: se o usuário informar os valores:

V[0]= 2 V[2]= -87 V[4]= 43 V[6]= 23 V[8]= 121
V[1]= 5 V[3]= 10 V[5]= -54 V[7]= -88 V[9]= 231

O programa em C deverá mostrar na tela:

```
Vetor digitado:
2 5 -87 10 43 -54 23 -88 121 231
Foram informados 3 números negativos.
```

8) Desenvolva um programa em C que solicite ao usuário informar 10 valores inteiros e armazene estes valores em um vetor. O programa em C deverá contar a quantidade de números primos informados. Após o programa em C deverá mostrar na tela os números primos informados e a quantidade de números primos.

Exemplo: se o usuário informar os valores:

V[0]= 2 V[2]= 87 V[4]= 43 V[6]= 23 V[8]= 121
V[1]= 76 V[3]= 10 V[5]= 54 V[7]= 88 V[9]= 231

O programa em C deverá mostrar na tela:

```
Números primos:
2 43 23
Foram informados 3 números primos.
```

9) Desenvolva um programa em C que solicite ao usuário informar 50 valores inteiros e armazene estes valores em um vetor. O programa em C deverá encontrar o maior valor informado e mostrar este valor na tela.

Exemplo: se o usuário informar os valores:

V[0]= 2 V[2]= -87 V[4]= 43 V[6]= 23 V[8]= 121 ... V[48]= 11
V[1]= 5 V[3]= 10 V[5]= -54 V[7]= -88 V[9]= 231 V[49]= 21

O programa em C deverá mostrar na tela que o maior valor informado foi 231.

Fonte: Adaptado de (LIMA; OLIVEIRA, 2024)

Figura 42 – Lista de exercício utilizada pelos alunos - Página 3

10) Desenvolva um programa em C que solicite ao usuário informar 80 valores inteiros e armazene estes valores em um vetor. O programa em C deverá encontrar o menor valor informado e a sua posição no vetor, mostrando estas informações na tela.

Exemplo: se o usuário informar os valores:

V[0]= 2	V[2]= -87	V[4]= 43	V[6]= 23	V[8]= 121	...	V[78]= 11
V[1]= 5	V[3]= 10	V[5]= -54	V[7]= -88	V[9]= 231		V[79]= 21

O programa em C deverá mostrar na tela que o menor elemento do vetor é -88 e a sua posição no vetor é 7.

11) Desenvolva um programa em C que solicite ao usuário informar 10 valores inteiros e armazene estes valores em um vetor. O programa em C deverá calcular a média dos valores informados e mostrar na tela todos os valores que estão acima da média calculada.

Exemplo: se o usuário informar os valores:

V[0]= 2	V[2]= 87	V[4]= 43	V[6]= 23	V[8]= 121
V[1]= 76	V[3]= 10	V[5]= 54	V[7]= 88	V[9]= 231

O programa em C deverá calcular a média 73,5 e mostrar os números 76, 87, 88, 121 e 231.

12) Desenvolva um programa em C que solicite ao usuário informar 15 valores inteiros e armazene estes valores em um vetor. O programa em C deverá calcular a média dos valores informados, encontrar o menor número maior que a média e mostrar na tela estas informações.

Exemplo: se o usuário informar os valores:

9,5	1,2	1,3	3,0	1,5	2,0	8,5	4,5	9,0	0,4	9,3	7,5	7,2	1,9	1,5

O programa em C deverá mostrar que a média é 4,55 e o menor número maior que a média é 7,2.

13) Desenvolva um programa em C que solicite ao usuário informar 20 valores inteiros e armazene estes valores em um vetor. Após ler e armazenar os dados no vetor, o programa em C deverá calcular e mostrar na tela a soma dos números pares e a quantidade dos números ímpares, mostrando o relatório abaixo. Caso não seja informado nenhum número ímpar, mostre a mensagem “Não existe nenhum número ímpar”. Caso não seja informado nenhum número par, mostre a mensagem “Não existe nenhum número par”.

Exemplo para um vetor de 6 posições:

V[0]= 2	V[2]= 5	V[4]= 3
V[1]= 4	V[3]= 6	V[5]= 7

Caso sejam digitados os valores acima, o programa em C deverá mostrar no final as seguintes informações:

```
Os números pares são:
número 2 posição 0
número 4 posição 1
número 6 posição 3
Soma dos pares = 12
Os números ímpares são:
número 5 posição 2
número 3 posição 4
número 7 posição 5
Quantidade dos ímpares = 3
```

14) Uma Progressão Aritmética (P.A) é uma sequência numérica em que cada termo, a partir do segundo, é igual a soma do termo anterior com uma constante. Esta constante é denominada razão (r), sendo a mesma obtida por meio da diferença de um termo da sequência pelo seu anterior.

Fonte: Adaptado de (LIMA; OLIVEIRA, 2024)

Figura 43 – Lista de exercício utilizada pelos alunos - Página 4

Por exemplo, a sequência 1, 6, 11, 16, 21, 26, 31 é uma P.A de 7 termos (termo é 1 (a1=1) e a razão é 5 (r=5).

Desenvolva um programa em C que solicite ao usuário informar 2 correspondam ao primeiro termo e a razão da PA. Gere os primeiros 20 tern armazenando os valores em um vetor de 20 posições. Após, mostre na tela os val vetor.

15) Desenvolva um programa que solicite ao usuário informar 10 valores inteir valores em um vetor. O programa deverá ler também um outro valor inteiro N e ve é um elemento do vetor. O programa deverá mostrar na tela uma mensagem infor ou não um elemento do vetor.

Exemplo: se o usuário informar os valores:

V[0]= 2 V[2]= 87 V[4]= 43 V[6]= 23 V[8]= 121

V[1]= 76 V[3]= 10 V[5]= 54 V[7]= 88 V[9]= 231

Se o usuário digitar o valor 23, o programa deverá mostrar na tela uma mer que este valor está contido no vetor. Se o usuário digitar o valor 50, o programa de uma mensagem indicando que este valor não está contido no vetor.

16) Desenvolva um programa que solicite ao usuário informar 10 valores inteiros e valores em um vetor. O programa deverá ler também um outro valor inteiro N e ver é um elemento do vetor. O programa deverá mostrar na tela uma mensagem inforr do vetor esse elemento se encontra. Se o valor estiver em mais de uma posição d todas as posições que este elemento se encontra.

Exemplo: se o usuário informar os valores:

V[0]= 2 V[2]= 87 V[4]= 43 V[6]= 23 V[8]= 121

V[1]= 23 V[3]= 10 V[5]= 23 V[7]= 88 V[9]= 231

Se o usuário digitar o valor 23, o programa deverá mostrar na tela uma mer que o valor 23 encontra-se nas posições 1,5 e 6 do vetor.

17) Desenvolva um programa que solicite ao usuário informar 10 valores inteiros e valores em um vetor. Após ler e armazenar os dados no vetor, o programa deverá (10) valores inteiros e verificar se cada um dos valores informados individualmente vetor. O programa deverá mostrar na tela, para cada um dos 10 valores informado informando se o valor N é ou não um elemento do vetor.

Exemplo: Supondo que o usuário informe os seguintes valores para o vetor

V[0]= 2 V[2]= 87 V[4]= 43 V[6]= 23 V[8]= 121

V[1]= 76 V[3]= 10 V[5]= 54 V[7]= 88 V[9]= 231

Após ele terá que Se o usuário digitar o valor 23, o programa deverá mostra mensagem indicando que este valor está contido no vetor. Se o usuário digitar o va deverá mostrar na tela uma mensagem indicando que este valor não está contido

Figura 44 – Lista de exercício utilizada pelos alunos - Página 5

Por exemplo, a sequência 1, 6, 11, 16, 21, 26, 31 é uma P.A de 7 termos (n=7), onde o primeiro termo é 1 (a1=1) e a razão é 5 (r=5).

Desenvolva um programa em C que solicite ao usuário informar 2 valores inteiros que correspondam ao primeiro termo e a razão da PA. Gere os primeiros 20 termos desta sequência, armazenando os valores em um vetor de 20 posições. Após, mostre na tela os valores armazenados no vetor.

15) Desenvolva um programa que solicite ao usuário informar 10 valores inteiros e armazene estes valores em um vetor. O programa deverá ler também um outro valor inteiro N e verificar se este valor N é um elemento do vetor. O programa deverá mostrar na tela uma mensagem informando se o valor N é ou não um elemento do vetor.

Exemplo: se o usuário informar os valores:

V[0]= 2 V[2]= 87 V[4]= 43 V[6]= 23 V[8]= 121

V[1]= 76 V[3]= 10 V[5]= 54 V[7]= 88 V[9]= 231

Se o usuário digitar o valor 23, o programa deverá mostrar na tela uma mensagem indicando que este valor está contido no vetor. Se o usuário digitar o valor 50, o programa deverá mostrar na tela uma mensagem indicando que este valor não está contido no vetor.

16) Desenvolva um programa que solicite ao usuário informar 10 valores inteiros e armazene estes valores em um vetor. O programa deverá ler também um outro valor inteiro N e verificar se este valor N é um elemento do vetor. O programa deverá mostrar na tela uma mensagem informando qual posição do vetor esse elemento se encontra. Se o valor estiver em mais de uma posição do vetor, deve mostrar todas as posições que este elemento se encontra.

Exemplo: se o usuário informar os valores:

V[0]= 2 V[2]= 87 V[4]= 43 V[6]= 23 V[8]= 121

V[1]= 23 V[3]= 10 V[5]= 23 V[7]= 88 V[9]= 231

Se o usuário digitar o valor 23, o programa deverá mostrar na tela uma mensagem indicando que o valor 23 encontra-se nas posições 1,5 e 6 do vetor.

17) Desenvolva um programa que solicite ao usuário informar 10 valores inteiros e armazene estes valores em um vetor. Após ler e armazenar os dados no vetor, o programa deverá ler também mais dez (10) valores inteiros e verificar se cada um dos valores informados individualmente estão contidos no vetor. O programa deverá mostrar na tela, para cada um dos 10 valores informados, uma mensagem informando se o valor N é ou não um elemento do vetor.

Exemplo: Supondo que o usuário informe os seguintes valores para o vetor:

V[0]= 2 V[2]= 87 V[4]= 43 V[6]= 23 V[8]= 121

V[1]= 76 V[3]= 10 V[5]= 54 V[7]= 88 V[9]= 231

Após ele terá que Se o usuário digitar o valor 23, o programa deverá mostrar na tela uma mensagem indicando que este valor está contido no vetor. Se o usuário digitar o valor 50, o programa deverá mostrar na tela uma mensagem indicando que este valor não está contido no vetor.

Fonte: Adaptado de (LIMA; OLIVEIRA, 2024)