Uma Ferramenta Educacional de Apoio ao Ensino de Compiladores e Análise da Complexidade de Algoritmos

Erik Pablo Schaefer Borela[1], **Alessandra Lima de Oliveira**[1], **Wilson Castello Branco Neto**[1], **Alexandre Perin de Souza**[1]

[1]Instituto Federal de Santa Catarina Campus Lages (IFSC)
Rua Heitor Villa Lobos, 225 – 88.506-400 – Lages – SC – Brasil

{erik.sb, alessandra.lo}@aluno.ifsc.edu.br,
{wilson.castello, alexandre.perin}@ifsc.edu.br

Abstract. *This paper presents an educational tool designed to assist in teaching compilers and analyzing algorithm complexity. The tool consists of a functional compiler for a language inspired by the C language. The compiler translates the source code into the LLVM intermediate language, which is responsible for optimization and target code generation. Through a web interface, it is possible to view all artifacts of the compilation process, such as the syntax tree and the intermediate code, as well as to visualize in detail the construction of the algorithm complexity calculation. The results of the tool evaluation carried out by students and teachers demonstrate its potential to assist in the teaching-learning process.*

Resumo. *Este artigo apresenta uma ferramenta para auxiliar o ensino de compiladores e análise da complexidade de algoritmos. A ferramenta consiste em um compilador funcional para uma linguagem inspirada na linguagem C. O compilador realiza a tradução do código-fonte para a linguagem intermediária do LLVM, que é responsável pela otimização e geração do código-alvo. Por meio de uma interface web, é possível visualizar todos os artefatos do processo de compilação, como a árvore sintática e o código intermediário, e a construção do cálculo da complexidade dos algoritmos de forma detalhada. Os resultados da avaliação da ferramenta realizada por alunos e professores demonstram o seu potencial para auxiliar no processo de ensino-aprendizagem.*

1. Introdução

A partir da popularização dos computadores nos anos 2000, diversos problemas podem ser resolvidos e facilitados por programas de computadores, desde encontrar o trajeto mais rápido de casa ao mercado (Singal e Chhillar, 2014), em soluções médicas que auxiliam em diagnósticos de doenças (Badnjević et al., 2013) e até em programas para o auxílio no processo de ensino-aprendizagem (Queiros et al., 2022). Todo programa de computador, em sua essência, é um algoritmo. De acordo com Cormen et al. (2012), um algoritmo é qualquer procedimento computacional bem definido que toma algum valor ou conjunto de valores como entrada e produz algum valor ou conjunto de valores como saída.

Algoritmos podem ser descritos através de linguagens de programação que, a partir de regras sintáticas e semânticas, os descrevem tanto para pessoas quanto para máquinas. Porém, antes de um programa poder ser executado, ele precisa ser traduzido

para um formato que possa ser compreendido pelo *hardware* do computador. Os sistemas de *software* responsáveis por essa tradução são denominados compiladores (Aho et al., 2008).

Um compilador pode ser simplificado e dividido em duas partes principais: o *front end* e o *back end*. O *front end* se concentra na compreensão do programa na linguagem-fonte, aplicando as regras léxicas, sintáticas e semânticas e transformando-o em uma representação intermediária (IR) que será utilizada posteriormente pelo *back end*. Este último é responsável pela otimização do código e mapeamento do programa para a máquina-alvo, possibilitando sua execução pelo *hardware*, conforme mostra a figura 1 (Cooper e Torczon, 2014).

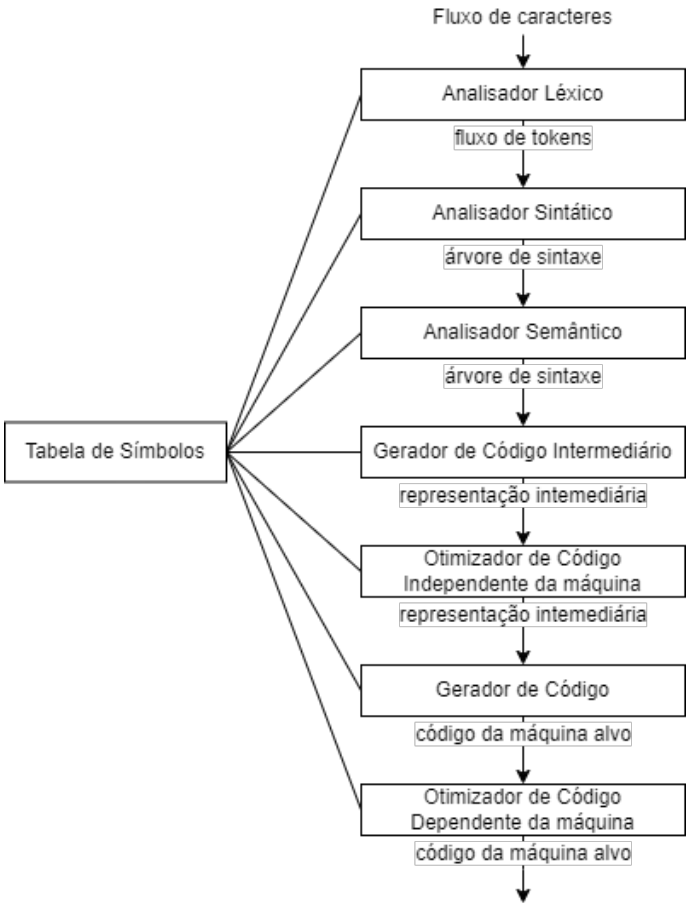


Figura 1. Fases de um compilador
Fonte: Adaptado de Aho et al. (2008)

Porém, o fato de um algoritmo resolver teoricamente um problema não implica necessariamente que ele seja aceitável na prática, pois os recursos de tempo e espaço requeridos têm grande importância. Através da análise da complexidade do código, pode-se estimar o número de operações computacionais que um algoritmo levará para executar uma determinada entrada, assim como a quantidade de memória necessária para sua execução (Toscani e Veloso, 2012). Diferentemente da análise empírica de um algoritmo, onde as medições de tempo e espaço dependem de fatores como a linguagem, o *hardware* e o conjunto de instruções da *Central Processing Unit (CPU)*, entre outros, a determinação da

complexidade de um algoritmo requer uma análise matemática do mesmo, independente de plataforma, linguagem ou tecnologia utilizada. Isso permite entender como o algoritmo se comporta à medida que o conjunto de entrada cresce e expressar uma relação com a quantidade de tempo ou espaço necessários para processar o conjunto (Backes, 2016).

Disciplinas que ensinam os conceitos de análise da complexidade de algoritmos e compiladores são comuns e fundamentais nas estruturas curriculares de cursos superiores na área de computação. Essas disciplinas trabalham conceitos teóricos, abstratos e complexos, que proporcionam diversos benefícios para o aprimoramento da qualidade de código escrito pelos programadores, além de serem importantíssimas para a formação de um profissional. No entanto, é comum que os professores enfrentem dificuldades no processo de ensino-aprendizagem, sobretudo devido à escassez de recursos que apresentem o conteúdo de forma didática, com recursos gráficos, que permitam executar diversos exemplos e apresentem soluções interativas, tornando a aprendizagem dos conceitos mais fácil e completa (Gramond e Rodger, 1999).

Este trabalho tem como objetivo desenvolver uma ferramenta para auxiliar no ensino de compiladores e análise da complexidade de algoritmos. Para isto foram usadas ferramentas empregadas na construção de compiladores reais, como o *ANTLR 4* [1] para a geração de analisadores léxicos e sintáticos a partir de uma gramática e o *LLVM* [2] servindo como otimizador e *back end*. Esta última ferramenta é amplamente reconhecida por ser a base de implementação de linguagens populares, como o compilador *Clang* para *C* e *C++*, *Rust*, *Swift*, entre outras. A ferramenta proposta é acessível através de um site na *Web*, onde o usuário pode inserir o código e executá-lo, além de consultar a análise de complexidade em um formato de fácil entendimento.

Os objetivos específicos para a execução desse projeto consistem em:

- Desenvolver as regras léxicas, sintáticas e semânticas da linguagem de programação usada pelo compilador.
- Implementar o compilador e o analisador da complexidade computacional com base na linguagem desenvolvida.
- Criar um sistema *Web* que permita ao usuário compilar e executar o código fornecido, assim como visualizar de forma interativa os passos da análise da complexidade computacional.
- Avaliar a contribuição do sistema desenvolvido para o ensino-aprendizagem da análise da complexidade de algoritmos.

Em relação à metodologia, este trabalho possui 4 etapas. A primeira etapa consiste em realizar um estudo sobre o funcionamento das ferramentas *ANTLR 4* e *LLVM*. Para isso, foram feitas pesquisas em livros e artigos, com o desenvolvimento prático de exemplos de uso. Após o entendimento das ferramentas, as definições das regras léxicas, sintáticas e semânticas da linguagem de programação compreendida pelo compilador foram elaboradas.

A segunda etapa consistiu no desenvolvimento do compilador e do analisador de complexidade baseado nas definições da linguagem criada. Ambas as implementações

[1] `https://www.antlr.org/`
[2] `https://llvm.org/`

foram feitas usando a linguagem *Java* com o *ANTLR 4* para a compreensão do código a ser compilado e analisado. Para a etapa da compilação, o código IR em forma textual é gerado e passado para o *LLVM* concluir a otimização e geração do executável. Na análise da complexidade de algoritmos, um arquivo *JavaScript Object Notation (JSON)* é gerado contendo a informação do custo computacional de cada linha e bloco de código.

A terceira etapa foi a criação de um sistema *Web*, que permite ao usuário executar o código compilado, visualizar de forma interativa os passos da análise da complexidade computacional e consultar informações da compilação do código, como o código IR gerado e a árvore sintática. Para isso, integrou-se o compilador anteriormente criado com o *framework Spring Boot* [3] para formar uma *Application Programming Interface (API)* e poder se comunicar com o ambiente *Web*. Para o desenvolvimento da aplicação *Web* utilizou-se usar o *framework Vue* [4] para a criação das interfaces com o usuário, com o componente *CodeMirror* [5] para a criação do editor de texto interativo.

Para a avaliação do projeto, a aplicação foi disponibilizada para um conjunto de alunos e professores, com o intuito de testarem a ferramenta desenvolvida e coletar dados sobre o desempenho obtido na facilitação do ensino-aprendizagem dos conceitos de compiladores e de análise de complexidade, assim como obter *feedbacks* sobre melhorias na usabilidade e aparência do sistema.

Pela natureza, este trabalho se classifica como pesquisa aplicada. Do ponto de vista da abordagem dos problemas, este trabalho se classifica como qualitativa e quantitativa. Pelos objetivos, este trabalho se classifica como pesquisa exploratória. Dos procedimentos técnicos, este trabalho se classifica como bibliográfica.

Este artigo está dividido em 5 seções. Após a introdução, a seção 2 apresenta os principais trabalhos similares encontrados. A seção 3 descreve as decisões tomadas na implementação do trabalho, incluindo a gramática da linguagem de programação desenvolvida, as estruturas utilizadas tanto pelo compilador quanto pelo analisador da complexidade e a arquitetura do seu funcionamento. Na seção 4 são apresentadas as avaliações e resultados do projeto obtidos dos usuários testadores. Finalizando com a seção 5, são apresentadas as conclusões obtidas do trabalho em relação ao problema dado.

2. Trabalhos Relacionados

Essa seção descreve os trabalhos similares encontrados, que serviram como base no planejamento e no desenvolvimento dos recursos do trabalho proposto. A seção é dividida em duas partes, a seção 2.1 descreve os trabalhos similares relacionados à compiladores enquanto a seção 2.2 descreve os trabalhos similares relacionados à análise da complexidade de algoritmo.

2.1. Compiladores

Para a obtenção dos trabalhos similares de compiladores, uma revisão bibliográfica foi realizada nas plataformas *SBC-OpenLib* e *IEEE Xplorer*. As chaves de pesquisa utilizadas foram "compilador educação", "*compiler teaching*" e "*compiler learning*". Feita a

[3] `https://spring.io/projects/spring-boot`
[4] `https://vuejs.org/`
[5] `https://codemirror.net/`

pesquisa dos trabalhos relacionados, os dez primeiros artigos de cada plataforma foram selecionados para terem os seus resumos lidos e os mais relevantes foram selecionados para uma leitura completa. Dos trabalhos lidos por completo, foram escolhidos três que ficaram mais próximos dos objetivos propostos por este trabalho para serem detalhados nessa seção.

LISA, sigla para *Language Implementation System based on Attribute Grammars*, apresentado em Mernik e Zumer (2003), é um ambiente de desenvolvimento criado na linguagem Java, voltado ao ensino da construção de compiladores. A ferramenta permite que uma linguagem de programação seja especificada utilizando expressões regulares para a análise léxica, a notação *Backus-Naur Form* (*BNF*) para a análise sintática e possibilitando a definição de regras semânticas usando atributos de gramática.

O ambiente do programa possui recursos visuais para demonstrar o funcionamento de cada etapa da compilação. Na análise léxica, a ferramenta gera um autômato finito determinístico e exibe uma animação do processo acontecendo para cada caractere lido, como pode ser visto na figura 2. Já na análise sintática, o programa mostra passo a passo a construção da árvore de derivação a partir dos *tokens* obtidos na etapa anterior. Por fim, na análise semântica, a ferramenta exibe uma árvore de avaliação, similar à mostrada na etapa sintática, mas que também contém os atributos utilizados e modificados no passo a passo da avaliação da árvore.

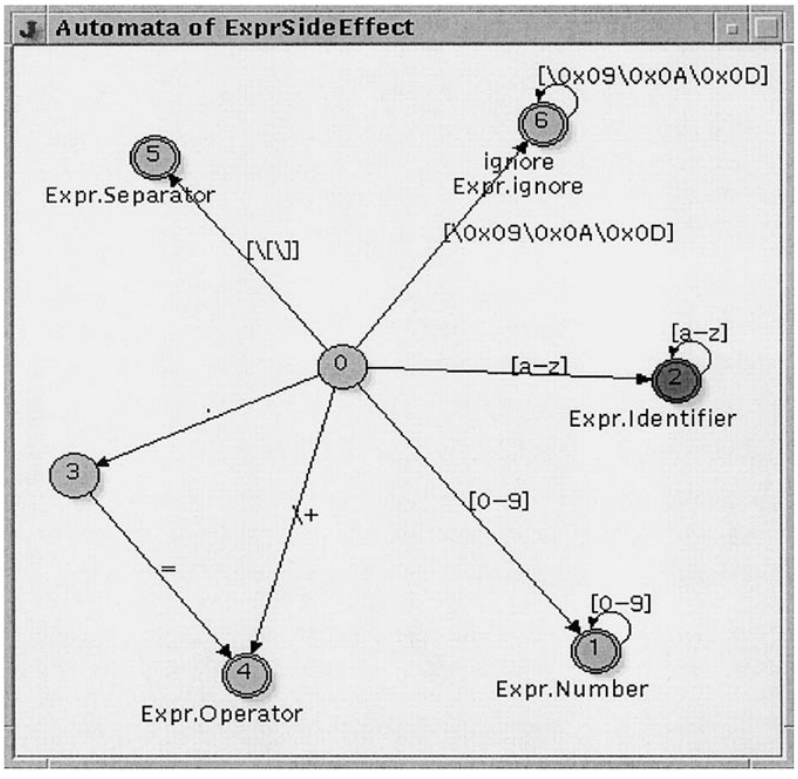


Figura 2. Autômato finito determinístico gerado pelo *LISA* para explicação dos passos de análise léxica.

Fonte: (Mernik e Zumer, 2003)

Verto, detalhada em Scheider et al. (2005), é uma ferramenta desenvolvida na linguagem Java, focada no aprendizado das fases de geração de código intermediário e

código alvo. A ferramenta aceita uma linguagem de programação com palavras-chave em português e utiliza o código alvo implementado pela máquina virtual hipotética *Cesar*, desenvolvida na Universidade Federal do Rio Grande do Sul (UFRGS) (Weber, 2001).

A interface do programa permite ao usuário escrever o código a ser compilado e apresenta de forma textual um registro detalhado dos processos ocorridos nas fases de análise léxica, sintática e semântica. O registro descreve, por exemplo, quais *tokens* foram identificados, quais regras sintáticas foram derivadas e quais comparações semânticas foram feitas. Também é possível visualizar a tabela de símbolos, que contém informações como endereço de memória, escopo e tipo do símbolo. Para o código intermediário, é apresentado o resultado da compilação na linguagem *MacroAssembler* do *Verto*, uma versão próxima à utilizada no *Cesar* que contém abstrações e comentários descrevendo o que cada instrução realiza, como pode ser visto na figura 3. Por fim, o código alvo é gerado e pode ser executado utilizando a ferramenta *Cesar*.

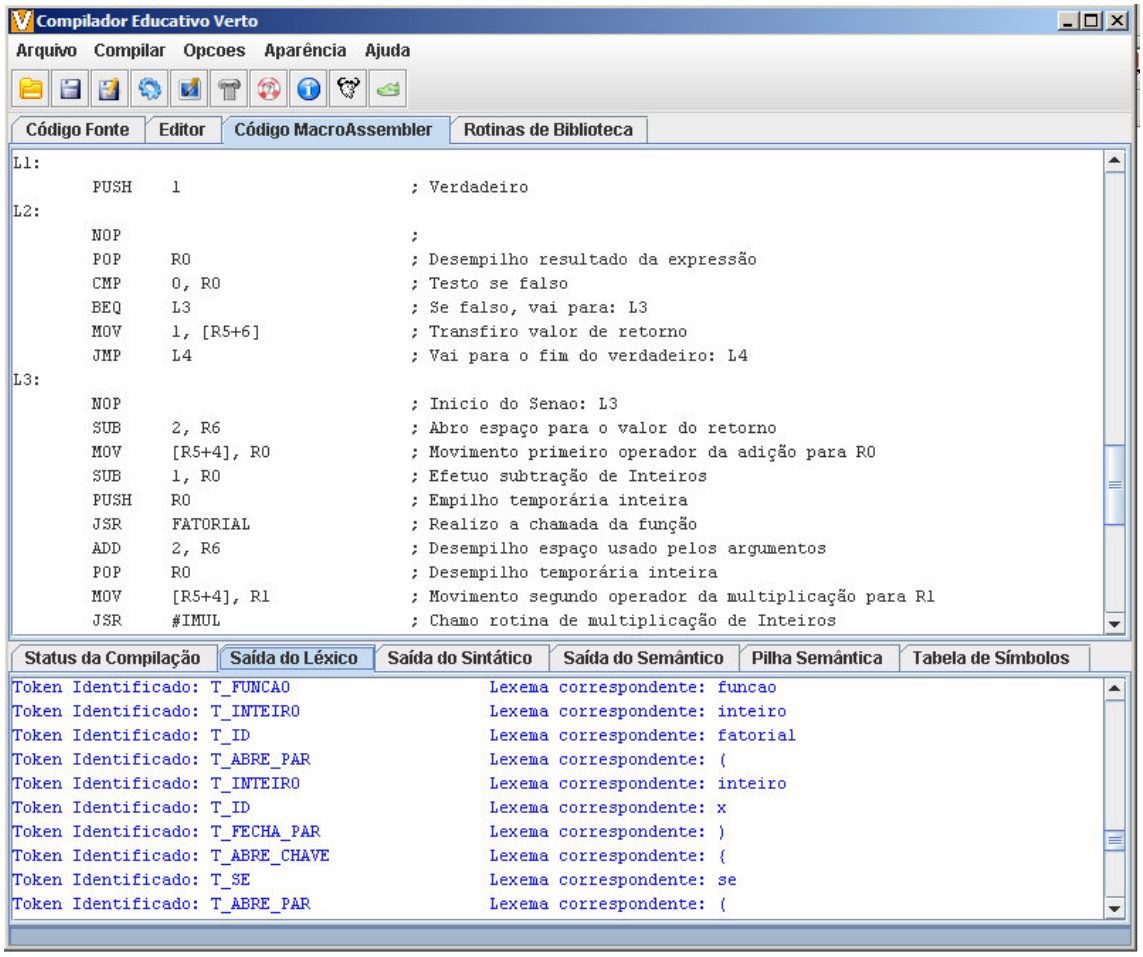


Figura 3. Exibição do código *MacroAssembler* gerado pelo *Verto*.
Fonte: (Scheider et al., 2005)

O trabalho descrito em Graciano Junior et al. (2022) é uma ferramenta *Web* para o ensino do funcionamento das etapas de análise léxica e sintática de compiladores. Ela apresenta os conteúdos de forma teórica, usando recursos textuais e gráficos e permite a visualização interativa de um passo a passo das etapas realizadas a partir de um código-

fonte fornecido pelo usuário.

Na apresentação teórica da análise léxica, o programa oferece os conteúdos separados por abas, que contêm a explicação de conceitos como *buffer*, lexemas, *tokens*, autômatos determinísticos e não determinísticos. Já na parte prática, a ferramenta mostra graficamente um autômato responsável por realizar o reconhecimento dos *tokens*, permitindo ao usuário acompanhar passo a passo a mudança de seus estados, como pode ser visto na figura 4. A explicação teórica da análise sintática também é dividida em abas, explicando os conteúdos necessários para o entendimento dessa etapa, como gramáticas livres de contexto, árvore de derivação e os conjuntos *First* e *Follow*. Na parte prática, é apresentado um passo a passo da derivação da árvore usando os *tokens* obtidos na análise léxica, com o qual o usuário pode interagir.

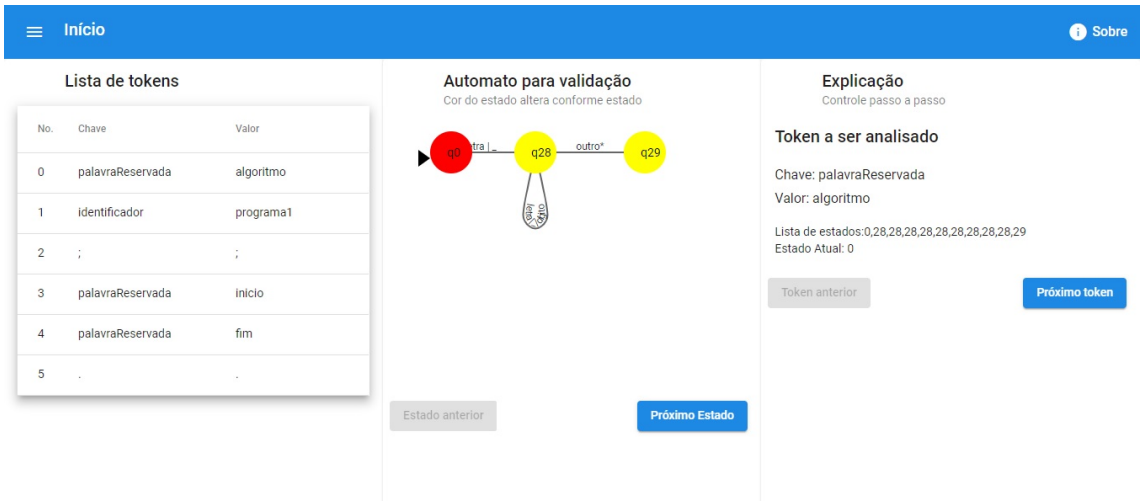


Figura 4. Exibição do autômato responsável por realizar o reconhecimento dos *tokens* de forma interativa.

Fonte: (Graciano Junior et al., 2022)

O quadro 1 apresenta uma comparação entre as principais funcionalidades de cada trabalho descrito, junto com as funcionalidades proposta neste trabalho.

Dentre os trabalhos analisados, uma das principais características observadas foi a plataforma utilizada. Ferramentas mais antigas tendem a usar a plataforma *desktop*, o que pode dificultar o acesso dos usuários, uma vez que requer a instalação no sistema operacional. Apenas o trabalho de Graciano Junior et al. (2022) foi desenvolvido para essa plataforma. Também foi constatado que apenas a ferramenta apresentada por Scheider et al. (2005) realiza todo o processo de compilação e permite a execução do programa resultante.

Com base nesses estudos, a ferramenta proposta busca, na área de compiladores, oferecer a visualização textual dos resultados das etapas de compilação, tanto para o código intermediário (IR) quanto para o código-alvo, similar à abordagem de Scheider et al. (2005). Além disso, propõe exibir a árvore de derivação de forma gráfica, como feito nos trabalhos de Mernik e Zumer (2003) e Graciano Junior et al. (2022), porém sem incluir a funcionalidade de passo a passo. Assim como os trabalhos mencionados não contemplam recursos de estruturas de dados avançadas e alocação dinâmica de memória,

Trabalho	Plataforma	Etapas	Visualização dos resultados	Permite algoritmos arbitrários	Permite execução do programa	Alocação dinâmica de memória e estruturas avançadas
Mernik e Zumer (2003)	*Desktop*	Léxica Sintática Semântica	Todas as etapas com passo a passo	Sim	Não	Não
Scheider et al. (2005)	*Desktop*	Todas	Mostra de forma textual	Sim	Sim	Não
Graciano Junior et al. (2022)	*Web*	Léxica Sintática	Todas as etapas com passo a passo	Sim	Não	Não
Sistema proposto	*Web*	Todas	Mostra de forma textual	Sim	Sim	Não

Quadro 1. Resumo das características das ferramentas de ensino de compiladores.

devido à complexidade envolvida, este projeto também opta por não fornecer tais funcionalidades. A ferramenta foi desenvolvida para um ambiente *web*, visando facilitar o acesso dos usuários, e permite a execução dos códigos compilados diretamente na plataforma, funcionalidade oferecida apenas pela ferramenta de Scheider et al. (2005).

2.2. Análise da complexidade de algoritmos

Uma revisão bibliográfica nas plataformas *Google Scholar* e *ScienceDirect* foi realizada para a obtenção dos trabalhos similares sobre a análise da complexidade de algoritmos. As chaves de pesquisa utilizadas foram "*algorithm complexity*", "*algorithm complexity learning*", "*algorithm complexity educacional*", "*algorithm complexity tool*", "complexidade de algoritmos" e "complexidade de algoritmos educacional". Dos trabalhos obtidos, uma pré-seleção foi feita com base no título e desses a leitura dos seus resumos foi realizada. Os três trabalhos mais relevantes para o ensino da análise da complexidade de algoritmos foram escolhidos para uma leitura completa e detalhamento nessa seção.

ANAC, sigla para Analisador de Complexidade, descrito por Barbosa et al. (2001), é uma ferramenta com objetivos educacionais criada na linguagem *Java* que realiza o cálculo da complexidade de forma semi-automática em notação assintótica no pior caso ou caso médio.

O programa aceita um algoritmo não recursivo escrito em uma linguagem similar ao *Pascal* e, ao iniciar a execução, é perguntado qual variável representará o tamanho da entrada do algoritmo. No decorrer da análise, a ferramenta pode requerer do usuário uma tomada de decisão para situações necessárias, como escolher um fluxo de execução es-

pecífico ao encontrar ramificações condicionais ou solicitar uma estimativa do número de execuções que uma estrutura de iteração condicional irá realizar. Internamente, é criada uma árvore de equações que é exibida na tela conforme o processo de análise da complexidade do algoritmo avança e, no final, a árvore de equações é simplificada e manipulada para retornar ao usuário a complexidade em notação assintótica. O sistema não possui nenhum recurso gráfico ou interativo, consistindo apenas na exibição das equações e no resultado final em forma de texto, como pode ser visto na figura 5.

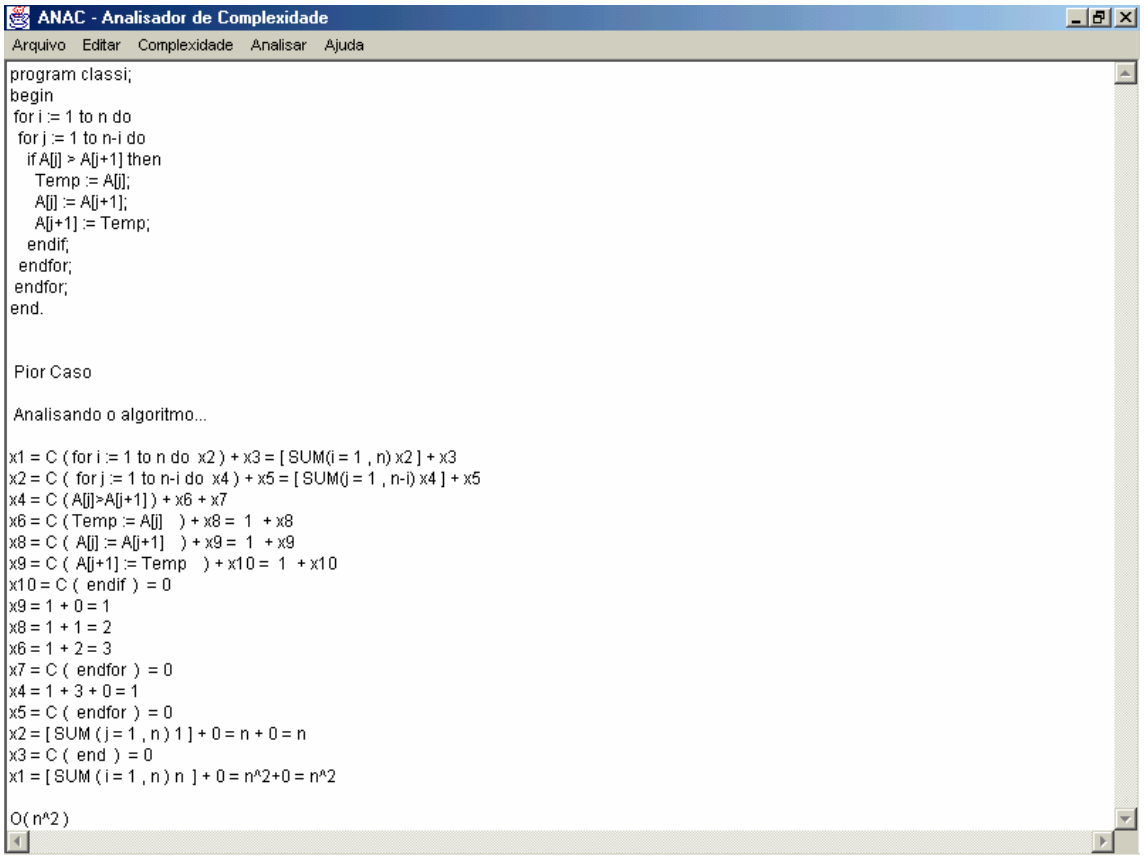


Figura 5. Exibição da interface do *ANAC*, com o código a ser analisado, a árvore de equações e o resultado final.

Fonte: (Barbosa et al., 2001)

O trabalho intitulado Ensino de Teoria da Complexidade, detalhado por Rese e Santiago (2012), apresenta diversas ferramentas didáticas voltadas ao ensino de tópicos da teoria da computação. Essas ferramentas incluem explicações visuais para as classes de complexidade, demonstrações interativas dos conceitos de árvore de *backtracking* e prova de NP-Completo. Para a análise da complexidade de algoritmos, a ferramenta disponibiliza um jogo no qual é possível escolher entre alguns algoritmos pré-determinados, informar o tamanho da entrada dos algoritmos e selecionar as operações fundamentais consideradas corretas. Para a verificação da seleção das operações fundamentais corretas, o programa exibe um campo de texto com o valor total da quantidade de vezes que as operações fundamentais foram executadas, e, para saber se a resposta está correta, é mostrado um segundo campo de texto com o valor correto de operações que deveriam ter sido executadas, como pode ser visto na figura 6. No entanto, a ferramenta não possui a

capacidade de calcular a análise da complexidade de algoritmos arbitrários[6], nem fornece uma explicação do resultado obtido.

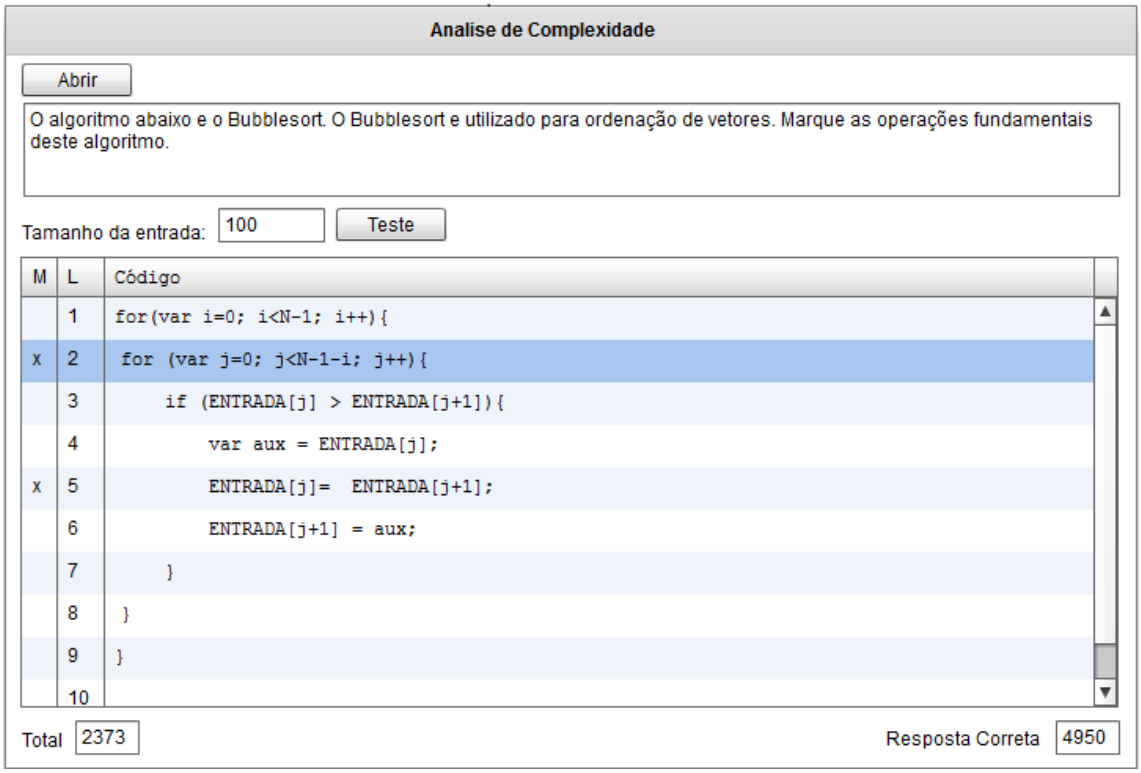


Figura 6. Exibição do jogo disponibilizado para o aprendizado da análise de complexidade.

Fonte: (Rese e Santiago, 2012)

No trabalho Um Avaliador Automático de Eficiência de Algoritmos para Ambientes Educacionais de Ensino de Programação, apresentado por Costa et al. (2014), são introduzidas duas ferramentas educacionais: IGED (Interpretador Gráfico de Estrutura de Dados) e *MOCCA* (*Measurement of Complexity of an Certain Algorithm*). A primeira é uma ferramenta que permite ao usuário desenvolver algoritmos em uma linguagem específica do programa, contendo animações para as estruturas de dados utilizadas.

O IGED possui integração com o *MOCCA*, que consegue realizar uma estimativa da classificação da complexidade dos algoritmos executando diversas instancias do algoritmo com tamanhos de entradas randomizadas e realizando a contagem dos passos a fim de gerar a função $T(n)$ do algoritmo para alimentar uma tabela de eficiência. Utilizando-se da interpolação polinomial de Lagrange e a tabela de eficiência gerada, o programa consegue estimar a classificação da complexidade do algoritmo e cria um gráfico com a função de crescimento da entrada, gerando também um relatório com as informações de custo obtidas. A ferramenta permite a especificação de múltiplos algoritmos diferentes para serem analisados e exibe um gráfico comparativo entre as funções de crescimento da entrada, como pode ser visto na figura 7.

[6]Não permitir algoritmos arbitrários significa que a ferramenta possui apenas códigos pré-determinados, não permitindo ao usuário inserir outros códigos escritos por ele para a análise da ferramenta.

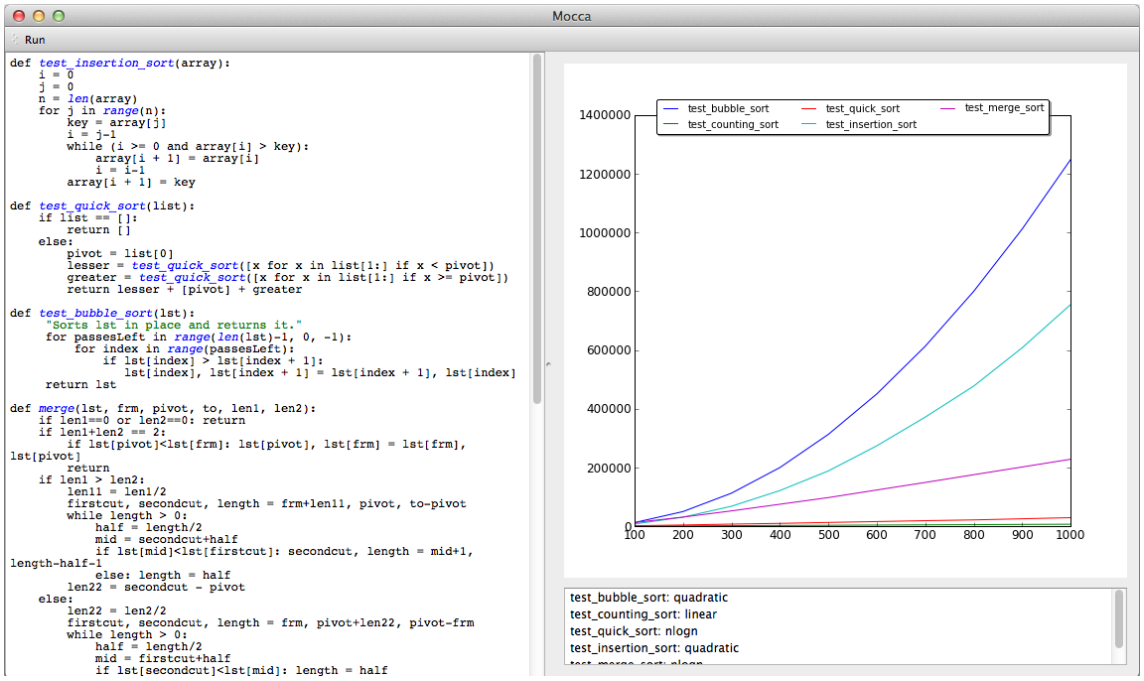


Figura 7. Exibição da interface do *MOCCA* com os códigos a serem analisados e com o gráfico comparativo entre eles.

Fonte: (Costa et al., 2014)

Com base nas ferramentas descritas, o quadro 2 apresenta a comparação entre as principais funcionalidades de cada uma, junto com as funcionalidades proposta neste trabalho.

Trabalho	**Plataforma**	**Mostra passo a passo**	**Cálculo automático**	**Permite algoritmos arbitrários**	**Trata recursão**	**Exibe a função $T(n)$**
Barbosa et al. (2001)	*Desktop*	Não	Semi automático	Sim	Não	Sim
Rese e Santiago (2012)	*Desktop*	Não	Sim	Não	Não	Não
Costa et al. (2014)	*Desktop*	Não	Sim	Sim	Sim	Não
Sistema proposto	*Web*	Sim	Sim	Sim	Não	Sim

Quadro 2. Resumo das características das ferramentas de ensino da análise da complexidade de algoritmos.

Nos trabalhos analisados, diversas características distintas foram observadas. A principal foi que nenhuma das ferramentas estava disponível para a *web*, o que dificulta

para os usuários encontrá-las e instalá-las, especialmente aqueles que desejam apenas testá-las. Outra característica notada é que, mesmo as ferramentas com propósitos educacionais, não oferecem a possibilidade de acompanhar um passo a passo do processo de análise de algoritmos. Além disso, apenas a ferramenta do estudo de Barbosa et al. (2001) exibe a função $T(n)$ para os usuários.

Com base nesses trabalhos, a ferramenta proposta tem como objetivo, na parte de análise da complexidade de algoritmos, oferecer um ambiente *web* para facilitar o acesso dos usuários, permitindo um passo a passo interativo com descrições do cálculo da função $T(n)$ e a exibição da função completa ao final, similar ao trabalho de Barbosa et al. (2001). Além disso, será possível a execução de códigos arbitrários, como nas ferramentas dos trabalhos de Barbosa et al. (2001) e Costa et al. (2014).

3. Implementação e apresentação do sistema

Essa seção apresenta os recursos que foram implementados. A seção 3.1 descreve os detalhes da arquitetura geral da implementação programa, a seção 3.2 descreve de forma especifica o funcionamento e implementações do módulo de compilação enquanto a seção 3.3 descreve a análise da complexidade de código.

3.1. Arquitetura geral

A arquitetura implementada segue o modelo cliente-servidor. O cliente, também conhecido como *front-end*, é a parte com a qual o usuário interage diretamente, tendo como principal objetivo a exibição da interface gráfica e dos dados retornados pelo servidor. A implementação do cliente é voltada para o ambiente *web*, utilizando o *framework Vue.js 3*, que facilita a criação de interações reativas com o usuário, por meio de renderizações dinâmicas de páginas. O *Vuetify* [7], uma biblioteca de componentes baseada no sistema de design *Material*, é empregado para a interface gráfica, juntamente com o editor de texto *CodeMirror*.

O servidor, ou *back-end*, é responsável por todo o processamento do programa, incluindo a compilação e a análise de complexidade, sem interação direta com o usuário. Ele é implementado em *Java*, utilizando o *framework Spring Boot* para criar uma *API REST*, permitindo a comunicação entre cliente e servidor por meio de requisições *HTTP*. O servidor também se comunica com o sistema operacional para realizar o cacheamento dos artefatos de compilação, se integrar com o compilador *LLVM* e também para a criação dos processos para a execução do código compilado, onde os dados de entrada e saída são trafegados usando o protocolo *WebSocket*.

Para facilitar a configuração do ambiente, a ferramenta de conteinerização *Docker* [8] é utilizada, permitindo a automação dos processos de compilação e instalação do programa, além de suas dependências, em uma máquina virtual. O sistema operacional utilizado é o *Linux*, e a comunicação com a máquina virtual é gerenciada por meio de um *proxy* reverso, utilizando o *Nginx* [9] para direcionar o acesso aos recursos do cliente e do servidor em rotas separadas. Para o desenvolvimento das implementações realizadas, foi utilizada a ferramenta online de versionamento de código *GitHub*. O repo-

[7] `https://vuetifyjs.com/`
[8] `https://docker.com/`
[9] `https://nginx.org/`

sitório pode ser consultado no endereço: `https://github.com/erikborella/projeto-compiladores-ifsc`.

A figura 8 mostra a arquitetura do projeto, com a comunicação entre os seus componentes de forma conceitual.

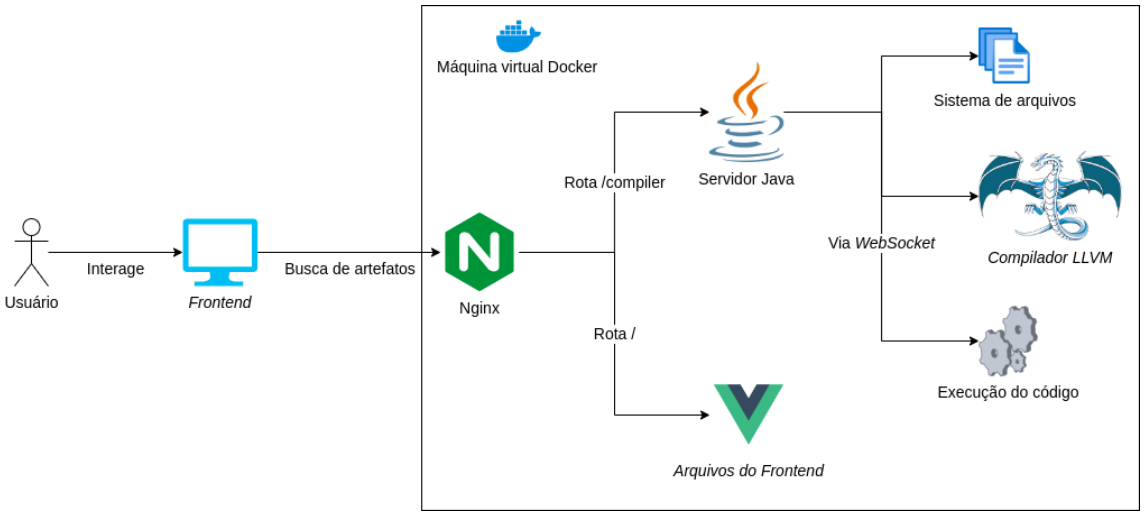


Figura 8. Arquitetura do projeto de forma conceitual.

3.2. Modulo de compilação

3.2.1. Descrição da linguagem

A linguagem de programação suportada pelo compilador implementado possui uma sintaxe inspirada na linguagem *C*, sendo ela uma linguagem compilada com tipos estáticos e fracamente tipada. Todas as regras léxicas estão definidas no código 1 do Apêndice A, enquanto as regras sintáticas estão definidas no código 2 do Apêndice A.

A linguagem suporta quatro tipos de dados básicos:

- $boolean$ de tamanho de 1 *bit*;
- $char$ de tamanho de 8 *bits*;
- int de tamanho de 32 *bits*;
- $float$ de tamanho de 64 *bits*.

Além disso, é permitido a declaração de *arrays* de tamanho fixo e multidimensionais. A regra sintática $tipo$ (código 2 do Apêndice A) define a estrutura geral das declarações de tipo.

Para que o nome de uma variável seja válido, ele deve seguir as regras de identificadores da linguagem. Um identificador válido deve iniciar com uma letra (maiúscula ou minúscula) ou o caractere sublinhado (_). Os caracteres subsequentes podem ser letras, dígitos ou o caractere sublinhado. Não são permitidos acentos ou caracteres especiais. É possível declarar múltiplas variáveis do mesmo tipo em uma única linha, separando-os por vírgulas e todas as declarações devem terminar com o ponto e vírgula. A regra sintática que define a estrutura para as declarações de variáveis é a $decvariavel$.

As variáveis devem ser declaradas no início de cada bloco de código, antes de qualquer outro comando ou atribuição. Os blocos são delimitados pelos caracteres { e }, conforme a regra *bloco*.

O código 1 apresenta um exemplo de declaração de variáveis.

```
main() {
    boolean var1_;
    char _var2;
    int var_3, var_4;
    float var5;

    int[5] arr;
    float[10][15] matriz;
}
```

Código 1. Exemplo de declarações de variáveis.

Após as declarações de variáveis de um bloco, é permitido especificar os comandos, definidos na regra *comando*. A linguagem possui dois tipos de comandos: comandos de linha, que podem ser expressos em apenas uma linha e devem ser terminados com ponto e virgula (;) e comandos de bloco, que requerem a definição de um bloco delimitado pelos caracteres { e }.

O primeiro comando de linha é a atribuição, que consiste em armazenar um valor em uma variável. Esse valor pode ser um número constante, o resultado de um cálculo matemático, o conteúdo de outra variável ou elemento de um *array*, ou ainda, o retorno de uma função. Pelo fato da linguagem ser fracamente tipada, caso o tipo da variável não seja igual ao do valor que está sendo atribuído nela, quando possível uma conversão será colocada pelo compilador de forma implícita. A regra sintática que define o comando de atribuição é a *atribuicao* (código 2 do Apêndice A).

O compilador aceita fazer cálculos com as seguintes operações matemáticas, respeitando a procedência de cada uma.

- $+$ para adições;
- $-$ para subtrações;
- $*$ para multiplicações;
- $/$ para divisões;
- $\%$ para o valor de resto da divisão.

O código 2 apresenta um exemplo das possibilidades de atribuição de valores à variáveis.

A linguagem também oferece comandos para leitura de valores do usuário e exibição de mensagens na tela.

O comando de leitura é definido pela regra *leitura*. Utiliza-se a palavra-chave *scanf*, e entre parênteses deve-se especificar a variável onde o valor fornecido pelo usuário será armazenado. O programa pausa a execução até que o usuário insira um valor.

Já o comando de escrita de mensagens possui duas variações, definidas pelas regras sintáticas *escrita* e *escritaln*. O comando *escrita* exibe uma mensagem na tela

sem adicionar uma nova linha ao final, enquanto $escritaln$ adiciona uma nova linha ao término da mensagem.

```
main() {
    int inteiro1, inteiro2;
    int[3] arr;
    float decimal1;
    // valor constante
    inteiro1 = 1;
    // expressão matemática
    inteiro2 = 1 + 2 * 3;
    // valor de uma constante
    arr[0] = inteiro1 + inteiro2;
    // valor de um array
    arr[1] = arr[0];
    // retorno de uma função
    arr[inteiro1] = func getInt();
    // conversão implicita
    decimal1 = inteiro2;
}
```

Código 2. Exemplo de atribuição de variáveis.

Seu uso é feito através das palavras reservadas $print$, para o comando $escrita$, e $println$, para o $escritaln$. Entre parênteses, deve-se especificar um texto entre aspas duplas contendo o modelo da mensagem a ser exibida e após deve ser passado os valores para serem exibidos, separados por vírgula. O modelo segue o mesmo padrão utilizado pela *glibc*[10]. Por exemplo, para exibir valores inteiros, utiliza-se a sequência $\%d$, enquanto para valores reais, a sequência $\%f$; é possível ainda especificar o número de casas decimais a serem exibidas usando a sequência $\%.Nf$, onde N é a quantidade de casas decimais desejadas.

O código 3 apresenta um exemplo de um programa para o cálculo da área de um círculo para exemplificar o uso dos comandos de leitura e escrita de dados.

```
main() {
  float pi, area;
  int raio;
  pi = 3.141592;
  print("Digite o raio do seu circulo: ");
  scanf(raio);
  area = pi * (raio * raio);
  println("Area do circulo = %.2f", area);
}
```

Código 3. Exemplo de leitura e escrita de dados com um programa que calcula a areá de um circulo.

Também é possível chamar funções declaradas utilizando o comando $funcao$. Para chamar uma função, utiliza-se a palavra reservada $func$ seguida pelo nome da função. Além disso, é necessário adicionar parênteses após o nome da função; caso a

[10] https://www.gnu.org/software/libc/manual/html_node/Formatted-Output.html

função receba argumentos, eles devem ser especificados dentro dos parênteses, separados por vírgulas.

A declaração de funções, definida pela regra $decfuncao$, deve ser feita antes do bloco $main$. Primeiro, especifica-se o tipo de retorno da função, que pode ser qualquer tipo suportado pela linguagem, ou $void$, caso a função não retorne nenhum valor. Em seguida, é atribuído um nome único à função, e, por fim, adicionam-se os parênteses. Caso a função aceite parâmetros, deve-se defini-los dentro dos parênteses, especificando primeiro o tipo e, em seguida, o nome de cada parâmetro, separados por vírgulas. *Arrays* são passados por referência e as demais variáveis são copiadas ao serem passadas para a função.

O retorno de funções é especificado pelo comando $retorno$, utilizando a palavra-chave $return$, opcionalmente seguida de um valor. Quando o retorno ocorre na função $main$, o valor retornado será o código de status de encerramento do programa. Por padrão, as funções retornam o valor 0, caso o tipo de retorno não seja $void$ e nenhum comando $return$ tenha sido especificado no escopo da função.

O código 4 apresenta um exemplo de declaração e chamada de funções com retorno de valores.

```
1  void iniciarArray(int[3] array) {
2    array[0] = 1;
3    array[1] = 2;
4    array[2] = 3;
5  }
6
7  int somarArray(int[3] array) {
8    return array[0] + array[1] + array[2];
9  }
10
11 main() {
12   int[3] array;
13   int resultado;
14   // Passagem do array por referencia
15   func iniciarArray(array);
16   resultado = func somarArray(array);
17   println("Soma do array = %d", resultado);
18 }
```

Código 4. Exemplo de declarações e chamadas de funções com retorno de valores.

A linguagem possui três comandos de bloco: $selecao$, $enquanto$ e $para$.

O comando $selecao$ permite o controle de fluxo condicional no programa. Seu uso é feito por meio da palavra reservada if, seguida de parênteses contendo uma expressão condicional. Após isso, deve-se definir um bloco de código que será executado caso a expressão condicional seja verdadeira. Opcionalmente, pode-se definir um bloco para ser executado caso a condição seja falsa, utilizando a palavra-chave $else$ ao final do bloco if, seguida pelo bloco de código que será executado em caso de falsidade.

Para construir expressões condicionais, a linguagem permite o uso dos seguintes operadores padrão da linguagem C, são eles: $==$, $!=$, $>$, $>=$, $<$, $<=$ e !.

Além disso, é possível combinar expressões condicionais utilizando os operadores lógicos && (operador e) e || (operador ou). A avaliação dos operadores lógicos é realizada com curto-circuito, ou seja, em uma expressão com o operador &&, se a expressão à esquerda for falsa, a expressão à direita não será avaliada, pois o compilador já sabe que o resultado será falso. O mesmo ocorre com o operador ||, onde, se a expressão à esquerda for verdadeira, a expressão à direita não será avaliada.

O código 5 apresenta um exemplo do uso do comando *selecao*.

```
main() {
  int valor;
  print("Digite o seu valor: ");
  scanf(valor);
  if (valor > 0 && valor < 10) {
    println("O seu valor esta entre 0 e 10");
  }
  else {
    if (valor <= -10 || valor >= 50) {
      println("O seu valor esta abaixo de -10 ou acima de 50");
    }
    else {
      println("O seu valor nao esta entre nenhuma margem definida");
    }
  }
}
```

Código 5. Exemplo do funcionamento do comando de bloco *selecao*.

O comando de bloco *enquanto* também permite o controle condicional do código, similar ao comando *selecao*, com a diferença de que o bloco de código é executado em um *loop* enquanto a expressão condicional permanecer verdadeira. Seu uso é feito utilizando a palavra-chave *while*, seguida de parênteses contendo a expressão condicional e após o bloco de código para ser executado.

O código 6 demonstra o uso do comando *enquanto*.

```
main() {
  int valor;
  print("Digite um numero par: ");
  scanf(valor);
  while (valor % 2 != 0) {
    println("Voce digitou um numero impar");
    print("Tente novamente digitando um numero par: ");
    scanf(valor);
  }
  println("O numero par digitado foi: %d", valor);
}
```

Código 6. Exemplo do funcionamento do comando de bloco *enquanto*.

Por fim, o comando de bloco *para* funciona de forma semelhante ao *enquanto*, permitindo a criação de *loops*. Entretanto, o *para* possui uma sintaxe especial que facilita a implementação de *loops* onde o número de iterações é bem definido, permitindo a inicialização e a modificação das variáveis de controle de forma mais compacta.

Seu uso é feito utilizando a palavra reservada *for*, seguida de parênteses. Dentro

dos parênteses, primeiro é colocada uma atribuição para as variáveis de controle, que será executada apenas uma vez, permitindo inicializar seus valores. Cada atribuição pode ser separada por vírgulas e, ao final, deve-se adicionar um ponto e vírgula para delimitar as atribuições. Após isso, é necessário definir uma expressão condicional, que determinará até quando o bloco de código será executado. Em seguida, adiciona-se outro ponto e vírgula para delimitar a expressão condicional. Por último, definem-se as atribuições que serão executadas ao final de cada iteração do bloco de código, permitindo assim modificar as variáveis de controle a cada ciclo.

O código 7 ilustra o uso do comando *para*.

```
1 main() {
2   int i, numero;
3   print("Digite um numero para listar todos os numero de 0 ate o numero
        escolhido: ");
4   scanf(numero);
5   for (i = 0; i < numero; i = i + 1) {
6     println("%d", i);
7   }
8 }
```

Código 7. Exemplo do funcionamento do comando de bloco *para*.

Para um exemplo completo na linguagem desenvolvida, o código 1 do Apêndice B, mostra um programa que recebe 5 notas de um aluno em um *array*, calcula a média das notas e exibe se o aluno atingiu a média 7.

3.2.2. Representação intermediária

O compilador, ao ser executado, gera uma representação intermediária na forma textual do *LLVM* (*LLVM IR*). O *LLVM IR* utiliza a forma *Static Single-Assignment* (*SSA*), ou seja, cada variável no código intermediário só pode ser atribuída uma única vez, sendo também fortemente tipado. O uso de *SSA* traz diversos benefícios, como maior facilidade na geração de código de máquina e na otimização do código (Lattner e Adve, 2004). Após a geração do código intermediário, o compilador do *LLVM* se encarrega de realizar otimizações e a geração do executável final.

Para facilitar a geração de código na forma *Static Single-Assignment* (SSA), o *LLVM IR* oferece recursos que simplificam o processo, como a possibilidade de atribuir nomes às variáveis de maneira incremental, permitindo que o compilador gere automaticamente identificadores numéricos conforme novas variáveis são criadas, além de possuir uma manipulação e alocação de memória na *stack* simplificada.

Embora o *LLVM IR* tenha muitos recursos avançados, devido à simplicidade da linguagem implementada, alguns deles não foram utilizados. Entre os recursos não utilizados estão: definição de estruturas compostas, que permite a criação de *structs*; uso de *vectors*, facilitando o acesso a instruções do tipo *Single Instruction, Multiple Data* (*SIMD*) em processadores compatíveis; uso de funções intrínsecas do *LLVM*, que incluem diversas funções matemáticas já otimizadas e operações de manipulação de memória, entre outras funcionalidades.

O *LLVM IR* permite definir os tipos de variáveis inteiras com um tamanho arbitrário, ou seja, pode-se ter inteiros de tamanhos comumente utilizados por outros compiladores, como variáveis de 8, 16 e 32 bits, assim como inteiros de tamanhos incomuns, como 100, 150 ou 500 bits. O *LLVM IR*, ao gerar o código de máquina, consegue produzir instruções *assembly* que manipulam registradores para operar com esses tipos customizados. Para valores de ponto flutuantes, o *LLVM IR* não permite essa flexibilidade na definição dos seus tamanhos, oferecendo opções pré-determinadas.

Para valores inteiros, os tipos no *LLVM IR* seguem o padrão iN, onde N representa a quantidade de bits da variável. Já para variáveis de ponto flutuante, o *LLVM IR* oferece tipos como $half$ (16 bits), $float$ (32 bits), $double$ (64 bits), entre outras variações.

Devido à maneira como os números de ponto flutuante são armazenados e manipulados internamente pelo processador, pequenas imprecisões podem ocorrer. Por exemplo, ao realizar o cálculo $0.1 + 0.2$, o resultado não será exatamente 0.3, mas sim um valor muito próximo. Esse comportamento pode gerar resultados inesperados em expressões condicionais, principalmente ao utilizar comparações de igualdade entre números de ponto flutuante.

No compilador implementado, os tipos tradicionais foram utilizados para mapear os tipos da linguagem com os do *LLVM IR*. Os mapeamentos são os seguintes:

- $boolean$: mapeado para $i1$;
- $char$: mapeado para $i8$;
- int: mapeado para $i32$;
- $float$: mapeado para $double$.

No *LLVM IR*, a declaração de qualquer variável é feita utilizando o comando $alloca$, que aloca memória na *stack* do programa e retorna um ponteiro para essa região alocada.

O código 8 ilustra o código intermediário gerado pelo compilador para a declaração de uma variável simples e de um *array*, com comentários adicionais que indicam as linhas correspondentes ao código-fonte original.

```
1  define i32 @main() {
2      ; int variavelInteira;
3      %variavelInteira = alloca i32
4      ; int[3][7] arrayDeDuasDimensoes;
5      %arrayDeDuasDimensoes = alloca [3 x [7 x i32]]*
6      %1 = alloca [3 x [7 x i32]]
7      store [3 x [7 x i32]]* %1, [3 x [7 x i32]]** %arrayDeDuasDimensoes
8      ; final do bloco com "return 0" implicito
9      ret i32 0
10 }
```

Código 8. Código intermediário gerado pelo compilador para a declaração de uma variável e de um *array*.

No caso da declaração de *arrays*, são feitas duas alocações de memória: a primeira aloca um ponteiro para a estrutura do *array*, facilitando a geração posterior de código para acessar os elementos e também facilitando a passagem do *array* como referência para funções. A segunda alocação é a que efetivamente reserva espaço para o *array* em si.

Para realizar a atribuição de valores em variáveis no *LLVM IR*, utiliza-se o comando $store$. Este comando recebe dois argumentos: o primeiro é o valor a ser armazenado, e o segundo é um ponteiro que aponta para a região de memória onde esse valor será salvo.

O exemplo no código 9 demonstra como o valor de uma variável inteira é armazenado usando o comando $store$ no código intermediário gerado pelo compilador.

```
; variavelInteira = 10;
store i32 10, i32* %variavelInteira
```

Código 9. Código intermediário gerado pelo compilador para atribuição de um valor numérico em uma variável.

Para realizar operações com os valores armazenados nas variáveis, como atribuir esses valores a outras variáveis, passá-los para funções ou utilizá-los em expressões matemáticas, é necessário carregar os valores da memória utilizando o comando $load$. Este comando recebe dois argumentos: o primeiro é o tipo do valor a ser carregado, e o segundo é um ponteiro para a região da memória de onde o valor será carregado. O código 10 mostra um exemplo de código intermediário gerado pelo compilador para carregar o valor de uma variável.

```
; carregamento do valor da váriavel "int variavelInteira"
%1 = load i32, i32* %variavelInteira
; armazena o valor carregado da "variavelInteira" na "variavel2"
store i32 %1, i32* %variavel2
```

Código 10. Código intermediário gerado pelo compilador para o carregamento do valor de uma variável inteira.

Para poder realizar cálculos matemáticos com valores inteiros, o *LLVM IR* disponibiliza comando matemáticos como: add para adições, sub para subtrações, mul para multiplicações, $sdiv$ para divisões com sinal e $srem$ para obter o valor do resto da divisão. Já para números de ponto flutuante é necessário usar a variante de comandos utiliza um prefixo f, como $fadd$ para adições ou $fdiv$ para divisões.

No *LLVM IR*, cada operação matemática deve ser realizada de forma individual. Portanto, para contas matemáticas com várias operações, é necessário encadear os resultados de duas operações consecutivas.

O código 11 apresenta um exemplo do cálculo de uma expressão matemática gerado pelo compilador.

```
; variavelInteira = 10 + 5 * 15 / 3;
; 5 * 15
%1 = mul i32 5, 15
; (5 * 15) / 3
%2 = sdiv i32 %1, 3
; 10 + (5 * 15 / 3)
%3 = add i32 10, %2
; variavelInteira = (10 + 5 * 15 / 3)
store i32 %3, i32* %variavelInteira
```

Código 11. Código intermediário gerado pelo compilador para o cálculo de uma expressão matemática.

É necessário também especificar as conversões numéricas que devem ser feitas para o *LLVM IR*. Por exemplo, a atribuição direta de um valor inteiro para uma variável com tipo de ponto flutuante não é aceita, é necessário usar antes o comando $sitofp$, que significa inteiro com sinal para ponto flutuante (*Signed Integer To Floating Point*). O quadro 1 do Apêndice C mostra todas as possibilidades de conversões que são geradas pelo compilador.

O código 12 apresenta um exemplo prático do funcionamento da conversão de valores no *LLVM IR* gerado pelo compilador.

```
1 ; converte o valor 10 para ponto flutuante (double)
2 %1 = sitofp i32 10 to double
3 ; armazena o 10 convertido na variável f
4 store double %1, double* %f
```

Código 12. Código intermediário gerado pelo compilador para a conversão de tipos de váriaveis.

Para calcular os índices de *arrays*, é necessário usar o comando $getelementptr$ do *LLVM IR*. Esse comando facilita o cálculo da posição do ponteiro dentro de estruturas de dados como *arrays*. Ele recebe como primeiro argumento o tipo do *array* que será acessado, seguido de um ponteiro que aponta para a estrutura do *array*. Após esses parâmetros, o comando aceita $n + 1$ valores inteiros, onde n é o número de dimensões do *array*. Esses valores inteiros são usados para calcular os índices dos elementos do *array*.

Como o $getelementptr$ começa o cálculo dos ponteiros inicialmente com base no tipo passado para ele (ou seja, um ponteiro para um *array*), no compilador implementado, o primeiro valor passado será sempre 0. Isso serve para realizar a desreferência inicial do ponteiro do *array* e permitir o cálculo dos índices com base no tipo do *array* de fato. Os valores subsequentes são os índices reais dos elementos que estão sendo acessados em cada dimensão do *array*.

Ao final, o comando $getelementptr$ retorna um ponteiro para a posição do elemento calculado, podendo então ser facilmente lido com o $load$ ou ter um valor escrito com o $store$.

O código 13 mostra um exemplo do código intermediário gerado pelo compilador para a escrita de um valor em uma posição de um *array*.

```
1 ; carregamento do ponteiro do array "int[5][10] arr"
2 %1 = load [5 x [10 x i32]]*, [5 x [10 x i32]]** %arr
3 ; calculo do elemento na posição "arr[1][5]"
4 %2 = getelementptr inbounds [5 x [10 x i32]], [5 x [10 x i32]]* %1, i32
      0, i32 1, i32 5
5 ; escrita do valor 10 no elemento da posição "arr[1][5]"
6 store i32 10, i32* %2
```

Código 13. Código intermediário gerado pelo compilador para a escrita de um valor em um *array*.

As operações de escrita e leitura de dados pelo usuário são realizadas por meio das funções $printf$ e $scanf$ da biblioteca *glibc*. Essas funções podem ser facilmente integradas ao *LLVM IR*, sendo necessário apenas a sua definição e chamada no código intermediário gerado.

Para o uso de $printf$ e $scanf$, é preciso também definir as *strings* utilizadas como argumentos. No compilador implementado, a técnica de *interning* é aplicada a todas as *strings*, ou seja, se duas *strings* idênticas forem definidas, ambas apontarão para o mesmo valor em memória. No *LLVM IR*, as *strings* são sempre terminadas com o valor literal 0, representado como *\00* em ASCII (*null terminator*), e os valores em ASCII devem ser usados para símbolos especiais e de controle, como o valor `0x0A` para indicar uma nova linha (*line feed*).

O código 14 exibe um exemplo do código intermediário gerado pelo compilador usando as funções $printf$ e $scanf$.

```
define i32 @main() {
    %numero = alloca i32
    ; uso do printf
    call i32 (i8*, ...) @printf(i8* @.str0)
    ; uso do scanf
    call i32 (i8*, ...) @scanf(i8* @.str1, i32* %numero)
    ret i32 0
}
; declaração das strings
@.str0 = private constant [21 x i8] c"Digite seu numero: \0A\00"
@.str1 = private constant [3 x i8] c"%d\00"
; declaração do printf e scanf
declare i32 @printf(i8*, ...)
declare i32 @scanf(i8*, ...)
```

Código 14. Código intermediário gerado pelo compilador usando as funções $printf$ e $scanf$.

Para a declaração de funções no *LLVM IR*, utiliza-se o comando $define$, seguido do tipo de retorno da função e, em seguida, do nome da função prefixado com o caractere @. Os parâmetros da função são especificados logo após o nome, entre parênteses. O corpo da função, que contém o código a ser executado, é delimitado por chaves.

Para simplificar a geração do código intermediário, o compilador implementado utiliza uma convenção onde todos os parâmetros são definidos com o sufixo $.param$. Logo no início do bloco de código da função, os valores desses parâmetros são armazenados em variáveis recém-alocadas, sem o sufixo. Dessa forma, o acesso aos parâmetros é feito da mesma maneira que o acesso às outras variáveis ao longo do código.

Para especificar o retorno de um valor da função, é utilizado o comando ret seguido do valor a ser retornado. Caso a função não tenha um tipo de retorno ($void$), pode-se usar apenas comando ret isolado para encerrar a execução da função.

O código 15 exemplifica o código intermediário gerado para a declaração de uma função com parâmetros.

A chamada de funções no *LLVM IR* é realizada utilizando o comando $call$. Esse comando é seguido pelo tipo de retorno da função e, em seguida, pelo nome da função, que é prefixado com o caractere @. Caso a função possua argumentos, eles devem ser fornecidos entre parênteses logo após o nome, sendo necessário especificar tanto o tipo quanto o nome de cada variável passada como argumento.

```
; int funcao(int n1, int n2) { ...
define i32 @funcao(i32 %n1.param, i32 %n2.param) {
    ; armazenamento do parametro %n1.param na váriavel n1
    %n1 = alloca i32
    store i32 %n1.param, i32* %n1
    ; armazenamento do parametro %n2.param na váriavel n2
    %n2 = alloca i32
    store i32 %n2.param, i32* %n2
    ; resto do bloco ...
    ; retorno do valor do parametro n1
    %1 = load i32, i32* %n1
    ret %1
}
```

Código 15. Código intermediário gerado pelo compilador para a declaração de funções.

O código 16 mostra um exemplo do código intermediário gerado para a chamada de uma função.

```
; chamada da função funcao
%1 = call i32 @funcao(i32 1)
; armazenamento do valor retorno na váriavel n
store i32 %1, i32* %n
```

Código 16. Código intermediário gerado pelo compilador para chamada de uma função.

Para a geração dos comandos de controle de fluxo da linguagem, o *LLVM IR* não oferece uma solução pronta, sendo necessário montar a estrutura dos comandos de controle utilizando *labels* e instruções de salto condicionais e incondicionais.

Para realizar saltos no *LLVM IR*, é preciso primeiro definir as *labels*, que indicam os pontos no código para onde a execução será direcionada após o salto. As *labels* são definidas por um nome único e são seguidas por dois-pontos ao final (:). O compilador implementado segue o padrão de geração de *labels* com um prefixo de dois pontos seguidos (..), facilitando a identificação no código e também a fim de evitar conflitos com outras *labels* um numero é adicionado após o seu nome.

Os saltos incondicionais são realizados com o comando br, que recebe como argumento o nome da *label* para onde o código será transferido.

Para saltos condicionais, primeiro é necessário calcular a expressão condicional usando o comando $icmp$. Esse comando recebe como argumentos: o tipo de comparação, seguido pelo tipo dos dois valores a serem comparados, e finalmente, os dois valores. Os tipos de comparação disponíveis são:

- eq para igualdade;
- ne para desigualdade;
- sgt para maior;
- sge para maior ou igual;
- slt para menor;
- sle para menor ou igual.

O comando $icmp$ retorna um valor do tipo $i1$, onde 0 significa falso e 1 significa verdadeiro.

O comando br para saltos condicionais recebe três argumentos: o primeiro é o resultado da expressão condicional (do tipo $i1$), e os dois seguintes são as *labels* para os saltos, sendo a primeira para o caso em que a condição seja verdadeira e a segunda para quando a condição seja falsa.

No *LLVM IR*, cada bloco de *labels* deve ser finalizado com um comando br, que define de forma explícita o salto para outra *label*. Isso significa que, após a execução do código dentro de um bloco identificado por uma *label*, não é possível simplesmente colocar uma nova *label* logo abaixo e esperar que o fluxo de execução continue automaticamente. Em vez disso, o salto para o próximo bloco de código deve ser declarado explicitamente por meio de um comando br.

O código 17 exibe um exemplo do código intermediário gerado para a realização de desvios condicionais e incondicionais.

```
1  ; pulo incondicional para a label1
2  br label %..label1
3  ..label1:
4  ; calculo da expressão x == 1
5  %1 = load i32, i32* %x
6  %2 = icmp eq i32 1, 1
7  ; caso o resultado da expressão tenha sido verdadeiro, pula para a
       label2. Caso tenha sido falso, pula para a label3
8  br i1 %2, label %..label2, label %..label3
9  ..label2:
10 ; ...
11 ..label3:
12 ; ...
```

Código 17. Código intermediário gerado pelo compilador para a realização de desvios condicionais e incondicionais.

Para o funcionamento dos operadores lógicos $\&\&$ e $||$, sua implementação também é feita utilizando *labels* e saltos condicionais. No caso do operador $||$, se o resultado da expressão à esquerda for verdadeiro, a expressão à direita é pulada, visto que a operação já foi satisfeita. No caso do operador $\&\&$, se a operação à esquerda for falsa, a expressão à direita é pulada, pois a condição para a expressão completa ser verdadeira não pode mais ser atendida.

Para implementar essa lógica, o compilador gera *labels* como $..orN$ e $..andN$, onde N é um número gerado automaticamente para evitar conflitos de nomes entre diferentes operações lógicas no código.

A fim de simplificar a geração do código intermediário, o compilador cria uma estrutura de apoio. Essa estrutura inclui a alocação de uma variável temporária do tipo $i1$, chamada $..retValHolder$, também com um número N no final para evitar conflitos de nome. O propósito de $..retValHolder$ é armazenar o valor *booleano* resultante da expressão condicional.

Além disso, são geradas duas *labels* auxiliares: uma para definir o caso em que o resultado da expressão lógica é verdadeiro, nomeada $..tN$, e outra para o caso em que o

resultado é falso, chamada $..fN$, onde N também é usado para evitar conflitos de nome. O resultado da expressão (verdadeiro ou falso) é armazenado em $..retValHolder$.

Por fim, após o cálculo da expressão lógica completa, é gerada uma *label* $..endN$, que unifica os fluxos de execução tanto de $..tN$ quanto de $..fN$. Nessa *label*, o código termina usando o valor booleano armazenado em $..retValHolder$ para, com o comando $icmp$, realizar um salto condicional baseado no valor final da expressão.

O código 18 exibe um exemplo do código intermediário gerado para a criação do cálculo de uma expressão lógica, contendo um operador $\&\&$ e um $||$ e a figura 9 mostra o grafo de fluxo de controle para o cálculo da expressão lógica apresentada.

```
1  ; expressão a ser feita: x == 2 && y == 3 || y > 5
2  ; definição da ..retValHolder
3  %..retValHolder0 = alloca i1
4  br label %..or0
5  ..or0:
6  %1 = load i32, i32* %x
7  %2 = icmp eq i32 %1, 2
8  ; como o operador sendo executado é o &&, caso o resultado tenha sido
       verdadeiro, vai para ..and1, caso tenha sido falso, vai para ..or1
9  br i1 %2, label %..and1, label %..or1
10 ..and1:
11 %3 = load i32, i32* %y
12 %4 = icmp eq i32 %3, 3
13 ; operação à direita do &&, caso o resultado tenha sido verdadeiro, vai
       para ..t0, caso tenha sido falso, vai para ..or1
14 br i1 %4, label %..t0, label %..or1
15 ..or1:
16 %5 = load i32, i32* %y
17 %6 = icmp sgt i32 %5, 5
18 ; operação à direita do ||, caso o resultado tenha sido verdadeiro, vai
       para ..t0, caso tenha sido falso, vai para ..f0
19 br i1 %6, label %..t0, label %..f0
20 ..t0:
21 ; define ..retValHolder0 com o valor verdadeiro
22 store i1 true, i1* %..retValHolder0
23 br label %..end0
24 ..f0:
25 ; define ..retValHolder0 com o valor falso
26 store i1 false, i1* %..retValHolder0
27 br label %..end0
28 ..end0:
29 %..retVal0 = load i1, i1* %..retValHolder0
30 ; caso o valor da expressão tenha sido verdadeiro, vai para ..label1,
       caso tenha sido falso, vai para ..label2
31 br i1 %..retVal0, label %..label1, label %..label2
```

Código 18. Código intermediário gerado pelo compilador para a criação do cálculo de uma expressão lógica.

Com base nos conceitos apresentados anteriormente, é possível montar facilmente o fluxo de execução para os comandos if, $while$ e for da linguagem.

Para o comando if, são geradas três *labels*: $..ifN$, que indica o bloco de execução do if; $..elseN$, que indica o bloco de execução do $else$; e, por fim, $..end.ifN$, que marca

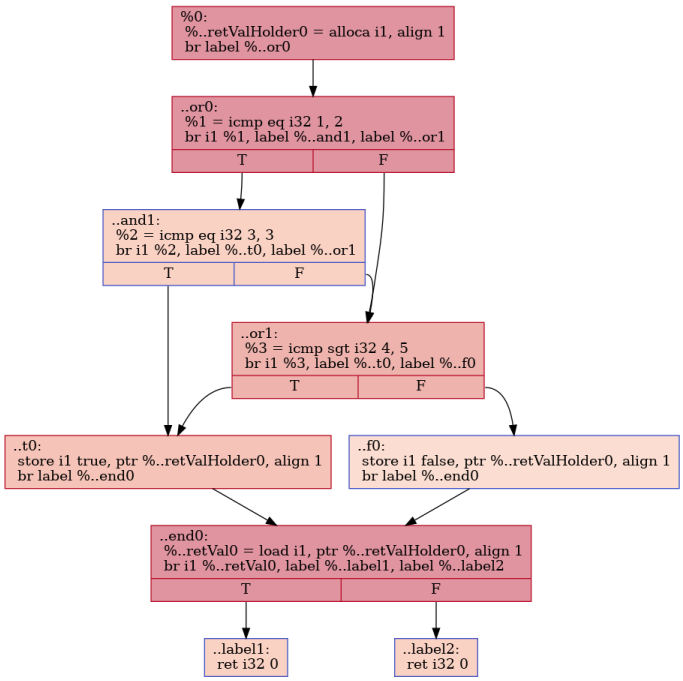


Figura 9. Grafo de fluxo de controle para o fluxo do cálculo de uma expressão condicional com os operadores $\&\&$ e $||$.

o término do comando if. Caso o if não tenha um bloco $else$, a *label* correspondente não é gerada.

O fluxo de execução inicia com o cálculo da expressão condicional. Se o resultado for verdadeiro, é feito um salto para a *label* $..ifN$; caso contrário, o salto é para a *label* $..elseN$, se o bloco $else$ existir, ou diretamente para $..end.ifN$, caso contrário. No final dos blocos $..ifN$ e $..elseN$, um pulo incondicional é realizado para a *label* $..end.ifN$, garantindo a continuidade do fluxo de execução.

O código 3 do Apêndice B demonstra a estrutura gerada pelo compilador para o comando if com $else$ na representação intermediária. A figura 1 do Apêndice D mostra um grafo de fluxo de controle gerado pelo *LLVM* para o comando if.

O comando $while$ gera uma estrutura contendo três *labels*: $..while.headN$, que indica o início da expressão condicional ; $..while.bodyN$, que indica o bloco de execução; e, por fim, $..while.endN$, que marca o término do $while$.

O fluxo de execução começa em $..while.headN$, onde o cálculo da expressão condicional é realizado. Caso o resultado seja verdadeiro, o fluxo salta para $..while.bodyN$; se for falso, o salto é para $..while.endN$. Ao final do bloco $..while.bodyN$, um pulo incondicional é feito de volta para $..while.headN$, onde a expressão será recalculada, determinando se o *loop* continuará ou não executando.

O código 4 do Apêndice B exemplifica o código intermediário para estrutura gerada pelo compilador para o comando $while$. A figura 2 do Apêndice D mostra um grafo de fluxo de controle gerado pelo *LLVM* para o comando $while$.

O comando for gera uma estrutura similar ao $while$, utilizando as mesmas *labels*,

mas com o prefixo $..for$. Neste comando, são adicionados os comandos de atribuição antes da *label* $..for.headN$, para realizar a inicialização das variáveis de controle. Ao final do bloco $..for.bodyN$, são inseridos os comandos de atribuição que serão executados ao final de cada iteração.

O código 5 do Apêndice B exemplifica o código intermediário para estrutura gerada pelo compilador para o comando for. A figura 3 do Apêndice D mostra um grafo de fluxo de controle gerado pelo *LLVM* para o comando for.

Com base em todas as estruturas de geração de código intermediário descritas nesta seção, o compilador as combina para gerar o código intermediário completo do programa a ser compilado. O código intermediário completo gerado pelo compilador para o programa apresentado no código 1, localizado no Apêndice B, pode ser encontrado no código 2, também no Apêndice B.

3.2.3. Implementação do compilador

Para a implementação do compilador, as etapas de análise léxica e sintática são realizadas pelo *ANTLR*, que é integrado com a ferramenta de compilação *Maven*. Ao especificar os arquivos que definem as regras léxicas e sintáticas, o *ANTLR* gera automaticamente diversas classes, contendo a implementação do analisador da linguagem, que produz uma árvore de derivação do código-fonte.

Para navegar e manipular a árvore de derivação, o *ANTLR* permite duas abordagens: a primeira é o padrão *Listener*, em que o *ANTLR* gera uma classe para implementar os métodos que serão executados na entrada e saída de cada regra da gramática, com a navegação pela árvore realizada pelo próprio *ANTLR*. A segunda abordagem é o padrão *Visitor*, em que o *ANTLR* gera uma classe para implementar as chamadas de cada regra da gramática. Na implementação do *Visitor*, é necessário especificar um parâmetro de tipo genérico que serve como o tipo de retorno dos métodos. Além disso, no padrão *Visitor*, a navegação na árvore é realizada pela própria implementação.

A diferença entre as duas abordagens é que o padrão *Listener* funciona de forma passiva em relação à navegação na árvore, enquanto o *Visitor* permite maior flexibilidade e, por retornar valores pelos métodos, facilita a construção de estruturas de dados para o código a ser compilado. A abordagem escolhida para as implementações do compilador é o padrão *Visitor*.

Para a geração do código intermediário, foram criadas várias classes e interfaces que auxiliam na construção do IR. A principal delas é a interface $Fragment$, que representa um fragmento do IR (código 1 do Apêndice E). Esta interface possui apenas um método, $getText()$, o qual retorna a *String* correspondente ao fragmento. A interface $Fragment$ é usada como tipo de retorno na implementação do *Visitor*.

A partir da interface $Fragment$, foram derivadas quatro estruturas de dados que auxiliam na construção do código IR:

- $FragmentBlock$: representa um conjunto de $Fragments$, onde cada fragmento equivale a uma linha do IR (código 2 do Apêndice E);
- $ReturnableFragment$: uma classe abstrata que representa um $Fragment$ com

uma variável de retorno, usada para instruções do IR que produzem valores, como operações aritméticas e manipulação de variáveis (código 3 do Apêndice E);
- $ReturnableFragmentBlock$: um $FragmentBlock$ que contém uma variável de retorno de um valor produzido pelo bloco, utilizado para abstrair operações envolvendo múltiplos comandos IR, como expressões matemáticas ou carregamento de valores de *arrays* (código 4 do Apêndice E);
- $LabeledFragmentBlock$: representa um $FragmentBlock$ associado a uma *label*, facilitando a criação da estrutura para o cálculo de expressões condicionais com curto-circuito (código 5 do Apêndice E).

Com base nas estruturas $Fragment$ e $ReturnableFragment$, foram implementados os comandos do *LLVM IR*. Todas essas implementações foram projetadas para que seus atributos sejam imutáveis após a criação dos objetos para permitir uma maior resiliência no seu uso. O código 6 do Apêndice E ilustra a implementação do comando $load$. As demais estruturas são usadas pelo $Visitor$ para a construção do *IR*.

Para complementar as estruturas apresentadas, foram implementadas quatro classes de serviço para auxiliar na criação do *IR*:

- $ScopeManager$: gerencia os escopos do código-fonte. Possui métodos como $startScope()$ e $finishScope()$ para o controle dos escopos, além de $declareVariable(Varible)$ para declarar variáveis e $declareFunction(Function)$ para declarar funções. Também inclui métodos de consulta para dados declarados. O $ScopeManager$ atua como a tabela de símbolos;
- $SingleUseVariablesManager$: gerencia variáveis sequenciais usadas no *LLVM IR* por meio de um contador interno. Oferece métodos como $getNewVariableOfType(Type)$, que cria uma nova variável para ser utilizada;
- $StringManager$: gerencia as *strings* usadas no código-fonte, aplicando a técnica de *interning* e formatando-as conforme o padrão do *LLVM IR*. O método $getStringVariable(String)$ retorna uma variável contendo a *string* formatada, pronta para uso;
- $LabelManager$: gerencia as *labels* para evitar conflitos de nomenclatura, utilizando um $HashMap$ interno. Inclui o método $createLabel(String)$, que recebe o nome da *label* e retorna um objeto $Label$ para uso.

Todas essas estruturas são utilizadas em conjunto pelo *Visitor* para a geração do *LLVM IR*. Ao todo, são empregadas 43 classes em *Java* para realizar a tradução do código-fonte. O código 7 do Apêndice E apresenta a implementação da regra sintática de $atribuicao$.

Além da geração do *IR*, o compilador oferece funcionalidades adicionais, permitindo obter a lista de *tokens*, a árvore de sintática e a tabela de símbolos em formato *JSON*. A implementação dessas funcionalidades segue os mesmos padrões da geração do *IR*, empregando estruturas de dados específicas para cada objetivo. Já para a geração do *LLVM IR* otimizado, *Assembly* e código alvo, o compilador se integra com o compilador do *LLVM* para a geração desses dados.

3.2.4. Implementação da interface

A *API*, construída com o *framework Spring Boot*, expõe *endpoints* que se comunicam com o compilador para gerar os artefatos que são exibidos ao usuário. Para isso, a *API* disponibiliza um *endpoint POST* $compiler/upload$, que recebe o código-fonte e, utilizando o algoritmo de *hash SHA-256*, gera um identificador exclusivo denominado $codeId$. Em seguida, o código é salvo em uma pasta com o mesmo nome do $codeId$. Todos os artefatos gerados para cada $codeId$ são armazenados nessa pasta, permitindo o *cache* dos artefatos e exigindo o processamento apenas uma vez para cada artefato que não esteja salvo.

Também são expostos pela *API* os seguintes *endpoints*:

- *GET* $compiler/\{codeId\}$: Retorna o código-fonte com o mesmo $codeId$ especificado;
- *GET* $compiler/\{codeId\}/llvm/ir$: Retorna o código *LLVM IR* do código-fonte com o mesmo $codeId$ especificado;
- *GET* $compiler/\{codeId\}/llvm/ir/opt/\{optLevel\}$: Retorna o código *LLVM IR* otimizado com o nível de otimização especificado do código-fonte com o mesmo $codeId$ especificado;
- *GET* $compiler/\{codeId\}/asm$: Retorna o código *Assembly* do código-fonte com o mesmo $codeId$ especificado;
- *GET* $compiler/\{codeId\}/asm/opt/\{optLevel\}$: Retorna o código *Assembly* otimizado com o nível de otimização especificado do código-fonte com o mesmo $codeId$ especificado;
- *GET* $compiler/\{codeId\}/syntax$: Retorna a árvore sintática em formato *JSON* do código-fonte com o mesmo $codeId$ especificado;
- *GET* $compiler/\{codeId\}/token$: Retorna a lista de *tokens* em formato *JSON* do código-fonte com o mesmo $codeId$ especificado;
- *GET* $compiler/\{codeId\}/symbols$: Retorna a tabela de símbolos em formato *JSON* do código-fonte com o mesmo $codeId$ especificado.

Além disso, a *API* oferece a funcionalidade de executar o código compilado de forma remota em tempo real via protocolo *WebSocket*. Ao iniciar uma nova sessão, um novo processo no sistema operacional é criado, e três *threads* são executadas para gerenciar o processo: duas para ler os dados de saída (*stdout*) e de erro (*stderr*) e enviá-los em tempo real pelo *WebSocket*, e uma terceira *thread* para detectar a finalização do programa, obter o código de saída e enviá-lo também pelo *WebSocket*. Os objetos dos processos são armazenados em um $ConcurrentHashMap$, tendo como chave o ID da sessão, o que permite o envio de dados de entrada em tempo real aos processos. Quando a sessão é encerrada, o processo é finalizado automaticamente.

O *frontend*, implementado com o *framework Vue* e as bibliotecas de componentes *Vuetify 3* e *CodeMirror* para o editor de código, apresenta em sua tela inicial um menu lateral retrátil com informações sobre a linguagem, funcionando como um guia rápido para consulta das funcionalidades disponíveis. A interface também conta com um editor de código onde o usuário pode inserir o código-fonte. É possível selecionar exemplos prontos através do botão de exemplos, que incluem recursos básicos da linguagem, exemplos com recursão, algoritmos de ordenação e jogos. Ao clicar no botão de compilar, o *frontend* se comunica com a *API* para obter o $codeId$, verificando se o código digitado é

válido e, caso positivo, redireciona o usuário para as próximas telas. Para melhorar a experiência, o código digitado é salvo no *LocalStorage*, garantindo que o usuário não perca o progresso ao retornar à tela inicial. A figura 10 apresenta a tela inicial.

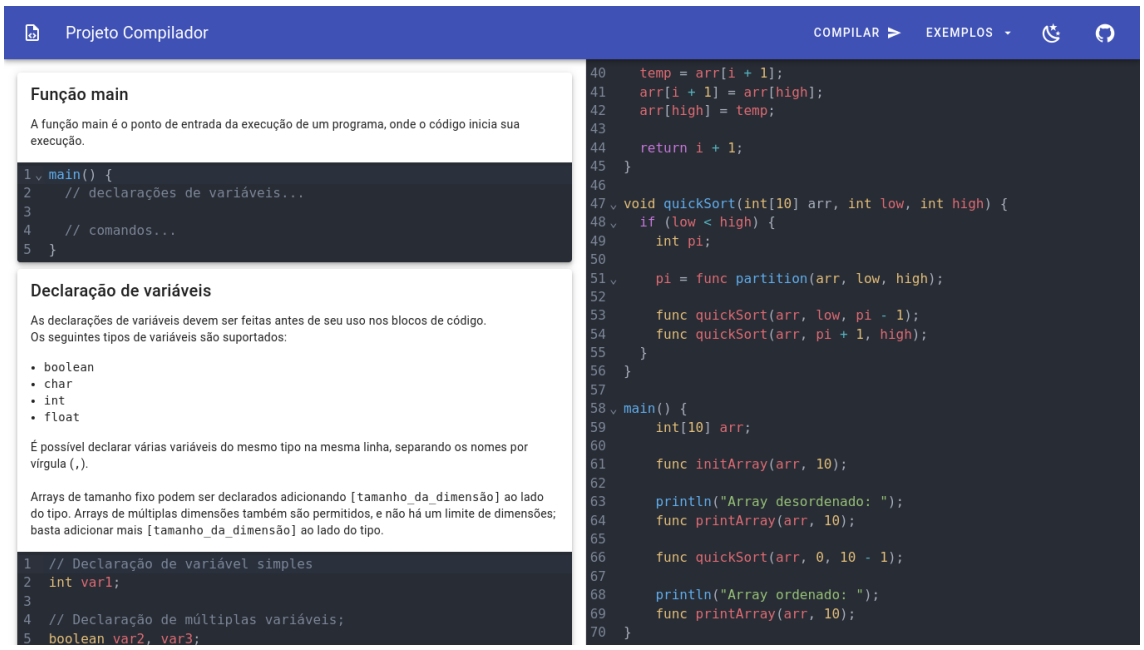


Figura 10. Tela inicial do programa.

A primeira tela exibida após a compilação do programa é a de lista de *tokens* (figura 11). Ao posicionar o *mouse* sobre um *token*, o usuário pode visualizar sua localização correspondente no código-fonte. Em seguida, a tela de árvore sintática, desenvolvida com a biblioteca *D3.js* [11], permite que, ao posicionar o *mouse* sobre um nó da árvore, a região correspondente seja destacada no código-fonte (figura 1 do Apêndice F). A próxima tela exibe a tabela de símbolos, mostrando as funções, escopos e *strings* declaradas no programa, permitindo também a seleção da região correspondente no código (figura 2 do Apêndice F).

A tela seguinte apresenta o código *LLVM IR*, onde é possível visualizar o código intermediário em diferentes níveis de otimização, selecionados através do menu lateral (figura 3 do Apêndice F). Há também um modo de comparação que permite analisar as diferenças entre dois níveis de otimização. Na tela seguinte, o código *Assembly* é exibido com os mesmos recursos de otimização e comparação (figura 4 do Apêndice F). As configurações de otimização e comparação são salvas como parâmetros na URL, permitindo que um usuário compartilhe facilmente suas configurações com outros.

A última tela do módulo do compilador permite a execução do código (figura 5 do Apêndice F). Nessa interface, o terminal exibe as saídas do programa em tempo real, enquanto o campo de texto na parte inferior permite que o usuário envie dados de entrada. O menu lateral oferece as opções de reiniciar o programa e limpar o terminal. Essa funcionalidade é implementada via conexão com a API de execução através de *WebSocket*.

[11] `https://d3js.org/`

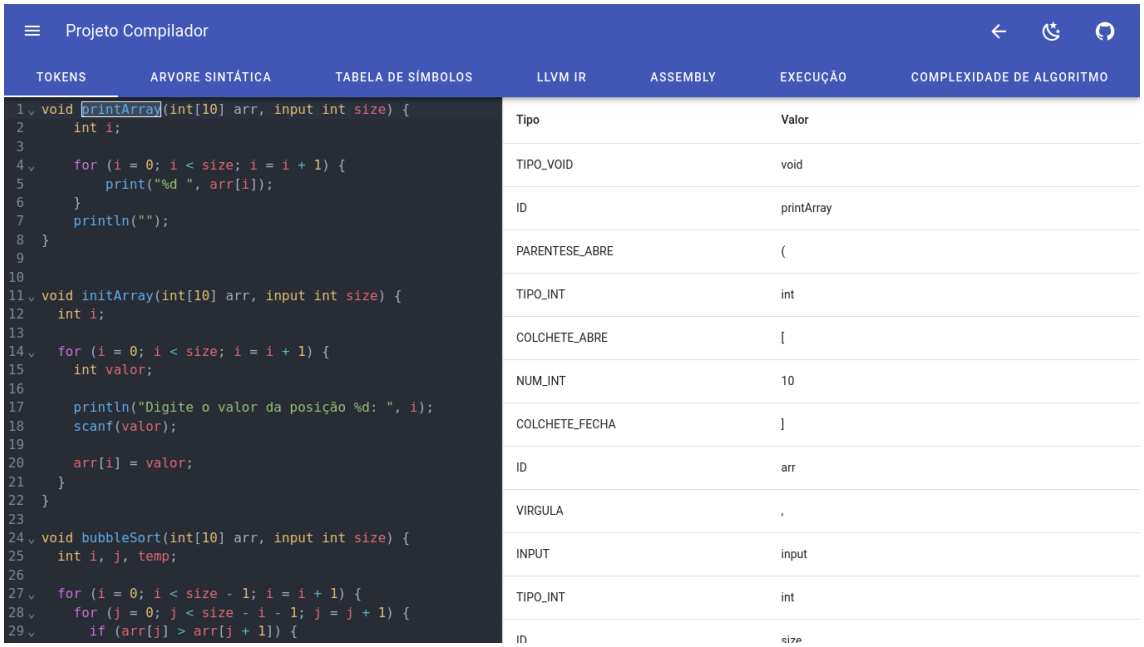


Figura 11. Tela de exibição da lista de *tokens*, com abas na parte superior que permitem visualizar também as demais interfaces.

3.3. Módulo de análise de complexidade

O módulo de análise da complexidade dos algoritmos adota uma estrutura semelhante ao módulo de compilação, utilizando o *ANTLR* com o padrão *Visitor* e a mesma gramática definida para o compilador. Essa abordagem permite a realização de uma análise estática do programa, calculando sua complexidade algorítmica na forma da função $T(n)$, onde n representa o tamanho da entrada do algoritmo. A função é determinada com base na soma dos custos individuais de cada instrução presente no programa.

Os custos das instruções utilizadas para os cálculos seguem as diretrizes apresentadas em Cormen et al. (2012). Segundo o autor, declarações de variáveis, funções e a instrução $return$ tem um custo 0. Já os comandos de atribuição de valores, leitura e escrita de dados tem custo 1. Para os blocos condicionais, é considerado apenas o bloco com o maior custo, ignorando o bloco com o custo menor.

Para identificar os valores de entrada que impactam a análise de complexidade, a linguagem desenvolvida permite o uso de um modificador $input$ na definição dos parâmetros de funções. Esse modificador deve ser colocado antes do tipo do parâmetro, indicando que ele será considerado como um fator de crescimento nas análises de complexidade. O código 19 mostra como o $input$ é usado nas declarações de funções.

```
1 int funcao(input int n) {
2   // ...
3 }
```

Código 19. Exemplo de uso do modificador $input$ para parâmetros de variáveis.

Para a realização do cálculo, diversas interfaces e classes foram desenvolvidas. A principal interface é a $CostResult$, que representa o custo de uma determinada parte do programa e é usada como o tipo de retorno dos métodos da implementação do *visitor*.

A interface possui a definição de um método para retornar a posição no código referente àquele custo e uma definição de um método para retornar o custo em si (código 1 do Apêndice G).

Três classes implementam a interface $CostResult$:

- $Cost$: Representa um custo simples, sendo usada para representar os custos de cada comando (código 2 do Apêndice G).
- $BlockCost$: Representa um bloco de custos com um custo total associado a ele. É usado para representar o custo de uma função e de blocos condicionais (código 3 do Apêndice G).
- $VariableCost$: Representa um custo com uma variável associada. É usado no cálculo de blocos de repetição, onde geralmente a variável de entrada é utilizada pela classe (código 4 do Apêndice G).

A implementação do *visitor*, utilizando as classes apresentadas anteriormente, realiza uma varredura no código, construindo uma estrutura que registra o valor e a posição no código de cada custo identificado. Para cada bloco de código, os comandos são processados e depois somados ao custo total da bloco. O último bloco de cada cálculo corresponde ao bloco da função, onde são reunidos os cálculos de todos os valores e blocos pertencentes a ela, resultando na definição final de $T(n)$ da função.

O código 5, apresentado no Apêndice G, demonstra a implementação do método do *visitor* responsável pelo cálculo da complexidade do bloco for.

Todos os registros dos cálculos são convertidos para um documento no formato *JSON* e disponibilizados pela *API* no *endpoint GET* $/compiler/\{codeId\}/complexity$. Esses dados são utilizados no *frontend* para exibir os resultados da análise de complexidade do programa. Na tela de exibição, o editor de código apresenta marcações em cada linha do programa, indicando o custo associado a ela, enquanto ao final de cada bloco é exibida a expressão matemática correspondente ao cálculo daquele bloco.

No menu lateral, além de listar os custos atribuídos a cada comando, são exibidas, para cada função, a fórmula completa de $T(n)$ com a soma de todos os custos, a sua versão simplificada em forma de polinômio e ao fim o valor do *Big O* do algoritmo. Como o *backend* não realiza manipulações algébricas diretamente nas fórmulas geradas, a simplificação e a identificação do grau do polinômio do $T(n)$ para o valor no *Big O* é realizada pela biblioteca de *Computer Algebra System* (*CAS*) *Nerdamer* [12]. Abaixo, um exemplo do processo que é exibido em tela:

$$\begin{aligned} & T(n) = (n+1) \cdot ((n+1) \cdot (1+1)) \\ & \downarrow \text{ simplificação} \\ & T(n) = 2 \cdot n^2 + 4 \cdot n + 2 \\ & \downarrow \text{ notação assintótica} \\ & O(n^2) \end{aligned} \tag{1}$$

A biblioteca *Nerdamer* também é capaz de converter as fórmulas geradas para o formato *TeX*, possibilitando a exibição das expressões em um formato adequado para

[12] `https://nerdamer.com/`

visualização matemática. Para renderizar as fórmulas em *TeX*, foi utilizada a biblioteca *KaTeX* [13].

A figura 12 mostra a tela de exibição do cálculo da complexidade de um programa com a implementação de funções para a manipulações de matrizes bidimensionais.

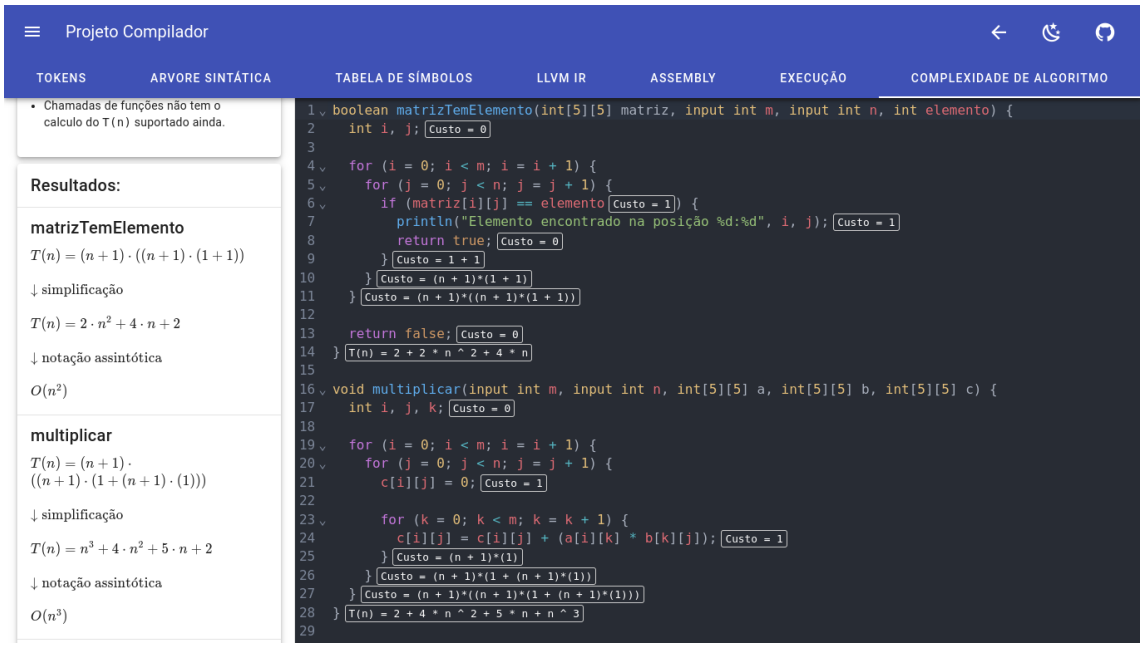


Figura 12. Tela de exibição do cálculo da complexidade de um programa com a implementação de funções para a manipulações de matrizes bidimensionais.

3.3.1. Limitações

O cálculo da complexidade de algoritmos apresenta algumas limitações em sua implementação. Para os laços de repetição for e $while$, são considerados apenas valores constantes e variáveis marcadas com o modificador $input$, desde que sigam o padrão em que a variável de controle é incrementada até atingir o valor da variável de entrada. Casos em que o limite da iteração depende de cálculos de expressões matemáticas envolvendo a variável de entrada não são suportados. Além disso, a implementação assume que o incremento da variável de controle será sempre de 1.

A funcionalidade suporta apenas o cálculo da complexidade de funções individuais, não incluindo cenários em que uma função realiza chamadas para outras funções ou utiliza chamadas recursivas. Também não há suporte para cálculos envolvendo variáveis de entrada na função $main$.

4. Resultados

Para avaliar o sistema desenvolvido, foi elaborado um questionário *online* anônimo utilizando a ferramenta *Google Forms* [14]. O questionário incluía um vídeo introdutório, que

[13] https://katex.org/
[14] https://docs.google.com/forms/

apresentava o sistema e demonstrava seu funcionamento, um *link* para acesso ao sistema publicado e 15 perguntas. Entre essas, 12 eram obrigatórias de múltipla escolha baseadas na escala Likert (Nemoto e Beglar, 2014), uma pergunta obrigatória de múltipla escolha de avaliação geral do sistema e duas perguntas abertas opcionais.

O questionário esteve disponível para respostas entre os dias 23/11/2024 e 30/11/2024. Ele foi encaminhado para estudantes do curso de Ciências da Computação do IFSC Campus Lages e para a lista de membros associados à Sociedade Brasileira de Computação (SBC). Ao final do período, foram recebidas 27 respostas.

A primeira pergunta do questionário abordou o perfil dos avaliadores em relação às disciplinas de compiladores e análise da complexidade de algoritmos, como mostra a figura 13.

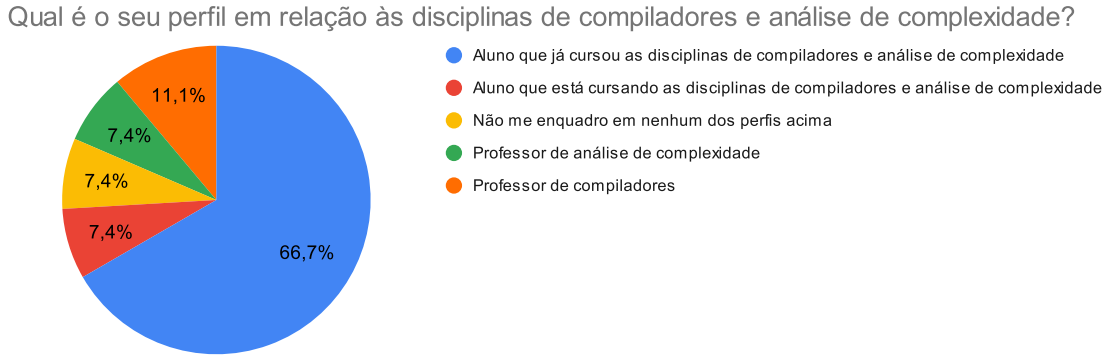


Figura 13. Gráfico referente ao perfil de quem estava avaliando o sistema em relação às matérias de compiladores e de análise da complexidade de algoritmos.

As três perguntas seguintes, cujos resultados estão na figura 14, avaliaram a usabilidade das interfaces de listagem de *tokens*, visualização da árvore sintática e tabela de símbolos. A funcionalidade de visualização dos *tokens* recebeu uma boa avaliação, com $96,3\%$ de respostas positivas. Já a funcionalidade de visualização da árvore sintática obteve $88,89\%$ de respostas positivas, enquanto a funcionalidade de visualização da tabela de símbolos registrou $92,6\%$ de respostas positivas. Embora os resultados sejam satisfatórios, as notas ligeiramente menores dessas duas funcionalidades reforçam a relevância das sugestões feitas nas questões descritivas, como melhorias na apresentação da árvore sintática e maior clareza na visualização da tabela de símbolos.

Os resultados das quatro perguntas subsequentes, apresentados na figura 15, focaram nas funcionalidades de visualização e comparação dos diferentes níveis de otimização do *LLVM IR* e *Assembly*. De forma geral, essas funcionalidades receberam avaliações positivas em mais de 80% das respostas. Contudo, houve um número considerável de respostas marcadas como "Não sei responder", correspondendo, em média, a 13% das respostas para essas questões. Esse resultado sugere que tais funcionalidades demandam um nível mais elevado de entendimento técnico e conhecimentos específicos, os quais muitas vezes não são explorados em profundidade em sala de aula. Isso reforça que a ferramenta, por si só, não substitui o ensino de novos conteúdos, sendo necessário um aprendizado prévio para que os usuários possam compreender plenamente esses recursos.

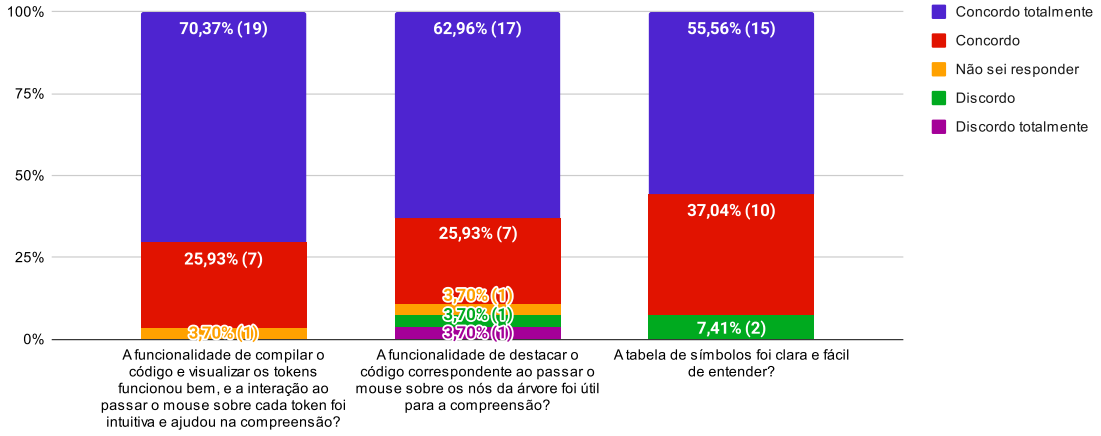


Figura 14. Gráfico com o resultado obtido da avaliação da intuitividade das telas da listagem de *tokens*, visualização da árvore sintática e da tabela de símbolos.

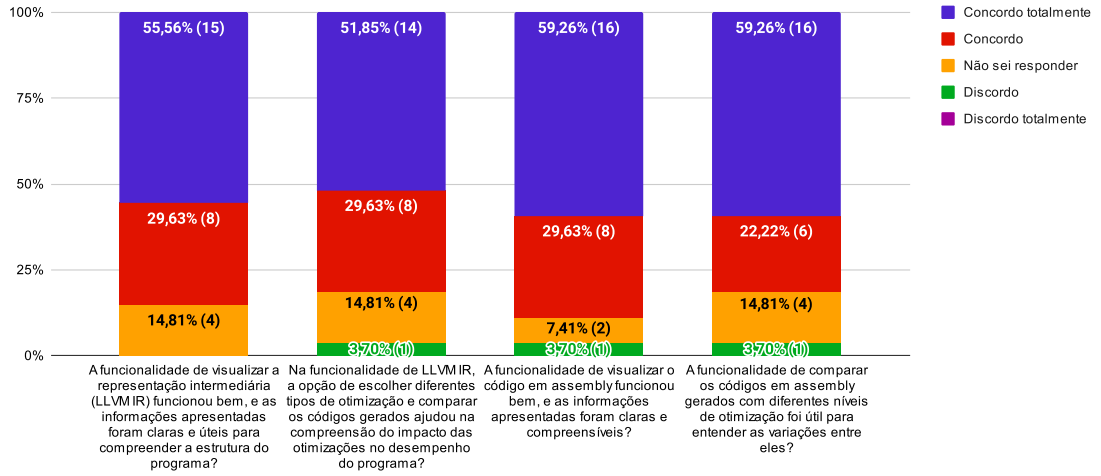


Figura 15. Gráfico com o resultado obtido da avaliação das funcionalidades de visualização e comparação de otimizações do *LLVM IR* e *Assembly*.

A figura 16 exibe as respostas relacionadas às funcionalidades específicas de execução dos códigos no sistema e a análise da complexidade de algoritmos. A funcionalidade de execução de código obteve $92,59\%$, já a funcionalidade de visualização da análise da complexidade de algoritmos alcançou $88,86\%$ de respostas positivas, indicando que ambas as funcionalidades foram bem aceitas. Embora a funcionalidade de análise da complexidade de algoritmos ainda não esteja completamente implementada, os resultados indicam que o desenvolvimento segue uma direção correta e bem-recebida pelos usuários.

Por fim, duas perguntas gerais abordaram as características do sistema e sua aplicabilidade fora da sala de aula, com os resultados mostrados na figura 17 (a). Além disso, os avaliadores atribuíram uma nota geral ao sistema, variando de 1 a 5, cujo resultado está ilustrado na figura 17 (b). As respostas obtidas nas questões gerais foram amplamente positivas, com apenas uma resposta neutra em relação à utilidade da ferramenta fora da sala de aula. A avaliação geral do sistema confirmou o alto nível de aceitação, alcançando

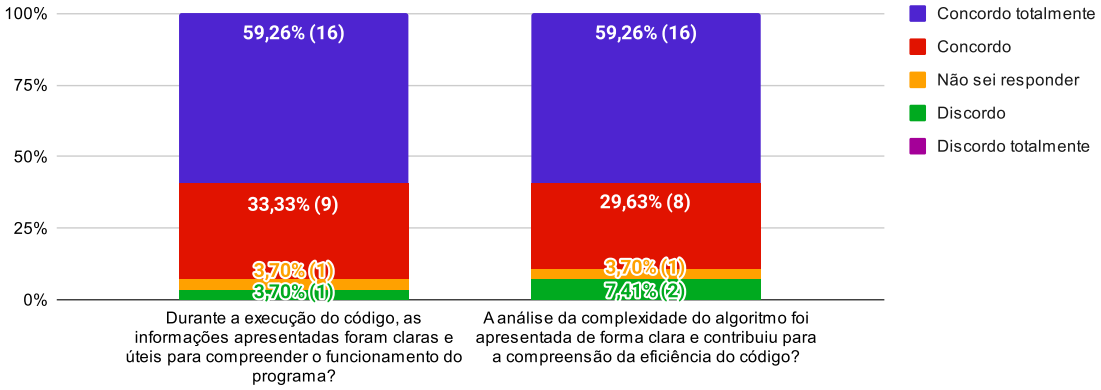
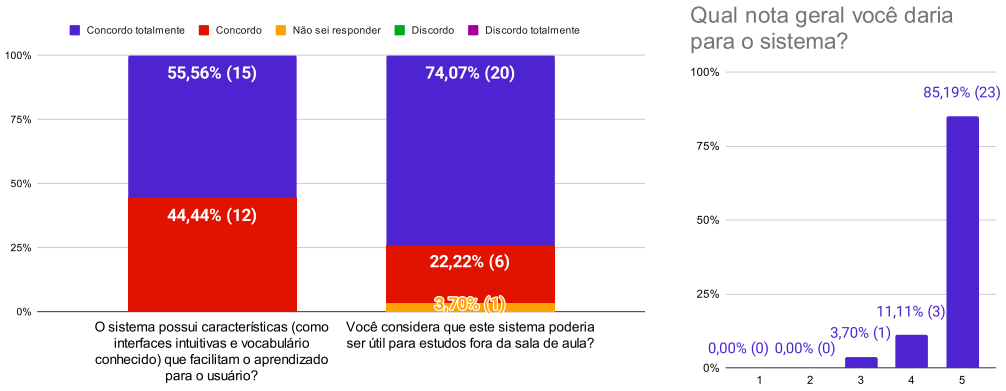


Figura 16. Gráfico com o resultado sobre a funcionalidade de execução dos códigos e sobre a funcionalidade da análise da complexidade de algoritmos.



(a) Gráfico com o resultado sobre a avaliação do sistema e a sua utilidade na sala de aula.

(b) Gráfico com as notas obtidas do sistema.

Figura 17. Gráficos com os resultados obtidos da avaliação geral do sistema desenvolvido.

uma média de $4,81$ em 5, evidenciando o potencial da ferramenta.

A primeira questão aberta do questionário visava coletar *feedbacks* relacionados a aspectos negativos do sistema com o enunciado: "Informe o que você mudaria ou aquilo que você não gostou no sistema". Alguns dos principais pontos levantados foram:

- Melhoria na cor de destaque dos trechos de código selecionados;
- Melhoria na visualização da árvore sintática, apresentando a árvore na horizontal, permitindo zoom e movimentação e sendo possível recolher e expandir os nós da árvore;
- Melhoria no cálculo da complexidade para considerar mais casos.

Com base nas sugestões recebidas, algumas das melhorias indicadas já foram implementadas após o encerramento do período de coleta de respostas. Sendo elas: a melhoria na visibilidade dos códigos selecionados, a adição ao suporte para dar zoom e mover a árvore sintática que também passou a ser exibida na vertical e a adição para a exibição do cálculo da complexidade na notação assintótica *Big O*.

A segunda questão aberta tinha como objetivo identificar os pontos positivos do sistema, apresentada com o enunciado: ”Informe o que você gostou no sistema”. Dentre os pontos levantado, se destaca a facilidade de compreensão, interface e utilização do sistema.

Entre algumas das respostas positivas, temos:

- “A proposta é muito interessante para que o aluno da disciplina de compiladores entenda o funcionamento de um compilador. Caso seja interessante para vocês, pensem em fazer mestrado na área e pensem na UFRN, que tem uma linha de pesquisa de linguagens de programação e métodos formais ;).”;
- “Gostei de tudo mesmo, principalmente dos analisadores. Sou professor da disciplina de compiladores há 10 anos e definitivamente gostaria muito de usar essa ferramenta em sala de aula.”;
- “É um ótimo sistema. Muito bem organizado e com ferramentas muito úteis. Teria sido muito bom ter uma ferramenta assim quando estudei compiladores”.

Essas respostas, vindas de alunos e professores, evidenciam o grande potencial da ferramenta como um recurso educacional em sala de aula. Além de apoiar os professores nas explicações dos conteúdos, o sistema também se destaca como um material interativo para os alunos, permitindo a visualização gráfica dos processos apresentados.

5. Conclusão

Este trabalho teve como objetivo desenvolver uma ferramenta para auxiliar no ensino de compiladores e análise da complexidade de algoritmos, abordando a definição da linguagem suportada pelo compilador, o processo de compilação e tradução da linguagem para o *LLVM IR*, o funcionamento do *LLVM*, o cálculo da complexidade dos algoritmos e as formas de interação dos usuários com o programa.

As análises léxica e sintática foram implementadas utilizando o *ANTLR 4*, que, a partir da definição das regras da gramática, gera automaticamente o analisador na linguagem *Java*. Com o uso do padrão *Visitor*, disponibilizado pelo analisador gerado, foi possível implementar a geração da lista de *tokens*, a construção da árvore sintática, a montagem da tabela de símbolos e a tradução do código para o *LLVM IR*. Esse padrão proporcionou uma abordagem estruturada e organizada, pois o *Visitor* inclui um método específico para cada regra da gramática, além de já fornecer um tratamento para erros léxicos e sintáticos do código analisado.

Após a geração do código intermediário, o *LLVM* é responsável por realizar otimizações em diferentes níveis no código intermediário, além de gerar o *Assembly* e criar o executável final. O *LLVM* demonstrou ser uma ferramenta poderosa para a construção de compiladores, por seu suporte a diversos sistemas operacionais e arquiteturas de processadores. Ele é capaz de realizar uma ampla variedade de otimizações no código, incluindo algumas específicas para as características do sistema de destino. Além disso, o *LLVM* permite a obtenção dos códigos resultantes após as etapas de otimização e geração do *Assembly*, os quais são apresentados na interface do sistema implementado.

Os artefatos gerados são disponibilizados por meio de uma *API REST* desenvolvida com o *framework Spring Boot* que atua como interface entre a execução do código alvo no sistema operacional e o sistema *web*. A interface *web* foi implementada utilizando

o *framework Vue*, em conjunto com a biblioteca de componentes *Vuetify*, além de outras bibliotecas auxiliares, como o *D3.js* para a exibição gráfica da árvore sintática e as bibliotecas *Nerdamer* e *KaTeX* para auxiliar na visualização dos cálculos de complexidade dos algoritmos. A integração desses *frameworks* e bibliotecas possibilitou um desenvolvimento simplificado do sistema *web*, com recursos visuais úteis para apresentar cada etapa do processo de compilação.

A avaliação do sistema foi realizada através de um formulário *web* junto com um vídeo auxiliar explicando o funcionamento do programa. Durante o período de uma semana em que o questionário permaneceu aberto, foram recebidas 27 respostas de alunos e professores.

A análise geral das respostas mostrou que a aceitação da ferramenta foi bastante positiva, com uma nota média atribuída pelos usuários de $4,81$ em 5. Após o período de coleta de respostas, algumas melhorias foram implementadas no sistema, com base nos comentários recebidos, como destacado no final da seção 4. Os demais *feedbacks* foram considerados na definição das atividades a serem realizadas em trabalhos futuros. A análise individual das respostas revelou que os professores de compiladores avaliaram o sistema de forma mais positiva em comparação aos professores de análise da complexidade de algoritmos. Esse resultado era esperado, dado que as funcionalidades relacionadas a compiladores estavam mais consolidadas.

Como trabalhos futuros, o compilador pode ser aprimorado para suportar uma linguagem com mais recursos, como a inclusão de $structs$ ou suporte ao paradigma orientado a objetos. Permitir a alocação de objetos na *heap* do programa, com a implementação de um *Garbage Collector*, possibilitando, por exemplo, que *arrays* tenham suas dimensões definidas em tempo de execução, e não apenas em tempo de compilação. A biblioteca padrão da linguagem também pode ser expandida, expondo mais funções da *glibc* e aproveitando funções intrínsecas do *LLVM*. Na geração do código alvo, pode ser adicionado o suporte à geração do *Assembly* para sistemas operacionais e arquiteturas de processadores diferentes, possibilitando que a ferramenta também seja usada em disciplinas que abordam outros conteúdos, como Arquitetura de Computadores e Programação *Assembly*.

As interfaces *web* podem ser melhoradas com explicações teóricas sobre cada etapa do compilador, auxiliando no aprendizado. O tratamento de erros no processo de compilação pode ser aprimorado, apresentando mensagens mais claras sobre os problemas e destacando visualmente a linha do código onde o erro ocorreu. Adicionar um modo de depuração na execução do código permitiria aos usuários executar linha por linha, inspecionando o conteúdo das variáveis para identificar problemas mais facilmente.

No módulo de análise da complexidade, é possível implementar suporte para casos atualmente não cobertos, como estruturas de repetição em que o limite depende de cálculos matemáticos e pode ser estendido para cenários com chamadas de funções, inclusive recursivas. Além disso, seria útil que o programa exibisse o cálculo da complexidade em outras notações assintótica, como *Big* Ω e *Big* Θ. A exibição de gráficos da função $T(n)$ permitiria comparar visualmente os custos de diferentes algoritmos, fornecendo uma representação mais intuitiva de suas eficiências relativas. Por fim, apesar de o sistema atual mostrar o passo a passo da obtenção dos custos até o valor do *Big O*,

esse recurso está em um estágio inicial e pode ser expandido para incluir uma abordagem interativa, com visualizações gráficas detalhadas e explicações teóricas em cada etapa, facilitando ainda mais o entendimento do processo por parte dos usuários.

Referências

Aho, A. V., Lam, M. S., Sethi, R., e Ullman, J. D. (2008). *Compiladores: Princípios, técnicas e ferramentas*. Pearson.

Backes, A. (2016). *Estrutura de dados descomplicada em linguagem C*. Elsevier.

Badnjević, A., Cifrek, M., e Koruga, D. (2013). Integrated software suite for diagnosis of respiratory diseases. In *Eurocon 2013*, pages 564–568.

Barbosa, M. A. C., Toscani, L. V., e Ribeiro, L. (2001). Anac–uma ferramenta para análise automática da complexidade de algoritmos. *ResearchGate*.

Cooper, K. D. e Torczon, L. (2014). *Construindo compiladores*. Campus.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., e Stein, C. (2012). *Algortimos: Teoria e Prática*. Elsevier.

Costa, E. J. F., Ramos, J. G. G. d. S., Barbosa, Y. d. A. M., de S Filho, G. F., e Brito, A. V. (2014). Um avaliador automático de eficiência de algoritmos para ambientes educacionais de ensino de programação. *Anais do Computer on the Beach*, pages 11–21.

Graciano Junior, W., Grossert, I., Neto, W. C. B., e Avila, A. (2022). Ferramenta interativa para o ensino de compiladores. In *Anais do II Simpósio Brasileiro de Educação em Computação*, pages 224–233, Porto Alegre, RS, Brasil. SBC.

Gramond, E. e Rodger, S. H. (1999). Using jflap to interact with theorems in automata theory. *SIGCSE Bull.*, 31(1):336–340.

Lattner, C. e Adve, V. (2004). Llvm: a compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, pages 75–86.

Mernik, M. e Zumer, V. (2003). An educational tool for teaching compiler construction. *IEEE Transactions on Education*, 46(1):61–68.

Nemoto, T. e Beglar, D. (2014). Developing likert-scale questionnaires. In *JALT2013*.

Queiros, L. M., Gomes, A. S., Pereira, J. W., Castro Filho, J. A. d., Santos, E. M. d., e Silva Neto, D. F. d. (2022). Enigmas de yucatàn: Recurso educacional digital para o ensino de geometria espacial. *Revista Brasileira de Informática na Educação*, 30:108–134.

Rese, A. L. R. e Santiago, R. (2012). Ensino de teoria da complexidade. In *Anais do XXXII Congresso da Sociedade Brasileira de Computação*.

Scheider, C., Passerino, L. M., e Oliveira, R. F. d. (2005). Compilador educativo verto: ambiente para aprendizagem de compiladores. *Revista Novas Tecnologias na Educação*, 3(2).

Singal, P. e Chhillar, R. S. (2014). Dijkstra shortest path algorithm using global positioning system. In *International Journal of Computer Applications*, volume 101, pages 12–18.

Toscani, L. V. e Veloso, P. A. S. (2012). *Complexidade de algoritmos*. Bookman.

Weber, R. F. (2001). Fundamentos de arquitetura de computadores. *Porto Alegre: Sagra Luzzato*, page 248.

```
lexer grammar LexerGrammar;

MAIN : 'main' ;
SCANF : 'scanf' ;
PRINTLN : 'println' ;
PRINT : 'print' ;
IF : 'if' ;
ELSE : 'else' ;
WHILE : 'while' ;
FOR : 'for' ;
FUNC : 'func' ;
RETURN : 'return';
INPUT : 'input';

TIPO_VOID : 'void' ;
TIPO_CHAR : 'char' ;
TIPO_FLOAT : 'float' ;
TIPO_INT : 'int' ;
TIPO_BOOLEAN : 'boolean' ;

VIRGULA : ',' ;
PONTO_VIRGULA : ';' ;

PARENTESE_ABRE : '(';
PARENTESE_FECHA : ')';

COLCHETE_ABRE : '[';
COLCHETE_FECHA : ']';

CHAVE_ABRE : '{';
CHAVE_FECHA : '}';

TEXTO : '"'.*?'"'|'\''.*?'\'';

SINAL_MAIS : '+' ;
SINAL_MENOS : '-' ;

OP_MULTIPLICACAO : '*' ;
OP_DIVISAO : '/' ;
OP_RESTO_DIVISAO : '%' ;

OP_NEGACAO : '!' ;
OP_E : '&&' ;
OP_OU : '||' ;

OP_IGUAL : '==' ;
OP_DIFERENTE : '!=' ;

OP_MAIOR : '>';
OP_MAIOR_IGUAL : '>=';

OP_MENOR : '<' ;
OP_MENOR_IGUAL : '<=' ;

```

```
55 OP_ATRIBUICAO : '=' ;
56
57 NUM_INT : [0-9]+ ;
58 NUM_DEC : [0-9]+'.'[0-9]+;
59
60 TRUE: 'true';
61 FALSE: 'false';
62
63 ID: [a-zA-Z_][a-zA-Z_0-9]* ;
64 WS: [ \t\n\r\f]+ -> skip ;
65
66 COMENTARIO_LINHA: '//' ~[\r\n]* -> skip ;
67 COMENTARIO_BLOCO: '/*' .*? '*/' -> skip ;
```

Código 1. Gramática léxica usado pelo ANTLR4 para o reconhecimentos dos Tokens da linguagem.

```
1  parser grammar ParserGrammar;
2  options { tokenVocab=LexerGrammar; }
3
4  @header {
5      import ifsc.compiladores.projeto.LLVM.translator.definitions.Label;
6  }
7
8  programa
9      : (decfuncao)* principal
10     ;
11
12 decfuncao
13     : tiporetorno ID PARENTESE_ABRE (parametros)? PARENTESE_FECHA bloco
14     ;
15
16 tiporetorno
17     : tipo
18     | TIPO_VOID
19     ;
20
21 tipo
22     : tipobase (dimensao)*
23     ;
24
25 tipobase
26     : TIPO_CHAR
27     | TIPO_FLOAT
28     | TIPO_INT
29     | TIPO_BOOLEAN
30     ;
31
32 dimensao
33     : COLCHETE_ABRE NUM_INT COLCHETE_FECHA
34     ;
35
36 parametros
37     : (INPUT)? tipo ID (VIRGULA tipo ID)*
38     ;
39
```

```
principal
    : MAIN PARENTESE_ABRE PARENTESE_FECHA bloco
    ;

bloco
    : CHAVE_ABRE (decvariavel)* (comando)* CHAVE_FECHA
    ;

decvariavel
    : tipo ID (VIRGULA ID)* PONTO_VIRGULA
    ;

comando
    : comando_linha PONTO_VIRGULA
    | comando_bloco
    ;

comando_linha
    : leitura       #ComandoLinhaLeitura
    | escritaln     #ComandoLinhaEscritaLn
    | escrita       #ComandoLinhaEscrita
    | atribuicao    #ComandoLinhaAtribuicao
    | funcao        #ComandoLinhaFuncao
    | retorno       #ComandoLinhaRetorno
    ;

comando_bloco
    : selecao
    | enquanto
    | para
    ;

leitura
    : SCANF PARENTESE_ABRE acesso_id PARENTESE_FECHA
    ;

dimensao2
    : COLCHETE_ABRE expr_aditiva COLCHETE_FECHA
    ;

escrita
    : PRINT PARENTESE_ABRE TEXTO (VIRGULA termoescrita)*
       PARENTESE_FECHA
    ;

escritaln
    : PRINTLN PARENTESE_ABRE TEXTO (VIRGULA termoescrita)*
       PARENTESE_FECHA
    ;

termoescrita
    : TEXTO         #TermoEscritaTexto
    | expressao     #TermoEscritaExpressao
    ;

selecao
```

```
94      : IF PARENTESE_ABRE expressao PARENTESE_FECHA bloco (senao)?
95      ;
96
97  senao
98      : ELSE bloco
99      ;
100
101 enquanto
102     : WHILE PARENTESE_ABRE expressao PARENTESE_FECHA bloco
103     ;
104
105 para
106     : FOR PARENTESE_ABRE (atribuicaoInicio=para_atribuicoes)?
          PONTO_VIRGULA (expressao)? PONTO_VIRGULA (atribuicaoFinal=
          para_atribuicoes)? PARENTESE_FECHA bloco
107     ;
108
109 atribuicao
110     : acesso_id OP_ATRIBUICAO complemento
111     ;
112
113 para_atribuicoes
114     : atribuicao (VIRGULA atribuicao)*
115     ;
116
117 complemento
118     : expressao
119     ;
120
121 funcao
122     : FUNC ID PARENTESE_ABRE (argumentos)? PARENTESE_FECHA
123     ;
124
125 argumentos
126     : expressao (VIRGULA expressao)*
127     ;
128
129 retorno
130     : RETURN (expressao)?
131     ;
132
133 expressao
134     : expr_ou
135     ;
136
137 expr_ou
138     : expr_e (OP_OU expr_e)*
139     ;
140
141 expr_e locals [Label label, Label trueLabel, Label falseLabel]
142     : expr_relacional (OP_E expr_relacional)*
143     ;
144
145 expr_relacional locals [Label label, Label trueLabel, Label falseLabel]
146     : expr_aditiva (op_relacional expr_aditiva)*
147     ;
```

```
148
149 op_relacional
150     : OP_IGUAL
151     | OP_DIFERENTE
152     | OP_MAIOR
153     | OP_MENOR
154     | OP_MAIOR_IGUAL
155     | OP_MENOR_IGUAL
156     ;
157
158 expr_aditiva
159     : expr_multiplicativa (op_aditivo expr_multiplicativa)*
160     ;
161
162 op_aditivo
163     : SINAL_MAIS
164     | SINAL_MENOS
165     ;
166
167 expr_multiplicativa
168     : fator (op_multiplicativo fator)*
169     ;
170
171 op_multiplicativo
172     : OP_MULTIPLICACAO
173     | OP_DIVISAO
174     | OP_RESTO_DIVISAO
175     ;
176
177 fator
178     : (sinal)? termo                                        #FatorTermo
179     | TEXTO                                                 #FatorText
180     | OP_NEGACAO fator                                      #
          FatorNegacaoFator
181     | (sinal)? PARENTESE_ABRE expressao PARENTESE_FECHA #FatorExpressao
182     ;
183
184 termo
185     : acesso_id #TermoVariavel
186     | constante #TermoConstante
187     | funcao    #TermoFuncao
188     ;
189
190 sinal
191     : SINAL_MAIS
192     | SINAL_MENOS
193     ;
194
195 constante
196     : NUM_INT
197     | NUM_DEC
198     | TRUE
199     | FALSE
200     ;
201
202 acesso_id
```

```
203     : ID                  #AcessoId
204     | ID (dimensao2)+     #AcessoIdArray
205     ;
```

Código 2. Gramática contendo as regras de derivação usado pelo ANTLR4 para o reconhecimento sintático da linguagem.

```
1  void lerNotas(int[5] notas) {
2    int i;
3
4    for (i = 0; i < 5; i = i + 1) {
5      int nota;
6
7      println("Digite a %d nota do aluno:", i + 1);
8      scanf(nota);
9
10     notas[i] = nota;
11   }
12 }
13
14 void exibirNotas(int[5] notas) {
15   int i;
16
17   println("As notas do aluno sao:");
18
19   for (i = 0; i < 5; i = i + 1) {
20     println("* %d", notas[i]);
21   }
22 }
23
24 int calcularMedia(int[5] notas) {
25   int i, soma;
26
27   soma = 0;
28
29   for (i = 0; i < 5; i = i + 1) {
30     soma = soma + notas[i];
31   }
32
33   return soma / 5;
34 }
35
36 main() {
37   int[5] notas;
38   int media;
39
40   func lerNotas(notas);
41   func exibirNotas(notas);
42   media = func calcularMedia(notas);
43
44   println("A media do aluno foi: %d", media);
45
46   if (media < 7) {
47     println("O aluno nao antigiu a media 7");
48   } else {
49     println("O aluno antigiu a media 7");
50   }
51 }
```

Código 1. Exemplo de um programa que obtêm 5 notas de um aluno e mostra a sua média, dizendo se atingiu a média 7 ou não.

```
define void @lerNotas([5 x i32]* %notas.param) {
    %notas = alloca [5 x i32]*
    store [5 x i32]* %notas.param, [5 x i32]** %notas
    %i = alloca i32
    store i32 0, i32* %i
    br label %..for.head0
    ..for.head0:
    %1 = load i32, i32* %i
    %2 = icmp slt i32 %1, 5
    br i1 %2, label %..for.body0, label %..for.end0
    ..for.body0:
    %nota = alloca i32
    %3 = load i32, i32* %i
    %4 = add i32 %3, 1
    call i32 (i8*, ...) @printf(i8* @.str0, i32 %4)
    call i32 (i8*, ...) @scanf(i8* @.str1, i32* %nota)
    %7 = load [5 x i32]*, [5 x i32]** %notas
    %8 = load i32, i32* %i
    %9 = getelementptr inbounds [5 x i32], [5 x i32]* %7, i32 0, i32 %8
    %10 = load i32, i32* %nota
    store i32 %10, i32* %9
    %11 = load i32, i32* %i
    %12 = add i32 %11, 1
    store i32 %12, i32* %i
    br label %..for.head0
    ..for.end0:
    ret void
}
define void @exibirNotas([5 x i32]* %notas.param) {
    %notas = alloca [5 x i32]*
    store [5 x i32]* %notas.param, [5 x i32]** %notas
    %i = alloca i32
    call i32 (i8*, ...) @printf(i8* @.str2)
    store i32 0, i32* %i
    br label %..for.head1
    ..for.head1:
    %2 = load i32, i32* %i
    %3 = icmp slt i32 %2, 5
    br i1 %3, label %..for.body1, label %..for.end1
    ..for.body1:
    %4 = load [5 x i32]*, [5 x i32]** %notas
    %5 = load i32, i32* %i
    %6 = getelementptr inbounds [5 x i32], [5 x i32]* %4, i32 0, i32 %5
    %7 = load i32, i32* %6
    call i32 (i8*, ...) @printf(i8* @.str3, i32 %7)
    %9 = load i32, i32* %i
    %10 = add i32 %9, 1
    store i32 %10, i32* %i
    br label %..for.head1
    ..for.end1:
    ret void
}
define i32 @calcularMedia([5 x i32]* %notas.param) {
    %notas = alloca [5 x i32]*
    store [5 x i32]* %notas.param, [5 x i32]** %notas
```

```
56      %i = alloca i32
57      %soma = alloca i32
58      store i32 0, i32* %soma
59      store i32 0, i32* %i
60      br label %..for.head2
61      ..for.head2:
62      %1 = load i32, i32* %i
63      %2 = icmp slt i32 %1, 5
64      br i1 %2, label %..for.body2, label %..for.end2
65      ..for.body2:
66      %3 = load i32, i32* %soma
67      %4 = load [5 x i32]*, [5 x i32]** %notas
68      %5 = load i32, i32* %i
69      %6 = getelementptr inbounds [5 x i32], [5 x i32]* %4, i32 0, i32 %5
70      %7 = load i32, i32* %6
71      %8 = add i32 %3, %7
72      store i32 %8, i32* %soma
73      %9 = load i32, i32* %i
74      %10 = add i32 %9, 1
75      store i32 %10, i32* %i
76      br label %..for.head2
77      ..for.end2:
78      %11 = load i32, i32* %soma
79      %12 = sdiv i32 %11, 5
80      ret i32 %12
81  }
82  define i32 @main() {
83      %notas = alloca [5 x i32]*
84      %1 = alloca [5 x i32]
85      store [5 x i32]* %1, [5 x i32]** %notas
86      %media = alloca i32
87      %2 = load [5 x i32]*, [5 x i32]** %notas
88      call void @lerNotas([5 x i32]* %2)
89      %3 = load [5 x i32]*, [5 x i32]** %notas
90      call void @exibirNotas([5 x i32]* %3)
91      %4 = load [5 x i32]*, [5 x i32]** %notas
92      %5 = call i32 @calcularMedia([5 x i32]* %4)
93      store i32 %5, i32* %media
94      %6 = load i32, i32* %media
95      call i32 (i8*, ...) @printf(i8* @.str4, i32 %6)
96      %8 = load i32, i32* %media
97      %9 = icmp slt i32 %8, 7
98      br i1 %9, label %..if0, label %..else0
99      ..if0:
100     call i32 (i8*, ...) @printf(i8* @.str5)
101     br label %..if.end0
102     ..else0:
103     call i32 (i8*, ...) @printf(i8* @.str6)
104     br label %..if.end0
105     ..if.end0:
106     ret i32 0
107 }
108 @.str0 = private constant [28 x i8] c"Digite a %d nota do aluno:\0A\00"
109 @.str1 = private constant [3 x i8] c"%d\00"
110 @.str2 = private constant [24 x i8] c"As notas do aluno sao:\0A\00"
111 @.str3 = private constant [6 x i8] c"* %d\0A\00"
```

```
112 @.str4 = private constant [26 x i8] c"A media do aluno foi: %d\0A\00"
113 @.str5 = private constant [31 x i8] c"O aluno nao antigiu a media 7\0A
        \00"
114 @.str6 = private constant [27 x i8] c"O aluno antigiu a media 7\0A\00"
115 declare i32 @printf(i8*, ...)
116 declare i32 @scanf(i8*, ...)
```

Código 2. Programa do código 1 do apêndice B compilado para a representação intermediária do código *LLVM IR*.

```
1  ; cálculo da expressão condicional
2  %1 = icmp eq i32 1, 2
3  br i1 %1, label %..if0, label %..else0
4  ..if0:
5  ; ...
6  br label %..if.end0
7  ..else0:
8  ; ...
9  br label %..if.end0
10 ..if.end0:
```

Código 3. Código intermediário gerado pelo compilador para o comando if com $else$.

```
1  ..while.head0:
2  ; cálculo da expressão condicional
3  %1 = icmp eq i32 1, 1
4  br i1 %1, label %..while.body0, label %..while.end0
5  ..while.body0:
6  ; ...
7  ; volta para ..while.head0
8  br label %..while.head0
9  ..while.end0:
10 ; ...
```

Código 4. Código intermediário gerado pelo compilador para o comando $while$.

```
; inicialização das variáveis de controle
store i32 0, i32* %i
br label %..for.head0
..for.head0:
%1 = load i32, i32* %i
; cálculo da expressão condicional
%2 = icmp slt i32 %1, 10
br i1 %2, label %..for.body0, label %..for.end0
..for.body0:
; ...
; atribuições nas variáveis de contole
%3 = load i32, i32* %i
%4 = add i32 %3, 1
store i32 %4, i32* %i
; volta para ..for.head0
br label %..for.head0
..for.end0:
; ...
```

Código 5. Código intermediário gerado pelo compilador para o comando for.

Apêndice C - Quadro de conversões

Conversão	**Comando Usado**	**Explicação do comando**
Booleano para inteiro	$zext$	"*Zero extends*". Estende o valor com zeros nos bits mais significativos.
Inteiro menor para inteiro maior	$sext$	"*Signed extends*". Estende o valor com zeros nos bits mais significativos, respeitando o sinal
Inteiro maior para inteiro menor	$trunc$	"*Truncate*". Remove o os bits mais significativos
Booleano para ponto flutuante	$uitofp$	"*Unsigned Integer To Floating Point*". Realiza a conversão de um inteiro para ponto flutuante sem respeitar o sinal.
Inteiro para ponto flutuante	$sitofp$	"*Signed Integer To Floating Point*". Realiza a conversão de um inteiro para ponto flutuante respeitando o sinal.
Ponto flutuante para Inteiro	$fptosi$	"*Floating Point To Signed Integer*". Realiza a conversão de um ponto flutuante para inteiro respeitando o sinal.

Quadro 1. Descrição das conversões e comandos usados pelo compilador.

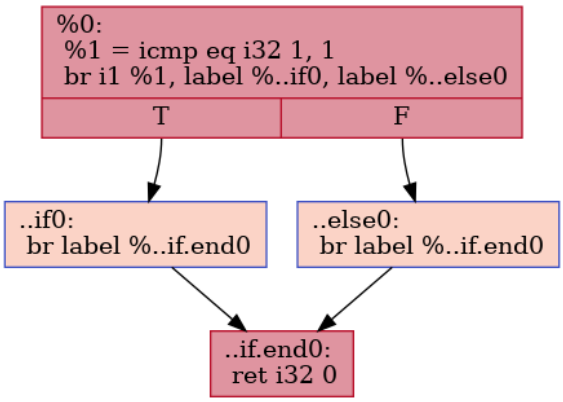


Figura 1. Grafo de fluxo de controle para o fluxo do comando if no código intermediário gerado pelo compilador.

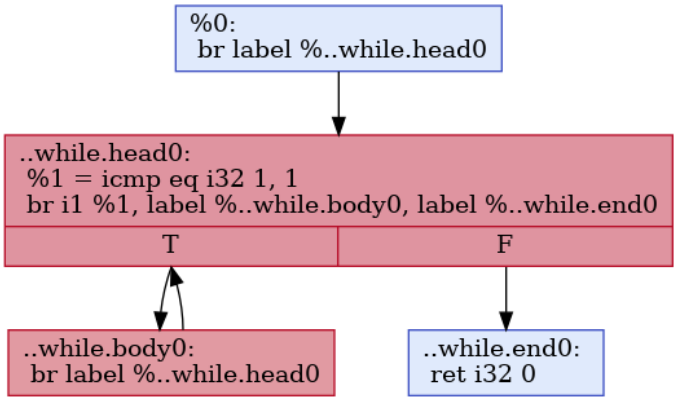


Figura 2. Grafo de fluxo de controle para o fluxo do comando $while$ no código intermediário gerado pelo compilador.

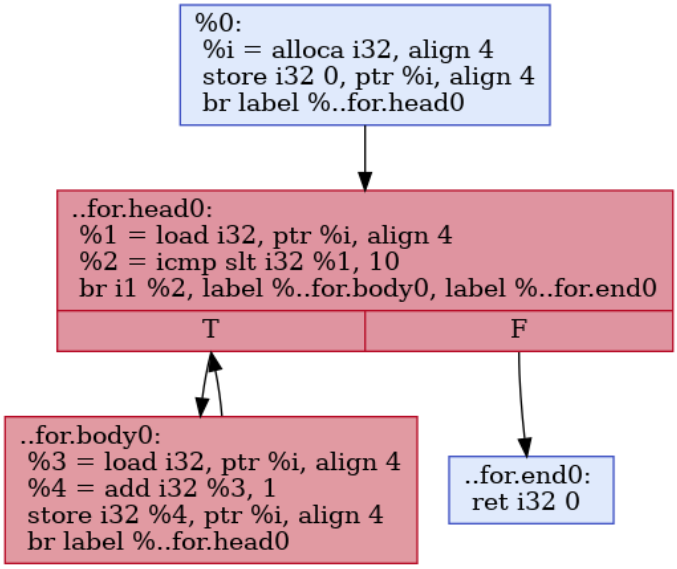


Figura 3. Grafo de fluxo de controle para o fluxo do comando for no código intermediário gerado pelo compilador.

Apêndice E - Exemplo de estruturas de dados utilizada pelo compilador.

```java
public interface Fragment {
    String getText();
}
```

Código 1. Definição da interface $Fragment$, utilizada para representar um fragmento do código IR.

```java
import java.util.ArrayList;
import java.util.stream.Collectors;

public class FragmentBlock extends ArrayList<Fragment> implements
    Fragment {

    private final char INDENT_VALUE = ' ';
    private final int INDENT_NUMBER = 4;

    @Override
    public String getText() {
        return this.stream()
                .map(Fragment::getText)
                .collect(Collectors.joining("\n"));
    }

    public String getIndentedText(int indentation) {
        String indentationString = String.valueOf(INDENT_VALUE)
                .repeat(INDENT_NUMBER * Math.max(0, indentation));

        return this.stream()
                .map(fragment -> indentationString + fragment.getText()
                    )
                .collect(Collectors.joining("\n"));
    }
}
```

Código 2. Implementação da classe $FragmentBlock$, utilizada para representar um conjunto de fragmentos.

```java
import ifsc.compiladores.projeto.LLVM.translator.definitions.Variable;

public abstract class ReturnableFragment implements Fragment {

    protected Variable returnVariable;

    public Variable getReturnVariable() {
        return returnVariable;
    }

}
```

Código 3. Implementação da classe $ReturnableFragment$, utilizada para representar um fragmento com uma variável de retorno.

```java
import ifsc.compiladores.projeto.LLVM.translator.definitions.Variable;

public class ReturnableFragmentBlock extends ReturnableFragment {

    protected final FragmentBlock fragmentBlock;

    public ReturnableFragmentBlock() {
        this.fragmentBlock = new FragmentBlock();
    }

    public ReturnableFragmentBlock(ReturnableFragment
       returnableFragment) {
        this.fragmentBlock = new FragmentBlock();
        this.fragmentBlock.add(returnableFragment);

        this.returnVariable = returnableFragment.getReturnVariable();
    }

    public FragmentBlock getFragmentBlock() {
        return fragmentBlock;
    }

    public void setReturnVariable(Variable returnVariable) {
        this.returnVariable = returnVariable;
    }

    @Override
    public String getText() {
        return this.fragmentBlock.getText();
    }
}
```

Código 4. Implementação da classe $ReturnableFragmentBlock$, utilizada para representar um bloco de fragmentos com uma variável de retorno.

```java
import ifsc.compiladores.projeto.LLVM.translator.definitions.Label;

public class LabeledFragmentBlock implements Fragment {

    private final FragmentBlock fragmentBlock;
    private final Label label;

    public LabeledFragmentBlock(Label label) {
        this.label = label;
        this.fragmentBlock = new FragmentBlock();
        this.fragmentBlock.add(label);
    }

    public FragmentBlock getFragmentBlock() {
        return fragmentBlock;
    }

    public Label getLabel() {
        return label;
    }

```

```
22      @Override
23      public String getText() {
24          return this.fragmentBlock.getText();
25      }
26
27  }
```

Código 5. Implementação da classe $LabeledFragmentBlock$, utilizada para representar um bloco de fragmentos com uma *label* associada.

```
1   import ifsc.compiladores.projeto.LLVM.translator.ReturnableFragment;
2
3   public class Load extends ReturnableFragment {
4
5       private final Variable loadVariable;
6
7       public Load(Variable returnVariable, Variable loadVariable) {
8           this.returnVariable = returnVariable;
9           this.loadVariable = loadVariable;
10      }
11
12      public Load(String returnVariableName, Variable loadVariable) {
13          this.returnVariable = new Variable(loadVariable.type().
                getNewDeferencePointerOfThis(), returnVariableName);
14          this.loadVariable = loadVariable;
15      }
16
17      @Override
18      public String getText() {
19          return String.format("%s = load %s, %s %s",
20                  this.returnVariable.getNameInIRForm(),
21                  this.returnVariable.type().getText(),
22                  this.loadVariable.type().getText(),
23                  this.loadVariable.getNameInIRForm());
24      }
25  }
```

Código 6. Implementação da classe do comando $load$.

```
1   public class LLVMIRGeneratorVisitor extends ParserGrammarBaseVisitor<
        Fragment> {
2       // ...
3       @Override
4       public FragmentBlock visitAtribuicao(ParserGrammar.
            AtribuicaoContext ctx) {
5           FragmentBlock attribuition = new FragmentBlock();
6
7           ReturnableFragmentBlock idAccess = (ReturnableFragmentBlock)
                visit(ctx.acesso_id());
8           attribuition.addAll(idAccess.getFragmentBlock());
9
10          Variable storeId = idAccess.getReturnVariable();
11
12          ReturnableFragmentBlock expressionReturn = (
                ReturnableFragmentBlock) visitExpressao(ctx.complemento().
                expressao());
```

```
13          attribuition.addAll(expressionReturn.getFragmentBlock());
14
15          ReturnableFragmentBlock expressionConversion =
                ConversionCreator.convert(
16                  this.singleUseVariablesManager,
17                  expressionReturn.getReturnVariable(),
18                  storeId.type().getNewDeferencePointerOfThis()
19          );
20          attribuition.addAll(expressionConversion.getFragmentBlock());
21
22          Store idStore = new Store(expressionConversion.
                getReturnVariable(), storeId);
23          attribuition.add(idStore);
24
25          return attribuition;
26      }
27      // ...
28  }
```

Código 7. Implementação da regra sintática $atribuicao$**.**

Apêndice F - Telas do programa

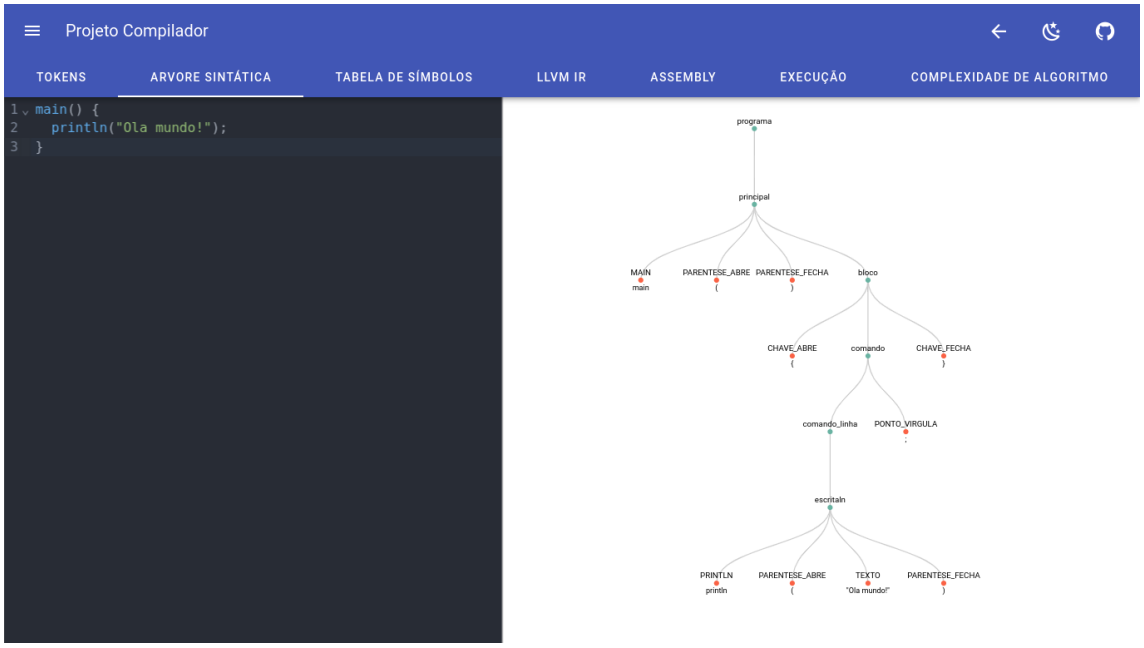


Figura 1. Tela de exibição da árvore sintática.

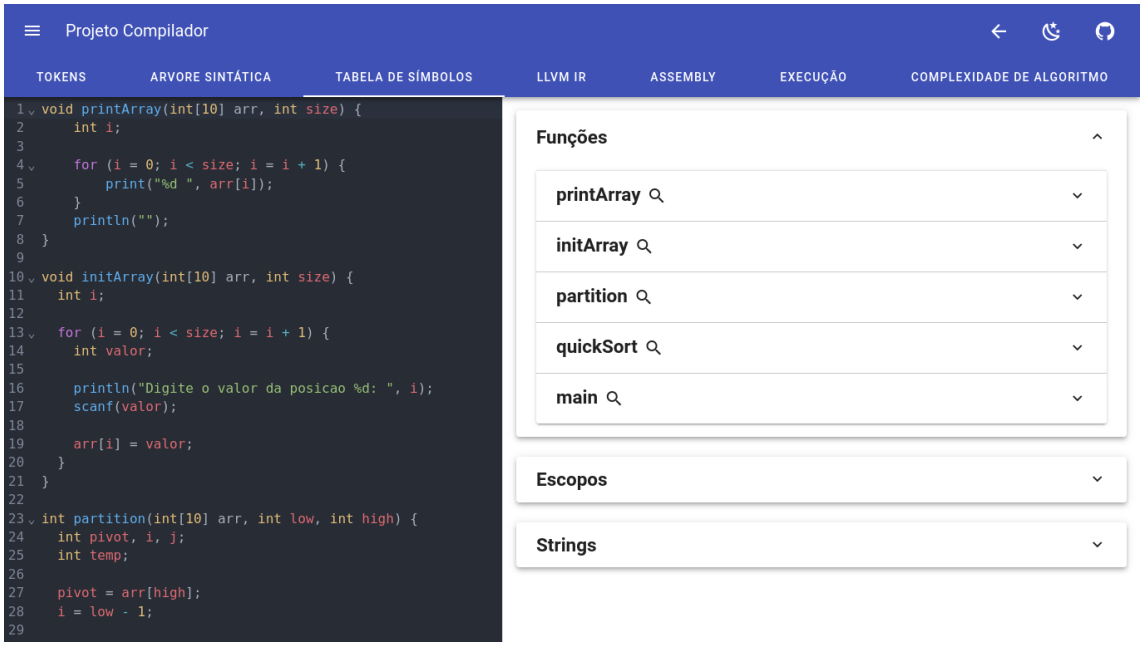


Figura 2. Tela de exibição da tabela de símbolos.

Figura 3. Tela de exibição do código *LLVM IR* com modo de comparação de otimizações ativado.

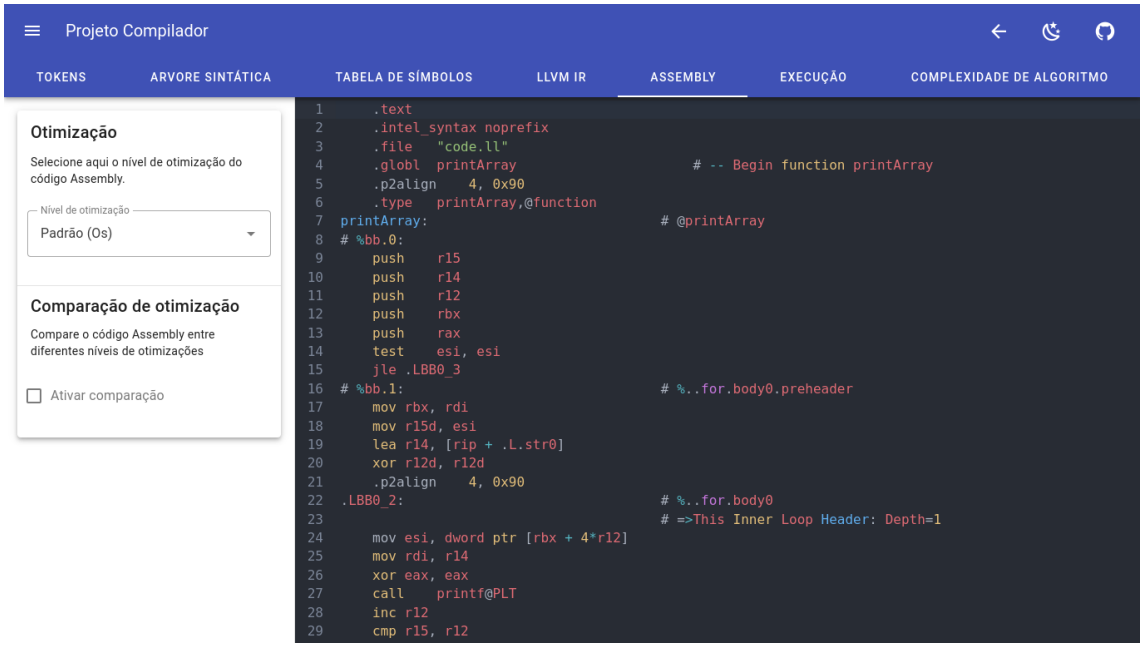


Figura 4. Tela de exibição do código *Assembly*.

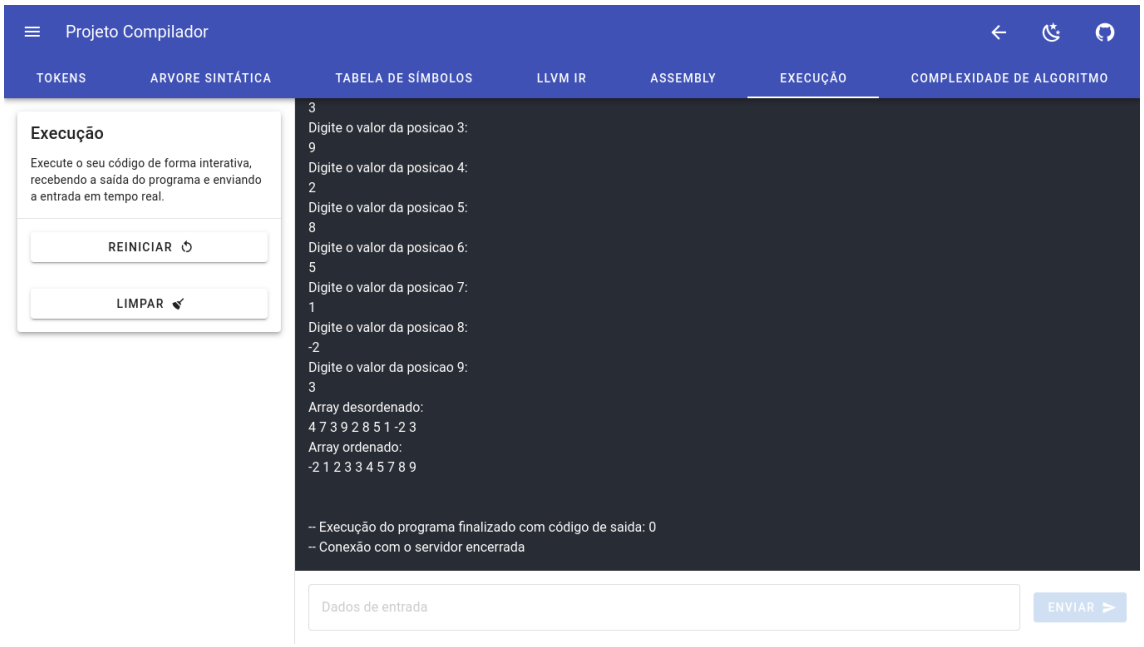


Figura 5. Tela de exibição da execução do código.

Apêndice G - Exemplo de estruturas de dados utilizada pelo módulo de análise de complexidade.

```java
import ifsc.compiladores.projeto.complexity.complexityAnalyserBuilder.
    definitions.position.TokenPosition;

public interface CostResult {
    TokenPosition getPosition();
    int getValue();
    String getStringRepresentation();
}
```

Código 1. Definição da interface $CostResult$, utilizada para representar um custo de uma determinada parte do código.

```java
import ifsc.compiladores.projeto.common.position.TokenPosition;

public class Cost implements CostResult {

    private final TokenPosition position;
    private final int value;
    private final boolean shouldShowInPlace;

    private Cost(TokenPosition position, int value, boolean
        shouldShowInPlace) {
        this.position = position;
        this.value = value;
        this.shouldShowInPlace = shouldShowInPlace;
    }

    public Cost(TokenPosition position, int value) {
        this(position, value, false);
    }

    public static Cost inPlace(TokenPosition position, int value) {
        return new Cost(position, value, true);
    }

    @Override
    public TokenPosition getPosition() {
        return this.position;
    }

    @Override
    public int getValue() {
        return this.value;
    }

    @Override
    public String getStringRepresentation() {
        return String.format("%d", this.value);
    }

    public boolean getShouldShowInPlace() {
        return this.shouldShowInPlace;
    }

```

```
}
```

Código 2. Definição da classe $Cost$, utilizada para representar um custo simples.

```
import ifsc.compiladores.projeto.common.position.TokenPosition;

import java.util.ArrayList;
import java.util.stream.Collectors;

public class BlockCost implements CostResult {

    private final CostResult blockCost;
    private final ArrayList<CostResult> costs;
    private final boolean isTopLevel;
    private final String topLevelId;

    public BlockCost(CostResult blockCost, ArrayList<CostResult> costs)
        {
        this.blockCost = blockCost;
        this.costs = costs;
        this.isTopLevel = false;
        this.topLevelId = null;
    }

    public BlockCost(ArrayList<CostResult> costs) {
        this.blockCost = new NullCost();
        this.costs = costs;
        this.isTopLevel = false;
        this.topLevelId = null;
    }

    private BlockCost(CostResult blockCost, ArrayList<CostResult> costs
        , boolean isTopLevel, String topLevelId) {
        this.blockCost = blockCost;
        this.costs = costs;
        this.isTopLevel = isTopLevel;
        this.topLevelId = topLevelId;
    }

    public BlockCost asTopLevel(String topLeveId) {
        return new BlockCost(this.blockCost, this.costs, true,
            topLeveId);
    }

    @Override
    public TokenPosition getPosition() {
        return this.blockCost.getPosition();
    }

    @Override
    public int getValue() {
        return this.blockCost.getValue();
    }

    @Override
    public String getStringRepresentation() {
```

```
50          ArrayList<String> expressionParts = new ArrayList<>();
51
52          String invariablePart = this.costs.stream()
53                  .filter(c -> c instanceof Cost)
54                  .map(CostResult::getValue)
55                  .filter(v -> v > 0)
56                  .map(String::valueOf)
57                  .collect(Collectors.joining(" + "));
58
59          if (!invariablePart.isEmpty())
60              expressionParts.add(invariablePart);
61
62          String variablePart = this.costs.stream()
63                  .filter(c -> c instanceof VariableCost)
64                  .map(CostResult::getStringRepresentation)
65                  .collect(Collectors.joining(" + "));
66
67          if (!variablePart.isEmpty())
68              expressionParts.add(variablePart);
69
70          String nestedBlocks = this.costs.stream()
71                  .filter(c -> c instanceof BlockCost)
72                  .map(CostResult::getStringRepresentation)
73                  .collect(Collectors.joining(" + "));
74
75          if (!nestedBlocks.isEmpty())
76              expressionParts.add(nestedBlocks);
77
78          String stringCost = String.join(" + ", expressionParts);
79
80          if (stringCost.isEmpty())
81              return "0";
82
83          return stringCost;
84      }
85
86      public CostResult getBlockCost() {
87          return blockCost;
88      }
89
90      public ArrayList<CostResult> getCosts() {
91          return costs;
92      }
93
94      public boolean isTopLevel() {
95          return isTopLevel;
96      }
97
98      public String getTopLevelId() {
99          return topLevelId;
100     }
101 }
```

Código 3. Definição da classe $BlockCost$, utilizada para representar um bloco de custos simple.

```java
import ifsc.compiladores.projeto.common.position.TokenPosition;

public class VariableCost implements CostResult {

    private final String variable;
    private final int costRange;
    private final CostResult additionalCost;

    private final BlockCost blockCost;

    public VariableCost(String variable, int costRange, CostResult
        additionalCost, BlockCost blockCost) {
        this.variable = variable;
        this.costRange = costRange;
        this.additionalCost = additionalCost;
        this.blockCost = blockCost;
    }

    @Override
    public TokenPosition getPosition() {
        return this.blockCost.getPosition();
    }

    @Override
    public int getValue() {
        return this.blockCost.getValue();
    }

    @Override
    public String getStringRepresentation() {
        String variableValue = (isInteger(this.variable)) ? String.
            valueOf(this.variable) : "n";
        return String.format("(%s + %s)*(%s)", variableValue, this.
            additionalCost.getStringRepresentation(), this.blockCost.
            getStringRepresentation());
    }

    public String getVariable() {
        return variable;
    }

    public int getCostRange() {
        return costRange;
    }

    public CostResult getAdditionalCost() {
        return additionalCost;
    }

    public BlockCost getBlockCost() {
        return blockCost;
    }

    private boolean isInteger(String str) {
        try {
            Integer.parseInt(str);
```

```
53              return true;
54          } catch (NumberFormatException e) {
55              return false;
56          }
57      }
58  }
```

Código 4. Definição da classe $VariableCost$, utilizada para representar um variável.

```
1   public class ComplexityAnalysisGeneratorVisitor extends
        ParserGrammarBaseVisitor<CostResult> {
2       // ...
3       @Override
4       public CostResult visitPara(ParserGrammar.ParaContext ctx) {
5           String forExpressionSizeVariableId = ctx.expressao().expr_ou().
                expr_e(0).expr_relacional(0).expr_aditiva(1).getText();
6           Variable forExpressionSizeVariable;
7
8           if (isInteger(forExpressionSizeVariableId))
9               forExpressionSizeVariable = new Variable(
                    forExpressionSizeVariableId, null, true);
10          else
11              forExpressionSizeVariable = this.variableManager.
                    getVariable(forExpressionSizeVariableId);
12
13          if (forExpressionSizeVariable == null || !
                forExpressionSizeVariable.isInput())
14              return visitBloco(ctx.bloco());
15
16          if (!ctx.expressao().expr_ou().expr_e(0).expr_relacional(0).
                op_relacional(0).getText().equals("<"))
17              return visitBloco(ctx.bloco());
18
19          ArrayList<CostResult> costs = new ArrayList<>();
20
21          int initialVariableValue = Integer.parseInt(ctx.
                atribuicaoInicio.atribuicao(0).complemento().getText());
22
23          int costRange = initialVariableValue * -1;
24          BlockCost forBlockCost = visitBloco(ctx.bloco());
25
26          Cost forExpressionCost = Cost.inPlace(TokenPosition.fromContext
                (ctx.expressao()), BasicCommandCost.EXPRESSION.getCost());
27
28          VariableCost forVariableCost = new VariableCost(
                forExpressionSizeVariableId, costRange, forExpressionCost,
                forBlockCost);
29          costs.add(forVariableCost);
30
31          return new BlockCost(costs);
32      }
33      // ...
34  }
```

Código 5. Implementação do método do *vistor* que realiza o cálculo da complexidade do bloco for.