Uma Ferramenta Educacional de Apoio ao Ensino de Compiladores

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. The tool consists of a 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. The overall evaluation of the system confirmed the high level of acceptance, reaching an average of* 4.81 *out of* 5*, highlighting the educational potential of the tool.*

Resumo. *Este artigo apresenta uma ferramenta para auxiliar o ensino de compiladores. A ferramenta consiste em um compilador 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. A avaliação geral do sistema confirmou o alto nível de aceitação, alcançando uma média de* $4,81$ *em* 5*, evidenciando o potencial educacional da ferramenta.*

1. Introdução

Os sistemas de *software* responsáveis por traduzir código-fonte para um formato que possa ser compreendido pelo hardware do computador 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* (Cooper e Torczon, 2014).

Disciplinas que ensinam os conceitos de compiladores são comuns e fundamentais em cursos superiores na área de computação. Essas disciplinas abordam conceitos teóricos, abstratos e complexos, fundamentais 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 apresenta uma ferramenta para auxiliar no processo de ensino-aprendizagem de compiladores. Para o seu desenvolvimento foi utilizado o *ANTLR 4* [1] para a geração dos analisadores léxico e sintático e o *LLVM* [2] para as etapas de otimização e geração do código alvo (*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, executá-lo, e visualizar os diferentes artefatos artefatos produzidos pelo compilador, tais como a lista de *tokens*, a árvore sintática, o código intermediário do *LLVM* e o código *Assembly*.

Este artigo está dividido em 5 seções. Após a introdução, a seção 2 apresenta alguns trabalhos relevantes. A seção 3 descreve a implementação do trabalho. Na seção 4 são apresentadas as avaliações dos usuários. Por fim, a seção 5 registra as conclusões sobre o trabalho.

2. Trabalhos Relacionados

Para a obtenção dos trabalhos similares, uma revisão bibliográfica foi realizada nas plataformas *SBC-OpenLib* e *IEEE Xplorer*. Os termos utilizados nas pesquisas foram "compilador educação", "*compiler teaching*" e "*compiler learning*". Após a pesquisa, os dez primeiros artigos de cada plataforma foram selecionados para leitura dos resumos e os mais relevantes foram selecionados para uma leitura e análise mais detalhada. Dos trabalhos lidos, foram escolhidos três que ficaram mais próximos dos objetivos propostos por este trabalho para serem apresentados nessa seção.

LISA, 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 possibilita 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. 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.

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 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. Para o código intermediário, é apresentado o resultado da compilação na linguagem *MacroAssembler* do *Verto*, uma versão próxima à

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

utilizada na máquina virtual *Cesar*, desenvolvida na Universidade Federal do Rio Grande do Sul (UFRGS). Por fim, o código alvo é gerado e pode ser executado utilizando a ferramenta *Cesar* (Weber, 2001).

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 análise léxica, 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. Na análise sintática, é apresentado um passo a passo da derivação da árvore usando os *tokens* obtidos na análise léxica, com o qual o usuário pode interagir.

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

Trabalho	**Plataforma**	**Etapas**	**Visualização dos resultados**	**Algoritmos arbitrários**	**Execução do programa**
Mernik e Zumer (2003)	*Desktop*	Léxica Sintática Semântica	Passo a passo	Sim	Não
Scheider et al. (2005)	*Desktop*	Todas	Forma textual	Sim	Sim
Graciano Junior et al. (2022)	*Web*	Léxica Sintática	Passo a passo	Sim	Não
Sistema proposto	*Web*	Todas	Forma textual	Sim	Sim

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

Dentre os trabalhos analisados, uma das características observadas foi a plataforma utilizada. Ferramentas mais antigas tendem a usar a plataforma *desktop*, o que pode dificultar o acesso dos usuários. Apenas o trabalho de Graciano Junior et al. (2022) foi desenvolvido para a plataforma *Web*. 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. Percebeu-se que todos os trabalhos apresentados permitem a digitação de algoritmos arbitrários, o que é uma caraterística importante a ser considerada. Alguns deles possibilitam a visualização dos resultados de forma gráfica e outros de forma textual, sendo que cada abordagem é mais adequada para determinadas etapas do processo.

A ferramenta proposta busca oferecer a visualização 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. 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).

3. Implementação e apresentação do sistema

Essa seção está dividida em duas partes. A seção 3.1 descreve os detalhes da arquitetura geral da implementação programa e a seção 3.2 descreve de forma específica o funcionamento e implementações do módulo de compilação.

3.1. Arquitetura geral

Com o objetivo de simplificar o desenvolvimento e visando uma melhor organização do projeto, a arquitetura implementada segue o modelo cliente-servidor. O cliente, ou *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*.

O servidor, ou *back-end*, é responsável por todo o processamento do programa, incluindo a compilação, 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*. O repositório pode ser consultado no endereço: `https://github.com/erikborella/projeto-compiladores-ifsc`. A figura 1 mostra a arquitetura do projeto, com a comunicação entre os seus componentes de forma conceitual.

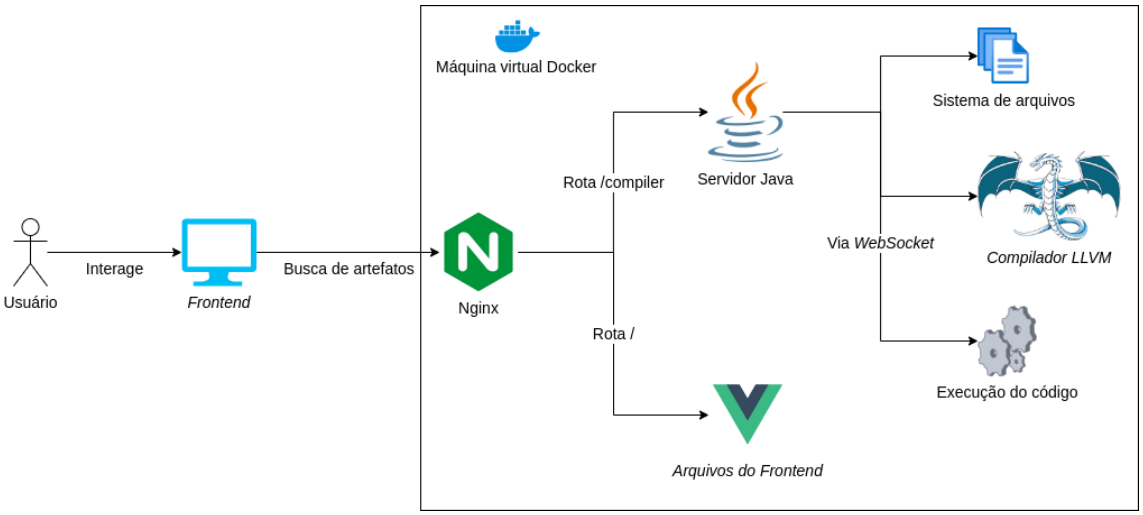

Figura 1. Arquitetura do projeto de forma conceitual.

3.2. Módulo 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 *LexerGrammar.g4*, disponível na pasta: `https://github.com/erikborella/projeto-compiladores-ifsc/blob/main/gramatica/`, enquanto as regras sintáticas estão definidas no código *ParserGrammar.g4*, disponível na mesma pasta.

A linguagem suporta quatro tipos de dados básicos ($boolean$, $char$, int e $float$), além disso, é permitido a declaração de *arrays* de tamanho fixo e multidimensionais. Para que o nome de uma variável seja válido, ele deve seguir as regras de identificadores utilizadas na linguagem C. As variáveis devem ser declaradas no início de cada bloco de código, antes de qualquer comando. Os blocos são delimitados pelos caracteres { e }.

Após as declarações de variáveis de um bloco, é permitido especificar os comandos. A linguagem possui dois tipos de comandos: comandos de linha, que são expressos em apenas uma linha e devem ser terminados com ponto e virgula (;) e comando 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, seguindo a sintaxe da linguagem C, tanto para a atribuição em si quanto para os operadores aritméticos das expressões. 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 linguagem também oferece comandos para leitura de valores do usuário e exibição de mensagens na tela. O comando de leitura utiliza a palavra-chave $scanf$, e entre parênteses deve-se especificar a variável onde o valor fornecido pelo usuário será armazenado. Já o comando de escrita de mensagens possui duas variações, definidas pelas palavras reservadas $print$ e $println$, que além de mostrar o texto, adiciona uma nova linha ao final. Entre parênteses, deve-se especificar um texto entre aspas duplas contendo o modelo da mensagem a ser exibida, conforme padrão utilizado pela *glibc*[3].

É possível chamar funções declaradas utilizando o comando $funcao$. Diferentemente da linguagem C e da maioria das outras linguagens, para chamar uma função, utiliza-se a palavra reservada $func$ seguida pelo nome da função. Caso a função receba argumentos, eles devem ser especificados após o nome da função, entre parênteses e separados por vírgulas. A declaração de funções, deve ser feita antes do bloco $main$. A sintaxe para a definição de funções é semelhante à da linguagem C, especificando primeiro o tipo de retorno da função, seguido por um nome e, em seguida, pelos argumentos que ela recebe. O retorno de funções é especificado utilizando a palavra-chave $return$, opcionalmente seguida de um valor.

A linguagem possui três comandos de bloco: if, $while$ e for. Esses comandos são semelhantes às estruturas da linguagem C, utilizando-se da mesma sintaxe. Os mesmos operadores lógicos são suportados, como: $==$, $!=$, $>$, $>=$, $<$, $<=$ e !. Além disso, através dos operadores lógicos $\&\&$ (operador e) e $||$ (operador ou), é possível construir expressões condicionais utilizando curto-circuito. No entanto, a linguagem implementada não suporta as estruturas $do - while$ e $switch$, presentes na linguagem C.

O código 1 apresenta um exemplo de programa que recebe um valor inteiro do

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

usuário e calcula o seu fatorial de forma iterativa.

```
int calcularFatorial(int valor) {
  int i, resultado;
  resultado = 1;
  for (i = 2; i <= valor; i = i + 1) {
    resultado = resultado * i;
  }
  return resultado;
}

main() {
  int valor, fatorial;
  println("Digite o valor que deseja calcular o fatorial: ");
  scanf(valor);
  if (valor < 1) {
    print("O valor deve ser maior ou igual a 1");
  }
  else {
    fatorial = func calcularFatorial(valor);
    println("O fatorial de %d é %d", valor, fatorial);
  }
}
```

Código 1. Exemplo de um programa para realizar o cálculo do fatorial de um número.

O compilador, ao ser executado, gera uma representação intermediária na forma do *LLVM* (*LLVM IR*). O *LLVM IR* utiliza a forma *Static Single-Assignment* (*SSA*), pois seu uso 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 gerar o executável.

3.2.2. 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 a geração do código intermediário, foram criadas classes e interfaces que auxiliam na construção do IR. A principal delas é a interface $Fragment$, que representa um fragmento do IR. Esta interface possui apenas o método $getText()$ que retorna a *String* correspondente ao fragmento.

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.

3.2.3. Implementação da interface

A *API*, construída com o *framework Spring Boot*, disponibiliza *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. Além disso, a *API* oferece a funcionalidade de executar o código compilado de forma remota em tempo real via protocolo *WebSocket*.

O *frontend* 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. A figura 2 apresenta a tela inicial. O sistema e todas as suas funcionalidades podem ser acessados em `http://pesquisa06.lages.ifsc.edu.br/`.

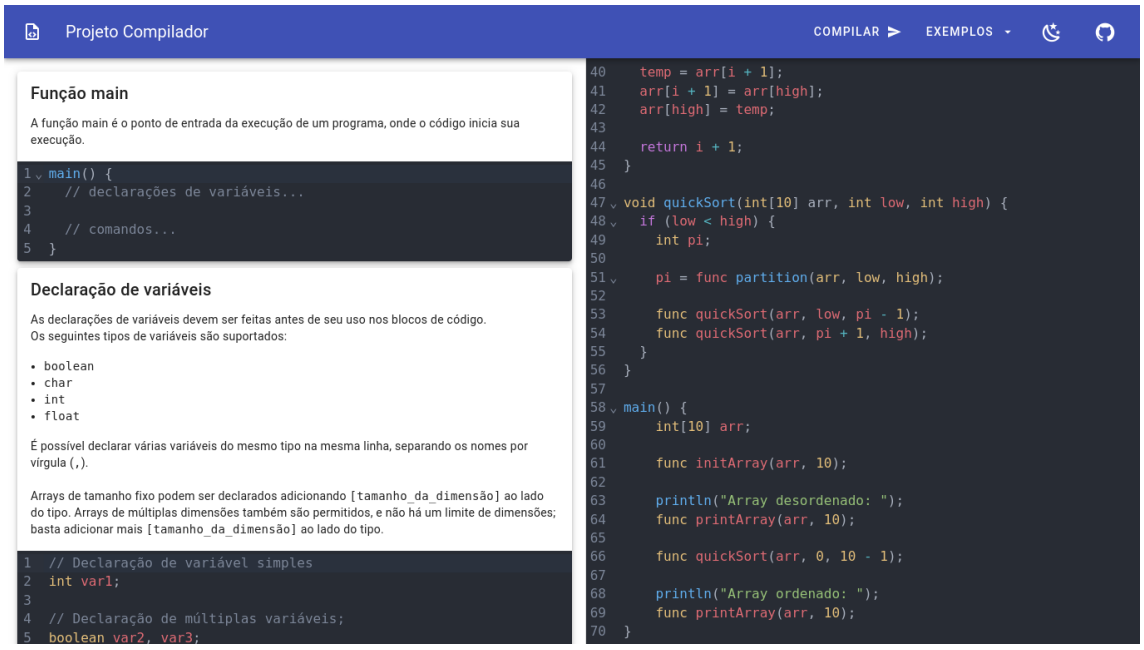

Figura 2. Tela inicial do programa.

Após a compilação, uma tela contendo a lista de *tokens* é exibida, seguida da tela que apresenta a árvore sintática correspondente. Ambas permitem que, ao posicionar o *mouse* sobre um token ou um nó da árvore, a região correspondente seja destacada no código-fonte. A tela seguinte exibe a tabela de símbolos, mostrando as funções, escopos e *strings* declarados no programa.

Na sequência é apresentada a tela com o código *LLVM IR*, onde é possível visualizar o código intermediário em diferentes níveis de otimização, selecionados através do menu lateral. 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 3).

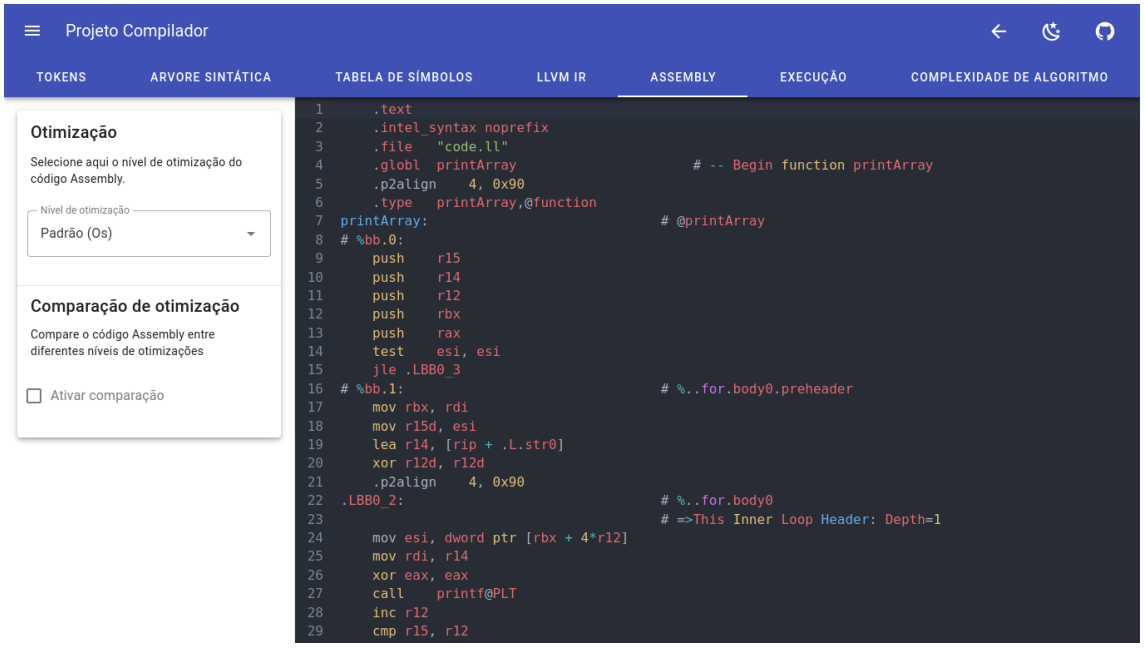

Figura 3. Tela de exibição do código *Assembly*.

A última tela do módulo do compilador permite a execução do código. Nela, 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. Essa funcionalidade é implementada via conexão com a API de execução através de *WebSocket*.

4. Resultados da avaliação dos usuários

Para avaliar o sistema desenvolvido, foi elaborado um questionário *online* anônimo, acompanhado de um vídeo introdutório, que apresentava o sistema e demonstrava o seu funcionamento, um *link* para acesso ao sistema e 15 perguntas. Das 15 perguntas, 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 de Ciência da Computação do IFSC - 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 tratava do perfil dos avaliadores em relação às disciplinas de compiladores. Os resultados indicaram que $18,5\%$ das respostas foram de professores, $74,1\%$ de alunos que já cursaram ou estão cursando a disciplina de compiladores e $7,4\%$ de outros participantes que não se encaixam nas duas categorias anteriores.

As três perguntas seguintes 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, 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 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.

Por fim, duas perguntas gerais abordaram as características do sistema e sua aplicabilidade fora da sala de aula. 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. Além disto, a avaliação geral do sistema confirmou o alto nível de aceitação, alcançando uma média de $4,81$ em 5.

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;

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:

- "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, abordando a definição da linguagem suportada pelo compilador, o processo de compilação e tradução da linguagem para o *LLVM IR*, o funcionamento do *LLVM* 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 realizar as etapas de análise 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.

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. 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.

A ferramenta não tem como objetivo ser o único recurso para aprendizagem dos conteúdos. A visualização dos resultados das etapas de compilação, assim como a consulta ao código-fonte da ferramenta no github, podem auxiliar na compreensão dos conteúdos ministrados em disciplinas de linguagens formais e compiladores, desde que façam parte de um conjunto maior de atividades planejadas pelo docente da disciplina. A ferramenta também pode contribuir com disciplinas nas áreas de arquitetura de computadores e sistemas operacionais, por permitir que os alunos visualizem como os códigos em alto nível criados por eles são representados na linguagem Assembly.

Como trabalhos futuros, o compilador pode ser aprimorado para suportar uma linguagem com mais recursos, permitir a alocação de objetos na *heap* do programa, 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. Também está em desenvolvimento um módulo para o cálculo da complexidade de algoritmos, que será capaz de exibir o cálculo de $T(n)$ e sua representação na notação assintótica *Big* O. Alguns testes já foram realizados previamente para esse módulo, e novas funcionalidades e melhorias estão sendo desenvolvidas para sua futura disponibilização.

Referências

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

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

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*.

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).

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