Ferramenta Interativa para o ensino de compiladores

Wagner Graciano Junior, Iara Tavares da S. Grossert, Wilson Castello Branco Neto, Alex Junior Avila
{wagner_graciano0101,iara_grossertjohn,alex_avila11}@hotmail.com,wilson.castello@ifsc.edu.br
Instituto Federal de Santa Catarina, Lages, SC

RESUMO

Este artigo apresenta uma ferramenta didática para auxiliar no processo de ensino-aprendizagem da disciplina de Compiladores, por meio de explicações teóricas e da visualização do processo prático das etapas de análise léxica e sintática. A ferramenta consiste em uma aplicação web responsiva que, além do estudo dos conceitos teóricos, permite que o usuário digite e acompanhe o processo e o resultado da análise do seu próprio código-fonte. Ela apresenta o passo a passo detalhado com explicações do processamento de acordo com a interação do usuário, facilitando a compreensão dos conceitos envolvidos nas duas primeiras etapas do processo de compilação. Os códigos analisados devem ser escritos em pseudo-linguagem para possibilitar que qualquer pessoa com conhecimentos de programação possa usá-la, independente de uma linguagem específica.

CCS CONCEPTS

• **Social and professional topics** → Computing education.

PALAVRAS-CHAVE

Ensino de compiladores, Análise Léxica, Análise Sintática, Ferramenta interativa

1 INTRODUÇÃO

O compilador, de acordo com José Neto [6], pode ser definido como um dos módulos do software básico de um computador, cuja função é a de efetuar a tradução de textos, redigidos em uma determinada linguagem de programação, para alguma outra forma que viabilize sua execução. Em geral, esta forma é uma linguagem de máquina, embora esta seja apenas uma das inúmeras alternativas possíveis. Todos os programas escritos em linguagem de alto nível precisam passar necessariamente pelo processo de compilação. Nele são definidos todos os aspectos e limites da linguagem. Existem centenas de linguagens, assim como diversas arquiteturas que precisam se comunicar, tarefa essa executada pelo compilador. Por estes motivos, segundo Pereira [8], compiladores são a roda que movimenta a ciência da computação.

Aprender os processos inerentes à construção de compiladores é uma tarefa benéfica. Segundo Aho et al. [1], ela proporciona conhecimentos fundamentais da ciência da computação, reforça as bases interdisciplinares, estimula a criatividade e melhora a abstração. Como essas técnicas são muito utilizadas na construção de diferentes sistemas computacionais, torna-se mais compreensível o desenvolvimento de aplicações com linguagens embutidas, manipulação de arquivos de configuração, construção de tradutores automáticos, geração de código a partir de modelos de alto nível, entre outros.

Aho et al. [1] também apontam que devido a abrangência da disciplina, entendê-la demanda conhecimento prévio de diversas áreas da computação, tais como: arquitetura e organização de computadores, linguagens formais, sistemas operacionais, estrutura de dados e programação de computadores. Além desta multidisciplinaridade, as fases do processo de compilação são extensas e específicas, tornando, assim, seu entendimento um processo bastante rigoroso.

Um dos principais entraves para compreensão dos conceitos se dá pela dificuldade para visualizar o seu funcionamento, de um ponto de vista prático. Segundo White et al. [13], um exemplo concreto de um compilador pode ajudar a examinar e entender melhor modelos formais, engenharia, estrutura de dados e algoritmos que ajudam a resolver problemas de processamento de linguagem.

Existem ferramentas que auxiliam a construção de um compilador. Elas atendem a demandas específicas, mas nenhuma tem como foco a intuitividade de experiência do usuário ou a demonstração passo a passo das etapas de compilação de maneira didática. De acordo com Mernik e Zumer [7], muitas ferramentas foram e ainda são utilizadas como geradores de scanner, parser e compilador. Entretanto, essas ferramentas possuem pouco ou nenhum valor didático.

Considerando os problemas levantados, este artigo tem como principal objetivo apresentar uma aplicação interativa para auxiliar no processo de ensino-aprendizagem das etapas de análise léxica e sintática da disciplina de compiladores. A ferramenta aqui apresentada é o resultado da primeira etapa de um projeto mais amplo que visa a construção de uma ferramenta que contempla todas as etapas do processo de compilação.

Este artigo está dividido em sete seções. Após esta introdução, a Seção 2 apresenta os conceitos básicos necessários para a compreensão do trabalho e a Seção 3 descreve alguns trabalhos similares. A Seção 4 expõe a metodologia utilizada em seu desenvolvimento. Na sequência, a Seção 5 apresenta o sistema desenvolvido e a Seção 6 discute os resultados obtidos, em comparação aos trabalhos similares. Por fim, a Seção 7 apresenta as considerações finais e as sugestões de trabalhos futuros.

2 REFERENCIAL TEÓRICO

Segundo Cooper e Torczon [4], o compilador é um programa de computador que traduz outros programas para prepará-los para execução. De acordo com Aho et al. [1], existem duas fases no

EduComp'22, Abril 24-29, 2022, Feira de Santana, Bahia, Brasil (On-line)

processo de compilação, a análise e a síntese. As primeiras etapas da compilação consistem no núcleo de análise, que é dividido em léxica, sintática e semântica e o gerador de código intermediário. A síntese consiste nas últimas etapas que são a otimização de código e a geração de código alvo.

As principais atividades realizadas em cada etapa do compilador, de acordo com Aho et al. [1], são:

- Análise léxica: lê os caracteres de um programa fonte e os agrupa num fluxo de *tokens*, no qual cada *token* representa uma sequência de caracteres logicamente coesa, como, por exemplo, um identificador ou uma palavra-chave. A sequência dos caracteres que formam um *token* é chamada lexema.
- Análise sintática: envolve o agrupamento dos *tokens* do programa fonte em frases gramaticais, representadas por uma árvore gramatical, para verificar se a sequência destes *tokens* está correta ou não.
- Análise semântica: verifica os erros semânticos no programa fonte e captura as informações de tipo para a fase de geração de código. Nesta etapa é importante ressaltar o componente da verificação de tipos que verifica se cada operador recebe os operandos que são permitidos pela especificação da linguagem fonte.
- Geração de código intermediário: pode-se considerar essa representação intermediária como um programa para uma máquina abstrata, que deve possuir duas propriedades importantes: ser fácil de produzir e fácil de traduzir no programa alvo. Uma forma de representar código intermediário consiste no *código de três endereços*.
- Otimização de código: realiza melhorias no código intermediário para gerar um código de máquina mais eficiente.
- Geração de código: as instruções intermediárias são traduzidas numa sequência de instruções de máquina que realizam a mesma tarefa. Esta etapa gera, normalmente, um código de máquina realocável ou código de montagem.

Como citado na introdução, a ferramenta apresentada neste trabalho é o resultado da primeira etapa do projeto e aborda as duas primeiras etapas do processo de compilação, detalhadas nas próximas seções. A implementação das etapas posteriores ainda está em desenvolvimento.

2.1 Análise Léxica

Varshney et al. [12] definem que a análise léxica processa a entrada de uma sequência de caracteres para produzir uma sequência de símbolos denominados *tokens*. Ela realiza a leitura dos símbolos caractere por caractere, agrupando-os em uma sequência chamada de *lexema*, que são classificados em *tokens*, de acordo com padrões definidos.

José Neto [6] indica que Expressões Regulares (ER) são uma maneira para representar linguagens e correspondem a formas gerais das sentenças das linguagens que representam, as quais são expressas através do uso exclusivo dos terminais da linguagem, sem o recurso de não-terminais. Os padrões que associam os lexemas aos *tokens* podem ser expressos por meio de ER e o reconhecimento dos *tokens* é implementado por meio de autômatos finitos (AF). Cooper e Torczon [4] afirmam que para qualquer ER é possível criar um AF que reconheça os símbolos descritos por ela.

Compreender como um AF reconhece cada lexema e classifica-o como um *token* é o principal elemento para a compreensão da etapa de Análise Léxica. A apresentação de figuras com AF capazes de reconhecer diferentes tipos de *tokens* e a análise manual de pequenos trechos de códigos são estratégias que os professores utilizam para possibilitar que os alunos compreendam estes conceitos. Entretanto, acredita-se que esta compreensão será mais fácil e abrangente, se o aluno puder criar seus próprios códigos e visualizar graficamente as transformações que ocorrem no AF durante a análise destes códigos. Assim, ele pode fazer alterações nos códigos, inclusive incorporando erros a ele, para verificar como estes erros são identificados pelo analisador, tendo como base o AF.

2.2 Análise Sintática

Segundo José Neto [6], a principal atividade realizada pelo Analisador Sintático é verificar a sequência de *tokens* provenientes do analisador léxico. A partir desta sequência, o analisador sintático efetua uma verificação acerca da ordem de apresentação dos *tokens*, identificando, em cada situação, o tipo da construção sintática por eles formada de acordo com a gramática na qual se baseia o reconhecedor.

Gramáticas Livres de Contexto (GLC), que são um conjunto de regras que descrevem como formar sentenças da linguagem, são usadas para realizar esta tarefa, além de facilitar a tradução do código nas etapas posteriores e a localização de erros.

Uma árvore de derivação é uma representação gráfica que demonstra a ordem na qual as produções de uma GLC são aplicadas para substituir os não-terminais, com o objetivo de determinar se um fluxo de palavras, ou *tokens* no caso de um compilador, se encaixa na sintaxe da linguagem

Duas estratégias de análise sintática podem ser utilizadas para a geração da árvore de derivação [4]:

- Análise descendente (*top-down*): começa com a raiz da árvore sintática (símbolo inicial da gramática) e a estende sistematicamente para baixo, até que suas folhas correspondam aos *tokens* retornados pelo analisador léxico.
- Análise ascendente (*bottom-up*): realiza o sentido inverso, começa nas folhas (*tokens*) e avança até a raiz da árvore.

De forma semelhante à Análise Léxica, o principal aspecto a ser trabalhado durante a Análise Sintática é dinâmico e fica difícil de ser representado por imagens estáticas. Nesta etapa, é fundamental que o estudante compreenda o processo de construção da árvore sintática, seja ela criada por um algoritmo descendente ou ascendente. Novamente, as principais estratégias usadas por professores e autores de livros é apresentar as gramáticas, pequenos trechos de código e figuras com as árvores geradas. Em muitos casos é apresentada apenas a árvore final, o que limita a compreensão pois não contempla a sequência de passos realizadas. Em outros exemplos até são apresentadas diversas figuras, representando o estado da árvore a cada nova iteração, mas em geral, estes exemplos são para trechos de código muito pequenos, em função do tempo e espaço necessário para desenhar todas elas, e também são para códigos prontos, o que impede que o aluno reflita sobre questões como, por exemplo, "O que aconteceria se eu apagasse este ponto e vírgula?".

3 TRABALHOS RELACIONADOS

Esta Seção descreve alguns sistemas utilizados para auxiliar no processo de compreensão e construção das fases de análise léxica e sintática do compilador. São ferramentas com finalidade semelhante à proposta no presente artigo, porém com abordagens diferentes. Essas aplicações foram encontradas e analisadas com base em artigos publicados em eventos e periódicos.

O LexSint é uma ferramenta educacional para o estudo de analisadores léxico e sintático que permite ao aluno gerar a GLC da linguagem a partir de exemplos de estruturas de código descritos em um arquivo de texto com padrões em português. O artigo justifica a ferramenta apontando dificuldades por parte dos alunos para compreender as fases de análise do compilador. O processo de análise léxica e sintática é mostrado passo a passo pela ferramenta, entretanto os analisadores não são gerados por ela.

O programa tem como entrada um arquivo de texto com os *tokens* e os padrões das estruturas sintáticas da linguagem para permitir a construção da GLC, que é mostrada pronta em um arquivo de texto separado. A partir da geração automática da GLC, a ferramenta permite inserir códigos para mostrar, através de linhas enumeradas, o processo de derivação das regras para realizar a análise sintática. Além disto, a ferramenta apresenta os conjuntos FIRST e FOLLOW, porém não dispõe de interface gráfica nem de explicação teórica dos processos [2].

Verto é uma aplicação que auxilia na compreensão das fases de um compilador, sobretudo das fases finais de geração de código intermediário e geração de código-objeto. Ele faz parte de um ambiente digital de aprendizagem para compiladores escrito na linguagem de programação Java e elaborado na forma de um software livre com licença GPL. Usa como linguagem fonte o português estruturado que é inserido na própria ferramenta em uma das abas fornecida por ela, seguindo uma sintaxe, próxima a da linguagem C.

A linguagem objeto utilizada é a linguagem César, implementada na Universidade Federal do Rio Grande do Sul (URFGS) para as disciplinas de Arquitetura de Computadores I e II do curso de Ciência da Computação. O processo de compilação ocorre em 2 etapas para fins pedagógicos:

- geração de um código-intermediário em um formato macro-assembler com formas mais simplificadas das instruções da máquina César;
- geração do arquivo destino final contendo as instruções no formato da máquina hipotética César.

Conforme Scheider et al. [11], a ferramenta apresenta as saídas das fases de análise e a tabela de símbolos através de abas em uma interface gráfica, porém possui detalhes e especificidades focados nas fases de geração de código intermediário e geração de código-objeto. Essas fases finais apresentam as instruções, linha a linha, seguidas por uma explicação em português detalhando seu significado.

A apresentação, a seguir, sobre as ferramentas Flex[1], JFlex[2] e GALS[3] foi embasada em uma comparação entre as mesmas realizada por Barbosa et al. [3].

[1]http://gnuwin32.sourceforge.net/packages/flex.html
[2]https://jflex.de/
[3]http://gals.sourceforge.net/

Flex é uma ferramenta que tem como objetivo gerar analisadores léxicos na linguagem C que não possui finalidade didática, portanto, não há explicação sobre nenhum processo. Contudo, é utilizada amplamente para auxiliar os alunos a entenderem o processo de análise léxica. A especificação de entrada é realizada por expressões regulares e comandos em linguagem de programação. O analisador é gerado após o reconhecimento de uma expressão regular, sem detalhar o processo em um passo a passo, seguido da execução dos comandos associados à mesma. Esta ferramenta não possui interface, ou seja, o usuário deve escrever e visualizar o código resultante em algum editor de texto.

JFlex é um gerador de analisador léxico para Java, implementado também nessa linguagem. O programa JFlex cria uma classe Java que faz a análise léxica de códigos-fonte armazenados em um arquivo de texto. Ele faz o reconhecimento dos códigos por meio de tabelas de transição de autômatos determinísticos geradas a partir de ER informadas pelo usuário. O software foi feito em inglês e produz uma interface gráfica que mostra o resultado da análise, porém, não exibe qualquer explicação sobre o processo. A aprendizagem pode se tornar um pouco mais complicada para quem não tem conhecimento prévio da linguagem Java.

GALS é um gerador de compiladores que é utilizado para a geração automática de analisadores léxicos e sintáticos. É o resultado de um trabalho de conclusão do curso de Ciências da Computação da Universidade Federal de Santa Catarina (UFSC), em 2003. Ele gera os analisadores a partir de especificações léxicas baseadas em expressões regulares e sintáticas baseadas em GLC.

Essa ferramenta oferece três opções de linguagens para a geração dos analisadores, que são: Java, C++ e Delphi. Esta ferramenta apresenta uma interface didática, que indica o erro no código, sendo que para utilização dele há apenas a necessidade de entender como funcionam suas expressões regulares. Entretanto, a ferramenta não apresenta nenhuma representação passo a passo do processo ou explicação teórica da criação dos analisadores.

LISA (*Language Implementation System Based on Attribute Grammars*) é uma IDE (*Integrated Development Environment*) na qual o usuário pode especificar, gerar, compilar e executar programas em uma linguagem definida por ele. Este é um ambiente computacional de aprendizagem em inglês com o objetivo de facilitar o entendimento conceitual da construção de compiladores [7].

O diferencial da ferramenta consiste no valor didático por demonstrar as fases de análise léxica, sintática e semântica no processo de compilação. Através da IDE é possível visualizar o passo a passo por meio de animações em telas separadas em cada uma das fases de análise. A análise léxica tem como entrada ER escritas na linguagem Java que são convertidas em Autômatos Finitos Determinísticos (AFD). Este AFD gerado é convertido em um Scanner Java. Nesta etapa é apresentado uma tela específica para visualizar o AFD e animações são utilizadas para demonstrar a validação dos *tokens*.

A análise sintática utiliza GLC representadas com a *Backus Naur Form* (BNF) para implementar um parser *top-down*. Através de uma tela, exibe-se o passo a passo da construção da árvore sintática com animações por meio de um diagrama. A ordem de avaliação para as regras semânticas é derivada a partir do grafo de dependência e a demonstração do passo a passo é similar ao da análise sintática. O software LISA apresenta abordagens didáticas interessantes, porém não possui uma explicação escrita para detalhar os processos em

suas fases de análise. Além disto, sua interface é toda em inglês e os códigos analisados devem estar escritos em Java, o que dificulta o seu uso por estudantes que não dominam alguma dessas linguagens.

Inspirado no VisiCLANG (uma ferramenta que permite visualizar passo a passo o processo de compilação da linguagem CLANG) o VCOCO foi criado em 1998 com objetivo principal de reproduzir as funcionalidades do VisiCLANG para todas as linguagens. Isto foi possível por meio da utilização da ferramenta COCO/R, um meta compilador gerado a partir de BNF [10].

As características do VCOCO consistem na geração de compiladores visuais a partir de especificações COCO/R, flexibilidade da interface a partir deste compilador gerado, além de portabilidade. A ferramenta possui 5 interfaces: programa fonte, compilador, analisador léxico, analisador sintático e gramática, cada qual com depurador e pontos de parada para demonstrar o funcionamento do compilador.

Durante o processo de compilação, é possível acompanhar o isolamento dos lexemas no código fonte e, ao mesmo tempo, rastrear a execução do analisador sintático do compilador. Conforme os *tokens* são reconhecidos e os não-terminais expandidos, o progresso do analisador sintático pode ser observado na tela da gramática. Com essas funcionalidades, a ferramenta possibilita visualizar a relação entre o atual lexema no código fonte, o *token* que a gramática espera e o código do compilador que isola e reconhece os símbolos.

A identificação e análise desses trabalhos foi importante para possibilitar a compreensão do que já existe relacionado ao assunto, contribuindo para a definição dos requisitos funcionais e não funcionais do sistema desenvolvido. A Seção 6 apresenta uma comparação da ferramenta resultante com o trabalhos similares apresentados.

4 MÉTODOS

Pela solução prática proposta, que tem como fim a resolução de um problema específico, a pesquisa realizada neste artigo se classifica como aplicada. Segundo Prodanov e Freitas [9], a pesquisa aplicada objetiva gerar conhecimentos para aplicação prática dirigidos à solução de problemas específicos, envolvendo verdades e interesses locais.

Devido ao foco no processo e seu significado, além de não requerer o uso de métricas e técnicas estatísticas somado ao fato da impossibilidade dos resultados serem quantificados, a pesquisa neste artigo se dá como qualitativa. De acordo com Prodanov e Freitas [9], a pesquisa qualitativa considera que há uma relação dinâmica entre o mundo real e o sujeito, isto é, um vínculo indissociável entre o mundo objetivo e a subjetividade do sujeito que não pode ser traduzido em números.

Com relação aos procedimentos técnicos, a presente pesquisa se classifica como bibliográfica, pois teve seu embasamento construído através de livros e artigos. As explicações foram embasadas, em sua maioria, à partir de livros base da disciplina, por sua consolidação e metodologia (tabelas, figuras, etc). Os artigos foram selecionados nos sites Google Schoolar, SciELO e SOL(SBC Open Lib)[4], com os seguintes termos de busca:

- Ferramentas para ensino de compiladores.
- Ensino de compiladores com ferramentas didáticas.
- Compiler teaching tool.

[4]Sociedade Brasileira de Computação

Conforme Gil [5], a pesquisa bibliográfica se dá quando elaborada a partir de material já publicado, constituído principalmente de livros, artigos de periódicos e atualmente com material disponibilizado na Internet.

Por se ater a processos específicos usados na construção de um compilador, a pesquisa também se enquadra como estudo de caso. De acordo com Gil [5], estudo de caso envolve o estudo profundo e exaustivo de um ou poucos objetos de maneira que se permita o seu amplo e detalhado conhecimento.

A primeira etapa executada é de natureza cíclica e consistiu no levantamento bibliográfico para o referencial teórico em paralelo ao estudo de ferramentas com objetivos similares. O estudo foi feito a partir da leitura de resumos e a construção de uma trilha bibliográfica para poder aprofundar em artigos mais pertinentes e próximos ao tema apresentado.

A segunda etapa consistiu na modelagem do sistema. Nela foram definidos os principais requisitos funcionais e não funcionais do sistema. A terceira etapa se deu com a implementação de uma aplicação web, responsiva e interativa, com o objetivo de proporcionar uma melhor experiência para o usuário, tendo como principal proposta exibir o passo a passo dos processos executados pelo compilador de acordo com as entradas recebidas. Para realizar a implementação, o projeto foi dividido em duas partes: *frontend* e *backend*. No *frontend* foi usado o *framework Angular*, que possibilitou a construção da interface web. Para o *backend* foi utilizada a linguagem de programação *Python* para realizar o processamento de acordo com as entradas do usuário. Nesta etapa também foram realizados os testes preliminares, envolvendo os autores e orientadores do projeto, assim como outros cinco alunos que já cursaram a disciplina de Compiladores. Os problemas identificados e as sugestões obtidas nestes testes estão sendo incorporadas à nova versão do sistema.

Após a conclusão destes ajustes, dar-se-á início a quarta etapa, a qual consistirá em um teste mais amplo, por professores e estudantes de diferentes instituições de educação superior, que já tenham cursado ou estejam cursando a disciplina de Compilaodres.

5 RESULTADOS

O objetivo principal do sistema proposto é auxiliar no processo de ensino-aprendizagem de compiladores por meio de uma ferramenta interativa, cujas funcionalidades são focadas nas primeiras etapas do processo de compilação, as análises léxica e sintática. Para cada uma das etapas, o sistema apresenta uma explicação sobre seus aspectos teóricos. O passo a passo de sua execução é realizado na prática pelo compilador de acordo com o avanço e interação do usuário. A Figura 1 apresenta a tela inicial do sistema, contendo informações e explicações gerais sobre o projeto. O sistema pode ser acessado a partir deste link https://compiler-tool-fe.herokuapp.com/home.

A explicação da teoria aborda os tópicos de maneira didática com ilustrações e textos a fim de embasar os conhecimentos necessários para melhor compreensão das funcionalidades práticas, além de relacionar as etapas constituintes do processo de compilação.

A interatividade se dá pela possibilidade de o usuário inserir o próprio código e operar o sistema da forma desejada para acompanhar como esse código é analisado. O resultado é expresso por meio de representações visuais com destaque para cada passo do

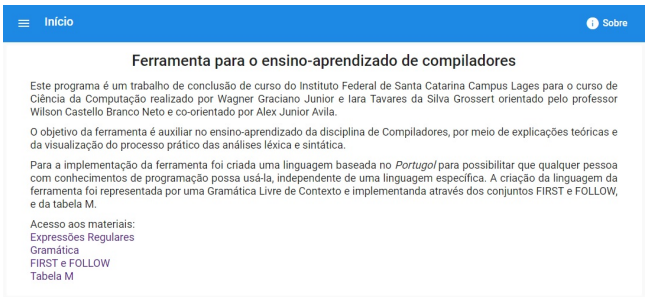

Figura 1: Página inicial

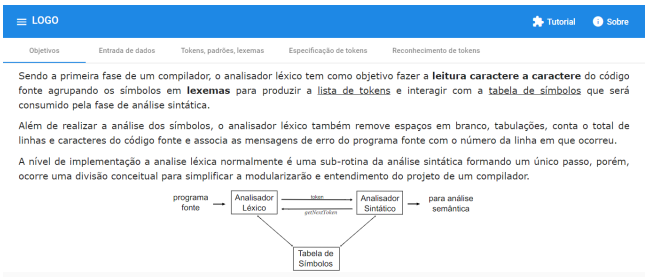

Figura 2: Menu horizontal com explicação teórica da análise léxica

processo de maneira animada, acompanhada de uma explicação específica do processo, com o propósito de possibilitar um melhor entendimento por tornar visível o conteúdo como um todo.

5.1 Análise Léxica

A explicação teórica sobre análise léxica está organizada em cinco partes por meio de um menu de navegação horizontal, como mostra a Figura 2.

A aba *Objetivos* apresenta o papel da análise léxica, o que ela recebe como entrada e gera com saída. A aba *Entrada de dados* contém explicação sobre a leitura do código fonte através de buffers duplos com especificações e imagens ilustrativas do processo. A aba *Tokens, padrões e lexemas* possui as definições destes termos e explica o relacionamento entre eles. A aba *Especificação tokens* cita que *tokens* são representados por meio de ER, define cadeia, alfabeto e linguagem. Por fim, a aba *Reconhecimento de tokens* define Autômatos Finitos Determinísticos (AFD) e Autômatos Finitos Não-Determinísticos (AFND) para citar a relação entre validação de *tokens* por meio de ER e a possibilidade de converte-las para autômatos.

Para auxiliar na explicação, palavras importantes são destacadas em negrito e definições cuja explicações mais detalhadas se fazem necessárias são sublinhadas e abordadas por meio de uma janela de diálogo acionada através de um clique.

Um menu vertical expansível possibilita a navegação entre os diferentes módulos do sistema, que são: Análise Léxica - Teoria; Análise Léxica - Prática; Análise Sintática - Teoria; Análise Sintática - Prática. A Figura 3 apresenta parte da interface do módulo

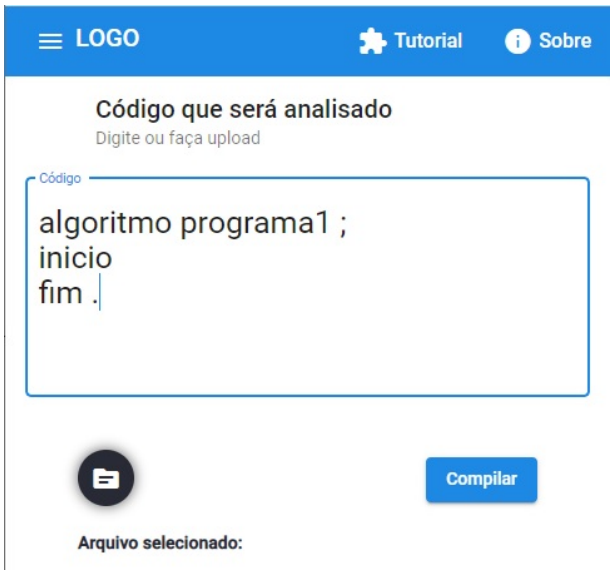

Figura 3: Código digitado pelo usuário

Análise Léxica - Prática, destacando o campo editável que permite ao usuário digitar o código fonte a ser verificado pelo analisador.

Para permitir ao usuário inserir programas para acompanhar o processo de compilação, foram criadas as expressões regulares (ER) para possibilitar a implementação da análise léxica, responsável pela leitura caractere a caractere do código fonte. O processo de criação das ER exigiu, primeiramente, a definição da linguagem. Visando a característica didática e simplicidade, utilizou-se uma pseudolinguagem baseada no português estruturado (Portugol)[5].

O Quadro 1 apresenta as ER utilizadas para permitir a implementação da primeira fase do compilador, que contêm todos os símbolos aceitos na linguagem. A Figura 4 expõe o respectivo autômato para todas ER apresentadas no Quadro 1 que pode ser controlado pelo usuário na interface prática do programa.

A Figura 5 apresenta a interface criada para demonstrar a parte prática da análise léxica. Na primeira coluna encontra-se a lista de *tokens* do código fonte digitado pelo usuário. O autômato indicado na coluna do meio é a parte do autômato completo que valida o *token* selecionado na lista da esquerda de forma animada, ressaltando com uma cor diferente o estado ativo e destacando, também, o símbolo que está sendo validado no código. A terceira coluna mostra a explicação dos passos durante a validação, com o botão *próximo* e *anterior* o usuário pode controlar o avanço do processo. Cada coluna desta interface é explicada detalhadamente nas Figuras 6, 7 e 8, respectivamente.

A lista de *tokens*, apresentada na Figura 6, possui o número identificador do *token*, chave e valor correspondentes. O conteúdo da lista consiste nos *tokens* associados a partir do código digitado apresentado na Figura 3, como por exemplo, na linha de número 1 a variável *programa1* corresponde à chave identificador, na linha número 2 o *;* é associado à chave *;*, já na linha de número 3 a palavra reservada *inicio* é associada a chave *palavraReservada*.

[5]https://pt.wikipedia.org/wiki/Portugol

Quadro 1: Expressões regulares construídas

Chave	Valor
Atribuição	<-
=	=
>=	>=
>	>
<	<
<=	<=
<>	<>
+	+
-	-
*	*
^	^
/	/
%	%
&&	&&
||	||

Chave	Valor
(	(
)	)
[	[
]	]
,	,
;	;
:	:
digito	[0-9]
letra	[a-zA-Z]
número_decimal	digito+ [.] digito+
número_inteiro	digito+
identificador	(letra | _)(letra | _ | digito)*
literal	"*"
comentário	// .*\n
comentário_bloco	/[*] .* [*]/

Na Figura 7 são mostradas as transições do autômato responsável por validar o código fonte digitado pelo usuário. Os botões permitem o avanço e retorno dos estados durante a validação. O autômato está validando o segundo lexema da lista de *tokens*, ou seja, *programa1*. O autômato do canto superior esquerdo mostra o estado inicial destacado, indicando o início do processo. O autômato do canto superior direto mostra o estado *q28* destacado após receber o caractere *p*. O autômato do canto inferior esquerdo demonstra que o autômato continua neste estado enquanto os próximos símbolos forem letras ou dígitos. O autômato do canto inferior direto representa que o autômato efetuou a transição para o estado final *q29* ao receber um caractere de espaço em branco, indicando o reconhecimento do *token* identificador.

A Figura 8 demonstra a explicação dos passos para validação deste *token*, permitindo a visualização de sua chave e valor, juntamente com os estados que foram percorridos pelo automato, permitindo que o usuário controle o avanço da validação através dos botões *Próximo token* e *Token anterior*. Em caso de erro, o sistema apresenta uma mensagem indicando o estado em que o autômato se encontrava e o caractere lido que levou ao erro, para que o usuário possa visualizar no autômato como o erro foi identificado.

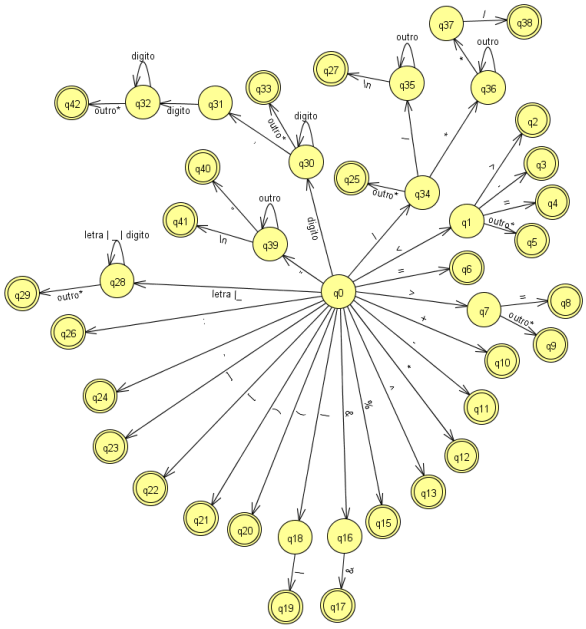

Figura 4: Autômato finito determinístico

Figura 5: Funcionamento da análise léxica

5.2 Análise Sintática

Para manter a simplicidade e familiaridade de usabilidade do usuário, a explicação teórica sobre análise sintática também está organizada em cinco partes por meio de um menu de navegação horizontal assim como a explicação teórica sobre análise léxica.

A aba *O que é* apresenta a função da análise sintática, o que ela recebe como entrada e processa em termos de *tokens* e mensagens de erro. A aba *Gramáticas Livres de Contexto* contém explicações, através de textos e imagens, de seus conceitos e relação com o analisador sintático . *Árvore de derivação* explica a abordagem teórica do analisador sintático assim como os tipos de analisadores ascendente e descendente. A aba *Conjuntos First e Follow* especifica os termos e apresenta exemplos por meio de imagens. Por fim, a aba *Tabela M* apresenta como esta é gerada a partir dos conjuntos First e Follow.

Para a implementação do analisador sintático preditivo foi construída uma Gramática Livre de Contexto para representar as construções válidas da linguagem. Algumas produções desta gramática

Figura 6: Lista de *tokens* do código digitado pelo usuário

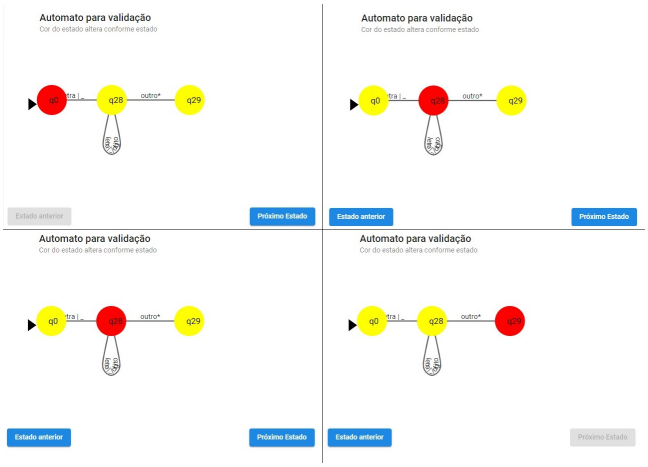

Figura 7: Autômato da tela prática da análise sintática

são apresentadas no Quadro 2 e a gramática completa encontra-se no seguinte link: t.ly/T9sZ. A partir desta gramática, foi possível montar os conjuntos FIRST, FOLLOW e a tabela M.

Pela definição da regra de produção do não-terminal de partida *PROGRAMA*, todo o código fonte começa com a palavra reservada *algoritmo*, seguido de um identificador e ;. Ainda na mesma derivação, encontra-se o não-terminal *BLOCO* e o terminal .. O não-terminal BLOCO divide o programa em seções de CONSTANTES, VARIÁVEIS, FUNÇÕES e COMANDOS, formando todas as possíveis derivações de escrita do código fonte.

O Quadro 3 mostra uma parte do conjunto FIRST, criado com base na gramática apresentada. Este conjunto representa todos

Figura 8: Explicação teórica dos passos do analisador léxico

Quadro 2: Gramática

PROGRAMA → algoritmo identificador ; BLOCO .
BLOCO → SEÇÃO_CONSTANTES SEÇÃO_VARIÁVEIS
SEÇÃO_FUNÇÕES SEÇÃO_COMANDOS
SEÇÃO_CONSTANTES → const LISTA_CONSTANTES | ϵ
LISTA_CONSTANTES → DEFINIÇÃO_CONSTANTE ; NOVA_CONSTANTE
NOVA_CONSTANTE → DEFINIÇÃO_CONSTANTE ; NOVA_CONSTANTE | ϵ
DEFINIÇÃO_CONSTANTE → identificador = CONSTANTE ;
SEÇÃO_VARIÁVEIS → var LISTA_VARIÁVEIS | ϵ
LISTA_VARIÁVEIS → DECLARAÇÃO_VARIÁVEL; NOVA_VARIÁVEL
NOVA_VARIÁVEL → DECLARAÇÃO_VARIÁVEL; NOVA_VARIÁVEL | ϵ
DECLARAÇÃO_VARIÁVEL → LISTA_IDENTIFICADORES : TIPO ;
LISTA_IDENTIFICADORES → identificador NOVO_IDENTIFICADOR
NOVO_IDENTIFICADOR → , identificador NOVO_IDENTIFICADOR | ϵ
TIPO → TIPO_BASE | vetor [num_int] de TIPO_BASE | string [num_int]
TIPO_BASE → caractere | real | inteiro | logico
TIPO_RETORNO → TIPO | void
SEÇÃO_FUNÇÕES → LISTA_FUNÇÕES | ϵ

os símbolos terminais que podem iniciar a derivação de um não-terminal. Por exemplo, o FIRST de *PROGRAMA* define que todo o código da linguagem deve começar com a palavra reservada *algoritmo*. Já o não-terminal *BLOCO* pode ser iniciado por diversos terminais, como por exemplo *const*, *var*, *void*, *vetor*, etc. Isto é possível pois o usuário pode declarar variáveis, funções, constantes, ou inserir diretamente o programa, definido pelo terminal *inicio* após o cabeçalho.

O conjunto FOLLOW, conforme mostra o Quadro 4, possui o conjunto de terminais que podem aparecer à direita de um não terminal em uma sentença válida. O FOLLOW de PROGRAMA consiste no símbolo $ (denota um terminal virtual marcador de fim da entrada ou fim de arquivo). O FOLLOW de BLOCO pode ser apenas . pois é o primeiro terminal após o não-terminal de BLOCO nas regras de produção.

Os conjuntos FIRST e FOLLOW completos encontram-se no seguinte link t.ly/6aJO

A tabela M, tem a função de auxiliar o analisador sintático na escolha da produção correta com base no próximo símbolo da entrada. A primeira linha, apresentada pelo Quadro 5, contém todos os símbolos terminais e a primeira coluna os não terminais. Como

Quadro 3: Conjunto FIRST

FIRST (PROGRAMA) = {algoritmo}
FIRST (BLOCO) = {const, var, void, vetor, string, caractere, real, inteiro...}
FIRST (SEÇÃO_CONSTANTES) = {const, ϵ}
FIRST (LISTA_CONSTANTES) = {identificador}
FIRST (NOVA_CONSTANTE) = {identificador, ϵ}
FIRST (DEFINIÇÃO_CONSTANTE) = {identificador}
FIRST (SEÇÃO_VARIÁVEIS) = {var, ϵ}
FIRST (LISTA_VARIÁVEIS) = {identificador}
FIRST (NOVA_VARIÁVEL) = {identificador, ϵ}
FIRST (DECLARAÇÃO_VARIÁVEL) = {identificador}
FIRST (LISTA_IDENTIFICADORES) = {identificador}
FIRST (NOVO_IDENTIFICADOR) = {, , ϵ}
FIRST (TIPO) = {vetor, string, caractere, real, inteiro, logico}
FIRST (TIPO_BASE) = {caractere, real, inteiro, logico}
FIRST (TIPO_RETORNO) = {vetor, string, void, caractere, real, inteiro, logico}
FIRST (SEÇÃO_FUNÇÕES) = {void, vetor, string, caractere, real, inteiro, logico, ϵ}
FIRST (LISTA_FUNÇÕES) = {vetor, string, void, caractere, real, inteiro, logico}
FIRST (NOVA_FUNÇÃO) = {void, vetor, string, caractere, real, inteiro, logico, ϵ}
FIRST (DECLARAÇÃO_FUNÇÃO) = {vetor, string, void, caractere, real...}

Quadro 4: Conjunto FOLLOW construído à partir das gramáticas

FOLLOW (PROGRAMA) = {$}
FOLLOW (BLOCO) = {.}
FOLLOW (SEÇÃO_CONSTANTES) = {var, void, vetor, string, caractere, real...}
FOLLOW (LISTA_CONSTANTES) = {var, void, vetor, string, caractere, real...}
FOLLOW (NOVA_CONSTANTE) = {var, void, vetor, string, caractere, real...o}
FOLLOW (DEFINIÇÃO_CONSTANTE) = {;}
FOLLOW (SEÇÃO_VARIÁVEIS) = {void, vetor, string, caractere, real, inteiro...}
FOLLOW (LISTA_VARIÁVEIS) = {void, vetor, string, caractere, real, inteiro...}
FOLLOW (NOVA_VARIÁVEL) = {void, vetor, string, caractere, real, inteiro...}
FOLLOW (DECLARAÇÃO_VARIÁVEL) = {;}
FOLLOW (LISTA_IDENTIFICADORES) = {:}
FOLLOW (NOVO_IDENTIFICADOR) = {:}
FOLLOW (TIPO) = {;, identificador}
FOLLOW (TIPO_BASE) = {;, identificador}
FOLLOW (TIPO_RETORNO) = {identificador}
FOLLOW (SEÇÃO_FUNÇÕES) = {inicio}
FOLLOW (LISTA_FUNÇÕES) = {inicio}
FOLLOW (NOVA_FUNÇÃO) = {inicio}
FOLLOW (DECLARAÇÃO_FUNÇÃO) = {void, vetor, string, caractere, real...}

exemplo, o terminal de partida PROGRAMA ao receber o terminal *algoritmo* realiza o processo de derivação de PROGRAMA por *algoritmo identificador; BLOCO* ., caso receba qualquer terminal diferente de algoritmo retorna um erro. O terminal BLOCO ao receber um *identificador* retorna um erro, pois o cruzamento desta linha por coluna na tabela M não apresenta nenhuma produção.

A Figura 9 apresenta a interface criada para demonstrar a parte prática da análise sintática. A primeira coluna contém a lista de *tokens* do código fonte digitado pelo usuário, validado pela análise léxica. A produção e a árvore de derivação, indicadas na coluna do meio, validam *token* por *token* gerando a árvore de derivação a partir da produção e da lista de *tokens* apresentada na parte superior da coluna. A produção é atualizada conforme a interação do usuário com a terceira coluna. A terceira coluna mostra a explicação dos passos durante a validação, com o botão *próximo* e *anterior* o usuário pode controlar o avanço do processo. Cada coluna tem uma explicação detalhada através das Figuras 10, 11 e 12.

A lista de *tokens*, apresentada na Figura 10 possui os *tokens* associados a partir do código digitado pelo usuário na análise léxica, como explicado na Seção 5.1.

Quadro 5: Tabela M construída a partir dos conjuntos FIRST e FOLLOW

	algoritmo	identificador
PROGRAMA	algoritmo identificador ; BLOCO .	
BLOCO		
SEÇÃO_CONSTANTES		
LISTA_CONSTANTES		DEFINIÇÃO_CONSTANTE ; NOVA_CONSTANTE
NOVA_CONSTANTE		DEFINIÇÃO_CONSTANTE ; NOVA_CONSTANTE
DEFINIÇÃO_CONSTANTE		identificador = CONSTANTE ;
SEÇÃO_VARIÁVEIS		
LISTA_VARIÁVEIS		DECLARAÇÃO_VARIÁVEL; NOVA_VARIÁVEL
NOVA_VARIÁVEL		DECLARAÇÃO_VARIÁVEL; NOVA_VARIÁVEL
DECLARAÇÃO_VARIÁVEL		LISTA_IDENTIFICADORES : TIPO ;
LISTA_IDENTIFICADORES		identificador NOVO_IDENTIFICADOR

Figura 9: Funcionamento análise sintática

Figura 10: Lista de *tokens* do código digitado pelo usuário

A árvore de derivação, apresentada na Figura 11, foi gerada à partir da lista de *tokens* e das regras de produção. A raiz da árvore começa pelo símbolo inicial da gramática PROGRAMA. Os níveis

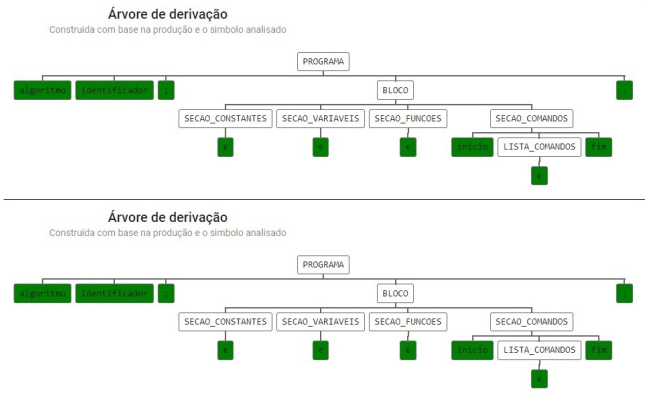

Figura 11: Dois últimos estados da árvore sintática validando o código fonte

Figura 12: Explicação teórica dos passos do analisador sintático

subsequentes da árvore apresentam as respectivas derivações até a validação do programa. A parte superior da imagem demonstra o texto base para todas explicações da análise sintática. Neste caso em específico, a ação foi *comparar*, pois o símbolo analisado na árvore era um terminal .. Como o símbolo está de acordo com o esperado na lista de *tokens*, o mesmo foi destacado em verde na árvore e na última frase da descrição da explicação. Caso um erro na derivação for encontrado, o destaque será em vermelho.

A parte inferior da imagem representa o último estado da árvore analisando um programa sintaticamente correto. O botão de *próximo* é desabilitado e a base to texto da explicação também é alterada para explicitar a validação da análise sintática. Devido a extensão da árvore, este componente é deslizável, permitindo ao usuário uma melhor visualização.

A Figura 12 representa a explicação dos passos do processo de derivação e comparação da árvore com a lista de *tokens*, permitindo a visualização do passo a passo da derivação quando símbolo analisado na folha da árvore for um não-terminal, e a comparação com a lista de *tokens* quando for um terminal. Este passo a passo pode ser controlado através dos botões *próximo* e *anterior*. Em caso de erro, o símbolo da árvore de derivação e o próximo símbolo da lista de *tokens* que não pode ser validado são pintados de vermelho para que o usuário compreenda exatamente em que passo o erro foi identificado.

6 DISCUSSÃO

As ferramentas citadas na Seção 3 atendem ao objetivo comum de auxiliar no processo de ensino/aprendizagem de compiladores, cada qual com suas especificidades. O diferencial da ferramenta proposta neste artigo encontra-se em sua abordagem intuitiva para apresentar, passo a passo, as etapas iniciais da compilação (análise léxica e análise sintática) através de uma interface gráfica interativa e em português. As vantagens proporcionadas pela interface no sistema proposto se fundamentam por sua portabilidade, pois é web e responsiva, permitindo assim interação através de dispositivos móveis. Além disto, a ferramenta apresenta o passo a passo detalhado, que como pôde ser percebido pelas ferramentas similares apresentadas, é uma funcionalidade escassa. Por fim, uma funcionalidade exclusiva da ferramenta, tendo como base as ferramentas similares apresentadas, consiste em sua explicação teórica das fases abordadas.

O Quadro 6 apresenta de forma resumida os principais pontos para a comparação das ferramentas citadas com a proposta. A coluna *Etapas* presente no Quadro 6 cita os termos Lex, Sint e Sem que representam, respectivamente, análise léxica, análise sintática e análise semântica.

A forma como a ferramenta pode ser usada dentro da disciplina de Compiladores pode variar, conforme o interesse e o planejamento de cada professor. Ela pode ser usada dentro de uma metodologia tradicional, com aulas expositivas e dialogadas, para que os alunos possam revisar os conteúdos teóricos vistos e resolver os exercícios propostos. Uma outra possibilidade é o seu uso dentro da metodologia de Sala de Aula Invertida. Neste caso, o professor pode solicitar que os alunos acessem a ferramenta para ler os conteúdos teóricos antes da aula, assim como analisar alguns códigos pré-definidos. A partir desta experiência, o professor pode conduzir a discussão para sanar as dúvidas e propor novos exercícios após a aula, também usando a ferramenta. Esta é a abordagem que será usada durante a avaliação a ser realizada com a turma de Compiladores do Instituto Federal de Santa Catarina, Câmpus Lages (IFSC-Lages), no semestre de 2022/1.

7 CONSIDERAÇÕES FINAIS

Este trabalho teve o objetivo de elaborar uma ferramenta web para o ensino-aprendizado da disciplina de compiladores nos cursos de computação, capaz de auxiliar o entendimento através de explicações teóricas juntamente com o passo a passo da parte prática. A ferramenta possibilita a visualização da validação do código fonte na análise léxica por meio da representação gráfica de autômatos que são gerados conforme cada símbolo da lista de *tokens*. A parte

Quadro 6: Resumo das características das ferramentas similares

	Etapas	Interface Gráfica	Explicação Teórica	Passo a Passo das Etapas
LexSint	Lex Sint	Não	Não	Resumido
Verto	Todas	Desktop	Não	Detalhado
Flex	Lex	Não	Não	Não
JFlex	Lex	Desktop	Não	Não
GALS	Lex Sint	Desktop	Não	Não
LISA	Lex Sint Sem	Desktop	Não	Resumido
VCOCO	Lex Sint	Desktop	Não	Resumido
Sistema Proposto	Lex Sint	Web	Sim	Detalhado

prática da análise sintática mostra a construção da árvore de derivação acompanhada da explicação passo a passo do processo. Ambas as análises permitem que o usuário controle o avanço da validação.

Para a implementação da ferramenta, na análise léxica foram elaborados as Expressões Regulares e os Autômatos Finitos Determinísticos. O processamento para a validação do código fonte é feito no *backend* que gera a lista de estados e a lista de *tokens* a ser enviado para o *frontend*, o qual utiliza essas informações para fazer a confecção do autômato animado e a explicação interativa.

Para a implementação das análises léxica e sintática, foi necessária a criação de uma linaguagem baseada no *Portugol* representada por uma Gramática Livre de Contexto e implementanda através dos conjuntos FIRST e FOLLOW, e da tabela M. O processamento também é feito no *backend* que recebe uma lista de *tokens* e retorna para o *frontend* uma lista de ações (comparação ou derivação) e uma lista de produções (obtida através da tabela M e do símbolo da lista de *tokens* em análise), utilizados para explicitar a construção passo a passo da árvore.

7.1 Trabalhos Futuros

Após a conclusão desta etapa do trabalho, o sistema foi disponibilizado aos alunos do curso de Ciência da Computação do IFSC-Lages para teste. Contudo, pelo fato do curso ter entrada anual, a disciplina de Compiladores não estava sendo ofertada no período de testes, sendo convidados para o teste aqueles alunos que já haviam cursado a disciplina no semestre ou em anos anteriores. Além disto, estes alunos não realizavam atividades presenciais em função da pandemia de COVID-19 no período. Estes dois fatores reduziram o engajamento no teste e apenas 5 alunos usaram o sistema e preencheram o formulário de avaliação. Em função do baixo número de avaliações, os resultados não foram incluídos no artigo.

Embora o número de respondentes tenha sido pequeno, percebeu-se nas respostas a necessidade de elaborar um manual de uso do sistema, com alguns exemplos prontos que possam ser analisados até que o usuário tenha condições de elaborar os seus próprios exemplos. Esta alteração será feita antes da próxima disponibilização para testes, que será realizada para que os alunos que cursarão a disciplina de Compiladores no semestre 2022/1 possam usá-la e avaliá-la à medida que avançam na disciplina. Após este teste e a realização das melhorias necessárias, professores e alunos de diferentes instituições de ensino serão convidados a usá-la durantes as suas aulas e após esta etapa, preencher um formulário de avaliação.

Em paralelo a esta atividade será dado início ao novo projeto que contempla a construção dos módulos que contemplarão as etapas de Análise Semântica e Geração de Código Intermediário.

REFERÊNCIAS

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. 2008. *Compiladores Princípios, Técnicas e Ferramentas* (2nd ed.). LTC.

[2] Gustavo Alkmim and Bráulio de Mello. 2012. Ferramenta de apoio às fases iniciais do ensino de linguagens formais e compiladores. *Brazilian Symposium on Computers in Education (Simpósio Brasileiro de Informática na Educação - SBIE)* 1, 1.

[3] Sachs Barbosa, Robson Bonidia, and Joao Coelho Neto. 2019. Flex, JFlex e GALS: Ferramentas de Apoio ao Ensino de Compiladores.

[4] Keith D. Cooper and Linda Torczon. 2012. *Engineering a Compiler* (2nd ed.). Elsevier.

[5] Antonio Carlos Gil. 1991. *Como Elaborar Projetos de Pesquisa* (3rd ed.). ATLAS S.A.

[6] João José Neto. 2016. *Introdução à compilação* (1st ed.). Elsevier.

[7] M. Mernik and V. Zumer. 2003. An educational tool for teaching compiler construction. *IEEE Transactions on Education* 46, 1, 61–68.

[8] Fernando Magno Quintão Pereira. 2020. Pesquisa em Compiladores. https://homepages.dcc.ufmg.br/ fernando/projects/CompilerResearchUFMG.pdf. https://homepages.dcc.ufmg.br/~fernando/projects/CompilerResearchUFMG.pdf Março 25, 2020.

[9] Cleber Cristiano Prodanov and Ernani Cesar de Freitas. 2013. *Metodologia do Trabalho Científico: Métodos e Técnicas da Pesquisa e do Trabalho Acadêmico* (2nd ed.). Universidade Feevale.

[10] R. Daniel Resler and Dean M. Deaver. 1998. VCOCO: A Visualisation Tool for Teaching Compilers. In *Proceedings of the 6th Annual Conference on the Teaching of Computing and the 3rd Annual Conference on Integrating Technology into Computer Science Education: Changing the Delivery of Computer Science Education* (Dublin City Univ., Ireland) *(ITiCSE '98)*. Association for Computing Machinery, New York, NY, USA, 199–202.

[11] Carlos Scheider, Liliana Passerino, and Ricardo Oliveira. 2005. Compilador Educativo VERTO: ambiente para aprendizagem de compiladores. *RENOTE* 3.

[12] Ravendra Singh, Vivek Sharma, and Manish Varshney. 2009. *Design and Implementation of Compiler* (1st ed.). New Age International.

[13] Elizabeth White, Ranjan Sen, and Nina Stewart. 2005. Hide and show: using real compiler code for teaching. *ACM Sigcse Bulletin* 37, 12–16.