Ferramenta para o ensino de compiladores

Wagner Graciano Junior, Iara Tavares da S. Grossert
Wilson Castello Branco Neto, Alex Junior Avila

[1]Instituto Federal de Santa Catarina – Campus Lages (IFSC)
Rua Heitor Villa Lobos, 222 – 88.506-400 – Lages – SC – Brasil

wagner_graciano0101@hotmail.com, iara_grossert@hotmail.com

wilson.castello@ifsc.edu.br, alex_avila11@hotmail.com

Abstract. *This paper proposes a didactic tool to assist in the teaching-learning of Compilers, through theoretical explanations and the visualization of the practical process of lexical and syntactic analysis. The tool is a responsive web application that enables the study of theoretical concepts and allows users to type its own source code and follow its process of analysis. It presents a detailed step-by-step with explanations of the processing according to the user's interaction. In order to implement such tool, a language based on Portugol was created to make it possible for anyone with programming knowledge to use it, regardless of a specific language.*

Resumo. *Este artigo propõe uma ferramenta didática para auxiliar no ensino-aprendizado da disciplina de Compiladores, por meio de explicações teóricas e da visualização do processo prático das análises 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. Para a implementação da ferramenta foi criada uma linguagem baseada no Portugol para possibilitar que qualquer pessoa com conhecimentos de programação possa usá-la, independente de uma linguagem específica.*

1. Introdução

O compilador, de acordo com José Neto (2016), 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 (2020), 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. (2008), 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. (2008) também apontam que devido a abrangência da disciplina, entendê-la demanda conhecimento prévio de diversas áreas da computação, dentre elas destacam-se: 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.

Tendo em vista estas peculiaridades, é possível inferir que 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. (2005), 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 (2003), muitas ferramentas foram utilizadas no passado como geradores de scanner, parser e compilador. Entretanto, essas ferramentas possuem pouco ou nenhum valor didático.

Considerando os problemas levantados, este projeto tem como principal objetivo construir uma aplicação interativa para auxiliar no processo de ensino-aprendizagem das etapas de análise léxica e sintática da disciplina de compiladores.

Os objetivos específicos para a execução deste projeto consistem em:

- Estudar ferramentas e projetos que tenham finalidades similares ao da solução proposta;
- Modelar o sistema embasado nas pesquisas e conhecimentos adquiridos;
- Implementar uma ferramenta web que apresente os conceitos teóricos e o funcionamento dos analisadores léxico e sintático.

Este trabalho está dividido em cinco seções. Após esta introdução, a seção 2 apresenta as pesquisas realizadas para embasar o artigo, além de incluir aspectos como dificuldades encontradas no ensino de compiladores, ferramentas e assuntos relacionados. A seção 3 trata das etapas e a classificação da pesquisa. A seção 4 possui detalhes sobre as fases de modelagem e implementação do sistema. A seção 5 especifica as conclusões a partir dos resultados obtidos com base nas avaliações realizadas.

2. Referencial Teórico

Nesta seção são apresentados os conceitos que embasam o sistema proposto neste artigo. Na seção 2.1 é realizada uma breve introdução sobre o compilador e suas fases. As funcionalidades, especificações, atribuições e processos da análise léxica e sintática são

introduzidos nas seções 2.2, 2.3 e 2.4, respectivamente. Por fim, são apresentadas algumas ferramentas com propósito similar na seção 2.5.

2.1. Compiladores

Segundo Cooper e Torczon (2012), o compilador é simplesmente um programa de computador que traduz outros programas para prepará-los para execução. Para Singh et al. (2009), o compilador traduz um programa em linguagem fonte para uma linguagem alvo, este processo denomina-se compilação.

De acordo com Aho et al. (2008), existem duas fases no processo de compilação, a análise e a síntese, também denominadas de *front-end* e *back-end*, respectivamente. Conforme é mostrado na Figura 1, as primeiras etapas da compilação consistem em um núcleo de análise, que são divididas em léxica, sintática e semântica e juntamente com o gerador de código intermediário formam o *front-end*. A síntese, que consiste nas últimas duas etapas apresentadas na mesma Figura 1, constrói o programa alvo desejado. O gerenciamento da tabela de símbolos e a manipulação de erros são outros dois módulos que interagem com as demais durante todo processo.

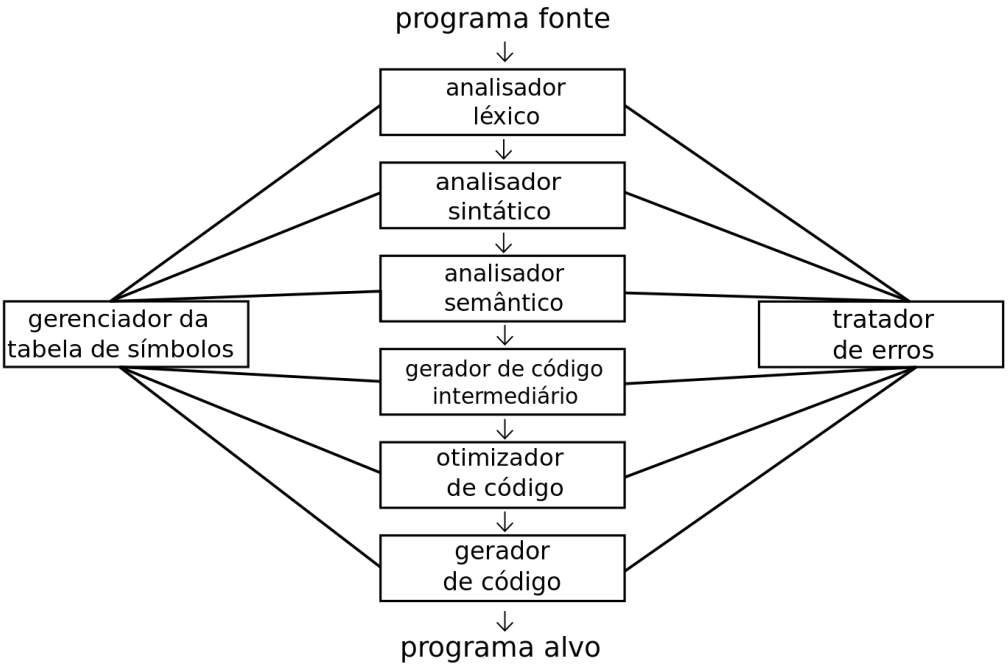

Figura 1. Fases do compilador
Fonte: Aho et al. (1986)

As principais atividades realizadas pelo compilador de acordo com Aho et al. (2008) 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 coesiva, como, por exemplo, um identificador ou uma palavra-chave (if, while, dentre outros). A sequência dos caracteres que formam um *token* é chamada lexema.

- Análise sintática: envolve o agrupamento dos *tokens* do programa fonte em frases gramaticais, que são usadas pelo compilador a fim de sintetizar a saída. Usualmente, as frases gramaticais do programa fonte são representadas por uma árvore gramatical.
- Análise semântica: verifica os erros semânticos no programa fonte e captura as informações de tipo para a fase subsequente de geração de código. Nesta etapa é importante ressaltar o componente da verificação de tipos, nela o compilador checa 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* (sequência de instruções com no máximo três operandos em cada uma delas).
- Otimização de código: realiza melhorias no código intermediário para gerar um código de máquina mais eficiente em tempo de execução.
- 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.

O processo de compilação, de acordo com Singh et al. (2009), é dividido em diversas fases que se comunicam entre si através de arquivos temporários. Estas fases contam com programas auxiliares presentes do início ao fim do processo, que são normalmente divididos em *pré-processador, montador* e *carregador*.

- Pré-processador: o programa fonte pode ser dividido em vários arquivos e módulos, o pré-processador faz a coleta destes módulos e os liga preparando-os para o processo de compilação.
- Montador: após o processo de compilação, normalmente um programa em *código de montagem* é gerado. O montador traduz este código de montagem em código de máquina que será efetivamente executado.
- Carregador: reúne os arquivos e módulos após a tradução para o código de máquina organizando-os na memória.

2.2. Análise Léxica

Singh et al. (2009) indicam 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, com o objetivo de incorporá-los a lista de símbolos válidos.

Segundo Aho et al. (2008), sua tarefa principal é a de ler os caracteres de entrada e produzir uma sequência de *tokens* que o analisador sintático utiliza para a análise. O analisador léxico e o *parser* formam um par produtor-consumidor. O analisador léxico produz *tokens* e o parser os consome. Além dessas tarefas, realiza a remoção de espacos em branco, comentários, reconhece constantes, identificadores e palavras chave além de controlar o número de caracteres para correlacionar o número da linha à mensagem de erro.

A realização da leitura do programa fonte pode ser acelerada com técnicas especializadas de *buffering* e *sentinelas*. *Buffering* consiste na leitura de um bloco de caracteres de um programa fonte para um *array* na memória. Segundo Aho et al. (2008), um esquema importante envolve dois *buffers* que são recarregados alternadamente. Cada *buffer* possui o mesmo tamanho, que normalmente corresponde ao tamanho do bloco de disco e contam com dois apontadores: *lexemeBegin* e *forward*. Esta técnica denomina-se *buffer duplo*.

- O apontador *lexemeBegin* marca o início do lexema corrente, cuja extensão está se tentando determinar.
- O apontador *forward* lê adiante, até que haja um casamento de padrão.

Sentinelas são caracteres especiais que servem para delimitar o final de um *buffer*. Conforme Aho et al. (2008), sentinela é um caractere especial que não pode fazer parte do programa fonte, e uma escolha natural é o caractere de fim de arquivo (eof). Qualquer eof que apareça em outro lugar que não seja o fim de um *buffer*, significa que a entrada chegou ao fim.

Expressões regulares (ER) são notações criadas para descrever linguagens, que podem ser formadas a partir da aplicação de operações de união, concatenação e fechamento aos símbolos de um alfabeto. José Neto (2016) indica que 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 (átomos ou símbolos) 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 (2012) afirmam que para qualquer AF, é possível descrever sua linguagem usando uma notação chamada ER. A linguagem descrita por uma ER e reconhecida por um AF é chamada linguagem regular.

O analisador léxico faz o reconhecimento dos caracteres que formam as cadeias pertencentes a uma linguagem através de um AF. Para facilitar o processo de construção do mesmo, pode-se utilizar ferramentas que convertem expressões regulares em AF correspondentes.

AF é um diagrama de transição baseado em nós e arestas, cujos nós representam os estados e as arestas representam a transição entre os nós de acordo com o caractere de entrada. Cooper e Torczon (2012) definem AF como um formalismo para reconhecer um alfabeto, que possui um conjunto finito de estados divididos em um estado inicial, função de transição e um ou mais estados aceitantes (estados finais). O conjunto de palavras aceito por um AF formam uma linguagem.

Segundo Aho et al. (2008), os autômatos finitos são reconhecedores sobre cada possível cadeia de caractere de entrada. Eles podem ser descritos de duas formas, autômatos finitos determinísticos (AFD) ou autômatos finitos não determinísticos (AFND).

- AFD possuem, para cada estado e para cada símbolo de seu alfabeto de entrada, exatamente uma aresta com o símbolo saindo desse estado.
- AFND não tem restrições sobre os rótulos de suas arestas, um símbolo pode rotular várias arestas saindo do mesmo estado e a cadeia vazia (ε) é um rótulo possível.

Os AFD e AFND são capazes de reconhecer as mesmas linguagens. Os AFND são mais simples de serem construídos e compreendidos, porém, durante a implementação, ele é convertido para um AFD equivalente para tornar o processo mais eficiente.

A Figura 2 exemplifica como um *token* pode ser especificado através de ER e reconhecido pelo autômato equivalente. A ER apresentada possui operações de união e fechamento transitivo.

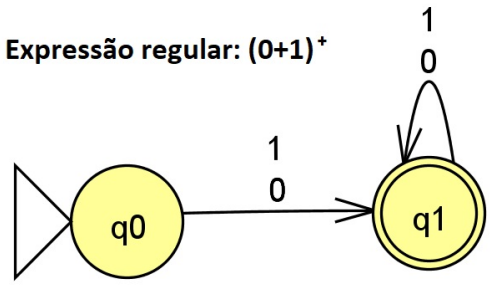

Figura 2. Exemplo de AFD e ER equivalente que valida cadeias binárias 0 e 1

A expressão regular define uma cadeia binária composta por 0 e/ou 1, contendo obrigatoriamente ao menos um dígito. O primeiro + na sentença indica o operador OU, o que possibilita a escolha do dígito 1 ou do dígito 0. Já o segundo + consiste no fecho transitivo, indicando a obrigatoriedade de ao menos um dos símbolos na sentença, possibilitando repeti-los indefinidamente.

Este AFD reconhece qualquer cadeia que inicie com os símbolos 0 ou 1, como demonstra a transição do estado inicial q0 para o estado q1. Após o reconhecimento do primeiro símbolo, pode-se encontrar uma cadeia de 0s e 1s alternados, como define a transição de q1 para ele mesmo, que é o estado final.

Um exemplo de cadeia que pode ser validado pelo autômato é 1011. No estado inicial (q0), o autômato recebe o caractere *1* e, então, passa para o próximo estado (q1, estado final), validando a cadeia de forma recursiva até verificar todos os caracteres de entrada, ou seja, 0 ou 1.

A tabela de símbolos é uma estrutura de dados usada pelo compilador para armazenar identificadores que nomeiam funções, variáveis, tipos de dados e, em algumas situações, palavras reservadas. A diferenciação entre palavras reservadas e identificadores é uma adversidade durante o reconhecimento de caracteres na análise léxica, pois ambos possuem formas léxicas semelhantes. Segundo Aho et al. (2008) pode-se lidar com palavras reservadas de duas formas:

- Instalar inicialmente as palavras reservadas na tabela de símbolos. Um campo da entrada da tabela de símbolos indica que essas cadeias de caracteres não são identificadores comuns e diz qual *token* elas representam. Por esta abordagem, o AF torna-se menor, porém, aumenta o custo em termos de processamento, uma vez que para cada sequência de letras encontrada pelo AF é preciso fazer a verificação na tabela de símbolos para possibilitar a diferenciação.
- Criar diagramas de transição (AF) separados para cada palavra reservada. Nesta abordagem, cada estado do autômato representa uma letra da palavra reservada a

ser reconhecida, cuja transição para o estado final garante que não haja letra ou dígito a mais. Esta estratégia é a mais usada devido ao seu desempenho em termos de processamento, embora gere um AF maior.

A implementação de AF pode ser abordada com diferentes estratégias. Uma abordagem defendida por Aho et al. (2008) consiste na utilização de desvios de múltiplos caminhos (uma cadeia de *ifs*, podendo ser aninhados). Nesta abordagem, uma variável *state* é usada para representar o número do estado corrente no AF. Um comando *switch*, baseado no valor de *state* e no próximo caractere de entrada, encaminha o código para cada um dos possíveis estados, onde é encontrada a ação deste estado.

Outra alternativa se dá através de tabelas de transição. Segundo Cooper e Torczon (2012), este método usa um esqueleto de *scanner* para controle e um conjunto de tabelas geradas que codificam o conhecimento específico da linguagem. O esqueleto do *scanner* é dividido em quatro seções: inicialização, laço de análise, laço de *rollback* (caso o AF ultrapasse o final da palavra reconhecida pelo *scanner*) e uma seção final que interpreta e relata os resultados. O laço de análise lê um caractere e simula a ação de um AF, a partir de duas tabelas que codificam todo o conhecimento sobre o AF. O laço *rollback* usa uma pilha de estados para reverter o *scanner* ao seu estado de aceitação mais recente. O esqueleto do *scanner* usa a variável *state* para manter o estado atual do AF simulado e atualiza *state* usando um processo de pesquisa em tabela em duas etapas. Primeiro classifica o caractere em um conjunto de categorias usando a tabela de classificação, em seguida, usa o estado atual e a categoria de caracteres como índices para a tabela de transição.

Esta tradução em duas etapas de caractere para categoria, depois estado e categoria para novo estado, permite que o *scanner* use uma tabela de transição compactada. A tabela completa eliminaria o mapeamento através da classificação, mas aumentaria o requisito de memória da tabela.

A Figura 3 demonstra como tabelas de transição são utilizadas para codificar o conhecimento do AF. O AF apresentado realiza o reconhecimento de nomenclaturas para registradores (usada para endereçar registradores no processador), tal nomenclatura é iniciada pelo caractere *r*, seguido por um ou mais dígitos de 0 a 9. A tabela de classificação organiza os caracteres de entrada em grupos com objetivo de reduzir o tamanho da tabela de transição, cuja função é representar o conhecimento do AF. Por exemplo, qualquer dígito de 0 a 9 lido será classificado como dígito pela tabela de classificação, desta forma, só é necessária uma coluna para tratar todos os dígitos na tabela de transição. Na tabela de transição, a primeira linha representa as classificações geradas na primeira tabela, a primeira coluna representa os estados do respectivo AF, e por meio do cruzamento entre as duas, é possível representar o comportamento do mesmo. A tabela de tipo de *token* classifica ou não o lexema ao *token*, de acordo com o estado em que a busca parou (válido se terminar em S2 e inválido se terminar em qualquer outro estado).

2.3. Análise Sintática

Segundo Aho et al. (2008), o analisador sintático recebe do analisador léxico uma cadeia de *tokens* que representam o programa fonte, e verifica se essa cadeia de *tokens* pertence a linguagem gerada pela gramática, além de identificar, recuperar e corrigir erros de sintaxe.

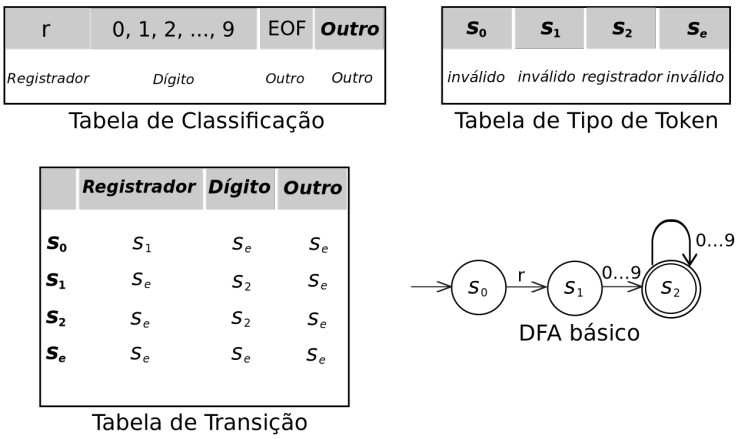

Figura 3. Scanner controlado por tabela para nomes de registrador
Fonte: Cooper e Torczon (2012)

José Neto (2016) indica que sua principal atividade é implementar a recepção de uma sequência de átomos (*tokens*) provenientes do texto fonte, do qual foram extraídos pelo analisador léxico. A partir desta sequência, o analisador sintático efetua uma verificação acerca da ordem de apresentação dos átomos na sequência, 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) são um conjunto de regras que descrevem como formar sentenças da linguagem, possibilitando seu desenvolvimento de forma iterativa através da inclusão de novas produções, além de facilitar a tradução do código e a localização de erros.

Menezes (2011) descreve uma gramática como uma quádrupla ordenada **G=(V,T,P,S)**, na qual:

- V é um conjunto finito de símbolos variáveis ou não-terminais. Não terminais são variáveis sintáticas que representam um conjunto de cadeias.
- T é um conjunto finito de símbolos terminais disjunto de V. Terminais são símbolos elementares da linguagem definida pela gramática, no caso de um compilador são os *tokens* gerados pelo analisador léxico.
- P é um conjunto finito de pares, denominado relação de produções ou produções. Cada par é denominado regra de produção ou produção. As produções de uma gramática especificam a forma que os terminais e não-terminais podem ser combinados para formar cadeias. Cada produção consiste em um não-terminal, chamado de cabeça ou lado esquerdo da produção, uma seta e uma sequência de terminais e/ou não-terminais, chamados de corpo ou lado direito da produção.
- S é um elemento distinguido de V denominado símbolo inicial ou variável inicial, cujo conjunto de cadeias que ele representa é a linguagem gerada pela gramática.

Tudo que pode ser descrito por uma ER pode ser descrito por uma GLC, porém, uma ER não pode descrever uma GLC. As ER permitem a implementação de analisadores léxicos mais eficientes por meio de regras mais simples, e são mais adequadas para descrever estruturas de construções como identificadores, constantes, palavras reservadas e espaços em branco. As GLC, por outro lado, são utilizadas para descrever estruturas

aninhadas, que não podem ser descritas por ER, tal como o balanceamento de símbolos como parênteses, chaves e pares de comando begin-end.

A descrição das regras de produção de uma GLC pode ser feita através da Backus-Naur Form (BNF), uma metassintaxe que permite representar de forma concisa a quádrupla que define uma gramática. A Figura 4 apresenta um exemplo de BNF que valida operações aritméticas de soma, multiplicação de identificadores que usam parênteses para alterar a ordem de precedência e o operador - para indicar operandos negativos.

$$
\begin{aligned}
E \rightarrow\ & E + E \\
\mid\ & E * E \\
\mid\ & -E \\
\mid\ & (E) \\
\mid\ & \mathrm{id}
\end{aligned}
$$

Figura 4. Exemplo de BNF para operações aritméticas
Fonte: Aho et al. (2008)

Árvore de derivação é uma representação gráfica que demonstra a ordem na qual as produções são aplicadas para substituir os não-terminais. Elas tem por objetivo determinar se um fluxo de palavras, ou *tokens* no caso de um compilador, se encaixa na sintaxe da linguagem. Aho et al. (2008) definem este processo como a substituição de um não-terminal pelo corpo de uma de suas produções à partir do símbolo inicial. A Figura 5 apresenta a construção teórica da arvore gerada para realizar as derivações da GLC apresentada na imagem anterior para validar a sentença -(a + b).

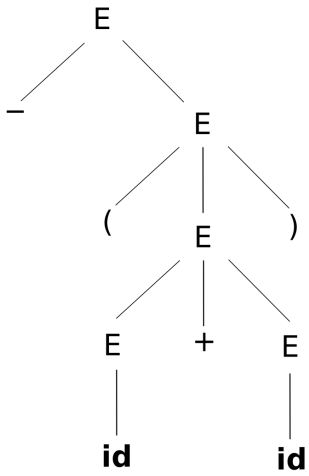

Figura 5. Representação gráfica da árvore gerada à partir da regra de produção apresentada na Figura 4

Fonte: Aho et al. (2008)

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

- Análise descendente (*top-down*): conforme Cooper e Torczon (2012), 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*): de acordo com Aho et al. (2008), realiza o sentido inverso, começa nas folhas (tokens) e avança até a raiz da árvore.

Em ambas estratégias, a entrada do analisador sintático é consumida da esquerda para direita, um símbolo de cada vez.

Em alguns casos o processo de derivação pode gerar duas árvores diferentes a partir de uma sentença, isto consiste em um problema denominado *ambiguidade*. Segundo Aho et al. (2008), para a implementação dos analisadores sintáticos a gramática não pode possuir ambiguidade, do contrário, não é possível determinar univocamente qual árvore de derivação selecionar para uma dada sentença. Para a criação de um analisador ascendente, apenas a eliminação da ambiguidade é necessária. Entretanto, a implementação de um analisador descendente preditivo requer a remoção da ambiguidade, fatoração a esquerda e remoção da recursão a esquerda.

Segundo Aho et al. (2008), a recursão à esquerda surge quando o símbolo mais à esquerda do lado direito da produção é igual ao não-terminal do lado esquerdo da produção. Os métodos de análise descendentes não podem tratar gramáticas com recursão à esquerda de modo que uma transformação é necessária para eliminar esta característica da gramática. Cooper e Torczon (2012) afirmam que com recursão à esquerda um analisador descendente pode aplicar indefinidamente as regras sem gerar um símbolo terminal no início que ele possa corresponder (e avançar a entrada).

Quando uma derivação possui duas alternativas para um não-terminal com um prefixo comum é necessário realizar a *fatoração a esquerda*. Conforme Aho et al. (2008), a fatoração à esquerda é uma transformação da gramática que possibilita a criação de um reconhecedor sintático descendente preditivo. Quando a escolha entre duas ou mais alternativas das produções de um mesmo não terminal começam com a mesma forma sentencial, é possível reescrevê-las para adiar a decisão até que se tenha lido uma cadeia da entrada longa o suficiente para tomar a decisão correta.

As análises ascendente e descendente preditiva podem ser implementadas através de GLC classificadas como LR e LL.

- LR: utilizada na análise ascendente, lê entrada da esquerda para a direita e a derivação é feita mais à direita da árvore.
- LL: utilizada na análise descendente preditiva, lê a entrada da esquerda para a direita e a derivação é feita mais a esquerda da árvore.

2.4. Analisadores Descendentes

Analisadores descendentes podem ser implementados seguindo estratégias *preditivas* ou de *descida recursiva*. Como explicado anteriormente, analisadores descendentes iniciam na raiz e estendem a árvore sistematicamente para baixo, realizando derivações a partir

das regras de produção. Neste contexto, o problema principal é determinar uma produção a ser aplicada para um não-terminal. Aho et al. (2008) afirmam que a análise sintática de descida recursiva pode exigir o processo de retrocesso para encontrar a produção correta a ser aplicada, ou seja, retroceder no reconhecimento, fazendo repetidas leituras sobre a entrada. Além disto, Cooper e Torczon (2012) indicam que o retrocesso aumenta o custo assintótico da análise sintática; na prática, este é um modo dispendioso de descobrir erros de sintaxe. A análise descendente preditiva elimina a necessidade do retrocesso, escolhendo a produção correta a partir da análise do próximo símbolo de entrada, porém sua gramática requer maiores especifidades.

A gramática necessária para implementação do analisador descendente preditivo exige eliminação de ambiguidade, inexistência de recursividade à esquerda além de fatoração à esquerda. Para esta abordagem é utilizada a classe de gramáticas LL(1). De acordo com Aho et al. (2008), o primeiro *L* significa que a cadeia de entrada é analisada da esquerda para a direita (L = *Left-to-right*), o segundo *L* representa uma derivação mais à esquerda (L = *Letfmost*); e o *1* pelo uso de um símbolo à frente na entrada para tomar as decisões quanto à ação de análise, possibilitando a eliminação do retrocesso.

O analisador sintático descendente preditivo pode ser implementado com o auxílio de um buffer de entrada que contém a cadeia a ser reconhecida e uma pilha contendo o símbolo inicial da gramática. Este símbolo é derivado ao longo da execução através de um algoritmo que cruza as informações da pilha com a gramática, consultando a tabela M, para realizar as derivações com a finalidade de validar ou recusar a cadeia. Este cenário é demonstrado na Figura 6.

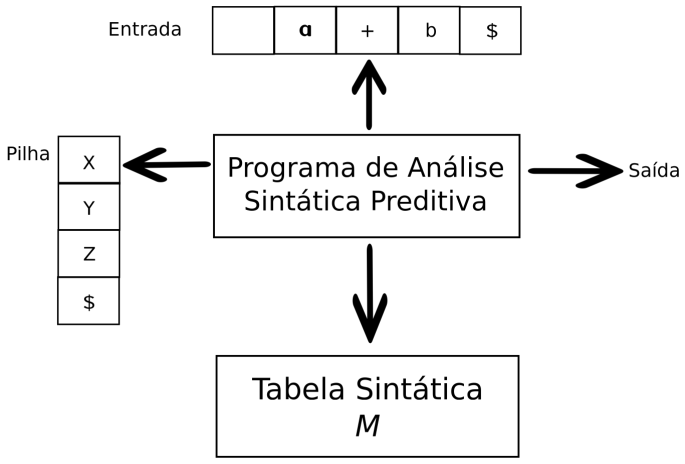

Figura 6. Modelo de um analisador preditivo dirigido por tabela
Fonte: Aho et al. (2008)

O processo de construção da *tabela M* para a implementação do analisador descendente preditivo a partir da gramática LL(1) requer a criação dos conjuntos *FIRST* e *FOLLOW*. Estes conjuntos são regidos por regras formais sobre os símbolos terminais e não-terminais fornecidos a partir da gramática. Para cada símbolo não-terminal da gramática são gerados os respectivos conjuntos.

O conjunto FIRST contém os primeiros símbolos terminais após a derivação de um não-terminal. Aho et al. (2008) apontam que para calcular o FIRST(X) de todos os

símbolos X da gramática, as seguintes regras devem ser aplicadas até que não haja mais terminais ou ε que possam ser acrescentados a algum dos conjuntos FIRST.

1. Se X é um símbolo terminal, então FIRST(X) = X.
2. Se X é um símbolo não-terminal e X $\rightarrow$ $Y_1, Y_2, ..., Y_k$ é uma produção para algum $k \geq 1$, então acrescente α a FIRST(X) se, para algum *i*, α estiver em FIRST(Y_i), e ϵ estiver em todos os FIRST(Y_1), ..., FIRST(Y_{i-1}); ou seja, $Y_1 ... Y_{i-1} \Rightarrow \epsilon$. Se ϵ está em FIRST(Y_j), para todo *j* = 1, 2,..., k, então adicione ϵ a FIRST(X).
3. Se X $\rightarrow$ ϵ é uma produção, então acrescente ϵ a FIRST(X).

O conjunto FOLLOW determina os símbolos terminais sucessores de um não-terminal do lado direito da produção. De acordo com Aho et al. (2008), para realizar o cálculo do FOLLOW(A) para todos os não-terminais A, aplica-se as seguintes regras até que nada mais possa ser acrescentado a nenhum dos conjuntos FOLLOW.

- Coloca-se $ em FOLLOW(S), onde S é o símbolo inicial da gramática, e $ é o marcador de fim da entrada ou fim de arquivo.
- Se houver uma produção A $\rightarrow$ αBβ, então tudo em FIRST(β) exceto ϵ está em FOLLOW(B).
- Se houver uma produção A $\rightarrow$ αB, ou uma produção A $\rightarrow$ αBβ, onde o FIRST(β) contém ϵ, então inclui-se o FOLLOW(A) em FOLLOW(B).

A Figura 7 apresenta uma gramática da classe LL(1) responsável por reconhecer expressões aritméticas de soma e multiplicação, seguida por seus conjuntos FIRST e FOLLOW gerados seguindo as regras apresentadas.

Gramática	**Conjunto FIRST**	**Conjunto FOLLOW**
E → T E'	FIRST (F) = FIRST (T) = FIRST (E) = {(, **id**}	FOLLOW (E) = FOLLOW (E') = {), **$**}
E'→ + T E' | ε		
T → F T'	FIRST (E') = {+, ε }	FOLLOW (T) = FOLLOW (T') = {+,), **$**}
T'→ * F T' | ε		
F → (E) | **id**	FIRST (T') = { *, ε }	FOLLOW (F) = {+, *,), **$**}

Figura 7. Gramática e conjuntos FIRST e FOLLOW
Fonte: Adaptado de Aho et al. (2008)

A tabela M é montada a partir dos conjuntos FIRST e FOLLOW e é utilizada pelo analisador para auxiliar na escolha da produção correta com base no próximo símbolo da entrada e no símbolo não terminal no topo da pilha. Conforme Aho et al. (2008), a tabela de reconhecimento sintático preditivo M[A,α] é um arranjo bidimensional, onde A é um não-terminal e α é um terminal ou símbolo $, o marcador de fim de entrada. Para cada produção A $\rightarrow$ α da gramática, é necessário:

- Para cada terminal α em FIRST(A), inclui-se A $\rightarrow$ α em M[A,α].
- Se ϵ pertence a FIRST(α), inclui-se A $\rightarrow$ α em M[A,*b*] para cada terminal *b* em FOLLOW(A). Se ϵ pertence a FIRST(α) e $ pertence a FOLLOW(A), acrescenta-se também A $\rightarrow$ α em M[A,$].

Caso não haja produção alguma em M[A,α], então define-se M[A,α] como **error** (que normalmente é representado por uma entrada vazia na tabela).

A Figura 8 demonstra a tabela M criada a partir dos conjuntos FIRST e FOLLOW apresentados na Figura 7.

Tabela *M*

NÃO-TERMINAL	SÍMBOLO DE ENTRADA					
	id	+	*	(	)	**$**
E	E → T E'			E → T E'		
E'		E' → +TE'			E' → ε	E' → ε
T	T → F T'			T → F T'		
T'		T' → ε	T' → *FT'		T' → ε	T' → ε
F	F → **id**			F → (E)		

Figura 8. Tabela M
Fonte: Adaptado de Aho et al. (2008)

2.5. 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 (Alkmim e de Mello, 2012).

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 (ADA) 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. (2005), 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 no artigo de Barbosa et al. (2019).

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 (Mernik e Zumer, 2003).

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

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 DFA. Este DFA gerado é convertido em um Scanner Java. Nesta etapa é apresentado uma tela específica para visualizar o DFA e animações são utilizadas para demonstrar a validação dos *tokens*.

A análise sintática utiliza GLC representadas com a notação 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.

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 (Resler e Deaver, 1998).

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 *debugger* e *breakpoints* 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.

O Quadro 1 apresenta de forma resumida os principais pontos para a comparação das ferramentas citadas com a proposta. A coluna *Etapas contempladas* presente no Quadro 1 cita os termos Lex, Sint e Sem que representam, respectivamente, análise léxica, análise sintática e análise semântica.

As ferramentas citadas atendem ao objetivo comum de auxiliar no processo de ensino/aprendizagem de compiladores, cada qual com suas especificidades e focos apresentados. 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 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.

Quadro 1. Resumo das características das ferramentas similares

	Etapas Contempladas	Interface Gráfica	Explicação Teórica	Passo a Passo das Etapas	Gera Analisador
LexSint	Lex Sint	Não	Não	Resumido	Não
Verto	Todas	Desktop	Não	Detalhado	Não
Flex	Lex	Não	Não	Não	C
JFlex	Lex	Desktop	Não	Não	Java
GALS	Lex Sint	Desktop	Não	Não	C++ Delphi Java
LISA	Lex Sint Sem	Desktop	Não	Resumido	Java
VCOCO	Lex Sint	Desktop	Não	Resumido	Qualquer linguagem especificada em EBNF
Sistema Proposto	Lex Sint	Web	Sim	Detalhado	Não

Por fim, uma funcionalidade exclusiva da ferramenta, tendo como base as ferramentas similares apresentadas, consiste em sua explicação teórica das fases abordadas.

3. Metodologia

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 de Freitas (2013), 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 de Freitas (2013), 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.

[4]Sociedade Brasileira de Computação

- Benefits compiler learning.
- Compiler teaching tool.

Conforme Gil (1991), 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 (1991), 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 consiste 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 do sistema. A aplicação desenvolvida é web, responsiva e interativa, de modo a 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.

Para a avaliação do projeto, serão executados testes durante as aulas ofertadas pela disciplina de compiladores no curso de Ciência da Computação do Instituto Federal de Santa Catarina (IFSC) - Campus Lages. Estes dados serão utilizados como *feedback* para reafirmar as demandas elencadas descritas na introdução e aprimorar o software.

4. Desenvolvimento do sistema

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 9 apresenta a tela inicial do sistema, contendo informações e explicações gerais sobre o projeto.

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 do usuário inserir o próprio código e operar o sistema da forma desejada para acompanhar como este código é analisado. O resultado é expresso por meio de representações visuais com destaque para cada passo do 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.

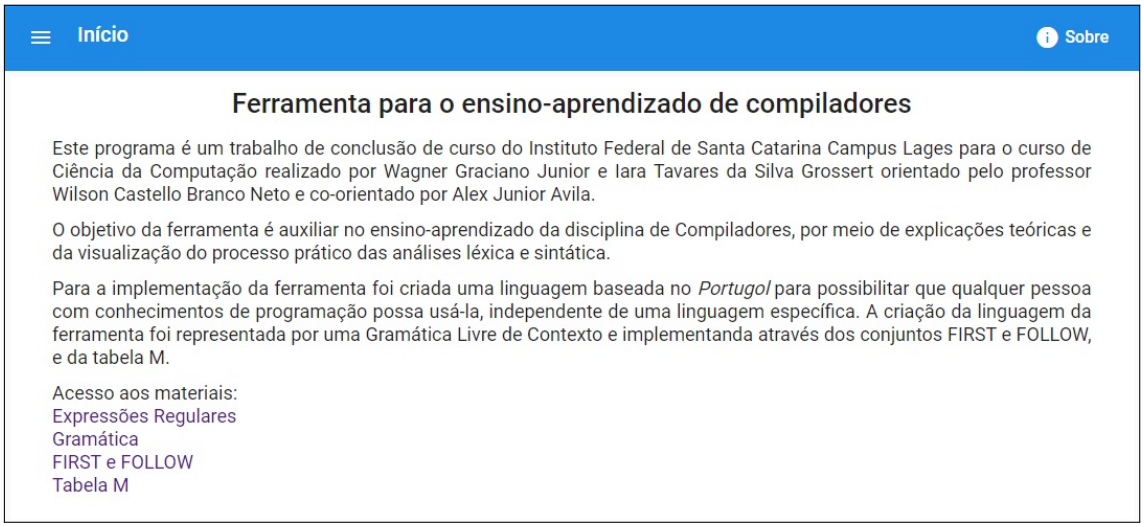

Figura 9. Página inicial

A Figura 10 apresenta o menu lateral expansível e vertical com os caminhos para as principais funcionalidades citadas. Os tópicos de análise léxica e sintática são expansíveis horizontalmente para apresentar os subtópicos *Explicações* e *Exemplos*, referentes à explicação teórica do conteúdo e funcionamento prático com o passo a passo da respectiva etapa.

Figura 10. Menu lateral com os assuntos abordados no sistema

4.1. 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 11.

A aba objetivos apresenta o papel da análise léxica, o que ela recebe como entrada e gera com saída. A entrada de dados contém explicação sobre a leitura do código fonte

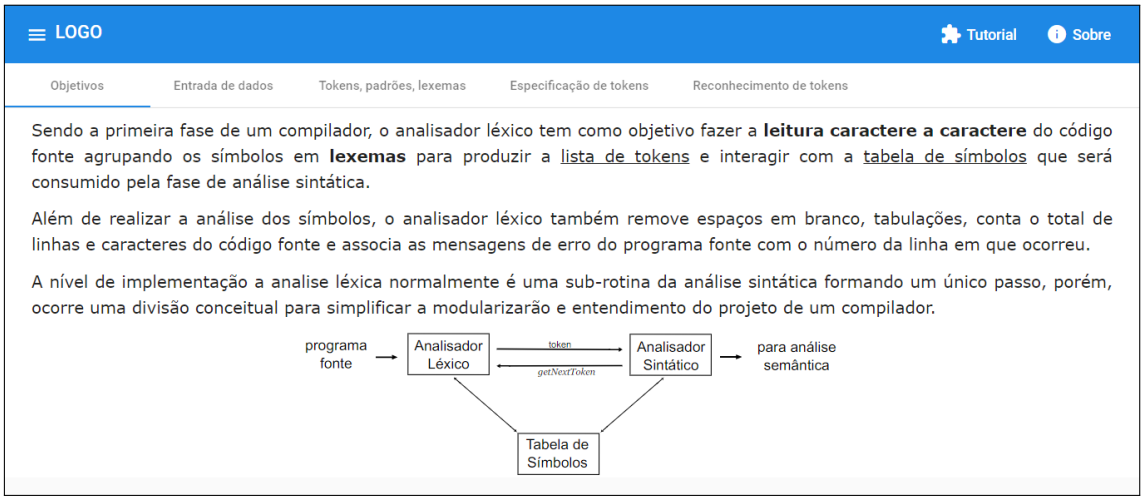

Figura 11. Menu horizontal com explicação teórica da análise léxica

através de buffers duplos com especificações e imagens ilustrativas do processo. *Tokens*, padrões e lexemas possui as definições destes termos e explica o relacionamento entre eles. A especificação de *tokens* cita que *tokens* são representados por meio de ER, define cadeia, alfabeto e linguagem. Por fim, o reconhecimento de *tokens* define AFD e 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. A Figura 12 demonstra a apresentação da janela de diálogo após o clique do usuário no termo *buffering*.

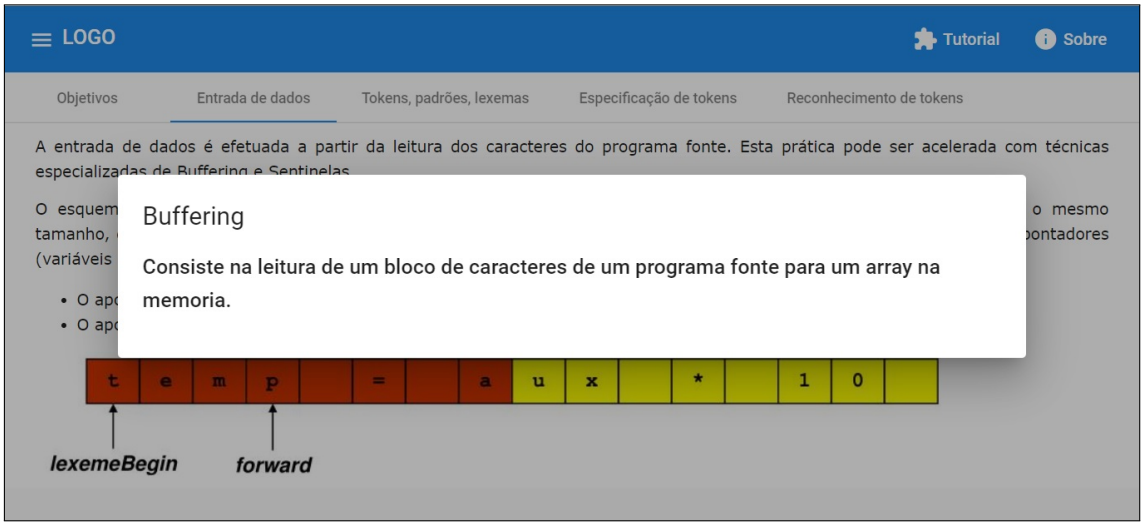

Figura 12. Caixa de diálogo contendo explicação sobre buffering

A Figura 13 demonstra o campo editável que permite ao usuário digitar o código fonte a ser verificado pelo analisador.

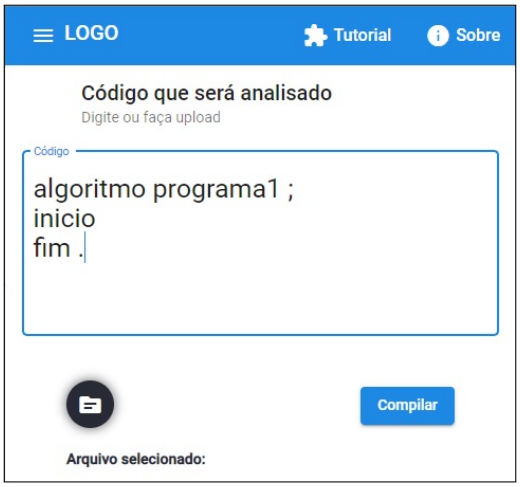

Figura 13. Código digitado pelo usuário

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, a linguagem criada para este trabalho foi baseada no Portugol (português estruturado)[5].

O Quadro 2, apresenta as ER utilizadas para permitir a implementação da primeira fase do compilador e contém todos os símbolos aceitos na linguagem.

Quadro 2. 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	// .*
comentário_bloco	/[*] .* [*]/

A Figura 14 expõe o respectivo autômato para todas ER apresentadas no Quadro 2 que pode ser controlado pelo usuário na interface prática do programa.

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

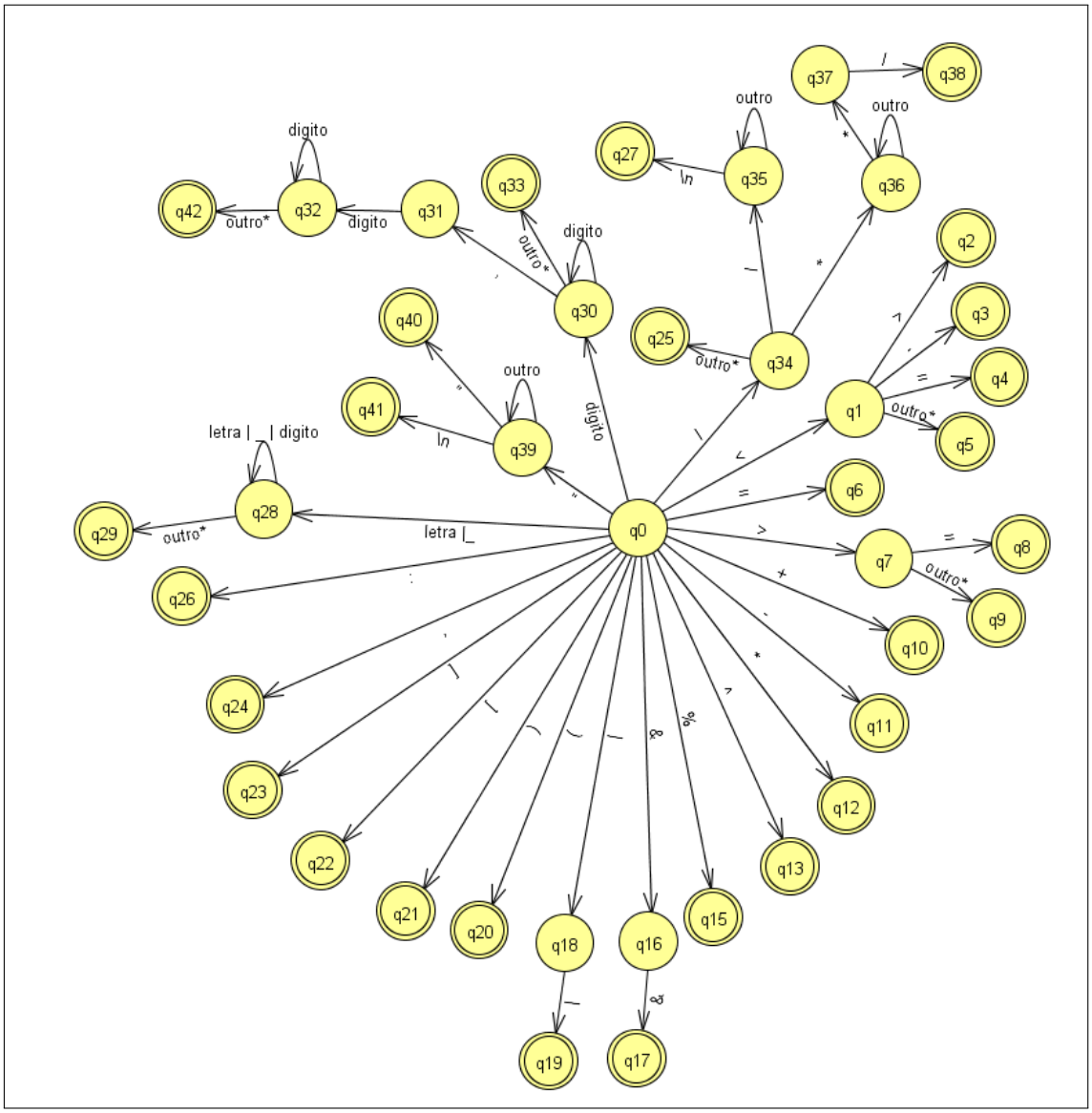

Figura 14. Autômato finito determinístico

A Figura 15 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, 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 é explicada detalhadamente nas figuras 16, 17 e 18, respectivamente.

A lista de tokens, apresentada na Figura 16, 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 13, 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

Figura 15. Funcionamento da análise léxica

chave *palavraReservada*.

Figura 16. Lista de tokens do código digitado pelo usuário

Na Figura 17 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.

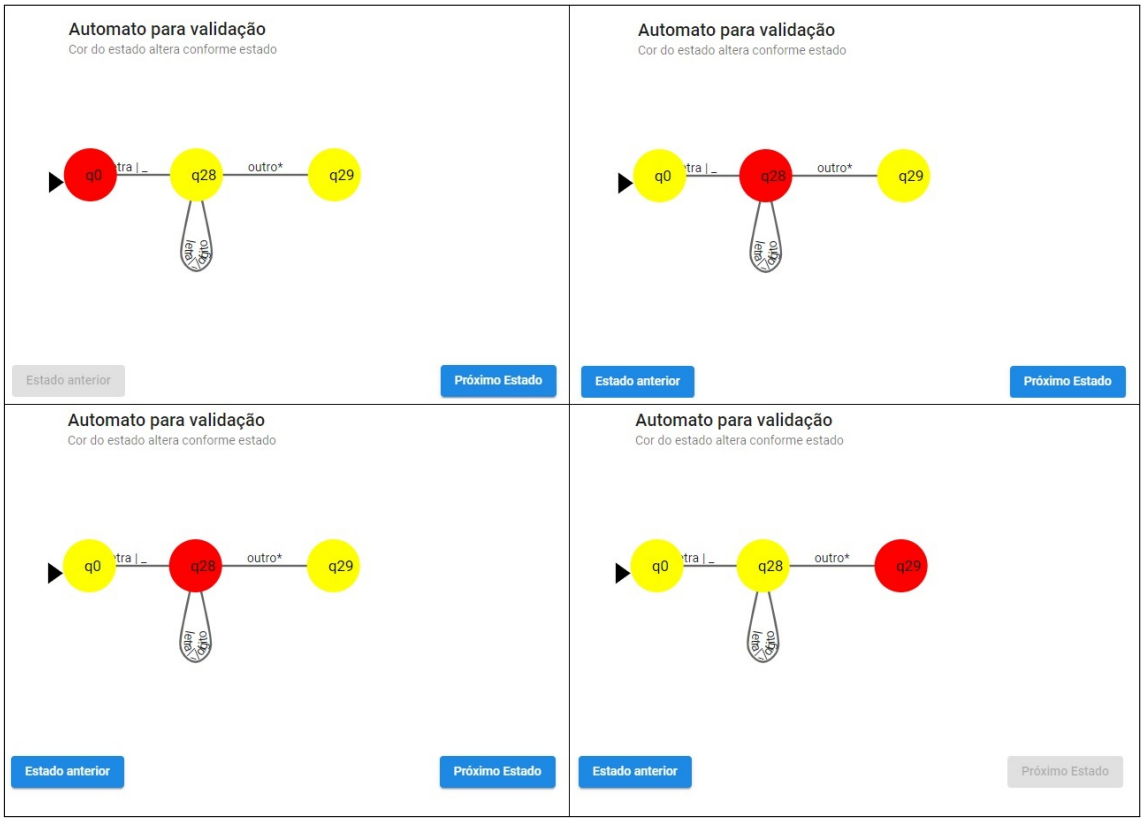

Figura 17. Autômato da tela prática da análise sintática

A Figura 18 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*.

Figura 18. Explicação teórica dos passos do analisador léxico

4.2. 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, conforme indica a Figura 19.

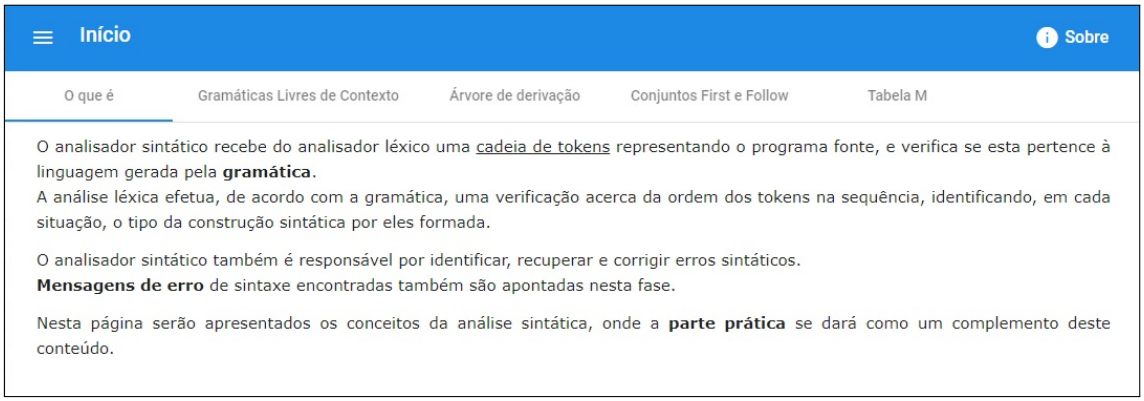

Figura 19. Menu horizontal com explicação teórica da análise sintática

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. *Conjuntos First e Follow* especifica os termos e apresenta exemplos por meio de imagens. Por fim, a tabela M apresenta como esta é gerada a partir dos conjuntos First e Follow.

Para a implementação do analisador sintático preditivo, cuja função é validar a sintaxe do código-fonte, foi construída uma Gramática Livre de Contexto para representar as construções válidas da linguagem.

Algumas produções desta gramática são apresentadas no Quadro 3 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.

Quadro 3. 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

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

Quadro 4. Conjunto FIRST

FIRST (PROGRAMA) = {algoritmo}
FIRST (BLOCO) = {const, var, void, vetor, string, caractere, real, inteiro, logico, inicio}
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, inteiro, logico}

O conjunto FOLLOW, conforme mostra o Quadro 5, 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) de acordo com a primeira regra apresentada na seção 2.4. 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 6, contém todos os símbolos terminais e a primeira coluna os não terminais. Como 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

Quadro 5. Conjunto FOLLOW construído à partir das gramáticas

FOLLOW (PROGRAMA) = {$}
FOLLOW (BLOCO) = {.}
FOLLOW (SEÇÃO_CONSTANTES) = {var, void, vetor, string, caractere, real, inteiro,logico, inicio}
FOLLOW (LISTA_CONSTANTES) = {var, void, vetor, string, caractere, real, inteiro,logico, inicio}
FOLLOW (NOVA_CONSTANTE) = {var, void, vetor, string, caractere, real, inteiro, logico, inicio}
FOLLOW (DEFINIÇÃO_CONSTANTE) = {;}
FOLLOW (SEÇÃO_VARIÁVEIS) = {void, vetor, string, caractere, real, inteiro, logico, inicio}
FOLLOW (LISTA_VARIÁVEIS) = {void, vetor, string, caractere, real, inteiro, logico , inicio}
FOLLOW (NOVA_VARIÁVEL) = {void, vetor, string, caractere, real, inteiro, logico, inicio}
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, inteiro, logico}

tabela M não apresenta nenhuma produção.

Quadro 6. 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

A Figura 20 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 21, 22 e 23.

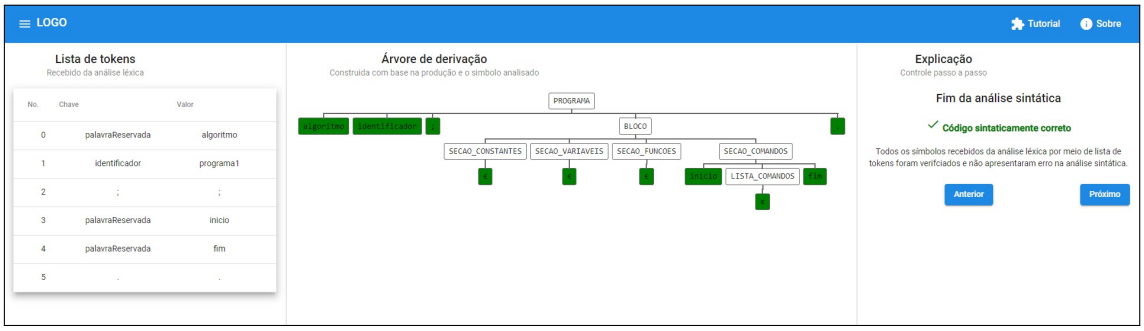

Figura 20. Funcionamento análise sintática

A lista de tokens, apresentada na Figura 21 possui os tokens associados a partir do código digitado pelo usuário na análise léxica, como explicado na seção 4.1.

Figura 21. Lista de tokens do código digitado pelo usuário

A árvore de derivação, apresentada na Figura 22, 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 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.

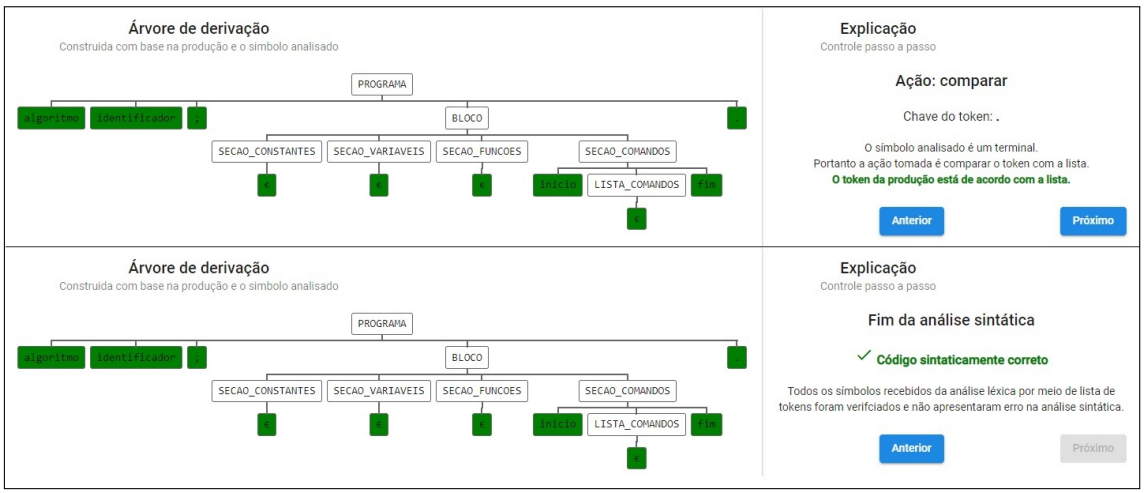

Figura 22. Dois últimos estados da árvore sintática validando o código fonte

A figura 23 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*.

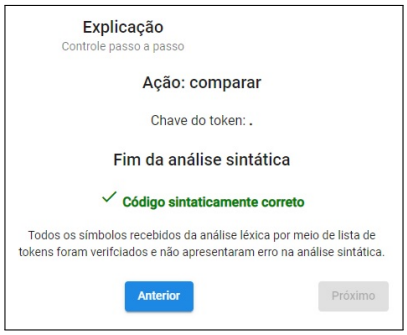

Figura 23. Explicação teórica dos passos do analisador sintático

5. Conclusão

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 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.

Para trabalhos futuros, será realizada a avaliação e disponibilização da ferramenta no curso de Ciência da Computação no Instituto Federal de Santa Catarina Campus Lages na disciplina de compiladores, os professores e alunos utilizarão a ferramenta durantes as aulas, e após esta etapa, será confeccionado um formulário com o intuito de receber o feedback dos participantes para efetuar possíveis melhorias.

A ferramenta poderá ser adaptada para possibilitar o usuário inserir suas próprias expressões regulares e gramáticas livres de contexto, para permitir ao programa gerar os respectivos autômatos, conjuntos FIRST, FOLLOW e tabela M. Desta forma, as explicações passo a passo se darão conforme a linguagem especificada pelo usuário. Esta abordagem também possibilitará a visualização do processo de geração dos autômatos à partir das expressões regulares e dos conjuntos FIRST e FOLLOW através da gramática livre de contexto para gerar a tabela M.

Referências

Aho, A. V., Sethi, R., e Ullman, J. D. (1986). *Compiladores Princípios, Técnicas e Ferramentas*. LTC, 1st edition.

Aho, A. V., Sethi, R., e Ullman, J. D. (2008). *Compiladores Princípios, Técnicas e Ferramentas*. LTC, 2nd edition.

Alkmim, G. e de Mello, B. (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).

Barbosa, S., Bonidia, R., e Coelho Neto, J. (2019). Flex, JFlex e GALS: Ferramentas de Apoio ao Ensino de Compiladores, doi = 10.5753/wei.2019.6628.

Cooper, K. D. e Torczon, L. (2012). *Engineering a Compiler*. Elsevier, 2nd edition.

Gil, A. C. (1991). *Como Elaborar Projetos de Pesquisa*. ATLAS S.A., 3rd edition.

José Neto, J. (2016). Introdução à compilação. In Ltc, editor, *Introdução à compilação*, page 222. Rio de Janeiro.

Menezes, P. B. (2011). *Linguagens Formais e Autômatos*. Bookman, 6th edition.

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

Pereira, F. M. Q. (2020). Pesquisa em compiladores. https://homepages.dcc.ufmg.br/ fernando/projects/CompilerResearchUFMG.pdf. Março 25, 2020.

Prodanov, C. C. e de Freitas, E. C. (2013). *Metodologia do Trabalho Científico: Métodos e Técnicas da Pesquisa e do Trabalho Acadêmico*. Universidade Feevale, 2nd edition.

Resler, R. D. e Deaver, D. M. (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*, ITiCSE '98, page 199–202, New York, NY, USA. Association for Computing Machinery.

Scheider, C., Passerino, L., e Oliveira, R. (2005). Compilador educativo verto: ambiente para aprendizagem de compiladores. *RENOTE*, 3.

Singh, R., Sharma, V., e Varshney, M. (2009). *Design and Implementation of Compiler*. New Age International, 1st edition.

White, E., Sen, R., e Stewart, N. (2005). Hide and show: using real compiler code for teaching. volume 37, pages 12–16.