UNIVERSIDADE FEDERAL DO CEARÁ

CAMPUS QUIXADÁ

CURSO DE GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

FRANCISCO FAGNER FERREIRA MESQUITA

VANSI: UMA FERRAMENTA PARA VISUALIZAÇÃO E ENSINO DE ALGORITMOS DE ANÁLISE SINTÁTICA

QUIXADÁ

2024

FRANCISCO FAGNER FERREIRA MESQUITA

VANSI: UMA FERRAMENTA PARA VISUALIZAÇÃO E ENSINO DE ALGORITMOS DE ANÁLISE SINTÁTICA

Projeto de Pesquisa apresentado ao Curso de Graduação em Ciência da computação do Campus Quixadá da Universidade Federal do Ceará, como requisito parcial à obtenção do grau de bacharel em Ciência da computação.

Orientador: Prof. Dr. João Marcelo Uchôa de Alencar.

QUIXADÁ

2024

```
A B C D E F G
K D R G V N A
G Q O L P E A
C V T J K L Z
K D R G V N A
G Q O L P E A
K D R G V N A
```

(Fagner Ferreira)

SUMÁRIO

1 INTRODUÇÃO

Um compilador é uma ferramenta usada para compilar código-fonte de uma linguagem de alto nível para código de máquina. Esse processo é feito em várias fases. As três primeiras fases do processo de compilação podem ser definidas como análise léxica, análise sintática e análise semântica. Elas são chamadas coletivamente de *front-end* do compilador (Mogensen, 2024). A fase de análise léxica gera *tokens* usados pela fase de análise sintática para validar que a entrada segue a estrutura da gramática da linguagem alvo e gerar uma árvore sintática para ser usada pelas próximas fases da compilação (Thain, 2020).

O programa que realiza a análise sintática é chamado analisador sintático ou *parser*. Os *parsers* podem ser classificados em dois tipos. O primeiro tipo é o *bottom-up* (ou ascendente) que funciona reduzindo os *tokens* às produções da gramática. O segundo tipo é o *top-down* (ou descendente) que segue o caminho oposto do *parser bottom-up* tentando encontrar produções correspondentes à estrutura da *string* de entrada comparando-a com as produções da gramática (Cooper; Torczon, 2022).

A disciplina de compiladores está presente em muitas grades curriculares de cursos de ciência da computação pelo mundo. Dentro dessa disciplina são ensinados vários algoritmos de análise sintática. No entanto, aprender o funcionamento desses algoritmos é uma tarefa difícil para os alunos, assim como também é difícil para os professores ensinarem esse assunto (Sangal; Kataria; Tyagi *et al.*, 2018).

Tendo em mente a dificuldade no ensino sobre compiladores, foram criadas ferramentas de visualização de algoritmos(ou *AV* do inglês *Algorithm Visualization*). Essas ferramentas procuram ajudar no ensino de algoritmos que fazem parte das fases da compilação e, ao serem usadas com alunos, tiveram uma resposta positiva. Elas podem auxiliar na compreensão do conteúdo não só por meio da visualização do funcionamento dos algoritmos, mas também pelas respostas instantâneas, que os estudantes podem usar para validar resultados que são difíceis de construir manualmente. (Jain; Goyal; Chakraborty, 2017).

O interesse pelo uso de *AV*'s cresceu nos últimos anos. Com disso, também houve o surgimento de tecnologias de desenvolvimento *web* modernas e os avanços na capacidade e qualidade dos gráficos de *browsers*. Esses avanços permitiram um desenvolvimento *web* mais fácil e a criação de várias aplicações baseadas inteiramente na *web* (Romanowska *et al.*, 2018). Dentre os avanços no desenvolvimento *web* está a criação de *frameworks web* que reutilizam componentes e implementam funcionalidades para acelerar a criação de aplicações *web* (Uppal;

Srivastava; Saini, 2022). Um exemplo de *framework web* é o *Svelte*[1] que foi utilizado para o desenvolvimento da ferramenta apresentada nesse trabalho.

Embora esse tema tenha já tenha sido abordado em trabalho similares como o proposto por Sangal, Kataria e Tyagi (2017), ainda há avanços que podem ser alcançados como será discutido na seção de trabalhos relacionados. Com isso, esse trabalho apresenta o desenvolvimento da ferramenta de visualização de analisadores sintáticos VANSI[2].

1.1 Objetivos

O objetivo desse trabalho é desenvolver uma ferramenta que auxilie na compreensão do funcionamento da análise sintática e quais os processos necessários para obter a saída de cada passo de seus algoritmos. Esse trabalho tem os seguintes objetivos específicos:

- Desenvolver uma forma de visualização para os algoritmos CLR, SLR e LL(1).
- Validar a ferramenta fazendo uma avaliação com alunos.

[1] https://svelte.dev/
[2] https://vansi.netlify.app

2 FUNDAMENTAÇÃO TEÓRICA

Neste capítulo apresentaremos os conceitos centrais que serviram como base e guia para a elaboração deste trabalho. Ao início é falado sobre o conhecimento básico sobre compiladores, depois sobre o *framework* utilizado para construção da ferramenta.

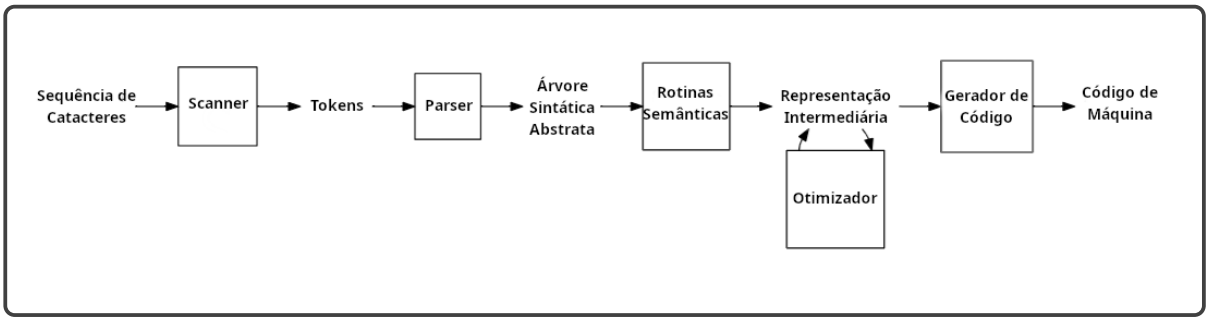
2.1 Compiladores

Programas que são executados em computadores são escritos no que é chamada de linguagem de máquina que usa comandos simples que são interpretados pela máquina. Escrever em linguagem de máquina é uma tarefa passível de erro e cansativa. Por essa razão foram criados os compiladores. Os compiladores traduzem linguagens de alto nível em linguagem de máquina e indicam erros cometidos pelos programadores no código-fonte (Mogensen, 2024).

2.1.1 Fases do compilador

As fases de um compilador podem ser divididas de várias formas. Os trabalhos de Cooper e Torczon (2022), Mogensen (2024) e Thain (2020) dão suas definições das fases de um compilador, mas para esse trabalho será seguida a definição de Thain (2020).

Figura 1 – Fases de um compilador UNIX



Fonte: adaptada de Thain (2020).

2.1.1.1 Análise Léxica

Na fase de análise léxica, o *scanner*, também chamado de *tokenizer*, consome texto simples de um programa e agrupa os caracteres individuais em sequências chamadas de *tokens*. Esse processo funciona de forma parecida com agrupar letras para formar palavras da lingua-

gem natural, esse agrupamento é feito usando expressões regulares implementadas através de autômatos.

2.1.1.2 Análise Sintática

Análise de sintática será o foco do trabalho sendo discutida de forma mais aprofundada nas próximas seções. A fase de análise sintática da compilação rearranja os *tokens* gerados pela fase de análise léxica, gerando assim uma estrutura chamada árvore sintática. Árvores sintáticas são estruturas de árvore como diz o nome, as folhas dessa árvore são os *tokens* e a leitura em ordem da árvore dá a sequência de *tokens* do texto de entrada dado ao analisador sintático. Ao construir a árvore sintática, o analisador sintático também checa se há erros de sintaxe no texto de entrada.

2.1.1.3 Análise semântica

Na fase de análise semântica, as rotinas semânticas percorrem a árvore sintática e buscam significado na entrada a partir das regras da gramática e da relação entre os elementos da entrada. Depois das rotinas semânticas, a árvore de análise sintática é convertida em uma representação intermediária que é uma versão simplificada de *assembly* que permite uma análise detalhada.

2.1.1.4 Otimização

Na fase de otimização, otimizadores são aplicados na representação intermediária para tornar o programa mais rápido, menor e eficiente. Normalmente os otimizadores recebem uma entrada em formato de representação intermediária e retornam um resultado no mesmo formato para que todos os otimizadores possam ser aplicados de forma independente e em qualquer ordem.

2.1.1.5 Geração de código

Na fase de geração de código, o gerador de código consome a representação intermediária otimizada e a transforma em um programa em *assembly* concreto. Para otimizar o uso dos registradores físicos limitados e gerar instruções de montagem de maneira eficiente, o gerador de código precisa executar as tarefas de alocação de registradores, seleção de instruções e sequenciamento de instruções.

2.2 Analisadores Sintáticos Descendentes

Cooper e Torczon (2022), Mogensen (2024) e Thain (2020) falam sobre analisadores sintáticos descendentes, mas para esse trabalhos será seguida a definição de Thain (2020). Analisadores sintáticos descendentes, também chamados *parsers top-down*, são métodos de análise sintática que iniciam a análise a partir do símbolo inicial da gramática. Eles fazem comparações entre os *tokens* do texto de entrada e os símbolos da gramática para encontrar a produção que deve ser escrita no lugar dos símbolos da gramática. Essas comparações são feitas até sobrar apenas símbolos terminais, esses símbolos terminais devem coincidir com a sequência de *tokens* da entrada caso contrário será considerado um erro.

2.2.1 Descendentes Recursivos

O conjunto de gramáticas que pode ser analisados usando algoritmos usando apenas um não terminal e o próximo símbolo da entrada é chamado conjunto de gramáticas LL(1). Uma das formas de fazer a análise dessas gramáticas é usando o analisador sintático descendente recursivo que usa funções recursivas para cada não terminal para processar a entrada. É um algoritmo que funciona como uma forma recursiva dos analisadores sintáticos LL(1) que já são tratados nesse trabalho, além disso, esse algoritmo não segue estruturas fixas e determinísticas, por isso não será abordado na ferramenta.

2.2.2 Analisadores Sintáticos LL(1)

Analisadores sintáticos LL(1) são um tipo de analisador sintático descendente, esses analisadores sintáticos levam em consideração um único *lookahead* que nesse algoritmo é o símbolo inicial do lado direito das produções. O *lookahead* é usado para decidir qual produção deve ser escrita, por essa razão apenas gramaticas não ambíguas podem ser analisadas pelos analisadores sintáticos LL(1).

O conjunto dos símbolos iniciais das produções de uma gramática é chamado conjunto *first*. No algoritmo LL(1) o conjunto *lookahead* corresponde ao conjunto *first*, a construção desse conjunto pode ser feita seguindo o Algoritmo 1. As definições dos algoritmos foram tiradas do trabalho de Thain (2020).

O conjunto *follow* é o conjunto de símbolos terminais da gramática que podem ocorrer depois de qualquer uma das derivações de um não terminal A, o conjunto também inclui

Algoritmo 1: First

Entrada: Gramática G, Símbolo X
Saida: Conjunto *first*
início
  $first \leftarrow \varnothing$
  se X é terminal **então**
    $first \leftarrow \{X\}$
  fim
  senão
    repetir
      para cada regra $X \rightarrow Y_1 Y_2 \ldots Y_k$ em G **fazer**
        se $a \in \text{FIRST}(Y_1) \vee (a \in \text{FIRST}(Y_n) \wedge Y_1 \ldots Y_{n-1} \Rightarrow^* \varepsilon)$ **então**
          $first \leftarrow first \cup \{a\}$
        fim
        se $Y_1 \ldots Y_k \Rightarrow^* \varepsilon$ **então**
          $first \leftarrow first \cup \{\varepsilon\}$
        fim
      fim
    até que não haja mais mudanças
  fim
  retornar *first*
fim

o símbolo \$ usado para se referir ao fim da *string*. Esse conjunto é usado no analisador LL para lidar com produções que derivam uma *string* vazia. A construção do conjunto *follow* pode ser feita seguindo o Algoritmo 2.

Algoritmo 2: Follow

Entrada: Gramática G
Saida: Dicionário *follow*
início
  $follow \leftarrow$ Dicionário$\langle$Símbolo, Conjunto$\rangle$
  $follow[S] \leftarrow \{\$\}$, Onde S é o símbolo inicial
  repetir
    para cada produção $A \rightarrow \alpha$ em G **fazer**
      se $A \rightarrow \alpha B \beta$ **então**
        $follow[B] \leftarrow follow[B] \cup (\text{FIRST}(\beta) - \{\varepsilon\})$
      fim
      se $A \rightarrow \alpha B \vee \varepsilon \in \text{FIRST}(\beta)$ **então**
        $follow[B] \leftarrow follow[B] \cup follow[A]$
      fim
    fim
  até que até não haver mudanças
  retornar *follow*
fim

Uma tabela de análise sintática LL(1) pode ser usada para determinar as regras a serem usadas na análise de uma trada para todas as combinações de não terminais e *tokens* da entrada. A construção dessa tabela pode ser feita usando os conjuntos de *first* e *follow* usando o Algoritmo 3. Tendo a tabela de análise sintática LL(1) em mãos, é possível fazer a análise de uma sequência de *tokens* usando uma *stack*. O Algoritmo 4 mostra a análise sintática usando a tabela.

Algoritmo 3: Construção da Tabela LL(1)

Entrada: Gramática G
Saida: Tabela M
início
  $follow \leftarrow \text{FOLLOW}(G)$
  para cada regra $A \rightarrow \alpha$ em G **fazer**
    para cada terminal $a \in \text{FIRST}(\alpha) - \{\varepsilon\}$ **fazer**
      $M[A,a] \leftarrow (A \rightarrow \alpha)$
    fim
    se $\varepsilon \in \text{FIRST}(\alpha)$ **então**
      para cada terminal $b \in follow[A]$ **fazer**
        $M[A,b] \leftarrow (A \rightarrow \alpha)$
      fim
    fim
  fim
fim

2.3 Analisadores Sintáticos Ascendentes

Cooper e Torczon (2022), Mogensen (2024) e Thain (2020) falam sobre analisadores sintáticos ascendentes, mas para esse trabalhos será seguida a definição de Thain (2020). Os analisadores sintáticos ascendentes levam uma abordagem oposta aos analisadores sintáticos descendentes. Ao invés de começar com o símbolo inicial da gramática, os analisadores sintáticos ascendentes procuram sequências de *tokens* que façam par com o lado direito das produções da gramática e substituem as sequências de *tokens* pelo símbolo não terminal do lado esquerdo da produção. Esse processo é repetido até que toda a sequência de *tokens* seja reescrita e apenas reste o símbolo inicial da gramática.

Algoritmo 4: Análise LL(1)

```
Entrada: Gramática G, Tabela M
início
    pilha ← [$, S], Onde S é o símbolo inicial
    token ← próximo token
    enquanto pilha ≠ ∅ fazer
        X ← topo(pilha)
        se X é terminal então
            se X = token então
                desempilhe(pilha)
                token ← próximo token
            senão
                erro("Token inesperado")
            fim
        senão
            se M[X, token] = (X → α) então
                desempilhe(pilha)
                empilhe(α), da direita para a esquerda
            senão
                erro("Regra não encontrada")
            fim
        fim
    fim
fim
```

2.3.1 Analisadores Sintáticos SLR

Há um conjunto de gramáticas que podem ser analisadas usando técnicas de *shift-reduce* e um único *lookahead*, esse conjunto de gramáticas pode ser chamado de LR(0). As ações de *shift-reduce* são usadas para reduzir *tokens* de uma entrada a não terminais, quando uma sequência de *tokens* pode ser reduzida ao símbolo inicial da gramática a análise da entrada teve sucesso, caso contrário há um erro na entrada.

Todas as ações de *shift-reduce* possíveis podem ser calculadas para uma gramática construindo um autômato LR(0), que também pode ser chamado coleção de itens canônicos. Esse autômato guarda todas as possíveis posições de leitura das produções da gramática representadas por um ponto escrito no lado direito das produções.

As ações de *shift-reduce* são definidas pelas transições do autômato e pela posição de leitura, caso uma transição seja feita com um terminal a ação será de *shift*, caso uma transição seja feita com um não terminal a ação será de *goto*, caso a posição de leitura esteja no fim da produção a ação será de *reduce*. O Algoritmo 5 mostra como construir o autômato LR(0) com o auxílio do *closure* mostrado no Algoritmo 6.

Algoritmo 5: Autômato LR(0)

Entrada: Gramática G
Saida: Autômato M
início
  $M \leftarrow \varnothing$
  $s_0 \leftarrow \{[S' \rightarrow \cdot S]\}$, Onde S' é o novo símbolo inicial
  CLOSURE(s_0)
  $M \leftarrow M \cup \{s_0\}$
  $newStates \leftarrow \{s_0\}$
  para cada $s \in newStates$ **fazer**
    para cada $X \in T \cup NT$ **fazer**
      $s' \leftarrow \varnothing$
      para cada item $[A \rightarrow \alpha \cdot X\beta] \in s$ **fazer**
        $s' \leftarrow s' \cup \{[A \rightarrow \alpha X \cdot \beta]\}$
      fim
      se $s' \neq \varnothing$ **então**
        continua para a próxima iteração
      fim
      CLOSURE(s')
      se $s' \notin M$ **então**
        $M \leftarrow M \cup \{s'\}$
        $newStates \leftarrow newStates \cup \{s'\}$
      fim
      adicione transição $s \xrightarrow{X} s'$
    fim
    $newStates \leftarrow newStates - \{s\}$
  fim
  retornar M
fim

Quando um símbolo não terminal produz um símbolo terminal, apenas um caminho para derivação é possível, já que um não terminal não pode ser derivado, no entanto, quando um não terminal produz outro não terminal, o não terminal produzido terá outras derivações. Por essa razão é preciso levar em consideração as produções dos não terminais que estão sob a posição de leitura dentro da produção de outro não terminal. *Closure* é o nome dado a ação de completar os estados do autômato adicionando essas produções.

Usando o autômato LR(0) podemos construir uma tabela de ações e *goto* para facilitar o acesso a essas informações durante o processo de análise sintática. Essa pode ser calculada usando o Algoritmo 7. A análise sintática do analisador sintático SLR pode ser feita usando a tabela de ações e *goto* seguindo o Algoritmo 8.

Algoritmo 6: Closure LR(0)

Entrada: Gramática G, Estado S
início
  repetir
    para cada item $[A \to \alpha \cdot X\beta] \in S$ **fazer**
      para cada produção $X \to \gamma \in G$ **fazer**
        adicione $[X \to \cdot\gamma]$ a S, se não existir
      fim
    fim
  até que não haja novos itens
fim

Algoritmo 7: Construção da Tabela SLR

Entrada: Autômato LR(0) M
início
  para cada estado $s_i \in M$ **fazer**
    para cada item $[A \to \alpha \cdot a\beta] \in s_i$ **fazer**
      $s_j \leftarrow \delta(s_i, a)$
      se $s_j \neq \varnothing$ **então**
        ACTION$[s_i, a] \leftarrow$ "shift s_j"
      fim
    fim
    para cada item $[A \to \alpha\cdot] \in s_i$ **fazer**
      para cada $a \in Follow[A]$ **fazer**
        ACTION$[s_i, a] \leftarrow$ "reduce $A \to \alpha$"
      fim
    fim
    para cada não-terminal A **fazer**
      $s_j \leftarrow \delta(s_i, A)$
      se $s_j \neq \varnothing$ **então**
        GOTO$[s_i, A] \leftarrow s_j$
      fim
    fim
  fim
fim

2.3.2 Analisadores Sintáticos CLR

O analisador sintático *canonical* LR (CLR) é um analisador sintático descendente para gramáticas LR(1). O analisador sintático CLR usa o autômato LR(1) para construção da tabela de ações e *goto*, esse autômato é parecido com o autômato LR(0), o que diferencia os dois é que todos os itens do autômato LR(1) tem uma anotação do conjunto de *tokens* que podem aparecer depois desses itens. Esse conjunto é chamado *lookahead*.

Algoritmo 8: Análise SLR

Entrada: Tabelas ACTION, GOTO
início
  $pilha \leftarrow [s_0]$
  $token \leftarrow$ próximo token
  repetir
    $s \leftarrow$ topo($pilha$)
    se ACTION$[s, token]$ = "shift s_j" **então**
      empilhe($token$)
      empilhe(s_j)
      $token \leftarrow$ próximo token
    senão, se ACTION$[s, token]$ = "reduce $A \rightarrow \beta$" **então**
      desempilhe($2 \times |\beta|$), remove símbolos e estados
      $s' \leftarrow$ topo($pilha$)
      empilhe(A)
      empilhe(GOTO$[s', A]$)
    senão, se ACTION$[s, token]$ = "aceitar" **então**
      retorne(sucesso)
    senão
      erro("Erro de sintaxe")
    fim
  até que até ação de aceitação ou erro
fim

Algoritmo 9: Closure LR(1)

Entrada: Gramática G, Estado s
início
  repetir
    para cada item $[A \rightarrow \alpha \cdot B\beta, \ell] \in s$ **fazer**
      para cada produção $B \rightarrow \gamma \in G$ **fazer**
        $lookahead \leftarrow$ FIRST(β)
        se $\varepsilon \in$ FIRST(β) **então**
          $lookahead \leftarrow lookahead \cup \{\ell\}$
        fim
        para cada $a \in lookahead$ **fazer**
          adicione $[B \rightarrow \cdot\gamma, a]$ a s, se não existir
        fim
      fim
    fim
  até que não haja novos itens
fim

A construção do autômato LR(1) segue o mesmo algoritmo da construção do autômato LR(0) com algumas modificações. A primeira produção a ser adicionada no primeiro estado do autômato vai ser adicionada com o *lookahead*. Esse *lookahead* tem $ como único

elemento. Ao computar o *closure* serão considerados dois casos:

- Para produções da forma $A \to \alpha.B$ com *lookahead* de $\{L\}$, deverão ser adicionadas novas produções da forma $B \to .\gamma$ com *lookahead* de $\{L\}$
- Para produções da forma $A \to \alpha.B\beta$, com *lookahead* de $\{L\}$, deverão ser adicionadas novas produções da forma $B \to .\gamma$ com *lookahead* da seguinte forma:
  - Se β não produz ε, o *lookahead* é $First(G,\beta)$.
  - Se β produz ε, o *lookahead* é $First(G,\beta) \cup \{L\}$.

Com essas modificações chegamos aos algoritmos de *closure* LR(1) e construção do autômato LR(1) que podem ser vistos no Algoritmo 9 e no Algoritmo 10 respectivamente. O algoritmo de construção da tabela LR(1), mostrado no Algoritmo 11, também é parecido com o algoritmo de construção da tabela SLR, a única diferença é que o *lookahead* serão usados no lugar do conjunto follow para definir as ações de *reduce*. Quanto ao algoritmo de *parsing* usando a tabela LR(1), ele é o mesmo algoritmo de *parsing* usando a tabela SLR.

Algoritmo 10: Autômato LR(1)

Entrada: Gramática G
Saida: Autômato M
início
  $s_0 \leftarrow \{[S' \to \cdot S, \$]\}$
  CLOSURE(s_0)
  $M \leftarrow \{s_0\}$
  $newStates \leftarrow \{s_0\}$
  para cada $s \in newStates$ **fazer**
    para cada $X \in T \cup NT$ **fazer**
      $s' \leftarrow \varnothing$
      para cada item $[A \to \alpha \cdot X\beta, \ell] \in s$ **fazer**
        $s' \leftarrow s' \cup \{[A \to \alpha X \cdot \beta, \ell]\}$
      fim
      se $s' \neq \varnothing$ **então**
        CLOSURE(s')
        se $s' \notin M$ **então**
          $M \leftarrow M \cup \{s'\}$
          $newStates \leftarrow newStates \cup \{s'\}$
        fim
        adicione transição $s \xrightarrow{X} s'$
      fim
    fim
    $newStates \leftarrow newStates - \{s\}$
  fim
  retornar M
fim

Algoritmo 11: Construção da Tabela LR(1)

Entrada: Autômato LR(1) M
Saida: Tabelas ACTION, GOTO
início
  para cada estado $s_i \in M$ **fazer**
    para cada item $[A \to \alpha \cdot a\beta, \ell] \in s_i$ **fazer**
      $s_j \leftarrow \delta(s_i, a)$
      se $s_j \neq \varnothing$ **então**
        ACTION$[s_i, a] \leftarrow$ "shift s_j"
      fim
    fim
    para cada item $[A \to \alpha\cdot, \ell] \in s_i$ **fazer**
      para cada $a \in \ell$ **fazer**
        ACTION$[s_i, a] \leftarrow$ "reduce $A \to \alpha$"
      fim
    fim
    se item $[S' \to S\cdot, \$] \in s_i$ **então**
      ACTION$[s_i, \$] \leftarrow$ "aceitar"
    fim
    para cada não-terminal A **fazer**
      $s_j \leftarrow \delta(s_i, A)$
      se $s_j \neq \varnothing$ **então**
        GOTO$[s_i, A] \leftarrow s_j$
      fim
    fim
  fim
  retornar (ACTION, GOTO)
fim

2.4 *Svelte*

Svelte é um *framework* de componentes criado em 2016 por Harry Rich e pode ser considerado uma tecnologia recente em relação a outras similares (Krill, 2016). *Svelte* é semelhante aos *frameworks React* e *Vue*, mas tem uma abordagem bastante diferente no processamento de código. Os *frameworks* tradicionais usam código declarativo dirigido a estado (*declarative state-driven*). Isso aumenta a carga de processamento para o *browser* que precisa transformar essas estruturas declarativas em operações no *Document Object Model* (DOM) usando técnicas como *Virtual* DOM que é uma representação intermediária do DOM real (Harris, 2019a).

Ao contrário dos *frameworks* tradicionais, *Svelte* age como um compilador funcionando em *build-time* para transformar os componentes criados em código *Javascript* imperativo

altamente eficiente que atualiza o DOM apenas onde necessário. Isso permite escrever código para aplicações robustas sem necessidade de se preocupar muito com optimizações para que as aplicações sejam leves e performáticas.

Além de ter código mais leve e performático, a quantidade de código escrito usando *Svelte* é menor em comparação a outros *frameworks*. Já que *Svelte* compila o código base, o *framework* é livre para escolher a forma como o código deve ser escrito e, para maior simplicidade o código em Svelte segue a sintaxe da linguagem *Javascript*. Tal coisa não é possível com outros *frameworks* como *React* que funciona em *runtime* e tem sua sintaxe limitada a isso, sendo necessário mais código para estar em conformidade com o funcionamento do *framework*. Um exemplo disso é a atualização do estado de uma variável. Enquanto usando *Svelte* é apenas necessário usar o operador de atribuição para dar um novo valor à variável assim como na sintaxe *Javascript*. Usando *React* é necessário a utilização de funções chamadas *hooks* para a atribuição do novo valor (Harris, 2019b).

3 TRABALHOS RELACIONADOS

Nesta seção, estão descritas algumas ferramentas de visualização de algoritmos de análise sintática, suas funcionalidades e limitações.

3.1 *Teaching Compilers: Automatic Question Generation and Intelligent Assessment of Grammars' Parsing*

No trabalho de Muñoz *et al.* (2024) foi desenvolvida uma aplicação usada como *plug-in* no sistema de avaliação *SIETTE*. O objetivo da aplicação é gerar gramáticas livres de contexto, determinar se elas atendem os requisitos LL(1) ou SLR, construir as tabelas de *parsing* e avaliar os alunos. Essa aplicação foi usada na Universidade de Málaga durante 7 anos para avaliar mais de mil alunos.

Para a criação de gramáticas livres de contexto foram usados blocos de construção. Esses blocos são conjuntos de produções que podem ter seus terminais substituídos por não terminais representando outros blocos. Combinando diferentes blocos é possível ser obtidas gramáticas aleatórias. Para gerar as tabelas de *parsing* são implementados os algoritmos de construção dessas tabelas e algoritmos auxiliares. Apesar da equivalência entre gramáticas livres de contexto ser um problema indecidível como as gramáticas livres de contexto usadas são pequenas, a aplicação consegue testar a equivalência entre elas usando um algoritmo de força bruta.

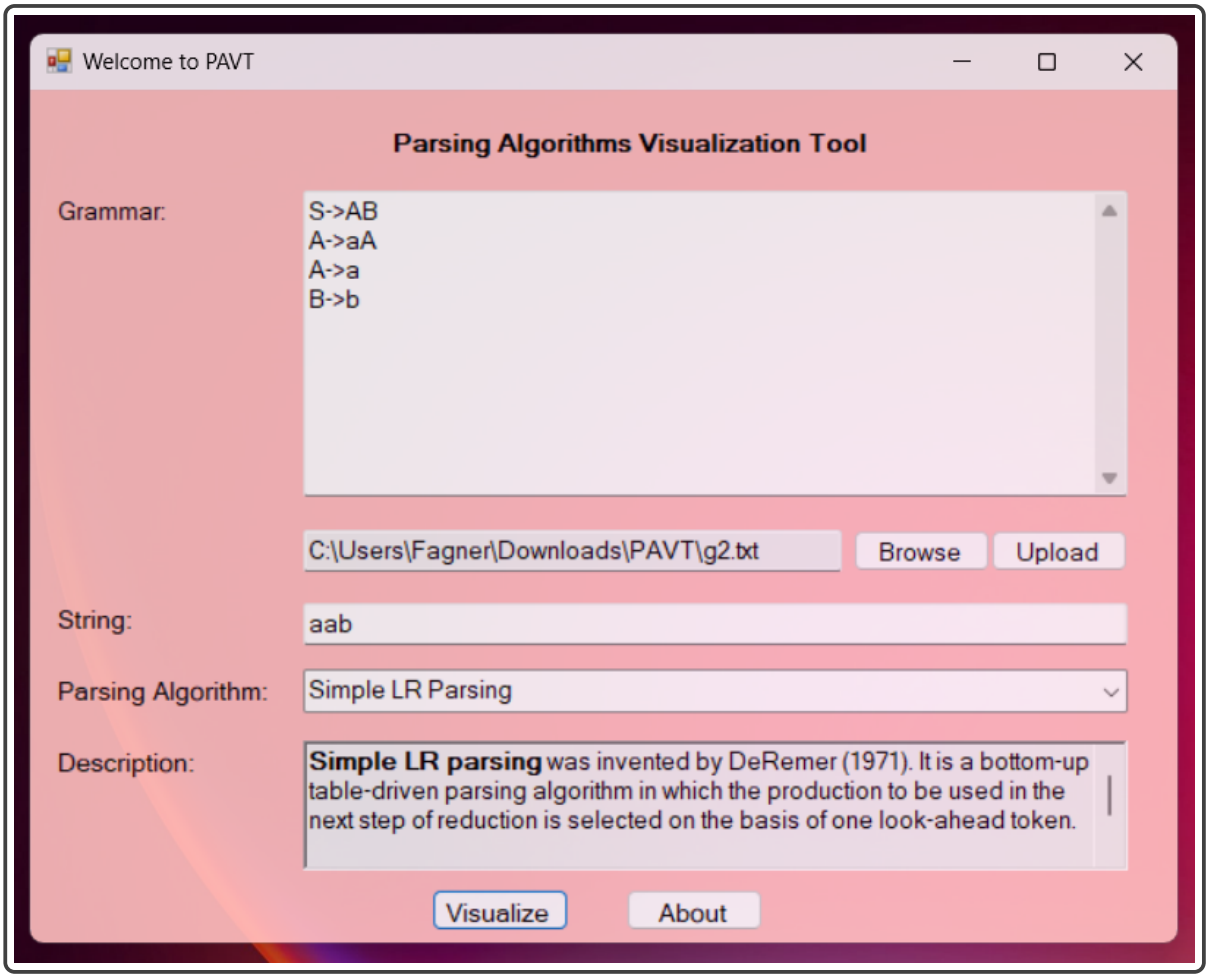
Após os dados da utilização da aplicação serem analisados, os autores concluíram que os testes gerados automaticamente têm dificuldade e resultados semelhantes aos obtidos com os testes criados por professores. Os autores também afirmam que os testes gerados automaticamente têm a vantagem de sempre diferir uns dos outros, requerendo um entendimento mais aprofundado dos alunos para resolução desses testes. Por fim, os autores citam como possibilidade de trabalhos futuros a criação de uma ferramenta com *feedback* gráfico e mais detalhado.

3.2 *PAVT: a tool to visualize and teach parsing algorithms*

No trabalho de Sangal, Kataria, Tyagi *et al.* (2018) foi introduzida a ferramenta *PAVT* para ensinar seis algoritmos de análise sintática. Os algoritmos abordados na ferramenta são *predictive parsing*, *simple LR (SLR) parsing*, *canonical LR(CLR) parsing*, *look-ahead*

LR(LALR) parsing, *earley parsing e Cocke-Younger-Kasami(CYK) parsing*. *PAVT* mostra uma breve descrição dos algoritmos e dá o resultado dos passos do algoritmo em formato de texto. Para utilizar a ferramenta o usuário deve digitar uma *string* para ser analisada e a gramática alvo. Também é possível fazer *upload* de um arquivo de texto contendo a gramática. A interface da ferramenta pode ser vista na Figura 2.

Figura 2 – Interface da ferramenta *PAVT*



Fonte: adaptada de Sangal, Kataria e Tyagi (2017).

PAVT tem módulos responsáveis pela visualização de cada algoritmo. Para todos é feita a análise da *string* de entrada. Caso a *string* seja aceita, é construída a árvore sintática representada da esquerda para a direita. Além disso, todos os elementos presentes nos algoritmos são apresentados. Esses elementos são o conjunto *first*, conjunto *follow*, conjunto de itens, tabela de *parsing* e derivação mais à direita.

A ferramenta foi usada no curso de construção de compiladores e ao fim do curso o

feedback dos alunos foi coletado. Os resultados obtidos indicaram que a ferramenta ajudou no aprendizado de algoritmos de análise sintática. Os autores afirmam que os resultados de cada passo dos algoritmos são dados em um formato comumente usado pelos professores, e ajudam os estudantes a praticar e entender os algoritmos.

3.3 *A Web-Based Educational System for Teaching Compilers*

O trabalho de Stamenković e Jovanović (2024) foi feito na universidade de Pristina para criar uma versão *web* de um sistema de simulação chamado *ComVis*. Essa versão foi criada com módulos que ensinam as fases da compilação. O sistema já havia sido desenvolvido em *Java* para *desktop*, no entanto, por questões de acessibilidade e melhor representação visual foi decidido criar a versão *web* do sistema. A motivação por trás desse trabalho foi a dificuldade dos alunos da universidade na disciplina de compiladores. Os autores também citam como a utilização de um *software* interativo pode ajudar na motivação.

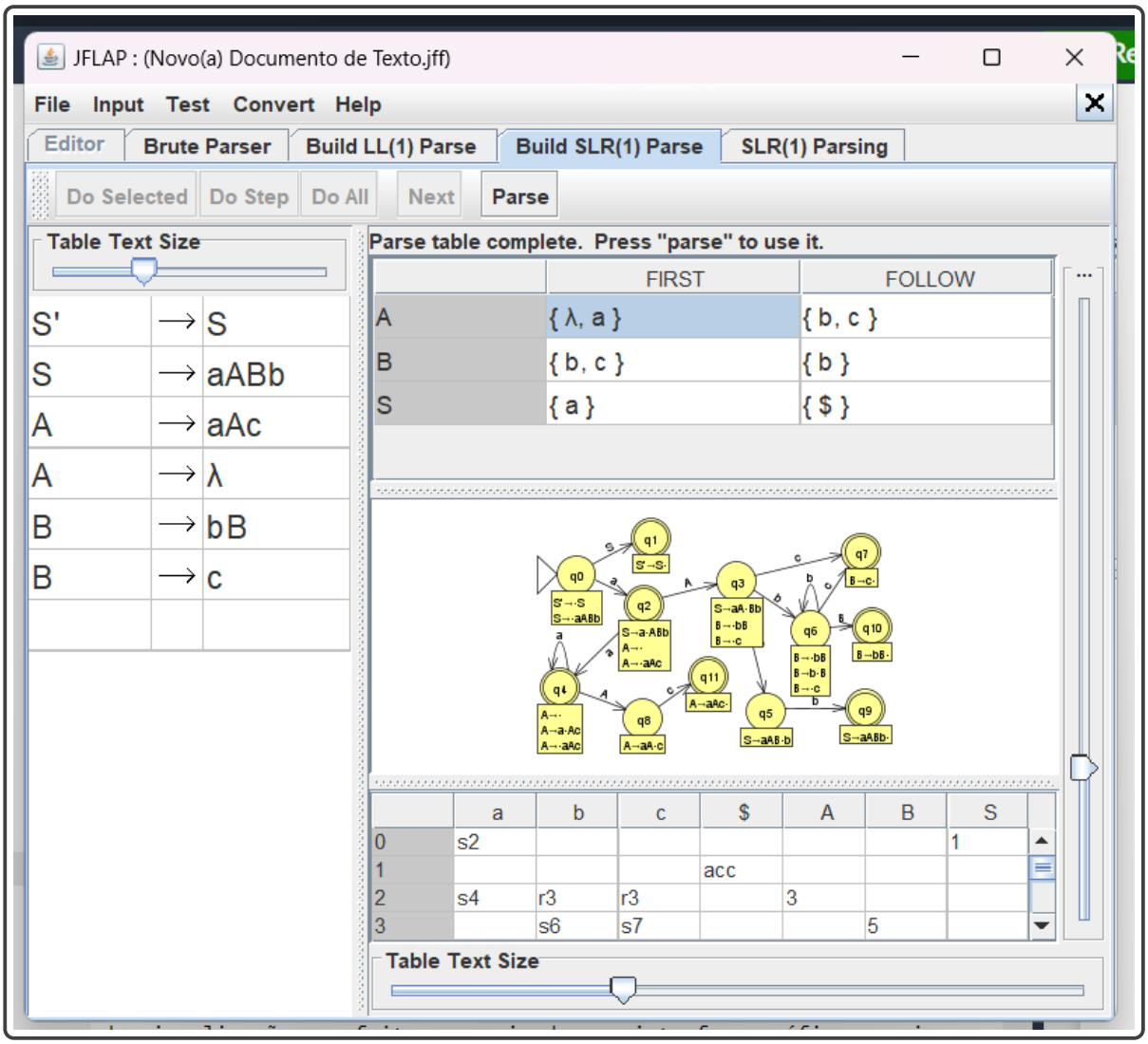
O sistema foi desenvolvido usando *Java Server Page*. Já que a versão *desktop* do sistema foi feita em *Java*, grande parte da base de código pôde ser reutilizada. Outras tecnologias usadas foram *HTML*, *CSS*, *JavaScript* e *Graphviz* para criação de gráficos e diagramas.

No estudo foi feita uma análise comparativa entre as versões *web* e *desktop* do sistema. A partir dessa análise os autores concluíram que a versão *web* criada tem melhor acessibilidade, visualização, controle da simulação e *feedback*. Também foi feita uma avaliação quantitativa da eficiência do sistema. Os resultados mostram que os estudantes que usaram o sistema tiveram melhor desempenho do que aqueles que não usaram o sistema.

3.4 *A Tool for Visualization of Parsers: JFLAP*

JFLAP (*Java Formal Languages and Automata Package*) é uma ferramenta *desktop* criada por Rodger e Duke University (2018) primariamente para a construção e simulação de autômatos e gramáticas livres de contexto. O trabalho de Devakumar, Naik e Sajja (2014) introduz os algoritmos LL(1) e SLR na ferramenta. Dessa forma, *JFLAP* também pode ser usado para visualização dos algoritmos LL(1), SLR e de força bruta. Como mostra a Figura 3, a ferramenta apresenta os conjuntos *first* e *follow*, o autômato dos estados do analisador sintático e a tabela de ações. O processo de *parsing* também é disponibilizado assim como a árvore sintática.

Figura 3 – Imagem da ferramenta *JFLAP*



Fonte: Rodger e Duke University (2018).

3.5 Considerações

Apesar de já existirem ferramentas de visualização de *parsers*, algumas desvantagens ainda precisam ser consideradas. Uma limitação é que o conteúdo não tem muita interatividade. A ferramenta apresentada no trabalho de Sangal, Kataria, Tyagi *et al.* (2018), por exemplo, apresenta apenas em um arquivo de texto. Isso pode dificultar a compreensão dos conceitos. Outra limitação é a falta de detalhamento do passo a passo dos algoritmos, por exemplo, a ferramenta apresentada no trabalho de Stamenković e Jovanović (2024) foca apenas no último passo da análise sintática, excluindo a parte de construção das tabelas de *parsing* e conjuntos de itens. Isso impede que os estudantes acompanhem o funcionamento interno dos processos

de análise. Essas lacunas representam oportunidades de melhoria para que as ferramentas de visualização de *parsers* se tornem ainda mais eficazes no apoio ao ensino e aprendizagem de análise sintática. O resumo do comparativo dos trabalhos citados anteriormente pode ser visto no Quadro 1.

Quadro 1 – Comparativo de trabalhos relacionados

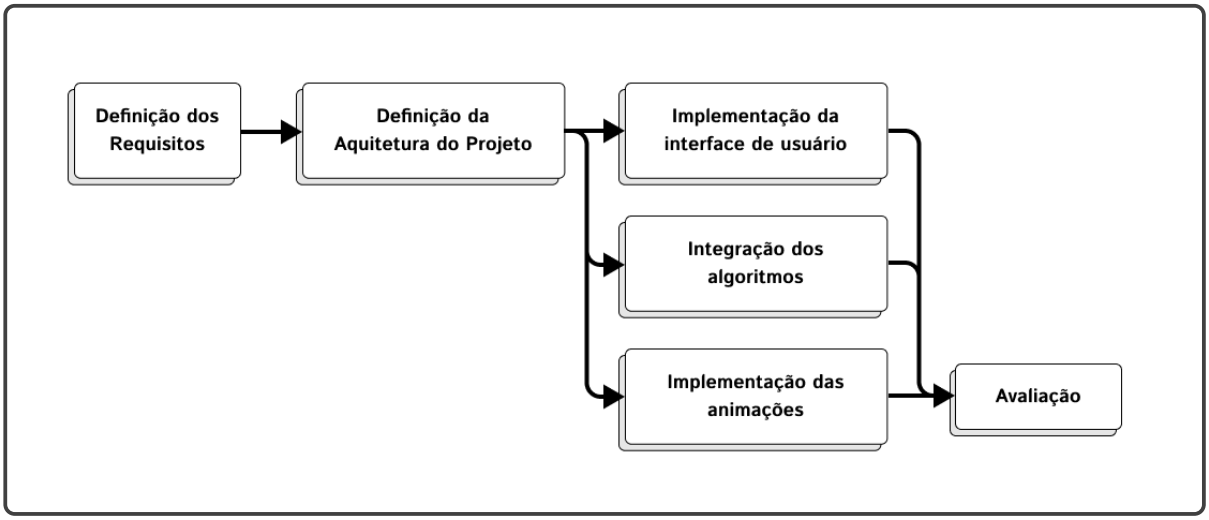
Trabalho	Algoritmos		
	LL(1)	SLR	LR(1)
Muñoz *et al.* (2024)	x	x	
Sangal, Kataria, Tyagi *et al.* (2018)		x	x
Stamenković e Jovanović (2024)		x	x
Devakumar, Naik e Sajja (2014)	x	x	
Esse trabalho	x	x	x

Fonte: fornecido pelo autor

4 METODOLOGIAS

Nesta seção serão apresentadas as metodologias usadas para o desenvolvimento desse trabalho. As etapas a serem seguidas estão representadas no fluxograma da Figura 4.

Figura 4 – Fluxograma das etapas



Fonte: fornecida pelo próprio autor

4.1 Definição dos requisitos

O primeiro passo no desenvolvimento de *software* é a definição dos requisitos, com eles tem-se um direcionamento sobre o que deve ser feito. A partir da revisão bibliográfica foram definidos alguns requisitos básicos que deveriam estar presentes na ferramenta. Apesar das ferramentas compartilharem as mesmas funcionalidades básicas já definidas inicialmente, outras têm características interessantes que podem ser reaproveitadas. Como requisito não funcional foi definido oferecer suporte multi-plataforma, para *mobile*, *desktop* e *web*. Como requisitos funcionais foram definidos os seguintes:

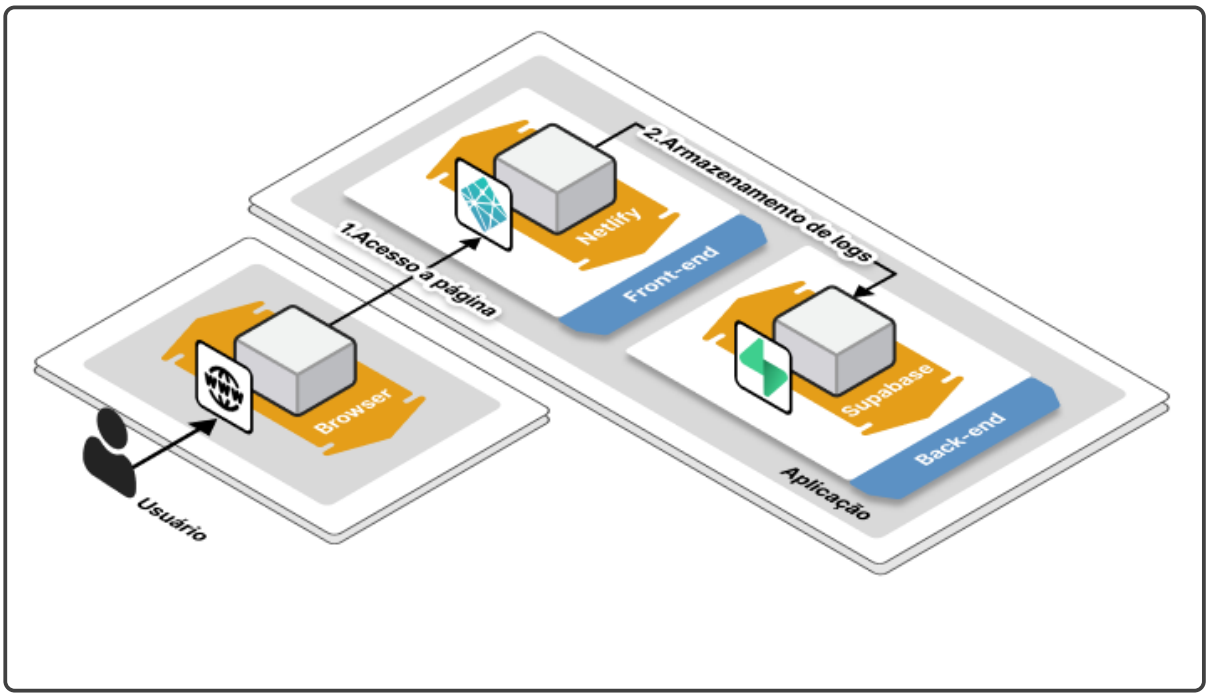
- Permitir que os usuários digitem uma gramática para ser analisada.
- Permitir que os usuários visualizem o estado das estruturas dos algoritmos.
- Permitir que os usuários avancem, retornem e reiniciem os passos da execução dos algoritmos.
- Permitir que os usuários selecionem o algoritmo a ser visualizado.
- Permitir que os usuários acessem o pseudocódigo dos algoritmos.

- Permitir que os usuários adicionem *breakpoints* no pseudocódigo.
- Permitir que os usuários visualizem o autômato dos algoritmos LR.
- Permitir que os usuários visualizem uma descrição dos botões da interface.
- Permitir que os usuários digitem uma *string* para ser analisada.
- Permitir que os usuários visualizem a análise de uma *string*.
- Permitir que os usuários visualizem a árvore sintática de uma *string*.
- Permitir que os usuários copiem em formato de texto o resultado dos algoritmos.

4.2 Definição da arquitetura do projeto

O projeto será construído usando o *framework Svelte*, sem *Server Side Rendering* (SSR), para ser possível o funcionamento *offline* da ferramenta, já que a ferramenta não teria acesso ao servidor não seria possível usá-lo para renderizar elementos. *Svelte* compila a base de código e cria uma coleção de arquivos estáticos que constituem a página *web* e a base para o suporte multi-plataforma da aplicação.

Figura 5 – Arquitetura da aplicação em nuvem



Fonte: fornecida pelo próprio autor

O *framework Capacitor*[1] consome essa coleção de arquivos e cria um projeto para plataforma *Android* usado para criar a versão *mobile* da ferramenta usando o *Android Studio*. O *framework Tauri*[2] constrói instaladores para *desktop* diretamente da coleção de arquivos estáticos. A plataforma alvo dos instaladores é a plataforma em que eles são construídos, já que o *framework* não tem suporte para construção *cross-platform* é necessária a utilização de máquinas virtuais para construir instaladores para diferentes plataformas *desktop*.

Para a versão *online* da aplicação, os serviços de computação em nuvem *Netlify*[3] e *Supabase*[4] serão usados para hospedar respectivamente a aplicação e a data-base usada por ela. *Netlify* e *Supabase* foram escolhidos pela facilidade de uso e pelo oferecimento gratuito dos serviços para aplicações pequenas como a proposta nesse trabalho.

O projeto da aplicação foi hospedado em um repositório no *Github*[5] que é uma plataforma *web* de hospedagem de repositórios do sistema de versionamento *Git*[6]. Graças a integração entre *Netlify* e *Github*, a aplicação é atualizada automaticamente com cada *commit* feito para o repositório do projeto. Por fim, a plataforma *Supabase* será utilizada para armazenar arquivos de *log* da utilização da aplicação durante os testes.

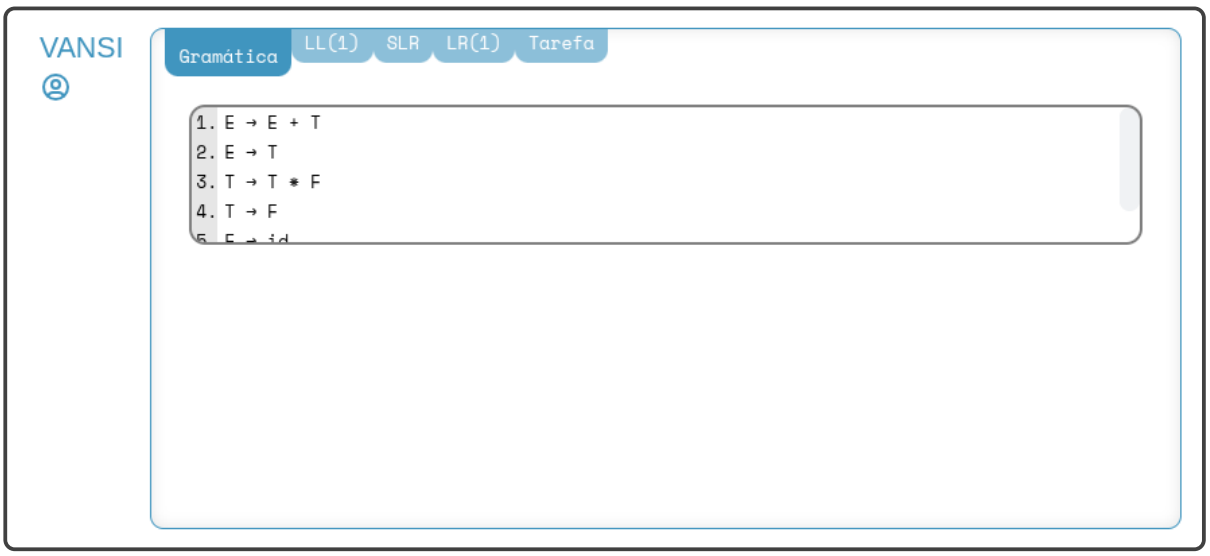
4.3 Implementação da interface de usuário

Nesta seção estão descritos o *layout* da ferramenta e os elementos que o constituem. Começando pela barra de navegação, que contém abas que levam para guias de entrada da gramática e dos algoritmos. A primeira aba é a de entrada da gramática, nessa aba há apenas uma caixa de texto para digitar a entrada. Os elementos descritos nesse parágrafo podem ser vistos na Figura 6.

Na aba dos algoritmos há uma barra de ações com botões separados em dois grupos. No grupo da esquerda, os dois primeiros botões abrem respectivamente uma janele de diálogo com os resultados em formato de texto e uma janela de diálogo com uma descrição do algoritmo. Os outros dois botões do grupo da esquerda alternam entre a visualização da construção do *parser* e a visualização da análise de uma *string* usando o *parser* construído. No grupo de botões da direita estão os controles de fluxo da execução do algoritmo. O primeiro botão vai

[1] https://capacitorjs.com
[2] https://tauri.app
[3] https://www.netlify.com
[4] https://supabase.com
[5] https://github.com
[6] https://git-scm.com

Figura 6 – Aba de entrada da gramática



Fonte: fornecida pelo próprio autor

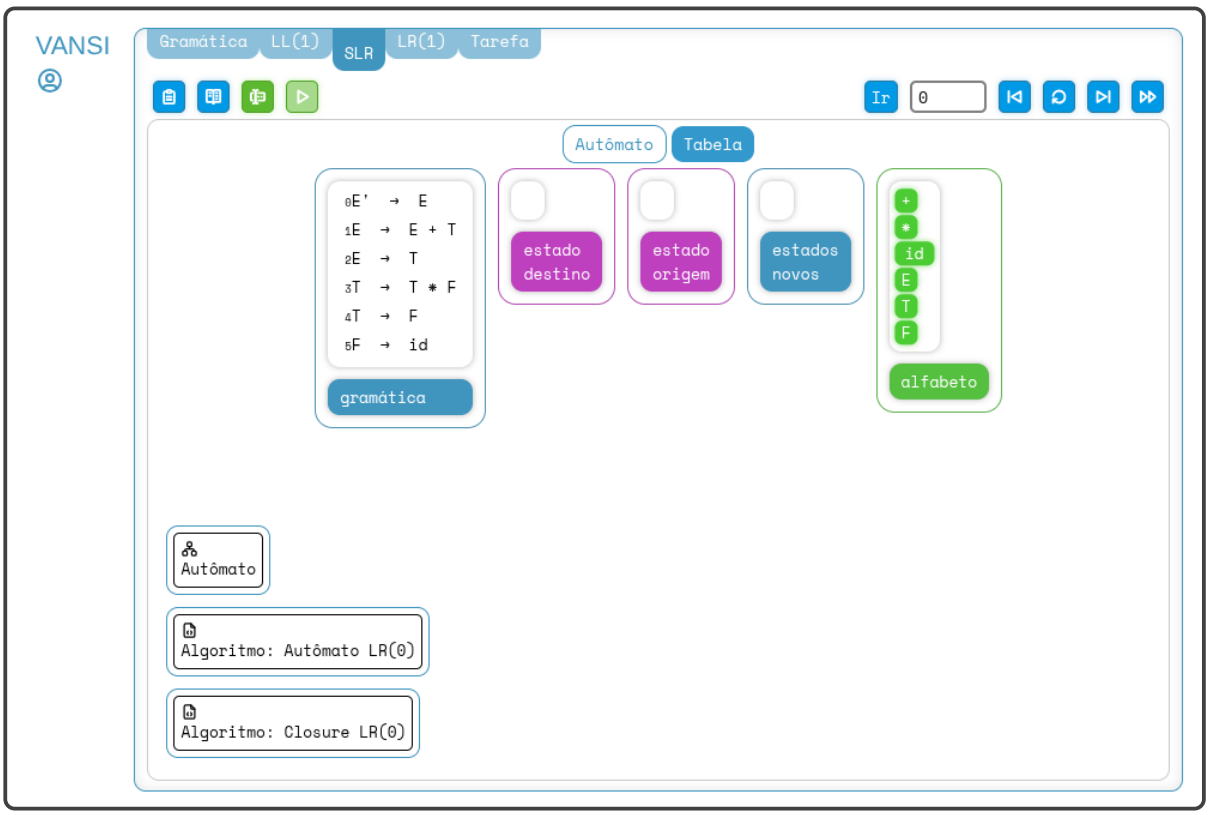
para o passo da execução digitado no campo de texto ao lado desse botão, os demais botões realizam as respectivas ações: ir para o passo anterior, reiniciar, ir para o passo seguinte e pular para o fim da execução.

Logo abaixo da barra de ações há uma lista de botões, cada um deles abre a visualização de um algoritmo secundário usado na construção do parser selecionado e abaixo desses botões estão os elementos que representam as estruturas de dados dos algoritmos. Os elementos descritos nesse parágrafo podem ser vistos na Figura 7.

Ainda na aba dos algoritmos, existem janelas flutuantes onde são mostrados os pseudocódigos dos algoritmos e autômatos. Essas janelas flutuantes são abertas clicando em um botão flutuante com um ícone e título descrevendo a janela. Dentro dessas janelas, no canto superior direito existe uma bandeja de ações com alguns botões. O primeiro botão minimiza a janela, o segundo ativa a ação de mover a janela ao clicar e arrastar ela e o terceiro ativa a interação com a janela.

Na janela flutuante que mostra um autômato há um quarto botão na bandeja de ações. Esse botão permite que os estados do autômato sejam movidos. Na janela flutuante de pseudocódigo os números das linhas são clicáveis para que o usuário adicione *breakpoints*. Por fim, nos cantos das janelas flutuantes existem pequenos quadrados que servem para redimensionar a janela. Os elementos descritos nesse parágrafo e no parágrafo anterior podem ser vistos na Figura 8.

Figura 7 – Aba de visualização do algoritmo SLR



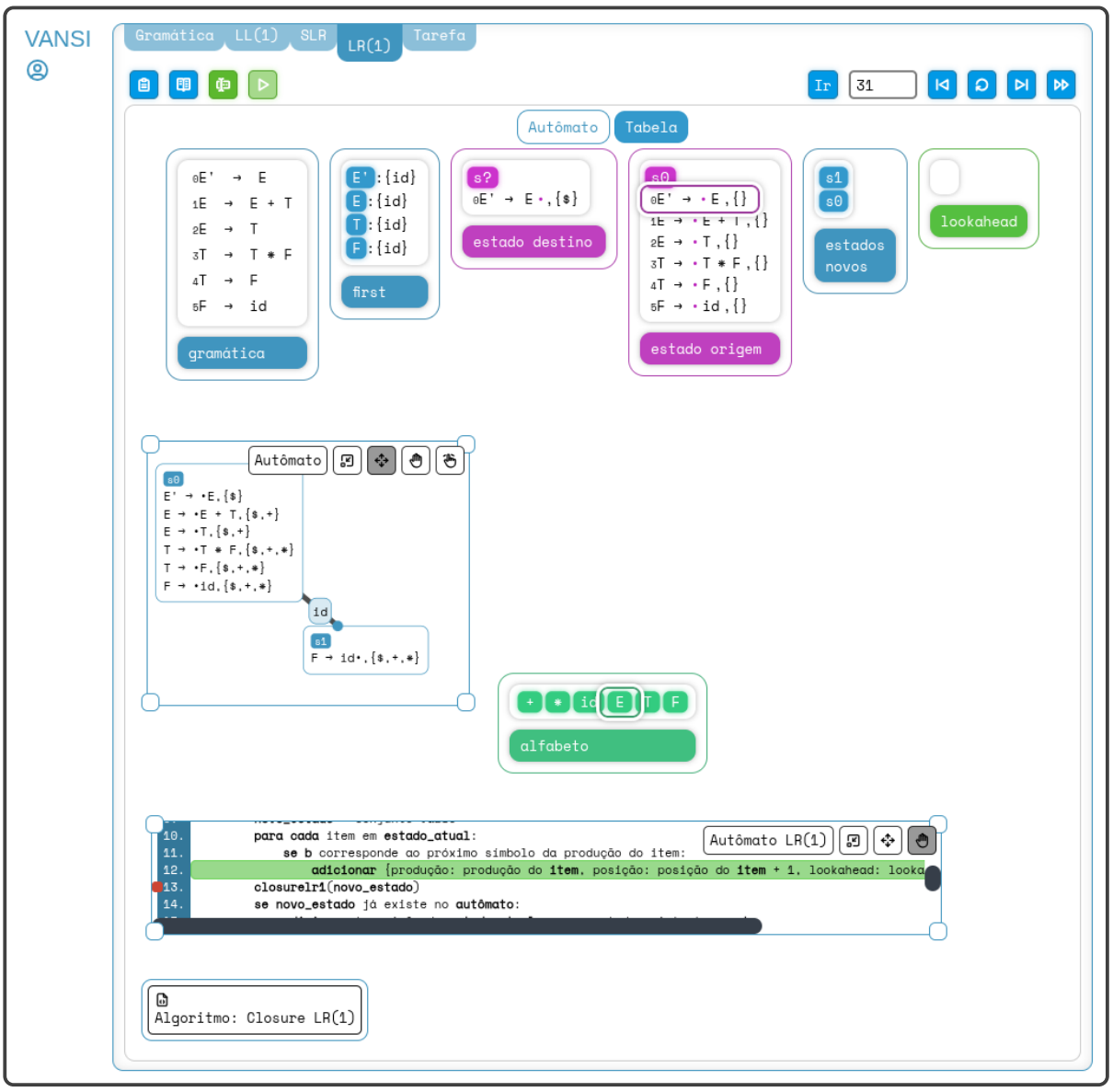
Fonte: fornecida pelo próprio autor

Dentro da aba de análise de *strings*, no lado esquerdo está uma área de visualização da árvore sintática, no lado direito está um campo de texto para digitar a *string* de entrada e logo abaixo estão os elementos que representam as estruturas de dados dos algoritmos. Os elementos descritos nesse parágrafo podem ser vistos na Figura 9.

As janelas de diálogo aparecem contidas na área de visualização da execução dos algoritmos. Acima de cada janela está um botão de fechar que cobre o comprimento inteiro da janela. Dentro da janela de diálogo para copiar os resultados dos algoritmos em formato de texto estão os resultados de cada algoritmo divididos em seções e ao lado do título de cada seção está um botão de copiar. Na janela de diálogo de descrição do algoritmo tem apenas conteúdo textual. Os elementos descritos nesse parágrafo podem ser vistos na Figura 10.

Por fim foi implementado um caixa de texto que mostra uma descrição dos elementos da interface de usuário ao passar o mouse por cima de um desses elementos. Essa caixa de texto funciona como um *tooltip* sendo escondida quando o mouse sai de perto dos elementos.

Figura 8 – Aba de visualização do algoritmo LR(1)



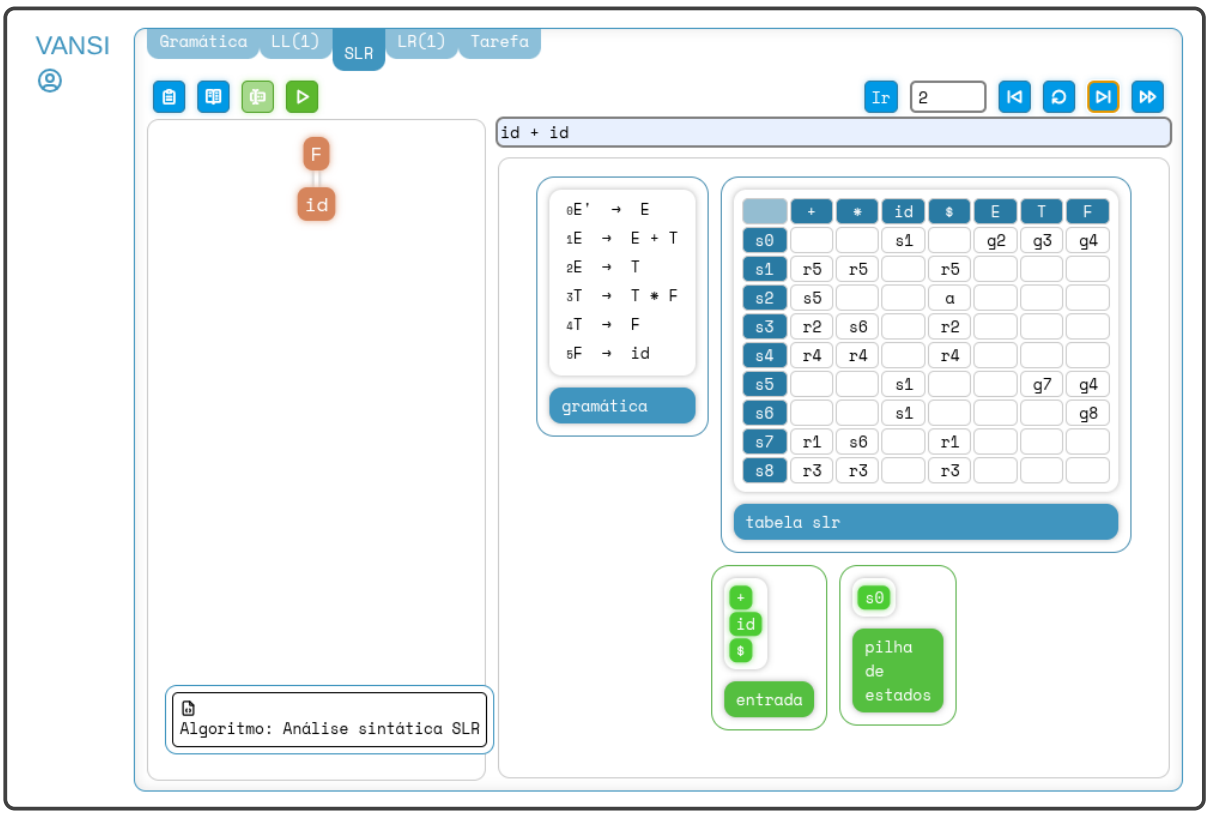
Fonte: fornecida pelo próprio autor

4.4 Integração dos algoritmos à ferramenta

O primeiro passo da integração dos algoritmos é a criação de elementos gráficos representando as estruturas de dados usados nesses algoritmos. Foram criados em total seis elementos para as estruturas de pilha, conjunto, tabela(matriz), estado de autômato, autômato e árvore sintática.

Os quatro primeiros elementos são caixas criadas com *HTML* já que seus dados podem ser representados com simples listas de texto. Na parte inferior dessas caixas está o título

Figura 9 – Aba de visualização da análise de uma *string*



Fonte: fornecida pelo próprio autor

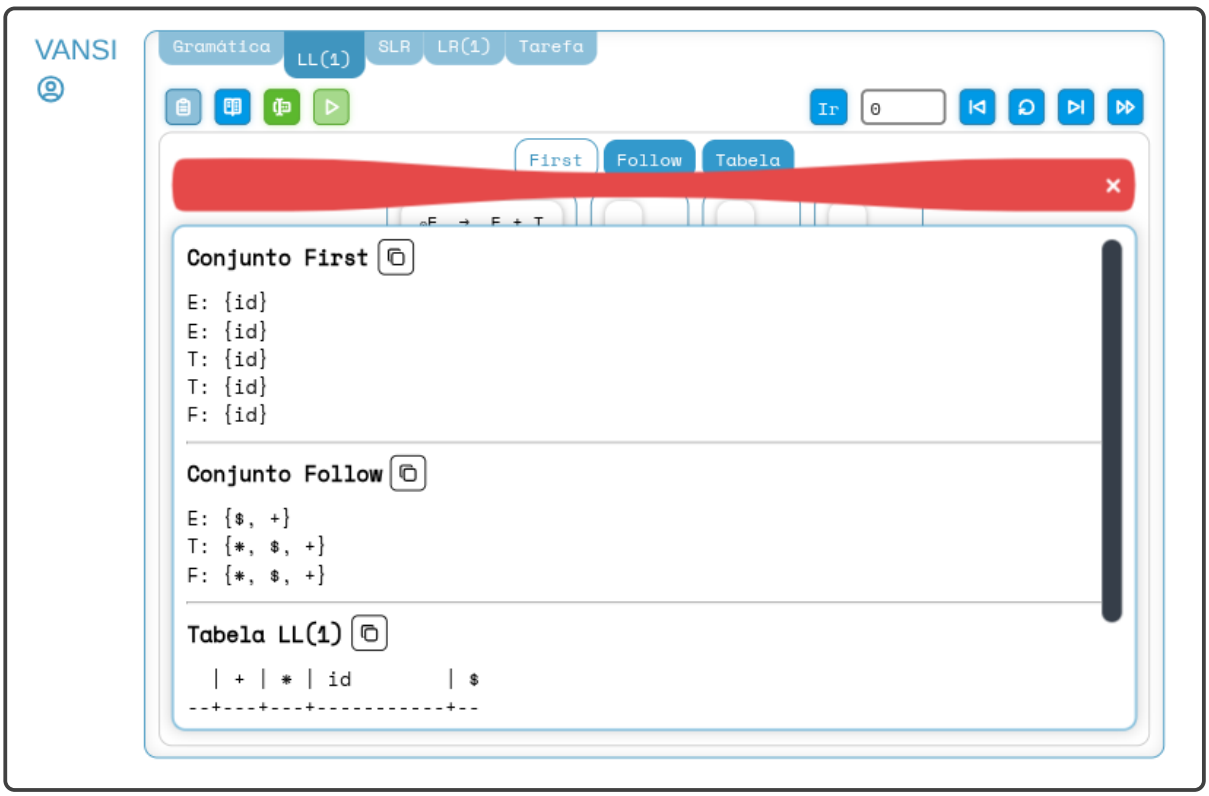
do elemento e na parte superior estão elementos textuais mostrando os dados das estruturas com destaques de cor nas chaves dos conjuntos e tabelas. Os dois últimos elementos são criados com SVG já que seus dados precisam ser estruturados de forma complexas.

Com os elementos criados para representar as estruturas é possível mostrar os passos da execução atualizando os dados desses elementos conforme as mudanças feitas nas estruturas. Após a sincronização dos elementos com os passos de execução é possível criar o controle de fluxo.

A primeira forma de implementação do controle de fluxo usada na aplicação foi usar uma única função que modifica as estruturas de dados e atualiza os elementos da interface em simultâneo. Essa função é controlada usando os mecanismos de execução assíncrona do *JavasScript* sendo eles *async*, *await* e *Promises*.

Nessa implementação as pausas são feitas usando uma *Promise* com retorno indefinido. A ação de ir para o próximo passo é feita rejeitando a *Promise* o que resume a execução da função. E por fim a ação de voltar, pular e reiniciar são feitas retornando uma nova chamada

Figura 10 – Janela de diálogo para copiar resultados



Fonte: fornecida pelo próprio autor

da função e executando a função e pulando as pausas até chegar no passo esperado.

O problema com a forma de implementação citada acima é que os passos da execução não são pré-calculados. Por essa razão, se a execução de um algoritmo tiver uma quantidade muito grande de passos, quando a ação de voltar ou pular for performada, vários passos serão executados em sequência até chegar no passo esperado.

Dessa forma, dado o fato que funções de atualização da interface também serão executadas, a aplicação vai requerer uma quantidade maior de poder computacional e vai parar de responder até que a execução dos passos seja finalizada.

Pela razão citada acima foi necessário criar uma segunda forma de implementação de controle de fluxo. Nessa implementação o estado das estruturas em cada passo é salvo em uma *array* usando apenas as estruturas de dados sem modificar a interface. Dessa forma, a aplicação consegue pré-calcular os passos da execução sem muito esforço. Assim, quando as ações de controle de fluxo são performadas, os estados salvos são carregados e a interface é atualizada de acordo com eles.

4.5 Implementação das animações dos algoritmos

As mudanças de estados que ocorrem nos algoritmos podem ser melhor compreendidas se poderem ser visualizadas como transições ao invés de mostrar as mudanças saltando do estado inicial para o estado final. Usar animações torna a visualização das mudanças muito mais dinâmica.

As primeiras animações implementadas são as transições de estilo usando a propriedade *CSS transition* que aplica uma transição suave entre mudanças de estilo dos elementos como tamanho e cor. As animações seguintes foram implementadas usando *JavasScript* para serem atualizadas dinamicamente a medida que os elementos fossem modificados mudando de tamanho ou posição na interface.

Começando com a animação de destaque dos dados dentro de um elemento da interface. Essa animação é usada quando um dado de um elemento é utilizado, seja quando um item é adicionado a um elemento ou quando um item é usado como chave para acessar o item de outro elemento. Essa animação põe um item em destaque com um bloco colorido exibido atrás do item. Quando itens são adicionados aos elementos de tabela, as linhas e colunas também são destacadas com uma cor de fundo.

Vários elementos têm iterações feitas sobre seus itens dispostos na vertical. Para demonstrar cada iteração, uma moldura se movimenta sobre os elementos deixando claro a qual item cada iteração se refere. Alguns elementos têm itens que são listas de itens dispostos na horizontal. Para demonstrar iterações sobre essas listas de itens a animação de destaque citada no parágrafo anterior é usada para cada iteração.

Alguns dados são movidos entre elementos como no algoritmo *first* que move símbolos da direita de uma produção para o conjunto *first*. Para demonstrar essa transferência entre os elementos uma linha é exibida que se estende do item de origem até o elemento de destino com uma animação.

4.6 Avaliação da ferramenta

Será realizado um teste prático com um grupo de estudantes, onde os eles serão solicitados a realizar tarefas específicas utilizando a ferramenta. Serão coletados dados quantitativos, como tempo de execução das tarefas e taxa de acerto, bem como dados qualitativos por questionários e entrevistas para avaliar a percepção dos estudantes sobre a eficácia da fer-

ramenta. Além disso, a comparação dos resultados obtidos com um grupo de controle que não utiliza a ferramenta ajudará a avaliar o impacto da visualização na compreensão e desempenho dos alunos. Essa abordagem abrangente de avaliação garantirá uma análise completa da eficácia e utilidade da ferramenta desenvolvida nesse trabalho.

REFERÊNCIAS

COOPER, K.D.; TORCZON, L. **Engineering a Compiler**. [*S. l.*]: Elsevier Science, 2022.

DEVAKUMAR, S; NAIK, Ds; SAJJA, V Ramakrishna. A TOOL FOR VISUALISATION OF PARSERS: JFLAP. **Journal of Innovative Research and Solutions(JIRAS)**, v. 1, p. 1–4, mar. 2014.

HARRIS, Rich. **Svelte 3: Rethinking reactivity**. 22 abr. 2019. Disponível em: https://svelte.dev/blog/svelte-3-rethinking-reactivity. Acesso em: 27 maio 2024.

HARRIS, Rich. **Write less code**. 20 abr. 2019. Disponível em: https://svelte.dev/blog/write-less-code. Acesso em: 27 maio 2024.

JAIN, Aashi; GOYAL, Archita; CHAKRABORTY, Pinaki. PPVT: a tool to visualize predictive parsing. **ACM Inroads**, Association for Computing Machinery, New York, NY, USA, v. 8, n. 1, p. 43–47, fev. 2017. ISSN 2153-2184.

KRILL, Paul. **Slim, speedy Svelte framework puts JavaScript on a diet | InfoWorld**. Slim, speedy Svelte framework puts JavaScript on a diet. 2 dez. 2016. Disponível em: https://www.infoworld.com/article/3146966/slim-speedy-svelte-framework-puts-javascript-on-a-diet.html. Acesso em: 27 maio 2024.

MOGENSEN, Torben Ægidius. **Introduction to compiler design**. [*S. l.*]: Springer Nature, 2024.

MUÑOZ, Ricardo Conejo *et al.* Teaching Compilers: Automatic Question Generation and Intelligent Assessment of Grammars' Parsing. **IEEE Transactions on Learning Technologies**, v. 17, p. 1734–1744, 2024.

RODGER, Susan H; DUKE UNIVERSITY. **JFLAP**. 27 jul. 2018. Disponível em: www.jflap.org. Acesso em: 2 maio 2024.

ROMANOWSKA, Katarzyna *et al.* Towards Developing an Effective Algorithm Visualization Tool for Online Learning. *In:* 2018 IEEE SmartWorld, Ubiquitous Intelligence & Computing, Advanced & Trusted Computing, Scalable Computing & Communications, Cloud & Big Data Computing, Internet of People and Smart City Innovation (SmartWorld/SCALCOM/UIC/ATC/CBDCom/IOP/SCI). [*S. l.: s. n.*], 2018. p. 2011–2016. DOI: 10.1109/SmartWorld.2018.00336.

SANGAL, Somya; KATARIA, Shreya; TYAGI, Twishi. **PAVT**. 9 fev. 2017. Disponível em: https://sourceforge.net/projects/pavt/. Acesso em: 2 maio 2024.

SANGAL, Somya; KATARIA, Shreya; TYAGI, Twishi *et al.* PAVT: a tool to visualize and teach parsing algorithms. **Education and Information Technologies**, Springer, v. 23, p. 2737–2764, 2018.

STAMENKOVIĆ, Srećko; JOVANOVIĆ, Nenad. A Web-Based Educational System for Teaching Compilers. **IEEE Transactions on Learning Technologies**, v. 17, p. 143–156, 2024.

THAIN, D. **Introduction to Compilers and Language Design: Second Edition**. [*S. l.*]: Amazon Digital Services LLC - Kdp, 2020.

UPPAL, Tanya; SRIVASTAVA, Saumitya; SAINI, Kavita. Web Development Framework : Future Trends. **2022 4th International Conference on Advances in Computing, Communication Control and Networking (ICAC3N)**, p. 2181–2184, 2022. Disponível em: https://api.semanticscholar.org/CorpusID:257809936.