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

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* que são 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 de analisador sintático ou *parser*. Os *parsers* podem ser classificados em dois tipos, *bottom-up* (ou ascendente) que funciona reduzindo os *tokens* a produções da gramática e *top-down* (ou descendente) que segue o caminho oposto do *parser bottom-up* tentando encontrar produções correspondentes a 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. 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 têm como objetivo 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.

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 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 e por fim, sobre a tecnologia usada para fazer a integração com o *Moodle*.

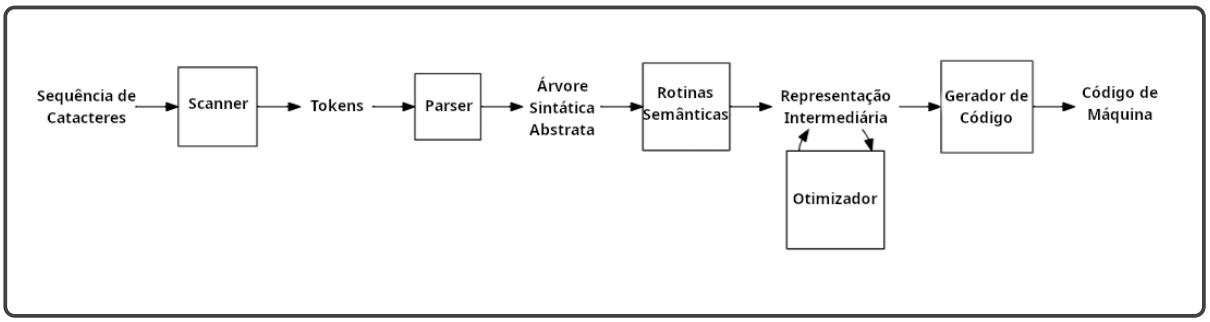
2.1 Compiladores

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, e 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 linguagem 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 definiçã 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 *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*, a construção desse conjunto pode ser feita seguindo o Algoritmo 1. As definições dos algoritmos foram tiradas do trabalho de Thain (2020).

Algoritmo 1: First

Entrada: Gramática G, Símbolo X
Saida: Conjunto first
início
  first $\leftarrow \varnothing$
  se X é Terminal **então**
    first $\leftarrow \{X\}$
  senão
    repetir
      para cada regra da forma $X \rightarrow Y_1 Y_2 ... Y_k$ na Gramática G **fazer**
        se a está em First(G,Y_1) ou a está em First(G,Y_n) e $Y_1..Y_{n-1} \Rightarrow \varepsilon$ **então**
          first $\leftarrow$ first$[X] \cup \{a\}$
        fim
        se $Y_1 ... Y_k \Rightarrow \varepsilon$ **então**
          first $\leftarrow$ first$[X] \cup \{\varepsilon\}$
        fim
      fim
    até que não haja mais mudanças
  fim
  retornar first
fim

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 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.

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.

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 definiçã de Thain (2020). Os analisadores sintáticos ascendentes levam uma abordagem oposta aos analisadores sintáticos

Algoritmo 2: Follow

```
Entrada: Gramática G
Saida: Dicionário follow
início
    follow ← Dicionário<Símbolo, Conjunto>
    follow[S] ← {$} onde S é o símbolo inicial
    repetir
        para cada produção P em G fazer
            se P é da forma A → αBβ então
                follow[B] ← follow ∪ First(G,β) (exceto ε)
            fim
            se P é da forma A → αB ou First(G,β) contém ε então
                follow[A] ← follow[A] ∪ follow[B]
            fim
        fim
    até que até não houver mais mudanças
    retornar follow
fim
```

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.

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

Algoritmo 3: Construção da tabela LL(1)

Entrada: Gramática G
Saida: Tabela M
início
  para cada regra da forma $A \rightarrow \alpha$ em G **fazer**
    para cada terminal a (exceto ε) em First(G,α) **fazer**
      $T[A,a] \leftarrow (A \rightarrow \alpha)$
    fim
    se ε está em First(G,α) **então**
      para cada terminal b (incluindo \$) em Follow(G)[A] **fazer**
        $T[A,b] \leftarrow (\mathrm{A} \rightarrow \alpha)$
      fim
    fim
  fim
fim

Algoritmo 4: Análise com tabela LL(1)

Entrada: Gramática G, Tabela T
início
  Pilha P
  monta \$ e S em P
  Token $c \leftarrow$ o primeiro token na entrada
  enquanto P não está vazio **fazer**
    Símbolo $X \leftarrow$ o topo de P
    se X faz par com c **então**
      remova X de P
      $c \leftarrow$ proximo token da entrada
      continue
    fim
    se X é um terminal **então**
      para com um erro
    fim
    se T [X, c] aponta para a regra $X \rightarrow \alpha$ **então**
      remova X de P
      monta os símbolos α em P
      continue
    fim
    se T [X, c] aponta para um estado de erro **então**
      para com um erro
    fim
  fim
fim

produção a ação será de *reduce*.

O Algoritmo 6 mostra como construir o autômato LR(0) com o auxílio do *closure* mostrado no Algoritmo 5.

Algoritmo 5: Closure LR(0)

```
Entrada: Gramática G, Estado S
início
    repetir
        para cada item da forma A → α.Xβ em S com X ∈ NT fazer
            para cada produção da forma X → γ em G fazer
                adiciona uma produção da forma X → .γ a S
            fim
        fim
    até que não tenha itens a serem adicionados
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.

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*.

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

Algoritmo 6: Construção do autômato LR(0)

Entrada: Gramática G
Saida: Autômato LR(0)
início
  Autômato M
  Estado $s0 \leftarrow \{P \mid$ para a produção do símbolo inicial $A \rightarrow \gamma$, P é uma produção da forma $A \rightarrow .\gamma\}$
  Closure($s0$)
  adiciona $s0$ a M
  conjunto de estados $newStates \leftarrow \{s0\}$
  para cada Estado s em $newStates$ **fazer**
    conjunto de estados $temp \leftarrow \varnothing$
    para cada símbolo a em $T \cup NT$ **fazer**
      Estado $s1 \leftarrow \varnothing$
      para cada produção da forma $A \rightarrow \alpha.a\beta$ em s **fazer**
        $s1 \leftarrow s1 \cup \{A \rightarrow \alpha a.\beta\}$
      fim
      Closure($s1$)
      se $s1$ não está em M **então**
        adiciona $s1$ a M
        adiciona a transição $\hat{\delta}(s,a) \rightarrow \hat{\delta}(s,a) \cup \{s1\}$ a M
        $temp \leftarrow temp \cup \{s1\}$
      fim
    fim
    $newStates \leftarrow temp$
  fim
  retornar M
fim

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 $A \rightarrow \alpha.B$ com *lookahead* de $\{L\}$, deverão ser adicionadas novas produções da forma $B \rightarrow .\gamma$ com *lookahead* de $\{L\}$
- Para produções da forma $A \rightarrow \alpha.B\beta$, com *lookahead* de $\{L\}$, deverão ser adicionadas novas produções da forma $B \rightarrow .\gamma$ 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 nos algoritmos 9 e 10 respectivamente.

Algoritmo 7: Construção da tabela SLR

Entrada: Autômato LR(0)
início
  Tabela ACTION
  Tabela GOTO
  para cada Estado s **fazer**
    para cada item da forma $A \to \alpha.a\beta$ **fazer**
      ACTION$[s,a] \leftarrow$ shift para o estado t de acordo com o autômato LR(0)
    fim
    para cada item da forma $A \to \alpha.B\beta$ **fazer**
      GOTO$[s,B] \leftarrow$ goto para o estado t de acordo com o autômato LR(0)
    fim
    para cada item da forma $A \to \alpha.$ **fazer**
      para cada terminal a em Follow$(G)[A]$ **fazer**
        ACTION$[s,a] \leftarrow$ reduce pela regra $A \to \alpha$
      fim
    fim
  fim
  Todos os estados restantes são estados de erro
fim

Algoritmo 8: Analise com tabela SLR

Entrada: Tabela de ações ACTION, Tabela goto GOTO
início
  Pilha de estados P
  monta $s0$ em P onde $s0$ é o estado inicial
  Token $a \leftarrow$ primeiro token da entrada
  enquanto verdade **fazer**
    Estado $s \leftarrow$ topo de P
    se ACTION$[s,a]$ for aceite **então**
      análise completa
    senão, se ACTION$[s,a]$ for shift t **então**
      monta estado t em P
      $a \leftarrow$ próximo token da entrada
    senão, se ACTION$[s,a]$ for reduce $A \to \beta$ **então**
      desmonta estados correspondentes a β de P
      Estado $t \leftarrow$ topo de P
      monta GOTO$[t,A]$ em P
    senão
      para com um erro
    fim
  fim
fim

Algoritmo 9: Closure LR(1)

Entrada: Gramática G, Estado s
início
  repetir
    para cada item da forma $A \rightarrow \alpha.B$ em s com lookahead de $\{L\}$ **fazer**
      $s \leftarrow s \cup \{$produções da forma $B \rightarrow .\gamma\}$
    fim
    para cada item da forma $A \rightarrow \alpha.B\beta$ em s com lookahead de $\{L\}$ **fazer**
      para cada item da forma $B \rightarrow \gamma$ em G **fazer**
        se β não produz ε **então**
          $s \leftarrow s \cup \{$produção da forma $B \rightarrow .\gamma$ com lookahead de $First(G, \beta)\}$
        senão
          $s \leftarrow s \cup \{$produção da forma $B \rightarrow .\gamma$ com lookahead de $First(G, \beta) \cup \{L\}\}$
        fim
      fim
    fim
  até que não tenha itens a serem adicionados
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 dirigidos a estado (*declarative state-driven*) o que 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

Algoritmo 10: Construção do autômato LR(1)

Entrada: Gramática G
Saida: Autômato LR(1)
início
  Autômato M
  Estado $s0 \leftarrow \{P \mid$ para a produção do símbolo inicial $A \rightarrow \gamma$, P é uma produção da forma $A \rightarrow .\gamma$ com *lookahead* de $\{\$\}$ $\}$
  Closure($s0$)
  adiciona $s0$ a M
  conjunto de estados $newStates \leftarrow \{s0\}$
  para cada estado s em $newStates$ **fazer**
    conjunto de estados $temp \leftarrow \varnothing$
    para cada Símbolo a em $T \cup NT$ **fazer**
      Estado $s1 \leftarrow \varnothing$
      para cada produção da forma $A \rightarrow \alpha.a\beta$ com *lookahead* de $\{L\}$ em s **fazer**
        $s1 \leftarrow s1\ cup$ {produção da forma $A \rightarrow \alpha a.\beta$ com *lookahead* de $\{L\}$}
      fim
      Closure($s1$)
      se $s1$ não está em M **então**
        adiciona $s1$ a M
        adiciona a transição $\hat{\delta}(s,a) \rightarrow \hat{\delta}(s,a) \cup \{s1\}$ a M
        $temp \leftarrow temp \cup \{s1\}$
      fim
    fim
    $newStates \leftarrow temp$
  fim
  retornar M
fim

outros *frameworks* como *React* que funciona em *runtime* e tem sua sintaxe limitada a isso sendo necessárias 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 a variável assim como na sintaxe *Javascript*, em *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 e foi usada 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 podem 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.

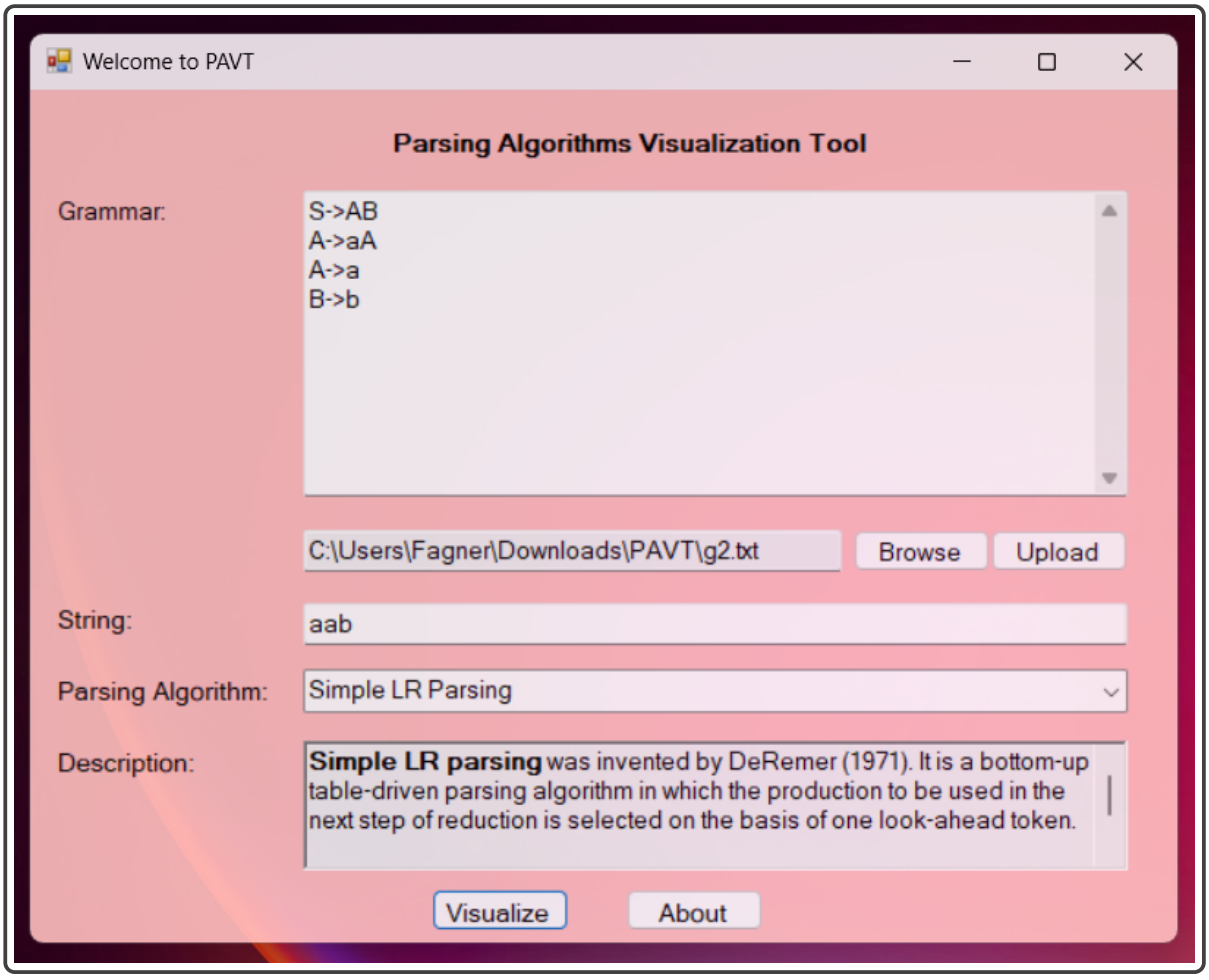
Após os dados da utilização da aplicação serem analisados, os autores concluíram que os testes gerados automaticamente têm dificuldade e resultados semelhantes aos obtidos com os testes criados por professores. Os autores também afirmam que os testes gerados automaticamente têm a vantagem de sempre serem diferentes 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 com o objetivo de ensinar seis algoritmos de análise sintática. Os algoritmos que são 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) par-*

sing. 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 ou 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 que são 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 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 com o objetivo de criar uma versão *web* de um sistema de simulação ComVis, um sistema 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.

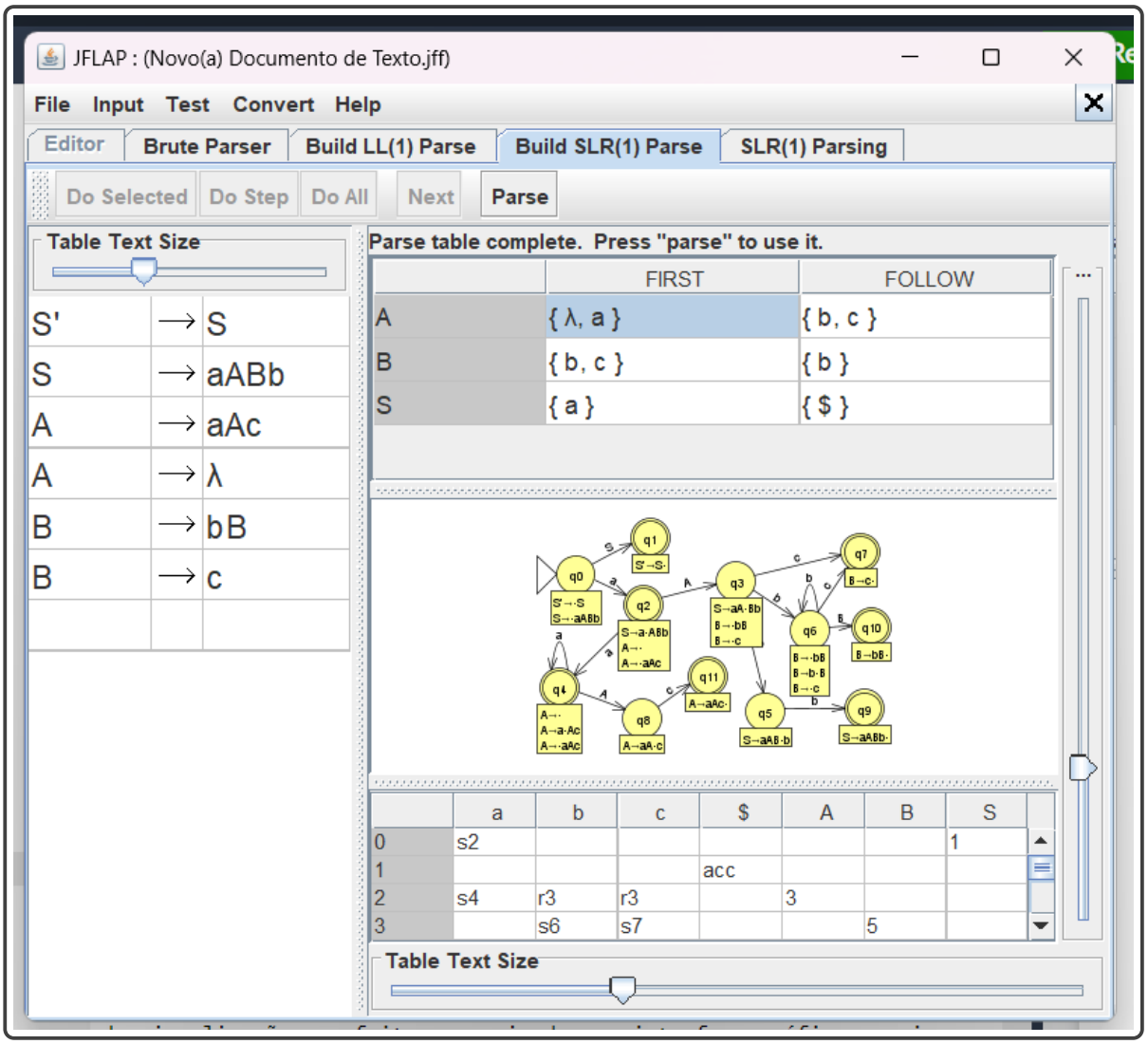
O sistema foi desenvolvido usando *Java Server Page*, já que a versão *desktop* do sistema foi feita em Java, grande parte da base de código pôde ser reutilizada dessa forma. Outras tecnologias usadas foram HTML, CSS, JavaScript e Graphviz para criação de gráficos e diagramas.

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. Por fim, nenhuma das ferramentas apresenta uma versão *mobile*. 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. No Quadro 1 pode-se ver o resumo do comparativo dos trabalhos citados anteriormente.

Quadro 1 – Comparativo de trabalhos relacionados

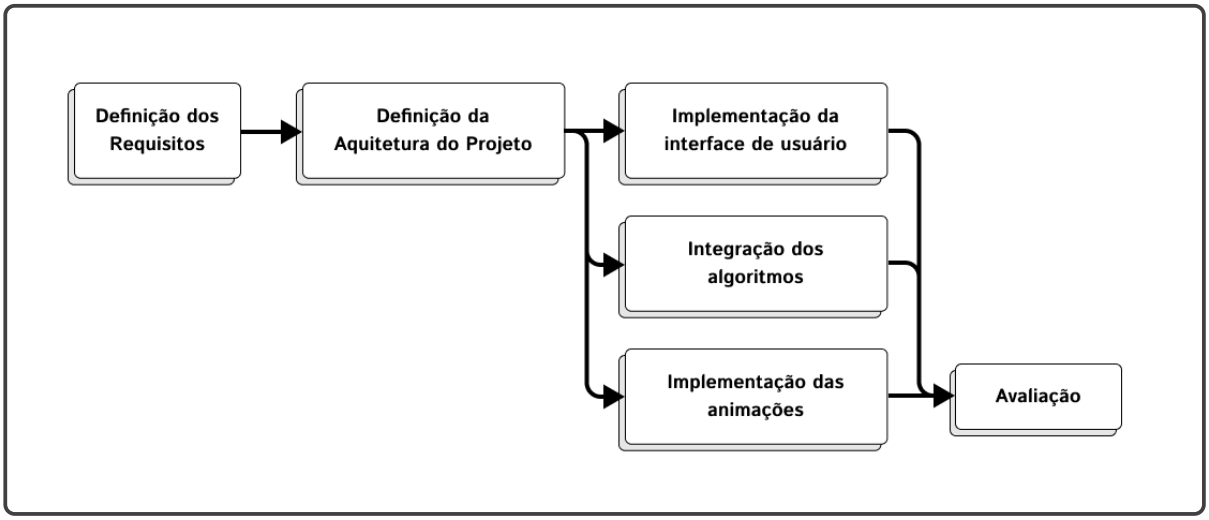
Trabalho	Algoritmos			Plataformas			Integração com Moodle
	LL(1)	SLR	CLR	Mobile	Desktop	Web	
Muñoz *et al.* (2024)	x	x				x	
Sangal, Kataria, Tyagi *et al.* (2018)		x	x		x		
Stamenković e Jovanović (2024)		x	x		x	x	
Devakumar, Naik e Sajja (2014)	x	x			x		
Este trabalho	x	x	x	x	x	x	x

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. 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.

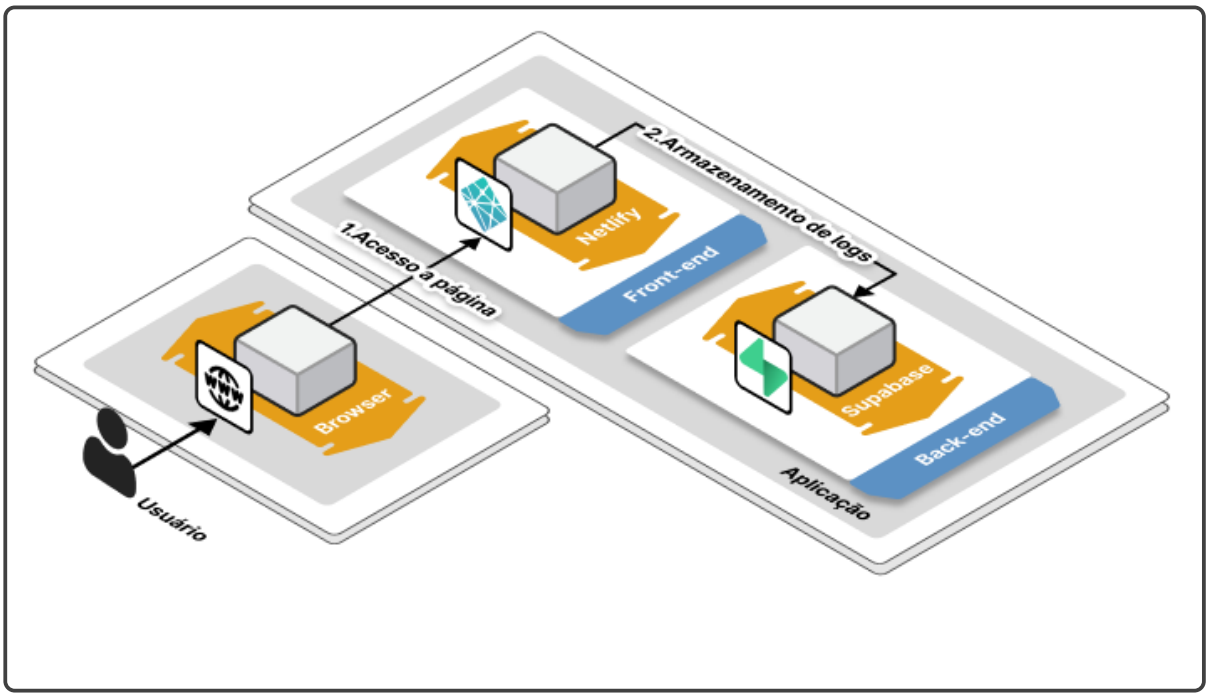
Apesar das ferramentas compartilharem as mesmas funcionalidades básicas já definidas inicialmente, outras têm características interessantes que podem ser reaproveitadas, partir delas foram definidos os seguintes requisitos:

- 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 que seja 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* que é 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 na qual 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 database 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.

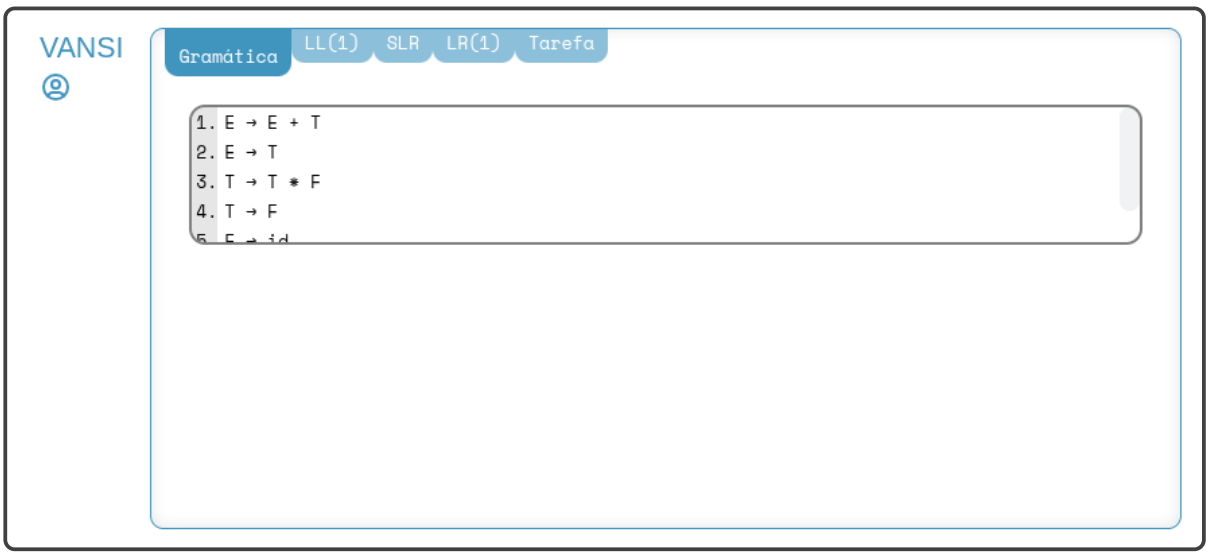
4.3 Implementação da interface de usuário

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 butões separados em dois grupos. No grupo da esquerda, os dois primeiros butões abrem respectivamente uma jenela de diálogo com os resultados em formato de texto e uma janela de diálogo com uma descrição do algoritmo. Os outros dois butões 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 butões da direita estão os controles de fluxo da execução do algoritmo. O primeiro butão 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



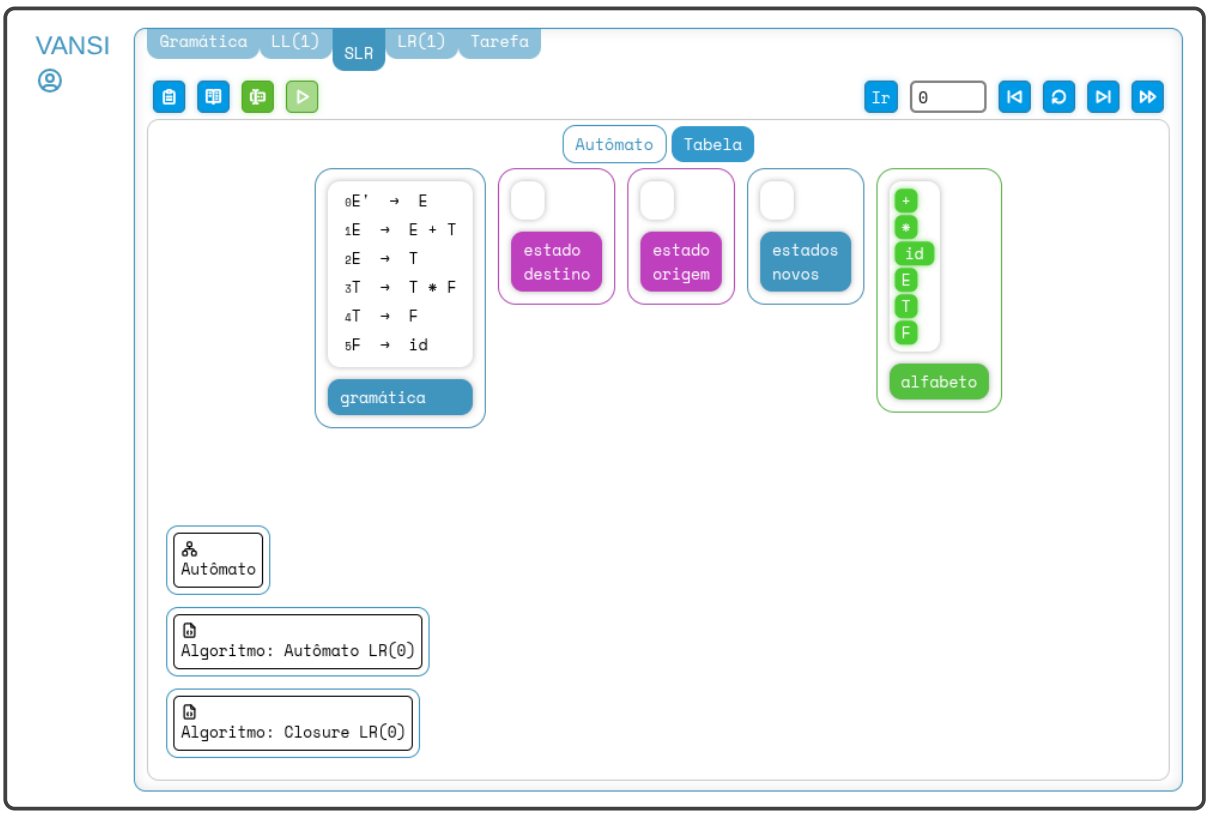
Fonte: fornecida pelo próprio autor

para o passo da execução digitado no campo de texto ao lado desse butão, os demais butões 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 butões, cada um deles abre a visualização de um algoritmo secundário usado na construção do parser selecionado e abaixo desses butões 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 icóne e título descrevendo a janela. Dentro dessas janelas, no canto superior direito existe uma bandeja de ações com alguns butões. O primeiro butão 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. Por fim, nos cantos das janelas flutuantes existem pequenos quadrados que servem para redimensionar a janela. Os elementos descritos nesse parágrafo podem ser vistos na Figura 8.

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.

Figura 7 – Aba de visualização do algoritmo SLR



Fonte: fornecida pelo próprio autor

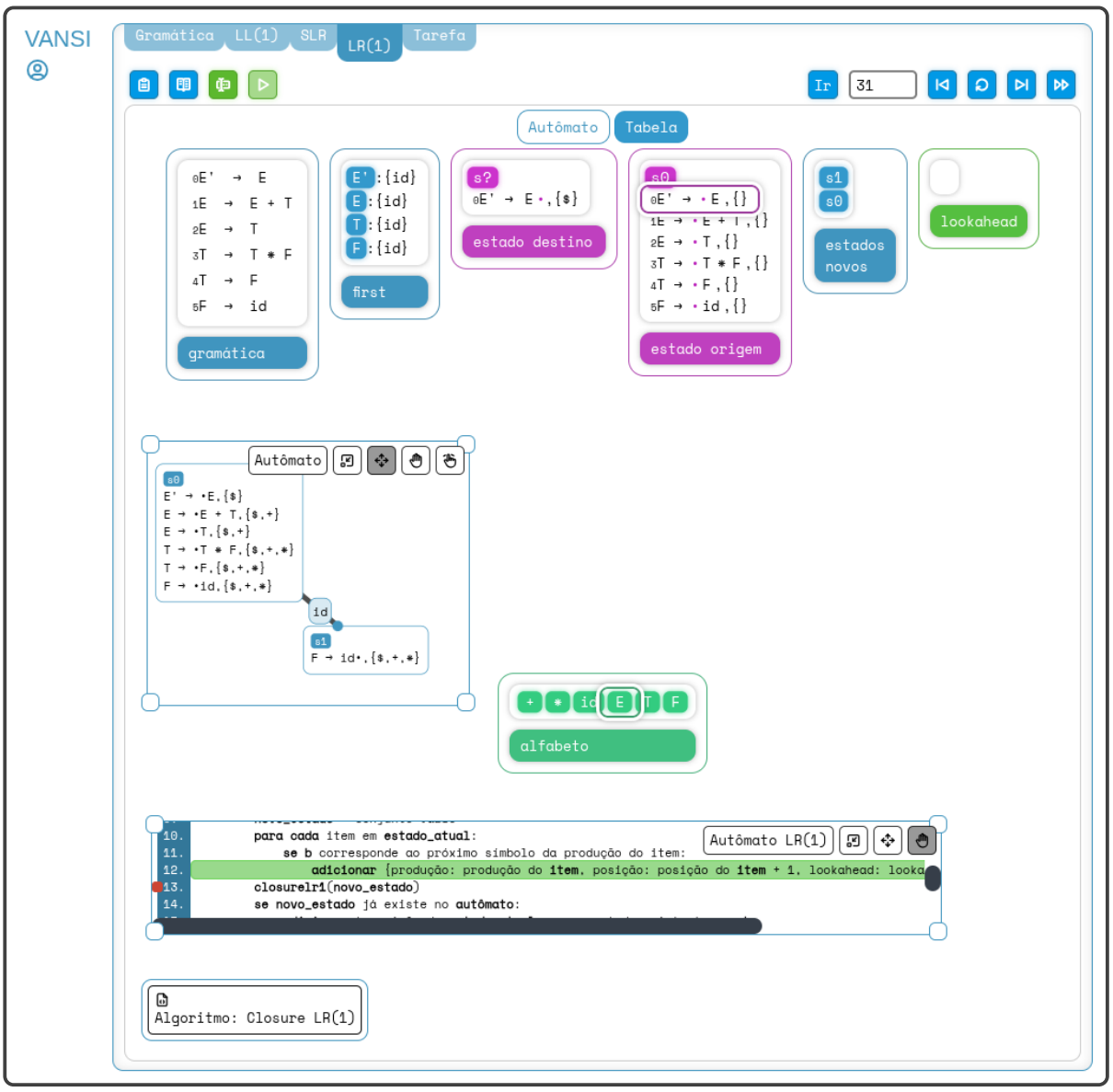
As janelas de diálogo aparecem contidas na área de visualização da execução dos algoritmos. Acima de cada janela está um botão de fechar que cobre o comprimento inteiro da janela. Dentro da janela de diálogo para copiar os resultados dos algoritmos em formato de texto estão os resultados de cada algoritmo divididos em seções e ao lado do título de cada seção está um butão 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.

Nas abas de visualização de algoritmos serão inclusos um campo de texto no qual o usuário poderá inserir uma *string* de entrada para ser analisada pelo algoritmo, um campo de texto para copiar os resultados dos algoritmos em forma de texto e um campo de texto para copiar a implementação do algoritmo.

4.4 Integração dos algoritmos à ferramenta

Para todas as estruturas de dados usadas nos algoritmos serão criadas representações visuais, dessa forma todos os passos do funcionamento poderão ser representados como estados

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



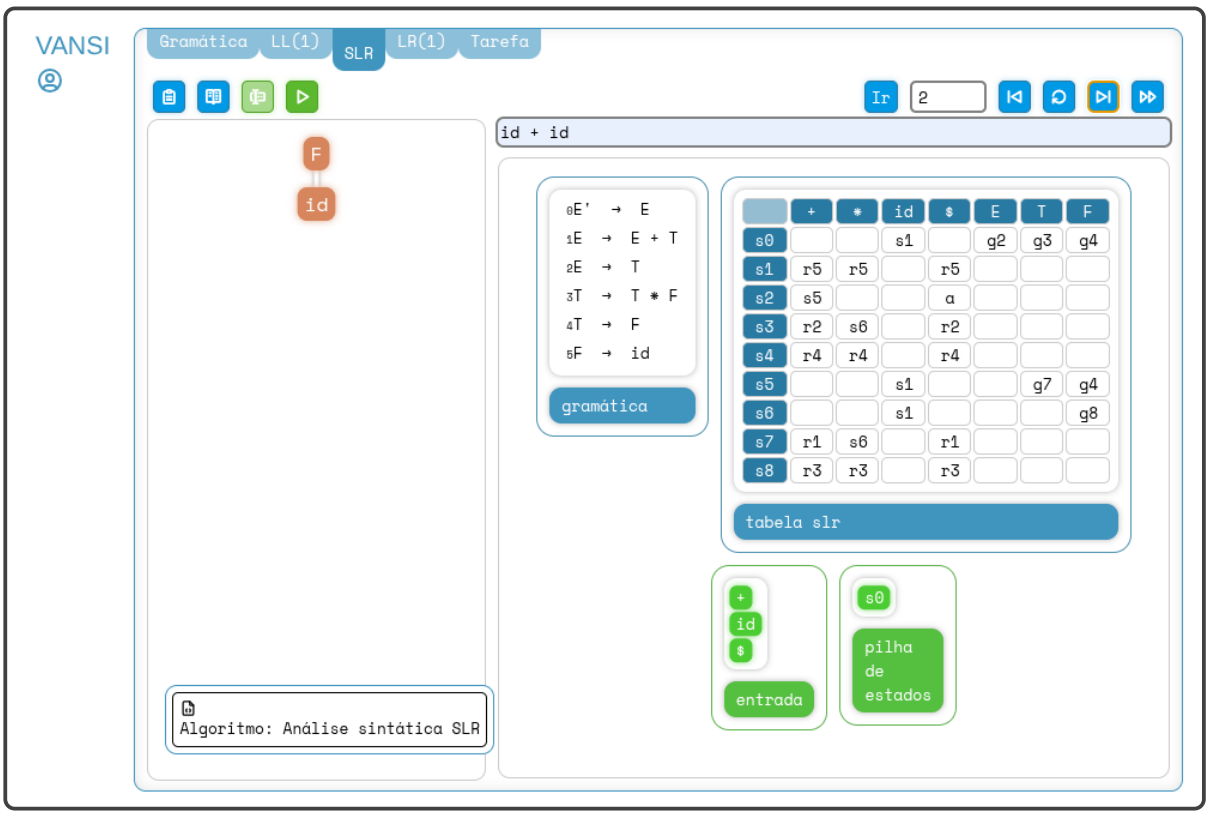
Fonte: fornecida pelo próprio autor

dessas estruturas. Calculando antecipadamente os estados dessas estruturas em cada passo dos algoritmos podemos fazer um controle de fluxo entre os passos dos algoritmos.

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

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



Fonte: fornecida pelo próprio autor

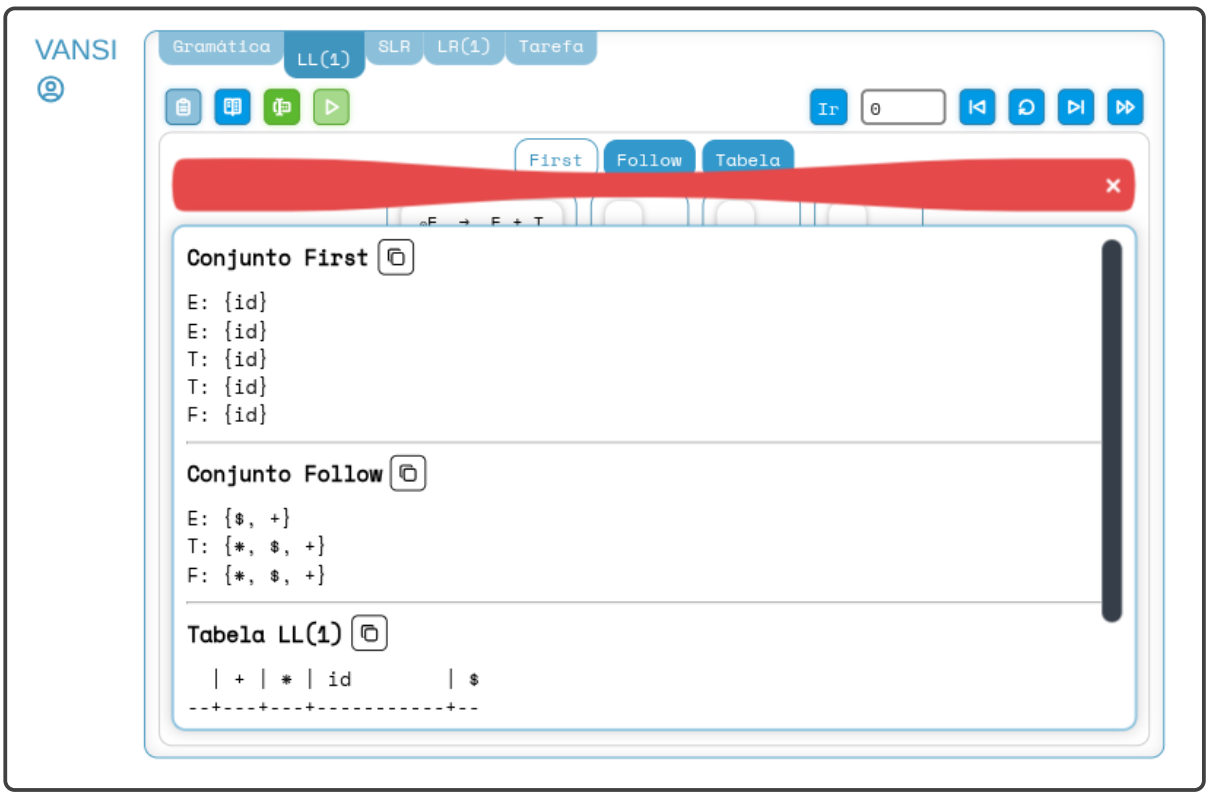
mais dinâmica. Um exemplo de animação é a animação do estado da estrutura de pilha que é usada em alguns algoritmos. Quando um item é adicionado ou removido da pilha, o elemento visual que representa esse item terá sua posição interpolada do ponto inicial ao ponto final.

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 meio de questionários e entrevistas para avaliar a percepção dos estudantes sobre a eficácia da ferramenta. 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.

O desenvolvimento desse trabalho seguirá o cronograma mostrado no Quadro 2.

Figura 10 – Janela de diálogo para copiar resultados



Fonte: fornecida pelo próprio autor

Quadro 2 – Cronograma

Atividades	Período					
	2024			2025		
	Outubro	Novembro	Dezembro	Janeiro	Fevereiro	Março
Definir dos requisitos	x					
Definir arquitetura do projeto	x					
Implementar da interface de usuário		x	x			
Integrar os algoritmos			x	x	x	
Implementar as animações			x	x	x	
Integrar ao Moodle				x	x	
Avaliar a ferramenta					x	
Escrita do TCC II	x	x	x	x	x	x
Defesa do TCC II						x

Fonte: fornecido pelo autor

GLOSSÁRIO

DOM *Document Object Model*

SSR *Server Side Rendering*

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.