UNIVERSIDADE FEDERAL DO CEARÁ

CURSO DE GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

FRANCISCO FAGNER FERREIRA MESQUITA

UMA FERRAMENTA DE VISUALIZAÇÃO DE ALGORITMOS DE ANÁLISE SINTÁTICA

QUIXADÁ

2024

LISTA DE ILUSTRAÇÕES

LISTA DE QUADROS

LISTA DE ALGORITMOS

LISTA DE ABREVIATURAS E SIGLAS

AGS	*Assignment and Grade Services* 2.0
API	*Application Programming Interface*
DL	*Deep Linking* 2.0
DOM	*Document Object Model*
LMS	*Learning Management System*
LTI	*Learning Tools Interoperability*
NRPS	*Names and Role Provisioning Services* 2.0
SSR	*Server Side Rendering*
UFC	Universidade Federal do Ceará

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 checagem de tipo, 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 e 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).

Ferramentas criadas para o ensino de conteúdos sobre construção de compiladores como análise sintática usando elementos visuais têm uma resposta positiva dos alunos que usaram as ferramentas. Essas ferramentas podem auxiliar na compreensão do conteúdo não só através das instruções mostradas na visualização do funcionamento dos algoritmos, mas também por oferecer respostas instantâneas que podem ser usadas pelos estudantes como correção sobre os resultados dos algoritmos que podem ser difíceis de se construir manualmente (JAIN; GOYAL; CHAKRABORTY, 2017).

A instabilidade na rede disponível no campus Quixadá da Universidade Federal do Ceará (UFC) é algo recorrente que pode atrapalhar o roteiro normal das aulas e impedir que os alunos concluam tarefas propostas em aula (PEREZ, 2023). O uso de aplicações desenvolvidas para *web* nas aulas é afetado por eventuais falhas na rede local do campus e em relação a isso uma aplicação *offline* desenvolvida para *desktop* tem a vantagem de poder ser acessada independente do acesso à *internet* (HOLZER; ONDRUS, 2012). Utilizar o servidor local do

campus para hospedar a ferramenta também pode melhorar a disponibilidade do *software* já que o servidor local do campus sofre instabilidade que a rede local.

Muitos alunos entram na vida universitária estando em situação de vulnerabilidade econômica e sem recursos necessários como computadores para acompanhar o conteúdo do curso, algo que faz alusão a essa realidade é a disponibilização de bolsas de inclusão feita pela UFC durante o período da pandemia para auxiliar na aquisição de computadores (OLIVEIRA, 2020). Assim, a utilização de uma aplicação *mobile* no lugar de uma aplicação *desktop* nas aulas seria mais inclusiva.

Apesar da desvantagem citada nos parágrafos anteriores, uma ferramenta *web* tem outras vantagens notáveis como não haver a necessidade de instalação do *software* para acessá-lo e a facilidade de integração com plataformas como o *Moodle*[1] (DESAI, 2023).

Todas as abordagens de desenvolvimento citadas têm suas vantagens e desvantagens, mas não é preciso escolher uma abordagem em detrimento da outra. *Frameworks* modernos como *Tauri*[2] e *Capacitor*[3] permitem que uma única base de código seja usada para desenvolver *software* para diferentes plataformas, graças a isso, torna-se possível o desenvolvimento multi-plataforma da ferramenta proposta nesse trabalho (SHEVTSIV; STRIUK, 2021).

1.1 Objetivos

O objetivo desse trabalho é criar uma ferramenta multi-plataforma que possa ser acessada *offline* e *online* sendo hospedada local e remotamente e que por meio de uma mistura de elementos visuais e textuais ajude a entender como funcionam os algoritmos de análise sintática e quais os processos necessários para obter a saída de cada passo dos algoritmos.

Esse trabalho tem os seguintes objetivos específicos:

- Criar a ferramenta usando *Svelte*[4].
- Oferecer uma forma de visualização dos algoritmos CLR, SLR e LL(1).
- Oferecer uma versão *web* da ferramenta
- Oferecer uma versão *desktop* da ferramenta
- Oferecer uma versão *mobile* da ferramenta
- Integração da ferramenta com a plataforma *Moodle*.

[1] https://moodle2.quixada.ufc.br/
[2] https://tauri.app/
[3] https://capacitorjs.com/
[4] https://svelte.dev/

- Validar a ferramenta fazendo uma avaliação com alunos.

2 FUNDAMENTAÇÃO TEÓRICA

Neste capítulo apresentaremos os conceitos centrais que serviram como base e guia para a elaboração deste trabalho.

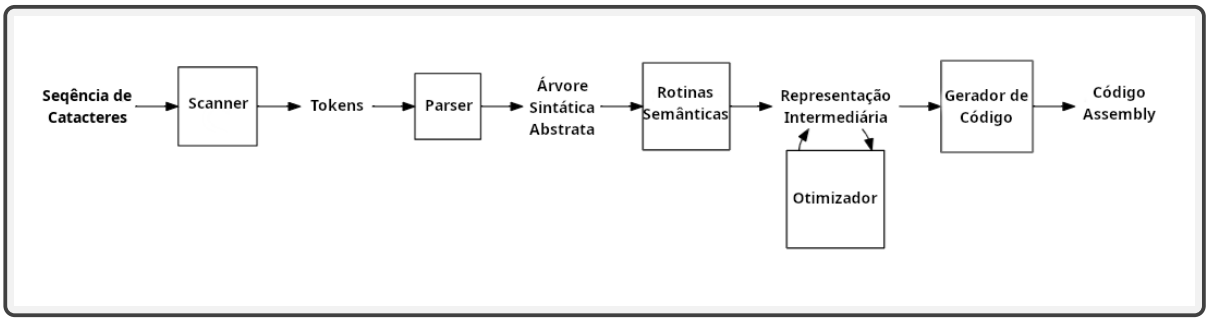
2.1 Compilador

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, 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.

2.1.1.2 Análise Sintática

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

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) além de não seguirem 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, esse analisador sintático leva 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).

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.

Algoritmo 1: First

Entrada: Gramática G, simbolo X
Saida: Conjunto first
início
  first = {}
  selecionar *X* **fazer**
    caso *Terminal* **fazer**
      first = {X}
    caso *Não terminal* **fazer**
      repetir
        para cada *regra* $X \rightarrow Y_1Y_2...Y_k$ *na Gramática G* **fazer**
          se *a está em First(*Y_1*) ou a está em First(*Y_n*) e* $Y_1..Y_{n-1} \Rightarrow \varepsilon$ **então**
            Adicionar a ao first
          fim
          se $Y_1...Y_k \Rightarrow \varepsilon$ **então**
            Adicionar ε ao first
          fim
        fim
      até que *não haja mais mudanças*;
    fim
  fim
  retornar *first*
fim

Algoritmo 2: Follow

Entrada: Gramática G
Saida: Conjunto follow
início
  follow = { }
  follow(S) = {$} onde S é o símbolo inicial.
  repetir
    se $A \rightarrow \alpha B \beta$ **então**
      adiciona First(β) (exceto ε) follow(B).
    fim
    se $A \rightarrow \alpha B$ *ou First(*β*) contém* ε **então**
      adiciona follow(A) ao follow(B).
    fim
  até que *até não houver mais 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.

Algoritmo 3: Construção da tabela LL(1)

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

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 4: Analise com tabela LL(1)

```
Entrada: Gramática G com simbolo inicial P, tabela T
início
    cria uma stack S.
    monta $ e P em S.
    token c = o primeiro token na entrada.
    enquanto S não está vazio fazer
        token X = o topo de S.
        se X faz par com c então
            remova X de S.
            avança c para o proximo token
            repetir.
        fim
        se X é um terminal então
            para com um erro.
        fim
        se T [X, c] aponta para a regra X → α então
            remova X de S.
            monta os símbolos α em S.
            repetir.
        fim
        se T [X, c] aponta para um estado de erro então
            para com um erro.
        fim
    fim
fim
```

2.3 Analisadores Sintáticos Ascendentes

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.

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(1). 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 6 mostra como construir o autômato LR(0) com o auxílio do *closure* mostrado no Algoritmo 5.

Algoritmo 5: Closure

Entrada: Gramática G, Estado S
início
  repetir
    para cada *item da forma* $A \rightarrow \alpha.X\beta$ *em S com* $X \in NT$ **fazer**
      para cada *produção da forma* $X \rightarrow \gamma$ *em G* **fazer**
        adiciona uma produção da forma $X \rightarrow .\gamma$ 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 isso 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.

Algoritmo 6: Construção do autômato LR(0)

Entrada: Gramática G
Saida: Autômato LR(0)
início
    cria um autômato m
    estado s0 = {P | 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 = {s0}
    para cada *estado s em newStates* **fazer**
        conjunto de estados temp = { }
        para cada *símbolo a em* $T \cup NT$ **fazer**
            estado s1 = { }
            para cada *produção da forma* $A \rightarrow \alpha.a\beta$ *em s* **fazer**
                adiciona uma produção da forma $A \rightarrow \alpha a.\beta$ em s1
            fim
            closure(s1)
            se *s1 não está em m* **então**
                adiciona s1 a m
                adiciona a transição $\hat{\delta}(s,a) = \hat{\delta}(s,a) \cup \{s1\}$ a m
                adiciona s1 a temp
            fim
        fim
        newStates = temp
    fim
    retornar *m*
fim

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 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] = 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] = 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(A)* **fazer**
                ACTION[s, a] = 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
    stack de estados S.
    monta S0 em S.
    token a = primeiro token da entrada.
    enquanto *verdade* **fazer**
        estado s = topo de S.
        se *ACTION[s, a] for aceite* **então**
            analise completa.
        else if *ACTION[s, a] for shift t* **then**
            monta estado t em S.
            token a = próximo token da entrada.
        else if *ACTION[s, a] for reduce* $A \to \beta$ **then**
            desmonta estados correspondentes a β de S.
            estado t = topo de S.
            monta GOTO[t, A] em S.
        senão
            para com um erro.
        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*.

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 $A \rightarrow \alpha.B$ 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 $First(\beta)$.
  - Se β produz ε, o *lookahead* é $First(\beta) \cup \{L\}$.

2.4 *Learning Tools Interoperability*

Learning Tools Interoperability (LTI) em português interoperabilidade de ferramentas de aprendizagem é um padrão técnico desenvolvido pela 1EdTec[1]. Esse padrão especifica métodos de comunicação entre *Learning Management System* (LMS) em português sistema de gerenciamento de aprendizagem e ferramentas de aprendizagem remotas (1EDTECH, 2024).

O padrão LTI permite a implementação das seguintes funcionalidades:

- *Assignment and Grade Services* 2.0 (AGS) que fornece uma maneira de criar uma coluna de boletim de notas e publicar notas associadas a um *link* de recurso.
- *Names and Role Provisioning Services* 2.0 (NRPS) que fornece acesso a dados sobre usuários e suas funções nas organizações; uma escola, plataforma LMS ou curso são exemplos de organização.
- *Deep Linking* 2.0 (DL) que permite que um professor ou usuário de plataforma LMS integre conteúdo coletado de uma ferramenta externa. Usando este serviço, os usuários

[1] https://www.1edtech.org/

da plataforma podem lançar um URI especificado pelo fornecedor do currículo digital (ferramenta externa), selecionar conteúdo específico e, em seguida, receber um URI que outros usuários podem usar para lançar diretamente esse conteúdo.

Além disso, o LTI oferece os seguintes serviços adicionais:

- *Dynamic Registration* que é uma forma de automatizar a troca de informação de registro de entre plataformas e ferramentas.
- *Submission Review Service* fornece uma maneira padrão para um instrutor ou aluno voltar do boletim de notas de uma plataforma para a ferramenta onde a interação ocorreu para exibir o envio do aluno para um item de linha específico.
- *Course Groups Service* que comunica para uma ferramenta os grupos disponíveis nos cursos e suas matrículas.

2.5 *Svelte*

Svelte é um *framework* de componentes criado em 2016 por Harry Rich, é 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 dá mais trabalho 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 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).

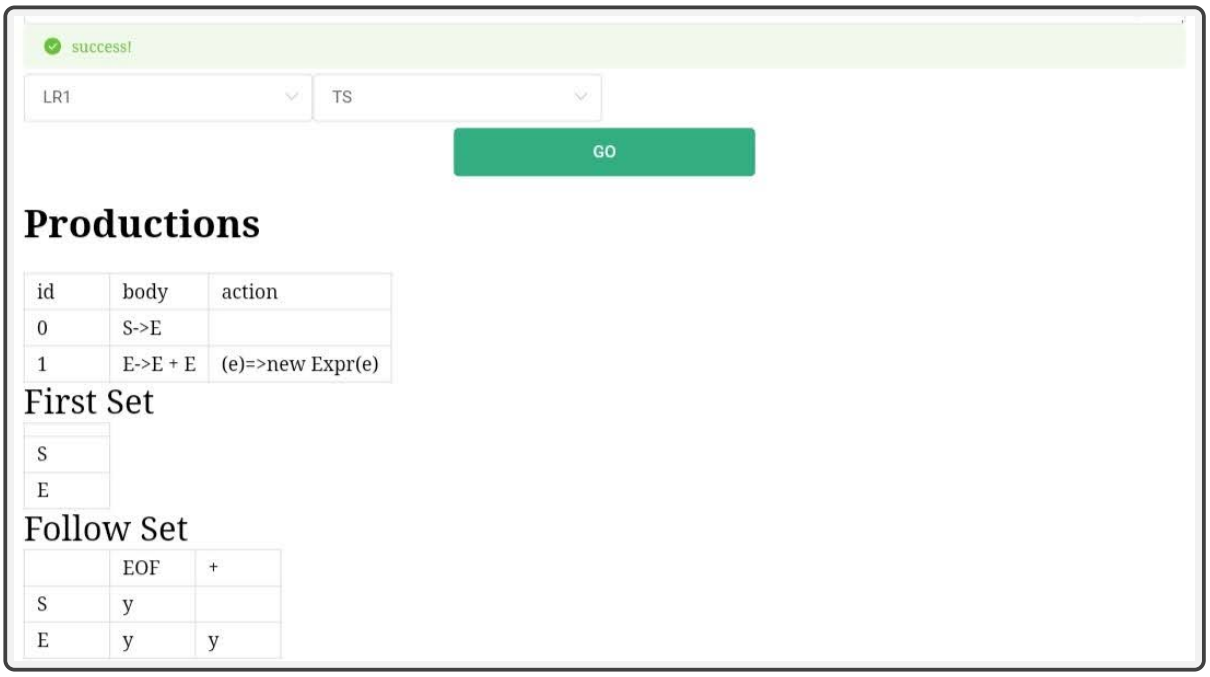
3 TRABALHOS RELACIONADOS

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

3.1 *Parser Generator Web Tools*

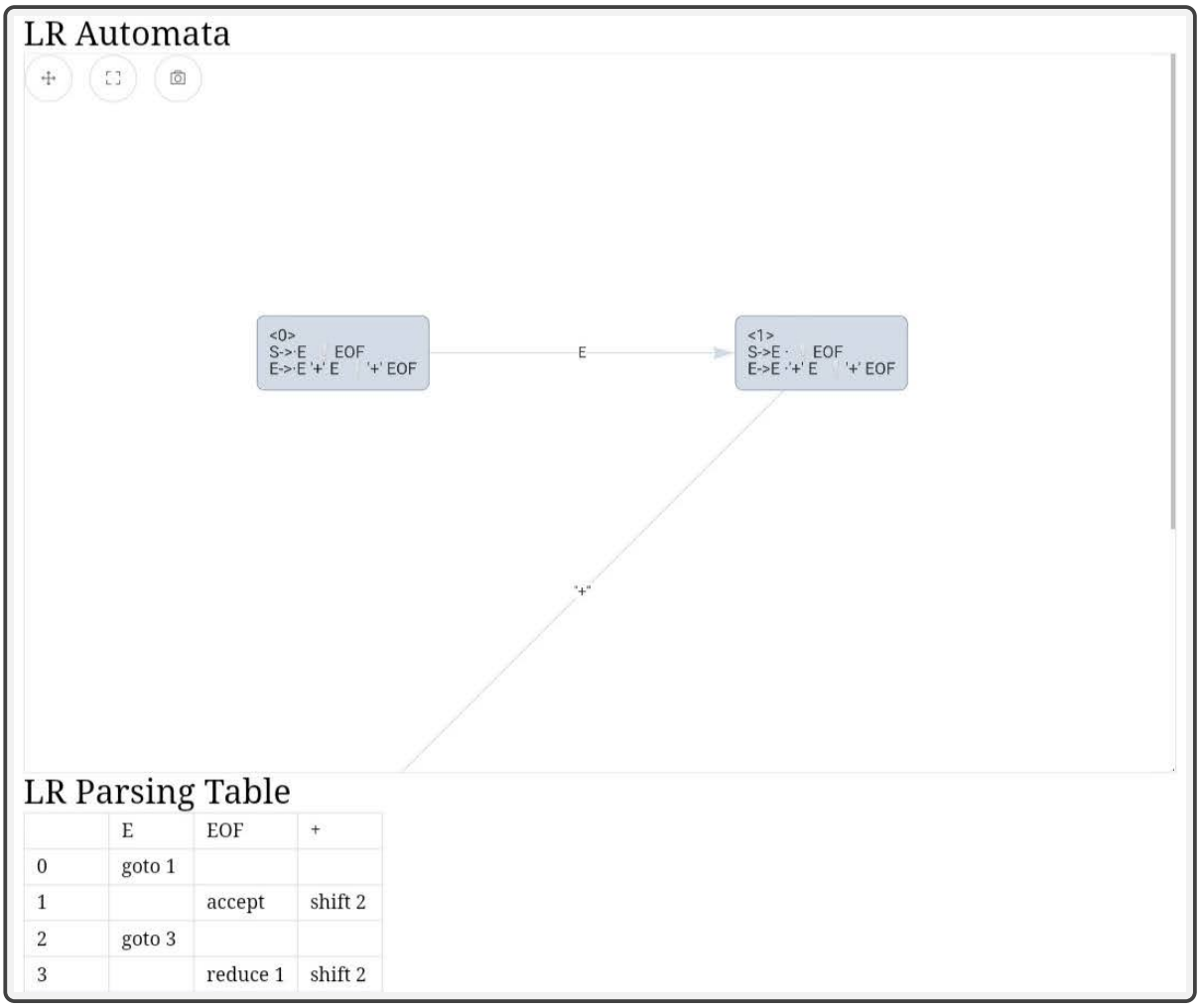
Essa ferramenta desenvolvida por Light (2024) é uma aplicação *web* que oferece a visualização de três algoritmos, LL(1), SLR(1) e CLR(1). Ao ser informada uma gramática pela caixa de entrada, a ferramenta cria a tabela dos conjuntos *first* e *follow* como mostra a Figura 2, um autômato dos estados do analisador sintático e a tabela de transição de estados como mostra a Figura 3.

Figura 2 – Imagem da ferramenta de *Parser Generator Web Tools*



Fonte: Light (2024).

Figura 3 – Imagem da ferramenta de *Parser Generator Web Tools*

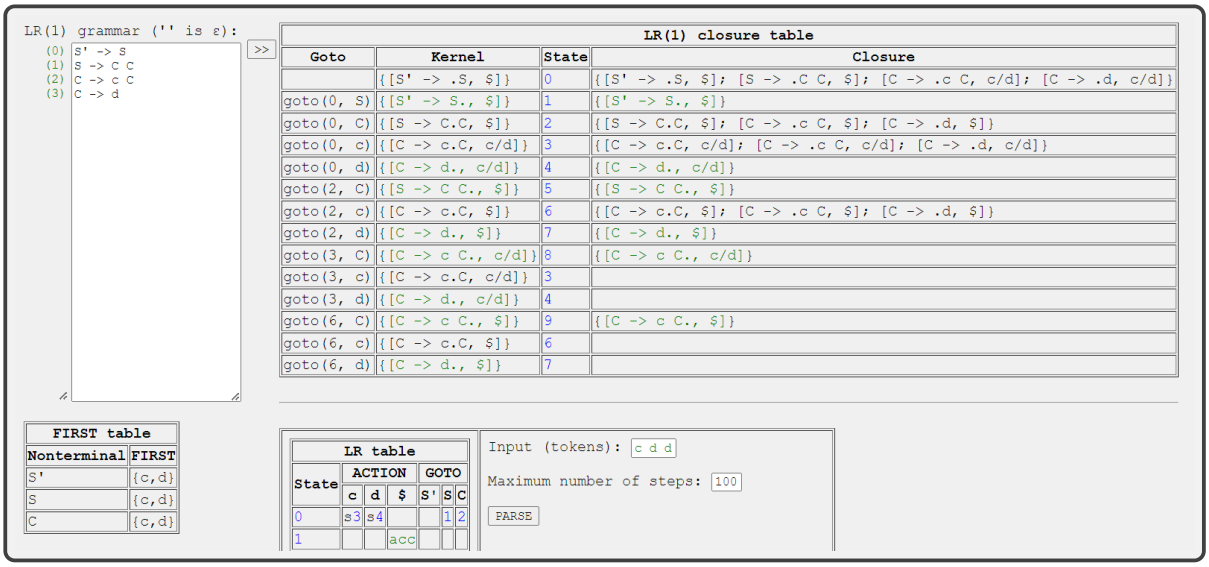


Fonte: Light (2024).

3.2 LR(1) *Parser Generator*

Feita especificamente para o algoritmo CLR(1) a ferramenta *web* criada por Apou (2024) gera o conjunto de *first* mostrado na Figura 4, o conjunto de itens canônicos mostrado na Figura 4 e a tabela de transição de estados mostrada na Figura 5. A ferramenta também disponibiliza o passo a passo da análise de uma *string* junto com uma árvore sintática representada por contêineres contidos um dentro do outro como mostra a Figura 5.

Figura 4 – Imagem da ferramenta de LR(1) *Parser Generator*



LR(1) grammar ('' is ε):

```
(0) S' -> S
(1) S -> C C
(2) C -> c C
(3) C -> d
```

>>

LR(1) closure table

Goto	Kernel	State	Closure
	{[S' -> .S, $]}	0	{[S' -> .S, $]; [S -> .C C, $]; [C -> .c C, c/d]; [C -> .d, c/d]}
goto(0, S)	{[S' -> S., $]}	1	{[S' -> S., $]}
goto(0, C)	{[S -> C.C, $]}	2	{[S -> C.C, $]; [C -> .c C, $]; [C -> .d, $]}
goto(0, c)	{[C -> c.C, c/d]}	3	{[C -> c.C, c/d]; [C -> .c C, c/d]; [C -> .d, c/d]}
goto(0, d)	{[C -> d., c/d]}	4	{[C -> d., c/d]}
goto(2, C)	{[S -> C C., $]}	5	{[S -> C C., $]}
goto(2, c)	{[C -> c.C, $]}	6	{[C -> c.C, $]; [C -> .c C, $]; [C -> .d, $]}
goto(2, d)	{[C -> d., $]}	7	{[C -> d., $]}
goto(3, C)	{[C -> c C., c/d]}	8	{[C -> c C., c/d]}
goto(3, c)	{[C -> c.C, c/d]}	3	
goto(3, d)	{[C -> d., c/d]}	4	
goto(6, C)	{[C -> c C., $]}	9	{[C -> c C., $]}
goto(6, c)	{[C -> c.C, $]}	6	
goto(6, d)	{[C -> d., $]}	7	

FIRST table

Nonterminal	FIRST
S'	{c,d}
S	{c,d}
C	{c,d}

LR table

State	ACTION			GOTO		
	c	d	$	S'	S	C
0	s3	s4			1	2
1			acc			

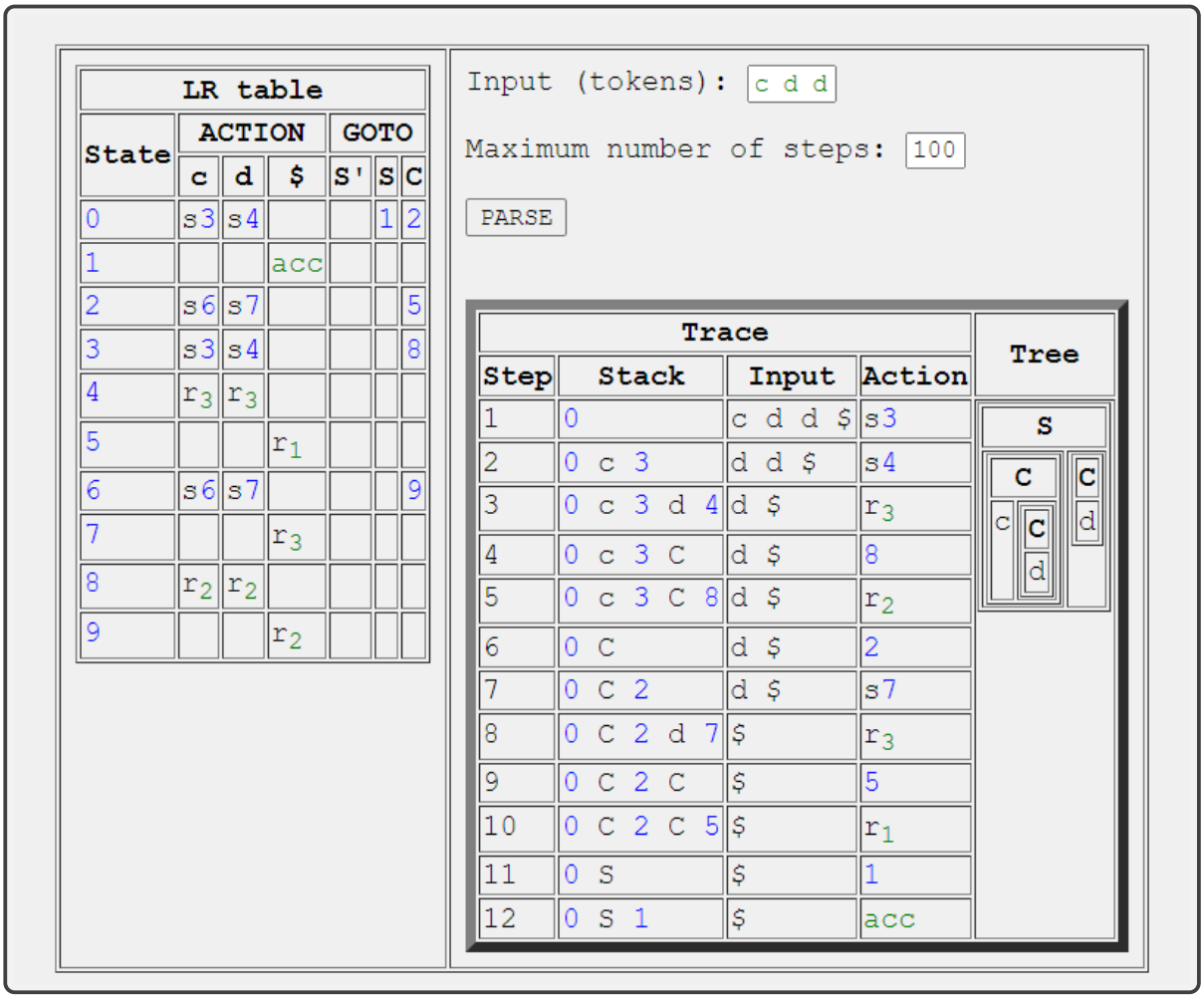
Input (tokens): c d d

Maximum number of steps: 100

PARSE

Fonte: Apou (2024).

Figura 5 – Imagem da ferramenta de LR(1) *Parser Generator*



LR table

State	ACTION			GOTO		
	c	d	$	S'	S	C
0	s3	s4			1	2
1			acc			
2	s6	s7				5
3	s3	s4				8
4	r_3	r_3				
5			r_1			
6	s6	s7				9
7			r_3			
8	r_2	r_2				
9			r_2			

Input (tokens): c d d

Maximum number of steps: 100

PARSE

Trace

Step	Stack	Input	Action
1	0	c d d $	s3
2	0 c 3	d d $	s4
3	0 c 3 d 4	d $	r_3
4	0 c 3 C	d $	8
5	0 c 3 C 8	d $	r_2
6	0 C	d $	2
7	0 C 2	d $	s7
8	0 C 2 d 7	$	r_3
9	0 C 2 C	$	5
10	0 C 2 C 5	$	r_1
11	0 S	$	1
12	0 S 1	$	acc

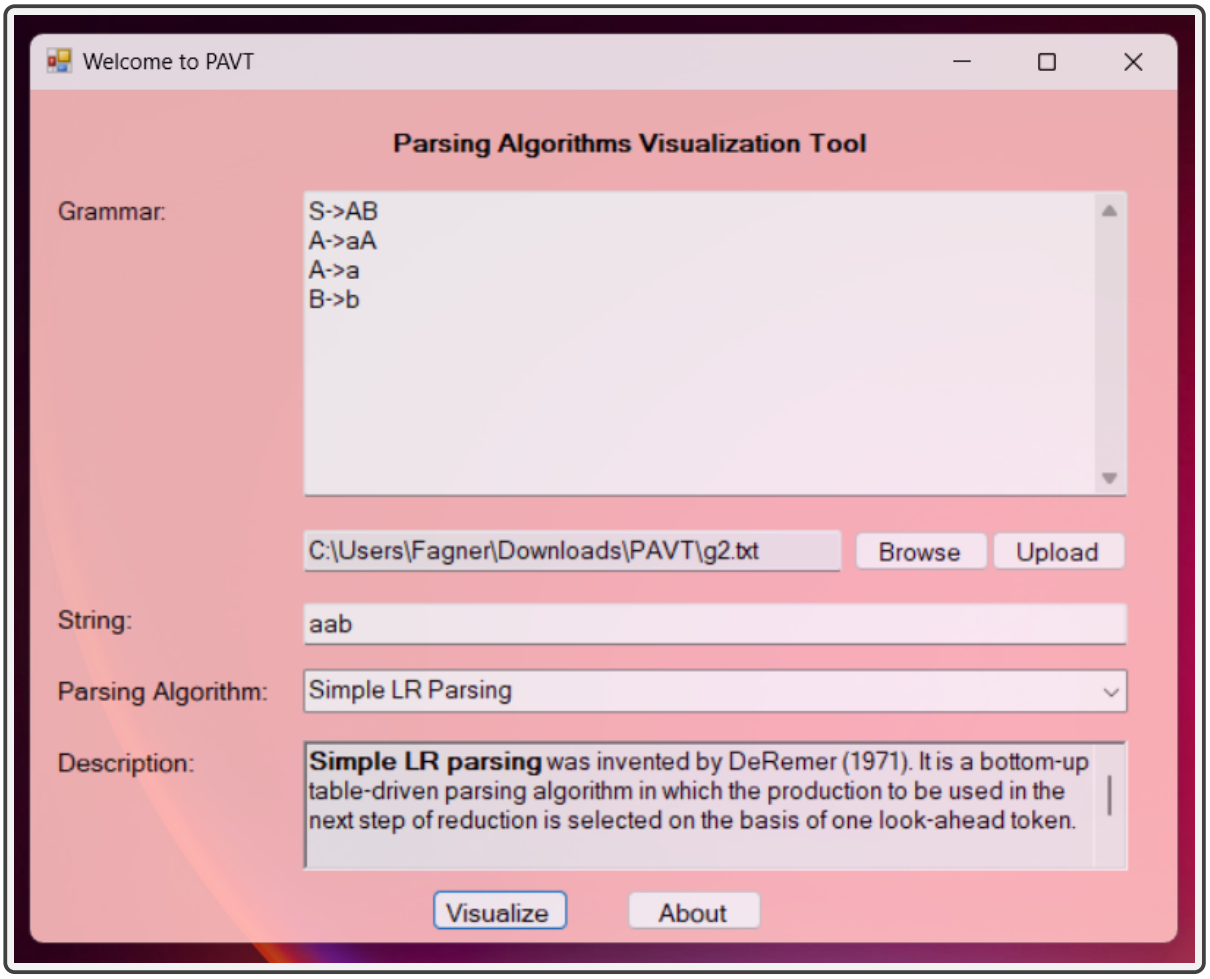
Tree

Fonte: Apou (2024).

3.3 PAVT

No trabalho de Sangal, Kataria, Tyagi *et al.* (2018) foi criada a ferramenta PAVT (*Parsing Algorithms Visualization Tool*) para visualização de seis algoritmos de análise sintática, no entanto, a visualização é realizada apenas pela leitura de um arquivo de texto gerado pela ferramenta, além de não mostrar instruções de como chegar ao resultado descrito no arquivo de texto. O *software* também está apenas disponível em versão *desktop* para o sistema operacional *Windows*.

Figura 6 – Imagem da ferramenta PAVT

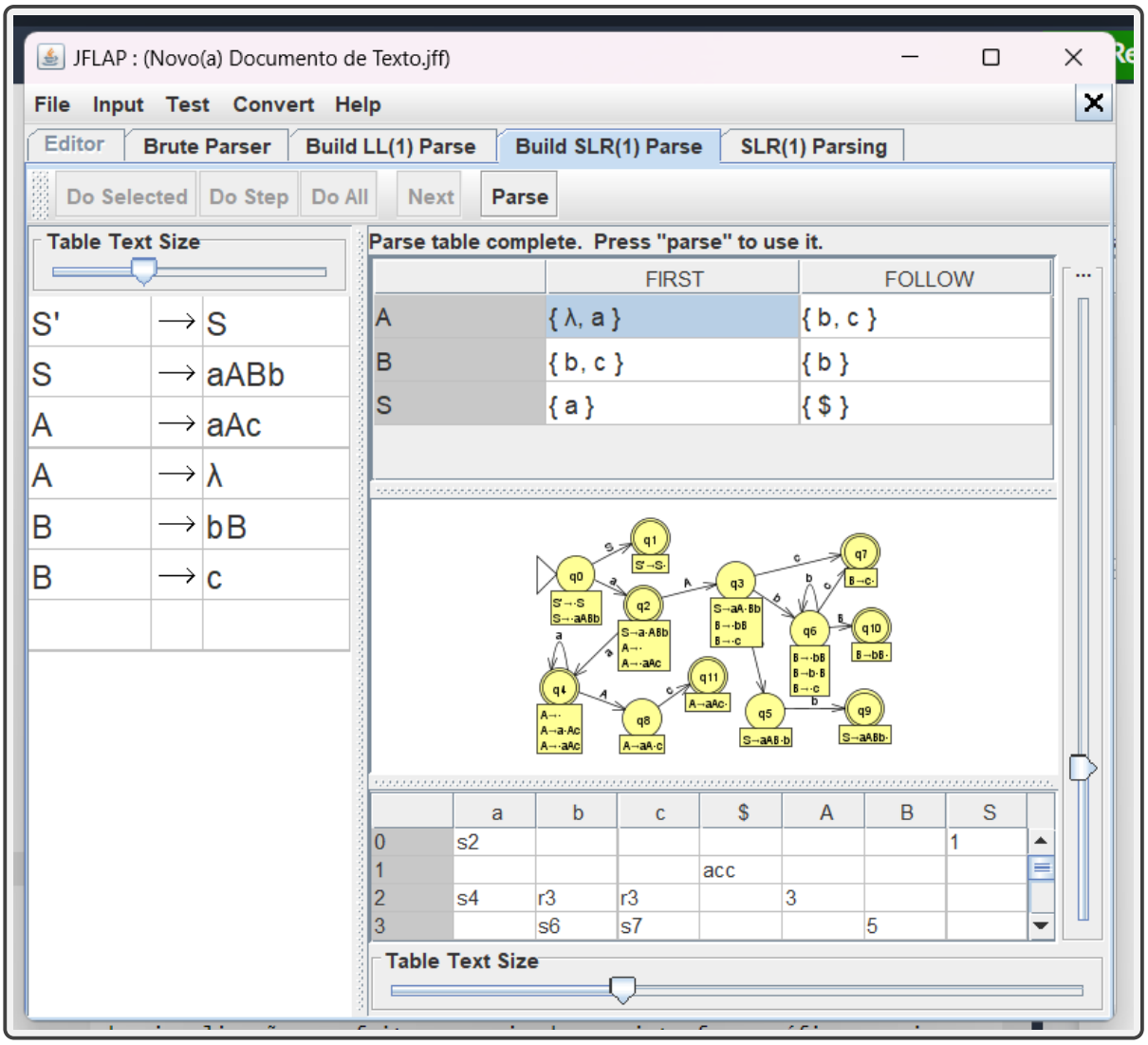


Fonte: Sangal, Kataria e Tyagi (2017).

3.4 JFLAP

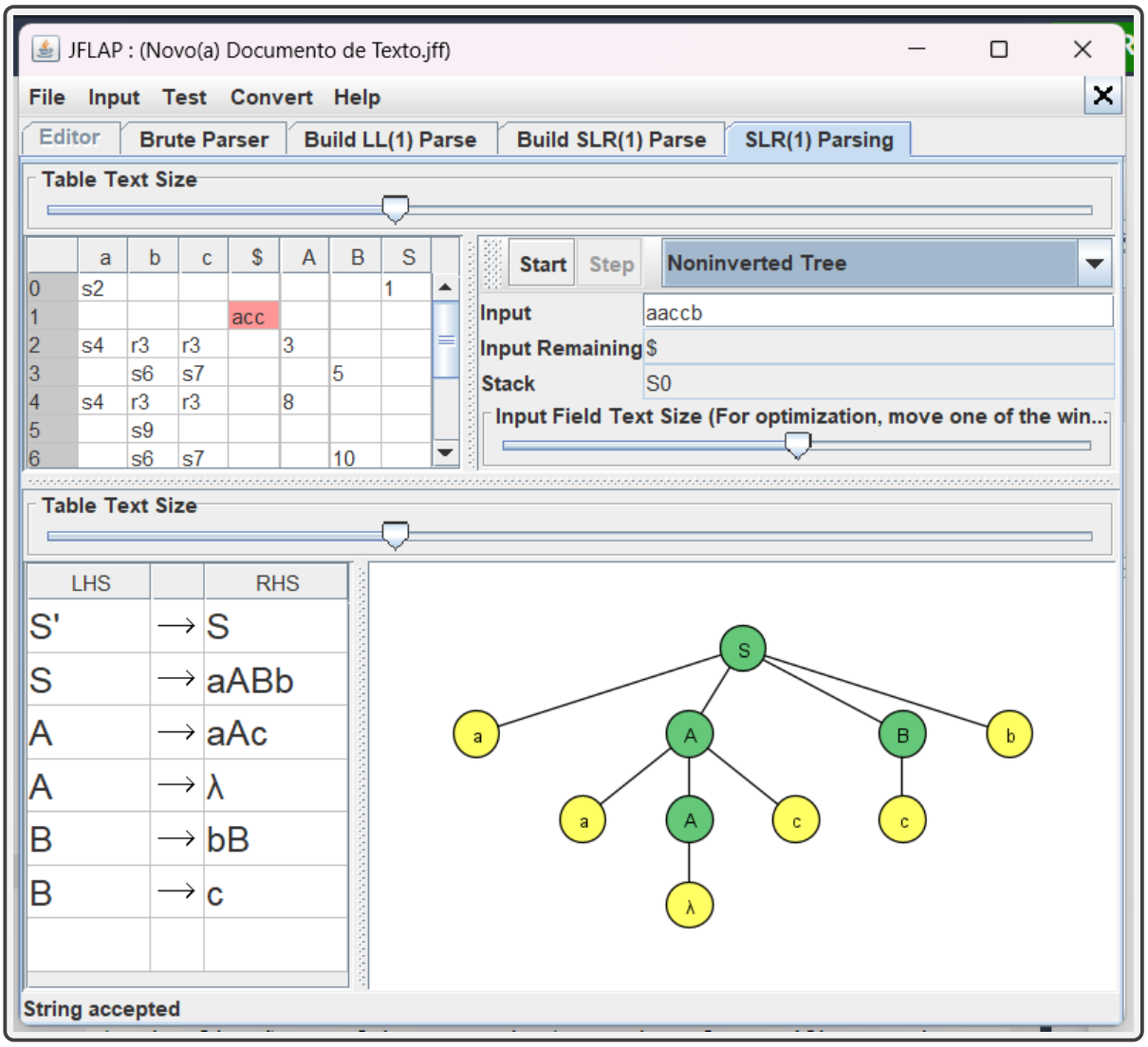
JFLAP (*Java Formal Languages and Automata Package*) é uma ferramenta *desktop* criada por Rodger e Duke University (2018) que pode ser usada para visualização dos algoritmos LL(1), SLR e de força bruta. Apesar de a visualização ser feita por meio de uma interface gráfica, assim como na ferramenta citada na seção anterior, JFLAP não dá instruções de como chegar nos resultados mostrados.

Figura 7 – Imagem da ferramenta JFLAP



Fonte: Rodger e Duke University (2018).

Figura 8 – Imagem da ferramenta JFLAP



Fonte: Rodger e Duke University (2018).

3.5 Considerações

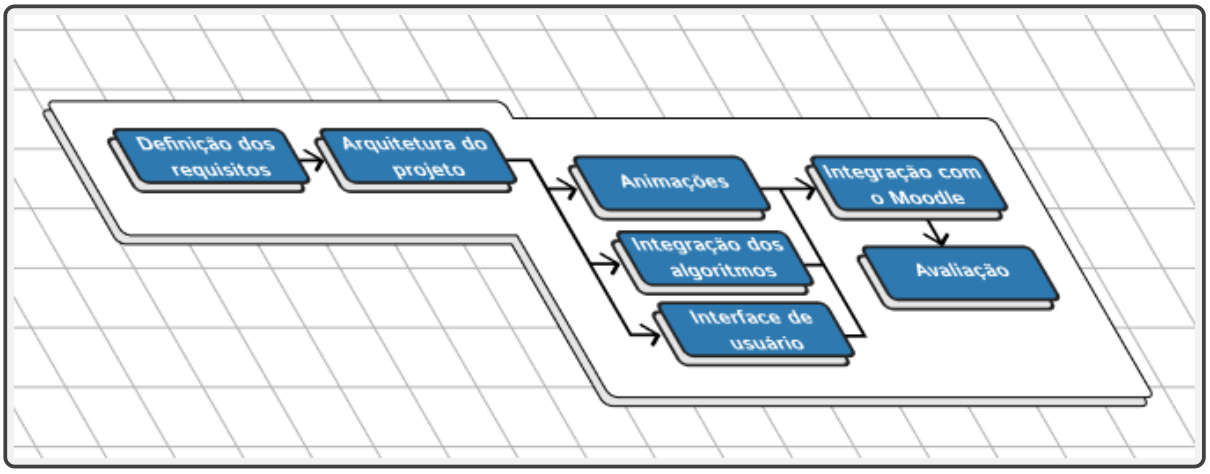
Apesar de já existirem ferramentas de visualização de *parsers*, algumas desvantagens ainda precisam ser consideradas. Uma limitação é que o conteúdo não tem muita interatividade, a ferramenta PAVT, por exemplo, apresenta apenas em um arquivo de texto. Isso pode dificultar a compreensão dos conceitos. A ferramenta *Parser Generator Web Tools* não oferece a visualização da árvore sintática que é um elemento fundamental para entender a estrutura da análise sintática. Outro ponto fraco é a falta de detalhamento do passo a passo dos algoritmos, o que impede que os estudantes acompanhem o funcionamento interno dos processos de análise. A variedade de algoritmos de análise sintática disponíveis nas ferramentas pode ser limitada,

restringindo a exposição dos alunos a diferentes abordagens e técnicas. 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.

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 9.

Figura 9 – Fluxograma das etapas



Fonte: fornecida pelo próprio autor

4.1 Definição dos requisitos

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 a gramática a 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

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* a 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 os resultados da análise de uma *string*
- Permitir que os usuários copiem implementações dos algoritmos

Com esses requisitos tem-se a base para o desenvolvimento da ferramenta.

4.2 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.

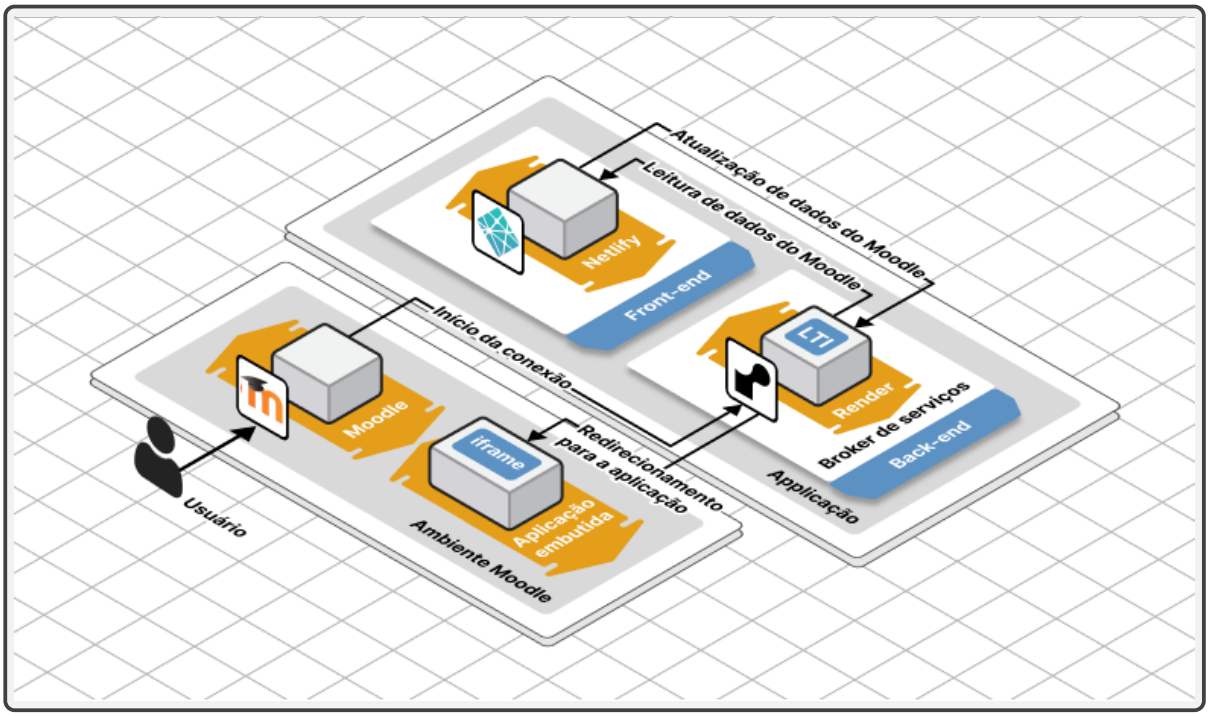
O *framework Capacitor* consome essa coleção de arquivos e cria um projeto para plataforma *Android* que é usado para criar a versão *mobile* da ferramenta usando o *Android Studio*.

O *framework Tauri* 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 plataforma os serviços de computação em nuvem das empresas *Netlify* e *Render* serão usadas para hospedar respectivamente o *front-end* e *back-end* da aplicação. *Netlify* e *Render* foram escolhidas para hospedagem da aplicação pelo oferecimento gratuito dos serviços para aplicações pequenas como a proposta nesse trabalho. A versão local da aplicação não utiliza serviços de computação em nuvem.

A aplicação pode ser acessada através do *Moodle* usando como comunicação entre os dois a *Application Programming Interface* (API) da aplicação. O diagrama na Figura 10 mostra o esquema da arquitetura da aplicação hospedada em nuvem com integração ao *Moodle*.

Figura 10 – Arquitetura da aplicação em nuvem



Fonte: fornecida pelo próprio autor

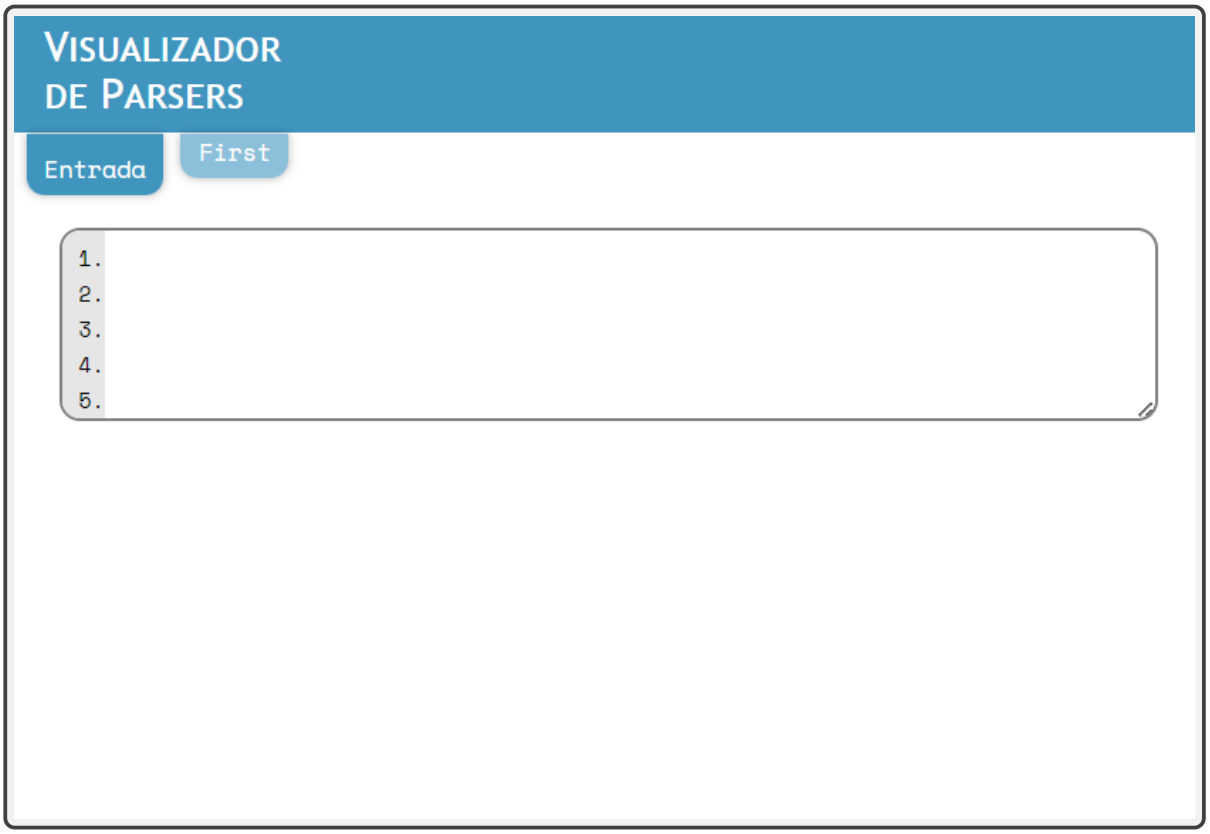
4.3 Interface de usuário

Essa seção irá falar sobre os detalhes de implementação da interface de usuário da ferramenta.

Para que o usuário selecione um algoritmo para visualizar a ferramenta terá opções no topo da tela que alternam entre abas. Dentre essas abas estarão inclusas uma para a entrada de gramáticas e uma aba para visualização de cada algoritmo disponível.

Para que o usuário possa dar uma gramática de entrada será criado um campo de texto como mostra Figura 11.

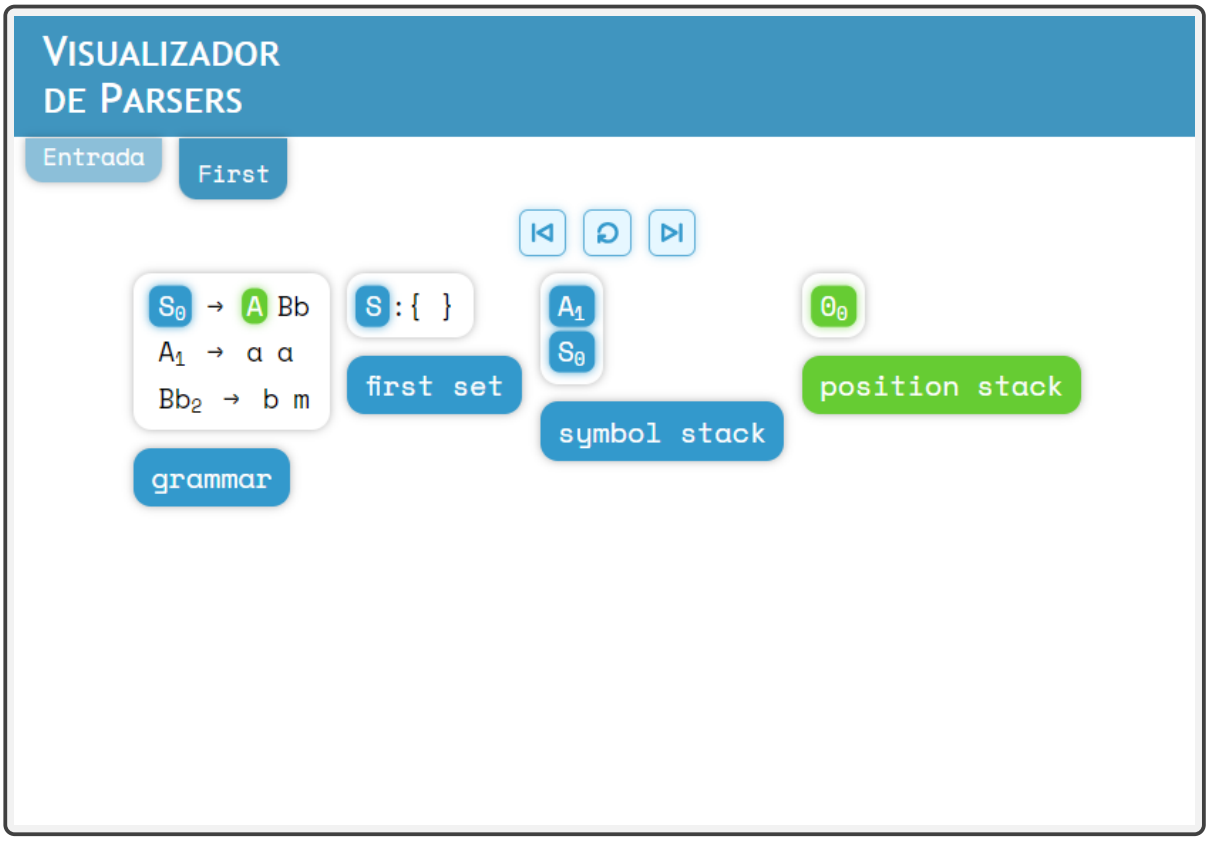
Figura 11 – Aba de entrada da gramática



Fonte: fornecida pelo próprio autor

Para a visualização dos algoritmos a ferramenta terá uma composição de elementos como mostra a Figura 12, esses elementos são modificados de acordo com a execução do algoritmo selecionado. Os passos da execução do algoritmo podem ser controlados pelo conjunto de controles acima dos elementos do algoritmo como mostra a Figura 12.

Figura 12 – Aba de visualização dos algoritmos



Fonte: fornecida pelo próprio autor

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

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 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 Animações

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. 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 Integração com o *Moodle*

A plataforma LMS *Moodle* implementa o padrão LTI o que permite a utilização da ferramenta criada nesse trabalho diretamente no *Moodle* sem necessidade de *login* externo. Para que a ferramenta possa utilizar o padrão LTI com o *Moodle* será implementado no *back-end* da aplicação uma API que manuseia as requisições relacionada ao LTI.

Os *endpoints* da API foram definidos de acordo com o Quadro 1.

Quadro 1 – Endpoints

Endpoint	Método	Finalidade
\	GET	redirecionar para página
\login	POST	inicia uma conexão com a plataforma
\register	POST	registrar uma nova plataforma

Fonte: fornecido pelo autor

Com a API pronta pode ser feita a conexão entre a ferramenta e a LMS. Utilizando o serviço LTI de *Dynamic Registration* é possível fazer o cadastro da ferramenta no *Moodle* utilizando um *link* para o *endpoint* de registro dinâmico da API.

4.7 Avaliação

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

Quadro 2 – Cronograma

Atividades	Período					
	1° mês	2° mês	3° mês	4° mês	5° mês	6° mês
Revisão bibliográfica	x					
Definição dos requisitos		x				
Implementação da interface de usuário		x	x			
Integração dos algoritmos			x	x	x	
Implementação das animações			x	x	x	
Avaliação					x	
Apresentação						x

Fonte: fornecido pelo autor

REFERÊNCIAS

1EDTECH. **Learning Tools Interoperability | 1EdTech**. Disponível em: https://www.1edtech.org/standards/lti#TechOverview. Acesso em: 27 maio 2024.

APOU, Gregory. **LR(1) Parser Generator**. Disponível em: https://jsmachines.sourceforge.net/machines/lr1.html. Acesso em: 2 maio 2024.

COOPER, K.D.; TORCZON, L. **Engineering a Compiler**. [*S. l.*]: Elsevier Science, 2022. ISBN 9780128189269. Disponível em: https://books.google.com.br/books?id=xcJrEAAAQBAJ.

DESAI, Jemin. **Web Application Vs Desktop Application: Pros and Cons**. Positiwise. 27 dez. 2023. Disponível em: https://positiwise.com/blog/web-application-vs-desktop-application-pros-and-cons. Acesso em: 16 maio 2024.

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.

HOLZER, Adrian; ONDRUS, Jan. Mobile app development: Native or web? *In:* PROC. Workshop eBus.(WeB). [*S. l.: s. n.*], 2012.

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. DOI: 10.1145/3002136. Disponível em: https://doi.org/10.1145/3002136.

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.

LIGHT. **Parser Generator Web Tools**. Disponível em: https://light0x00.github.io/parser-generator/. Acesso em: 2 maio 2024.

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

OLIVEIRA, Lais. **UFC abre inscrições para auxílio estudantil direcionado à compra de computador ou tablet**. O POVO. 31 jul. 2020. Disponível em: https://www.opovo.com.br/noticias/fortaleza/2020/07/31/ufc-abre-inscricoes-para-auxilio-estudantil-de-r--1-500-direcionado-a-compra-de-computador-ou-tablet.html. Acesso em: 16 maio 2024.

PEREZ, Katin Yovany. **The Impact of Lack of Internet and Technology Access on Students' Academic Achievement: An Analysis of the United States**. 2023. Tese (Doutorado) – Georgetown University.

RODGER, Susan H; DUKE UNIVERSITY. **JFLAP**. Versão 7.1. [*S. l.*], 27 jul. 2018. Disponível em: www.jflap.org. Acesso em: 2 maio 2024.

SANGAL, Somya; KATARIA, Shreya; TYAGI, Twishi. **PAVT**. Versão 0.0. [*S. l.*], 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.

SHEVTSIV, Nikita A; STRIUK, Andrii M. Cross platform development vs native development. *In:* CEUR WORKSHOP PROCEEDINGS.

THAIN, D. **Introduction to Compilers and Language Design: Second Edition**. [*S. l.*]: Amazon Digital Services LLC - Kdp, 2020. Disponível em: https://books.google.com.br/books?id=tQSYzQEACAAJ.