UNIVERSIDADE FEDERAL DE SANTA CATARINA
CAMPUS REITOR JOÃO DAVID FERREIRA LIMA
CIÊNCIAS DA COMPUTAÇÃO

Vitor Della Torre dos Santos

Rosetta: um webservice didático para auxiliar no aprendizado da conversão de expressões regulares em autômatos finitos determinísticos

Florianópolis
2023

Vitor Della Torre dos Santos

Rosetta: um webservice didático para auxiliar no aprendizado da conversão de expressões regulares em autômatos finitos determinísticos

Trabalho de Conclusão de Curso em Ciências da Computação do Campus Reitor João David Ferreira Lima da Universidade Federal de Santa Catarina para a obtenção do título de Bacharel em Ciências da Computação.
Orientadora: Profa. Jerusa Marchi, Dra.

Florianópolis
2023

Ficha de identificação da obra elaborada pelo autor,
através do Programa de Geração Automática da Biblioteca Universitária da UFSC.

Santos, Vitor Della Torre dos
Rosetta : um webservice didático para auxiliar no aprendizado da conversão de expressões regulares em autômatos finitos determinísticos / Vitor Della Torre dos Santos ; orientadora, Jerusa Marchi, 2023.
94 p.

Trabalho de Conclusão de Curso (graduação) - Universidade Federal de Santa Catarina, Centro Tecnológico, Graduação em Ciências da Computação, Florianópolis, 2023.

Inclui referências.

1. Ciências da Computação. 2. Linguagens Formais. 3. Aprendizado. I. Marchi, Jerusa. II. Universidade Federal de Santa Catarina. Graduação em Ciências da Computação. III. Título.

Vitor Della Torre dos Santos

Rosetta: um webservice didático para auxiliar no aprendizado da conversão de expressões regulares em autômatos finitos determinísticos

Este Trabalho de Conclusão de Curso foi julgado adequado para obtenção do Título de "Bacharel em Ciências da Computação" e aprovado em sua forma final pelo Programa de Graduação em Ciências da Computação do Departamento de Informática e Estatística do Centro Tecnológico.

Florianópolis, 24 de novembro de 2023.

Renato Cislaghi, Dr.
Coordenador do Programa de Graduação
em Ciências da Computação

Banca Examinadora:

Profa. Jerusa Marchi, Dra.
Orientadora
Universidade Federal de Santa Catarina -
UFSC

Prof. Maicon Rafael Zatelli, Dr.
Avaliador
Universidade Federal de Santa Catarina -
UFSC

Prof. Mauro Roisenberg, Dr.
Avaliador
Universidade Federal de Santa Catarina -
UFSC

AGRADECIMENTOS

Esse trabalho carrega uma força além de mim e que não poderia ter sido obtida unicamente da minha pessoa. Quero agradecer ao pessoal do Laboratório de Inteligência Artificial e Algoritmos; especialmente Evandro, Gabriel Turatti, Gabriel Cardoso, Tiago e César, que foram aqueles que me animaram e reforçaram minha resiliência durante o processo árduo de desenvolvimento e de escrita. Quero agradecer às minhas melhores amigas Ana, Isabelli, Sabrina, Sofia e Thaís, que foram pacientes comigo e que, mesmo em suas próprias batalhas, me ensinaram também a manter a calma e a compostura. Quero agradecer à minha amada namorada Gabriele, que quis saber a cada instante sobre o que projeto tratava mesmo que eu não soubesse por onde começar a explicar. Tive ela em mente quando fiz o texto e, a cada linha, espero que ela ame cada vez mais Computação e linguagens. E queria, por último, agradecer à professora Jerusa Marchi e aos meus pais Alexandre e Márcia e ao meu irmão Thales, que me ensinaram o valor do amor pelo que se faz e pelos outros, e o valor de cada uma de minhas gotas de suor — tanto quando concebia meus trabalhos quanto quando eu observava-os em seus processos criativos. Eu seria menos cientista e, principalmente, menos eu sem a presença de qualquer uma dessas pessoas e, por isso, eu só tenho o que ser grato.

"Ah! What a good day for science!"
(Dexter, Dexter's Laboratory, 1997)

RESUMO

O célebre livro *Compilers: Principles, Techniques and Tools* carrega em si um algoritmo de difícil manuseio e implementação: a conversão de expressões regulares em autômatos finitos determinísticos. Esta se põe como desafio àqueles que fazem a cadeira de Linguagens Formais e Compiladores na Universidade Federal de Santa Catarina, tendo em vista que o processo de transformação é propenso a erros quando feito no papel em decorrência do excesso de minúcias que devem ser levadas em consideração pelos alunos. Tendo isto posto, teve-se a iniciativa de descrever todos os passos do dito processo de conversão em um *webservice* a fim de ser de fácil compreensão e de fácil alcance, de modo a ajudar na didática e no aprendizado.

Palavras-chave: Linguagens Formais. Linguagens Regulares. Autômatos Finitos. Expressões Regulares. Aprendizado. Didática. Didático.

ABSTRACT

The famous book *Compilers: Principles, Techniques and Tools* carries a hard-to-use and hard-to-develop algorithm: converting regexes to deterministic finite automata. This stands as an obstacle to whom may try the Regular Languages and Compilers classes at the Federal University of Santa Catarina, as the transformation process is error prone when done by hand due to the excessive quantity of details that must be taken into account by the students. With that being said, surged the initiative of describing it step-by-step in a webservice in order to make it easy to reach and easy to understand, trying to enhance the learning process.

Keywords: Formal languages. Regular Languages. Finite Automata. Regular Expressions. Learning. Didactic.

RÉSUMÉ

Le célèbre livre *Compilers: Principles, Techniques and Tools* emporte un prcédé de difficile usage et d'être développé: la conversion d'une expression régulière à un automate fini. Cela se mettre comme un obstacle à tous que peuvent essayer de suivre la métier de Langages Formels et Compilateurs à l'Université Fédéral de Santa Catarina. Étant donné que le processus de transformation rendre facilement des erreurs quand fait à la main, une initiative est née pour en décrire étape par étape en un *webservice* dans le but de le faire plus facile d'être atteignable et compréhensible, en essayent de simplifier le processus d'apprentissage.

Mots-clés: Langages formels. Langages Réguliers. Automates Finis. Expressions Réguliers. Enseignement. Didactique.

LISTA DE FIGURAS

Figura 1 – O estado q_0 corresponde tanto ao estado inicial, denotado pela seta à esquerda, quanto ao estado final, formado por dois círculos concêntricos. 18

Figura 2 – Crescimento potencialmente exponencial da computação em NFAs (SIPSER, 2013, p. 49). 18

Figura 3 – Exemplo de AFND equivalente ao AFD da figura 1. 19

Figura 4 – Por transitividade, conclui-se, também que gramáticas do Tipo 3 e *regexes* são equivalentes. 20

Figura 5 – AFND obtido por meio do algoritmo de Thompson sobre expressão regular `(a|b)*abb`. Figura retirada de (AHO *et al.*, 2013, p. 175, exemplo 3.32). 20

Figura 6 – Árvore de análise léxica obtida da expressão regular `(a|b)*abb`, retirada de (AHO *et al.*, 2013, p. 174). 21

Figura 7 – Retirada de (AHO *et al.*, 2013, p. 178), contém os *firstpos* e *lastpos* calculados para nó da árvore. 23

Figura 8 – Diagrama de classes do software proposto, com as classes mais importantes em laranja mais saturado. 26

Figura 9 – Árvore binária de análise léxica de `(a|b)*abb` em que todos os nodos não tiveram seus respectivos `first_pos` e `follow_pos` calculados. . . . 33

Figura 10 – Árvore da expressão `(a|b)*abb` semi-cheia. Nodos a serem calculados são opacos e, aqueles que já o foram, lúcidos. 34

Figura 11 – Árvore com nodos associados à sua ordem de criação 36

Figura 12 – Exemplificação do funcionamento de `page_notes(DashPage)` para a `regex bbb`. Divide-se a árvore tanto pela ordem de criação dos nós quanto pela ordem do *reverse level order traversal* 40

Figura 13 – Aplicação com *regex* a ser interpretada 45

Figura 14 – Autômato gerado a partir de `(a|b)*abb#`. 45

SUMÁRIO

1 INTRODUÇÃO . . . 13
1.1 TRABALHOS RELACIONADOS . . . 14
1.1.1 JFLAP . . . 14
1.1.2 Automata Tutor v3 . . . 14
1.1.3 Software `qtype_flwarrior` . . . 15
1.2 OBJETIVOS . . . 15
1.2.1 Objetivo Geral . . . 15
1.2.2 Objetivos Específicos . . . 15
1.3 APRESENTAÇÃO DO TRABALHO . . . 16
2 FUNDAMENTAÇÃO TEÓRICA . . . 17
2.1 LINGUAGENS REGULARES . . . 17
2.1.1 Autômatos finitos . . . 17
2.1.2 Expressões regulares . . . 19
2.1.3 Equivalência entre representações . . . 20
2.1.4 Conversão de expressões regulares em autômatos finitos determinísticos . . . 20
2.1.4.1 Árvore binária de análise léxica, *nullability*, nodos e seus respectivos conjuntos 21
2.1.4.2 Construção de *followpos(n)* . . . 22
2.1.4.3 Uso de *followpos(n)* para criação dos estados do AF . . . 23
2.2 FERRAMENTAS UTILIZADAS . . . 23
2.2.1 Bibliotecas Plotly e Dash . . . 24
2.2.2 Biblioteca graphviz . . . 24
3 DESENVOLVIMENTO DO SOFTWARE PROPOSTO . . . 26
3.1 DIRETÓRIO *AUTOMATA* - ARQUIVOS `FA.PY` E `STATE.PY` . . . 26
3.2 DIRETÓRIO `REGEX` . . . 27
3.2.1 Arquivo `sanitization.py` . . . 28
3.2.2 Arquivo `format.py` . . . 28
3.2.3 Arquivo `syntax_tree.py` . . . 29
3.2.3.1 Classe `Node` . . . 29
3.2.3.2 Classe `SyntaxTree` . . . 30
3.2.4 Arquivo `conversion.py` . . . 32
3.3 DIRETÓRIO *COMPONENTS* . . . 33
3.3.1 Diretório `dash_page_progression` . . . 33
3.3.1.1 Arquivo `node.py` . . . 35
3.3.1.2 Arquivo `tree.py` . . . 35
3.3.1.3 Arquivo `graph.py` . . . 37
3.3.1.4 Arquivo `figure.py` . . . 38

3.3.1.5 Arquivo page.py 39
3.3.2 Diretório dash_follow_pos_table 40
3.3.3 Diretório dash_fa 42
3.3.3.1 Arquivo table.py 42
3.3.3.2 Arquivo figure.py 43
3.4 *ENTRY POINT* APP.PY 43
4 GUIA DE USO 44
4.1 PARA *DEVELOPERS* 44
4.2 PARA ALUNOS E PROFESSORES 44
5 CONSIDERAÇÕES FINAIS 47
REFERÊNCIAS 48
ANEXO A – ROSETTA EM SUA INTEGRALIDADE 49
A.1 SRC 49
A.1.1 automata 49
A.1.1.1 fa.py 49
A.1.1.2 state.py 50
A.1.2 components 51
A.1.2.1 dash_fa 51
A.1.2.1.1 figure.py 51
A.1.2.1.2 table.py 53
A.1.2.2 dash_follow_pos_table 54
A.1.2.2.1 table.py 54
A.1.2.3 dash_page_progression 56
A.1.2.3.1 figure.py 56
A.1.2.3.2 graph.py 60
A.1.2.3.3 node.py 62
A.1.2.3.4 page.py 63
A.1.2.3.5 tree.py 65
A.1.3 regex 70
A.1.3.1 conversion.py 70
A.1.3.2 format.py 73
A.1.3.3 sanitization.py 75
A.1.3.4 syntax_tree.py 76
A.2 APP.PY 81
ANEXO B – ARTIGO 87

1 INTRODUÇÃO

Em geral cursos de Bacharelado em Ciência da Computação, assim como tantos outro cursos de nível superior, apresentam disciplinas desafiadoras. Em Ciência da Computação, as disciplinas que abordam Linguagens Formais e Compiladores, devido as suas dualidades entre teoria e prática, enquadram-se neste grupo. Tais disciplinas abordam conteúdos relacionados à definição de aspectos formais de linguagens de programação e à construção de seus compiladores.

Uma linguagem de programação é definida através de seus aspectos léxicos, sintáticos e semânticos e a construção de compiladores para o reconhecimento destes aspectos envolve uma série de algoritmos que apresentam-se como desafiadores tanto para estudantes que os precisam compreender, quanto para professores, que buscam formas didáticas de os ensinar.

Os algoritmos necessários para o pleno compreendimento das etapas se opõem como obstáculos — tanto aos estudantes, que enfrentam a complexidade daqueles no aprendizado, quando aos professores, que lidam com as dificuldades de escrutiná-los.

Dentre os algoritmos estudados, há aqueles que são melhor acompanhados com a ajuda de extensos testes de mesa. Um destes é aquele que diz respeito ao processo de conversão de expressões regulares em autômatos finitos determinísticos (AHO *et al.*, 2013, p. 179). Para seu aprendizado e utilização, construir uma árvore binária léxica, extrair seus dados e construir um autômato com o casamento das informações são passos essenciais.

Contudo, há espaço para melhorias – se todo estudante pudesse ter acesso a um mecanismo facilitador para este tipo de estudo, certamente o processo de aprendizado ocorreria de maneira mais suave. Não obstante a isto, o processo laboral de ensino também seria menos difícil — professoras e professores da área de Computação poderiam apresentar o desenvolvimento de exercícios de tópicos relacionados a (AHO *et al.*, 2013, p. 179) em sala de aula concomitantemente com a utilização de um mecanismo que tornaria todo o processo de construção do algoritmo menos propenso a erros. Este trabalho se desenvolve, portanto, sobre a criação de `Rosetta`, software responsável por executar todos os passos do algoritmo mencionado, de maneira visualmente palatável aos discentes e docentes.

A peça de software `Rosetta` é uma aplicação desenvolvida na linguagem de programação de propósito geral `Python` que executa em um servidor web por meio de uma interface `Dash`, a ser explicada na seção 2.2.1. É acessível tanto por meio de execução em máquina própria pela desenvolvedora ou pelo desenvolvedor interessados no projeto, vide seção 4.1, quanto por meio de seu próprio domínio de *webservice*, explícito na seção 4.2.

1.1 TRABALHOS RELACIONADOS

1.1.1 JFLAP

Criado pelo *Department of Computer Science* da *Duke University*, *Java Formal Language and Automata Package* (JFLAP) é uma aplicação desktop escrita em Java Swing para descrição do comportamento de Autômatos Finitos, Gramáticas Regulares, Expressões Regulares, Autômatos de Pilha, Gramáticas Livre de Contexto e Linguagens Recursivamente Enumeráveis (RODGER, 2018) e também principal inspiração para este projeto. JFLAP detém sua própria interface gráfica interativa e, portanto, é um intenso facilitador no processo de ensino e aprendizagem de disciplinas que dialogam com o ramo de Linguagens Formais.

Contudo, JFLAP não provê pleno suporte para todos os algoritmos e procedimentos a serem estudados nas disciplinas de Linguagens Formais e Compiladores. No contexto desta disciplina, docentes não ensinam o Algoritmo de Thompson (AHO *et al.*, 2013, p. 159, seção 3.7.4), o qual é o utilizado pelo software JFLAP para converter expressões regulares em autômatos finitos não-determinísticos; estes são passíveis de serem determinizados posteriormente. O trabalho atual, não obstante, pauta-se em dissecar a conversão de expressões regulares em autômatos finitos determinísticos, sem que surjam aqueles que sejam não-determinísticos como uma etapa intermediária.

1.1.2 Automata Tutor v3

O software `Automata Tutor v3` descrito em `Scala` e `C#` foi desenvolvido pela *Technische Universität München* e vai além de funcionalidades previamente descritas pelo JFLAP. Permite criar máquinas abstratas como autômatos finitos (determinísticos ou não), autômatos de pilha e máquinas de Turing em um *canvas*, mas, também, tem suporte à geração automatizada de atividades relacionadas às disciplinas de Teoria da Computação e Linguagens Formais e Compiladores, desde encontrar palavras reconhecíveis em uma expressão regular até jogos com o Lema do Bombeamento (D'ANTONI *et al.*, 2020, p. 7, seção 3.2). Esta versão de *Automata Tutor* também provê avaliação automática e *feedback* de exercícios selecionados por professores de uma cadeira a serem resolvidos pelos estudantes. Este software já foi apropriadamente testado por mais de 900 estudantes (D'ANTONI *et al.*, 2020, p. 3).

Não obstante ao que fora dito, `Automata Tutor v3` não implementa o processo de conversão de expressões regulares em autômatos finitos determinísticos, processo descrito de maneira extensiva neste trabalho, com suas partes dissecadas ao usar o software proposto `Rosetta`.

1.1.3 Software `qtype_flwarrior`

O plugin do Moodle `qtype_flwarrior` (ALBORNOZ, 2022) desenvolvido em PHP pelo estudantes Enzo Coelho Albornoz tem como objetivo fazer a avaliação automática de exercícios relacionados a autômatos finitos, autômatos de pilha e máquinas de Turing a serem propostos por docentes em disciplinas relacionadas ao domínio de Linguagens Formais e Compiladores. Este trabalho foi diretamente inspirado por JFLAP, descrito na seção 1.1.1. Este software, todavia, também não analisa o processo de conversão de expressões regulares em autômatos finitos determinísticos.

1.2 OBJETIVOS

1.2.1 Objetivo Geral

Deseja-se criar um software que possa usufruir da fundamentação teórica descrita na seção 2 para auxiliar no processo didático dos discentes e docentes que quiserem aprender a realizar, passo a passo, conversões de expressões regulares em autômatos finitos determinísticos. Quer-se, então, facilitar o processo de aprendizado do citado algoritmo tendo em vista que, manualmente, apresenta-se como um processo sujeito a erros.

1.2.2 Objetivos Específicos

Como objetivos específicos deste trabalho, quer-se:

- Criar um software que receba expressões regulares por *text inputs* e, destes, gere:
  - Uma árvore binária de análise léxica;
  - Uma tabela para um conjunto *followpos*;
  - Uma tabela representativa de um autômato finito;
  - Um diagrama de estados de um autômato finito.
- Providenciar um serviço acessível àqueles que desejarem aprender, ensinar por meio de, ou, até, realizar melhorias no software `Rosetta`.

1.3 APRESENTAÇÃO DO TRABALHO

A monografia completa tem 5 capítulos. O capítulo 2 fundamenta conceitos formais, a fim de fornecer o arcabouço teórico utilizado no processo de criação e desenvolvimento do software proposto pelo trabalho, descrito no capítulo 3. Adiante, o capítulo 4 apresenta um guia de uso do software criado para que se possa elucidar quaisquer dúvidas a respeito do uso deste, sejam estas de desenvolvedores ou de estudantes e professores. Por fim, o capítulo 5 descreve as considerações finais do trabalho, conclusões e ideias para projetos futuros, tais quais *features* a serem eventualmente desenvolvidas e adicionadas ao trabalho.

2 FUNDAMENTAÇÃO TEÓRICA

Este capítulo inicialmente se debruça sobre as estruturas abstratas a serem manuseadas no software proposto — autômatos finitos e expressões regulares; a critério didático, explica-se, também, gramáticas regulares a fim de providenciar melhor compreensão das subseções. Feita a fundamentação das mencionadas estruturas, cita-se, por fim, as bibliotecas utilizadas no desenvolvimento do projeto.

2.1 LINGUAGENS REGULARES

Linguagens regulares compreendem o conjunto de linguagens denotadas pelo Tipo 3 na Hierarquia de Chomsky (HOPCROFT; ULLMAN, 1969, p. 13). São geradas por gramáticas regulares, reconhecidas por autômatos finitos e podem ser descritas por expressões regulares. Uma linguagem é formada por um alfabeto, o qual compreende um conjunto de símbolos, e regras a serem utilizadas para produzir sequências de símbolos pertences àquela, conhecidas como palavras.

Exemplo 1 *Considere a linguagem abaixo.*

$$L = \{w \mid w \in \{a, b\}^*,\ |w|\ mod\ 2 = 0\}$$

Para a linguagem L do exemplo 1, a's e b's podem permutar dentro de palavras que pertencem àquela, sob a condição de que a cardinalidade de qualquer sequência de caracteres seja um múltiplo de 2. Apesar de breve, o dado exemplo será aprofundado nas seções que seguem nas perspectivas de um autômato finito e de uma expressão regular.

2.1.1 Autômatos finitos

De acordo com (SIPSER, 2013, p. 35, definição 1.5), um autômato finito (*finite automata*, FA ou AF) é um reconhecedor de linguagens regulares. É uma quíntupla $(Q, \Sigma, \delta, q0, F)$ constituída de:

- Um conjunto finito Q de estados;
- Um conjunto finito Σ de caracteres conhecido como alfabeto;
- Uma função $\delta : Q \times \Sigma \rightarrow Q$ sobrejetiva de transição;
- Um *único* estado inicial $q_0 \in Q$;
- Um conjunto de estados finais $F \subseteq Q$.

Exemplo 2 *Apresenta-se novamente o exemplo 1, desta vez como um autômato finito.*

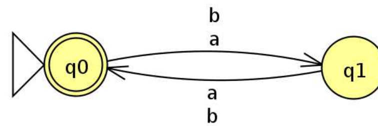

Figura 1 – O estado q_0 corresponde tanto ao estado inicial, denotado pela seta à esquerda, quanto ao estado final, formado por dois círculos concêntricos.

Um autômato finito reconhece sentenças produzidas por uma gramática regular, não aprofundada neste trabalho, equivalente à uma dada linguagem de mesmo tipo. Para cada caractere de entrada, a partir do estado inicial, realizam-se transições de um estado-fonte a um estado-destino. Na ocorrência deste último pertencer aos estados finais no momento em que há o término da leitura completa da entrada, tem-se um reconhecimento da sentença, ou, caso contrário, sua rejeição.

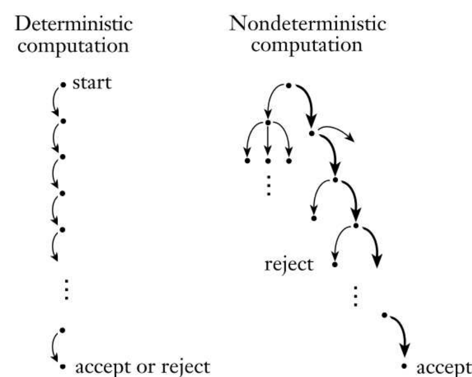

Figura 2 – Crescimento potencialmente exponencial da computação em NFAs (SIPSER, 2013, p. 49).

Autômatos podem apresentar comportamento determinístico ou não. Àqueles que compreendem o último tipo, muda-se a função de transição para a forma $\delta : Q \times \{\Sigma \cup \varepsilon\} \rightarrow 2^Q$ (SIPSER, 2013, p. 53) caso possuam transições por ε de um dado estado ou duas ou mais transições pelo mesmo símbolo de um único estado-fonte. Para tais autômatos, pode-se obter um número potencialmente exponencial de histórias de computação, ou seja, cômputos paralelos que podem ser realizados até que pelo menos um destes alcance um estado final ao término da leitura da sentença de entrada, como exemplificado na figura 2. Caso haja a exaustão da sentença de entrada sem que tenha sido possível alcançar um estado final em quaisquer um dos ramos, rejeita-se a entrada.

Autômatos finitos determinísticos (DFA, do inglês *Deterministic Finite Automata*, ou AFD) e não-determinísticos (NFA, do inglês *Non-deterministic Finite Automata*, ou AFND) são equivalentes, de modo que será *sempre* possível fazer a conversão de um NFA

a um DFA. Este processo, reconhecido por *determinização*, é formalmente provado em (SIPSER, 2013, p. 55, teorema 1.39).

Exemplo 3 *Análogo aos exemplos anteriores, apresenta-se uma versão não-determinística do FA evidenciado no exemplo 2.*

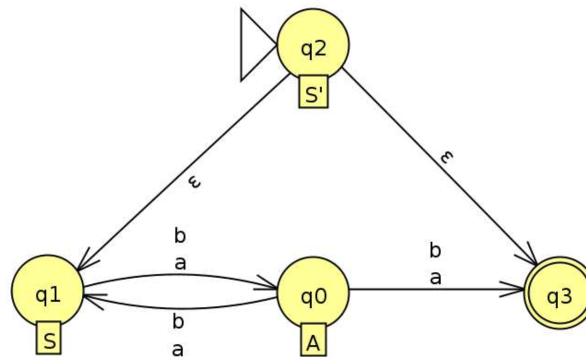

Figura 3 – Exemplo de AFND equivalente ao AFD da figura 1.

2.1.2 Expressões regulares

Expressões regulares (*regex*, do inglês *regular expression*) são mecanismos descritores e sintetizadores de linguagens regulares (SIPSER, 2013, p. 63, seção 1.3), de modo que são intensamente utilizadas em uma miríade de softwares em decorrência de seu forte poder expressivo enquanto permanecem visualmente enxutas. A definição deste mecanismo formal de representação das linguagens regulares faz uso de recursão como mecanismo central de sua explicação formal. Como visto em *Introduction to the Theory of Computation* - p.64, compreende-se R como *regex* se for:

- a para um a pertencente a um alfabeto Σ qualquer;
- ε;
- $\emptyset$;
- $(R1 \cup R2)$, se $R1$ e $R2$ forem *regexes*, em que R equivale à união de $R1$ com $R2$;
- $(R1 \circ R2)$, se $R1$ e $R2$ forem *regexes*, em que R equivale à concatenação de $R1$ com $R2$;
- $(R1^*)$, se $R1$ for *regex*, em que o operador unário $*$ corresponde ao fecho de Kleene.

Exemplo 4 *Por último, retoma-se a linguagem L visualizada no exemplo 1 representada na forma de uma regex R.*

$$R(L) = ((a + b)(a + b))*$$

2.1.3 Equivalência entre representações

Autômatos finitos e expressões regulares são equivalentes no âmbito de representação de uma dada linguagem, assim como são equivalentes às Gramáticas Regulares (SIPSER, 2013), fora do escopo deste trabalho. Como formalmente demonstrado em (HOPCROFT; ULLMAN, 1969, p. 33, seção 3.4), não só há, para toda gramática regular, um autômato finito que reconhece a linguagem que esta gera, como também há, para todo autômato finito, uma gramática regular que gere a linguagem que aquele reconheça. Em paralelo, ao converter uma gramática regular a um autômato finito (HOPCROFT; ULLMAN, 1969, p. 34, teorema 3.5), pode-se obter um AFND. No contexto em que há a obtenção de um AFND, pode-se entender que há um AFD que o representa (SIPSER, 2013, p. 55, teorema 1.39), como dito anteriormente. Por fim, observa-se a equivalência de *regexes* e autômatos finitos em (SIPSER, 2013, p. 66).

Figura 4 – Por transitividade, conclui-se, também que gramáticas do Tipo 3 e *regexes* são equivalentes.

2.1.4 Conversão de expressões regulares em autômatos finitos determinísticos

A conversão de expressões regulares pode ser feita pelo Algoritmo de Thompson, como apresentado em (AHO *et al.*, 2013, p. 159, seção 3.7.4), por meio do uso os elementos fundamentais da *regex* e suas propriedades, notadamente união, fecho e concatenação; e, também, com o uso de ε-transições para construir expressões mais complexas.

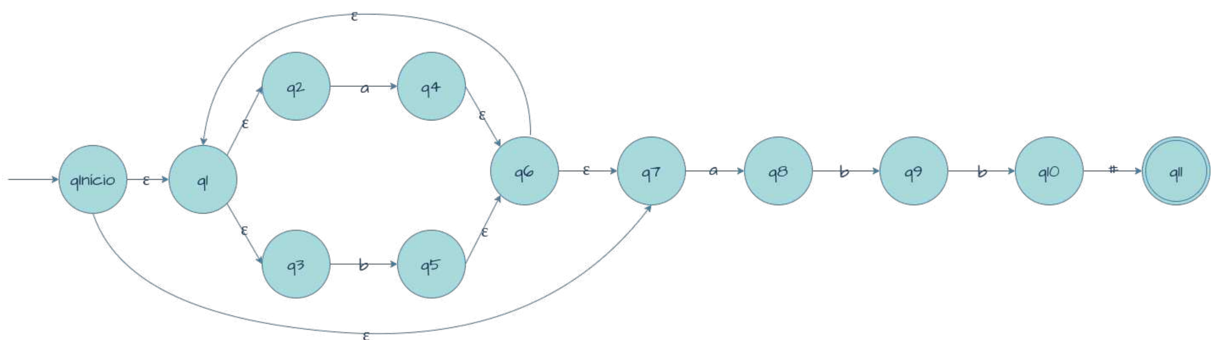

Figura 5 – AFND obtido por meio do algoritmo de Thompson sobre expressão regular `(a|b)*abb`. Figura retirada de (AHO *et al.*, 2013, p. 175, exemplo 3.32).

Este processo desempenha complexidade de ordem $O(n)$. Porém, o resultado desta transformação é um AFND, que ao ser determinizado, pode resultar em uma FA com

um número exponencial de estados, a ser determinizado por algoritmo à escolha do usuário; caso faça-se uso do algoritmo *Powerset Construction*, a complexidade desta etapa surge como $O(n^2)$. Destarte, visando simplificar o custo deste processo, para que tal transformação seja utilizada na construção de analisadores léxicos, em (AHO *et al.*, 2013, p. 173, seção 3.9) é apresentado um algoritmo que transforma uma *regex* diretamente em um automato finito determinístico.

Este algoritmo desempenha complexidade temporal assintótica equivalente a $O(n^2)$, em que n representa o tamanho da expressão regular a ser utilizada. A complexidade quadrática advém do cálculo dos conjuntos *followpos*, explicados na seção 2.1.4.2, processo que envolve realizar a travessia da árvore de análise léxica e atualizar os conjuntos *followpos* de acordo com os *firstpos* e *lastpos* dos nós, o que pode levar à uma complexidade que vale o tamanho ao quadrado da `regex` dada.

Vide explicado em (AHO *et al.*, 2013, p. 173, seção 3.9), o algoritmo de conversão de expressões regulares pode ser dividido em 3, menores, partes de um todo.

2.1.4.1 Árvore binária de análise léxica, *nullability*, nodos e seus respectivos conjuntos

Uma árvore de análise léxica é um tipo de árvore binária em que seus nodos-folha representam operandos e, os interiores, operadores (AHO *et al.*, 2013, p. 174).

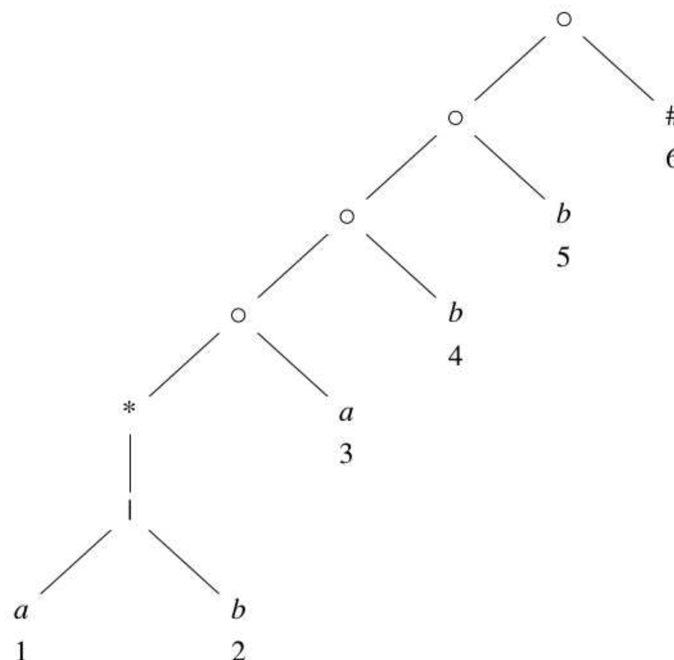

Figura 6 – Árvore de análise léxica obtida da expressão regular `(a|b)*abb`, retirada de (AHO *et al.*, 2013, p. 174).

Folhas são associados a um identificador inteiro único, o qual é chamado de *posição*. O algoritmo de construção da árvore de análise léxica pode ser obtido em (AHO *et al.*,

2013, p. 69, seção 2.5.1).

Para cada nodo, folha ou não, deve-se fazer a verificação de sua anulabilidade e, de acordo com o resultado desta, pode-se calcular seus conjuntos *firstpos* e *lastpos*. De acordo com (AHO *et al.*, 2013, p. 175, seção 3.9.2), o conjunto *firstpos* diz respeito às posições de um subárvore enraizada em um nodo n que correspondem ao primeiro símbolo de pelo menos uma *string* na linguagem da subexpressão encontrada a partir do nodo n. O conjunto *lastpos* age como contraparte de *firstpos*: é o conjunto de posições de uma subárvore enraizada em n que correspondem ao último símbolo de pelo menos uma *string* na linguagem da subexpressão encontrada a partir do nodo n.

As regras para cálculo dos conjuntos *lastpos* são as mesmas dos conjuntos *firstpos*: basta substituir *lastpos* por *firstpos* na leitura da tabela durante a aplicação dos passos, com a ordem dos nodos-filhos invertida. Deve-se começar a aplicação de nodos-folha, sempre, visto que nodos superiores irão depender dos resultados dos conjuntos de seus nodos-filhos. A tabela a seguir, retirada de (AHO *et al.*, 2013, p. 177), exemplifica como obter os ditos conjuntos.

Node n	*nullable(n)*	*firstpos(n)*
Nodo-folha ε	*True*	$\emptyset$
Nodo-folha i	False	i
Nodo-*or* n = *c1*/*c2*	*nullable(c1)* **OR** *nullable(c2)*	*firstpos(c1)* ∪ *firstpos(c2)*
Nodo-concatenação n = *c1c2*	*nullable(c1)* **AND** *nullable(c2)*	*firstpos(c1)* ∪ *firstpos(c2)* **if** *nullable(c1)* **else** *firstpos(c1)*
Nodo-estrela $n = c1^*$	*True*	$firstpos(c1)$

2.1.4.2 Construção de *followpos(n)*

De acordo com (AHO *et al.*, 2013, p. 177, seção 3.9.4), há apenas 2 maneiras de calcular um *followpos*:

- Se *node n* for representativo de uma *concatenação*, com filho-esquerdo *c1* e filho-direito *c2*, então, para cada posição i em *lastpos(c1)*, todas as *posições* em *firstpos(c2)* estarão em *followpos(i)*.
- Se *node n* for representativo de um nó-estrela, e i é uma *posição* em *lastpos(n)*, então todas as *posições* em *firstpos(n)* estarão em *followpos(i)*.

Segue abaixo a tabela dos *followpos* para a árvore da figura 6, retirada de (AHO *et al.*, 2013, p. 178).

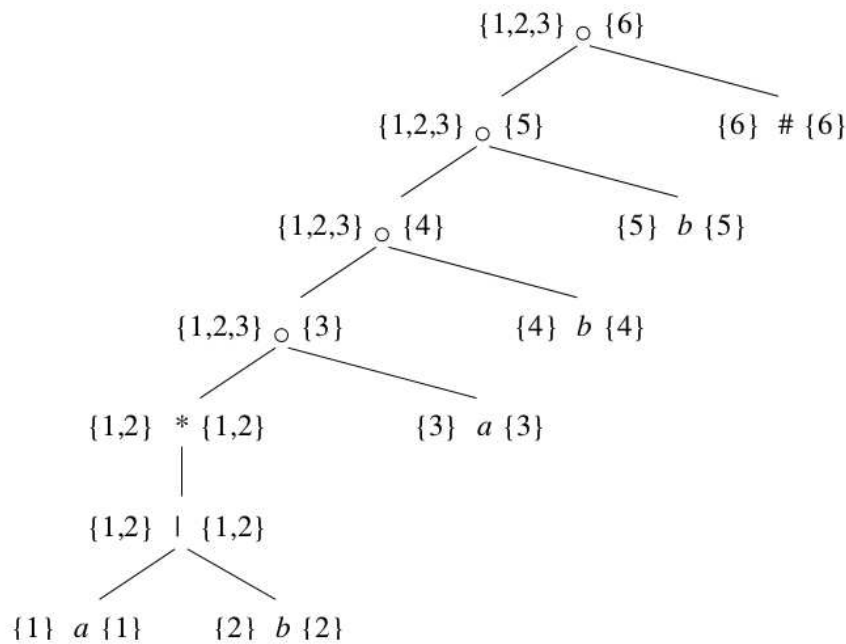

Figura 7 – Retirada de (AHO *et al.*, 2013, p. 178), contém os *firstpos* e *lastpos* calculados para nó da árvore.

Position n	*followpos(n)*
1	{1,2,3}
2	{1,2,3}
3	{4}
4	{5}
5	{6}
6	∅

2.1.4.3 Uso de *followpos(n)* para criação dos estados do AF

Segundo (AHO *et al.*, 2013, p. 179, seção 3.9.5), deve-se criar um conjunto *DStates* e *DTransitions* para atuarem, respectivamente, como estados e transições de um autômato *D* a ser gerado a partir da *regex* de uma árvore T. O estado inicial de D é dado por *firstpos* da raiz da árvore, enquanto seus estados finais são todos aqueles que contém a *posição* do marcador de final de sentença estipulado pelo algoritmo, o qual é a cerquilha (#). Os passos de computação do conjunto *Dstates* e das *Dtransitions* podem ser encontrado no Algoritmo 1.

2.2 FERRAMENTAS UTILIZADAS

Para a construção do software proposto, descrito no próximo capítulo, fez-se necessária a utilização de bibliotecas a fim de apresentar graficamente componentes ao usuário

Algorithm 1 Passos para obtenção de *Dstates* e *Dtransitions*. Retirada de (AHO *et al.*, 2013, p. 180).

```
Dstates ← firstpos(root), em que 'root' é a raiz da árvore de análise léxica.
while exista um elemento S em DStates do
    marque S
    for cada símbolo A na entrada do
        seja U a união de followpos(p) para todo p em S que corresponda a A
        if U não está em Dstates then
            Adicione U como não marcado a Dstates
        end if
        Dtransitions[S, a] ← U
    end for
end while
```

e criar um serviço web para que este pudesse acessar. Esta seção se debruça em descrever essas funcionalidades asseguradas pelas *libraries* `Plotly` e `Dash`, e `graphviz`.

2.2.1 Bibliotecas Plotly e Dash

`Plotly` é uma biblioteca *open-source* utilizada para geração de gráficos e grafos (PLOTLY. . . , s.d.). Pode ser utilizada em uma pequena parcela de linguagens de programação de propósito geral, como `Python`, `R`, `Julia` e `JavaScript`.

Não obstante a isto, `Plotly` também é acompanhada de uma interface para construção de *webservices*, caso queira-se criar algum tipo de serviço de demonstração das estruturas criadas com a biblioteca mencionada. Esta dita interface é chamada `Dash` (DASH. . . , s.d.).

Neste trabalho, fez-se uso de `Dash` para a execução do *webservice* responsável por apresentar as estruturas `Plotly` da árvore de análise léxica, da tabela *followpos* (ou `follow_pos` no código, para facilitar legibilidade) e da tabela do autômato finito obtido da regex colocada pelo usuário.

`Plotly` permite customização de todos os seus parâmetros, mas também vem com o custo de um *knowhow* extenso a respeito do layout e do formato desejado para uma dada estrutura. É útil e de fácil manuseio quando esta é meramente uma tabela, mas torna-se brutalmente complexa quando quer-se *plotar* uma árvore. A árvore binária de análise léxica precisava obedecer a um design em especial, em que sua raiz fica centralizada no topo da figura, enquanto seus nodos estarão igualmente espaçados entre si e orientados para crescerem apenas para baixo.

2.2.2 Biblioteca graphviz

Para a geração do diagrama de estados do autômato finito, fez-se uso de uma biblioteca diferente de `Plotly`, tendo em vista que o design do diagrama de estados do autômato finito não precisava obedecer a tantas restrições quanto o da árvore. Cita-se

aqui, novamente, a árvore de análise léxica visto que são ambos estruturas de grafos direcionados e as únicas deste tipo no projeto.

Objetivou-se uma biblioteca que habilitasse um custo baixo de implementação para o diagrama de estados de AFs. `graphviz` (GRAPHVIZ. . . , s.d.) tem interface mais direta que a de `Plotly` e exige menos customização prévia para a obtenção de um resultado. Implementou-se o autômato finito gerado da *regex* dada pelo usuário como um `graphviz.Digraph`.

3 DESENVOLVIMENTO DO SOFTWARE PROPOSTO

O software `Rosetta` é composto de 4 principais módulos: 3 diretórios dentro do diretório de códigos-fonte `src/` e um programa que age como *entry point* da aplicação, chamado de `app.py`. A descrição do desenvolvimento é pautada por dissecar os submódulos de `src/` – arquivos com prefixo `.py` ou o diretório que os contempla –, dos mais triviais e menos dependentes de demais estruturas de código às estruturas mais complexas e que fazem uso daquelas que foram anteriormente citadas – uma abordagem *bottom-up*.

A estrutura do código obedece o diagrama de classes visível na Figura 8, com os módulos mais importantes salientados em laranja mais escuro que os demais.

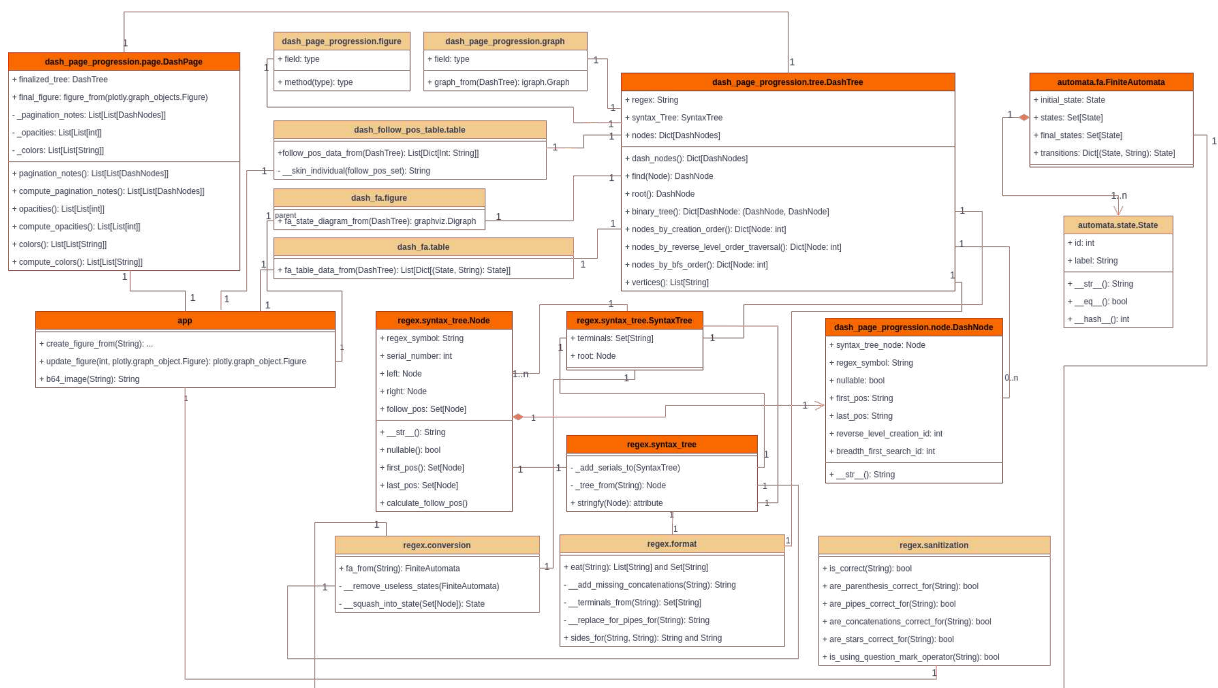

Figura 8 – Diagrama de classes do software proposto, com as classes mais importantes em laranja mais saturado.

3.1 DIRETÓRIO *AUTOMATA* - ARQUIVOS `FA.PY` E `STATE.PY`

Antes de explicar o funcionamento da classe dos FAs, explica-se a classe `State`, visto que, para que funcione, um `FiniteAutomata` necessita de uma classe que descreva o que se espera do comportamento de um estado para suas variáveis `self.initial_state` e `self.final_states`

```
1  class State:
2
3      def __init__(self, label: str = None):
```

```
4           self.id = process_time_ns()
5           self.label = label
6
7       def __str__(self):
8           return self.label if self.label is not None else '{}'.format(self.id)
9
10      def __hash__(self):
11          return hash(self.label)
12
13      def __eq__(self, other):
14          return hash(self.label) == hash(other.label)
```

Um `State` de classe de mesmo nome, para `Rosetta`, enquanto fundamental para a descrição de um `FiniteAutomata`, é manuseado para compor seus conjuntos de estados e estados finais e também para ser apresentado ao usuário.

Para que possa ser dito elemento de um conjunto, um estado deve ser *hashable* (passível de gerar um número *hash* por meio da chamada de função homônima) e ser comparável a outro `State` quando solicitado. Um `State` só é dito igual a outro se seu número *hash* é idêntico ao de outro. De modo a assegurar *hashability* de um estado, necessita-se realizar a sobre-escrita de suas funções `__hash__()` (acima, na linha 10) e `__eq__(other)` (acima, na linha 13).

Explicada `State`, prossegue-se à classe `FiniteAutomata`. Classe de um autômato finito, de acordo com o que fora mencionado na seção 2.1.1, obedece à definição do tipo mais trivial de AF – contém um único estado, o qual é inicial (abaixo, linha 5), ao mesmo tempo que é despido de transições e de estados finais, visível nas linhas 6 e 7, com a inicialização de um dicionário e um conjunto vazios.

```
1   class FiniteAutomata:
2
3       def __init__(self):
4           self.initial_state = State()
5           self.states = {self.initial_state}
6           self.transitions = dict()
7           self.final_states = set()
```

3.2 DIRETÓRIO REGEX

O diretório `regex` compreende 3 arquivos diferentes:

- `sanitization.py`: arquivo com funções necessárias para verificar se a dada `regex` dada pelo usuário pode ser de fato utilizada.

- `format.py`: módulo que empunha um conjunto de funções responsáveis por preparar uma expressão regular a ser apropriadamente lida por uma `SyntaxTree`.
- `syntax_tree.py`: responsável por gerar uma árvore binária de análise léxica a partir de uma expressão regular a ser digerida pelo construtor da classe `SyntaxTree`.
- `conversion.py`: manuseia uma árvore de análise léxica a fim de extrair as informações necessárias para gerar um `FiniteAutomata` destas.

3.2.1 Arquivo `sanitization.py`

O arquivo `sanitization.py` se pauta em fazer verificações triviais quanto ao uso dos operadores nas expressõres regulares digitadas pelo usuário, *e.g.* procura por concatenações duplas como "..". É um arquivo repleto de funções auxiliares que estão descritas em detalhes no Anexo A.1.3.3. Caso sejam encontrados erros na escrita da expressão regular, ou seja, caso obtenha-se `False` para a função `is_correct(regex)`, um *banner* de alerta irá surgir sobre o *input* de texto do usuário avisando-o das possíveis causas do erro.

```
def is_correct(regex):
    return (are_parenthesis_correct_for(regex) and
            are_pipes_correct_for(regex) and
            are_concatenations_correct_for(regex) and
            are_stars_correct_for(regex) and
            not is_using_question_mark_operator(regex))
```

3.2.2 Arquivo `format.py`

Com a sanitização da `regex` concluíuda, pode-se entender `format.py`. O arquivo `format.py` carrega em si diversas funções auxiliares, descritas em detalhes no Anexo A.1.3.2, e uma função principal `eat(regex)`, que digere uma expressão regular e, desta, retorna, na linha 15, uma lista de `strings` na variável `digest` e o conjunto de todos os terminais, com ε incluso neste.

```
1  def eat(regex) -> list and set:
2      """Eats a non formatted regex and returns its list digested form
3      with its terminal symbols."""
4
5      if len(regex) == 1:
6          # If a regex is 1 char long, that regex is already digested.
7          return [regex], {regex}
8
```

```
9       regex = ['('] + list(regex) + [')'] + (['#'] if regex[-1] != '#' else [])
10      regex = ''.join(regex)
11
12      digest = regex.replace(' ', '')
13
14      digest = __add_missing_concatenations(digest)
15      return list(digest), __terminals_from(digest) | {'&'}
```

A lista `digest`, *e. g.* para uma `regex` `(a|b)*abb#`, retorna uma lista com cada um dos strings alocados a uma única posição do *array*, de modo que as operações de concatenação implícitas da expressão regular sejam de fato colocadas no resultado.

$$(a|b) * abb\# \xrightarrow{\text{digest('(a|b)*abb\#')}} [(, a, |, b,), *, ., a, ., b, ., b, ., \#] \quad (1)$$

3.2.3 Arquivo `syntax_tree.py`

3.2.3.1 Classe `Node`

Assim como não há autômato finito sem estados, não há árvore de análise léxica sem nodos – sem nós, não haveria árvore alguma. Isto posto, antes da compreensão da classe que compreende de fato a dita árvore a ser manuseada, faz-se necessário descrever a classe `Node`.

```
1   class Node:
2
3       def __init__(self, symbol: str, left=None, right=None):
4           self.regex_symbol = symbol
5           self.serial_number = 0
6
7           self.left: Node = left
8           self.right: Node = right
9
10          self.follow_pos = set()
```

Um nodo carrega em si um símbolo representativo da `regex` (linha 4), terminal ou não-terminal; um número serial (linha 5), diferente de zero caso seja representativo de um símbolo do conjunto dos terminais; um nodo-filho à esquerda (linha 7), assim como à direita (linha 8); e, por fim, um conjunto referente ao seu próprio `follow_pos` (linha 10).

Os métodos referentes à obtenção dos conjuntos `first_pos`, `last_pos` e `follow_pos` seguem à risca a definição formal estipulada em nas seções 2.1.4.1 e 2.1.4.2, com implementações detalhadas no Anexo A.1.3.4.

3.2.3.2 Classe SyntaxTree

A árvore de análise léxica do projeto é gerada de maneira encadeada – um nodo aponta para seus filhos possivelmente não-nulos `left` e `right`, como em (AHO *et al.*, 2013, p. 69, seção 2.5.1) e visto na seção 2.1.4.1. A fim de percorrer uma `SyntaxTree`, portanto, deve-se realizar isso de maneira recursiva a partir de sua `root`.

```
1  class SyntaxTree:
2
3      def __init__(self, regex):
4          digest, self.terminals = eat(regex)
5          self.root: Node = _tree_from(digest)
6          _add_serials_to(self)
```

Com o resultado de `eat(regex)`, na linha 4, acima, pode-se realizar a construção da árvore, feita de maneira recursiva, por meio do uso de `_tree_from(regex)` na linha 5, acima.

```
1  def _tree_from(regex):
2      left, right = sides_for('|', regex)
3      if left:
4          return Node('|', _tree_from(left), _tree_from(right))
5
6      left, right = sides_for('.', regex)
7      if left:
8          return Node('.', _tree_from(left), _tree_from(right))
9
10     left, _ = sides_for('*', regex)
11     if left:
12         return Node('*', _tree_from(left))
13
14     left, _ = sides_for('?', regex)
15     if left:
16         return Node('|', _tree_from(left), Node('&'))
17
18     if regex[0] == '(' and regex[-1] == ')':
19         return _tree_from(regex[1:-1])
20
21     return Node(regex[0])
```

A função `sides_for(operator, regex)` retorna os *substrings* aos lados esquerdo e direito de um dado operador. Caso ocorra chamada desta última com um operador

que aponta a um único nodo-filho, *e. g.* operador de fecho-de-Kleene (linha 10, acima), será sempre `left` que receberá o resultado em `_tree_from(regex)`, enquanto `right` permanecerá como `None`. Se porventura houver parametrização com uma `regex` com parênteses tanto à esquerda quanto à direita, procurar-se-ia apenas aquilo que se encontra dentro dos parênteses a fim de construir a árvore (linha 19, acima).

Não obstante às funcionalidades mencionadas, deve-se também realizar uma busca em profundidade com `_add_serials_to(SyntaxTree)` (linha 6 do trecho de código referente ao construtor da classe `SyntaxTree`) para dar aos terminais, da esquerda à direita, identificadores inteiros únicos a fim de criar seus conjuntos `first_pos` e `last_pos`.

```
1  def _add_serials_to(tree: SyntaxTree):
2  """Adds a serial number to every terminal,
3  from left to right, in a tree."""
4      serial = 1
5
6      def seek_from(node):
7          """Ints are immutable, so we need to declare an
8          outer scope variable in order to change it through
9          recursion."""
10         nonlocal serial
11
12         left, right = node.left, node.right
13
14         if left:
15             if left.regex_symbol not in tree.terminals:
16                 seek_from(left)
17             else:
18                 left.serial_number = serial
19                 serial += 1
20
21         if right:
22             if right.regex_symbol not in tree.terminals:
23                 seek_from(right)
24             else:
25                 right.serial_number = serial
26                 serial += 1
27
28     seek_from(tree.root)
```

3.2.4 Arquivo `conversion.py`

`conversion.py` tem uma função principal, chamada de `fa_from(regex)`, a qual segue o algoritmo estipulado na seção 2.1.4.3. Sua implementação em código-fonte é extensa e, tendo isto em vista, está incluída, novamente, em sua integralidade, no Anexo A.1.3.1. No entanto, a fim de preservar a facilidade na leitura, encontra-se visível no Algoritmo 2, sua implementação em pseudo-código a fim de prover mais meios de compreensão do algoritmo de conversão ao leitor.

Algorithm 2 fa_from(regex)

```
Seja um autômato finito M e uma árvore binária T
M ← FiniteAutomata()
T ← SyntaxTree()
T.calcula_follow_pos()
Seja U a lista de conjuntos de posições não visitados
Seja R o conjunto DStates visto no Algoritmo 1
initial_state_not_set ← True
while Exista um elemento E em U do
    S ← U.remove_primeiro_elemento()
    for Cada terminal A nos terminais de T associado a uma posição do
        Seja G um conjunto para agrupamento de estados
        for Para cada nodo N em S do
            if N.símbolo_regex = A then
                G ← G ∪ N.follow_pos
            end if
        end for
        Sejam src_state e dst_state estados das posições de S e G
        if G não está em R then
            DStates ← DStates ∪ G
            U ← U ∪ G
            if initial_state_not_set é verdadeiro then
                M.estado_inicial ← src_state
                initial_state_not_set ← False
            end if
        end if
        M.estados ← M.estados ∪ src_state ∪ dst_state
        if Se src_state.rotulo tem o número da posição de # then
            M.estados_finais ← M.estados_finais ∪ src_state
        end if
        if Se dst_state.rotulo tem o número da posição de # then
            M.estados_finais ← M.estados_finais ∪ dst_state
        end if
        if Se as transições de M não tem (src_state, A) até dst_state then
            Adicione essa transição ao autômato resultante
        end if
    end for
end while
```

As únicas liberdades tomadas que diferem este pseudo-código daquele visto na seção 2.1.4.3 foram referentes às funções auxiliares, que são chamadas para remover estados inúteis do autômato a ser criado e, também, para criar um estado com rótulo representado por todos os números de um dado conjunto de inteiros.

3.3 DIRETÓRIO *COMPONENTS*

Todos os diretórios dentro de `components/` são diretamente relacionados à apresentação das estruturas ao *end user*, quando não relacionados apenas à facilitação daquela. A *frontpage* a ser vista pelo usuário pode ser dividida em 3 partes:

- `dash_page_progression`: trata da recepção do *text input* do usuário e geração da árvore interativa desenhada em `Plotly`;
- `dash_follow_pos_table`: constrói os dados da tabela dos `follow_pos` da árvore gerada;
- `dash_fa`: cria tanto a tabela quanto o diagrama de estados representativos de um autômato finito obtido a partir da expressão regular dada pelo usuário.

3.3.1 Diretório `dash_page_progression`

Na aplicação principal, faz-se a apresentação da árvore binária de análise léxica ao usuário por meio do uso da biblioteca `Plotly`. Exemplificado na seção 2.1.4.1, a leitura da estrutura binária deve começar de baixo para cima, da esquerda para a direita. Tendo isto em mente, objetivou-se, a fim de tornar o uso mais didático, permitir que o usuário pudesse acompanhar o algoritmo de conversão nó-a-nó.

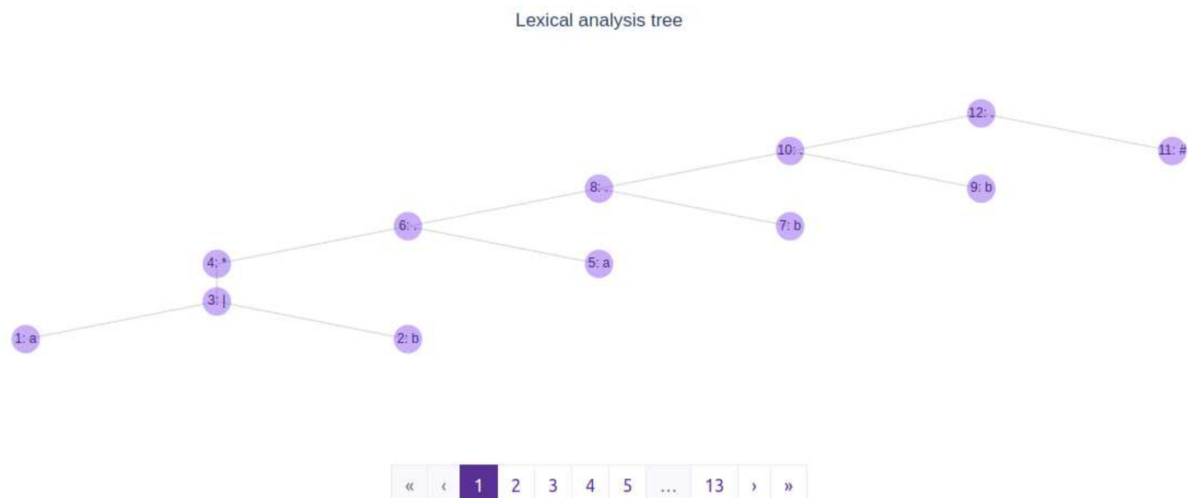

Figura 9 – Árvore binária de análise léxica de `(a|b)*abb` em que todos os nodos não tiveram seus respectivos `first_pos` e `follow_pos` calculados.

Dada a figura 9, pode-se observar um componente `Dash` de paginação abaixo da árvore violeta. Este é responsável por assegurar interatividade do usuário com o processo de conversão de uma `regex` em um autômato finito.

A árvore apresentada representa a expressão regular `(a|b)*abb` e contém 12 nós. Para tanto, a paginação contém uma página para cada nodo a ter seus `first_pos` e `last_pos` calculados e uma página adicional para contemplar a árvore final.

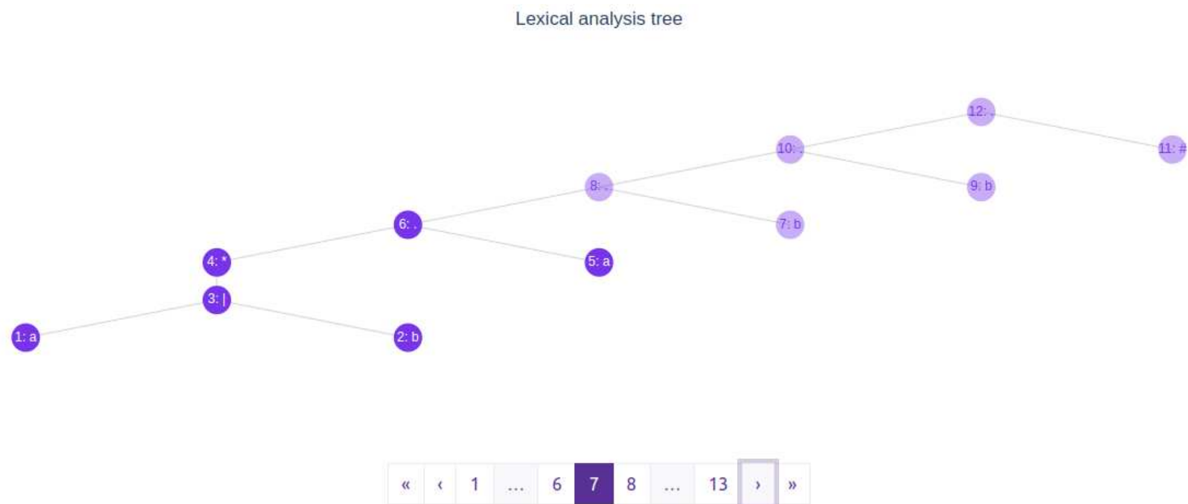

Figura 10 – Árvore da expressão `(a|b)*abb` semi-cheia. Nodos a serem calculados são opacos e, aqueles que já o foram, lúcidos.

Todo o processo descrito até então nesta seção é compreendido pela amálgama dos arquivos presentes no diretório `dash_page_progression`. São 5 arquivos. A ideia central por trás destes é realizar o *wrapping* de um objeto da classe `SyntaxTree` e apresentar o resultado na *frontpage* do *webservice*.

- `node`: contém `DashNode` para fazer o *wrapping* de objetos `Node` de uma `SyntaxTree`;
- `tree`: tem dentro de si `DashTree` para, de modo análogo a `DashNode` com `Node`, realizar o *wrapping* de uma `SyntaxTree`;
- `graph`: módulo da implementação de `graph_from(tree: DashTree)`, a ser detalhada em subseção homônima;
- `figure`: com algumas funções auxiliares e `figure_from(regex)`, assim como o item acima, será detalhado em subseção de mesmo nome;
- `page`: módulo responsável por assegurar a ponte entre o componente de paginação e a interatividade com a árvore apresentada.

3.3.1.1 Arquivo node.py

A classe DashNode age como interface de um Node a uma DashTree.

```
1  class DashNode:
2
3      def __init__(self, node: Node):
4          if node:
5              self.syntax_tree_node = node
6              self.regex_symbol = node.regex_symbol
7
8              self.nullable = node.nullable()
9              self.first_pos, self.last_pos = stringfy(node)
10
11             self.reverse_level_creation_id = 0
12             self.breadth_first_search_id = 0
```

Carrega em si o wrapped Node (linha 5, acima) e, como este, um símbolo representativo da regex (linha 6, acima) e conjuntos first_pos e last_pos – linha 9, acima. No entanto, há novas adições – uma variável booleana que afirma se é ou não anulável ao invés de uma função e, também inteiros que associam um dado DashNode a uma ordem de criação. Recomenda-se dar atenção a estes, visto que são gerados, respectivamente, de acordo com o *snippet* acima, de um *reverse level order traversal* (linha 11) e uma *breadth-first search (BFS)* (linha 12).

O *reverse level order traversal* é realizado para dar um identificador inteiro a cada nó, de baixo para cima, da esquerda à direita, a fim de ligar cada nodo individualmente a uma página da paginação, como visto na figura 10. Em paralelo a isto, a geração de grafos de uma DashTree é realizada pela travessia de todos os nós, a começar pela raiz; faz-se isto por meio de uma BFS e, portanto, a critério de pragmatismo, cada nodo tem um ID que associa este à sua ordem dentro da busca em largura.

3.3.1.2 Arquivo tree.py

Compreendida a classe DashNode, prossegue-se à DashTree.

```
1  class DashTree:
2
3      def __init__(self, regex):
4          self.regex = regex
5          self.syntax_tree = SyntaxTree(self.regex)
6          self.nodes: dict = self.dash_nodes()
```

Uma `DashTree` tem uma `regex` (linha 4, acima) a qual é associada, uma árvore de análise léxica sobre a qual realiza *wrapping* (linha 5, acima) e seu dicionário de `DashNodes` (linha 6, acima). Quis-se fazer uso de um dicionário pois quer-se-ia associar cada nodo a um valor inteiro representativo da ordem em que foi criado.

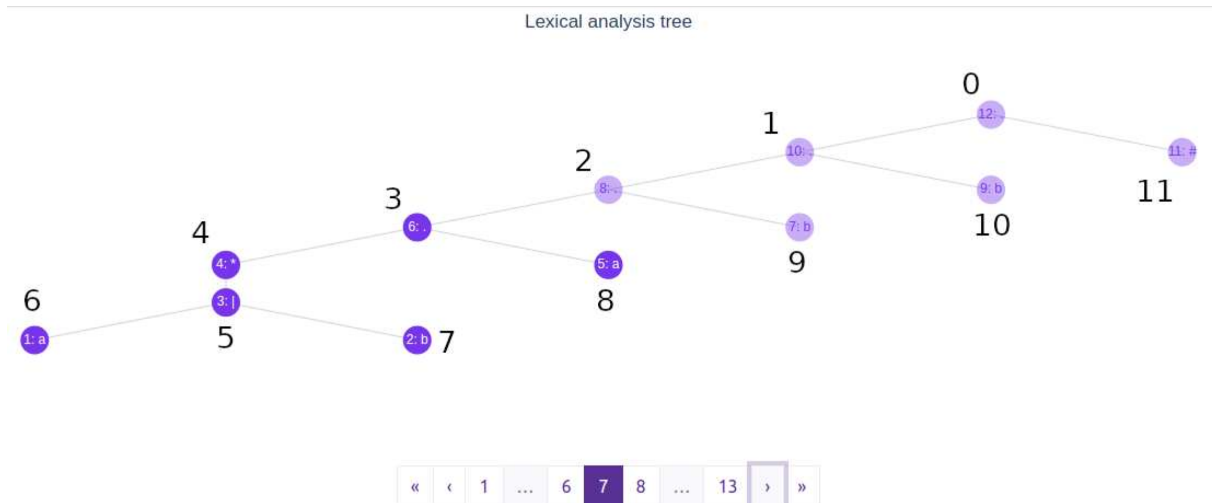

Figura 11 – Árvore com nodos associados à sua ordem de criação

Optou-se por sempre criar `DashTrees`, recursivamente, da esquerda para a direita. Para tanto, os nodos são criados à mesma maneira, da esquerda à direita, da raiz aos nodos que não têm filhos. Funções auxiliares estão em detalhes no Anexo A.1.2.3.5.

```
def binary_tree(self) -> dict:
    binary_tree = OrderedDict()

    syntax_tree_root = self.syntax_tree.root
    root = self.find(syntax_tree_root)

    def build_from(node):
        nonlocal binary_tree

        left = self.find(node.syntax_tree_node.left)
        right = self.find(node.syntax_tree_node.right)
        binary_tree[node] = (left, right)

        if left:
            build_from(left)

        if right:
```

```
            build_from(right)

    build_from(root)
    return dict(binary_tree)
```

3.3.1.3 Arquivo graph.py

Com apenas `graph_from(tree: DashTree)`, `graph.py` é o módulo responsável por tratar da conversão de uma `DashTree` em um grafo direcionado a ser utilizado na criação da figura a ser apresentada ao usuário.

```
def graph_from(tree: DashTree):
    graph = Graph(directed=True)

    nodes = tree.binary_tree()
    vertices = set()

    root: DashNode = tree.root()

    ...
    # Implementation of build_graph_from(node)
    ...

    build_graph_from(root)

    # Gathers all vertices without edges and...
    to_delete_ids = [v.index for v in graph.vs if v.degree() == 0]

    # Removes them from the graph.
    graph.delete_vertices(to_delete_ids)

    return graph
```

Um grafo direcionado, para uma `DashTree`, é criado de maneira encadeada, vide código-fonte abaixo. Nodos são colocados no grafo por meio de uma busca em largura. O trecho de código a seguir encontra-se inserido na linha 10 do *code snippet* acima.

```
    def build_graph_from(node):
        nonlocal graph, nodes, vertices

        current = str(node.breadth_first_search_id)
```

```
5
6         if current not in vertices:
7             graph.add_vertex(current)
8             vertices |= {current}
9
10        left_children: DashNode
11        right_children: DashNode
12        left_children, right_children = nodes[node]
13
14        if left_children:
15            left = str(left_children.breadth_first_search_id)
16
17            graph.add_vertex(left)
18            graph.add_edge(current, left)
19            build_graph_from(left_children)
20
21        if right_children:
22            right = str(right_children.breadth_first_search_id)
23
24            graph.add_vertex(right)
25            graph.add_edge(current, right)
26            build_graph_from(right_children)
```

3.3.1.4 Arquivo `figure.py`

Uma figura `plotly.graph_objects.Figure` é compatível com aplicações `Dash`. Não obstante a isto, `Plotly` permite customização de suas estruturas. Para tanto, uma `Figure` soou como a escolha ideal na apresentação da árvore de análise léxica – quer-se-ia fixar a raiz desta no meio de uma figura, de modo que todos os nodos ficassem igualmente fisicamente distantes.

```
1 def figure_from(regex):
2     tree = DashTree(regex)
3
4     fig = go.Figure()
5
6     Xn, Yn = __build_node_coordinates_for(tree)
7     Xe, Ye = __build_edge_coordinates_for(tree)
8
9     __edge_trace_for(fig, Xe, Ye)
```

```
10
11      dash_nodes = tree.nodes
12      __node_trace_for(fig, Xn, Yn, dash_nodes)
13
14      fig.update_layout(title_text='Lexical analysis tree',
15                        title_x=0.5,
16                        font_size=12,
17                        showlegend=False,
18                        xaxis_visible=False,
19                        yaxis_visible=False,
20                        margin=dict(l=40, r=40, b=85, t=100),
21                        hovermode='closest',
22                        plot_bgcolor='rgb(255,255,255)'
23                        )
24      return fig
```

As funções auxiliares podem ser encontradas no Anexo A.1.2.3.1.

3.3.1.5 Arquivo page.py

`DashPage` é a estrutura final para a representação de todos os componentes de software que envolvem desde a árvore até sua apresentação.

```
1 class DashPage:
2
3     def __init__(self, regex):
4         self.finalized_tree = DashTree(regex)
5         self.final_figure = figure_from(self.finalized_tree.regex)
```

Ela guarda em si, como atributos, apenas sua `DashTree` (linha 4) e a figura finalizada (linha 5). Esta é editada durante a execução, de acordo com a escolha da página da paginação pelo usuário vide função a seguir

```
1 def pagination_notes(self):
2     pagination_notes = [
3         self.finalized_tree.nodes.copy() for _ in range(self.page_quantity())
4                         ]
5
6     for active_page, page_note in enumerate(pagination_notes.copy()):
7         for index, node in enumerate(page_note.values()):
8             node: DashNode
```

```
9
10            if node.reverse_level_creation_id > active_page:
11                pagination_notes[active_page][index] = None
12            else:
13                pagination_notes[active_page][index] = str(node)
14
15    return pagination_notes
```

`pagination_notes(DashPage)` é a função que assegura o comportamento de alteração da figura relativa à árvore binária. Altera-se a figura, nodo a nodo, de baixo para cima, da esquerda para a direita.

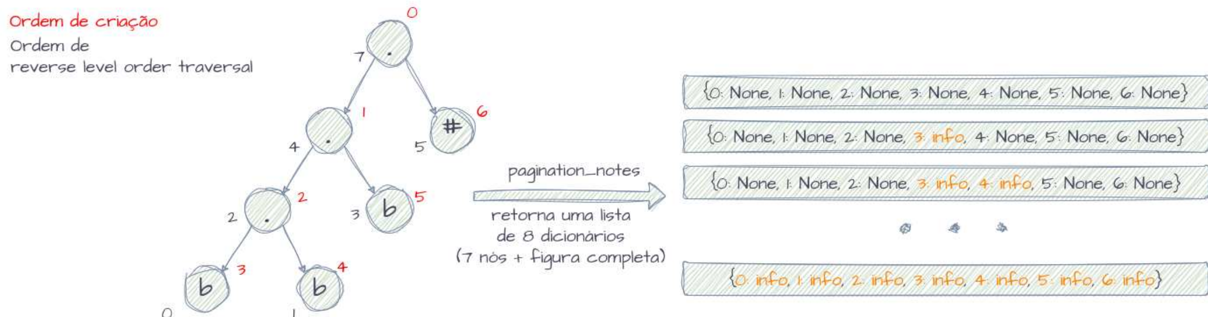

Figura 12 – Exemplificação do funcionamento de `page_notes(DashPage)` para a `regex` `bbb`. Divide-se a árvore tanto pela ordem de criação dos nós quanto pela ordem do *reverse level order traversal*

.

A cada alteração da pagina do elemento de paginação, também alteram-se a opacidade e a cor dos nodos. Ambas as funções que lidam com isso apenas fazem uso de uma cópia do valor de retorno da função exemplificada e, para valores `not None`, segundo a figura 12, fazem a escrita da opacidade ou da cor de um nodo em uma dada posição. Podem ser encontradas no A.1.2.3.4.

3.3.2 Diretório `dash_follow_pos_table`

Contém um único arquivo: `table.py`, com apenas 2 funções. A auxiliar, como de praxe, é delegada ao Anexo A.1.2.2.1, enquanto detalha-se `follow_pos_data_from(tree: DashTree)`.

Dado o tamanho da função mencionada, ela foi dividida em *chunks* menores a fim de facilitar a compreensão.

Num primeiro passo, calculam-se os conjuntos `follow_pos` da `SyntaxTree` *wrapped* dentro da `DashTree` passada como parâmetro. Fez-se uso de uma fila a fim de visitar os nodos da árvore.

```python
def follow_pos_data_from(tree: DashTree):
    tree.syntax_tree.root.calculate_follow_pos()

    follow_pos = dict()  # Needs to be a list of dicts.

    root = tree.syntax_tree.root
    queue = [root]
```

Enquanto houver nodos na fila (linha 1, abaixo), deve-se dar atenção ao primeiro, no índice 0, e, então, colocar seu conjunto `follow_pos` dentro do dicionário homônimo. Faz-se a verificação (linhas 5 e 7, abaixo) se, a partir do nodo atualmente observado, pode-se alcançar um nodo-filho não-nulo e, caso possível, este é colocado dentro da fila (linhas 6 e 8, abaixo).

```python
    while queue:
        current = queue.pop(0)
        follow_pos[current] = current.follow_pos

        if current.left:
            queue.append(current.left)
        if current.right:
            queue.append(current.right)
```

Como não deseja-se obter `follow_pos` de nodos representativos de operadores dentro de uma `regex`, realiza-se a remoção do conjunto de mesmo nome destes abaixo.

```python
    for key in follow_pos.copy():
            if key.regex_symbol in {'.', '*', '|'}:
                del follow_pos[key]
```

Um dicionário em Python é uma estrutura *hash* desordenada. Para facilitar o entendimento da tabela pelo usuário, ordena-se este de acordo com os valores dos inteiros representativos dos nodos terminais da árvore de análise léxica dada. Para todo conjunto `follow_pos`, faz-se o *squash* de todos os inteiros armazenados naquele em um único *string*.

```python
    unorganized_follow_pos: dict = follow_pos.copy()
    follow_pos = dict()

    for terminal_node, follow_pos_set in unorganized_follow_pos.items():
        follow_pos[terminal_node.serial_number] = \
            (
```

```
7                skin_individual_follow_pos_set(follow_pos_set)
8                )
```

Por fim, dado o dicionário ordenado, gera-se uma lista de dicionários a ser retornada que compreende a estrutura de colunas da tabela a ser apresentada ao usuário. A tabela tem apenas 2 colunas – `Node n` e `follow_pos(n)`.

```
1     sorted_follow_pos = dict(sorted(follow_pos.items()))
2     return [
3         {'Node n': k, 'follow_pos(n)': v}
4         for k, v in sorted_follow_pos.items()
5            ]
```

3.3.3 Diretório `dash_fa`

Tendo em vista que um autômato finito pode ser representado tanto por uma tabela de transição quanto que por um diagrama de estados, cada representação tem seu respectivo arquivo de código. Ambos independem um do outro.

3.3.3.1 Arquivo `table.py`

A funcionalidade responsável por colocar um autômato finito no formato da tabela de transições é `fa_data_from_(tree: DashTree)`.

```
1 def fa_table_data_from(tree: DashTree):
2     regex = tree.regex
3     fa = fa_from(regex)
4     return [
5         {
6          '(source_state, symbol)': '(' + str(k[0]) + ', ' + k[1] + ')',
7          'destiny_state': str(v.pop())
8          }
9         for k, v in fa.transitions.items()]
```

Com um autômato finito obtido de uma expressão regular, apenas organiza-se este para, mais uma vez, se adequar à estrutura das colunas da tabela a ser gerada. As colunas são 2 – `(source_state, symbol)` e `destiny_state`. No dicionário de transições iterado, com a variável `k` (linha 9) sendo uma tupla de (`State`, `str`), deve-se fazer o *unwrapping* desta por meio de `k[0]` e `k[1]`.

3.3.3.2 Arquivo `figure.py`

A figura de um autômato finito, diferentemente da de uma `DashTree`, não faz uso da biblioteca `Plotly`, e, sim, de `graphviz`. Enquanto a alta capacidade de parametrização e customização oferecida por `Plotly` é bem-vinda quando se quer ter controle total sobre o grafo gerado, são estas características que também a tornam de difícil manuseio.

Utilizou-se esta na construção da árvore a ser apresentada para o usuário, pois quer-se-ia controlar a orientação do grafo – a raiz tinha que ser o nodo mais alto, de uma árvore binária em que nodos têm igual distanciamento físico um do outro na figura. Quando se trata dos autômatos, contudo, a forma do grafo não foi levada em consideração. Bastou-se que fosse um grafo dirigido, com um estado inicial apontado por uma seta que não é oriunda de nenhum outro estado, e estados finais denotados por duas circunferências concêntricas.

Destarte, há a geração de um autômato finito representado por um dígrafo. É gerado como uma *image* em `.png` e guardado no diretório de `images/`, do código-fonte. A função `fa_state_diagram_from(tree: DashTree)`, responsável pela geração de uma imagem de um autômato finito, é encontrável em sua integralidade no Anexo A.1.2.1.1.

3.4 *ENTRY POINT* APP.PY

O arquivo `app.py` age como o *entry point* principal da aplicação. Para garantir interatividade, implementou-se 2 estruturas chamadas por *callbacks*, responsáveis por garantir o controle da aplicação todas as vezes que o estado de algum componente `html` muda. O objetivo destes é:

- Obter dados da `regex` dada pelo usuário, gerar a figura da árvore de análise léxica, dar *unhide* em elementos até então escondidos antes de *user input* e apresentar as tabelas de `follow_pos` e de *finite automata*, assim como o diagrama de estados deste último;
- Alterar a figura de acordo com a página ativa escolhida pelo usuário no elemento visual de paginação localizado logo abaixo da árvore binária de análise léxica gerada no item anterior.

`app.py` pode ser encontrado em sua integralidade em A.2.

4 GUIA DE USO

O guia de uso pode ser dividido àqueles que desejam adicionar novas *features* e àqueles que apenas almejam utilizar o *webservice* em decorrência da didática e do aprendizado. O código pode ser encontrado em sua integralidade na `main branch` deste repositório: https://github.com/vdella/rosetta.

4.1 PARA *DEVELOPERS*

Caso haja ambição de prover alterações ao projeto, deve-se, em primeira mão, instalar uma versão maior ou igual à 3.10 da linguagem de programação `Python`. Assumindo utilização de um sistema operacional Linux Ubuntu 22.04 a interface de desenvolvimento PyCharm (PYCHARM. . . , s.d.), pode- se obter `Python3.10` por meio dos seguintes passos.

```
sudo apt update && sudo apt upgrade -y
sudo add-apt-repository ppa:deadsnakes/ppa
sudo apt install python3.10
```

Com a instalação concluída, pode-se usar o próprio *package installer* da versão baixada para obter acesso a 3 bibliotecas: `Plotly`, `Dash` e **graphviz**.

```
pip install plotly==5.18.0
pip install dash
pip install graphviz
```

Com as *libraries* devidamente instaladas, pode-se manusear o código-fonte do repositório mencionado acima e abrir as portas para a realização de quaisquer alterações. Para a execução de `Rosetta` basta executar `App.py` por meio da interface de desenvolvimento do PyCharm. Com a aplicação em execução, colocando-se no papel de discente ou docente, refira-se à próxima seção para uso do software.

4.2 PARA ALUNOS E PROFESSORES

Como usuário discente ou docente, com acesso ao serviço, encontrar-se-ia a página principal que fica em aguardo pela digitação de uma expressão regular.

Rosetta

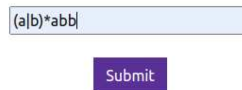

Figura 13 – Aplicação com *regex* a ser interpretada

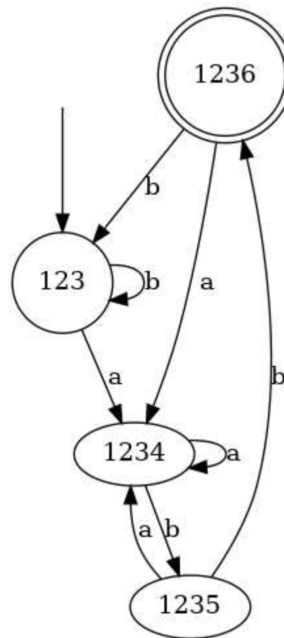

Figura 14 – Autômato gerado a partir de `(a|b)*abb#`.

Expressões regulares terão sempre caracteres unários, *e.g.* a expressão regular *cat* será interpretada sempre como *c.a.t.#*. Isto ocorre para poupar o usuário de ter que colocar o alfabeto de uma dada linguagem, visto que seria necessário reconhecer *cat* como uma palavra completa ao invés das concatenações das letras “c”, “a” e “t”. Além disso, diferentemente das *regexes* explicitadas em (AHO *et al.*, 2013, p. 173, seção 3.9.1), não faz-se necessário adicionar a cerquilha ao fim daquilo que for digitado: por padrão, isto já é feito em conjunto com o englobamento da entrada do usuário por parênteses. Ou seja, toda entrada *abc* será interpretada como *(abc)#*. São tratados como operadores para todas e quaisquer regexes o *pipe* como operador *or* (`|`), o ponto final como concatenação e o asterisco como fecho de Kleene.

Para alunos e professores, basta fazer acesso ao *webservice* e digitar uma *regex* a fim de observar os resultados. Pode-se acessar `Rosetta` em https://rosetta-b4b3447c02da.herokuapp.com/.

5 CONSIDERAÇÕES FINAIS

Neste trabalho providenciou-se o desenvolvimento de um *webservice* que tem como objetivo facilitar o entendimento e aprendizado dos estudantes quanto ao algoritmo de conversão de expressões regulares em autômatos finitos determinísticos (AHO *et al.*, 2013, p. 179). Embora possa-se converter expressões regulares em autômatos finitos não-determinísticos e, então, realizar sua determinização, o algoritmo descrito em software propicia um resultado que não exige a realização de uma segunda operação.

Como adições futuras, poder-se-ia sugerir *features* a serem adicionadas em `Rosetta`, como a geração de uma outra figura de uma árvore preenchida, em que apenas os nodos-folha referentes a um dado *followpos* escolhido seriam coloridos. Poder-se-ia alterar o *followpos* da vez por outro elemento de paginação a ser colocado abaixo da árvore. Este processo agiria como etapa intermediaria da tabela *followpos* e da geração do autômato finito. Não obstante, funcionalidade similar poderia ser criada sobre a construção do autômato finito determinístico final, em que poder-se-ia, também, criar um outro elemento de paginação em que, a cada página da paginação, criar-se-ia uma nova transição até que pudesse-se chegar ao autômato resultado final completo; isto foi desconsiderado neste trabalho no momento atual visto que o AF final é gerado em formato `.png` e não é um componente `Plotly` como a árvore de análise léxica.

Por fim, espera-se tornar o processo de aprendizado e de ensino das disciplinas de Linguagens Formais e Compiladores menos árduo, de modo a facilitar a compreensão por parte dos estudantes e auxiliar o docente na apresentação deste tópico.

REFERÊNCIAS

AHO, A V *et al.* **Compilers: Pearson new international edition**. 2. ed. London, England: Pearson Education, jun. 2013. (Pearson custom library).

ALBORNOZ, Enzo Coelho. Avaliação Automática de Questões sobre Autômatos na Plataforma Moodle, 2022.

D'ANTONI, Loris *et al.* Automata Tutor v3. *In:* LECTURE Notes in Computer Science. [*S.l.*]: Springer International Publishing, 2020. P. 3–14. ISBN 9783030532918. DOI: 10.1007/978-3-030-53291-8_1. Disponível em: http://dx.doi.org/10.1007/978-3-030-53291-8_1.

DASH. [*S.l.*: *s.n.*]. https://dash.plotly.com/. Accessed: 2023-11-12.

GRAPHVIZ. [*S.l.*: *s.n.*]. https://graphviz.org/. Accessed: 2023-11-12.

HOPCROFT, John E.; ULLMAN, Jeffrey D. **Formal Languages and Their Relation to Automata**. USA: Addison-Wesley Longman Publishing Co., Inc., 1969.

PLOTLY Open Source Graphing Library for Python. [*S.l.*: *s.n.*]. https://plotly.com/python/. Accessed: 2023-11-12.

PYCHARM. [*S.l.*: *s.n.*]. https://www.jetbrains.com/pt-br/pycharm/. Accessed: 2023-11-12.

RODGER, Susan H. **Java Formal Language and Automata Package (JFLAP)**. [*S.l.*: *s.n.*], 2018. https://www.jflap.org/.

SIPSER, Michael. **Introduction to the Theory of Computation**. Third. Boston, MA: Course Technology, 2013. ISBN 113318779X.

ANEXO A – ROSETTA EM SUA INTEGRALIDADE

A.1 SRC

A.1.1 automata

A.1.1.1 fa.py

```
from src.automata.state import State


class FiniteAutomata:

    def __init__(self):
        self.initial_state = State()
        self.states = {self.initial_state}
        self.transitions = dict()
        self.final_states = set()
```

A.1.1.2 state.py

```
from time import process_time_ns


class State:

    def __init__(self, label: str = None):
        self.id = process_time_ns()
        self.label = label

    def __str__(self):
        return self.label if self.label is not None else '{}'.format(self.id)

    def __hash__(self):
        return hash(self.label)

    def __eq__(self, other):
        return hash(self.label) == hash(other.label)
```

A.1.2 components

A.1.2.1 dash_fa

A.1.2.1.1 figure.py

```
import graphviz
from src.automata.fa import FiniteAutomata
from src.components.dash_page_progression.tree import DashTree
from src.regex.conversion import fa_from


def fa_state_diagram_from(tree: DashTree):
    fa: FiniteAutomata = fa_from(tree.regex)

    graph = graphviz.Digraph('finite-automata',
                             comment='Finite Automata',
                             format='png',
                             directory='assets')

    useless_states = {state for state in fa.states
                      if state not in fa.transitions.keys()}
    common_states = (fa.states
                     - fa.final_states
                     - {fa.initial_state}
                     - useless_states)

    for state in common_states:
        graph.node(str(state),
                   shape='circle',
                   color='black',
                   style='filled',
                   fillcolor='white')

    only_finals = fa.final_states - {fa.initial_state}
    for state in only_finals:
        graph.node(str(state),
                   shape='doublecircle',
                   color='black',
                   style='filled',
                   fillcolor='white')
```

```
initial_state = fa.initial_state
graph.node(str(initial_state),
           shape='doublecircle'
           if initial_state in fa.final_states else 'circle',
           color='black',
           style='filled',
           fillcolor='white')

for transition, arrival in fa.transitions.items():
    origin, symbol = transition
    for dst in arrival:
        graph.edge(str(origin), str(dst), label=symbol)

graph.node('start', _attributes={'style': 'invis'})
graph.edge('start', str(initial_state))

return graph
```

A.1.2.1.2 table.py

```
from src.components.dash_page_progression.tree import DashTree
from src.regex.conversion import fa_from


def fa_table_data_from(tree: DashTree):
    regex = tree.regex
    fa = fa_from(regex)
    return [
        {
         '(source_state, symbol)': '(' + str(k[0]) + ', ' + k[1] + ')',
         'destiny_state': str(v.pop())
         }
        for k, v in fa.transitions.items()]
```

A.1.2.2 dash_follow_pos_table

A.1.2.2.1 table.py

```
from src.components.dash_page_progression.tree import DashTree


def follow_pos_data_from(tree: DashTree):
    tree.syntax_tree.root.calculate_follow_pos()

    follow_pos = dict()  # Needs to be a list of dicts.

    root = tree.syntax_tree.root
    queue = [root]

    while queue:
        current = queue.pop(0)
        follow_pos[current] = current.follow_pos

        if current.left:
            queue.append(current.left)
        if current.right:
            queue.append(current.right)

    for key in follow_pos.copy():
        if key.regex_symbol in {'.', '*', '|'}:
            del follow_pos[key]

    unorganized_follow_pos: dict = follow_pos.copy()
    follow_pos = dict()

    for terminal_node, follow_pos_set in unorganized_follow_pos.items():
        follow_pos[terminal_node.serial_number] = skin_individual_follow_pos_set(
                                                     follow_pos_set
                                                  )

    sorted_follow_pos = dict(sorted(follow_pos.items()))
    return [
        {'Node n': k, 'follow_pos(n)': v} for k, v in sorted_follow_pos.items()
    ]
```

```
def skin_individual_follow_pos_set(follow_pos_nodes):
    string = [str(node.serial_number) for node in follow_pos_nodes]
    string.sort()

    label = ''.join(string)
    return label
```

A.1.2.3 dash_page_progression

A.1.2.3.1 figure.py

```
import plotly.graph_objects as go
from igraph import Layout, Graph
from src.components.dash_page_progression.tree import DashTree
from src.components.dash_page_progression.node import DashNode
from src.components.dash_page_progression.graph import graph_from


def figure_from(regex):
    tree = DashTree(regex)

    fig = go.Figure()

    Xn, Yn = __build_node_coordinates_for(tree)
    Xe, Ye = __build_edge_coordinates_for(tree)

    __edge_trace_for(fig, Xe, Ye)

    dash_nodes = tree.nodes
    __node_trace_for(fig, Xn, Yn, dash_nodes)

    fig.update_layout(title_text='Lexical analysis tree',
                      title_x=0.5,
                      font_size=12,
                      showlegend=False,
                      xaxis_visible=False,
                      yaxis_visible=False,
                      margin=dict(l=40, r=40, b=85, t=100),
                      hovermode='closest',
                      plot_bgcolor='rgb(255,255,255)'
                      )
    return fig


def __build_node_coordinates_for(tree: DashTree):
    graphed_tree: Graph = graph_from(tree)
    nr_vertices = len(tree.vertices())
```

```
    lay: Layout = graphed_tree.layout_reingold_tilford(root=[str(0)])

    vertices_positions = {k: lay[k] for k in range(nr_vertices)}
    all_target_vertices = [lay[k][1] for k in range(nr_vertices)]
    highest_y_coordinate = max(all_target_vertices)

    L = len(vertices_positions)
    nodes_x_axis = [vertices_positions[k][0] for k in range(L)]
    nodes_y_axis = [2 * highest_y_coordinate - vertices_positions[k][1]
                    for k in range(L)]

    return nodes_x_axis, nodes_y_axis


def __build_edge_coordinates_for(tree: DashTree):
    graphed_tree: Graph = graph_from(tree)
    nr_vertices = len(tree.vertices())

    lay: Layout = graphed_tree.layout_reingold_tilford(root=[str(0)])

    vertices_positions = {k: lay[k] for k in range(nr_vertices)}
    all_target_vertices = [lay[k][1] for k in range(nr_vertices)]
    highest_y_coordinate = max(all_target_vertices)

    edges = [e.tuple for e in graphed_tree.es]

    edges_x_axis = list()
    edges_y_axis = list()

    for edge in edges:
        edges_x_axis += [
            vertices_positions[edge[0]][0],
            vertices_positions[edge[1]][0],
            None]

        edges_y_axis += [
            2 * highest_y_coordinate - vertices_positions[edge[0]][1],
            2 * highest_y_coordinate - vertices_positions[edge[1]][1],
```

```
            None]

    return edges_x_axis, edges_y_axis


def __edge_trace_for(figure, Xe, Ye):
    figure.add_trace(go.Scatter(x=Xe,
                                y=Ye,
                                mode='lines',
                                line=dict(color='rgb(210,210,210)', width=1),
                                hoverinfo='none'
                                ))


def __node_trace_for(figure, Xn, Yn, node_annotations: dict):
    regex_symbols = list()
    nodes_metadata = list()

    for creation_order_id, dash_node in node_annotations.items():
        dash_node: DashNode

        reverse_node_id = dash_node.reverse_level_creation_id
        regex_symbol = dash_node.regex_symbol

        regex_symbols.append(str(reverse_node_id) + ': ' + regex_symbol)
        nodes_metadata.append(str(dash_node))

    figure.add_trace(go.Scatter(x=Xn,
                                y=Yn,
                                mode='markers+text',
                                marker=dict(
                                    size=26,
                                    opacity=0.4,
                                    color='rgb(119, 52, 235)'),
                                textfont=dict(color='rgb(64, 28, 128)'),
                                text=regex_symbols,
                                customdata=nodes_metadata,
                                hovertemplate="%{customdata}<extra></extra>",
                                hoverinfo='text',
```

```
opacity=1.0
))
```

A.1.2.3.2 graph.py

```
from src.components.dash_page_progression.tree import DashTree, DashNode
from igraph import Graph


def graph_from(tree: DashTree):
    graph = Graph(directed=True)

    nodes = tree.binary_tree()
    vertices = set()

    root: DashNode = tree.root()

    def build_graph_from(node):
        nonlocal graph, nodes, vertices

        current = str(node.breadth_first_search_id)

        if current not in vertices:
            graph.add_vertex(current)
            vertices |= {current}

        left_children: DashNode
        right_children: DashNode
        left_children, right_children = nodes[node]

        if left_children:
            left = str(left_children.breadth_first_search_id)

            graph.add_vertex(left)
            graph.add_edge(current, left)
            build_graph_from(left_children)

        if right_children:
            right = str(right_children.breadth_first_search_id)

            graph.add_vertex(right)
            graph.add_edge(current, right)
            build_graph_from(right_children)
```

```
build_graph_from(root)

# Gathers all vertices without edges and...
to_delete_ids = [v.index for v in graph.vs if v.degree() == 0]

# Removes them from the graph.
graph.delete_vertices(to_delete_ids)

return graph
```

A.1.2.3.3 node.py

```
from src.regex.syntax_tree import Node, stringfy


class DashNode:

    def __init__(self, node: Node):
        if node:
            self.syntax_tree_node = node
            self.regex_symbol = node.regex_symbol

            self.nullable = node.nullable()
            self.first_pos, self.last_pos = stringfy(node)

            self.reverse_level_creation_id = 0
            self.breadth_first_search_id = 0

    def __str__(self):
        return (f"nullable: {self.nullable}<br />"
                f"first_pos: {self.first_pos}<br />"
                f"last_pos: {self.last_pos}<br />") if self else ''
```

A.1.2.3.4 page.py

```python
from src.components.dash_page_progression.tree import DashTree
from src.components.dash_page_progression.node import DashNode
from src.components.dash_page_progression.figure import figure_from


class DashPage:

    def __init__(self, regex):
        self.finalized_tree = DashTree(regex)
        self.final_figure = figure_from(self.finalized_tree.regex)

    def pagination_notes(self):
        pagination_notes = [self.finalized_tree.nodes.copy()
                            for _ in range(self.page_quantity())]

        for active_page, page_note in enumerate(pagination_notes.copy()):
            for index, node in enumerate(page_note.values()):
                node: DashNode

                if node.reverse_level_creation_id > active_page:
                    pagination_notes[active_page][index] = None
                else:
                    pagination_notes[active_page][index] = str(node)

        return pagination_notes

    def opacities(self):
        opacities = self.pagination_notes().copy()

        for active_page, page_note in enumerate(self.pagination_notes()):
            for index, note in enumerate(page_note.values()):

                opacities[active_page][index] = 1 if note else 0.4

        return opacities

    def colors(self):
        colors = self.pagination_notes().copy()
```

```
        for active_page, page_note in enumerate(self.pagination_notes()):
            for index, note in enumerate(page_note.values()):

                colors[active_page][index] = 'rgb(255, 255, 255)' if note\
                                             else 'rgb(119, 52, 235)'

        return colors

    @staticmethod
    def empty_dash_page():
        return DashPage('a#')

    def page_quantity(self):
        """Returns the quantity of tree vertices to be iterated."""
        return len(self.finalized_tree.vertices()) + 1
```

A.1.2.3.5 tree.py

```
from src.regex.syntax_tree import eat, SyntaxTree
from src.components.dash_page_progression.node import DashNode
from collections import OrderedDict


class DashTree:

    def __init__(self, regex):
        self.regex = regex
        self.syntax_tree = SyntaxTree(self.regex)
        self.nodes: dict = self.dash_nodes()

    def dash_nodes(self):
        nodes = dict()

        reverse_nodes = self.nodes_by_reverse_level_order_traversal()
        creation_order_nodes = self.nodes_by_creation_order()
        breadth_first_search_ids = self.nodes_by_bfs_order()

        def seek_from(node):
            nonlocal nodes

            if node:

                dash_node = DashNode(node)

                dash_node.reverse_level_creation_id = reverse_nodes[node]
                dash_node.breadth_first_search_id = breadth_first_search_ids[node]

                creation_order_id = creation_order_nodes[node]

                nodes[creation_order_id] = dash_node

                seek_from(node.left)
                seek_from(node.right)

        seek_from(self.syntax_tree.root)
        return dict(sorted(nodes.items()))
```

```
    def find(self, node):
        if node:

            for current in self.nodes.values():
                current: DashNode

                if current.syntax_tree_node == node:
                    return current

        return None

    def root(self):
        return self.find(self.syntax_tree.root)

    def binary_tree(self) -> dict:
        binary_tree = OrderedDict()

        syntax_tree_root = self.syntax_tree.root
        root = self.find(syntax_tree_root)

        def build_from(node):
            nonlocal binary_tree

            left = self.find(node.syntax_tree_node.left)
            right = self.find(node.syntax_tree_node.right)
            binary_tree[node] = (left, right)

            if left:
                build_from(left)

            if right:
                build_from(right)

        build_from(root)
        return dict(binary_tree)

    def nodes_by_creation_order(self):
        nodes_by_ids = dict()
```

```
        root = self.syntax_tree.root
        serial = 0

        def gather_from(node):
            nonlocal nodes_by_ids, serial

            if node not in nodes_by_ids:
                nodes_by_ids[node] = serial
                serial += 1

            left, right = node.left, node.right

            if left:
                gather_from(left)
            if right:
                gather_from(right)

        gather_from(root)

        return nodes_by_ids

    def nodes_by_reverse_level_order_traversal(self):
        if not self.syntax_tree.root:
            return []

        syntax_tree = self.syntax_tree

        serial = len(self.vertices())
        queue = [(syntax_tree.root, serial)]
        stack = []
        traversal = []

        while queue:
            current, serial = queue.pop(0)
            stack.append((current, serial))

            if current.right:
                serial -= 1
```

```
                queue.append((current.right, serial))
            if current.left:
                serial -= 1
                queue.append((current.left, serial))

        while stack:
            traversal.append(stack.pop())

        return dict(traversal)

    def nodes_by_bfs_order(self):
        if not self.syntax_tree.root:
            return []

        syntax_tree = self.syntax_tree

        serial = 0
        to_be_visited_queue = [syntax_tree.root]

        nodes_by_ids = {syntax_tree.root: serial}

        while to_be_visited_queue:
            current = to_be_visited_queue.pop(0)

            if current.left:
                to_be_visited_queue.append(current.left)
                serial += 1
                nodes_by_ids[current.left] = serial

            if current.right:
                to_be_visited_queue.append(current.right)
                serial += 1
                nodes_by_ids[current.right] = serial

        return nodes_by_ids

    def vertices(self) -> list:
        nodes, _ = eat(self.regex)
```

```
    # Parenthesis should not be counted as tree nodes.
    parenthesis = {'(', ')'}
    nodes = [e for e in nodes if e not in parenthesis]

    return nodes
```

A.1.3 `regex`

A.1.3.1 `conversion.py`

```
from src.regex.syntax_tree import SyntaxTree
from src.automata.fa import FiniteAutomata, State


def fa_from(regex) -> FiniteAutomata:
    result = FiniteAutomata()

    tree = SyntaxTree(regex)
    final_states_id = str(tree.root.right.serial_number)

    tree.root.calculate_follow_pos()

    unmarked = [tree.root.first_pos()]
    d_states = [tree.root.first_pos()]

    initial_state_not_set = True

    while unmarked:
        first_pos = unmarked.pop(0)

        for terminal in tree.terminals:
            gatherer = set()

            for node in first_pos:
                if node.regex_symbol == terminal:
                    gatherer |= node.follow_pos

            src_state = __squash_into_state(first_pos)
            dst_state = __squash_into_state(gatherer)

            if gatherer not in d_states:
                d_states.append(gatherer)
                unmarked.append(gatherer)

                if initial_state_not_set:
                    result.initial_state = src_state
```

```
                    initial_state_not_set = False

                result.states |= {src_state} | {dst_state}

                if final_states_id in src_state.label:
                    result.final_states |= {src_state}
                if final_states_id in dst_state.label:
                    result.final_states |= {dst_state}

            if not result.transitions.get((src_state, terminal)):
                result.transitions[(src_state, terminal)] = {dst_state}

    __remove_useless_states(result)

    return result


def __remove_useless_states(fa: FiniteAutomata):
    # Copy as we will change the size of fa.transactions during iteration.
    cached_transitions = dict(fa.transitions)

    for key in cached_transitions.keys():
        state, symbol = key

        # The resulting automata will always be deterministic.
        # Thus, the destiny set of states contains only one.
        # If that said destiny state has an empty string as
        # a label, it's useless and needs to be removed.
        arrival = list(cached_transitions[key])[0]

        if state.label == '' or symbol == '#' or arrival.label == '':
            del fa.transitions[key]


def __squash_into_state(nodes: set) -> State:
    """:param nodes: as the symbols from the first_pos() of a node.
    :returns: a state with all symbols joined by '-' as labels."""
    string = [str(node.serial_number) for node in nodes]
    string.sort()
```

```
    label = ''.join(string)
    return State(label)
```

A.1.3.2 format.py

```
operators = {'*', '.', '|', '+', '?'}
parenthesis = {'(', ')'}
non_terminals = operators | parenthesis


def eat(regex) -> list and set:
    """Eats a non formatted regex and returns its list digested form
    with its terminal symbols."""

    if len(regex) == 1:
        # If a regex is 1 char long, that regex is already digested.
        return [regex], {regex}

    if regex[-1] == '#':  # No need to add # if the regex already has it.
        digest = __trim_blank_spaces(regex)
    else:
        digest = __trim_blank_spaces(regex) + '#'

    digest = __add_missing_concatenations(digest)
    return list(digest), __terminals_from(digest) | {'&'}


def __add_missing_concatenations(regex):
    stripped = list(regex.replace('.', ''))

    # Join all strings in a list of strings with periods.
    concatenated = '.'.join(stripped)

    # As we added '.' between every string, we need to trim the wrong additions.
    return (concatenated.replace('(.', '(').
            replace('.)', ')').
            replace('.|.', '|').
            replace('.*', '*').
            replace('.?', '?'))


def __left_parenthesis_surrounded(regex: str, place) -> bool:
    """As the missing concatenations will be added by the left of a symbol,
```

```
    we have to avoid adding concatenations at its left if there is a '('."""
    return regex[place - 1] == '(' and regex[place] in __terminals_from(regex)


def __terminals_from(regex) -> set:
    """Scans a regex and gathers its terminals inside a set."""
    terminals = set()

    for symbol in regex:
        if symbol not in operators and symbol not in parenthesis:
            terminals.add(symbol)
    return terminals


def sides_for(operator, regex):
    """:returns the inner regexes of a :param regex
    at the side of a given :param operator"""

    left_tree, right_tree = str(), str()
    parenthesis_count = 0

    # We'll be looking from right to left.
    for i in range(len(regex) - 1, -1, -1):
        if regex[i] == operator and parenthesis_count == 0:
            left_tree = regex[:i]
            return left_tree, right_tree[::-1]

        if regex[i] == ')':
            parenthesis_count += 1
        elif regex[i] == '(':
            parenthesis_count -= 1

        right_tree += regex[i]

    return left_tree, right_tree[::-1]


def __trim_blank_spaces(regex: str) -> str:
    return regex.replace(' ', '')
```

A.1.3.3 sanitization.py

```
def is_correct(regex):
    return (are_parenthesis_correct_for(regex) and
            are_pipes_correct_for(regex) and
            are_concatenations_correct_for(regex) and
            are_stars_correct_for(regex) and
            not is_using_question_mark_operator(regex))


def are_parenthesis_correct_for(regex):
    parenthesis_count = 0

    sentence = regex.replace(' ', '')

    for letter in sentence:
        if letter == '(':
            parenthesis_count += 1
        elif letter == ')':
            parenthesis_count -= 1

    return parenthesis_count == 0


def are_pipes_correct_for(regex):
    return ('||' not in regex
            and '(|' not in regex
            and '|)' not in regex
            and '.|' not in regex
            and '|.' not in regex
            and '*|' not in regex
            and '|*' not in regex) and regex[0] != '|' and regex[-1] != '|'


def are_concatenations_correct_for(regex):
    return ('..' not in regex
            and '(.' not in regex
            and '.)' not in regex
            and '*.' not in regex
            and '.*' not in regex) and regex[0] != '.' and regex[-1] != '.'
```

```
def are_stars_correct_for(regex):
    return ('**' not in regex
            and '(*' not in regex
            and '*)' not in regex) and regex[0] != '*'


def is_using_question_mark_operator(regex):
    return '?' in regex
```

A.1.3.4 syntax_tree.py

```
from src.regex.format import eat, sides_for


class SyntaxTree:

    def __init__(self, regex):
        digest, self.terminals = eat(regex)
        self.root: Node = _tree_from(digest)
        _add_serials_to(self)


def _add_serials_to(tree: SyntaxTree):
    """Adds a serial number to every terminal,
    from left to right, in a :param tree:."""
    serial = 1

    def seek_from(node):
        # Ints are immutable, so we need to declare an outer
        # scope variable in order to change it through recursion.
        nonlocal serial

        left, right = node.left, node.right

        if left:
            if left.regex_symbol not in tree.terminals:
                seek_from(left)
            else:
```

```
                left.serial_number = serial
                serial += 1

        if right:
            if right.regex_symbol not in tree.terminals:
                seek_from(right)
            else:
                right.serial_number = serial
                serial += 1

    seek_from(tree.root)
    tree.root.serial_number = serial


def _tree_from(regex):
    left, right = sides_for('|', regex)
    if left:
        return Node('|', _tree_from(left), _tree_from(right))

    left, right = sides_for('.', regex)
    if left:
        return Node('.', _tree_from(left), _tree_from(right))

    left, _ = sides_for('*', regex)
    if left:
        return Node('*', _tree_from(left))

    left, _ = sides_for('?', regex)
    if left:
        return Node('|', _tree_from(left), Node('&'))

    if regex[0] == '(' and regex[-1] == ')':
        return _tree_from(regex[1:-1])

    return Node(regex[0])


class Node:
```

```
    def __init__(self, symbol: str, left=None, right=None):
        self.regex_symbol = symbol
        self.serial_number = 0

        self.left: Node = left
        self.right: Node = right

        self.follow_pos = set()

    def __str__(self):
        """Returns a node's regex symbol with its
        first_pos() and last_pos() by its sides."""
        str_first_pos, str_last_pos = stringfy(self)
        return str_first_pos + ' ' + self.regex_symbol + ' ' + str_last_pos

    def nullable(self) -> bool:
        if self.regex_symbol == '&' or self.regex_symbol == '*':
            return True
        elif self.regex_symbol == ".":
            return self.left.nullable() and self.right.nullable()
        elif self.regex_symbol == "|":
            return self.left.nullable() or self.right.nullable()
        return False

    def first_pos(self) -> set:
        match self.regex_symbol:
            case '&':
                return set()
            case '*':
                return self.left.first_pos()
            case '.':
                if self.left.nullable():
                    return self.left.first_pos() | self.right.first_pos()
                else:
                    return self.left.first_pos()
            case '|':
                return self.left.first_pos() | self.right.first_pos()
            case _:
                return {self}
```

```
    def last_pos(self) -> set:
        match self.regex_symbol:
            case '&':
                return set()
            case '*':
                return self.left.last_pos()
            case '.':
                if self.right.nullable():
                    return self.left.last_pos() | self.right.last_pos()
                else:
                    return self.right.last_pos()
            case '|':
                return self.left.last_pos() | self.right.last_pos()
            case _:
                return {self}

    def calculate_follow_pos(self) -> ():
        for children in {self.left, self.right}:
            if children:
                children.calculate_follow_pos()

        if self.regex_symbol == '.':
            for children in self.left.last_pos():
                children.follow_pos |= self.right.first_pos()

        elif self.regex_symbol == '*':
            for children in self.last_pos():
                children.follow_pos |= self.first_pos()


def stringfy(node):
    """returns the first_pos() and last_pos() of a node as strings
    in order to print them in Node.__str__()."""
    first_pos = [n.serial_number for n in node.first_pos()]
    first_pos.sort()

    last_pos = [n.serial_number for n in node.last_pos()]
    last_pos.sort()
```

```
    return str(first_pos), str(last_pos)
```

A.2 APP.PY

```
from dash import Dash, html, dcc, Input, Output, callback, dash
import os
import shutil
from src.components.dash_fa.figure import fa_state_diagram_from
from src.components.dash_fa.table import fa_table_data_from
from src.components.dash_page_progression.page import DashPage
from src.components.dash_follow_pos_table.table import follow_pos_data_from
import dash_bootstrap_components as dbc
from plotly.graph_objects import Figure

page = DashPage.empty_dash_page()

external_stylesheets = [dbc.themes.PULSE]
app = Dash(__name__, external_stylesheets=external_stylesheets)

app.title = 'Rosetta'

server = app.server

app.layout = html.Div(
    html.Div([
        html.Br(),

        html.H1(children='Rosetta', style={'textAlign': 'center'}),

        html.Br(),
        html.Br(),

        html.Div(
            children=[
                dcc.Input(
                    id='regex-input',
                    placeholder='Type your regex! (e.g. (a|b)*abb#)',
                    debounce=True,
                    readOnly=False),
            ],
            style={'textAlign': 'center'},
        ),
```

```
        html.Br(),

        html.Div(
            children=[
                dbc.Button('Submit', id='submit-button'),
            ],
            style={'textAlign': 'center'}),

        html.Br(),
        html.Br(),

        html.Div(
            id='figure-parent',
            children=[
                dcc.Graph(
                    id='bin-tree',
                    figure={},
                ),
            ],
            hidden=True,
        ),

        html.Div(
            id='tree-page-handler',
            children=[
                dbc.Pagination(
                    id='pagination',
                    max_value=0,
                    first_last=True,
                    previous_next=True,
                    fully_expanded=False),
            ],
            style={'display': 'flex', 'justifyContent': 'center'},
            hidden=True
        ),

        html.Br(),
        html.Br(),
```

```
html.Div(
    id='follow-pos-table-parent',
    children=[
        dash.dash_table.DataTable(
                id='follow-pos-table',
                columns=[
                    {'name': i, 'id': i}
                    for i in ['Node n', 'follow_pos(n)']],
                data=[],
            ),
    ],
    style={'display': 'flex', 'justifyContent': 'center'},
    hidden=True,
),

html.Br(),
html.Br(),

html.Div(
    id='fa-parent',
    children=[
        dash.dash_table.DataTable(
            id='fa-table',
            columns=[
                {'name': i, 'id': i}
                for i in ['(source_state, symbol)', 'destiny_state']],
            data=[],
        ),

        html.Img(
            id='fa-img',
            src='',
            style={"margin-left": "100px"}
        )
    ],
    style={'display': 'flex', 'justifyContent': 'center'},
    hidden=True,
),
```

```
        html.Br(),
        html.Br(),
    ])
)


@callback(
    Output('figure-parent', 'hidden'),
    Output('tree-page-handler', 'hidden'),
    Output('follow-pos-table-parent', 'hidden'),
    Output('fa-parent', 'hidden'),

    Output('pagination', 'max_value'),
    Output('bin-tree', 'figure', allow_duplicate=True),
    Output('follow-pos-table', 'data'),
    Output('fa-table', 'data'),
    Output('fa-img', 'src'),

    Output('regex-input', 'readOnly'),

    Input('regex-input', 'value'),
    prevent_initial_call=True)
def create_figure_from(user_text_entry):
    (hidden_figure,
     hidden_page_handler,
     hidden_follow_pos,
     hidden_fa_table) = True, True, True, True

    read_only_text_input = True

    if user_text_entry:
        global page
        page = DashPage(user_text_entry)

        fa_diagram = fa_state_diagram_from(page.finalized_tree)

        if os.path.exists('assets/finite-automata.gv.png'):
            os.remove('assets/finite-automata.gv.png')
```

```
        fa_diagram.render()

        return (not hidden_figure,
                not hidden_page_handler,
                not hidden_follow_pos,
                not hidden_fa_table,

                page.page_quantity(),
                page.final_figure,
                follow_pos_data_from(page.finalized_tree),
                fa_table_data_from(page.finalized_tree),
                app.get_asset_url('finite-automata.gv.png'),

                read_only_text_input)
    return (hidden_figure,
            hidden_page_handler,
            hidden_follow_pos,
            hidden_fa_table,

            0,
            {},
            [],
            [],
            '',

            not read_only_text_input)


@callback(
    Output('bin-tree', 'figure'),

    Input('pagination', 'active_page'),
    Input('bin-tree', 'figure')
)
def update_figure(active_page, bin_tree_figure):
    bin_tree_figure = Figure(bin_tree_figure)  # Has to load its data; comes as a

    if active_page:
```

```
        page_note = page.pagination_notes()[active_page - 1].values()

        opacity = page.opacities()[active_page - 1].values()

        colors = page.colors()[active_page - 1].values()

        bin_tree_figure.update_traces(hovertemplate=list(page_note))
        bin_tree_figure.update_traces(marker=dict(opacity=list(opacity)))
        bin_tree_figure.update_traces(textfont=dict(color=list(colors)))
    else:
        bin_tree_figure.update_traces(hoverinfo='none')

    return bin_tree_figure


if __name__ == '__main__':
    app.run(debug=True)
```

ANEXO B – ARTIGO

Rosetta: um webservice didático para auxiliar no aprendizado da conversão de expressões regulares em autômatos finitos determinísticos

Vitor Della Torre dos Santos[1]

[1]Departamento de Informática e Estatística
Universidade Federal de Santa Catarina (UFSC)
Santa Catarina – SC – Brazil

Abstract. *The famous book Compilers: Principles, Techniques and Tools carries a hard-to-use and hard-to-develop algorithm: converting regexes to deterministic finite automata. This stands as an obstacle to whom may try the Regular Languages and Compilers classes at the Federal University of Santa Catarina, as the transformation process is error prone when done by hand due to the excessive quantity of details that must be taken into account by the students. With that being said, surged the initiative of describing it step-by-step in a webservice in order to make it easy to reach and easy to understand, trying to enhance the learning process.*

Resumo. *O célebre livro Compilers: Principles, Techniques and Tools carrega em si um algoritmo de difícil manuseio e implementação: a conversão de expressões regulares em autômatos finitos determinísticos. Esta se põe como desafio àqueles que fazem a cadeira de Linguagens Formais e Compiladores na Universidade Federal de Santa Catarina, tendo em vista que o processo de transformação é propenso a erros quando feito no papel em decorrência do excesso de minúcias que devem ser levadas em consideração pelos alunos. Tendo isto posto, teve-se a iniciativa de descrever todos os passos do dito processo de conversão em um webservice a fim de ser de fácil compreensão e de fácil alcance, de modo a ajudar na didática e no aprendizado.*

1. Introdução

Em geral cursos de Bacharelado em Ciência da Computação, assim como tantos outro cursos de nível superior, apresentam disciplinas desafiadoras. Em Ciência da Computação, as disciplinas que abordam Linguagens Formais e Compiladores, devido as suas dualidades entre teoria e prática, enquadram-se neste grupo. Tais disciplinas abordam conteúdos relacionados à definição de aspectos formais de linguagens de programação e à construção de seus compiladores.

Dentre os algoritmos estudados em Linguagens Formais e Compiladores, há aqueles que são melhor acompanhados com a ajuda de extensos testes de mesa. Um destes é aquele que diz respeito ao processo de conversão de expressões regulares em autômatos finitos determinísticos [Aho et al. 2013, p. 179]. Para seu aprendizado e utilização, construir uma árvore binária léxica, extrair seus dados e construir um autômato com o casamento das informações são passos essenciais.

Contudo, há espaço para melhorias – se todo estudante pudesse ter acesso a um mecanismo facilitador para este tipo de estudo, certamente o processo de aprendizado ocorreria de maneira mais suave. Não obstante a isto, o processo laboral de ensino também seria menos difícil — professoras e professores da área de Computação poderiam apresentar o desenvolvimento de exercícios de tópicos relacionados a [Aho et al. 2013, p. 179] em sala de aula concomitantemente com a utilização de um mecanismo que tornaria todo o processo de construção do algoritmo menos propenso a erros. Este trabalho se desenvolve, portanto, sobre a criação de `Rosetta`, software responsável por executar todos os passos do algoritmo mencionado, de maneira visualmente palatável aos discentes e docentes.

A peça de software `Rosetta` é uma aplicação desenvolvida na linguagem de programação de propósito geral `Python` que executa em um servidor web por meio de uma interface `Dash`. É acessível tanto por meio de execução em máquina própria pela desenvolvedora ou pelo desenvolvedor interessados no projeto, vide seção 5.1, quanto por meio de seu próprio domínio de *webservice*, explícito na seção 5.2.

2. Objetivos

2.1. Objetivo Geral

Deseja-se criar um software para auxiliar no processo didático dos discentes e docentes que quiserem aprender a realizar, passo a passo, conversões de expressões regulares em autômatos finitos determinísticos. Quer-se, então, facilitar o processo de aprendizado do citado algoritmo tendo em vista que, manualmente, apresenta-se como um processo sujeito a erros.

2.2. Objetivos Específicos

Como objetivos específicos deste trabalho, quer-se:

- Criar um software que receba expressões regulares por *text inputs* e, destes, gere:
  - Uma árvore binária de análise léxica;
  - Uma tabela para um conjunto *followpos*;
  - Uma tabela representativa de um autômato finito;
  - Um diagrama de estados de um autômato finito.
- Providenciar um serviço acessível àqueles que desejarem aprender, ensinar por meio de, ou, até, realizar melhorias no software `Rosetta`.

3. Trabalhos correlatos

3.1. JFLAP

Criado pelo *Department of Computer Science* da *Duke University*, *Java Formal Language and Automata Package* (JFLAP) é uma aplicação desktop escrita em Java Swing para descrição do comportamento de Autômatos Finitos, Gramáticas Regulares, Expressões Regulares, Autômatos de Pilha, Gramáticas Livre de Contexto e Linguagens Recursivamente Enumeráveis [Rodger 2018] e também principal inspiração para este projeto. JFLAP detém sua própria interface gráfica interativa e, portanto, é um intenso facilitador no processo de ensino e aprendizagem de disciplinas que dialogam com o ramo de Linguagens Formais.

Contudo, JFLAP não provê pleno suporte para todos os algoritmos e procedimentos a serem estudados nas disciplinas de Linguagens Formais e Compiladores. No contexto desta disciplina, docentes não ensinam o Algoritmo de Thompson [Aho et al. 2013, p. 159, seção 3.7.4], o qual é o utilizado pelo software JFLAP para converter expressões regulares em autômatos finitos não-determinísticos; estes são passíveis de serem determinizados posteriormente. O trabalho atual, não obstante, pauta-se em dissecar a conversão de expressões regulares em autômatos finitos determinísticos, sem que surjam aqueles que sejam não-determinísticos como uma etapa intermediária.

3.2. Automata Tutor v3

O software `Automata Tutor v3` descrito em `Scala` e `C#` foi desenvolvido pela *Technische Universität München* e vai além de funcionalidades previamente descritas pelo JFLAP. Permite criar máquinas abstratas como autômatos finitos (determinísticos ou não), autômatos de pilha e máquinas de Turing em um *canvas*, mas, também, tem suporte à geração automatizada de atividades relacionadas às disciplinas de Teoria da Computação e Linguagens Formais e Compiladores, desde encontrar palavras reconhecíveis em uma expressão regular até jogos com o Lema do Bombeamento [D'Antoni et al. 2020, p. 7, seção 3.2]. Esta versão de *Automata Tutor* também provê avaliação automática e *feedback* de exercícios selecionados por professores de uma cadeira a serem resolvidos pelos estudantes. Este software já foi apropriadamente testado por mais de 900 estudantes [D'Antoni et al. 2020, p. 3].

Não obstante ao que fora dito, `Automata Tutor v3` não implementa o processo de conversão de expressões regulares em autômatos finitos determinísticos, processo descrito de maneira extensiva neste trabalho, com suas partes dissecadas ao usar o software proposto `Rosetta`.

3.3. Software `qtype_flwarrior`

O plugin do Moodle `qtype_flwarrior` [Albornoz 2022] desenvolvido em PHP pelo estudantes Enzo Coelho Albornoz tem como objetivo fazer a avaliação automática de exercícios relacionados a autômatos finitos, autômatos de pilha e máquinas de Turing a serem propostos por docentes em disciplinas relacionadas ao domínio de Linguagens Formais e Compiladores. Este trabalho foi diretamente inspirado por JFLAP, descrito na seção 3.1. Este software, todavia, também não analisa o processo de conversão de expressões regulares em autômatos finitos determinísticos.

4. Estrutura do software

O software `Rosetta` é composto de 4 principais módulos: 3 diretórios dentro do diretório de códigos-fonte `src/` e um programa que age como *entry point* da aplicação, chamado de `app.py`. A descrição do desenvolvimento é pautada por dissecar os submódulos de `src/` – arquivos com prefixo `.py` ou o diretório que os contempla –, dos mais triviais e menos dependentes de demais estruturas de código às estruturas mais complexas e que fazem uso daquelas que foram anteriormente citadas – uma abordagem *bottom-up*.

A estrutura do código obedece o diagrama de classes visível na Figura 1, com os módulos mais importantes salientados em laranja mais escuro que os demais.

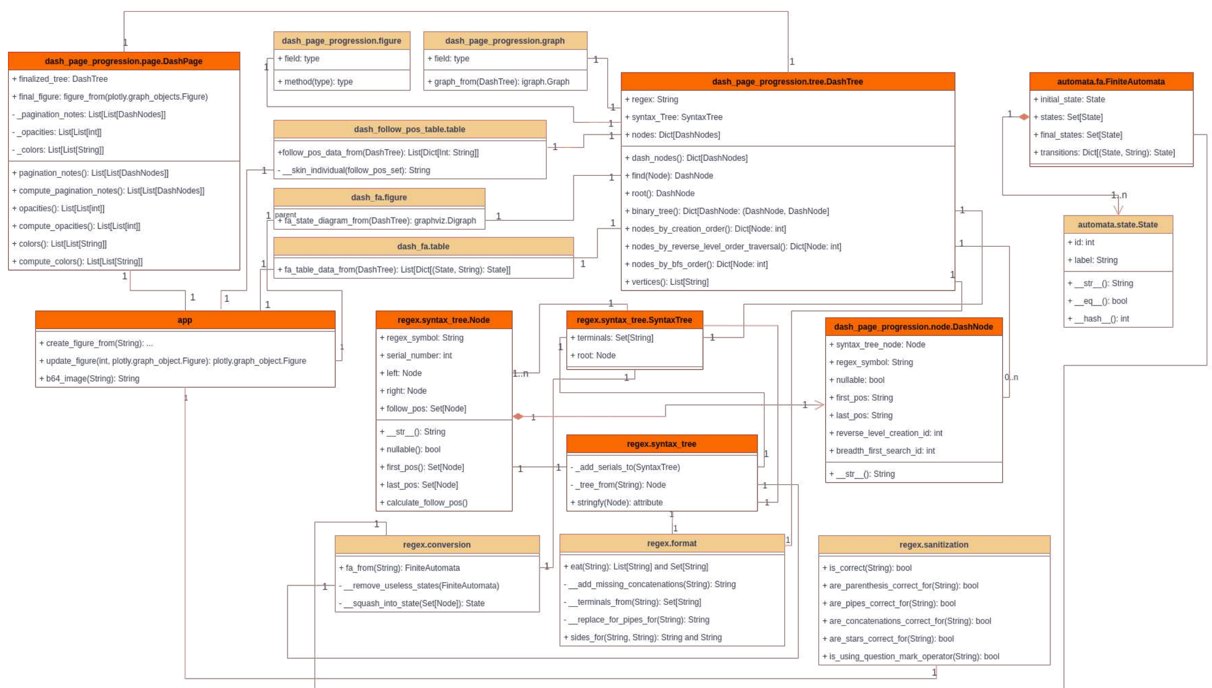

Figura 1. Diagrama de classes do software proposto, com as classes mais importantes em laranja mais saturado.

5. Guia de uso

O guia de uso pode ser dividido àqueles que desejam adicionar novas *features* e àqueles que apenas almejam utilizar o *webservice* em decorrência da didática e do aprendizado. O código pode ser encontrado em sua integralidade na `main branch` deste repositório: https://github.com/vdella/rosetta.

5.1. Para *developers*

Caso haja ambição de prover alterações ao projeto, deve-se, em primeira mão, instalar uma versão maior ou igual à 3.10 da linguagem de programação `Python`. Assumindo utilização de um sistema operacional Linux Ubuntu 22.04 a interface de desenvolvimento PyCharm [pyc], pode- se obter `Python3.10` por meio dos seguintes passos.

```
sudo apt update && sudo apt upgrade -y
sudo add-apt-repository ppa:deadsnakes/ppa
sudo apt install python3.10
```

Com a instalação concluída, pode-se usar o próprio *package installer* da versão baixada para obter acesso a 3 bibliotecas: `Plotly`, `Dash` e **graphviz**.

```
pip install plotly==5.18.0
pip install dash
pip install graphviz
```

Com as *libraries* devidamente instaladas, pode-se manusear o código-fonte do repositório mencionado acima e abrir as portas para a realização de quaisquer alterações. Para a execução de `Rosetta` basta executar `App.py` por meio da interface de desenvolvimento do PyCharm. Com a aplicação em execução, colocando-se no papel de discente ou docente, refira-se à próxima seção para uso do software.

5.2. Para alunos e professores

Como usuário discente ou docente, com acesso ao serviço, encontrar-se-ia a página principal que fica em aguardo pela digitação de uma expressão regular.

Figura 2. Aplicação com *regex* a ser interpretada

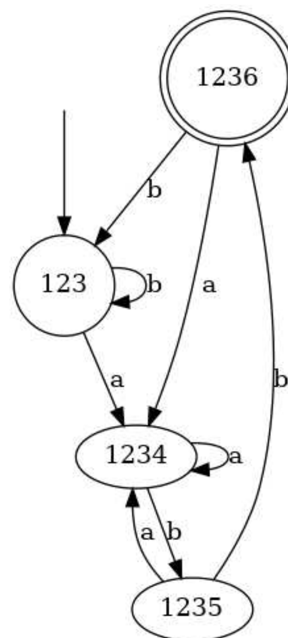

Figura 3. Autômato gerado a partir de `(a|b)*abb#`.

Expressões regulares terão sempre caracteres unários, *e.g.* a expressão regular *cat* será interpretada sempre como *c.a.t.#*. Isto ocorre para poupar o usuário de ter que colocar o alfabeto de uma dada linguagem, visto que seria necessário reconhecer *cat* como uma palavra completa ao invés das concatenações das letras "c", "a" e "t". Além disso, diferentemente das *regexes* explicitadas em [Aho et al. 2013, p. 173, seção 3.9.1], não faz-se necessário adicionar a cerquilha ao fim daquilo que for digitado: por padrão, isto já é feito em conjunto com o englobamento da entrada do usuário por parênteses. Ou seja, toda entrada *abc* será interpretada como *(abc)#*. São tratados como operadores para todas

e quaisquer regexes o *pipe* como operador *or* (|), o ponto final como concatenação e o asterisco como fecho de Kleene.

Para alunos e professores, basta fazer acesso ao *webservice* e digitar uma *regex* a fim de observar os resultados. Pode-se acessar `Rosetta` em https://rosetta-b4b3447c02da.herokuapp.com/.

6. Considerações Finais

Neste trabalho providenciou-se o desenvolvimento de um *webservice* que tem como objetivo facilitar o entendimento e aprendizado dos estudantes quanto ao algoritmo de conversão de expressões regulares em autômatos finitos determinísticos [Aho et al. 2013, p. 179]. Embora possa-se converter expressões regulares em autômatos finitos não-determinísticos e, então, realizar sua determinização, o algoritmo descrito em software propicia um resultado que não exige a realização de uma segunda operação.

Como adições futuras, poder-se-ia sugerir *features* a serem adicionadas em `Rosetta`, como a geração de uma outra figura de uma árvore preenchida, em que apenas os nodos-folha referentes a um dado *followpos* escolhido seriam coloridos. Poder-se-ia alterar o *followpos* da vez por outro elemento de paginação a ser colocado abaixo da árvore. Este processo agiria como etapa intermediaria da tabela *followpos* e da geração do autômato finito. Não obstante, funcionalidade similar poderia ser criada sobre a construção do autômato finito determinístico final, em que poder-se-ia, também, criar um outro elemento de paginação em que, a cada página da paginação, criar-se-ia uma nova transição até que pudesse-se chegar ao autômato resultado final completo; isto foi desconsiderado neste trabalho no momento atual visto que o AF final é gerado em formato `.png` e não é um componente `Plotly` como a árvore de análise léxica.

Por fim, espera-se tornar o processo de aprendizado e de ensino das disciplinas de Linguagens Formais e Compiladores menos árduo, de modo a facilitar a compreensão por parte dos estudantes e auxiliar o docente na apresentação deste tópico.

Referências

Pycharm. `https://www.jetbrains.com/pt-br/pycharm/`. Accessed: 2023-11-12.

Aho, A. V., Lam, M. S., Sethi, R., and Ullman, J. D. (2013). *Compilers: Pearson new international edition*. Pearson custom library. Pearson Education, London, England, 2 edition.

Albornoz, E. C. (2022). Avaliação automática de questões sobre autômatos na plataforma moodle.

D'Antoni, L., Helfrich, M., Kretinsky, J., Ramneantu, E., and Weininger, M. (2020). *Automata Tutor v3*, page 3–14. Springer International Publishing.

Rodger, S. H. (2018). Java Formal Language and Automata Package (JFLAP). `https://www.jflap.org/`.