UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ

FELLIPE DE SOUZA REIS

DESENVOLVIMENTO DE UM INTERPRETADOR DE EXPRESSÕES DE TEORIA DOS CONJUNTOS PARA FINS DIDÁTICOS

TOLEDO

2023

FELLIPE DE SOUZA REIS

DESENVOLVIMENTO DE UM INTERPRETADOR DE EXPRESSÕES DE TEORIA DOS CONJUNTOS PARA FINS DIDÁTICOS

DEVELOPMENT OF A SET THEORY EXPRESSION INTERPRETATOR FOR DIDACTIC PURPOSES

Trabalho de Conclusão de Curso de Graduação apresentado como requisito para obtenção do título de Bacharel em Engenharia de Computação do Curso de Bacharelado em Engenharia de Computação da Universidade Tecnológica Federal do Paraná.

Orientador: Gustavo Henrique Paetzold

TOLEDO

2023

FELLIPE DE SOUZA REIS

DESENVOLVIMENTO DE UM INTERPRETADOR DE EXPRESSÕES DE TEORIA DOS CONJUNTOS PARA FINS DIDÁTICOS

Trabalho de Conclusão de Curso de Graduação apresentado como requisito para obtenção do título de Bacharel em Engenharia de Computação do Curso de Bacharelado em Engenharia de Computação da Universidade Tecnológica Federal do Paraná.

Data de aprovação: 19/junho/2023

Prof. Dr. Gustavo Henrique Paetzold
Universidade Tecnológica Federal do Paraná

Prof. Dr. Ricardo Tavares de Oliveira
Universidade Tecnológica Federal do Paraná

Prof. Dr. Maurício Zardo Oliveira
Universidade Tecnológica Federal do Paraná

Prof. Dr. Daniel Cavalcanti Jeronymo
Universidade Tecnológica Federal do Paraná

TOLEDO
2023

Dedico este trabalho à meus pais e ao sonho
deles de ter um filho engenheiro.

AGRADECIMENTOS

Agradeço ao meu orientador Prof. Dr. Gustavo Henrique Paetzold, pela extrema paciência, orientação e conhecimentos passados para a realização deste trabalho.

Aos demais professores, por todos os conhecimentos que me passaram ao longo da graduação.

Aos meus antigos colegas de curso, pelo apoio mútuo e horas de estudo em conjunto.

Aos meus amigos mais próximos, pelo apoio em vários momentos durante a graduação, pois provavelmente sem eles eu teria desistido muito antes.

Agradeço também à meus pais, pois sem o apoio dos mesmos é certo que eu não teria chego até aqui.

RESUMO

O ensino de engenharia na área de Matemática Computacional é fundamental para a formação do egresso e abrange diversos conteúdos, tais como: matemática discreta, lógica, entre outros. No entanto, há escassez de software para auxílio no aprendizado dos alunos dos cursos superiores da área de computação, o que motiva a implementação de ferramentas do gênero. No âmbito tanto da computação como da matemática, a teoria dos conjuntos se demonstra vital para fundamentar as bases de ambas, sendo o conhecimento dessa teoria um requisito básico para a compreensão da teoria da computação. Entre os diversos tipos de software que podem auxiliar no aprendizado, incluem-se os interpretadores, cujo uso pode ser tanto para auxiliar na resolução de exercícios, quanto na criação de exemplos para o entendimento. Interpretadores são programas que executam instruções escritas em uma determinada linguagem. Neste trabalho, foi abordada a construção e o projeto de um interpretador completo de expressões de teoria dos conjuntos para fins didáticos com interface interativa. Para isso, foi utilizada a literatura básica de compiladores em conjunto com a linguagem Python. Como resultado final do trabalho, foi construído o interpretador completo com analisador lexical, sintático e semântico. Experimentos de desempenho e corretude apresentados evidenciam sua eficácia e eficiência.

Palavras-chave: interpretador; expressões; conjuntos; didático; python.

ABSTRACT

The teaching of engineering in the field of Computational Mathematics is fundamental for the training of the graduate and encompasses various subjects, such as discrete mathematics, logic, among others. However, there is a scarcity of software to aid in the learning of students in higher education computer science courses, which motivates the implementation of such tools. In both the field of computer science and mathematics, set theory proves to be vital in establishing the foundations of both disciplines, and knowledge of this theory is a basic requirement for understanding the Computer Theory. Among the various types of software that can assist in learning are interpreters, which can be used both for assisting in the resolution of practice exercises and creating examples for comprehension. Interpreters are programs that execute instructions written in a specific language. This work addressed the construction and design of a complete interpreter for set theory expressions for didactic purposes with an interactive interface. The basic literature of compilers, in conjunction with the Python language, was used for this purpose. As a final result of the work, a complete interpreter with lexical, syntactic, and semantic analyzers was constructed. Performance and correctness experiments presented demonstrate its effectiveness and efficiency.

Keywords: interpreter; expression; set; didatic; python.

LISTA DE FIGURAS

LISTA DE GRÁFICOS

LISTA DE QUADROS

LISTAGEM DE CÓDIGOS FONTE

SUMÁRIO

1 INTRODUÇÃO

Nos cursos da área de computação, como ciência da computação, engenharia de computação, engenharia de software e outros, é evidente a presença constante de disciplinas relacionadas à matemática. Um exemplo que ilustra essa relação é o documento oficial intitulado "Parecer CNE/CES nº 136/2012, aprovado em 8 de março de 2012 - Diretrizes Curriculares Nacionais para os cursos de graduação em Computação" (Ministério da Educação - Brasil, 2012). Nesse documento, que lista os "Conteúdos Curriculares da Formação Tecnológica e Básica" para os cursos da área de computação, apenas 8 das 50 matérias não estão diretamente relacionadas com a disciplina matemática.

Um dos tópicos que surgem com frequência considerável em ementas de disciplinas básicas como matemática discreta e em disciplinas intermediárias é a “teoria dos conjuntos”, dado o notável papel que a mesma tem em firmar as bases da matemática e, por consequência, da computação.

Entretanto, mesmo quando se trata de ferramentas didáticas nos tópicos referentes a conjuntos, não há muitas ferramentas comuns como geradores e interpretadores de expressões que podem auxiliar nas tarefas de construção de provas, listas de exercício e *slides*. Como exemplo de interpretador semelhante já implementados existe o “Interpretador de expressões lógicas” (MOREIRA *et al*., 2001) também voltado para fins didáticos, porém o mesmo tem seu foco em expressões da lógica boolena.

Entre os diversos tipos de softwares que podem trabalhar com as expressões da teoria dos conjuntos estão os interpretadores. Interpretadores são comumente compostos por um analisador lexical, que converte uma entrada textual para uma sequência de *tokens* e verifica se os caracteres de entrada pertencem ao alfabeto da linguagem; um analisador sintático, que avalia a estrutura já tokenizada da entrada a validando e preparando para a próxima etapa; e por fim um analisador semântico, que valida o significado da expressão e gera a saída esperada. A escolha de um interpretador como o tipo de software é justificada devido à sua capacidade de oferecer versatilidade para alterações, manutenção e exibição das etapas de funcionamento, além de contar com uma quantidade abundante de artigos sobre o tema. Em uma visão mais prática, um interpretador pode ser utilizado para a construção de exercícios para listas, auxiliar no desenvolvimento de avaliações, ilustrar os processos de interpretação durante as aulas e ajudar os alunos na interpretação de expressões fora da sala de aula também.

Neste trabalho será abordada a construção de um interpretador completo de expressões de teoria dos conjuntos para fins didáticos. Além disso, estarão expressos neste trabalho os processos de implementação da ferramenta e seus devidos testes de validação e, quando cabível, desempenho. Neste trabalho, o Capítulo 2 contém os fundamentos teóricos utilizados para o desenvolvimento do interpretador, o Capítulo 3 contém as ferramentas e algoritmos utilizados para implementação do software, o Capítulo 4 contém os resultados obtidos desse trabalho e, por fim, o Capítulo 5 contém as conclusões do projeto.

Esta ferramenta está disponível em uma plataforma de versionamento que permita sua distribuição livre para toda a comunidade. O *link* encontra-se disponível no Capítulo 5.

1.1 Objetivos

Como objetivo principal existe a implementação de um interpretador de expressões de teoria dos conjuntos para fins didáticos. Para este fim serão cumpridos os seguintes objetivos específicos:

• Conduzir uma revisão bibliográfica acerca da teoria de compiladores e teoria de conjuntos;

• Implementar um módulo de análise léxica com base na estrutura léxica clássica da teoria dos conjuntos;

• Realizar a comparação entre métodos e desenvolver um analisador sintático utilizando a variante mais eficiente;

• Desenvolver um analisador semântico que seja capaz de produzir o conjunto resultante da expressão;

• Desenvolver uma interface gráfica que integre todos os analisadores de forma didática.

1.2 Justificativa

A motivação do trabalho se encontra na escassez de softwares de cunho didático para auxílio das matérias de matemática computacional dos cursos superiores de engenharia e computação. Tal fato foi observado de forma prática pelo orientador deste trabalho de conclusão de curso. Já existem ferramentas didáticas como o “Interpretador de expressões lógicas” de Moreira (MOREIRA *et al.*, 2001) e até mesmo o “Programa Interpretador e Resolvedor para Máquinas de Estado Finito” de Postal (POSTAL *et al.*, 2007), porém nenhuma delas trabalha especificamente com a teoria dos conjuntos ou apresenta suporte a funcionalidades como geração de expressões para formulação de exercícios, provas, demonstrações práticas, etc.

Assim que implementada, esta ferramenta servirá tanto para auxiliar os discentes dos cursos superiores de computação quanto os docentes, nas tarefas de correção e geração de expressões em listas de exercícios, avaliações e explicações práticas de conceitos do conteúdo, seja pelo professor ou por estudantes curiosos no assunto. Além disso, devido à natureza de um interpretador em si, o programa aqui desenvolvido poderá ser utilizado no ensino de tópicos tangentes à teoria dos conjuntos, como a teoria da computação e, por exemplo, nos processos de análise léxica, sintática e semântica da disciplina de compiladores.

2 REFERENCIAL TEÓRICO

Este trabalho tem como pilares teóricos fundamentais a teoria de conjuntos e a teoria de compiladores. Os tópicos relacionados a interface gráfica são explicados na Seção 2.4.

2.1 Teoria dos conjuntos

A teoria dos conjuntos tem sua origem formal no século XIX nos trabalhos de George Cantor sobre os diversos conjuntos numéricos. Como dito por José Ferreirós (FERREIRÓS, 2008) a teoria dos conjuntos tem papel crucial, durante os séculos XIX e XX, em fundamentar novos conceitos da álgebra, aritmética e geometria.

Devido ao aparecimento de paradoxos na teoria dos conjuntos, como o famoso “paradoxo de Russell”, construiu-se a teoria axiomática dos conjuntos que através dos axiomas da Separação, Substituição e da Fundação eliminou os diversos problemas da antiga teoria dos conjuntos e fundou os axiomas “Zermelo-Fraenkel com o Axioma da Escolha” (ZFC) que são os axiomas padrões da teoria de conjuntos moderna, como explicado em mais detalhes no artigo online "Set Theory"(BAGARIA, 2014).

É conhecimento básico que um conjunto, de forma “primitiva”, nada mais é que uma coleção de elementos distintos (podendo ser esses elementos números, palavras, outros conjuntos, entre outros). Quando um determinado elemento y pertence a um conjunto X é correto afirmar que y é elemento de X ou “$y \in X$ ”. Outro conceito bastante importante é o de subconjunto. De forma informal, se diz que um conjunto X é subconjunto de um conjunto Y quando todos os elementos de um conjunto X também estão em Y, então é correto afirmar que “$X \subseteq Y$”.

Entre as operações básicas binárias usadas na teoria de conjuntos, e que serão tratadas neste trabalho, existem a união que gera o conjunto que possui os elementos dos dois conjuntos operandos (ilustrado na Equação (1)), a intersecção que resulta no conjunto que possui todos os elementos comuns aos operandos (exemplificado na Equação (2)), a diferença que retorna o conjunto que possui todos os elementos do operando da esquerda e que não sejam elementos do operando direita (como mostrado na Equação (3)) e o produto cartesiano que gera o conjunto com todas os pares ordenados de todas as combinações entre os elementos dos operandos (como no exemplo da Equação (4)).

$$\{3,4,5\} \cup \{4,5,6,7\} = \{3,4,5,6,7\} \tag{1}$$

$$\{3,4,5\} \cap \{4,5,6,7\} = \{4,5\} \tag{2}$$

$$\{3,4,5\} - \{4,5,6,7\} = \{3\} \tag{3}$$

$$\{3,4,5\} \times \{5,6\} = \{(3,5),(3,6),(4,5),(4,6),(5,5),(5,6)\} \tag{4}$$

Quando se trata de operações binárias, é interessante ressaltar a propriedade da comutatividade que, de conhecimento básico, é a propriedade que certas operações tem em, alterando-se a ordem dos operandos, o resultado se manter inalterado. Porém especificamente das operações acima as únicas que são comutativas é a união e a intersecção. No geral a estrutura sintática das operações binárias de conjuntos, no âmbito deste trabalho, pode ser resumida em “<operando a esquerda> <operador> <operando a direita>”.

Já nas operações unárias existe a operação de complemento que resulta em um conjunto de todos os elementos do universo que não sejam elementos do operando (mostrado na Equação (5) e considerando que o conjunto universo é os naturais) e a operação de conjunto das partes que retorna o conjunto de todos os subconjuntos do conjunto operando (exemplificado na Equação (6)). Todos os operadores unários abordados neste trabalho são prefixados e tem sua estrutura sintática da forma “<operador> <operando>”.

$$\sim \{3,4,5\} = \{1,2,6,7,8,9,10,11,12\ldots\} \tag{5}$$

$$P(\{3,4,5\}) = \{\{\},\{3\},\{4\},\{5\},\{3,4\},\{3,5\},\{4,5\},\{3,4,5\}\} \tag{6}$$

No Quadro 1 apresenta-se formalmente a gramática das expressões de teoria dos conjuntos utilizada no começo da primeira metade do projeto. Em resumo, S é a variável inicial da gramática que gera a sequência "identificador = A", onde "identificador" e "=" são terminais fixos que não podem ser derivados em outras variáveis ou terminais. Por outro lado, A é uma variável que pode ser derivada em uma variedade de elementos, incluindo terminais, outras variáveis e até mesmo sequências que envolvem ambos. Como exemplo de derivação, existe a Equação (7), onde a partir do símbolo inicial S, chega-se a uma expressão do tipo "identificador = identificador $\cup$ conjunto". Os parênteses à direita representam as operações realizadas em uma determinada variável, enquanto à esquerda está a expressão derivada. A expressão final pode representar várias expressões, como por exemplo "$B = C \cup \{\}$" ou até mesmo "$A0 = A1 \cup \{1,2,abc\}$". As variáveis derivadas em cada linha estão ressaltadas em vermelho e negrito.

$$\begin{aligned}
&\boldsymbol{S} && (S \rightarrow identificador = A)\\
&identificador = \boldsymbol{A} && (A \rightarrow ABA)\\
&identificador = A\boldsymbol{B}A && (B \rightarrow \cup)\\
&identificador = \boldsymbol{A} \cup A && (A \rightarrow identificador)\\
&identificador = identificador \cup \boldsymbol{A} && (A \rightarrow conjunto)\\
&identificador = identificador \cup conjunto
\end{aligned} \tag{7}$$

Quadro 1 – Gramática das expressões de teoria dos conjuntos do projeto.

Derivações
$S \rightarrow identificador = A$
$A \rightarrow (A) \mid ABA \mid CA \mid identificador \mid conjunto$
$B \rightarrow \cup \mid \cap \mid - \mid \times$
$C \rightarrow p \mid \sim$

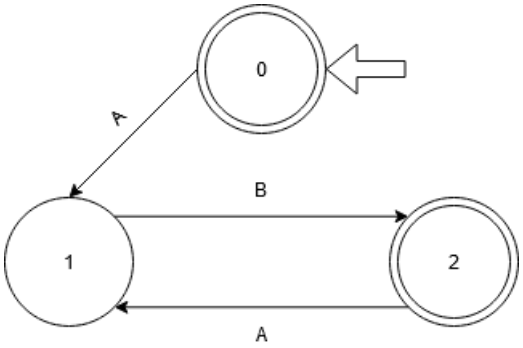
Fonte: Autoria própria (2023).

2.2 Teoria dos autômatos

A teoria de autômatos é intimamente relacionada ao estudo das linguagens formais e construção de compiladores. O tipo mais simples de autômato é o autômato finito determinístico (AFD) que analisa cadeias de caracteres, validando as mesmas conforme as regras de uma linguagem regular. De forma concisa, uma expressão regular é uma forma compacta de descrever uma linguagem regular utilizando as operações de concatenação, união e fecho de Kleene. Essa abordagem é amplamente estudada e aprofundada na teoria da computação. É importante notar que autômatos finitos determinísticos não trabalham com entradas que dependam de recursos como balanceamento de parênteses, porém os mesmos são adequados para identificação de operadores, variáveis e palavras reservadas das diversas linguagens de programação. Consequentemente, uma das principais aplicações deste tipo de autômato é na construção de analisadores léxicos como será visto no Capítulo 3.

Como mostrado na Figura 1, os autômatos são representados por máquinas de estados onde a seta maior mostra o estado inicial do autômato. Os estados com um círculo concêntrico em seu interior são os estados finais que validam a entrada e as setas entre estados mostram qual caractere de entrada deve ser lido para que a transição seja realizada. A rigor um AFD é definido pela quíntupla $(Q, \Sigma, \delta, q_0, F)$ sendo Q o conjunto dos estados do autômato, Σ o alfabeto da linguagem processada pelo autômato, q_0 o estado inicial, F o conjunto dos estados finais e δ o conjunto das transições do autômato.

Figura 1 – Autômato que valida as expressões regulares na forma [AB]*.



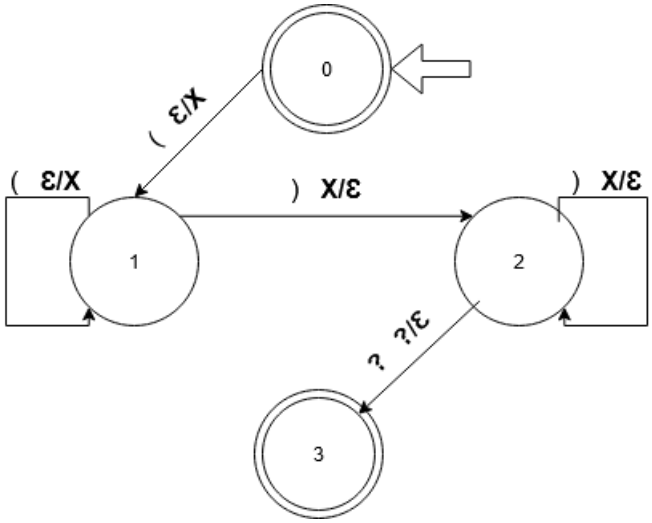
Fonte: Autoria própria (2023).

Para lidar com questões como o balanceamento de parênteses e trabalhar com gramática livres de contexto, existe o autômato com pilha. O autômato com pilha, como mostrado na Figura 2, nada mais é que o autômato finito determinístico com o uso de uma pilha auxiliar para gravar certas informações dos estados anteriores. Ao contrário dos AFDs que sua transição é definida apenas como “caractere de entrada lido”, os autômatos com pilha têm suas transições definidas no formato “caractere de entrada lido, elemento retirado do topo da pilha/elemento inserido no topo da pilha”.

A rigor um AP é definido pela sêxtupla $(Q, \Sigma, \Gamma, \Delta, q0, F)$ sendo Q o conjunto dos estados do autômato, Σ o alfabeto da linguagem processada pelo autômato, Γ o alfabeto utilizado pela pilha, q0 o estado inicial, F o conjunto dos estados finais e Δ o conjunto das transições do autômato. O caractere “?” quando na posição de leitura da entrada verifica se a entrada já chegou ao fim e na posição de consumo da pilha confere se a mesma se encontra vazia. Como mostrado no capítulo 3, os autômatos com pilha podem ser utilizados na análise sintática para validar a estrutura das entradas do interpretador.

De forma resumida, uma tupla é uma sequência ordenada de elementos. Por exemplo, (a, b, c) é diferente de (c, b, a). Uma quintupla é uma tupla com cinco elementos, enquanto uma sextupla é uma tupla com seis elementos.

Figura 2 – Autômato que valida as expressões regulares na forma $[(]^n[)]^n$ com $n \geq 0$.



Fonte: Autoria própria (2023).

2.3 Teoria de compiladores

No princípio, os primeiros programas de computador eram implementados diretamente na linguagem Assembly. Após algum tempo o primeiro compilador seria desenvolvido por Grace Hopper para a linguagem A-0 o que iniciaria os estudos nesta área. Nas palavras da própria Hopper, registradas no trabalho de Wexelblat (WEXELBLAT, 1981), a linguagem A-0 tratava-se

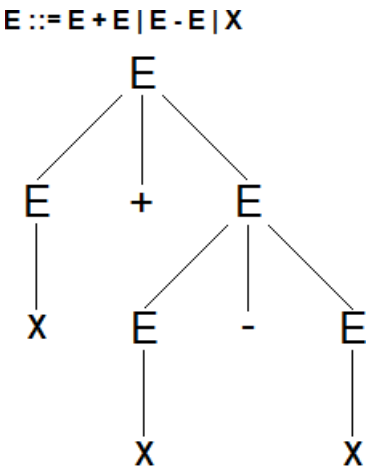
de uma série de especificações, ou seja, para cada sub-rotina do programa tinha uma palavra de chamada que a chamava da biblioteca para o programa principal do computador UNIVAC I.

De forma simplificada, existem entre os tipos de linguagens de programação as compiladas e as interpretadas. Como explicado por Farias (FARIAS; MEDEIROS, 2013) uma linguagem é compilada quando a mesma utiliza de um compilador para converter código de alto nível para linguagem de máquina ou de montagem. Por outro lado, uma linguagem interpretada utiliza de um interpretador para executar instruções escritas em uma determinada linguagem de programação. Geralmente, um programa de uma linguagem compilada só deve ser novamente compilado quando há modificações no código fonte da aplicação. Já uma linguagem interpretada terá que fazer o processo de interpretação sempre quando o programa for executado. É importante notar que, mesmo comumente tendo uma estrutura mais simples e sendo mais fáceis de desenvolver, os interpretadores ainda têm a necessidade de passar pelas etapas de análise e validação de entradas que é dividida em três.

Primeiramente é realizada análise léxica, que possui a função de identificar os símbolos básicos da linguagem (*tokens*) a partir dos caracteres de entrada e verifica se todos os caracteres pertencem ao alfabeto da linguagem definida. Um lexema é uma sequência de caracteres, todos pertencentes ao alfabeto da linguagem trabalhada, que pertencem a um certo grupo de lexemas denominado *token*. Por exemplo, “0”, “1001” e “255” são lexemas que geram o *token* “INTEIRO”.

Em seguida é necessária a análise sintática que tem como objetivo reconhecer a estrutura das sequências de *tokens*, gerados na análise léxica, validando e organizando para a próxima etapa. Para validar a sequência de *tokens*, o analisador sintático realiza as diversas derivações da linguagem gerando a “árvore de derivação sintática” (exemplificada na Figura 3). Também nesta etapa são detectados os erros de sintaxe e tratados quando existe um “Tratador de Erros” aliado ao analisador sintático.

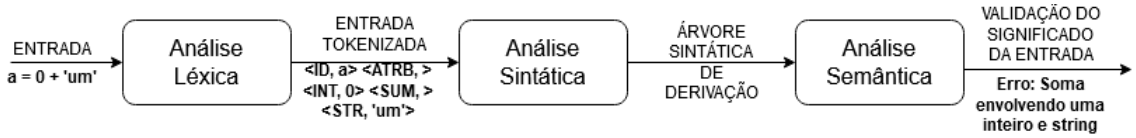
Figura 3 – Exemplo de árvore sintática de derivação, gramática da árvore no canto superior da figura.



Fonte: Autoria própria (2023).

Por fim a análise semântica processa a “árvore de derivação sintática” verificando a coerência e significado da entrada validada pelas etapas anteriores gerando a saída desejada. A análise semântica tem como parte de seus objetivos detectar erros como referência a variáveis não declaradas, atribuição de um determinado tipo a uma variável de um tipo totalmente distinto e operações entre tipos diferentes como exemplificado na Figura 4.

Figura 4 – Etapas de compilação encadeadas.



Fonte: Autoria própria (2023).

2.4 Visualização e animação de algoritmos

Trata-se da área que estuda a implementação e eficiência da apresentação visual e animação de diversos algoritmos, expondo como os mesmos manipulam as estruturas de dados e geram suas saídas. Como escrito no artigo de revisão "Algorithm Visualization: The State of the Field"(SHAFFER *et al.*, 2010), essa área atrai principalmente a atenção e esforços de professores de graduação que veem o potencial educacional da mesma.

Diversos trabalhos da área abordam tópicos diferentes de estudo da visualização de algoritmos, mas como exposto no trabalho “Effective Features of Algorithm Visualizations” (SARAIYA *et al.*, 2004), pode-se tomar como tópicos chave a “facilidade de uso”, “feedback apropriado”, “mudanças de estado”, “gerenciamento de janelas”, “múltiplas visões”, “controle do usuário”, “pseudocódigo” e “exemplos e inserção de dados".

Ainda seguindo a ideia base dos autores citados no parágrafo anterior, a “facilidade de uso” pode ser resumida como o quão intuitiva a interface é e quanto esforço é necessário para o usuário da aplicação aprender a utilizar a mesma. Também se incluem neste tópico os bugs operacionais que tornem impossível o uso do software. O conceito “feedback apropriado” trata do fato de a aplicação tomar como certo o nível de conhecimento que o usuário tem em determinados tópicos que são pré-requisitos do algoritmo apresentado. Assim, este conceito trata de como a aplicação deixa explicito o nível de conhecimento que ela espera do usuário através de mensagens de avisos ou outros recursos.

"Mudanças de estado" se refere a como a interface exibe a interação entre o algoritmo e as estruturas de dados, podendo até mesmo utilizar informações textuais para explicar o estado em que o algoritmo se encontra. Já o "gerenciamento de janelas" trata de quantas janelas a aplicação utiliza e se elas são apresentadas de maneira "confortável" para o usuário. Também é nesse tópico que se avalia se é possível reposicionar e redimensionar as janelas do software. A ideia de "múltiplas visões" baseia-se no conceito de exibir a visão lógica do algoritmo através de diagramas que representam as estruturas de dados em sua forma gráfica, como pilhas e

árvores, e também a visão física, que se baseia na representação da memória do sistema e dos dados sendo manipulados nela. Muito relacionado ao tópico anterior, está a presença ou não de "pseudocódigo" na aplicação, o que permite ao usuário ter uma visão de qual parte do algoritmo está sendo executada em um determinado momento.

"Controle do usuário" trata-se da opção ou não de o usuário controlar a execução do algoritmo, definindo a velocidade de execução, controlando os passos do algoritmo, podendo avançar para o próximo estado ou retornar ao anterior e repetir toda a execução do algoritmo se necessário. Por fim, "exemplos e inserção de dados" é a ideia de se o usuário pode ou não inserir os próprios dados de teste no algoritmo e se o algoritmo possui bons exemplos de teste para demonstrar o comportamento do sistema. Como mostrado pelos autores desses tópicos chave, a presença desses dois últimos tópicos são os que mais agregam valor pedagógico ao sistema (SARAIYA *et al.*, 2004).

Além dos diversos elementos chave que a interface de um algoritmo didático pode ter, a área de visualização de algoritmos também se preocupa com a interação do software com os alunos e o uso que o docente faz em sala do sistema. Em seu trabalho, Naps, T. L. et al. (NAPS *et al.*, 2003) criaram o seguinte sistema de classificação para definir o envolvimento do aluno com a tecnologia de visualização:

• Sem visualizar: trata-se do caso onde não é utilizado nenhum “software de visualização” durante o ensino.

• Visualizando: é a forma principal de envolvimento entre o aluno e o “software de visualização” e está presente nas demais formas abaixo também, pois neste caso o aluno apenas vê passivamente a animação do algoritmo executando a sua frente.

• Respondendo: esta forma apoia-se em o aluno ver a execução do algoritmo, porém durante a execução ser abordado por perguntas como “Qual o próximo passo do algoritmo?” ou ainda “Qual seria o melhor e o pior caso do algoritmo?”

• Mudando: a principal ideia desta forma é o professor e o estudante mudar o conjunto dos dados de entrada e visualizar a animação do algoritmo e seus resultados. Nesta forma também pode ser usados os métodos do tópico anterior.

• Construindo: os alunos constroem seu próprio “software de visualização” entendendo como o algoritmo funciona e mapeando os requisitos para construção do mesmo.

• Apresentando: neste ultimo tópico os alunos apresentam e discutem o “software de visualização” explicando o seu uso e funcionamento do algoritmo que esta sendo exibido. O software apresentado pode ou não ter sido implementado pelos alunos.

Por fim, trabalhos como o de Törley, G. (TöRLEY, 2014) mostram que o ensino acompanhado de algoritmos de visualização gera resultados positivos e queda nas taxas de reprovação das matérias dos cursos de computação, por isso parte dos fundamentos teóricos deste trabalho são esta área e sua preocupação com os recursos apresentadas na interface do sistema final implementado.

3 MATERIAIS E MÉTODOS

Esta seção do trabalho é dividida em duas. A primeira parte, denominada “Interpretador”, trata dos resultados já apresentados durante o pré-projeto e a segunda parte discorre sobre a interface gráfica do software desenvolvida durante a última etapa do trabalho.

3.1 Interpretador

Para implementação da primeira versão da ferramenta foi escolhido o uso da linguagem Python (em sua versão 3.7) por ser uma linguagem de prototipagem ágil, possuir uma grande comunidade de usuários ativos e ser usada no meio profissional, como dito por Van Rossum (ROSSUM *et al.*, 1999). A priori, cogitou-se o uso da biblioteca PLY para o desenvolvimento do interpretador de expressões de conjuntos. A biblioteca PLY nada mais é que uma implementação das ferramentas de *parsing lex* (que realiza análise lexical) e Yacc (que realiza análise sintática) voltada ao Python (BEAZLEY, 2020), para construção das análises léxica e sintática. Entretanto a Yacc demonstrou diversos erros de compilação relacionados a versão em que a PLY se encontrava e a versão do Python utilizada. Por fim optou-se pela criação de implementações autorais de analisadores léxicos e sintáticos.

3.1.1 Operações de conjuntos e implementação das mesmas

Para uma melhor compreensão do texto, esta subseção falará sobre a implementação das operações de conjuntos em Python utilizadas neste projeto. Entretanto reforça-se que elas foram implementadas durante a construção da análise semântica. Entender essas operações e sua implementação nesta parte do texto é vantajoso para um melhor entendimento das partes que se seguirão.

A operação de união entre conjuntos pode ser definida como $A \cup B = \{x : x \in A \vee x \in B\}$, onde A e B são conjuntos (HAMMACK, 2020). Em termos mais simples, a união de conjuntos contém todos os elementos presentes tanto no conjunto A quanto no conjunto B como mostrado na Equação (1) do capítulo anterior. Para implementar essa operação em código, utilizou-se o método *union* da própria classe *Set* do Python. Antes disso, uma verificação foi realizada para garantir que nenhum dos operandos fosse uma variável nula, conforme apresentado na Listagem 1.

De forma breve, a operação de interseção entre conjuntos pode ser definida como a coleção de elementos que são comuns tanto a A quanto a B, ou seja, $A \cap B = \{x : x \in A \wedge x \in B\}$ onde A e B são conjuntos (HAMMACK, 2020). Em termos mais simples, a interseção seleciona apenas os elementos que são compartilhados por ambos os conjuntos A e B, como exemplificado na Equação (2) do capítulo anterior. Novamente, antes de executar a operação, é

Listagem 1 – Operação de união entre conjuntos implementada em Python

```
1 def uniao(self, conjA, conjB):
2     if conjA == None or conjB == None:
3         return None
4     saida = conjA.union(conjB)
5     return saida
```

Fonte: Autoria própria (2023).

verificado se algum dos operandos é nulo e caso contrário, utiliza-se o método *intersection* da própria classe *Set* como mostrado na Listagem 2.

Listagem 2 – Operação de intersecção entre conjuntos implementada em Python

```
1 def interseccao(self, conjA, conjB):
2     if conjA == None or conjB == None:
3         return None
4     saida = conjA.intersection(conjB)
5     return saida
```

Fonte: Autoria própria (2023).

Definida como $A - B = \{x : x \in A \land x \notin B\}$, onde A e B são conjuntos (HAMMACK, 2020), a operação de diferença pode ser resumida como "o conjunto que contém os elementos de A que não estão presentes em B" (ver exemplo na Equação (3) do capítulo anterior). Ao contrário das operações anteriores, nesta operação de diferença há validações adicionais. Além da validação de variáveis nulas, também são tratados os casos em que os conjuntos A ou B são o conjunto vazio, retornando as respostas esperadas para esses casos específicos. Caso contrário, o fluxo do algoritmo continua utilizando a função *difference* do tipo *Set*, conforme demonstrado na Listagem 3.

Listagem 3 – Operação de diferença entre conjuntos implementada em Python

```
1 def diferenca(self, conjA, conjB):
2     if conjA == None or conjB == None:
3         return None
4     if conjB == set():
5         return conjA
6     elif conjA == set():
7         return set()
8     else:
9         return conjA.difference(conjB)
```

Fonte: Autoria própria (2023).

O conjunto dos pares ordenados gerado pela operação de produto cartesiano é definido como $A \times B = \{(a, b) : a \in A, b \in B\}$, sendo A e B conjuntos (HAMMACK, 2020). Um exemplo mais claro da operação de produto cartesiano pode ser consultado no capítulo anterior na Equação (4). A operação de produto cartesiano foi implementada de forma explícita, conforme

mostrado na Listagem 4. Nessa implementação, cada elemento de conjA é associado a cada elemento de conjB, e esses pares são adicionados a um conjunto de retorno utilizando o método *add*.

Listagem 4 – Operação de produto entre conjuntos implementada em Python

```
1 def produtoCartesiano(self, conjA, conjB):
2     if conjA == None or conjB == None:
3         return None
4     saida = set()
5     for elemA in conjA:
6         for elemB in conjB:
7             saida.add((elemA, elemB))
8     return saida
```

Fonte: Autoria própria (2023).

Definido como $\overline{A} = U - A$, onde A é um conjunto e U é o conjunto universo (HAMMACK, 2020) que corresponde a todos os elementos possíveis. Um exemplo prático dessa operação pode ser visto na Equação (5) no capítulo anterior. Sendo a implementação mais simples, apenas se realiza a operação de subtração já implementada entre o conjunto A e o conjunto universo, como mostrado na Listagem 5. O conjunto universo definido neste projeto é representado por $U = \{-101, -100, ...100, 101\}$. No entanto, é importante salientar dois pontos. Primeiro, para não limitar os valores com os quais o interpretador pode trabalhar, caso algum elemento de conjA não exista no conjunto universo, a operação de complemento será tratada como uma operação de diferença normal entre conjUniverso e conjA, e as operações continuarão a ser executadas. Segundo, em versões futuras, espera-se que o software tenha uma tela de parâmetros, na qual o conjunto universo possa ser definido pelo próprio usuário junto com outras configurações que surjam conforme as futuras necessidades da aplicação.

Listagem 5 – Operação de complemento implementada em Python

```
1 def complemento(self, conjA):
2     if conjA == None or self.conjUniverso == None:
3         return None
4     return self.diferenca(self.conjUniverso, conjA)
```

Fonte: Autoria própria (2023).

Considerando um conjunto A qualquer, o conjunto das partes de A é denotado por $P(A) = \{X : X \subseteq A\}$ (HAMMACK, 2020). Em termos leigos, é possível descrever o conjunto das partes como "o conjunto que contém todos os subconjuntos de A", conforme exemplificado na Equação (6) do capítulo anterior. Como exibido na Listagem 6, devido à complexidade na implementação da operação de conjunto das partes, foi primeiro desenvolvido um método que gera o conjunto de todos os subconjuntos de um conjunto A com uma certa cardinalidade n (omitido nas linhas 1 e 2 devido ao tamanho do método). Em seguida, no método principal,

é feita a união de cada um desses conjuntos gerados, começando pela cardinalidade 1 até a cardinalidade do conjunto original e a cardinalidade 0.

Listagem 6 – Operação de conjunto das partes implementada em Python

```
1  def conjuntoDasPartesCardinalidade(self, conjAOri, cardinalidade):
2      #codigo...
3
4  def conjuntoDasPartes(self, conjA):
5      if conjA == None:
6          return None
7      saida = set()
8      if len(conjA) == 0:
9          saida.add(frozenset({}))
10         return saida
11     cont = 1
12     while cont <= len(conjA):
13         saida = self.uniao(saida,
14             self.conjuntoDasPartesCardinalidade(conjA, cont))
15         cont += 1
16     saida.add(frozenset({}))
17     return saida
```

Fonte: Autoria própria (2023).

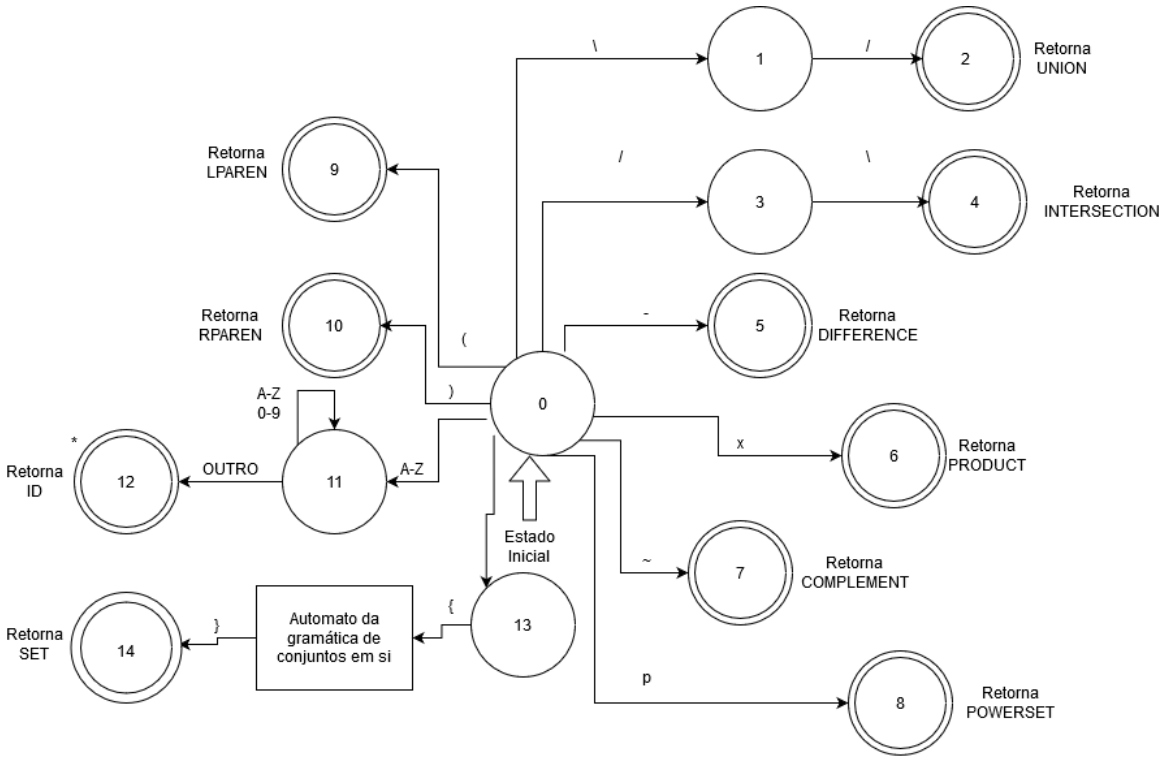
3.1.2 Analisador léxico

A construção do analisador léxico, em sua primeira versão, usou como base a teoria do capítulo 3 do livro “Compiladores: Princípios, técnicas e ferramentas” (AHO *et al.*, 2007), que é uma leitura clássica e fundamental a qualquer um que trabalhe com construção de compiladores. Mais especificamente foi implementado o autômato finito determinístico que representa os lexemas de teoria dos conjuntos, representado nas Figuras 5 e 6, em uma estrutura de “if e else” para extração de tokens.

Entre os estados 13 e 14, representados na Figura 5, encontra-se a estrutura completa do autômato da Figura 6. A escolha de mantê-lo separado do resto do autômato geral ocorreu devido à implementação do mesmo ter sido feita como uma parte separada do autômato que interpreta as expressões como um todo. Esse autômato específico é acionado pela leitura do caractere "{", que serve como gatilho inicial para iniciar a conversão de uma entrada textual em um conjunto e *token SET* utilizável nas operações do autômato geral.

É importante salientar que, devido à complexidade que o autômato da Figura 6 apresentaria e à preocupação do autor com a sua legibilidade, o autômato encontra-se resumido, não representando certos casos que ele admite, como o conjunto vazio e o consumo de espaços excedentes dentro dos conjuntos. Entretanto, ainda é possível evidenciar as transições principais e os *tokens* gerados pelo autômato. Por exemplo, é possível observar que, ao receber o caractere ".", o autômato sempre prossegue até reconhecer uma variável float dentro do

Figura 5 – Autômato que representa a gramática que rege expressões de teoria dos conjuntos.



Fonte: Autoria própria (2023).

conjunto declarado. No entanto, é importante ressaltar que ele não será capaz de reconhecer, por exemplo, strings que contenham esse caractere no meio, gerando assim um erro de leitura devido a um valor não admitido pelo interpretador.

Ressalta-se também que o autômato só trabalha com conjuntos que não possuam em si outros conjuntos como subelemento, aceita conjuntos vazios e ignora espaços excedentes que vem antes e após elementos do conjunto. Exemplos de valores aceitos internamente nos conjuntos estão listados no Quadro 3. Já no Quadro 2, está expressa a estrutura sintática de um conjunto, levando em consideração as vantagens e limitações mencionadas anteriormente neste parágrafo e no parágrafo anterior.

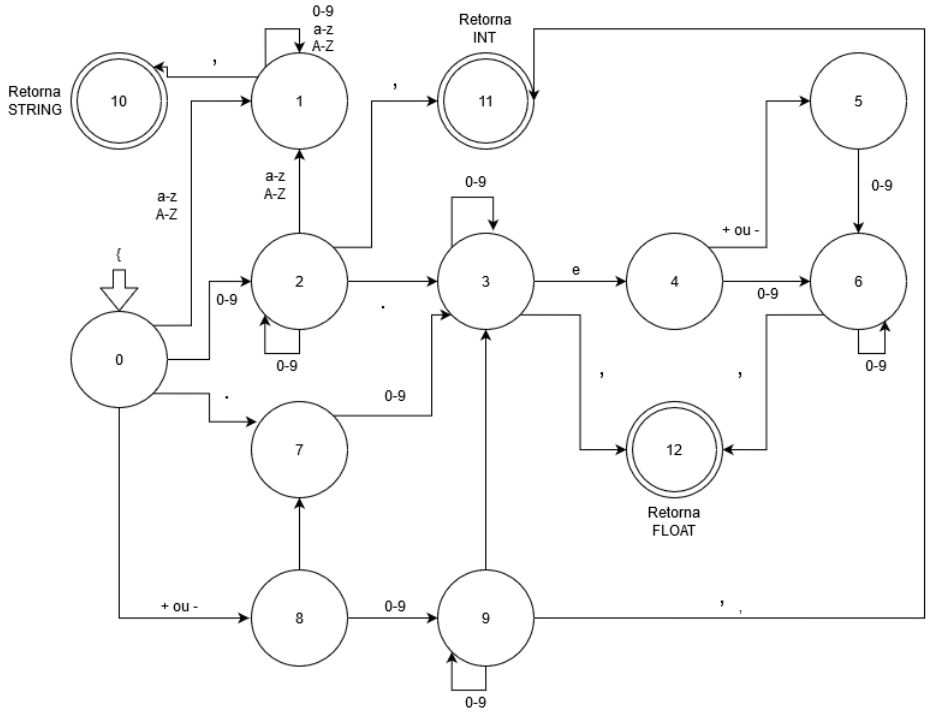
Quadro 2 – Gramática de declarações de conjuntos.

Derivações
$S \rightarrow \{A\} \mid \{\}$
$A \rightarrow inteiro\ B \mid decimal\ B \mid palavra\ B$
$B \rightarrow ,A \mid , \mid \epsilon$

Fonte: Autoria própria (2023).

Para a implementação do autômato que converte as expressões como um todo, foram utilizados dois apontadores para marcar o início e o fim dos lexemas nas cadeias de caracteres, cujo o apontador fim de lexema deve retornar uma posição da entrada sempre quando encontrar um estado com “*” acima. Um exemplo de estado com "*" pode ser encontrado na Figura

Figura 6 – Autômato que rege a gramática de declarações de conjuntos.



Fonte: Autoria própria (2023).

Quadro 3 – Exemplo de *tokens* e lexemas aceitos pelo interpretador de conjuntos.

Exemplos de entrada	**Token**
1, +303, -4000, 0...	INT
vermelho, teste123, 12tres, TESTE...	STRING
10.567, .009, -10.3e+10, +9.8765...	FLOAT

Fonte: Autoria própria (2023).

5 acima do estado 12. Essa implementação foi baseada nas sugestões da bibliografia citada anteriormente (AHO *et al.*, 2007).

Ao contrário do autômato anterior, o autômato que processa as expressões como um todo segue fielmente o apresentado na Figura 6 e no Quadro 5. Por exemplo, quando é feita a leitura de operadores simples como "$-$", "$\times$", "$\sim$" e "p", ou dos parênteses "$($" e "$)$", a geração de *token* é direta, conforme mostrado nas transições do estado 0 para os estados 5, 6, 7, 8, 9 e 10, respectivamente. Em termos de implementação em código, isso se traduz em uma condicional simples que lê um único caractere com os ponteiros inicial e final posicionados sobre o mesmo lexema.

Ainda seguindo o autômato da Figura 5, existem os casos dos operadores de união e intersecção que, por serem representados por mais de um caractere, necessitam de duas transições de estado. Essas transições ocorrem do estado 0 para o estado 1 e, em seguida, para o estado 2, no caso da união, e do estado 0 para o estado 3 e, por fim, para o estado 4, no caso da intersecção. Em termos de código, isso se resume a uma primeira leitura do caractere

"$\backslash$" ou "$/$", que move o ponteiro final para o próximo caractere e analisa se é o esperado para gerar os *tokens* de "UNION"e "INTERSECTION".

Como casos distintos do autômato da Figura 5, existe a transição já explicada do estado 0 para o estado 13 e, em seguida, para o estado 14. Além disso, existe também o trecho do estado 0 até o estado 12, que representa os identificadores de conjuntos. Conforme pode ser observado na figura, a string sempre deve começar com uma letra, mas em seu meio e final podem conter tanto letras quanto números. Essa escolha foi feita para seguir a convenção de linguagens como o C, que não permitem que identificadores de variáveis comecem com números e restringe outros tipos de caracteres. Além disso, essa escolha também segue o formato em que os conjuntos são geralmente escritos em materiais didáticos da área de matemática computacional. Ao seguir essa convenção, mantém-se a consistência com outras linguagens de programação e facilita a compreensão e o uso dos identificadores de conjuntos pelos usuários. O Quadro 4 ilustra exemplos de expressões admissíveis e inadmissíveis pela ferramenta criada neste projeto e o Quadro 5 relaciona *tokens* e lexemas da análise léxica.

Quadro 4 – Entradas admitidas pelo interpretador de expressões de conjuntos.

Entrada	**É admitida?**
$\sim \{0\}$ - $\{1,2\}\backslash/\{3\}/\backslash p\{0,1\} \times \{\}$	**Sim**.
$B0 = \{0,1,2\} - \{3\}$	**Sim**. Atribuição a uma variável é admitido.
$A = \{\}$	**Sim**. O conjunto vazio é admitido.
$E0$	**Sim**. E0 tem que ter sido previamente declarado.
$\{0$,1,2,$azul, -23.4, fibonacci1123\}$	**Sim**. Conjuntos podem ter *int*, *string* e *float* como elementos.
$\forall \in A \exists 0$	**Não**. Os símbolos $\forall$, $\in$ e $\exists$ não são reconhecidos.
$\{3,4,5,\{0,1\}\}$	**Não**. Conjunto com conjunto como elemento não é válido.

Fonte: Autoria própria (2023).

3.1.3 Analisador sintático

Em seguida foram construídos 4 algoritmos de análise sintática para fins de comparação. Primeiro foram implementados o “método de Unger sem produções vazias” e o “algoritmo Cocke-Younger-Kasami” (CYK) seguindo a referência (GRUNE; JACOBS, 1990). Sendo o método de Unger o "menos exigente", pois não requer que a gramática esteja em uma forma específica, como a forma normal de Chomsky ou LL(1), foi construída a gramática apresentada no Quadro 6. No entanto, devido à presença de ambiguidade na gramática e ao fato de o algoritmo Unger gerar todas as derivações possíveis para uma mesma expressão, a saída desse algoritmo resultou em uma complicada rede de classes instanciadas e encadeadas entre si de

Quadro 5 – Lista de *tokens* e lexemas.

Entrada	**Token**
$=$	EQUALS
$\backslash/$	UNION
$/\backslash$	INTERSECTION
-	DIFFERENCE
$\times$	PRODUCT
$\sim$	COMPLEMENT
p	POWERSET
$($	LPAREN
$)$	RPAREN
$[A-Z][A-Z\|0-9]^*$	ID
$[\{]\ [nmerointeiro, \mid nmerofracionrio, \mid cadeiadecaracteres,]^*\ [\}]$	SET

Fonte: Autoria própria (2023).

forma complexa. Isso já indicava, antes mesmo dos testes de desempenho, que o uso desse método não seria adequado para o projeto.

Quadro 6 – Gramática desenvolvida para o método de Unger.

Derivações
$S \rightarrow id = A \mid A$
$A \rightarrow A \cup A \mid A \cap A \mid A \times A \mid A - A \mid \sim A \mid pA \mid (A) \mid id \mid set$

Fonte: Autoria própria (2023).

Sendo necessário uma gramática na forma normal de Chomsky, a gramática do Quadro 6 foi convertida para a gramática do Quadro 7, a qual pode ser utilizada no algoritmo CYK. Como resultado do CYK, foi gerada uma tabela característica do próprio algoritmo. Além disso, foi necessária a conversão dessa tabela em uma árvore utilizável para a aplicação. No entanto, assim como no método de Unger, em caso de ambiguidade, várias árvores poderiam ser geradas, e em alguns casos, a expressão poderia gerar uma tabela muito grande, o que não justificaria o uso de memória e tempo de processamento, indicando que o algoritmo poderia não ser adequado para o uso final do interpretador.

O uso excessivo de memória era agravado pela utilização de uma matriz $n \times n$ do próprio Python para representar a tabela do CYK. Isso resultava em vários espaços da estrutura de dados sendo desperdiçados, uma vez que nem todos eram utilizados. No entanto, nesse ponto, já estava decidido que otimizar essa questão geraria complexidade e exigiria mais tempo na implementação do método, o que não justificava o seu uso. Também é importante notar que a

forma normal de Chomsky naturalmente "infla" a linguagem, criando certas derivações que não são tão intuitivas e que podem comprometer o uso didático da ferramenta.

Quadro 7 – Gramática na forma normal de Chomsky para o algoritmo CYK.

Derivações
$S \rightarrow HG \mid AB \mid AC \mid AD \mid AE \mid IF \mid JA \mid KA \mid id \mid set$
$A \rightarrow AB \mid AC \mid AD \mid AE \mid IF \mid JA \mid KA \mid id \mid set$
$B \rightarrow LA$
$C \rightarrow MA$
$D \rightarrow NA$
$E \rightarrow OA$
$F \rightarrow AP$
$G \rightarrow QA$
$H \rightarrow id$
$I \rightarrow ($
$J \rightarrow \sim$
$K \rightarrow p$
$L \rightarrow \cup$
$M \rightarrow \cap$
$N \rightarrow \times$
$O \rightarrow -$
$P \rightarrow)$
$Q \rightarrow =$

Fonte: Autoria própria (2023).

Posteriormente, utilizando as definições apresentadas na construção do analisador léxico, foi implementado o "analisador sintático preditivo não-recursivo" (AHO *et al.*, 2007) em conjunto com uma gramática LL(1) capaz de gerar as expressões da teoria dos conjuntos. Essa implementação foi baseada não apenas nas instruções da referência principal, mas também no conhecimento prévio da comunidade de desenvolvedores (SASAKI, 2019) e em artigos (ALI *et al.*, 2020) que abordaram o algoritmo ou construíram gramáticas LL(1) relacionadas à aritmética com sucesso. Foram feitas adaptações específicas para utilizar a teoria dos conjuntos em vez da aritmética que podem ser vistas no Quadro 8.

É importante salientar que, como a gramática do Quadro 8 não possui suporte para o operador de atribuição "EQUALS", foi necessário implementar certas regras antes de aplicar o algoritmo diretamente. Por exemplo, verificou-se a existência de apenas um operador de atribuição na entrada, a verificação de que o que está à esquerda do operador de atribuição é apenas

um *token* ID e a verificação de que o *token* "EQUALS" está na segunda posição da entrada, que é a única posição admitida neste projeto. Após essas verificações, caso exista uma atribuição, tudo à direita dela é processado seguindo a gramática LL(1). Caso contrário, o "analisador sintático preditivo não-recursivo" é aplicado diretamente.

Por fim, sendo a saída desse analisador uma única árvore de derivação devido à ausência de ambiguidade na gramática desenvolvida, ainda seria necessário realizar o passo intermediário de converter a árvore de derivação para uma árvore em que seja possível percorrer seus nós e realizar as operações com conjuntos e variáveis. Essa conversão permitiria uma representação adequada para a manipulação dos conjuntos e variáveis durante a execução do programa.

Quadro 8 – Gramática na forma LL(1) para o analisador sintático preditivo não-recursivo.

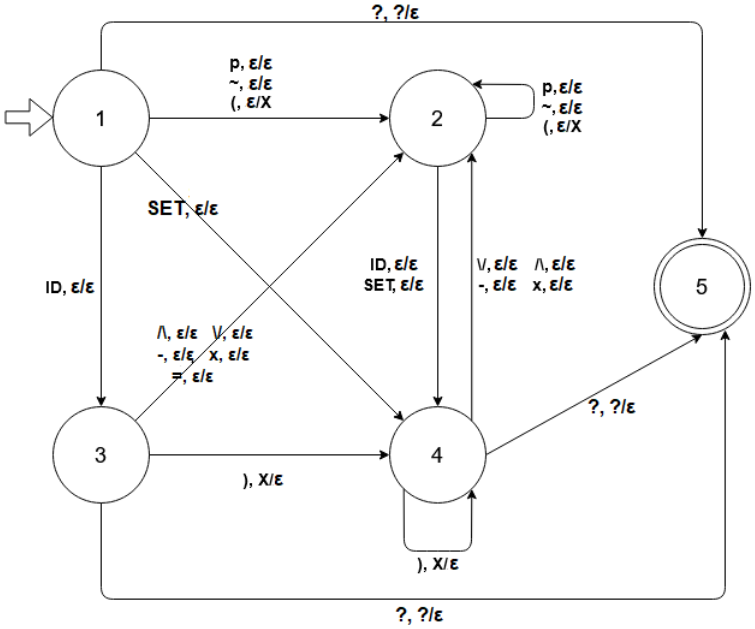
Derivações
$A \rightarrow BA'$
$A' \rightarrow \cup BA' \mid \epsilon$
$B \rightarrow CB'$
$B' \rightarrow \cap CB' \mid \epsilon$
$C \rightarrow DC'$
$C' \rightarrow \times DC' \mid \epsilon$
$D \rightarrow ED'$
$D' \rightarrow -ED' \mid \epsilon$
$E \rightarrow pE \mid \sim E \mid F$
$F \rightarrow id \mid set \mid (A)$

Fonte: Autoria própria (2023).

O último método implementado foi um autômato com pilha (expresso na Figura 7) em conjunto com o “algoritmo Shunting-yard” explicado por (WOLF, 2011), para validação de expressões e geração da saída da análise sintática respectivamente. Devido à natureza do autômato da Figura 7, que é utilizado apenas para validação, não houve preocupações com questões como precedência de operadores. O principal objetivo desse autômato é analisar o balanceamento de parênteses e a estrutura sintática das operações, garantindo que o que chegue para ser processado pelo Shunting-yard não gere erros.

No entanto, é importante ressaltar que esse autômato foi construído com base na primeira gramática desenvolvida no Quadro 6. Caso o leitor esteja procurando por gramáticas não ambíguas que lidem com a precedência de operadores, as gramáticas do Quadro 8 e a gramática proposta no Quadro 9 podem ser utilizadas. Essas gramáticas seguem a ordem de precedência dos operadores da seguinte maneira: atribuição, união, intersecção, produto cartesiano, diferença e as operações unárias de complemento e conjunto das partes. Essa ordem de precedência é utilizada de forma intrínseca no Shunting-yard implementado.

Figura 7 – Autômato com pilha que rege a gramática de declarações de expressões da teoria dos conjuntos.



Fonte: Autoria própria (2023).

Quadro 9 – Gramática das expressões aceitas pelo interpretador deste trabalho.

Derivações
$S \rightarrow id = A \mid A \mid \epsilon$
$A \rightarrow A \cup B \mid B$
$B \rightarrow B \cap C \mid C$
$C \rightarrow C \times D \mid D$
$D \rightarrow D - Z \mid Z$
$Z \rightarrow conjunto \mid id \mid \sim Z \mid pZ \mid (A)$

Fonte: Autoria própria (2023).

Como exemplo de funcionamento do autômato de validação, é possível pegar uma expressão com os parenteses desbalanceados como "$(A \backslash / B$" que tokenizado estaria "LPAREN ID UNION ID". Em um primeiro momento, o autômato da Figura 7 consumiria o *token* "LPAREN" transicionando para o estado 2 e adicionando X a sua pilha, logo depois leria as operações fazendo várias transições entre o estado 2 e 4 até chegar definitivamente em 4. Porém como não ha mais nada na entrada e a pilha ainda se encontra com elementos não existe transição, logo um erro é gerado e a entrada não segue para o Shunting-Yard. É importante ressaltar que esse autômato também foi implementado utilizando uma estrutura condicional, juntamente com uma pilha e uma variável para armazenar o estado atual. Todo esse conjunto foi encapsulado em uma classe própria para facilitar o seu uso e organização.

Originalmente proposto por Edsger Dijkstra, o algoritmo Shunting-yard tem como principal objetivo a conversão de expressões infixas para expressões posfixas, que podem ser ava-

liadas utilizando uma pilha para sua resolução. Em termos de implementação, foi criada uma classe que contém a pilha e a fila de saída características do algoritmo. Além disso, foram utilizados dois dicionários internos à classe para relacionar os *tokens* dos operadores ("EQUALS", "UNION", "INTERSECTION", "DIFFERENCE", "PRODUCT", "COMPLEMENT" ou "POWERSET") com sua associatividade e número de precedência, como mostrado no código fonte da Listagem 7.

Listagem 7 – Dicionários que alimentam a classe do Shunting-yard.

```
self.precedencia = {
    'EQUALS': 0,
    'UNION': 1,
    'INTERSECTION': 2,
    'PRODUCT': 3,
    'DIFFERENCE': 4,
    'COMPLEMENT': 5,
    'POWERSET': 6
}
self.associatividade = {
    'EQUALS':'left',
    'UNION':'left',
    'INTERSECTION':'left',
    'PRODUCT':'left',
    'DIFFERENCE':'left',
    'COMPLEMENT':'left',
    'POWERSET':'left'
}
```

Fonte: Autoria própria (2023).

Seguindo a ordem de precedência e associatividades expressas na Listagem 7, o algoritmo Shunting-yard, implementado internamente na classe, ao receber a entrada na forma de uma lista de *tokens* já validada, segue as seguintes regras:

• Caso o *token* lido seja "SET" ou "ID" ele é adicionado a fila de saída;

• Caso o *token* lido seja um "LPAREN" ele é adicionado ao topo da pilha;

• Caso o *token* lido seja "RPAREN", ele é descartado e todos os elementos da pilha são desempilhados e adicionados ao final da fila de saída até encontrar um "LPAREN", que também é desempilhado e descartado;

• Caso o *token* lido seja um operador e a pilha esteja vazia ou o elemento do topo seja um "LPAREN", o operador é empilhado;

• Caso o *token* lido seja um operador e a pilha contenha outro operador no topo cuja precedência seja menor, o operador lido é empilhado. No caso em que o *token* lido tem sua associatividade à direita, a condição muda para menor ou igual;

• Caso o *token* lido seja um operador e a pilha contenha outro operador no topo cuja precedência seja maior, então os operadores da pilha são desempilhados e adicionados à fila de saída até chegar a um operador que obedeça à condição do ponto anterior, chegue a um

token "LPAREN" ou até a pilha ficar vazia. Em seguida, o *token* lido pode ser adicionado ao topo da pilha. No caso em que o *token* lido tem sua associatividade à esquerda, a condição muda para maior ou igual;

• Ao final da entrada, todos os elementos restantes na pilha são desempilhados e adicionados à fila de saída, gerando assim o resultado final em forma de uma lista de *tokens* em notação polonesa reversa.

Todos os algoritmos de análise sintática foram implementados utilizando a estrutura de dicionário do Python para alimentá-los com suas respectivas gramáticas ou configurações necessárias. Isso também abriu a possibilidade que eles trabalhassem com diferentes tipos de operações, por exemplo, boolenas ou numéricas. No entanto, neste artigo, essa propriedade não será desenvolvida, pois este trabalho foca apenas na teoria dos conjuntos. Além disso, foram realizados testes de desempenho em todos os algoritmos sintáticos, o que levou à escolha definitiva do uso do Shunting-yard no projeto. Os resultados desses testes estão apresentados na Seção 4.1.

3.1.4 Analisador semântico

Tendo já sido estabelecida a abordagem "autômato com pilha em conjunto com o Shunting-yard" como o analisador sintático principal do projeto(justificada no Capítulo 4), o analisador semântico teve a função de resolver as expressões na notação polonesa reversa (RPN). Para isso, foi criada uma classe chamada Solver, que tem a responsabilidade de resolver as expressões e também gerenciar sua tabela de endereços interna, buscando ou inserindo conjuntos na tabela quando necessário.

Proposta por Jan Łukasiewicz em 1924 para eliminar a necessidade de uso de parênteses em expressões (HAMBLIN, 1962), a notação polonesa reversa utiliza uma pilha de operandos para realizar cálculos conforme a leitura da expressão de entrada. No contexto deste projeto, o método interno da classe Solver segue as seguintes regras:

• Caso o *token* lido na entrada seja "SET" ou "ID" ele é empilhado na pilha interna do Solver;

• Ao encontrar um operador na entrada, é verificado o número de operandos necessários para a operação. Em seguida, são desempilhadas da pilha a mesma quantidade de variáveis e a operação é realizada entre esses operandos, considerando a primeira variável desempilhada como o operando mais à direita e a última variável desempilhada como o operando mais à esquerda. Após a realização da operação, o resultado é inserido no topo da pilha;

• Ao final da entrada, espera-se que a pilha contenha um único elemento, que é o resultado final da expressão.

Como exemplo prático de uma operação em notação polonesa reversa, é exposta a Figura 8, na qual é possível observar as alterações na entrada e na pilha em cada etapa, seguindo as regras da notação polonesa reversa mencionadas anteriormente. É importante destacar que

a expressão "$\{0,1,2\}\{1,2\} \cap \{2\}-$" é equivalente à expressão "$(\{1,2\} \cap \{0,1,2\}) - \{2\}$". É importante notar também que todas as operações explicadas na Subseção 3.1.1 são métodos internos da classe Solver.

Figura 8 – Exemplo de operação envolvendo uma entrada em notação polonesa reversa.

PASSO	ENTRADA	PILHA	OPERAÇÃO
1	{0,1,2} {1,2} ∩ {2} -		
2	{1,2} ∩ {2} -	{0,1,2}	
3	∩ {2} -	{1,2} {0,1,2}	
4	{2} -	∩ {1,2} {0,1,2}	{0,1,2} ∩ {1,2}
5	{2} -	{1,2}	
6	-	{2} {1,2}	
7		- {2} {1,2}	{1,2} - {2}
8		{1} ⇦ RESULTADO	

Fonte: Autoria própria (2023).

Durante a análise semântica, também foram impostas certas limitações ao sistema para garantir um desempenho aceitável. Por exemplo, devido ao custo computacional exponecial, a operação "conjunto das partes" só pode ser aplicada a conjuntos de cardinalidade igual ou inferior a 8. Da mesma forma, a operação "produto cartesiano" só pode ser realizada se o produto das cardinalidades dos conjuntos envolvidos for igual ou inferior a 1000. Além disso, em caso de erro durante a análise semântica, o sistema retornará um valor nulo em vez de um conjunto como resposta. Isso é feito para indicar que ocorreu um problema na expressão ou nas operações realizadas.

Por fim os testes de desempenho e corretude de cada uma das etapas de análise do interpretador serão tratados na Seção 4.1.

3.2 Interface Gráfica

Para a implementação da interface gráfica foi utilizada a biblioteca PyQt5, pois como descrito por (JOST, 2013) o PyQt possui uma grande quantidade de componentes, plataforma gráfica robusta, bastante próximo ao “padrão de programação” da linguagem Python e grande qualidade de software.

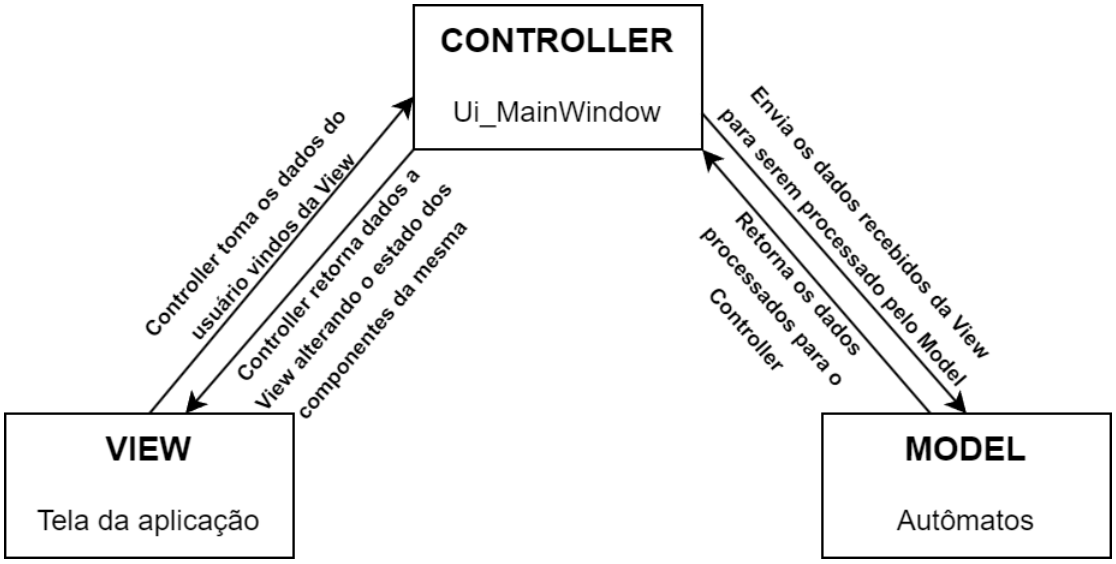
Na construção das telas foi utilizada a parte cabível do padrão de projeto MVC. Nascido durante os anos 80, o padrão de projeto MVC (*Model-View-Controller*) tem como objetivo a separação das regras de negócio da interface do sistema e fluxo da aplicação. Tal separação traz como vantagens a facilidade de manutenção do software e a possibilidade de diversas interfaces acessarem as mesmas regras de negócio.

Como explicado por (DEACON, 2009), o *Model* pode ser descrito como a parte imutável do modelo, ou seja, as classes que modelam o problema e não interagem diretamente com o

"mundo externo". A *View* são as classes que entregam as janelas de exibição ou outra forma de interação ao usuário, o *View* deve saber que a camada *Model* existe e sua natureza. Por fim o *Controller* é um objeto que manipula a *View* tomando as entradas da mesma e as devolvendo ao *Model* para gerar uma saída na exibição.

Devido ao software implementado não necessitar de interação com um banco de dados o *Model* se tornou restrito às classes dos autômatos léxico, sintático e semântico e seus métodos como mostrado nos códigos fonte reduzidos nas Listagens 8, 9 e 10. O *Controller*, como mostrado na Listagem 11, tem todas as funções que recebem os cliques e entradas do usuário e manda as mesmas para os autômatos da camada *Model* processarem. Um exemplo gráfico do que foi explicado neste parágrafo e no anterior pode ser visto na Figura 9.

Figura 9 – Padrão MVC adaptado para as necessidades do projeto.



Fonte: Autoria própria (2023).

Listagem 8 – Classe do autômato léxico

```
1  class AutomatoLexico:
2      def __init__(self, entrada):
3          #codigo...
4      def resetar(self, entrada):
5          #codigo...
6      def proximo(self):
7          #codigo...
8      def saida(self):
9          #codigo...
10     def geraConjunto(self, entrada):
11         #codigo...
```

Fonte: Autoria própria (2023).

Todos os autômatos da aplicação têm como aspectos comuns um método "próximo", que faz com que o autômato prossiga para o próximo estado de leitura, uma função "saída" que

Listagem 9 – Classe do autômato sintático

```
1 class AutomatoShuntingYard:
2     def __init__(self, entrada):
3         #codigo...
4     def resetar(self, entrada):
5         #codigo...
6     def proximo(self):
7         #codigo...
8     def saida(self):
9         #codigo...
```

Fonte: Autoria própria (2023).

retorna apenas a saída final sem fornecer detalhes sobre o funcionamento do processo, uma função "resetar" que coloca o autômato no estado inicial com uma determinada entrada e o próprio construtor da classe, que também inicializa o autômato usando a entrada fornecida.

Como particularidade, o autômato léxico possui o método "geraConjunto", que recebe uma entrada textual de um conjunto e a converte em uma variável do tipo *Set*. Já a classe responsável pelo processamento semântico possui as funções "adicionarConjuntoAMemoria" e "removerConjuntoDaMemoria" para gerenciar seu dicionário interno, chamado "memConjunto", que representa a memória do autômato. Além disso, o autômato semântico possui todas as funções necessárias para operar variáveis do tipo *Set*.

Todas as telas seguem os principios colocados na seção 2.4, um botão de controle do passo do algoritmo e um cabeçalho onde o usuário inserir as próprias expressões. Para a interface do analisador léxico os autores idealizaram uma tela onde o usuário insere a expressão e em seguida o mesmo controla o processo visualizando a leitura da linha de entrada e a geração da lista de *tokens* de saída.

A tela de análise sintática foi projetada de forma bastante semelhante à do léxico, porém a mesma exibe a entrada digitada pelo usuário juntamente com a pilha e a fila características do Shuntig-yard. Ja a tela da análise semântica possui em sua exibição a saída com o resultado da expressão e um local para visualizar os conjuntos salvos em memória.

Como exibido em mais detalhes nas três Figuras seguintes (10, 11 e 12), todas as telas possuem em seu topo um campo para inserir uma expressão qualquer, um botão "gerar" que preenche o campo com uma expressão aleatória, o botão "confirmar" que inicia todos os autômatos do software utilizando a expressão digitada e o botão "automático" que, além de inicializar os autômatos, exibe todo o processo de compilação de forma cadenciada.

A tela da análise léxica, como mostrada na Figura 10, consiste em uma caixa de texto de entrada que mostra os ponteiros iniciais e finais do processo de tokenização e o trecho da leitura em vermelho, uma caixa de texto com as tuplas de *tokens* gerados e o botão "avançar" que prossegue para o próximo passo da leitura da expressão.

Na tela da análise sintática (Figura 11) existem caixas de texto que representam as estruturas de fila e pilha utilizadas pelo Shunting-yard; uma caixa de texto com a expressão de

Listagem 10 – Classe do autômato semântico

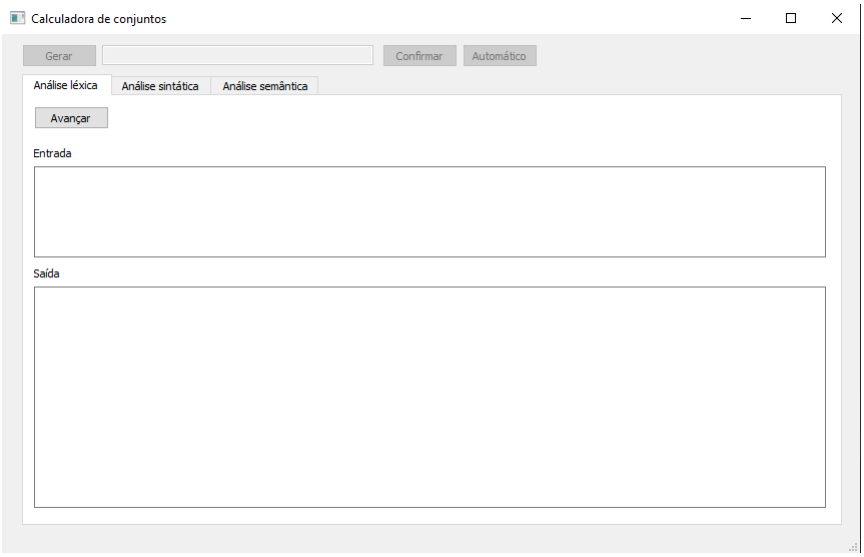
```
class AutomatoSolver:
    def __init__(self, entrada, conjUniverso):
        self.memConjunto = {}
        #codigo...
    def resetar(self, entrada):
        #codigo...
    def adicionarConjuntoAMemoria(self, strId, conj):
        #codigo...
    def removerConjuntoDaMemoria(self, strId):
        #codigo...
    def uniao(self, conjA, conjB):
        #codigo...
    def interseccao(self, conjA, conjB):
        #codigo...
    def diferenca(self, conjA, conjB):
        #codigo...
    def produtoCartesiano(self, conjA, conjB):
        #codigo...
    def complemento(self, conjA):
        #codigo...
    def conjuntoDasPartesCardinalidade(self, conjAOri, cardinalidade):
        #codigo...
    def conjuntoDasPartes(self, conjA):
        #codigo...
    def proximo(self):
        #codigo...
    def saida(self):
        #codigo...
```

Fonte: Autoria própria (2023).

entrada e o botão "avançar" que controla a leitura da expressão. Já os botões de "avançar" e "próximo", ao lado do nome do algoritmo, estão atrelados a essa tela para em versões futuras mudar o método sintático da tela.

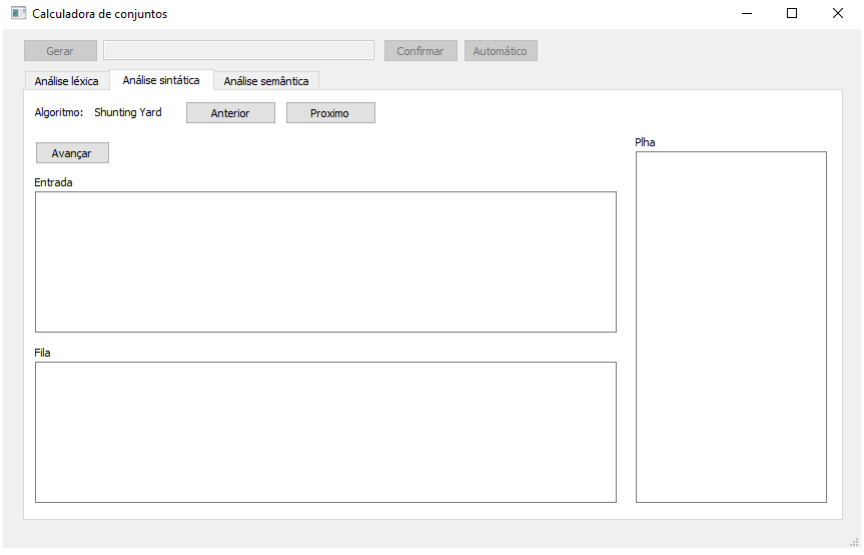
Já a tela da análise semântica (Figura 12) pode ser dividida em duas. A parte direita da tela possui uma tabela de dados mostrando os conjuntos armazenados na memória; duas entradas de texto para a identificação e valor de um novo conjunto e o botão "adicionar" para efetivamente realizar a inserção. A esquerda o campo “entrada” mostra, de forma similar a um console, como a operação vai se realizando conforme o usuário clica no botão "avançar". Exemplos de uso das três telas podem ser encontrados na Seção 4.2.

Figura 10 – Tela da análise léxica.



Fonte: Autoria própria (2023).

Figura 11 – Tela da análise sintática.



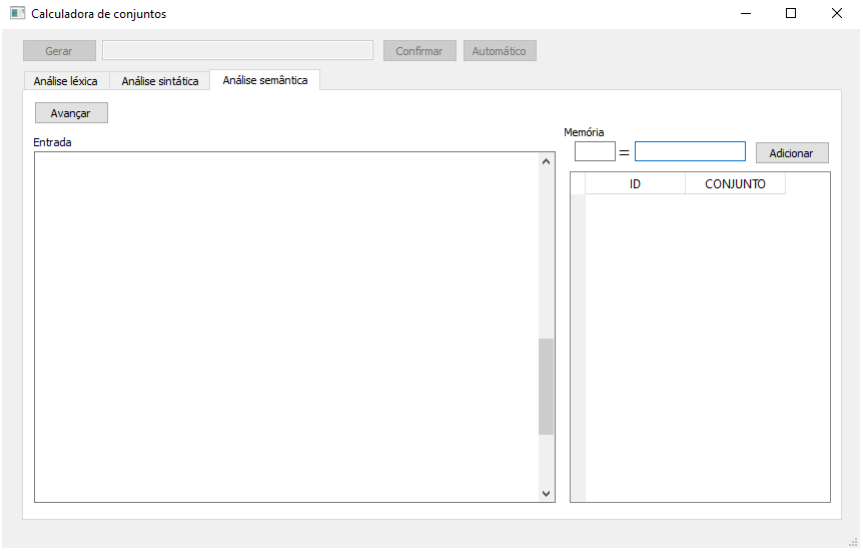
Fonte: Autoria própria (2023).

Listagem 11 – Controller da aplicação

```
class Ui_MainWindow(object):
    # Funcoes padroes do PyQt
    def setupUi(self, MainWindow):
        #codigo...
    def retranslateUi(self, MainWindow):
        #codigo...

    # Metodos gerais
    def iniciarAutomatos(self):
        #codigo...
    def gerarExpressao(self):
        #codigo...
    def confirmarExpressao(self):
        #codigo...
    def passoTmrAutomatico(self):
        #codigo...
    def iniciarTmrAutomatico(self):
        #codigo...
    def finalizarTmrAutomatico(self):
        #codigo...

    # Timers da aplicacao
    def habilitarLexicoTmrAutomatico(self):
        #codigo...
    def habilitarSintaticoTmrAutomatico(self):
        #codigo...
    def habilitarSemanticoTmrAutomatico(self):
        #codigo...

    # Metodos da aba Lexico
    def proximoLexico(self):
        #codigo...

    # Metodos da aba Sintatico
    def proximoSintaticoShuntingYard(self):
        #codigo...
    def formatacaoFilaSintaticoShuntingYard(self, lstTokens):
        #codigo...
    def formatacaoPilhaSintaticoShuntingYard(self, lstTokens):
        #codigo...

    # Metodos da aba Semantico
    def proximoSemantico(self):
        #codigo...
    def formatacaoEntradaSemantico(self):
        #codigo...
    def adicionarSemantico(self):
        #codigo...
    def gerarCelulaConjuntoVazia(self):
        #codigo...
```

Fonte: Autoria própria (2023).

Figura 12 – Tela da análise semântica.



Fonte: Autoria própria (2023).

4 EXPERIMENTOS

4.1 Comparação de Analisadores Sintáticos

Após a implementação, foram realizados testes de desempenho com os algoritmos de análise sintática e o sistema como um todo. O computador utilizado para os testes foi um notebook Dell Inspiron 3442 com as especificações expressas no Quadro 10.

Quadro 10 – Especificações do notebook usado nos testes.

Processador Intel(R) Core(TM) i5-4210U CPU @ 1.70GHz
Memoria ram 8 GB DDR3L-SDRAM 1600 MHz 1 x 8 GB
Memoria 1000 GB HDD DVD Super Multi
Sistema operacional Windows 10 Home Single Language Versão 21H2

Fonte: Autoria própria (2023).

O primeiro passo para todos os testes foi a construção de um método que gera expressões aleatórias em função do número de operadores binários, juntamente com a resposta esperada. Resumidamente, esse método gera uma árvore aleatória em função do número de operadores, onde os nós intermediários representam operadores e as folhas representam conjuntos também gerados aleatoriamente. Em seguida, essa árvore pode ser convertida em uma string ou lista de *tokens*, dependendo da parte do analisador que se deseja verificar, juntamente com a resposta esperada. Isso permite testar a análise do sistema como um todo.

4.1.1 Análise de desempenho dos algoritmo sintáticos

Para os testes de desempenho dos algoritmos sintáticos, foram geradas expressões aleatoriamente. As entradas variaram de 1 a 30 *tokens* com progressão de um em um. Para cada número de *tokens* e cada parte do analisador foram selecionadas 1.000.000 expressões de teste. A medição do tempo de execução de cada algoritmo foi realizada utilizando as ferramentas das bibliotecas padrões do Python, que fornecem o tempo em milissegundos. Em seguida, foi calculada a média dos resultados, separando-os de acordo com o número de operadores binários da linha.

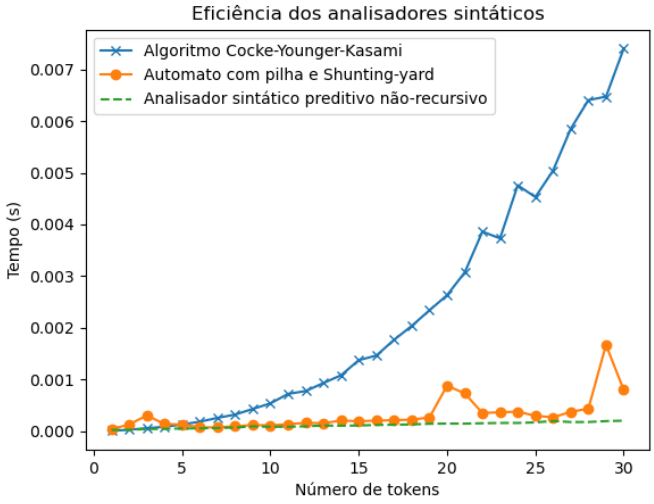
Nos Gráficos 1 e 2, o eixo x representa o número de *tokens* da expressão, enquanto o eixo y representa o tempo de execução necessário para a análise das expressões (em segundos). É importante ressaltar que o método de Unger foi separado dos outros algoritmos no Gráfico 1, devido ao seu comportamento assintótico exponencial, que dificultaria a visualização do desempenho dos demais algoritmos. Assim como expresso em suas respectivas literaturas fonte, já citadas anteriormente, o algoritmo CYK tem sua complexidade computacional sugerida como polinomial (assim como ilustrado no Gráfico 1), o método de Unger possui complexidade

aparentemente exponencial (observado no Gráfico 2), e tanto o Shunting-yard quanto o analisador sintático preditivo não-recursivo estão próximos a um comportamento linear (evidenciado empiricamente no Gráfico 2).

Gráfico 1 – Desempenho do método de Unger sem produções vazias

Fonte: Autoria própria (2023).

Gráfico 2 – Desempenho dos algoritmos de análise sintática



Fonte: Autoria própria (2023).

4.1.2 Analise de corretude

Nesta parte do projeto, optou-se por utilizar a abordagem do autômato com pilha em conjunto com o algoritmo Shunting-yard devido ao seu desempenho linear. O uso do Shunting-yard também é justificado pelo fato de sua saída ser a expressão em notação polonesa reversa (RPN), eliminando a necessidade de converter a árvore sintática abstrata em uma estrutura útil.

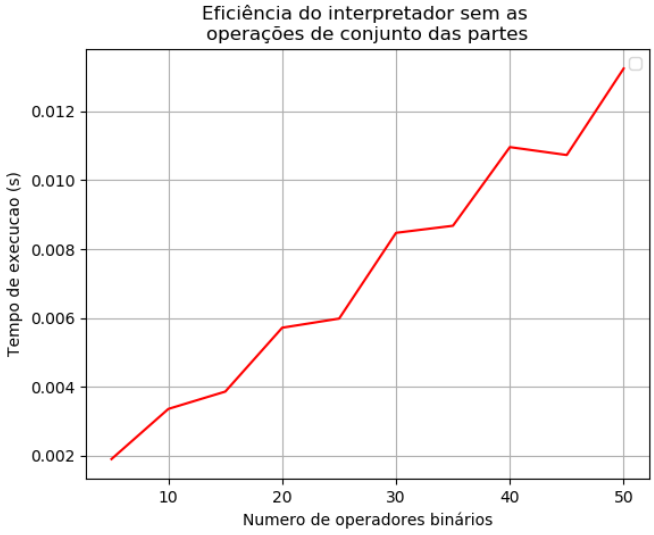
Isso facilita a análise semântica e a obtenção do conjunto resultante da expressão. Portanto, os testes de corretude concentraram-se nos algoritmos léxico, semântico e no Shunting-yard.

Na parte dos teste de corretude, todos os testes trabalharam com expressões geradas aleatoriamente em função do número de operadores, começando de 5 operações binárias até 50 em um intervalo de 5. Para cada número de operadores binários e cada parte do analisador foi gerado 10.000 expressões de teste.

De forma mais especifica os testes da análise léxica consistiam em gerar as expressões de entrada e as saídas tokenizadas corretas, comparando-as com os resultados produzidos pelo analisador léxico. Os testes da análise sintática seguiam uma abordagem semelhante, gerando as entradas e as saídas esperadas em notação polonesa reversa (RPN) e comparando-as com as saídas produzidas pelo analisador sintático. Já os testes da análise semântica e do sistema como um todo envolviam a geração de expressões de entrada e suas saídas esperadas, comparando-as com as saídas obtidas através do interpretador. Durante esses testes, as entradas e saídas esperadas foram geradas com base no número de operadores, variando como descrito no parágrafo anterior.

Os testes de validação realizados em cada parte da análise das expressões, bem como no sistema como um todo, não identificaram erros. Além disso, foi obtido o resultado ilustrado no Gráfico 3, que representa o desempenho geral do interpretador. O desempenho em cada ponto do gráfico foi calculado através da média simples do tempo de execução de cada expressão gerada em função do número de operadores binários da linha. O gráfico 3 sugere que, dentro do intervalo de testes, o interpretador possui um desempenho linear, sendo necessária analise teórica para confirmar.

Gráfico 3 – Desempenho do interpretador como um todo, ou seja, desde o processo de análise léxica até a obtenção do conjunto resposta



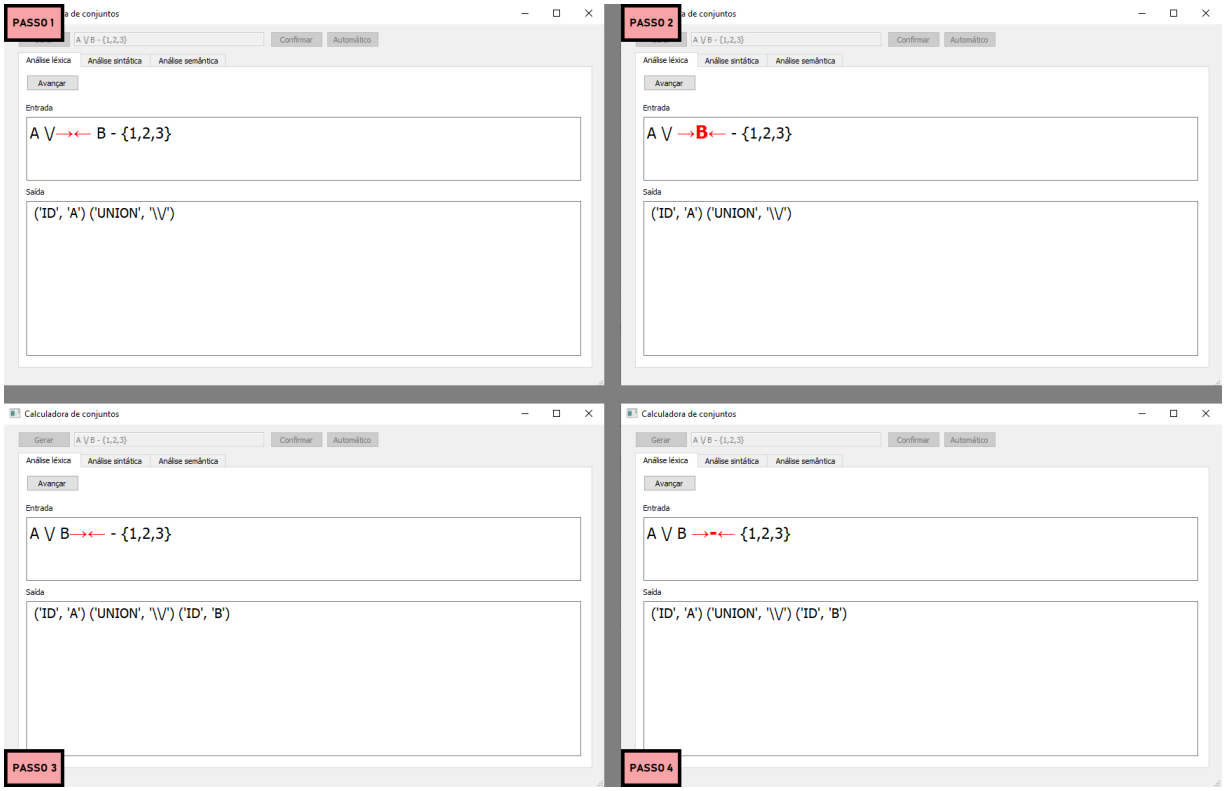
Fonte: Autoria própria (2023).

4.2 Usabilidade da Interface Gráfica

Como citado anteriormente, após analisado os algoritmos demonstrados na subseção anterior e suas performances, optou-se pelo uso do algoritmo Shunting-yard como *parser* da aplicação didática, porém os algoritmos de todas as etapas do processo foram reescritos para realizar a compilação de forma “passo a passo” e retornar os estados das estruturas de dados utilizadas em suas implementações.

Os resultados finais deste projeto podem ser resumidos nas interfaces gráficas e seus funcionamentos. Como exemplo de funcionamento da tela léxica, pode-se observar o processamento da entrada “$A \backslash / B - \{1,2,3\}$” na Figura 13. No Passo 1, os ponteiros iniciais e finais de leitura estão posicionados antes do identificador "B". Em seguida, no Passo 2, o ponteiro final é movido para a posição em que se encontra "B", identificando-o como o *token* "ID". Depois disso, no Passo 3, os ponteiros são deslocados para a posição anterior ao caractere "—". Por fim, no Passo 4, os ponteiros estão posicionados para a leitura do lexema "—".

Figura 13 – Exemplo de funcionamento da tela de análise léxica.

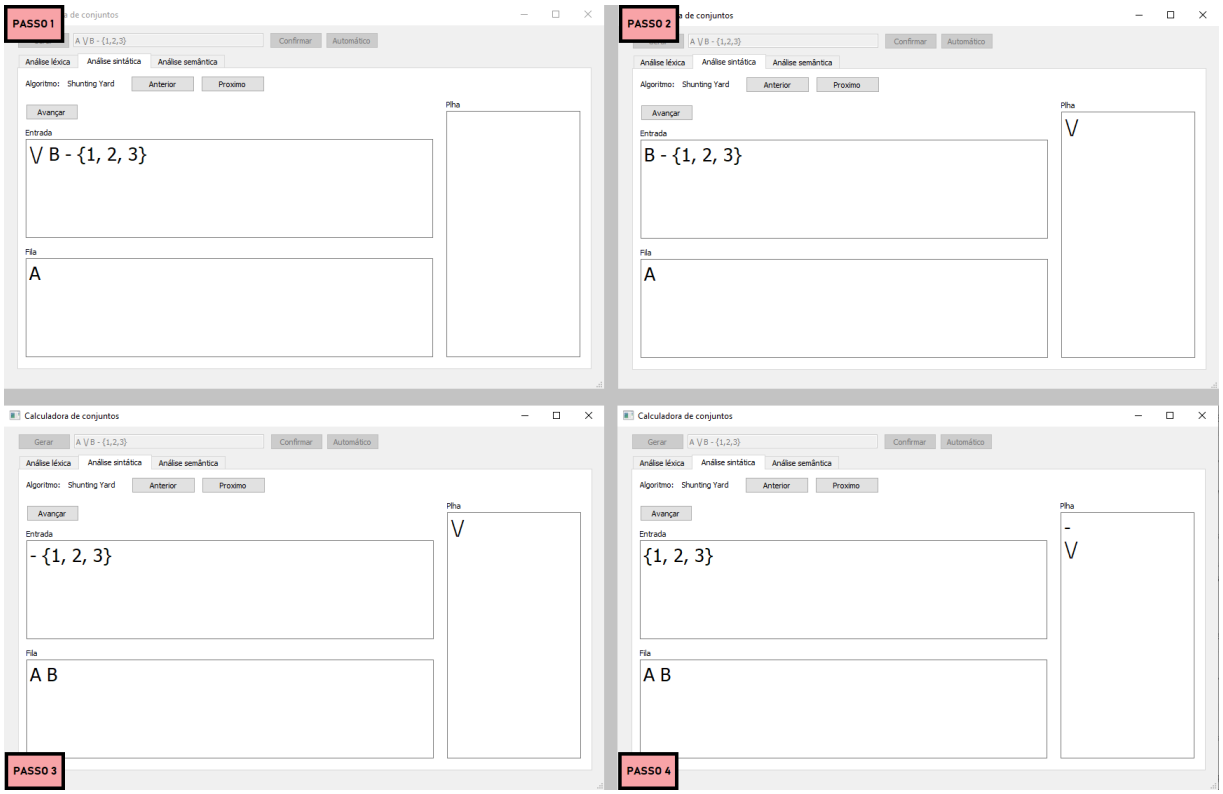


Fonte: Autoria própria (2023).

Também para exemplificar funcionamento da tela sintática, pode-se observar o processamento da mesma entrada do parágrafo anterior na Figura 14. No Passo 1, o identificador "A" entra na fila pois o mesmo trata-se de uma variável. Em seguida, no Passo 2, o operador de união é lido e inserido no topo da pilha vazia. Depois disso, no Passo 3 a variável B também

entra na fila. Por fim, no Passo 4, o operador de diferença fica acima do de união na pilha, pois o mesmo tem precedência maior em relação ao operador anterior.

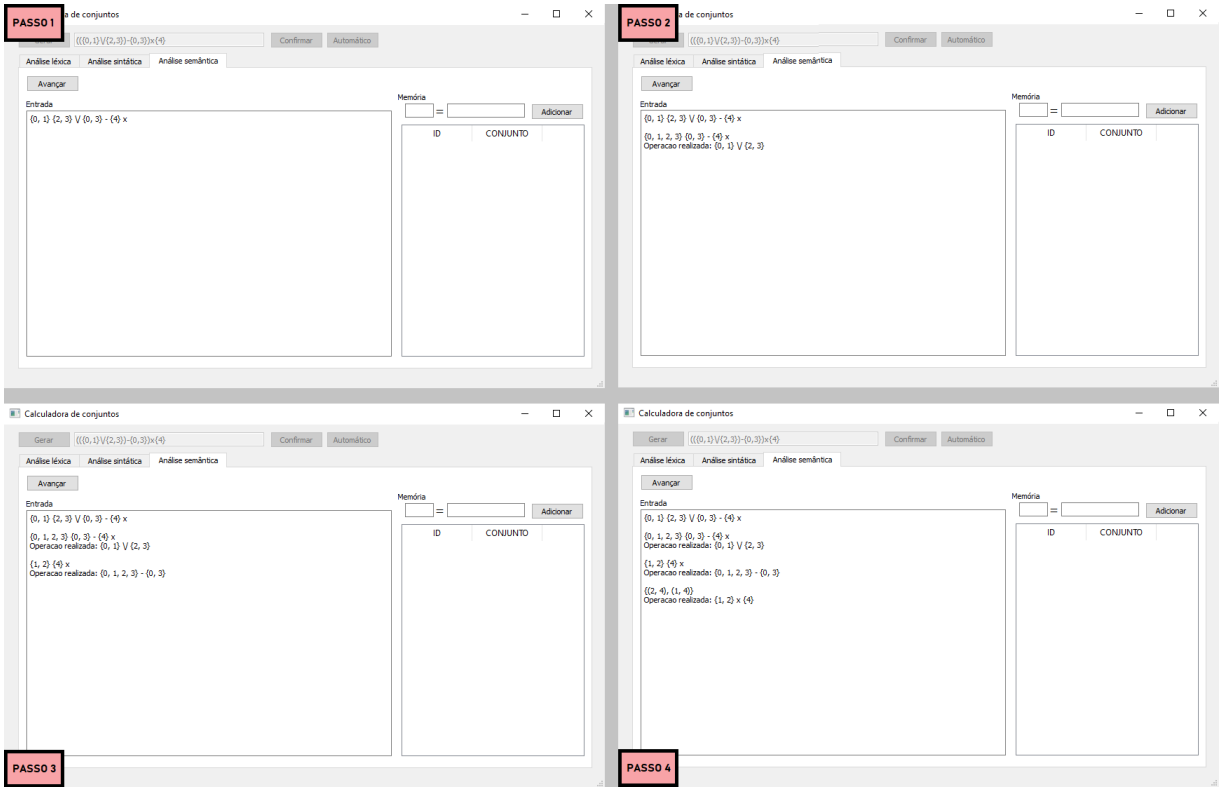
Figura 14 – Exemplo de funcionamento da tela de análise sintática.



Fonte: Autoria própria (2023).

Na tela semântica (Figura 15), é exposto um exemplo de processamento de entrada em que, a cada passo, a expressão é reduzida até chegar ao resultado final. Abaixo de cada linha é destacada a operação realizada para gerar a mesma.

Figura 15 – Exemplo de funcionamento da tela de análise semântica.



Fonte: Autoria própria (2023).

5 CONCLUSÃO

Neste trabalho, foi apresentado o processo de criação de um analisador didático de expressões de teoria de conjuntos. O trabalho abrangeu várias etapas, começando pela coleta de referências em teoria de conjunto, computação, compiladores e visualização de algoritmos. Em seguida, foram desenvolvidos autômatos léxicos e sintáticos, juntamente com a construção de gramáticas ambíguas e não ambíguas em diferentes formas. Além disso, foram implementados vários analisadores, incluindo analisadores sintáticos, um analisador léxico e um analisador semântico. Em seguida, ocorreu a reestruturação desses analisadores, ajustando seus funcionamentos e os integrando a interface gráfica desenvolvida.

Também neste trabalho foram conduzidos experimentos para avaliar o desempenho dos algoritmos sintáticos, o que levou à escolha do algoritmo "Shunting-yard" como o principal na aplicação. A partir dos experimentos de desempenho, também foi sugerido o comportamento assintótico exponencial do método de Unger, o comportamento não linear do algoritmo Cocke-Younger-Kasami e a linearidade dos algoritmos Shunting-yard e do analisador sintático preditivo não-recursivo.

Além do desempenho dos algoritmos sintáticos, também foi analisado o desempenho do interpretador como um todo. Foram realizados testes para medir o tempo de execução e a eficiência do interpretador ao processar expressões de diferentes tamanhos. Além dos testes de desempenho, também foram realizados testes para verificar empiricamente a corretude do sistema desenvolvido como um todo.

Por fim, foi desenvolvido um analisador de expressões de teoria dos conjuntos com um desempenho aparentemente linear. O analisador possui uma interface gráfica interativa que permite resolver diversas operações relacionadas à teoria de conjuntos. Além disso, o analisador demonstra as três etapas básicas do processo de compilação: análise léxica, sintática e semântica. O analisador oferece suporte para a geração de expressões aleatórias, permitindo ao usuário explorar diferentes casos de teste. Além disso, o usuário pode inserir expressões manualmente e observar o processo de análise passo a passo. O analisador também possibilita o gerenciamento e a observação do comportamento interno dos conjuntos durante a análise semântica.

Portanto, é certo dizer que o trabalho atingiu os objetivos específicos e geral propostos, dentro do domínio cabível da engenharia de computação.

5.1 Trabalhos Futuros

Visando o aprimoramento do software, é possível a implementação de interfaces gráficas para os demais algoritmos de análise sintática já implementados durante a primeira etapa deste trabalho. O projeto pode ser estendido também com a implementação de outros algoritmos presentes na literatura, tanto para os processos sintáticos quanto semânticos. É válido

considerar que, com o crescimento do projeto a longo prazo, como mencionado na seção 3.1.1, pode ser necessário incluir uma tela de parâmetros com as configurações que o projeto venha a necessitar. Essa tela de parâmetros permitiria aos usuários personalizarem e ajustarem diversas opções e configurações do sistema de acordo com suas necessidades e preferências.

É possível também adaptar o projeto visando uma melhor experiência do usuário, seja por meio de mudanças visuais para uma interface mais amigável ou pela adição de recursos que ainda não foram previstos no escopo inicial do projeto. Já em relação ao desempenho de ensino e à definição de parâmetros de comparação com outros softwares didáticos, bem como ao levantamento desses parâmetros e dos próprios softwares, também é possível realizar o trabalho em sala de aula com a ajuda de um pedagogo especializado.

Atualmente, o projeto está disponibilizado na plataforma de controle de versão GitHub. O autor destaca o link em negrito no final do último parágrafo do trabalho para leitores de outras seções interessados em acessar o código-fonte junto ao texto para uma melhor compreensão. O link é: "**https://github.com/Fellipesre/Analisador-de-express-es-de-teoria-dos-conjuntos-completo.git**".

REFERÊNCIAS

AHO, A. V. *et al.* **Compiladores: Princípios, Técnicas e Ferramentas - 2.ed.** [*S.l.*]: Pearson Universidades, 2007. 648 p. ISBN 9788588639249.

ALI, H. *et al.* Ll (1) parser versus gnf inducted ll (1) parser on arithmetic expressions grammar: A comparative study. **Quaid-E-Awam University Research Journal of Engineering, Science & Technology, Nawabshah.**, v. 18, n. 2, p. 89–101, 2020.

BAGARIA, J. **Set Theory**. 2014. Site Stanford Library. Disponível em: https://stanford.library.sydney.edu.au/entries/set-theory/. Acesso em: 23 set. 2020.

BEAZLEY, D. **Ply (Python Lex-Yacc)**. 2020. Dabeaz site do autor. Disponível em: http://www.dabeaz.com/ply/. Acesso em: 25 ago. 2020.

DEACON, J. Model-view-controller (mvc) architecture. **Online][Citado em: 10 de março de 2006.] http://www. jdl. co. uk/briefings/MVC. pdf**, v. 28, 2009. Acesso em: 07 fev. 2023.

FARIAS, G.; MEDEIROS, E. S. Introdução à computação. **v1. 0, Universidade Aberta do Brasil**, 2013.

FERREIRÓS, J. **Labyrinth of thought: A history of set theory and its role in modern mathematics**. [*S.l.*]: Springer Science & Business Media, 2008.

GRUNE, D.; JACOBS, C. **Parsing Techniques: A Practical Guide**. 1st. ed. [*S.l.*]: Ellis Horwood Ltd, 1990. (Ellis Horwood Series in Computers and Their Applications). ISBN 0136514316,9780136514312.

HAMBLIN, C. L. Translation to and from Polish Notation. **The Computer Journal**, v. 5, n. 3, p. 210–213, 11 1962. ISSN 0010-4620. Disponível em: https://doi.org/10.1093/comjnl/5.3.210.

HAMMACK, R. **Book of Proof (Edition 3)**. 3. ed. Richard Hammack, 2020. ISBN 0989472132,9780989472135. Disponível em: https://www.people.vcu.edu/~rhammack/BookOfProof/.

JOST, N. G. **PyQt**. 2013. Site da Universidade Federal do Rio Grande do Sul. Disponível em: https://cta.if.ufrgs.br/projects/visuino/wiki/PyQt. Acesso em: 15 nov. 2020.

Ministério da Educação - Brasil. **Parecer CNE/CES nº 136/2012, aprovado em 8 de março de 2012 - Diretrizes Curriculares Nacionais para os cursos de graduação em Computação**. 2012. http://portal.mec.gov.br/index.php?option=com_docman&view=download&alias=11205-pces136-11-pdf&category_slug=julho-2012-pdf&Itemid=30192.

MOREIRA, A. *et al.* Interpretador de expressões lógicas. *In*: **Anais do 04° Encontro de Atividades Científicas**. Londrina: [*s.n.*], 2001.

NAPS, T. *et al.* Exploring the role of visualization and engagement in computer science education. **SIGCSE Bulletin**, v. 35, p. 131–152, 05 2003.

POSTAL, A. *et al.* Desenvolvimento de um programa interpretador e resolvedor para máquinas de estado finito. *In*: **II Congresso da Academia Trinacional de Ciências–C3N–Foz do Iguaçu–PR**. [*S.l.*: *s.n.*], 2007.

ROSSUM, G. V. *et al.* Computer programming for everybody. **Proposal to the Corporation for National Research initiatives**, Citeseer, 1999.

SARAIYA, P. *et al.* Effective features of algorithm visualizations. *In*: **Proceedings of the 35th SIGCSE technical symposium on Computer Science Education**. [*S.l.*: *s.n.*], 2004. p. 382–386.

SASAKI, K. **How to construct LL(1) grammar of arithmetic operations**. 2019. Disponível em: https://www.lewuathe.com/how-to-construct-grammar-of-arithmetic-operations.html. Acesso em: 25 ago. 2020.

SHAFFER, C. A. *et al.* Algorithm visualization: The state of the field. **ACM Transactions on Computing Education (TOCE)**, ACM New York, NY, USA, v. 10, n. 3, p. 1–22, 2010.

TöRLEY, G. Algorithm visualization in teaching practice. **Acta Didactica Napocensia**, v. 7, p. 17, 09 2014.

WEXELBLAT, R. L. **History of Programming Languages, Volume I**. [*S.l.*]: Academic Press, 1981. v. 1. ISBN 0127450408,9780127450407.

WOLF, C. E. **Infix to postfix conversion algorithm**. 2011. Site da Pace University. Disponível em: http://csis.pace.edu/~wolf/CS122/infix-postfix.html. Acesso em: 25 ago. 2020.