RELATÓRIO TÉCNICO: AVALIADOR DE LÓGICA PROPOSICIONAL E DE PREDICADOS

Data: 03 de dezembro de 2025

Assunto: documentação de decisões de design, implementação e análise técnica da codebase.

1 INTRODUÇÃO

Este relatório técnico descreve a arquitetura e as decisões de implementação adotadas no desenvolvimento do Sistema de Validação Lógica. O software foi projetado para avaliar argumentos na lógica proposicional, por meio de tabela-verdade, e na lógica de predicados, em domínios finitos, além de identificar formas de argumento comuns.

O foco principal do projeto foi a construção de uma ferramenta confiável, didática e com baixa sobrecarga de dependências, priorizando a compreensão dos algoritmos fundamentais da lógica computacional. O sistema processa fórmulas lógicas, gera tabelas-verdade completas e valida argumentos com base em premissas e conclusões fornecidas pelo usuário.

2 DECISÕES DE DESIGN E ARQUITETURA

2.1 Linguagem e bibliotecas (Python standard library)

Uma das diretrizes centrais do projeto foi a não utilização de bibliotecas externas, como numpy, pandas ou sympy. Optou-se pelo uso exclusivo da biblioteca padrão do Python, em especial os módulos re e itertools.

• Justificativa: essa escolha reduz a complexidade de configuração do ambiente, elimina problemas de compatibilidade de versões ("dependency hell") e garante que a lógica central, como a geração de combinações binárias, fosse implementada manualmente pelos desenvolvedores, reforçando o aprendizado.

• Impacto: maior controle sobre o fluxo de execução e confiabilidade no comportamento do código em diferentes máquinas. Além disso, facilita a execução do código em ambientes restritos, nos quais a instalação de pacotes externos pode não ser permitida.

2.2 Interface de linha de comando (CLI) versus interface gráfica (GUI)

O sistema foi desenvolvido inteiramente como uma aplicação de linha de comando (CLI).

• Motivação: a decisão baseou-se na experiência prévia da equipe na disciplina de Algoritmos e Técnicas de Programação, na qual a robustez lógica foi priorizada em detrimento de interfaces gráficas complexas. Interfaces gráficas introduzem complexidade de gerenciamento de eventos e layout que desviariam o foco do objetivo principal: a correção lógica.

• Benefício: a CLI permite uma iteração mais rápida no desenvolvimento das funções de avaliação lógica, evitando o overhead de manter estados de interface gráfica. Isso possibilitou concentrar o tempo de desenvolvimento na correção dos algoritmos de validação e na clareza da saída textual.

2.3 Paradigma procedural e modularização

Optou-se por não utilizar orientação a objetos de forma extensiva. O código está estruturado em módulos funcionais, como avaliador_predicados.py, tabela_verdade.py e parser_proposicional.py, em que cada arquivo contém funções puras ou procedimentos específicos.

• Organização: a modularização permite separar claramente as responsabilidades: o parser apenas processa texto, o avaliador apenas computa valores verdade e o módulo de menu apenas gerencia a entrada do usuário.

• Complexidade: evitar classes desnecessárias simplificou a estrutura de dados, mantendo o estado mutável no mínimo necessário. A lógica proposicional, em particular, mapeia-se naturalmente para funções matemáticas de entrada e saída, o que torna o paradigma funcional ou procedural adequado neste contexto.

3 DETALHES DE IMPLEMENTAÇÃO

3.1 Uso de expressões regulares (regex)

O uso de expressões regulares foi essencial para a detecção de padrões, validação de sintaxe e "transpilação" da notação lógica para código Python executável. O sistema converte operadores lógicos, como ->, ^ e v, para seus equivalentes em Python, como <=, and e or, antes da avaliação.

Em particular, no avaliador de predicados, utilizam-se expressões regulares para identificar quantificadores universais (∀) e existenciais (∃) e transformá-los em expressões lambda em Python.

Exemplo: a expressão regular pattern_forall = r"\(∀([a-z])\)" captura o padrão (∀x), em que x é qualquer letra minúscula, permitindo isolar a variável ligada ao quantificador para

processamento posterior. Em seguida, utiliza-se a função re.sub para transformar essa string em uma chamada de função do tipo forall(lambda x: ...).

3.2 Algoritmos e complexidade (O(2■) e comportamento exponencial)

Para a geração da tabela-verdade, o algoritmo produz todas as 2■ combinações possíveis de valores verdade para n variáveis proposicionais.

• Abordagem de força bruta: não foram implementadas otimizações heurísticas, como tableaux, método de Quine-McCluskey ou árvores de decisão binária. O sistema verifica todas as linhas da tabela-verdade.

• Complexidade: a complexidade é exponencial em relação ao número de variáveis (O(2■)). Para cada linha, a avaliação da fórmula apresenta custo aproximadamente linear em relação ao tamanho da expressão.

• Aceitação: dado o escopo acadêmico e o tamanho esperado das entradas (fórmulas com poucas variáveis, geralmente menos de dez), essa complexidade foi considerada aceitável. Para n igual a 10, por exemplo, obtêm-se 1024 linhas, o que é trivial para processadores modernos.

4 ESTRATÉGIA DE TESTES

Não foi utilizada nenhuma biblioteca de testes unitários padrão, como unittest ou pytest, a fim de manter o projeto sem dependências externas. Em vez disso, foi criado um módulo próprio, denominado testes_logica.py, contendo uma lista de dicionários com casos de teste. Cada caso descreve premissas, conclusão e resultado esperado.

• Funcionamento: o sistema itera sobre essa lista, executa as funções de avaliação e compara a saída obtida com o gabarito especificado.

• Vantagem: o modelo adotado oferece simplicidade e facilidade para adicionar novos casos de teste, sem a necessidade de aprender um framework específico.

• Cobertura: os testes abrangem casos clássicos, como Modus Ponens, Modus Tollens e falácias comuns, garantindo a integridade das regras lógicas fundamentais.

5 PONTOS DE MELHORIA E TRABALHOS FUTUROS

5.1 Evolução do parser: de regex para gramática livre de contexto

Diagnóstico: o uso de expressões regulares para parsing, embora prático, é teoricamente limitado. Expressões regulares reconhecem linguagens regulares, que não suportam adequadamente estruturas aninhadas recursivas arbitrárias, como ((A -> B) -> (C -> D)).

Casos complexos de aninhamento podem falhar ou exigir padrões excessivamente complexos.

Proposta de melhoria: implementar um parser descendente recursivo ou adotar o algoritmo shunting-yard, de Dijkstra. Isso inclui definir uma gramática formal em BNF (Backus–Naur Form), tokenizar a entrada (lexer) e construir uma árvore de sintaxe abstrata (AST).

Benefício: essa abordagem eliminaria erros de parsing em fórmulas mais complexas e permitiria mensagens de erro sintático mais precisas, tais como "erro na coluna 5: esperado ')'."

5.2 Otimização de algoritmos e mitigação da complexidade exponencial

Diagnóstico: o algoritmo atual de tabela-verdade apresenta complexidade O(2■). Se o usuário inserir vinte variáveis, o sistema tentará gerar cerca de um milhão de linhas, o que pode tornar a execução impraticável.

Proposta de melhoria: implementar métodos mais eficientes, como tableaux semânticos (árvores de refutação), que tentam provar a insatisfazibilidade da negação da conclusão, frequentemente fechando ramos da árvore de forma antecipada. Outra possibilidade é a adoção de algoritmos SAT, baseados em heurísticas modernas de resolvedores de satisfazibilidade, como o algoritmo DPLL.

Além disso, é possível explorar estratégias de curto-circuito: em certas situações, se uma premissa for falsa em uma linha específica, não é necessário avaliar completamente a conclusão para fins de verificação da validade do argumento.

5.3 Validação de entrada e tratamento de erros

Diagnóstico: o sistema atual assume que o usuário digitará fórmulas bem formadas. Erros de digitação, como P -> -> Q, podem causar falhas de execução no interpretador Python, gerando mensagens pouco claras.

Proposta de melhoria: criar uma função específica de validação sintática, responsável por verificar o balanceamento de parênteses e a correção dos operadores antes de qualquer tentativa de avaliação.

Além disso, recomenda-se envolver chamadas críticas em blocos try-except, capturando exceções como SyntaxError e retornando mensagens amigáveis, por exemplo: "expressão inválida, verifique os operadores".

5.4 Interface e usabilidade

Diagnóstico: a saída em texto puro pode ser difícil de interpretar em tabelas muito extensas.

Proposta de melhoria: utilizar códigos de escape ANSI para colorir, por exemplo, valores verdadeiros em verde e valores falsos em vermelho, facilitando a leitura visual. Também é possível, em futuro trabalho, adotar bibliotecas como curses ou textual para construir interfaces textuais mais ricas, com menus navegáveis pelo teclado.

5.5 Infraestrutura de testes

Diagnóstico: o script de testes atual é manual e, em alguns cenários, pode interromper a execução no primeiro erro ou apenas imprimir logs no terminal.

Proposta de melhoria: migrar para frameworks consolidados, como unittest ou pytest. Isso permitiria a execução automática de todos os testes, geração de relatórios detalhados de falhas e futura integração contínua (CI), caso o projeto seja hospedado em plataformas como o GitHub.

6 CONCLUSÃO

O projeto atendeu ao objetivo central de desenvolver um sistema automatizado de verificação de argumentos lógicos em Python, capaz de receber argumentos em lógica proposicional e de predicados e determinar se são válidos ou inválidos, fornecendo ao usuário uma saída estruturada. Foram implementados os componentes essenciais especificados: parser para fórmulas proposicionais, verificador por tabela-verdade, identificador de formas de argumento, parser para quantificadores e predicados em domínio finito e aplicação de regras básicas de inferência.

As decisões de design adotadas — uso exclusivo da biblioteca padrão, interface em linha de comando e modularização em arquivos especializados — mostraram-se adequadas ao propósito didático do trabalho. Elas favoreceram a transparência dos algoritmos de verificação, facilitaram a experimentação com diferentes casos de teste e resultaram em uma base de código relativamente simples de compreender e estender, ainda que com limitações assumidas, como a geração exaustiva de tabelas-verdade e o uso de expressões regulares no parsing.

Os pontos de melhoria mapeados, como a adoção de um parser sintático completo, a introdução de métodos mais eficientes de decisão (tableaux, resolução, resolvedores SAT), a ampliação da infraestrutura de testes automatizados e a evolução da interface, delineiam um caminho claro para transformar este protótipo acadêmico em uma ferramenta mais robusta de verificação formal. Dessa forma, o sistema cumpre seu papel tanto como solução funcional para o problema proposto quanto como plataforma de aprendizagem para temas avançados em lógica, compiladores e verificação automática de provas.