



Antônio Kássio Aragão da Cunha

Eduardo Bandeira Oliveira

Hamilton Isac Moreira da Silva

Mário Silva de Oliveira Junior

Matheus Barreto de Oliveira

Victória Yasmim Miranda Maciel

**Relatório de Análise Detalhada do Código Python (Final.py)**

**Manaus/AM  
2025**

## **Lista de Figuras**

Imagen 1: Bloco de código da tradução.....	3
Imagen 2: Tabela verdade do exemplo prático.....	5
Imagen 3: Identificação das formas no código.....	6
Imagen 4: Utilização da Instanciação em Domínio Finito.....	6
Imagen 5: Demonstração da Verificação e Explicação.....	7
Imagen 6: Exibição e interação com o usuário.....	8
Imagen 7: Uso do eval() no código.....	9

## 1. Introdução e Visão Geral do Projeto

O arquivo Final.py apresenta um sistema robusto e didático, escrito em Python, destinado à **validação de argumentos lógicos** em dois domínios distintos: a **Lógica Proposicional** e a **Lógica de Predicados** (Lógica de Primeira Ordem). O código é bem-estruturado, utilizando duas classes principais, MotorProposicional e MotorPredicados, para encapsular a lógica de cada domínio, e um conjunto de funções de interface para interação via linha de comando.

O objetivo principal do sistema é determinar se uma conclusão é uma consequência lógica de um conjunto de premissas, utilizando métodos clássicos da lógica formal:

1. **Lógica Proposicional:** Geração e análise de **Tabelas Verdade**.

2. **Lógica de Predicados:** **Instanciação em Domínio Finito**, que reduz o problema de predicados a um problema proposicional.

A seguir, detalhamos a análise funcional, a estrutura do código e, crucialmente, os pontos de atenção e sugestões de melhoria para aumentar a segurança e a robustez do sistema.

## 2. Análise Funcional Detalhada

### 2.1. Motor de Lógica Proposicional ( MotorProposicional )

Esta classe é a base de todo o sistema, pois o motor de predicados depende dela para a validação final.

#### 2.1.1. Tradução de Fórmulas ( traduzir )

O método traduzir (linhas 19-34) é responsável por converter a notação lógica simbólica, que é mais amigável ao usuário, para uma expressão Python que pode ser avaliada. Esta conversão é realizada através de um dicionário de mapeamento ( self.mapa\_ops ) e o uso de expressões regulares ( re.findall ) para tokenização.

```
19     def traduzir(self, formula):
20         texto = formula.strip().replace(" ", "") #Remove espaços
21         tokens = re.findall("<=>|>|[A-Z][a-zA-Z0-9_]*[V|v]\|\&\|\(\)\)", texto)
22         cod_py = [] #array que vai receber o a fórmula traduzida em python
23         vars_found = set() #define um objeto para receber as variáveis
24         for t in tokens: #para cada token lido
25             if t in self.mapa_ops: #se for um operador
26                 cod_py.append(self.mapa_ops[t])
27             elif t[0].isupper() and t != 'V' and t not in self.mapa_ops: #se não for
28                 cod_py.append(t)
29                 vars_found.add(t)
30             elif t.lower() == 'v': #caso extra pra reconhecer 'ou'
31                 cod_py.append(' or ')
32             else:
33                 cod_py.append(t)
34         return ''.join(cod_py), sorted(list(vars_found)) #retorna uma string formata
35
36 Tabnine | Edi | Test | Explain | Document
37 def identificar_forma(self, premissas, conclusao):
38     Idem ilustrado no código régua usado
39     p_txt = " ".join(premissas) # Junta tudo em uma string gigante
40     qtd_setas = p_txt.count(">>") #Conta a quantidade ed setas usadas (implica)
```

Imagen 1: Bloco de código da tradução

Fonte: Autoral

Símbolo Lógico	Operador Python	Tabela Verdade Equivalente	Justificativa
-> (Implicação)	<code>&lt;=</code>	$P \rightarrow Q \equiv \neg P \vee Q$	A implicação é falsa apenas quando o antecedente, $P$ , é verdadeiro e o consequente, $Q$ , é falso. Em Python, $P <= Q$ com booleanos (onde True é 1 e False é 0) retorna False apenas quando $P$ é True (1) e $Q$ é False (0), $1 \leq 0$ pois é falso. Em todos os outros casos, é <b>verdadeiro</b> . Esta é uma técnica inteligente para simular a tabela verdade da implicação usando operadores de comparação.
$\leftrightarrow$ (Bi implicação)	<code>==</code>	$P \leftrightarrow Q \equiv (P \rightarrow Q) \wedge (Q \rightarrow P)$	A bi-implicação é verdadeira se e somente se $Q$ e $P$ tiverem o mesmo valor de verdade, o que é diretamente modelado pelo operador de igualdade <code>==</code> em Python.
& , ^ (Conjunção)	<code>and</code>	$P \wedge Q$	Uso direto do operador lógico <b>and</b>
$\vee$ , v (Disjunção)	<code>or</code>	$P \vee Q$	Uso direto do operador lógico <b>or</b>
$\sim$ (Negação)	<code>not</code>	$\neg P$	Uso direto do operador lógico <b>not</b>

O método garante que as variáveis proposicionais (identificadas por letras maiúsculas, exceto 'V' que é um operador) sejam preservadas para posterior substituição.

### 2.1.2. Geração da Tabela Verdade ( gerar\_tabela )

Este método (linhas 57-107) implementa o algoritmo central para a validação

de argumentos proposicionais.

### Exemplo Prático: **Modus Ponens**

$$P \rightarrow Q, P \mid \neg Q$$

#### Premissas: 2

1.  $P \rightarrow Q$
2.  $P$

#### Conclusão:

1.  $Q$

*Sequência de passos do sistema:*

1. **Variáveis:** O sistema identifica a quantidade de premissas e transforma elas em linhas da tabela verdade, a quantidade de linhas é igual a 2 elevado a quantidade de premissas.

$$2^2 = 4$$

2. **Combinações:** Gera combinações de valores de verdade:  
 $(P=V, Q=V)$ ,  $(P=V, Q=F)$ ,  $(P=F, Q=V)$ ,  $(P=F, Q=F)$ .

#### 3. Avaliação (Linha 2):

$$P = \text{True}, Q = \text{False}$$

Contexto:  $P \rightarrow Q$ , P verdadeiro, Q falso

Premissa 1:  $P \rightarrow Q$

Premissa 2:  $P$

Conclusão:  $Q$  (inválido)

4. **Validação:** A linha 2 não é uma linha crítica, pois nem todas as premissas são verdadeiras. O sistema busca apenas as linhas onde **todas as premissas são Verdadeiras** (premissas conjuntas = V). Se em alguma dessas linhas a conclusão for Falsa, o argumento é declarado **INVÁLIDO**.

O método retorna um dicionário completo com a validade final, a lista de variáveis, as linhas da tabela e a forma do argumento identificada.

=====							
ARGUMENTO VÁLIDO!							
Método: Tabela verdade							
Forma: Modus Ponens							
Justificativa: Em todas as linhas onde as premissas são V, a conclusão também é V.							
Tabela Verdade:							
P	Q	$P \rightarrow Q$	P	$(P \rightarrow Q) \wedge P$	Q	válido?	
V	V	V	V	V	V	SIM	
V	F	F	V	F	F	-	
F	V	V	F	F	V	-	
F	F	V	F	F	F	-	

Imagen 2: Tabela verdade do exemplo prático

Fonte: Autoral

### 2.1.3. Identificação Heurística de Formas ( `identificar_forma` )

Este método (linhas 36-54) tenta classificar o argumento em uma das regras de inferência mais comuns (Modus Ponens, Modus Tollens, Silogismo Hipotético, Silogismo Disjuntivo). É importante notar que esta é uma abordagem **heurística** baseada em contagem de símbolos e número de premissas. Embora útil para fins didáticos, ela não é um parser lógico completo e pode falhar em identificar argumentos válidos que não se encaixam exatamente nos padrões de contagem definidos.

```

36     def identifica_forma(self, premissas, conclusao):
37         # Identifica o nome da regra usada
38         p_txt = ".join(premissas) # Junta tudo em uma string gigante
39         qtd_setas = p_txt.count("→") # Conta a quantidade ed setas usadas (implicações)
40
41         #Vai ler e se identificar o padrão dentro dos lens de setas vai dar o processo usado
42         if len(premissas) >= 3 and qtd_setas >= 2 and "→" not in conclusao:
43             return "Silogismo Hipotético + Modus Ponens"
44
45         if len(premissas) == 2 and qtd_setas == 2 and "→" in conclusao:
46             return "Silogismo Hipotético"
47
48         if len(premissas) == 2 and qtd_setas == 1:
49             if "¬" not in conclusao and conclusao in p_txt and "→" not in conclusao:
50                 return "Modus Ponens"
51
52         if len(premissas) == 2 and qtd_setas == 1 and "¬" in conclusao:
53             return "Modus Tollens"
54
55         if len(premissas) == 2 and ("V" in p_txt or "v" in p_txt) and "¬" in p_txt:
56             return "Silogismo Disjuntivo"
57
58     return "Argumento Dedutivo (Geral)" #caso não seja nada do que está acima
59

```

Imagen 3: Identificação das formas no código

Fonte: Autoral

### 2.2. Motor de Lógica de Predicados ( `MotorPredicados` )

A classe `MotorPredicados` (linhas 113-224) utiliza a técnica de **Instanciação em Domínio Finito** para verificar a validade de argumentos de primeira ordem.

```

113     class MotorPredicados:
114         def expandir_formula(self, formula, dominio):
115             # loop de Expansão
116             partes = []
117             for elemento in dominio:
118                 # Substitui a variável pelo elemento do domínio
119                 # Ex: H(X) vira H(a)
120                 nova_formula = corpo.replace(f"({{var}})", f"{{{elemento}}}")
121
122                 # Recursão: Se houver mais quantificadores dentro, expande de novo
123                 partes.append(self.expandir_formula(nova_formula, dominio))
124
125             # Junta as partes
126             if tipo_quant in ['A', 'V']:
127                 return f"({} & .join(partes))" # Universal = E
128             else:
129                 return f"({} v .join(partes))" # Existencial = ou
130

```

Imagen 4: Utilização da Instanciação em Domínio Finito  
Fonte: Autoral

### 2.2.1. Expansão de Fórmulas ( expandir\_formula )

O método `expandir_formula` (linhas 117-150) é o coração do motor de predicados. Ele converte a fórmula quantificada em uma expressão proposicional equivalente, assumindo um domínio finito.

**Exemplo de Expansão:** Considere o domínio .

$$D = \{a, b\}$$

- **Quantificador Universal ( $\forall$ )**: O código implementa isso como uma **Conjunção** (  $\wedge$  ) das instâncias.

Fórmula:

$$(\forall x)P(x)$$

Expansão:

$$P(a) \wedge P(b)$$

- **Quantificador Existencial ( $\exists$ )**: O código implementa isso como uma **Disjunção** (  $\vee$  ) das instâncias.

Fórmula:

$$(\exists x)P(x)$$

Expansão:

$$P(a) \vee P(b)$$

Além disso, o método converte os predicados instanciados (e.g.,  $P(a)$  ) em variáveis proposicionais únicas (e.g.,  $P_a$  ) para que o MotorProposicional possa tratá-los como variáveis atômicas.

### 2.2.2. Verificação e Explicação Pedagógica

O método `verificar` (linhas 182-224) coordena o processo:

1. Expande todas as premissas e a conclusão.
2. Chama `motor_prop.gerar_tabela` para validar a forma proposicional expandida.
3. Utiliza `gerar_explicacao` para construir um texto pedagógico que simula o  $(\forall x)P(x)$  raciocínio da prova, por exemplo: “Se  $P(a)=V$ ,  $P(b)=V$  (para satisfazer)  $Q(a)$ . Então deve ser  $V$ ”.

```

182     def verificar(self, dominio, premissas, conclusao):
183         if not dominio: return {'erro': 'Dominio vazio.'}
184
185         try:
186             # expandir todas as fórmulas para Lógica Proposicional
187             novas_premissas = [self.expandir_formula(p, dominio) for p in premissas]
188             nova_conclusao = self.expandir_formula(conclusao, dominio)
189
190             # usar o Motor Proposicional para validar a expansão
191             res_prop = self.motor_prop.gerar_tabela(novas_premissas, nova_conclusao)
192             if 'erro' in res_prop: return res_prop
193
194             # mostrar dados para exibição

```

Imagen 5: Demonstração da Verificação e Explicação  
Fonte: Autoral

### 3. Estrutura, Estilo e Interface de Usuário

#### 3.1. Modularidade e Estilo

O código adota uma estrutura orientada a objetos clara, com classes bem definidas para cada motor lógico. O uso de comentários e a separação em módulos (Proposicional, Predicados, Interface) demonstram uma preocupação com a legibilidade e manutenção do código. O uso de bibliotecas padrão como itertools e re é apropriado e eficiente para as tarefas de combinação e tokenização.

#### 3.2. Fluxo de Interação (Interface de Usuário)

As funções de interface ( limpar , ler\_int , exibir\_resultado , main ) criam um menu de linha de comando interativo (CLI) que guia o usuário:

1. O usuário escolhe entre Lógica Proposicional (Opção 1) ou Lógica de Predicados (Opção 2).
2. O sistema solicita o número de premissas e a conclusão.
3. Para Lógica de Predicados, é solicitado o **Domínio** (separado por vírgulas), que é essencial para a Instanciação em Domínio Finito.
4. O resultado é exibido de forma formatada, incluindo a Tabela Verdade completa (para proposicional) ou a regra aplicada e o texto de verificação (para predicados).

A função exibir\_resultado (linhas 238-285) é particularmente bem elaborada, formatando a Tabela Verdade de forma legível no console, com cabeçalhos e alinhamento dinâmico de colunas.

```
237     def exibir_resultado(dados):  
238         if 'erro' in dados:  
239             print("\n[ERRO]: " + dados['erro'])  
240             return  
241         print("\n" + "="*60)  
242         if dados['tipo'] == 'predicados':  
243             print("ARGUMENTO ('VÁLIDO' if dados['valido'] else 'INVÁLIDO')")  
244             print("Metodo: " + dados['metodo'])  
245             print("Regra aplicada: " + dados['regras_aplicadas'])  
246             print("\nVerificação:")  
247             print(dados['txt_verificacao'])  
248         if not dados['valido'] and dados['contra']: # Contradição  
249             print("\nContradição encontrada na enumeração")  
250         else: # Proposicional  
251             print("ARGUMENTO ('VÁLIDO' if dados['valido'] else 'INVÁLIDO')")  
252             print("Metodo: " + dados['metodo'])  
253             print("Forma: " + dados['forma'])  
254             if dados['valido']:  
255                 print("Justificativa: Es + todos os literais condizentes com V nos conclusões também é V")  
256             else:  
257                 print("Justificativa: Es + todos os literais condizentes com V nos conclusões também é V")
```

Imagen 6: Exibição e interação com o usuário  
Fonte: Autoral

#### 4. Pontos Críticos e Sugestões de Melhoria

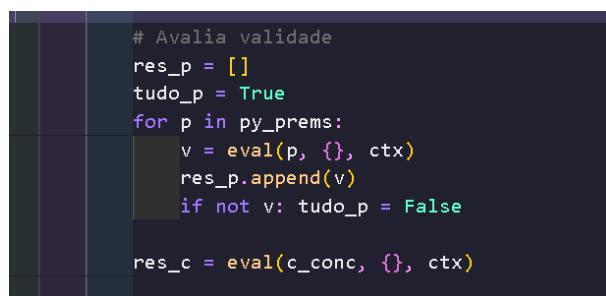
Embora o código seja funcional, existem pontos críticos de segurança e robustez que devem ser abordados.

##### 4.1. Vulnerabilidade de Segurança: O Uso Inseguro de eval()

O ponto mais crítico do código é o uso da função nativa do Python eval() nos métodos gerar\_tabela (linhas 84 e 88).

**eval() executa código Python arbitrário.** Quando uma string de entrada não é completamente sanitizada, ela pode ser injetada com comandos maliciosos, levando a uma vulnerabilidade de **Execução Remota de Código (RCE)**.

No contexto do código, a fórmula lógica traduzida para Python é passada diretamente para eval(). Embora o código tente limitar as variáveis e operadores, um usuário mal-intencionado poderia injetar código Python válido.



```
# Avalia validade
res_p = []
tudo_p = True
for p in py_prems:
    v = eval(p, {}, ctx)
    res_p.append(v)
    if not v: tudo_p = False

res_c = eval(c_conc, {}, ctx)
```

Imagen 7: Uso do eval() no código  
Fonte: Autoral

**Recomendação:** A função eval() deve ser **removida** e substituída por um mecanismo de avaliação seguro. As alternativas incluem:

1. **Construção de uma Árvore de Sintaxe Abstrata (AST):** O código deve ser reescrito para analisar a fórmula, construir uma árvore de operações lógicas e avaliá-la recursivamente, garantindo que apenas os nós de operação lógica ( and , or , not , <= , == ) sejam permitidos.

**2. Uso de Bibliotecas de Parsing Seguras:** Bibliotecas como `ast.literal_eval` (embora limitada) ou parsers dedicados como `ply` ou `lark` oferecem uma maneira mais segura e robusta de analisar e avaliar expressões.

## 4.2. Fragilidade da Expansão de Predicados

O método `expandir_formula` (linhas 117-150) utiliza substituições de string simples (`.replace()`) para instanciar os predicados.

**Problema:** A substituição é feita em toda a string, o que pode levar a erros se a variável quantificada for uma substring de outra variável ou constante.

**Exemplo:** Se a variável quantificada for `x`, e houver um predicado `P(x_grande)`, a substituição de `(x)` por `(a)` pode afetar o predicado maior de forma não intencional.

**Recomendação:** A substituição deve ser mais precisa, idealmente utilizando o *módulo re* para garantir que a substituição ocorra apenas na variável correta dentro do escopo do quantificador, ou após uma análise sintática que identifique claramente a variável de ligação.

## 4.3. Limitações na Identificação de Formas e Tratamento de Erros

O método `identificar_forma` é uma simplificação didática. Para um sistema de lógica mais completo, a identificação de regras de inferência deve ser baseada em um algoritmo de **dedução natural** ou **resolução**, que analise a estrutura lógica profunda do argumento.

**Recomendação:** Implementar a captura de exceções mais específicas, como:

`SyntaxError` ou uma exceção personalizada para fórmulas malformadas. `ValueError` para entradas de domínio inválidas.

## 5. Conclusão

O código `Final.py` é um projeto de programação que atinge seu objetivo de validar argumentos lógicos em dois níveis de complexidade. Ele serve como uma demonstração dos conceitos de Tabela Verdade e Instanciação em Domínio Finito.

No entanto, a dependência da função `eval()` introduz uma vulnerabilidade de segurança crítica que deve ser a prioridade máxima de correção. Ao substituir `eval()` por um parser seguro e refinar a lógica de expansão de predicados, o sistema pode ser transformado de uma ferramenta didática funcional em uma aplicação robusta e segura para análise lógica. O projeto, em sua essência, é uma base sólida para o

desenvolvimento de um toolkit de lógica formal mais avançada.