https://eventos.utfpr.edu.br//sicite/sicite2020

Um Analisador Sintático para um interpretador da Lógica Proposicional

A Parser for an interpreter of Propositional Logic

Jordano Vinicius Lahm
lahmvini@hotmail.com
Universidade Tecnológica Federal do Paraná, Toledo, Paraná, Brasil

Gustavo Henrique Paetzold
ghpaetzold@utfpr.edu.br
Universidade Tecnológica Federal do Paraná, Toledo, Paraná, Brasil

Recebido: 19 ago. 2020.
Aprovado: 01 out. 2020.

RESUMO

A lógica proposicional, assim como o estudo da matemática, engloba diversas áreas de aplicação, cuja finalidade é compor proposições com base na interpretação de sentenças do cotidiano. desde a aplicação em teoria de circuitos lógicos, até a introdução a operadores fundamentais dentro da matemática discreta, softwares e algoritmos específicos que auxiliam o processo de aprendizado e operação da lógica proposicional, são de grande interesse científico. para tanto torna-se proveitoso simplificar e operar tais expressões com objetivo de apoderar o aprendizado e a solução de problemas relacionados a proposições lógicas. por isso utilizando python em função da sua larga escala de aplicação e curva atenuada de aprendizado, juntamente com as referências da bibliografia clássica, foi implementado um analisador sintático de um interpretador genérico da lógica proposicional que tem como intuito ser utilizado como recurso didático. como resultado, foram conduzidos testes de validação para avaliar a corretude do algoritmo implementado.

PALAVRAS-CHAVE: Lógica proposicional. Interpretador. Algoritmos. Teoria da Computação.

ABSTRACT

Propositional logic, like the study of mathematics, encompasses several areas of application, whose practice is to compose propositions based on the interpretation of everyday sentences. From the application in logic circuit theory, to the introduction to operators within discrete mathematics, specific software and algorithms that assist the process of operation and operation of propositional logic, are of great scientific interest. To this end, it becomes beneficial to simplify and operate such expressions in order to empower the solution of problems related to logical propositions. Therefore, using Python due to its large scale of application and attenuated learning curve, together with the references of the classic bibliography, a syntactic analyzer of a generic interpreter of propositional logic was implemented, which is intended to be used as a teaching resource. As a result, validation tests were conducted to assess the correctness of the implemented algorithm.

KEYWORDS: Propositional logic. Interpreter. Algorithms. Computer Theory.

INTRODUÇÃO

A lógica proposicional é um formalismo matemático dentro da álgebra de proposições que trata de conectivos e sentenças relacionadas a atribuições lógicas. Em busca de novas técnicas didáticas e ferramentas de desenvolvimento de atividades dentro do estudo da lógica, foi proposto um interpretador como ferramenta auxiliar do estudo da análise, síntese e solução de expressões lógicas. Dentre as diversificadas formas de solucionar a problemática, um dos elementos da teoria da computação mais empregado em relação a forma gramatical de escrita e operação, são os interpretadores.

Com base em (AHO, pág.5), um compilador ou interpretador são compostos de fase, as quais podem ser resumidas em três etapas: análise léxica, análise sintática e análise semântica, cuja funcionalidade é analisar e sintetizar esse código enquanto estrutura e significado, conforme o algoritmo de alto nível. Um interpretador é um programa que executa instruções escritas em determinada linguagem. Quando aplicado a estrutura da lógica computacional, pode-se ter como resultado um estudo mais aprofundado de expressões complexas e bem definidas dentro da bibliografia atual. Nesse sentido, propõe-se desenvolver um analisador sintático implementado em Python utilizando métodos de validação e tratamento de expressões lógicas, com base no algoritmo de análise sintática denominado *Shuting Yard*.

LÓGICA PROPOSICIONAL

A lógica proposicional é uma linguagem formal responsável por expressar proposições com base em um conjunto de operadores e operandos que dão corpo a modelagem. Este sistema formal de proposições serve para a formulação de proposições lógicas que podem representar conjecturas, teoremas, argumentos, entre outros.

Tabela 1 – Tokens da Gramática.

Operador	*Símbolo*	*Tokens*
Disjunção	\/	DISJ
Conjunção	/\	CONJ
Condicional	>	COND
Operandos ou ID	([A-Z_] [A-Z0-1_] *')	VAR
Valores binários	0,1	BINARY
Parêntesis	(,)	RPAREN/LPAREN
Negação lógica	~	NOT

Fonte: Autoria própria.

Com base na Tabela 1, estão listados os elementos do interpretador, os conectivos lógicos (operador), os Tokens que são caracteres com significado dentro da gramática interpretada, e seus respectivos, símbolos. Tal simbologia compõe a lógica proposicional e representa a forma como o analisador sintático criado, possa interpretar corretamente.

AUTÔMATO

Para descrever um analisador sintático com base numa estrutura léxica predefinida, o qual valida a expressão traduzindo para uma estrutura *Tokenizada*, emprega-se autômatos para execução da leitura e classificação com base nas definições e conjuntos de regras. Um autômato é uma máquina de estados finita, dados por um estado inicial e final, que relaciona transições, estados e regras de operação. Matematicamente, define-se uma máquina de estados finita:

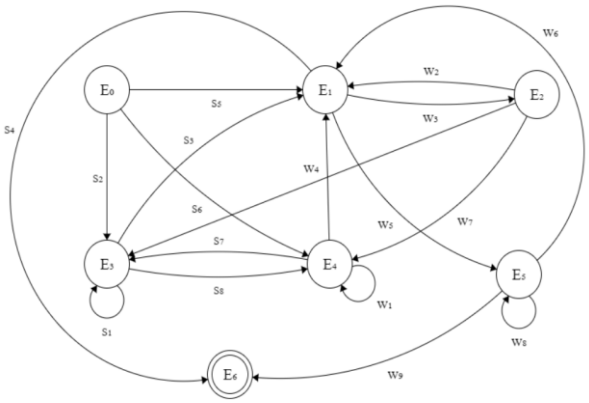
$$M = (I, O, S, f, g, \sigma) \quad (1)$$

sendo I (conjunto de símbolos de entrada), O (conjunto de símbolos de saída), S (conjunto finito de estados), f sendo a função do próximo estado ($f: SxI \rightarrow S$), g uma função de saída ($g: SxI \rightarrow O$) e σ o estado inicial do autômato ($\sigma \in S$).

O autômato de Pilha não-determinístico representado na Figura 1 foi o utilizado na descrição do analisador sintático implementado, nota-se que (E_0) é o estado inicial do autômato enquanto (E_6) é o estado final da máquina. Essa classe de autômato é denominada (ε-NFA) um autômato não determinístico com (ε) transições. Cada uma das transições indica uma regra para avançar ao próximo estado, de forma análoga, dentro da lógica proposicional, a pergunta que se faz para o processo de estruturação da máquina de estados após este operando ou símbolo, qual operador ou símbolo pode sequenciar?

Dentro da Teoria de Computação, a conversão para árvores quando atreladas a algoritmos de análise de precedência de operador, é comumente explorada. Conforme o modelo instituído de análise sintática com base na estrutura de Autômatos de Pilha, optou-se pela notação RPN como formato otimizado de avaliação. Perante, (SIPSER, pág.91), "...A máquina pode usar alfabetos diferentes para sua entrada e sua pilha", *i.e*, nessa estrutura de operação e transição que estabelece entre os estados determina o autômato operar com base na leitura do próximo operando/operador presentes na expressão de entrada.

Figura 1 – Autômato de Pilha.



Fonte: Autoria própria.

Com base na Figura 1, o autômato é composto pelos Tokens gerais da lógica proposicional presentes na Tabela 1. Cada estado dado por E, são elencados de forma a representar os tokens relacionados na gramática, construídos com base

na BNF (*Formalismo de Backus-Naur*) expressa em construções de interpretadores. Para isso são definidas transições oriundas das regras de derivação do dicionário formal da linguagem descrita na análise léxica, as quais descrevem como deve ser lida a expressão. Nesse contexto, foram divididas entre S (transições a esquerda) e W (transições a direita) chamadas de *Tuplas*, determinadas dentro da estrutura de autômatos.

Este autômato pode apresentar mais de um caminho ao receber um token de leitura, *i.e*, ao ler o símbolo da expressão de entrada as transições são sequenciadas até serem totalmente computadas, tendo mais de um único estado para seguir. No processo de validação, quando o autômato atinge um estado final a palavra é computada e aceita. Em caso contrário, a palavra computada é rejeitada ao finalizar o processo. Nessa situação, as *Tuplas* desempenham o papel fundamental de empilhar, escrever e desempilhar, ao ler os caracteres selecionados.

Tabela 2 – Transições do autômato.

Índice	Transição	Índice	Transição
S1	(~,λ,λ)	W1	((,X,λ)
S2	(~,λ,λ)	W2	(),λ,X)
S3	(Id; 0-1,λ,λ)	W3	(0-1,λ,λ)
S4	(ε,λ,ε)	W4	(Id; 0-1,λ,λ)
S5	(Id; 0-1,λ,λ)	W5	(0-1,λ,λ)
S6	((,X,λ)	W6	(),λ,X)
S7	((,X,λ)	W7	((,X,λ)
S8	(~,λ,λ)	W8	(~,λ,λ)
-	-	W9	(ε,λ,ε)

Fonte: autoria própria.

Portanto, ao receber um operando ou símbolo inicial, o autômato percorre o caminho conforme a expressão. Se a expressão conter todos os estados do autômato, e por logo concluir no estado final, a validação é feita. Nesta Tabela 2, encontramos as transições dadas ao encaminhamento do autômato conforme (AHO, pag.52) em ler o primeiro elemento da tripla, e operar e agir conforme os rótulos (ε) e (λ).

ANÁLISE SINTÁTICA

A análise sintática ou análise gramatical é a segunda fase do interpretador onde é realizada a verificação de frases gramaticais segundo uma gramatica formal, no caso baseada na estrutura da lógica proposicional. Com base em (GRUNE, pág. 79), “analisar uma *string* de acordo com uma gramática significa reconstruir a árvore de produção...”, nesse sentido, busca-se aplicar algoritmos de *Parsing* que utilizam de estruturas de dados para trabalhar com sequencias de Tokens produzidas por um analisador lexical. Conforme (AHO, pág.72), no contexto de construção de um analisador sintático, se é projetado um algoritmo de *parsing* com base em métodos automatizados já explorados em produção de árvores.

Existem variados tipos de algoritmos de análise sintática na sua maioria obedecendo a grandes duas classificações: *Top-Down* (cima para baixo) ou *Bottom-*

up (Baixo para cima) como relatados em (AHO, pág.72). Nesse interim, utilizou-se de um algoritmo de *Parsing* taxado também como *Parsing/RPN calculator algorithm* o qual utiliza da técnica *Bottom-Up* para detalhamento e operação sintática. Esse algoritmo é denominado *Shuting Yard* conhecido como "*pátio de manobra*" e correlaciona produção de árvores como saídas na forma de Notação Polonesa Reversa (RPN), facilitando o desenvolvimento do tratamento semântico.

SHUTING YARD

Shunting Yard é um algoritmo avançado de pesquisa, o qual utiliza a técnica de análise de expressões matemáticas em notação infixa. Introduzido em (DIJKSTRA, 1961), o algoritmo converte uma entrada infixa numa saída pós-fixa. Quando aplicado para avaliar expressões e convertê-las na forma pós-fixa, pode estruturar árvores de sintaxe com base na predição do modelo. Com base no pseudocódigo presente na Figura 2, o algoritmo aplica uma pilha de armazenamento de operadores. O autômato de pilha foi descrito internamente ao algoritmo sendo estruturado na forma de dicionários da linguagem Python que relaciona os estados e retornos de operação da pilha, na forma de chaves. Nesse contexto, utilizando a mesma estrutura, foi possível descrever as propriedades de associatividade e precedência dos operadores, tal como a sintaxe formal de validação.

Figura 2 – Pseudocódigo

```
PARA CADA token na entrada FAÇA
  SE token é número ENTÃO
    insere token no topo da pilha de saída
  SE token é um operador o1 ENTÃO
    ENQUANTO existe operador o2 no topo da pilha de
             operadores
             E precedência de o1 ≤ precedência de o2 FAÇA
      remove o2 do topo da pilha de operadores
      insere o2 no topo da pilha de saída
    insere token o1 no topo da pilha de operadores
  SE token é parêntese esquerdo "(" ENTÃO
    insere token no topo da pilha de operadores
  SE token é parêntese direito ")" ENTÃO
    ENQUANTO topo da pilha de operadores não é um
             parêntese esquerdo FAÇA
      remove operador do topo da pilha de operadores
      insere operador no topo da pilha de saída
    SE pilha de operadores está vazia ENTÃO
      ERRO
    remove parêntese esquerdo da pilha de operadores
ENQUANTO pilha de operadores não vazia FAÇA
  remove operador do topo da pilha de operadores
  SE operador é parêntese ENTÃO
    ERRO
  insere operador no topo da pilha de saída
```

Fonte: http://mathcenter.oxford.emory.edu/site/cs171/shuntingYardAlgorithm/

RESULTADOS

Com base na análise léxica predefinida foi implementado o algoritmo *Shuting yard*, aplicado nas etapas descritas acima. Dessa forma, na Tabela 3 são exemplificadas entradas e saídas do analisador. De maneira concisa, ao incrementar a análise ascendente que interpreta a gramática de precedência é possível a conversão conforme a validação percorrida de trás para frente. Como incremento da análise, foi proposto balanceamento de parênteses para garantir o funcionamento adequado da pilha.

Tabela 3– Execução de expressões

Expressão Lógica	Saída do Analisador
(Q ⋁ ~ F) /\~ (Q ⋁ ~ F)	Q F NOT DISJ Q F NOT DISJ NOT CONJ
(~ (Q ⋁ F) /\ (~ (Q ⋁ F) ⋁ R))	Q F DISJ NOT Q F DISJ NOT R DISJ CONJ
~ (~ R /\ (Q ⋁ P))	R NOT Q P DISJ CONJ NOT
(~ (Q ⋁ F) /\ (~ (Q ⋁ F) R))	ERRO

Fonte: Autoria Própria.

Visto a expressão lógica dada na Tabela 3, esta é lida pelo analisador léxico e retorna ao analisador sintático uma lista de *Tokens*. O *Shunting Yard* recebe duas listas dividas em tipo e valor, as quais são relacionadas com as dependências da Tabela 1. Ao efetuar o algoritmo, a saída é dada por uma lista na forma RPN, que integra os tokens dentro do autômato de pilha, e instancia as classes do autômato para análise de precedência. Ao passo que é lida a expressão, os operadores são empilhados numa pilha de memória e os elementos são comparados com os símbolos instanciados, neste caso retornados da análise léxica.

Neste contexto tem-se a saída em árvores binárias para complementar a solução como arquivos de teste. O gerador de expressões para teste trabalha com a leitura de um algoritmo randômico de expressões com símbolos e operadores lógicos. O algoritmo de *Parsing* contendo o script geral é instanciado, chamando o arquivo de entrada com várias expressões, comparando as entradas do arquivo de leitura com as entradas *Tokenizadas* na forma RPN, permitindo a alocação dos resultados num arquivo de saída, dado em RPN e em árvores para encaminhamento.

CONCLUSÃO

O projeto fundamentou-se em implementar uma ferramenta para auxiliar alunos no aprendizado de lógica proposicional utilizando a linguagem Python. Buscou implementar métodos já conhecidos de análise sintática, com o objetivo final de construir um interpretador para tratar várias variáveis para cálculo de expressões lógicas válidas. O analisador sintático desenvolvido para o interpretador fomenta a utilização de ferramentas da teoria da computação para resolver problemas que englobam gramáticas intrínsecas. Nesse sentido, aplicar novas técnicas de análise sintática para melhor desempenho do interpretador é

sugerido como melhoria, tal como otimização destes para solução de problemas do gênero.

Para os geradores de teste foram criados dois arquivos para verificação tanto na forma RPN em listas validadas, quanto a estruturação da saída em árvores binárias. Ao gerar as sentenças randômicas, o gerador produz uma validação rápida para a determinada saída gerada. Em caso de o resultado da validação ser idêntico ao retornado pelo analisador sintático, o processo é validado num terceiro arquivo que contém para cada expressão uma classificação correta (*True*) ou incorreta (*False*).

Com a geração de um número elevado de testes, notou-se a viabilidade e integridade dos resultados quando comparados com a transcrição clara das regras formais da lógica proposicional. Baseado nesses resultados, almeja-se o desenvolvimento engajado da análise semântica, tal como desenvolver uma interface gráfica que permita o usuário interagir mais facilmente com a ferramenta.

AGRADECIMENTOS

Agradecimento a UTFPR instituição que possibilitou o desenvolvimento da pesquisa, ao meu orientador e amigos do grupo de pesquisa, envolvidos no projeto.

REFERÊNCIAS

AHO, Alfred V.; SETHI, Ravi; ULLMAN, Jeffrey D. **Compiladores: Princıpios, técnicas e ferramentas.** LTC, Rio de Janeiro, Brasil, 1995.

DIJKSTRA, E.W. (1961). **Algol 60 translation: An algol 60 translator for the x1 and making a translator for algol 60.** Stichting Mathematisch Centrum. Rekenafdeling. Stichting Mathematisch Centrum.

GRUNE, Dick; JACOBS, Ceriel. **Parsing Techniques–A Practical Guide**. 1990. VU University. Amsterdam.

SIPSER, Michael. **Introdução à teoria da computação**. Thomson Learning, p.91-95, 2007.

VAN ROSSUM, Guido et al**. Computer programming for everybody**. Proposal to the Corporation for National Research Initiatives, 1999.

WOLF, E. Carol. **O algoritmo Shunting Yard**. Disponível em: http://mathcenter.oxford.emory.edu/site/cs171/shuntingYardAlgorithm. Acesso em: 01 de Nov.2020.