UNIVERSIDADE FEDERAL DO TOCANTINS
CÂMPUS UNIVERSITÁRIO DE PALMAS
CURSO DE CIÊNCIA DA COMPUTAÇÃO

PATRYCK HENRYCK MOREIRA SILVA

DELIMITAÇÃO DA EXPRESSIVIDADE DO OPERADOR PUSHDOWN: RESTRIÇÕES ESTRUTURAIS PARA A CONSTRUÇÃO DE AUTÔMATOS DE PILHA DETERMINÍSTICOS

PALMAS (TO)

2025

Patryck Henryck Moreira Silva

Delimitação da expressividade do operador pushdown: restrições estruturais para a construção de autômatos de pilha determinísticos

Trabalho de Conclusão de Curso II apresentado à Universidade Federal do Tocantins para obtenção do título de Bacharel em Ciência da Computação.

Orientador:
Dr. Alexandre Tadeu Rossini da Silva
Coorientador: Dr. Tanilson Dias dos Santos

PALMAS (TO)

2025

Dados Internacionais de Catalogação na Publicação (CIP)
Sistema de Bibliotecas da Universidade Federal do Tocantins

S586d Silva, Patryck Henryck Moreira.

Delimitação da expressividade do operador pushdown: restrições estruturais para a construção de autômatos de pilha determinísticos. / Patryck Henryck Moreira Silva. – Palmas, TO, 2025.

74 f.

Monografia Graduação - Universidade Federal do Tocantins – Câmpus Universitário de Palmas - Curso de Ciências da Computação, 2025.

Orientador: Alexandre Tadeu Rossini da Silva

Coorientador: Tanilson Dias dos Santos

1. Linguagens livres de contexto. 2. Operador pushdown. 3. Autômatos de pilha. 4. Expressões formais. I. Título

CDD 004

Elaborado pelo sistema de geração automática de ficha catalográfica da UFT com os dados fornecidos pelo(a) autor(a).

Patryck Henryck Moreira Silva

Delimitação da expressividade do operador pushdown: restrições estruturais para a construção de autômatos de pilha determinísticos

Trabalho de Conclusão de Curso II apresentado à UFT – Universidade Federal do Tocantins – Câmpus Universitário de Palmas, Curso de Ciência da Computação foi avaliado para a obtenção do título de Bacharel e aprovada em sua forma final pelo Orientador e pela Banca Examinadora.

Data de aprovação: 11 / 12 / 2025

Banca Examinadora:

Prof. Dr. Alexandre Tadeu Rossini da Silva

Prof. Dr. Tanilson Dias dos Santos

Prof. Dr. Warley Gramacho da Silva

Prof. Dr. Janio Carlos Nascimento Silva / IFTO

A Deus, pela vida, e aos meus
pais, pelo amor incondicional e
por acreditarem em mim mesmo
quando eu duvidava.

AGRADECIMENTOS

Agradeço a Deus, por me conceder o dom da vida. Aos familiares e aos meus pais, por sempre me ajudarem. Aos meus professores e aos meus pais, por sempre me ensinarem. Aos meus amigos e aos meus pais, por sempre me motivarem. E, principalmente, aos meus pais, por serem a constante em todas essas etapas.

RESUMO

As expressões livres de contexto estendidas pelo operador de empilhamento (PDOp - *Pushdown Operator*) constituem um formalismo denotacional capaz de representar todo o conjunto das linguagens livres de contexto. Entretanto, o algoritmo de conversão dessas expressões para autômatos de pilha produz, em geral, modelos não determinísticos, cuja execução não é eficiente em máquinas determinísticas e pode apresentar comportamentos indesejáveis, como ciclos infinitos decorrentes da presença da palavra vazia. Este trabalho investiga restrições formais e estruturais que permitam a construção de autômatos de pilha determinísticos (APDs) a partir de expressões com PDOp, com foco inicial no caso base do operador. A metodologia adota uma abordagem teórico-formal aliada a experimentação computacional, envolvendo a análise de algoritmos existentes, identificação de fontes de não determinismo, definição de critérios estruturais para eliminação de recursões improdutivas e proposição de uma nova arquitetura algorítmica. Como contribuição principal, apresenta-se um processo de conversão linearizado e um *pipeline* heurístico para redução de não determinismo estrutural, visando tornar o PDOp aplicável em ambientes computacionais determinísticos. Os resultados obtidos indicam que a abordagem proposta constitui um passo relevante para o desenvolvimento de um compilador de expressões com PDOp capaz de gerar APDs corretos e eficientes.

Palavra-chave: Linguagens livres de contexto. Operador pushdown. Autômatos de pilha. Expressões formais.

ABSTRACT

Context-free expressions extended with the Pushdown Operator (PDOp) form a denotational formalism capable of representing the entire class of context-free languages. However, the existing algorithm for converting such expressions into pushdown automata typically yields nondeterministic models whose execution is inefficient on deterministic machines and may exhibit undesirable behaviors, such as infinite computational cycles triggered by the presence of the empty word. This work investigates formal and structural restrictions that enable the construction of deterministic pushdown automata (DPDAs) from PDOp expressions, with an initial focus on the operator's base case. The methodology combines a theoretical–formal approach with controlled computational experimentation, involving the analysis of existing algorithms, the identification of sources of nondeterminism, the definition of structural criteria for eliminating unproductive recursions, and the proposal of a new algorithmic architecture. As its main contribution, the study introduces a linearized conversion process and a heuristic pipeline for reducing structural nondeterminism, thereby making the PDOp more suitable for deterministic computational environments. The results indicate that the proposed approach represents a significant step toward the development of a PDOp-based expression compiler capable of generating correct and efficient DPDAs.

Keywords: Context-free languages. Pushdown operator. Pushdown automata. Formal expressions.

LISTA DE FIGURAS

LISTA DE TABELAS

SUMÁRIO

1 INTRODUÇÃO

As expressões regulares desempenham papel importante na modelagem de padrões formais, sendo amplamente utilizadas em ferramentas de análise léxica, linguagens de programação e sistemas de verificação formal. No entanto, por definição, elas são limitadas à representação de linguagens regulares. Nesse contexto, o Operador de Empilhamento (*Pushdown Operator* – PDOp), proposto por Silva, Oliveira e Santos (2025), surge como uma proposta inovadora de extensão do formalismo clássico das expressões regulares, permitindo que linguagens livres de contexto também possam ser expressas por formalismo denotacional.

Apesar desse avanço, a conversão de expressões contendo o PDOp em Autômatos de Pilha Não Determinísticos (APNDs) permanece associada a desafios substanciais. A ausência de mecanismos formais de controle sobre o não determinismo pode conduzir à exploração indefinida de ramificações computacionais, particularmente na presença de transições ε ou de transições não redutoras. No âmbito teórico, tal comportamento pode produzir situações análogas ao problema da parada (TURING, 1936a), caracterizadas pela possibilidade de que a computação não termine para cadeias que não pertencem à linguagem.

Cumpre assinalar, entretanto, que esse fenômeno decorre do modelo abstrato de não determinismo inerente aos APNDs. Em implementações concretas atuais, necessariamente determinísticas, a exploração das ramificações ocorre de modo sequencial e finitamente controlada. Nesses casos, a não terminação não se manifesta como expansão simultânea de múltiplos ramos de computação, mas como a ocorrência de ciclos específicos no espaço de estados do autômato implementado, resultantes de definições inadequadas de transições ou de loops envolvendo movimentos que não consomem entrada. Essa distinção entre o comportamento teórico e o comportamento observado em implementações reais evidencia a necessidade de mecanismos que conciliem a expressividade do formalismo com garantias práticas de terminação e eficiência.

Assim, surge a necessidade de investigar os limites e as condições que permitem a construção segura e eficiente dos autômatos de pilha gerados a partir de expressões que empregam o PDOp, de modo a garantir que o processo de conversão produza estruturas determinísticas, livres de ciclos não redutores e adequadas para implementação prática. Esse esforço é fundamental para que o PDOp se consolide como uma ferramenta confiável para a modelagem operacional de linguagens livres de contexto.

1.1 Objetivo geral

Este trabalho tem como objetivo investigar restrições formais e estruturais aplicáveis às expressões regulares clássicas estendidas com PDOp, denominadas expressões livres de contexto (SILVA; OLIVEIRA; SANTOS, 2025), de modo a garantir que sua interpretação operacional resulte na construção de Autômatos de Pilha Determinísticos (APDs). A escolha por estabelecer condições que assegurem determinismo não é arbitrária: diferentemente dos autômatos não determinísticos, os APDs apresentam propriedades de execução significativamente mais controladas, evitando a explosão de ramificações computacionais e eliminando ambiguidades inerentes ao não determinismo.

Do ponto de vista computacional, os APDs oferecem vantagens decisivas, especialmente no que diz respeito à eficiência. A existência de um único curso de computação para cada cadeia permite que o reconhecimento da linguagem seja realizado de maneira previsível e sem a necessidade de explorar múltiplas alternativas de forma paralela ou sequencial. Assim, a imposição de restrições sobre o formalismo de expressões com PDOp estabelece fundamentos concretos para sua implementação prática, garantindo que o processo de reconhecimento seja finito, eficiente e compatível com máquinas determinísticas reais.

1.2 Objetivos específicos

Com base no objetivo geral, esta pesquisa se desdobra em uma série de objetivos específicos que visam aprofundar a compreensão teórica e prática do operador PDOp. A seguir, são listados os objetivos específicos que nortearam o desenvolvimento deste trabalho.

- Propor uma solução teórica para o problema de parada em autômatos de pilha não determinísticos gerados a partir de expressões livres de contexto com o PDOp.
- Categorizar as formas de ambiguidade e não determinismo geradas durante a tradução de diferentes padrões de uso do operador PDOp.
- Definir um subconjunto restrito de expressões livres de contexto, baseado em critérios formais, que garanta a geração de autômatos de pilha determinísticos.
- Propor regras sintáticas ou construtivas para orientar a escrita de expressões livres de contexto que levem à construção de autômatos de pilha determinísticos.
- Validar as restrições propostas por meio de experimentação com a geração e execução de autômatos de pilha com expressões livres de contexto reais.

A definição dos objetivos desta pesquisa estabelece as bases para uma investigação teórica e prática sobre os limites e possibilidades do uso do operador PDOp na construção

de autômatos de pilha determinísticos. No entanto, para que tais objetivos sejam perseguidos de forma objetiva e factível, faz-se necessário delimitar com precisão o escopo do problema abordado.

1.3 Delimitação do problema

Esta pesquisa concentra-se na análise formal das expressões livres de contexto definidas pelo PDOp, com o propósito de caracterizar condições estruturais sob as quais tais expressões admitem uma interpretação operacional determinística. Em particular, busca-se identificar restrições sintáticas e semânticas que garantam que a tradução dessas expressões produza unicamente APDs. A investigação contempla tanto a formulação de critérios que eliminem fontes potenciais de ambiguidade e de não determinismo quanto a validação, por meio de instâncias representativas, da corretude e do comportamento previsível dos autômatos derivados. Dessa forma, delimita-se o escopo teórico no qual o formalismo estendido pode ser interpretado de maneira determinística e coerente com os modelos clássicos de reconhecimento de linguagens livres de contexto.

Para assegurar foco e precisão analítica, algumas delimitações foram explicitamente estabelecidas. Em particular, o estudo restringe-se ao caso base do PDOp, isto é, àquelas instâncias do operador que não envolvem múltiplas chamadas múltiplas ou chamadas alternadas (SILVA; OLIVEIRA; SANTOS, 2025). Os casos de chamadas múltiplas e chamadas alternadas do PDOp não são investigadas nesta pesquisa, permanecendo como desdobramentos potenciais para trabalhos futuros. Além disso, estão excluídos do escopo deste estudo:

- A definição ou implementação de linguagens de programação completas fundamentadas no formalismo proposto;
- A comparação empírica com técnicas tradicionais de análise sintática, tais como analisadores LR, LL ou variantes generalizadas;
- A análise rigorosa de complexidade temporal ou de desempenho computacional dos autômatos gerados.

Dessa forma, esta investigação restringe-se ao exame teórico-conceitual das propriedades estruturais do PDOp e à verificação formal de sua compatibilidade com modelos determinísticos de autômatos de pilha. Não se pretende extrapolar para considerações de eficiência operacional, para o desenvolvimento de ferramentas completas de processamento de linguagens ou para estudos comparativos entre diferentes paradigmas de reconhecimento. Essa delimitação permite concentrar a análise nos aspectos fundamentais que definem o domínio do problema investigado, evitando generalizações que excedam o escopo conceitual estabelecido.

1.4 Justificativa

A presente pesquisa justifica-se sob múltiplos aspectos que evidenciam sua relevância tanto do ponto de vista teórico quanto do ponto de vista aplicado. Contribui para o avanço da Teoria de Linguagens Formais e Autômatos ao investigar condições formais sob as quais expressões livres de contexto com PDOp podem ser convertidas, de modo rigoroso, em APDs. Esta investigação aborda questões centrais relacionadas à decidibilidade, à eliminação de ambiguidade, controle de não determinismo e poder computacional dos modelos formais envolvidos, temas de importância reconhecida dentro da Teoria da Computação.

Além disso, esta pesquisa reforça o potencial aplicado do formalismo, uma vez que modelos determinísticos de reconhecimento são particularmente desejáveis em contextos como desenvolvimento de compiladores, construção de validadores sintáticos, ferramentas de análise estática e sistemas de processamento de linguagem natural. Nesses ambientes, propriedades como previsibilidade operacional, ausência de múltiplos cursos de computação e comportamento determinístico são essenciais para garantir robustez, desempenho e confiabilidade.

A abordagem proposta insere-se na continuidade natural dos trabalhos de Oliveira (2024) e Silva, Oliveira e Santos (2025), que introduziram o operador PDOp, demonstraram sua equivalência ao conjunto das linguagens livres de contexto e apresentaram um algoritmo de conversão de expressões com PDOp em APND. A presente pesquisa avança essa linha ao examinar as condições sob as quais tal conversão pode ser restringida ao âmbito determinístico.

Quanto à viabilidade, trata-se de um estudo essencialmente conceitual e teórico, cuja validação computacional limita-se à geração e simulação de autômatos derivados. Os experimentos requeridos podem ser conduzidos por meio de ferramentas de código aberto, bibliotecas formais e linguagens como Python, não demandando infraestrutura computacional além daquela já disponível na instituição, o que reforça a factibilidade prática do projeto executado.

1.5 Procedimento metodológico da pesquisa

Esta pesquisa adota uma abordagem teórico-dedutiva, baseada na análise formal do PDOp, formulação de critérios estruturais e validação por meio da construção e simulação de autômatos representativos. As etapas metodológicas desenvolvem-se de modo incremental ao longo dos capítulos 3 e 4, onde são propostas e analisadas as regras construtivas e os mecanismos de determinação.

1.6 Metodologia de Pesquisa

Para atingir os objetivos propostos, este trabalho adota uma abordagem metodológica **teórica e dedutiva**. A pesquisa classifica-se como:

- **Exploratória:** ao investigar a extensão de operadores formais em um campo ainda pouco consolidado na literatura, buscando expandir a capacidade expressiva de expressões regulares.
- **Descritiva:** ao analisar e caracterizar formalmente as propriedades estruturais e sintáticas das expressões com PDOp, identificando as restrições necessárias para o determinismo.

O procedimento técnico baseou-se no método formal, partindo de definições matemáticas e construtos teóricos (gramáticas e autômatos) para desenvolver raciocínios lógicos, propor algoritmos e demonstrar a viabilidade da conversão determinística. A validação dos resultados foi conduzida através de provas de conceito e experimentação computacional controlada.

1.7 Organização do trabalho

Este trabalho está estruturado em quatro capítulos, organizados de forma a conduzir o leitor desde a contextualização inicial até os resultados parciais e as perspectivas futuras da pesquisa.

O Capítulo 1 apresenta a introdução ao tema, incluindo a definição dos objetivos, a delimitação do problema, a justificativa da pesquisa e a descrição da organização do trabalho.

Em seguida, o Capítulo 2 aborda a fundamentação teórica, apresentando os conceitos essenciais para a compreensão do estudo, como linguagens formais, autômatos de pilha, expressões livres de contexto e o PDOp.

No Capítulo 3, são descritos os procedimentos metodológicos adotados, detalhando a classificação da pesquisa e o fluxo metodológico.

Por fim, o Capítulo 4 apresenta as perspectivas e direções para o desenvolvimento futuro da pesquisa, incluindo o cronograma da pesquisa.

2 FUNDAMENTAÇÃO TEÓRICA

Neste capítulo, são apresentados os conceitos fundamentais da Teoria das Linguagens Formais que embasaram o desenvolvimento deste trabalho. São abordadas as definições essenciais sobre alfabetos, palavras e linguagens, bem como a Hierarquia de Chomsky e os principais formalismos de representação: axiomático, operacional e denotacional.

2.1 Linguagens formais

As linguagens formais constituem um ramo da ciência da computação e da linguística teórica, dedicado ao estudo rigoroso das linguagens estruturadas segundo regras bem definidas. Ao contrário das línguas naturais[1], que são ambíguas pois uma mesma sentença pode admitir múltiplas interpretações dependendo do contexto, as linguagens formais são criadas com o propósito de serem precisas e terem regras de formação bem definidas, de modo a permitir sua manipulação por máquinas. Em muitos contextos, busca-se também reduzir ou eliminar ambiguidades, embora nem toda linguagem formal seja necessariamente não ambígua. Esse campo surgiu da confluência entre matemática, lógica e linguística, e tem papel central em diversas áreas, como compiladores, interpretação de linguagens de programação, verificação de software, teoria da computação, criptografia, entre outras.

Tendo expostas essas características gerais, são apresentados conceitos fundamentais que constituem a base formal das linguagens. O primeiro deles é o alfabeto, definido como um conjunto finito e não vazio de símbolos (HOPCROFT; ULLMAN, 1979), geralmente denotado por Σ.

Exemplos de alfabetos:

1. $\Sigma = \{0, 1\}$, alfabeto binário.
2. $\Sigma = \{a, b, ..., z\}$, conjunto de todas letras minúsculas.
3. $\Sigma = \{0, 1, ..., 9\}$, conjunto de dígitos numéricos.

Símbolo é qualquer elemento pertencente a um alfabeto Σ e é utilizado para formar cadeias e linguagens (HOPCROFT; ULLMAN, 1979) (SIPSER, 2006a).

Uma sequência finita de símbolos pertencentes a um mesmo alfabeto é denominada sentença, cadeia, *string* ou palavra (MENEZES, 2000). Como exemplo, a palavra 10001 pertence ao alfabeto $\Sigma = \{0,1\}$. O número de símbolos que compõe uma palavra, isto é,

[1] Línguas naturais são sistemas de comunicação usados por comunidades humanas, como o português e o inglês, caracterizados por evolução histórica, variação sociocultural e ausência de regras completamente formais.

seu tamanho ou comprimento, é representado por $|w|$, onde w denota a palavra, que neste exemplo é 10001.

Caso uma palavra tenha zero ocorrência de simbolos, esta palavra é chamada de palavra vazia e é denotada por ε ou λ (HOPCROFT; ULLMAN, 1979). Neste trabalho foi adotado o símbolo ε para denotar a palavra vazia, que possui comprimento zero, ou seja, $|\varepsilon| = 0$.

O conjunto de todas as palavras possíveis formadas a partir de um alfabeto Σ é denotado por Σ^*. Assim, uma linguagem formal L é definida como um subconjunto do conjunto de Σ^*, isto é, $L \subseteq \Sigma^*$. Desse modo, uma linguagem é um conjunto (finito ou infinito) de palavras sobre um determinado alfabeto Σ.

Para manipular e combinar linguagens, é fundamental compreender as operações básicas que podem ser aplicadas a esses conjuntos. As operações com linguagens formais são ferramentas importantes tanto na teoria quanto em aplicações práticas, como no projeto de linguagens de programação, compiladores e sistemas de verificação. Possíveis operações com linguagens (HRUŠA, 2021):

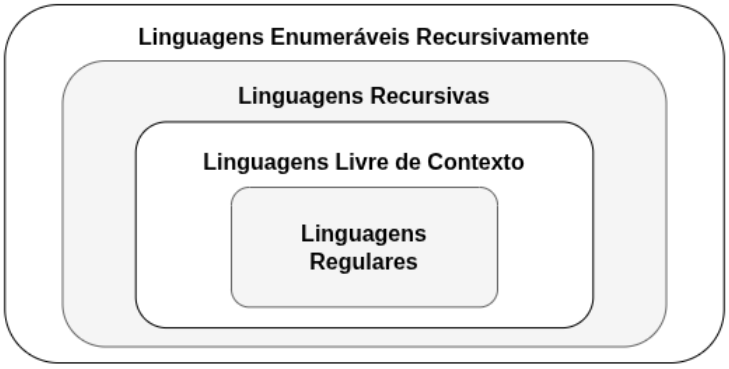
- Operações de conjunto: união, intersecção, subtração.
- Concatenação: L_1L_2 ou $(L_1.L_2) = \{xy \mid x \in L_1, y \in L2\}$
- Exponenciação: $L^n = L \cdot L^{n-1}$ e $L^0 = \{\epsilon\}$
- Iteração: $L^* = \bigcup_{n=0}^{\infty} L^n$ (Chamada de Fecho de Kleene)

Essas operações fornecem os fundamentos para a manipulação e construção de linguagens formais. No entanto, para compreender os limites expressivos dessas linguagens e a complexidade dos mecanismos capazes de reconhecê-las, é necessário considerar uma classificação mais ampla para organizar as linguagens formais em diferentes classes hierárquicas.

Chomsky (1956) propôs a teoria das gramáticas gerativas, com o objetivo de estruturar sistematicamente como as sentenças das línguas naturais são formadas. Durante essa investigação, Chomsky observou que algumas gramáticas formais, como as gramáticas de estados finitos, não possuíam capacidade expressiva suficiente para representar adequadamente as estruturas da língua inglesa. Essa limitação motivou a formulação da chamada Hierarquia de Chomsky.

A hierarquia de Chomsky é uma classificação das linguagens formais com base no poder expressivo de suas respectivas gramáticas. Essa hierarquia organiza as linguagens em quatro classes principais: linguagens regulares, linguagens livres de contexto, linguagens sensíveis ao contexto e linguagens recursivamente enumeráveis. Cada classe representa um tipo de gramática com diferentes restrições na forma das regras de produção, e, consequentemente, diferentes capacidades computacionais. A Figura 1 ilustra essa hierarquia.

Figura 1 – Hierarquia de Chomsky (MENEZES, 2000)



O estudo de Chomsky sobre linguagens formais contribuiu significativamente para o desenvolvimento das linguagens artificiais, especialmente ao permitir avanços na análise léxica e sintática das linguagens de programação. A classificação proposta por Chomsky estruturou a compreensão das capacidades expressivas de diferentes modelos computacionais e se tornou referência fundamental no estudo das linguagens formais.

A partir dessa classificação, torna-se igualmente importante compreender as diferentes formas de representação das linguagens formais, uma vez que cada formalismo enfatiza aspectos distintos de sua estrutura e comportamento, oferecendo perspectivas complementares para análise, especificação e modelagem. A próxima seção examina esses formalismos em maior detalhe.

2.1.1 Representações de linguagens

Há diferentes maneiras de descrever as linguagens formais, cada uma destacando aspectos específicos de sua estrutura e comportamento. Os principais formalismos são: axiomático, operacional e denotacional. Compreender essas representações é essencial para a análise e uso apropriado das linguagens.

2.1.1.1 Representação axiomática

A representação axiomática de linguagens formais é feita por meio de gramáticas, que definem estritamente regras de produção e um símbolo inicial (axioma) da linguagem. Também pode ser chamada de *formalismo gerador*, pois, ao aplicar corretamente as regras de produção, é possível derivar todas as palavras pertencentes à linguagem.

Uma gramática é uma quádrupla ordenada $G = (V, T, P, S)$, onde:

- V: conjunto finito de variáveis (ou símbolos não terminais);
- T: conjunto finito de símbolos terminais, disjunto de V;
- P: conjunto finito de regras de produção, nas quais cada regra tem uma parte esquerda formada por uma palavra de $(V \cup T)^+$ e uma parte direita formada por

uma palavra de $(V \cup T)^*$;

- S: símbolo inicial, um elemento de V.

As variáveis representam símbolos que podem ser substituídos (derivados) por outras sequências de símbolos utilizando regras de produção definidas em P, enquanto os terminais são os símbolos finais da linguagem e não podem ser substituídos.

Devido à diferença no poder de representação, diferentes classes de linguagens formais requerem diferentes restrições sobre as regras de produção, de acordo com o nível da linguagem na hierarquia de Chomsky.

O processo de aplicar regras de produção é chamado de derivação. Para representar derivações, utiliza-se o símbolo $\rightarrow$, onde o lado esquerdo contém uma variável e o lado direito contém a sequência de variáveis e terminais que a substitui.

Exemplo: Se temos $A \rightarrow aA \mid b$, então a variável A pode ser derivada para a cadeia aA ou $(A \Rightarrow aA)$ para b $(A \Rightarrow b)$.

A partir de uma sequência de derivações, pode-se construir uma árvore de derivação, também denominada árvore sintática quando utilizada na análise sintática de uma linguagem de programação por um compilador (AHO et al., 2007).

A linguagem gerada por uma gramática G é comumente denotada por $L(G)$, e é composta por todas as cadeias de símbolos terminais deriváveis a partir de S. Duas gramáticas que geram a mesma linguagem são consideradas equivalentes (MENEZES, 2000).

Enquanto a representação axiomática descreve quais cadeias pertencem à linguagem por meio de regras sintáticas de derivação, a representação operacional foca nos mecanismos computacionais capazes de reconhecer essas cadeias. Nesse contexto, utilizam-se autômatos como modelos formais de processamento.

2.1.1.2 Representação operacional

A representação operacional descreve como uma cadeia de símbolos é processada ou reconhecida por meio de modelos computacionais abstratos. Esses modelos, denominados autômatos, simulam o comportamento de linguagens formais sob uma perspectiva procedural. Um autômato é uma máquina matemática capaz de ler uma cadeia de entrada, realizar transições entre estados com base em regras formais, e ao final determinar se a cadeia pertence à linguagem especificada (LINZ, 2016).

A entrada de um autômato consiste em uma sequência finita de símbolos, lida da esquerda para a direita. A cada símbolo lido, o autômato realiza uma transição de um estado para outro, conforme definido por sua função de transição. O estado representa a configuração interna da máquina em determinado momento da computação, e sua evolução

determina o comportamento da máquina frente à cadeia de entrada. Ao final da leitura, o estado final alcançado é avaliado para determinar a aceitação ou rejeição da cadeia.

Autômatos podem ser classificados como determinísticos ou não determinísticos. Em modelos determinísticos, para cada par (estado, símbolo) existe no máximo uma transição possível. Já em modelos não determinísticos, podem existir múltiplas transições para um mesmo par, o que permite explorar diferentes caminhos computacionais simultaneamente.

Distintos modelos de autômatos são utilizados para representar diferentes classes de linguagens, conforme a hierarquia de Chomsky:

- **Autômatos Finitos Determinísticos (AFDs)** e **Não Determinísticos (AFNs)** representam *linguagens regulares*. Esses modelos não utilizam memória auxiliar; toda a informação relevante é armazenada no estado atual da máquina (MENEZES, 2000).

  Ambos os modelos são formalmente definidos por uma quintúpla:

  $$M = (Q, \Sigma, \delta, q_0, F)$$

  onde:

  - Q: conjunto finito de estados;
  - Σ: alfabeto de entrada (conjunto de símbolos);
  - δ: função de transição, que define o próximo estado (ou conjunto de estados, no caso de AFNs) a partir do estado atual e do símbolo lido;
  - $q_0 \in Q$: estado inicial;
  - $F \subseteq Q$: conjunto de estados finais (de aceitação).

  Embora autômatos não determinísticos possam ser mais simples de construir em certos contextos, eles não possuem maior poder computacional que os modelos determinísticos: qualquer AFN pode ser convertido em um AFD equivalente, reconhecendo a mesma linguagem (MENEZES, 2000).

- **Autômatos com Pilha** (*Pushdown Automata* – PDA) são utilizados para representar *linguagens livres de contexto*. Esses modelos utilizam uma memória auxiliar do tipo pilha, permitindo reconhecer estruturas recursivas e aninhadas, como expressões com parênteses balanceados (MENEZES, 2000).

  Formalmente, um PDA é definido por uma 7-tupla (LINZ, 2016):

  $$M = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$$

  onde:

  - Q: conjunto finito de estados;
  - Σ: alfabeto de entrada;
  - Γ: alfabeto da pilha;
  - δ: função de transição: $Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma \to \mathcal{P}(Q \times \Gamma^*)$;
  - $q_0 \in Q$: estado inicial;
  - $Z_0 \in \Gamma$: símbolo inicial da pilha;
  - $F \subseteq Q$: conjunto de estados finais.

- **Máquinas de Turing** são modelos mais poderosos, capazes de reconhecer *linguagens sensíveis ao contexto* e *linguagens recursivamente enumeráveis*. Diferentemente dos autômatos anteriores, sua memória é composta por uma fita teoricamente infinita em ambas as direções, e seu poder computacional é equivalente ao de qualquer algoritmo efetivamente computável. A Máquina de Turing é o modelo central no estudo da decidibilidade e da complexidade computacional. (SIPSER, 2006b) Formalmente, uma Máquina de Turing é definida por uma 7-tupla:

$$M = (Q, \Sigma, \Gamma, \delta, q_o, q_{accept}, q_{reject})$$

  onde:

  - Q: conjunto finito de estados;
  - Σ: alfabeto de entrada (conjunto de símbolos), não contem o blank symbol;
  - Γ: alfabeto da fita;
  - δ: função de transição: $Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$;
  - $q_0 \in Q$: estado inicial;
  - $q_{accept} \in Q$: estados de aceitação;
  - $q_{reject} \in Q$: estados de rejeição.

A definição das Máquinas de Turing encerra a apresentação dos principais modelos computacionais associados à Hierarquia de Chomsky. A seguir, serão discutidas as diferentes formas de representação de linguagens formais. Dentre elas, destaca-se a representação denotacional, que, assim como na aritmética, utiliza operadores específicos para construir expressões que descrevem conjuntos de cadeias de maneira declarativa (SIPSER, 2006b).

2.1.1.3 Representação denotacional

Essa abordagem descreve formalmente quais cadeias pertencem a uma linguagem, sem detalhar como essas cadeias são geradas (como nas gramáticas) ou reconhecidas (como

nos autômatos). Trata-se, portanto, de uma forma direta de especificar linguagens a partir de conjuntos e operações.

O exemplo mais representativo de representação denotacional ocorre nas linguagens regulares, por meio das chamadas expressões regulares. Tais expressões utilizam operadores como:

- **Concatenação**, para sequenciar símbolos ou subexpressões;
- **União** (alternativa), representada por `|`, para indicar escolha entre cadeias;
- **Fecho de Kleene** (`*`), que permite repetição de uma subcadeia qualquer número de vezes (inclusive zero).

Por exemplo, a expressão regular `a(b|c)*` denota todas as cadeias que iniciam com a letra `a`, seguidas por qualquer número de ocorrências das letras `b` ou `c`.

Expressões regulares são amplamente utilizadas em áreas como análise léxica de compiladores, busca textual, validação de padrões e sistemas de correspondência de *strings*, graças à sua eficiência computacional e simplicidade sintática.

No entanto, essas expressões são limitadas ao escopo das linguagens regulares. Não são capazes, por exemplo, de representar linguagens que exigem estruturas aninhadas ou correspondência de símbolos em diferentes posições, características típicas de linguagens livres de contexto.

Para linguagens de maior complexidade (como livres de contexto ou sensíveis ao contexto), embora existam formas declarativas equivalentes como certas gramáticas formais ou formas normais (ex.: Forma Normal de Chomsky), não há um formalismo denotacional amplamente aceito ou padronizado como ocorre com expressões regulares (HOPCROFT; ULLMAN, 1979).

Logo, entende-se que diferentes formas de expressar uma linguagem focam em distintos aspectos, como a geração, o reconhecimento ou a apresentação matemática da linguagem.

2.2 Linguagens livres de contexto

A classe das linguagens livres de contexto, também chamadas de *linguagens tipo 2*, segundo a hierarquia de Chomsky, está situada entre as linguagens regulares (tipo 3) e as linguagens sensíveis ao contexto (tipo 1). Ela contém adequadamente todas as linguagens regulares, o que significa que qualquer linguagem regular pode ser descrita por meio dos formalismos próprios das linguagens livres de contexto (MENEZES, 2000).

Uma das características marcantes desta classe é a sua capacidade de representar estruturas recursivas e aninhadas. Por exemplo, a linguagem que representa cadeias

de parênteses corretamente balanceados não pode ser reconhecida por um autômato finito, mas é reconhecível por um *autômato de pilha* (PDA), que é o modelo operacional correspondente às linguagens livres de contexto.

Devido a essa expressividade, as linguagens livres de contexto são amplamente utilizadas na construção de compiladores, especialmente na etapa de análise sintática (*parsing*), onde é necessário compreender estruturas hierárquicas como comandos compostos, blocos de código e expressões aritméticas aninhadas (LINZ, 2016).

Essas linguagens podem ser formalmente descritas por meio de um modelo axiomático conhecido como gramática livre de contexto (GLC), que fornece uma definição precisa baseada em regras de produção.

2.2.1 Gramática livre de contexto

A gramática livre de contexto (GLC) representa o formalismo axiomático das linguagens livres de contexto (tipo 2 na hierarquia de Chomsky). É definida por uma quádrupla:

$$G = (V, T, P, S)$$

onde:

- V é um conjunto finito de variáveis (ou símbolos não-terminais);
- T é um conjunto finito de símbolos terminais, com $V \cap T = \emptyset$;
- P é um conjunto finito de regras de produção, cada uma da forma $A \to \alpha$, com $A \in V$ e $\alpha \in (V \cup T)^*$;
- $S \in V$ é o símbolo inicial.

A principal característica das GLCs é a restrição de que o lado esquerdo de toda produção contenha exatamente uma variável. Isso significa que as regras de derivação são aplicadas independentemente do contexto em que a variável aparece, ou seja, a substituição de uma variável por uma sequência de símbolos depende apenas dessa variável e não dos símbolos ao seu redor. É justamente essa propriedade que dá origem à nomenclatura "livre de contexto". Uma linguagem é classificada como *livre de contexto* se existe uma gramática livre de contexto que a gera (MENEZES, 2000).

Caracterização das Linguagens Livres de Contexto

Formalmente, uma linguagem L é considerada livre de contexto se há uma gramática $G = (V, T, P, S)$ tal que:

$$L(G) = \{w \in T^* \mid S \Rightarrow^* w\}$$

Ou seja, a linguagem é composta por todas as cadeias de símbolos terminais deriváveis a partir do símbolo inicial, por meio de uma sequência finita de aplicações das regras de produção da gramática - representado por $\Rightarrow^*$.

Exemplo 1 — Linguagem livre de contexto com duplo balanceamento: Considere a linguagem:

$$L_1 = \{a^n b^n \mid n \geq 0\}$$

Esta linguagem pode ser gerada pela gramática:

$$G_1 = (\{S\}, \{a, b\}, P_1, S)$$

com as produções:

$$P_1 = \{S \to aSb,\ S \to \varepsilon\}$$

A linguagem gerada por G_1 inclui cadeias como ε, `ab`, `aabb`, `aaabbb`, etc. Essa gramática modela o balanceamento simétrico entre os símbolos `a` e `b`. Ao substituir `a` por um parêntese de abertura `(` e `b` por um parêntese de fechamento `)`, a mesma gramática pode ser utilizada para gerar cadeias com parênteses corretamente aninhados, um exemplo clássico de linguagem livre de contexto.

Árvore de derivação e ambiguidade

A derivação de cadeias em uma GLC pode ser representada por uma árvore de derivação, cuja raiz é o símbolo inicial e as folhas são os símbolos terminais da cadeia. Esse tipo de representação é fundamental em compiladores, pois reflete a estrutura sintática dos programas.

Uma gramática é dita ambígua se existir ao menos uma cadeia da linguagem que possua mais de uma árvore de derivação distinta. Essa ambiguidade pode se manifestar como:

- *Derivação mais à esquerda (leftmost)* vs. *mais à direita (rightmost)*;
- Ambiguidades sintáticas estruturais (como em expressões aritméticas não parentizadas).

Gramáticas ambíguas são indesejáveis em muitas aplicações, especialmente na construção de compiladores, pois dificultam a interpretação semântica de estruturas sintáticas. Conforme será discutido na próxima seção, nem todas as linguagens livres de contexto — em especial aquelas geradas por gramáticas ambíguas — podem ser reconhecidas por um

Autômato de Pilha Determinístico (DPDA), o que ressalta a importância do estudo do determinismo no reconhecimento de linguagens.

2.2.2 Autômato de pilha

Como formalismo operacional para as linguagens livres de contexto, utiliza-se o *autômato de pilha* (PDA — *Pushdown Automaton*), que estende o autômato finito com o uso de uma pilha como memória auxiliar. Essa pilha permite o reconhecimento de estruturas aninhadas, característica típica das LLCs, e a facilidade de não determinismo torna o modelo capaz de reconhecer todo o conjunto das linguagens livres de contexto.

A pilha, componente central do PDA, é independente da fita de entrada e é considerada como tendo capacidade ilimitada. Operando segundo a lógica LIFO (*Last In, First Out*), o último símbolo inserido é o primeiro a ser removido. Frequentemente, a pilha é inicializada com um símbolo especial que marca sua base, garantindo controle sobre seu esvaziamento.

Existem duas formas clássicas de aceitação em um PDA:

- Aceitação por estado final: o autômato aceita a entrada se, ao final da leitura, estiver em um estado final, independentemente do conteúdo da pilha;
- Aceitação por pilha vazia: a entrada é aceita se, ao término da leitura, a pilha estiver completamente vazia, independentemente do estado atual.

Adicionalmente, é importante notar que a aceitação por estado final e a aceitação por pilha vazia são equivalentes em termos de poder expressivo, ou seja, qualquer PDA que aceita por um desses critérios pode ser transformado em outro que utiliza o critério alternativo sem alterar a linguagem reconhecida (HOPCROFT; ULLMAN, 1979).

Os PDAs operam por meio de dois mecanismos de leitura: uma cabeça de leitura sobre a fita de entrada e um ponteiro que acessa o topo da pilha. A função de transição é definida com base no estado atual, no símbolo atualmente lido da fita (ou ε) e no símbolo presente no topo da pilha. Com essas informações, a transição determina um novo estado, bem como uma operação sobre a pilha, como empilhar ou desempilhar símbolos (MENEZES, 2000).

Além disso, PDAs admitem transições por movimento vazio (*epsilon transitions*), que permitem à máquina mudar de estado sem consumir qualquer símbolo da entrada e/ou sem alterar o conteúdo da pilha. Essas transições aumentam a flexibilidade do modelo utilizando a facilidade de não determinismo.

É possível representar um autômato de pilha de forma gráfica por meio de diagramas de transição, os quais auxiliam na visualização do processo de reconhecimento de uma cadeia. Nesses diagramas, cada nó corresponde a um estado do autômato, enquanto as transições entre estados são representadas por setas direcionadas.

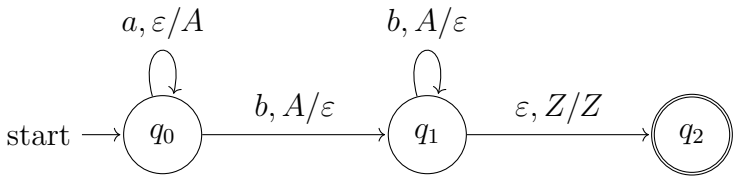
Cada seta é rotulada com uma tripla no formato (a, X, Y), onde:

- a representa o símbolo lido da fita de entrada (ou ε);
- X é o símbolo que deve estar no topo da pilha antes da transição;
- Y é a sequência de símbolos que substituirá X no topo da pilha (podendo ser ε para remoção).

Esse tipo de representação facilita a compreensão intuitiva das operações realizadas pelo PDA durante o reconhecimento de uma palavra.

A figura 2 apresenta um exemplo de autômato de pilha representado por um diagrama de transição para o reconhecimento da linguagem $L = \{a^n b^n \mid n > 0\}$. Nesse autômato, adota-se o critério de aceitação por estado final em conjunto com a verificação de pilha vazia, uma vez que, antes de alcançar o estado final, o autômato verifica se a pilha contém apenas o símbolo inicial `Z`.

Figura 2 – Exemplo de PDA para reconhecimento da linguagem $L = \{a^n b^n \mid n > 0\}$



A notação da função de transição é dada por: `<símbolo da fita>, <símbolo do topo da pilha> / <símbolo(s) a serem empilhados>`, onde o símbolo vazio é representado por ε. A mudança de estado é indicada pelas setas direcionadas entre os nós do diagrama, que representam os estados. O estado final é representado por um círculo duplo, enquanto o estado inicial é indicado por uma seta que aponta para ele, com *start*.

2.2.2.1 PDA determinístico (DPDA)

Um autômato com pilha é dito determinístico (DPDA) se, em qualquer configuração da computação, houver no máximo uma transição possível. Isso implica (HOPCROFT; ULLMAN, 1979):

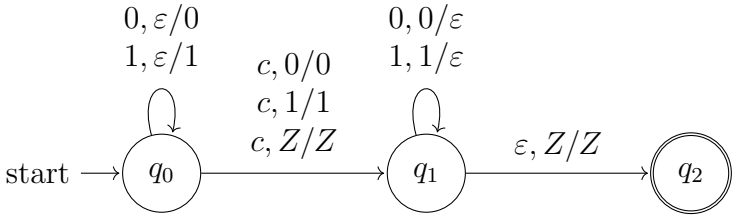
1. Para cada tripla (q, a, X) — estado atual, símbolo da entrada e símbolo do topo da pilha — existe no máximo uma transição definida;
2. Não pode haver conflito entre transições com leitura de símbolo da entrada e transições ε-move para a mesma condição de pilha. Ou seja, se uma transição com leitura real existe para (q, X), não pode haver outra com ε.

Como exemplo, considere a linguagem:

$$L = \{wcw^R \mid w \in \{0,1\}^*\}$$

Essa linguagem é reconhecida por um autômato de pilha determinístico conforme demonstrado na Figura 3, é possível reconhecer a linguagem utilizando um autômato determinístico pois o símbolo central fixo (`c`) permite identificar de forma determinística o ponto de inversão da palavra.

Figura 3 – Exemplo de DPDA para reconhecimento da linguagem $L = \{wcw^R \mid w \in \{0,1\}^*\}$ (HOPCROFT; ULLMAN, 1979)



A notação da função de transição utilizada no diagrama é dada por: `<símbolo da fita>, <símbolo do topo da pilha> / <símbolo(s) a serem empilhados>`,

onde o símbolo vazio é representado por ε. A mudança de estado é indicada por setas direcionadas entre os nós do grafo, que representam os estados. O estado final é representado por um círculo duplo, enquanto o estado inicial é indicado por uma seta *start*. Quando existem múltiplas transições originando-se de um mesmo estado, elas são posicionadas uma acima da outra no diagrama.

É importante observar que, conforme definido anteriormente, embora existam múltiplas transições possíveis para a leitura de um mesmo símbolo da fita, o autômato permanece determinístico. Isso ocorre porque a escolha da transição é unicamente determinada pelo símbolo atualmente presente no topo da pilha — e para cada combinação de estado, símbolo de entrada e símbolo da pilha, existe no máximo uma transição válida.

No entanto, essa previsibilidade, que torna os DPDAs eficientes para a análise sintática, também impõe certas limitações. Ao restringir as possibilidades de transição em favor do determinismo, exclui-se a capacidade de tratar algumas estruturas mais complexas que exigem escolhas alternativas durante a computação.

Com isso, torna-se evidente que os DPDAs não reconhecem todas as linguagens livres de contexto, mas apenas um subconjunto próprio delas. Esse subconjunto é conhecido como LINGUAGENS LIVRES DE CONTEXTO DETERMINÍSTICAS (LLCD). Como exemplo de linguagem que está fora desse subconjunto, temos:

$$L = \{ww^R \mid w \in \{a,b\}^*\}$$

Essa linguagem é livre de contexto (pode ser gerada por uma GLC), mas não pode ser reconhecida por nenhum autômato de pilha determinístico. Ela exige uma separação não determinística entre as duas metades da cadeia, o que requer um PDA não determinístico.

Teorema: Se $L = N(P)$ para algum autômato de pilha determinístico P, então L é uma linguagem livre de contexto não ambígua. (HOPCROFT; ULLMAN, 1979, p. 253)

Ou seja, toda linguagem reconhecida por um DPDA pode ser gerada por uma gramática livre de contexto não ambígua. Contudo, o contrário não é verdadeiro: existem linguagens livres de contexto não ambíguas que não podem ser reconhecidas por nenhum DPDA.

2.2.2.2 PDA Não Determinístico (NPDA)

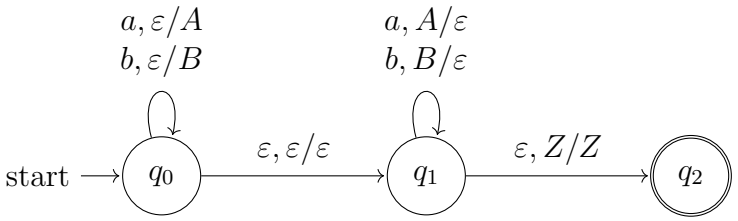
A introdução de não determinismo — ou seja, a capacidade de realizar escolhas múltiplas — é fundamental para que um PDA reconheça toda a classe das linguagens livres de contexto. Um autômato com pilha não determinístico (NPDA) pode explorar múltiplos caminhos computacionais simultaneamente, o que é necessário para lidar com estruturas sintáticas mais complexas.

Como exemplo, a linguagem:

$$L = \{ww^R \mid w \in \{a, b\}^*\}$$

Não pode ser reconhecida por nenhum DPDA, mas pode ser aceita por um NPDA. Nesse caso, o autômato pode, de forma não determinística, "adivinhar"o ponto de divisão entre as duas metades da palavra.

Figura 4 – Exemplo de NDPDA para reconhecimento da linguagem $L = \{ww^R \mid w \in \{a, b\}^*\}$.



A existência de múltiplos caminhos de execução é essencial nesse modelo, pois permite manipular estruturas simétricas e aninhadas mesmo quando não há marcadores explícitos, como o símbolo `c` usado no caso determinístico.

Portanto, enquanto os DPDAs são mais restritos, os NPDAs são necessários para reconhecer a totalidade das linguagens livres de contexto. Essa distinção é crucial em contextos como a construção de compiladores, onde o determinismo impacta diretamente na viabilidade da análise sintática eficiente.

Equivalência entre PDA e GLC

Existe uma equivalência formal entre autômatos com pilha (PDAs) e as gramáticas livres de contexto (GLCs). Mais precisamente, para toda linguagem livre de contexto

existe um PDA que a reconhece, e vice-versa. Logo é comprovado que o poder expressivo dos PDAs coincide exatamente com o das GLCs (HOPCROFT; ULLMAN, 1979).

Ambiguidade em gramáticas

Uma gramática é considerada ambígua quando uma mesma cadeia da linguagem pode ser derivada de múltiplas formas distintas, resultando em diferentes árvores de derivação. Essa ambiguidade pode ser detectada através da análise das derivações mais à esquerda (leftmost) ou mais à direita (rightmost), que devem ser únicas em gramáticas não ambíguas (SIPSER, 2006b).

É possível que uma gramática seja ambígua por construção, mas existam versões não ambíguas equivalentes. Contudo, existem linguagens chamadas **inerentemente ambíguas**, para as quais todas as gramáticas que as geram são necessariamente ambíguas.(LINZ, 2016).

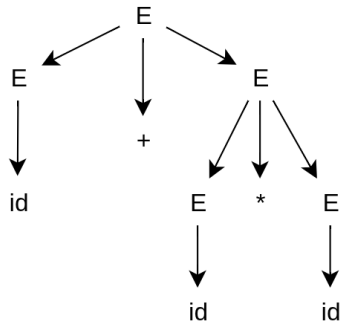
Como exemplo, considere as seguintes regras de produção de uma GLC:

$$E \rightarrow E + E \mid E * E \mid id$$

Esta gramática é ambígua, pois a mesma cadeia pode ser derivada de mais de uma maneira, resultando em diferentes árvores de derivação. Observe, nas Figuras 5 e 6, duas derivações distintas para a cadeia `id + id * id`.

Figura 5 – Derivação com prioridade do operador *, interpreta como `id + (id * id)`

$$E \Rightarrow E + E \Rightarrow id + E \Rightarrow id + E * E \Rightarrow id + id * id$$



Como as duas derivações produzem estruturas sintáticas diferentes para a mesma cadeia, concluímos que a gramática é ambígua.

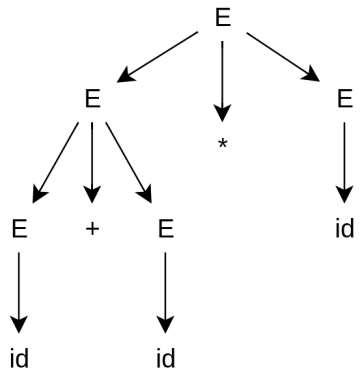
A seguir, apresenta-se um exemplo clássico de linguagem inerentemente ambígua, o que significa que todas as gramáticas que a geram são ambíguas (HOPCROFT; ULLMAN, 1979).

$$L = \{a^i b^j c^k \mid i = j \text{ ou } j = k\}$$

Também é possível ilustrar uma linguagem livre de contexto não ambígua que não

Figura 6 – Derivação com prioridade do operador +, interpreta como `(id + id) * id`

$$E \Rightarrow E * E \Rightarrow E + E * E \Rightarrow id + id * id$$



pertence à classe LLCD (Linguagens Livres de Contexto Determinísticas), conforme o exemplo a seguir:

$$L = \{a^n b^n \mid n \geq 0\} \cup \{a^n c^n \mid n \geq 0\}$$

Embora determinadas linguagens livres de contexto possam ser reconhecidas de forma determinística, a união de duas linguagens livres de contexto não ambíguas nem sempre resulta em uma linguagem que também possa ser reconhecida por um DPDA. Isso ocorre porque a classe das LLCD não é fechada sob a operação de união (HOPCROFT; ULLMAN, 1979).

Em outras palavras, mesmo que duas linguagens L_1 e L_2 sejam reconhecidas individualmente por DPDAs, não há garantia de que $L_1 \cup L_2$ pertença à classe LLCD. A construção de um único DPDA capaz de distinguir, de maneira determinística, a qual linguagem uma entrada pertence pode não ser possível, especialmente quando os prefixos das cadeias em L_1 e L_2 se sobrepõem de forma que impeçam a tomada de decisão determinística baseada apenas no símbolo corrente e no topo da pilha.

Esse resultado evidencia uma limitação importante do modelo determinístico em comparação ao não determinístico, cujas classes de linguagens são mais amplas e fechadas sob operações como união. O não fechamento da classe LLCD sob união é, portanto, uma distinção teórica relevante entre DPDA e NPDA, com implicações diretas no projeto de analisadores sintáticos e na escolha de formalismos gramaticais em compiladores.

Ambiguidade e análise sintática

Em compiladores, é desejável que a gramática da linguagem de programação seja não ambígua, pois isto garante que exista uma única interpretação sintática para cada construção, o que evita *backtraking*. Ferramentas como o *YACC* permitem resolver ambiguidades declarando precedência e associatividade dos operadores. Outra abordagem formal para lidar com ambiguidade é converter a gramática para formas normais, como

a Forma Normal de Chomsky e a Forma Normal de Greibach, a qual facilita a aplicação de algoritmos como o Cocke-Younger-Kasami (CYK), que determina se uma cadeia de caracteres pode ser gerada por uma determinada gramática livre de contexto e, se ela puder, como ela pode ser gerada.

Embora existam algoritmos que tentam eliminar ambiguidades em gramáticas (como transformação para formas normais ou uso de análise descendente com retrocesso), nem todas as ambiguidade são removíveis. Quando a linguagem é inerentemente ambígua, não existe gramática equivalente alguma que não seja ambígua.

Nesse sentido, as LLCD são uma subclasse importante e prática das LFCs. Seu uso é comum em compiladores devido à sua capacidade de serem analisadas por autômatos determinísticos, o que simplifica a construção de analisadores sintáticos, e torna os mais eficiente devido a ausência de regressão que é necessária no analisador sintático de gramáticas ambiguas. No entanto, sua limitação em termos de fechamento e poder de reconhecimento em relação a linguagens mais gerais torna necessário o uso de autômatos não determinísticos ou técnicas mais complexas para certos tipos de linguagem.

2.2.3 Linguagens livres de contexto determinísticas

As linguagens livres de contexto determinísticas (LLCD) são aquelas que podem ser reconhecidas por um autômato de pilha determinístico (DPDA). Além disso, são caracterizadas por não possuírem ambiguidade em sua representação gramatical.

Formalmente, uma linguagem L é dita *livre de contexto determinística* se existe um DPDA P tal que $L = L(P)$, ou seja, L é aceita por um DPDA. Este modelo não realiza escolhas ambíguas em nenhuma configuração de leitura e transição (HOPCROFT; ULLMAN, 1979).

As LLCDs formam uma subclasse própria das LFC, com propriedades distintas. Por exemplo, enquanto a classe das LFCs é fechada para as operações de união, concatenação e estrela de Kleene, a classe das LLCDs é fechada apenas sob complemento. Portanto, operações como união, intersecção e concatenação não são garantidamente fechadas no domínio determinístico.

Tabela 1 – Propriedades de Fechamento: GLC vs. LLCD

Operação	GLC (Geral)	LLCD (Determinística)
União	Fechada	**Não fechada**
Interseção	Não fechada	Não fechada
Complemento	Não fechada	**Fechada**
Concatenação	Fechada	Não fechada
Fecho de Kleene	Fechada	Não fechada

2.2.4 Complexidade Computacional e Eficiência

A distinção entre LLCD e LLC geral não é meramente teórica; ela possui implicações profundas na complexidade assintótica dos algoritmos de reconhecimento e análise sintática (*parsing*).

No contexto de autômatos finitos (Linguagens Regulares), a execução consiste apenas em atualizar o estado atual para cada símbolo da entrada. Portanto, conforme Sipser (2006b), o tempo de execução é $\Theta(n)$, onde n é o comprimento da palavra de entrada. Essa eficiência linear e o uso de espaço constante são ideais para processamento de texto.

Ao avançarmos para as Linguagens Livres de Contexto, a eficiência depende estritamente do determinismo. A Figura abaixo ilustra o impacto dramático dessa diferença no tempo de processamento à medida que a entrada cresce.

1. **Eficiência das LLCDs:** Toda linguagem livre de contexto determinística pode ser reconhecida em tempo linear. Segundo Aho e Ullman (1974), autômatos de pilha determinísticos executam em tempo linear $O(n)$. Na prática de compiladores, isso se reflete nos algoritmos de análise LR(k). Aho, Sethi e Ullman (1986) estabelecem que as linguagens analisáveis por parsers LR(1) correspondem exatamente às LLCDs, garantindo processamento rápido e previsível. Kozen (1997) reforça que todo DPDA opera em tempo $O(n)$.

2. **Ineficiência das LLCs Gerais:** Para gramáticas não-determinísticas ou inerentemente ambíguas, não existe algoritmo de *parsing* que garanta tempo linear. Algoritmos genéricos, como o de Earley ou CYK, possuem complexidade de pior caso cúbica $O(n^3)$ (EARLEY, 1970; AHO, 1972).

A ambiguidade e o não-determinismo impedem a análise linear. Knuth (1965a) demonstrou que apenas gramáticas LR(k) (que geram LLCDs) evitam a explosão exponencial ou cúbica causada pela ambiguidade estrutural. Como sintetizado por Sipser (2006b), “enquanto DCFLs podem ser analisadas deterministicamente em tempo linear, CFLs gerais podem exigir tempo cúbico” .

2.2.5 Implicações Práticas da Complexidade Computacional

Com base na fundamentação acima, torna-se evidente a importância de métodos que priorizem a geração de DPDAs. Embora o Operador Pushdown seja capaz de denotar linguagens livres de contexto gerais, a conversão indiscriminada para Autômatos de Pilha Não-Determinísticos (NPDAs) acarretaria em uma penalidade de desempenho inaceitável ($O(n^3)$) para aplicações de compiladores, que exigem tempo linear ($O(n)$).

Portanto, a identificação de restrições e a aplicação de otimizações (como a arquitetura ternária e a eliminação de transições vazias) são fundamentais para garantir, sempre

que a linguagem permitir, a geração de autômatos determinísticos. Isso assegura que o analisador resultante pertença à classe de complexidade eficiente das DCFLs.

2.3 PDOp e expressões livres de contexto

As expressões regulares clássicas, conforme propostas por Kleene (1956), foram concebidas como um formalismo denotacional destinado à representação precisa das linguagens regulares, utilizando três operadores fundamentais: união (+), concatenação e fecho de Kleene (*). Contudo, essas expressões são intrinsecamente limitadas ao reconhecimento de linguagens regulares, não sendo capazes de denotar linguagens com estruturas mais complexas, como linguagens livres de contexto que exigem correspondência aninhada ou recursiva.

Silva, Oliveira e Santos (2025) propõem a introdução de um novo operador, denominado operador de empilhamento (*Pushdown Operator* - PDOp), que pode ser incorporado às expressões regulares clássicas. O PDOp em combinação aos operadores de expressões regulares possibilita a representação formal de linguagens livres de contexto preservando a clareza sintática e a natureza declarativa das expressões regulares. A construção desse operador possibilitou a denotação de LLC por meio de expressões, aproximando o formalismo de expressões regulares.

2.3.1 Modelo denotacional do PDOp

Silva, Oliveira e Santos (2025) apresentaram o PDOp como um novo mecanismo de extensão das expressões regulares, constituindo um *formalismo denotacional* para linguagens livres de contexto. O formalismo denotacional busca descrever, de forma sintática e declarativa, os elementos de uma linguagem a partir de expressões que a definem diretamente, sem a necessidade de mecanismos operacionais como autômatos.

Ao introduzir o PDOp, pretende-se estabelecer um modelo que una a simplicidade das expressões regulares com o poder expressivo dos PDAs, permitindo representar diretamente linguagens livres de contexto de forma mais compacta, legível e manipulável.

Formalmente, o PDOp é definido por Silva, Oliveira e Santos (2025) da seguinte maneira:

$$L(a{:}(b_{1,1}\#b_{1,2}\#...\#b_{1,n_1}\ ;\ b_{2,1}\#b_{2,2}\#...\#b_{2,n_2}\ ;\ ...\ ;\ b_{m,1}\#b_{m,2}\#...\#b_{m,n_m})) \qquad (1)$$

onde o símbolo ";" separa os grupos $(b_{i,1}\#\ldots\#b_{i,n_i})$. Cada $b_{i,j}$ representa uma expressão regular ou uma expressão com o operador de empilhamento. O índice n_i é um número inteiro tal que $n_i \geq 2$, e m é um número inteiro $m \geq 1$, que determina o número de chamadas recursivas alteradas do operador.

Note que o uso PDOPp permite várias possibilidades de uso. Casos como os apresentados a seguir, são exemplos que podem ocorrer sem que haja erros sintáticos

associados (OLIVEIRA, 2024).

Tabela 2 – Exemplos de expressões com PDOp e respectivos valores de m e n

Expressão	Valores de m e n
$\varepsilon : \varepsilon \# \varepsilon$	$m = 1, n_1 = 2$
$a : \varepsilon \# \varepsilon$	$m = 1, n_1 = 2$
$\varepsilon : b \# \varepsilon$	$m = 1, n_1 = 2$
$\varepsilon : \varepsilon \# c$	$m = 1, n_1 = 2$
$a : b \# \varepsilon$	$m = 1, n_1 = 2$
$a : \varepsilon \# c$	$m = 1, n_1 = 2$
$\varepsilon : b \# c$	$m = 1, n_1 = 2$
$a : b \# c$	$m = 1, n_1 = 2$
$\varepsilon : a \# b \# c$	$m = 1, n_1 = 3$
$\varepsilon : a(\varepsilon : a \# b \# c) b \# c$	$m = 1, n_1 = 2$
$\varepsilon : a \# b(\varepsilon : a \# b \# c) c$	$m = 1, n_1 = 2$
$\varepsilon : a(\varepsilon : a(\varepsilon : a \# b \# c) b \# c) b \# c$	$m = 1, n_1 = 2$
$\varepsilon : a(\varepsilon : a \# b(\varepsilon : a \# b \# c) c) b \# c$	$m = 1, n_1 = 2$
$\varepsilon : a(\varepsilon : a \# b \# c) b(\varepsilon : a \# b \# c) c$	$m = 1, n_1 = 1$
$\varepsilon : a(\varepsilon : a(\varepsilon : a \# b \# c) b(\varepsilon : a \# b \# c) c) b \# c$	$m = 1, n_1 = 2$
$(\varepsilon : (b \# b; c \# c)) : a \# a$	$m = 1, n_1 = 2$
$(\varepsilon : a \# a) : (b \# b; c \# c)$	$m = 2, n_1 = 2, n_2 = 2$
$\varepsilon : ((b(\varepsilon : a \# a)) \# b; c \# c)$	$m = 2, n_1 = 2, n_2 = 2$
$\varepsilon : ((b \#((\varepsilon : a \# a) b)); c \# c)$	$m = 2, n_1 = 2, n_2 = 2$
$\varepsilon : (b \# b; (c(\varepsilon : a \# a)) \# c)$	$m = 2, n_1 = 2, n_2 = 2$
$\varepsilon : (b \# b; c \#((\varepsilon : a \# a) c))$	$m = 2, n_1 = 2, n_2 = 2$
$\varepsilon : (b \# b; c \# c \# a \# a)$	$m = 2, n_1 = 2, n_2 = 4$
$(\varepsilon : a \# a; b \# b \# \varepsilon) : (b \# b; c \# c)$	$m = 2, n_1 = 2, n_2 = 2$
$\varepsilon : ((b(\varepsilon : a \# a)) \# b \# b; c \# c)$	$m = 2, n_1 = 3, n_2 = 2$
$\varepsilon : ((b \#((\varepsilon : a \# a) b) \# c); c \# c)$	$m = 2, n_1 = 1, n_2 = 2$
$\varepsilon : (b \# b \# c \# b; (c(\varepsilon : a \# a)) \# c)$	$m = 2, n_1 = 4, n_2 = 2$
$\varepsilon : (b \# b; c \# c \#((\varepsilon : a \# a) c))$	$m = 2, n_1 = 2, n_2 = 3$

De acordo com Oliveira (2024), Silva, Oliveira e Santos (2025), o PDOp possui maior precedência do que os operadores clássicos de expressões regulares, conforme apresentado a seguir.

- União: $c : a \# a + b = (c : a \# a) + b$
- Concatenação: $cb : a \# d = c(b : a \# d)$
- Fecho de Kleene: $\varepsilon : c \# b^* = (\varepsilon : c \# b)^*$
- Pushdown operator: $a : b \# c : d \# e = (a : b \# c) : d \# e$

No último caso, como os dois PDOps têm a mesma precedência, aplica-se a avaliação da esquerda para a direita (associatividade à esquerda).

Para demonstrar que o conjunto das LLC é equivalente ao conjunto de expressões que utilizam o PDOp em combinação com os operadores regulares clássicos, (OLIVEIRA, 2024) apresentou duas demonstrações formais. A primeira demonstração mostra que toda gramática livre de contexto pode ser transformada em uma expressão equivalente com o uso do PDOp. A segunda demonstração comprova que toda expressão contendo o operador

de empilhamento pode ser convertida, por meio de um algoritmo, em um autômato de pilha não determinístico. Esses resultados formalizam que as expressões com PDOp, combinadas com operadores regulares, denotam exatamente o conjunto das linguagens livres de contexto. Diante dessa prova, Oliveira (2024) chamou as expressões que utilizam o PDOp em combinação com os operadores regulares clássicos de EXPRESSÕES LIVRES DE CONTEXTO.

2.3.2 Algoritmo de conversão de expressões livres de contexto em NPDA

A entrada do algoritmo é uma expressão livre de contexto z, definida sobre o alfabeto Σ. A saída é um autômato de pilha $M = (Q, \Sigma, \Gamma, \delta, q_0, Z, F)$ que reconhece a mesma linguagem que a expressão z, ou seja, reconhece $L(z)$. O procedimento foi inspirado no algoritmo de construção de Thompson (também conhecido como algoritmo de McNaughton-Yamada-Thompson) (AHO et al., 2007).

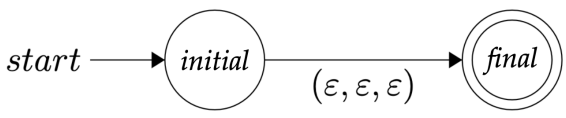
A ideia central do algoritmo é construir, de forma recursiva, um PDA que reconhece, por estado final, a linguagem denotada pela expressão livre de contexto E. Suponha que $L = L(E)$, para uma expressão livre de contexto E para algum autômato de pilha M com as seguintes características (OLIVEIRA, 2024):

1. Exatamente um estado de aceitação;
2. Nenhuma transição entra no estado inicial;
3. Nenhuma transição sai do estado de aceitação.

Cada operação (concatenação, união, fecho de Kleene e o próprio PDOp) é representada por um conjunto de transições entre os estados do PDA, seguindo uma abordagem de construção incremental. A etapa inicial do algoritmo envolve a análise da expressão símbolo por símbolo, identificando operandos, operadores e subexpressões. A seguir, são apresentadas a base da indução e os passos indutivos do algoritmo.

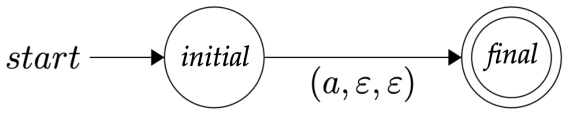
BASE: Para a expressão ε, constrói-se o PDA conforme ilustrado na Figura 7. Aqui, *initial* é um novo estado, que serve como estado inicial do PDA, e *final* é outro novo estado, o qual atua como estado de aceitação do PDA.

Figura 7 – Base do PDA para ε (OLIVEIRA, 2024).



Para qualquer símbolo a pertencente ao alfabeto Σ, constrói-se o PDA conforme ilustrado na Figura 8.

Figura 8 – Base do PDA para qualquer símbolo a em Σ (OLIVEIRA, 2024).

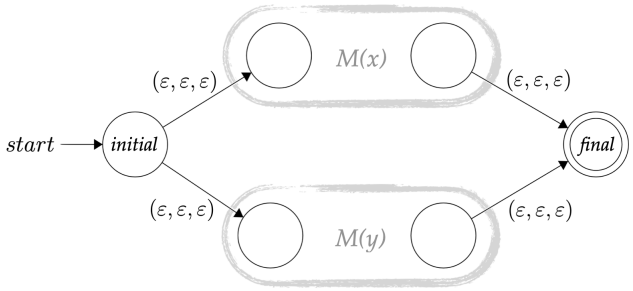


A saber, *initial* e *final* representam novos estados, atuando como estado inicial e estado de aceitação, respectivamente. Vale destacar que, nas construções apresentadas nas Figuras 7 e 8, um único PDA foi gerado para cada caso, com novos estados, para cada ocorrência de ε ou de algum símbolo a como subexpressão da expressão de entrada.

INDUÇÃO: Suponha que $M(x)$, $M(y)$ e $M(z)$ sejam PDAs para as expressões livres de contexto x, y e z, respectivamente.

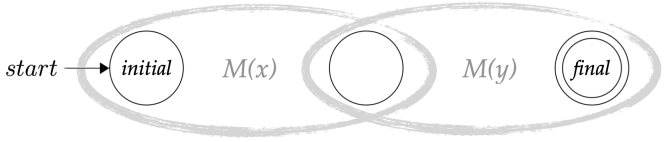
a) Suponha que $E = x \mid y$. Então, $M(E)$, o PDA para E, é construído conforme mostrado na Figura 9.

Figura 9 – PDA para a operação de união (OLIVEIRA, 2024).



b) Suponha que $E = xy$. Então, constrói-se $M(E)$ conforme mostrado na Figura 10.

Figura 10 – PDA para a operação de concatenação (OLIVEIRA, 2024).



c) Suponha que $E = x^*$. Então, para E constrói-se o PDA $M(E)$ conforme ilustrado na Figura 11.

d) $E = x : (y_{1,1}\#y_{1,2}\#...\#y_{1,n_1}\ ;\ y_{2,1}\#y_{2,2}\#...\#y_{2,n_2}\ ;\ ...\ ;\ y_{m,1}\#y_{m,2}\#...\#y_{m,n_m})$. A construção do PDA para a extensão do operador de empilhamento está ilustrada na Figura 12.

As LLC são reconhecidas por PDAs, modelos que operam com memória auxiliar em pilha e que lidam com estruturas aninhadas e recursivas. No entanto, tais modelos podem entrar em comportamentos indefinidamente recursivos ou cíclicos. Esse aspecto

Figura 11 – PDA para o fecho de Kleene (OLIVEIRA, 2024).

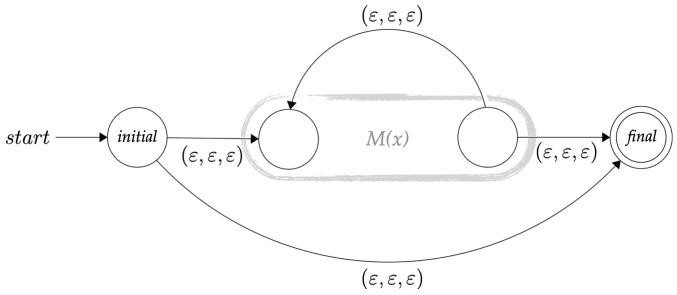
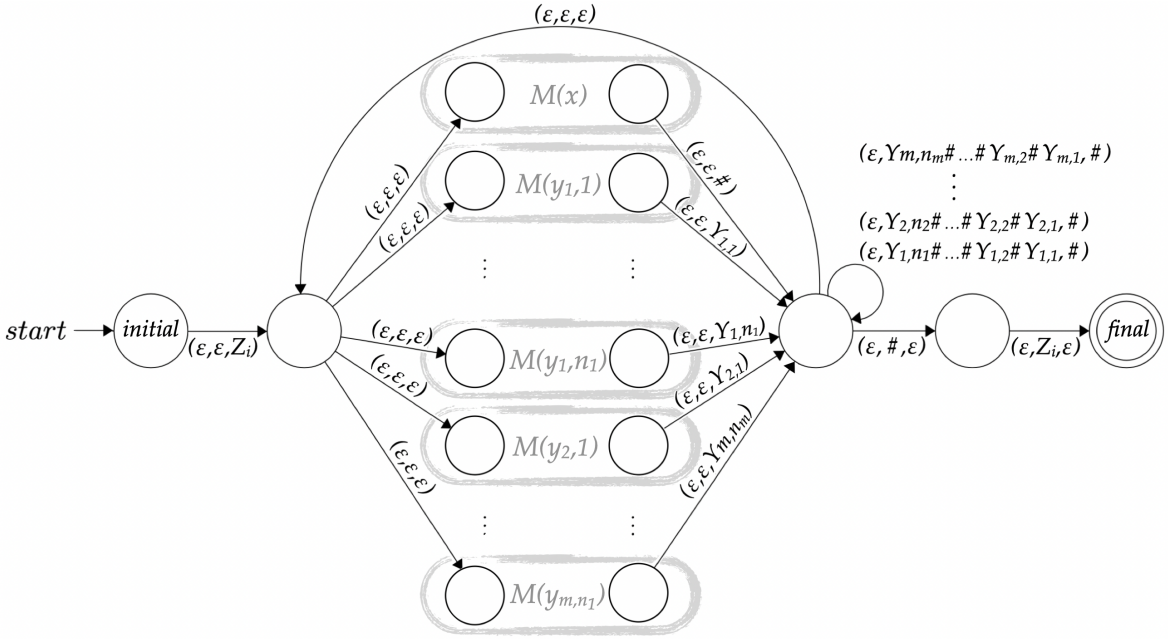


Figura 12 – PDA para o PDOp (OLIVEIRA, 2024).



torna evidente a necessidade de refletir sobre a possibilidade de parada dessas construções. Em determinadas situações, existe a possibilidade de a execução do PDA nunca terminar. Esse problema está intimamente ligado a um dos resultados mais fundamentais da ciência da computação: o PROBLEMA DA PARADA, tema a ser discutido na próxima seção Decibilidade de Problemas.

2.4 Decidibilidade de problemas

A análise dos limites computacionais e da decidibilidade de problemas é realizada com base no modelo computacional mais expressivo conhecido: a máquina de Turing. Proposta por Turing (1936b), essa máquina teórica é capaz de simular qualquer algoritmo — ou seja, qualquer processo computável. Por essa razão, é considerada o modelo padrão para a definição de computabilidade (SIPSER, 2006b).

A máquina de Turing utiliza uma fita de comprimento ilimitado como memória, sobre a qual uma cabeça de leitura/gravação pode se mover livremente para a esquerda ou para a direita. Diferentemente de modelos anteriores, que possuíam leitura unidirecional, a máquina de Turing é bidirecional e permite reescrita sobre a fita. Isso implica que sua execução pode não necessariamente terminar, uma vez que a cabeça pode mover-se indefinidamente sem atingir um estado de parada.

A função de transição de uma máquina de Turing é definida como:

$$\delta : Q \times \Gamma \to Q \times \Gamma \times \{E, D\}$$

Ou seja, se a máquina está no estado q, lendo o símbolo a na fita, e $\delta(q, a) = (r, b, E)$, então ela escreve o símbolo b sobre o a, transita para o estado r e move a cabeça para a esquerda (indicado por E; à direita, se D).

Durante a computação, a entrada é posicionada no início da fita, e o restante das células é preenchido com um símbolo branco (β ou $\square$). A fita é considerada ilimitada à direita (SIPSER, 2006b).

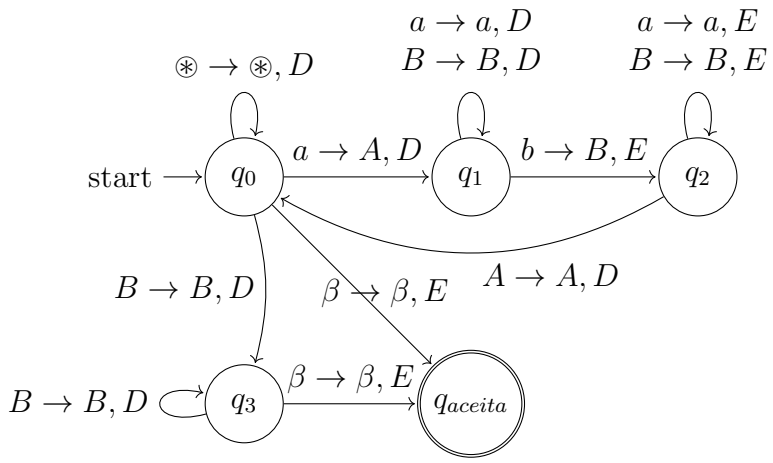
Como exemplo de linguagem reconhecida por uma máquina de Turing, tem-se:

$$L = \{a^n b^n \mid n \geq 0\}$$

Essa linguagem é reconhecida pela máquina representada na Figura 13.

A função de transição utilizada segue a notação: `<símbolo lido → símbolo escrito, direção>`. O símbolo inicial é denotado por $\circledast$ e símbolo branco, utilizado para representar espaços vazios na fita, aparece como β na Figura 13. Apesar de sua expressividade, a máquina de Turing também apresenta limitações. Para compreender esses limites, introduz-se o conceito de Máquina de Turing Universal.

Figura 13 – Máquina de Turing para $L = \{a^n b^n \mid n \geq 0\}$



Máquina de Turing universal

Proposta também por Turing (1936b), esta máquina é capaz de simular qualquer outra máquina de Turing a partir de sua descrição codificada e da entrada correspondente. Em termos práticos, ela recebe como entrada um par $\langle M, w \rangle$, onde M é a codificação de uma máquina de Turing e w é uma cadeia de entrada, e executa exatamente os mesmos passos que M realizaria sobre w.

A máquina universal é útil para descrever algoritmos em um nível mais abstrato, ocultando detalhes de implementação, e para demonstrar propriedades fundamentais da computação, como o limite da decidibilidade.

Problemas decidíveis e reconhecíveis

Dado esse contexto, distinguem-se dois tipos fundamentais de problemas (SIPSER, 2006b):

- Um problema é decidível se existe uma máquina de Turing que, para toda entrada w, termina a computação (aceitando ou rejeitando) em tempo finito. As linguagens associadas a esses problemas são chamadas *Turing-decidíveis*.

- Um problema é reconhecível se existe uma máquina de Turing que aceita toda palavra pertencente à linguagem. No entanto, se a palavra não pertencer à linguagem, a máquina pode rejeitar ou entrar em loop. Essas linguagens são chamadas *Turing-reconhecíveis*.

Entre as propriedades importantes, destacam-se (SIPSER, 2006b):

- Toda linguagem decidível é também reconhecível;

- O complemento de uma linguagem decidível é decidível;

- O complemento de uma linguagem reconhecível pode não ser reconhecível.

Esse último ponto é particularmente relevante, pois evidencia que há problemas para os quais não apenas não se conhece uma solução, mas sequer se pode verificar computacionalmente se uma dada entrada pertence à linguagem — delimitando a computação feita por máquinas de Turing, e apresentando uma classe de linguagem que não está inserida na hierarquia de Chomsky. A seguir, são apresentados exemplos clássicos de problemas computacionais classificados de acordo com sua decidibilidade:

Problemas decidíveis

São problemas para os quais existe uma máquina de Turing que sempre termina (aceitando ou rejeitando) qualquer entrada. Exemplos:

- Problema da aceitação para autômatos finitos: Dado um AF A e uma cadeia w, decidir se A aceita w. Esse problema pode ser resolvido percorrendo a cadeia segundo as transições definidas.
- Problema de equivalência para gramáticas regulares: Dadas duas expressões regulares R_1 e R_2, decidir se $L(R_1) = L(R_2)$. Pode ser resolvido por conversão para autômatos finitos e comparação de suas linguagens.
- Problema de aceitabilidade em GLC: Dada uma GLC G e uma cadeia w, decidir se G gera w, conforme descrito anteriormente por meio da Forma Normal de Chomsky.

Problemas reconhecíveis, mas indecidíveis

Nesses problemas, existe uma máquina de Turing que aceita entradas da linguagem, mas pode não parar se a entrada não for aceita. Exemplos:

- Problema da Parada (*Halting Problem*): Dado um par $\langle M, w \rangle$, onde M é uma máquina de Turing e w uma entrada, decidir se M para ao executar w. Esse problema será discutido na próxima subseção.
- Problema da aceitação em máquinas de Turing: Dado uma máquina de Turing M e uma cadeia w, decidir se M aceita w. A máquina pode aceitar, rejeitar ou entrar em loop.

Problemas irreconhecíveis

São problemas para os quais não existe sequer uma máquina de Turing que reconheça a linguagem correspondente. Exemplos:

- Complemento do problema da parada: A linguagem

$$\overline{\text{PARA}_{MT}} = \{\langle M, w\rangle \mid M \text{ não pára ao processar } w\}$$

não é Turing-reconhecível, pois não é possível confirmar que uma máquina *nunca* parará.

- Problema da equivalência para máquinas de Turing: Dado duas máquinas M_1 e M_2, decidir se $L(M_1) = L(M_2)$. Não existe algoritmo que resolva esse problema para todas as entradas possíveis.

Esses exemplos evidenciam que, embora muitas linguagens e problemas possam ser tratados de forma automática, existe um limite fundamental imposto pelo próprio modelo da máquina de Turing. No entanto, é importante compreender como esse limite se manifesta também em modelos computacionais menos expressivos, como os PDAs. A próxima seção apresenta o PROBLEMA DA PARADA no contexto das máquinas de Turing e discute como ele é tratado no caso dos PDAs.

2.4.1 Problema da parada

Entre os problemas fundamentais da computação está o Problema da Parada. Formulado por Turing (1936b), esse problema busca responder se uma máquina de Turing M irá parar ao processar uma entrada w, ou continuará executando indefinidamente.

Formalmente, o problema pode ser expresso como a linguagem:

$$\text{PARA}_{MT} = \{\langle M, w\rangle \mid M \text{ é uma máquina de Turing que pára sobre a entrada } w\}$$

Embora seja possível simular a execução de M sobre w e aceitar se ela termina, não é possível construir um algoritmo que decida isso para todos os casos. Isso ocorre porque, em caso de laço infinito, não há como distinguir entre uma computação que nunca termina e uma que apenas está demorando para concluir. Assim, o PROBLEMA DA PARADA é reconhecível, mas não decidível.

Essa limitação é diretamente consequência do teorema de Turing e representa um marco teórico: há problemas que nenhuma máquina de Turing pode resolver, o que estabelece um limite claro para o poder computacional.

Por outro lado, modelos computacionais menos expressivos, como os autômatos finitos ou PDAs, operam sobre classes de linguagens mais restritas, e, portanto, possuem propriedades decidíveis. Nesses casos, o problema análogo ao da parada assume a forma do problema de aceitabilidade, o qual é decidível: existe um algoritmo que determina, em tempo finito, se uma cadeia pertence à linguagem reconhecida por esses modelos.

No contexto de GLCs, esse problema consiste em decidir se uma gramática G gera uma palavra w. Segundo Sipser (2006b), seria suficiente tentar todas as derivações

possíveis. Contudo, como uma GLC pode gerar infinitas derivações para uma mesma palavra, esse método não é viável. Para contornar essa limitação, utiliza-se a Forma Normal de Chomsky, na qual uma cadeia w de comprimento n terá no máximo $2n - 1$ passos de derivação.

Com isso, é possível construir uma máquina de Turing que decide o problema de aceitabilidade A_{GLC}, da seguinte forma conforme descrito por Sipser (2006b):

> Máquina M para decidir A_{GLC}: Dada a entrada $\langle G, w \rangle$, onde G é uma GLC e w uma cadeia:
>
> 1. Converta G para a Forma Normal de Chomsky;
> 2. Liste todas as derivações com no máximo $2n - 1$ passos (ou 1, se $n = 0$);
> 3. Se alguma delas gera w, aceite; caso contrário, rejeite.

Como toda GLC possui um PDA equivalente, segue-se que o problema de reconhecimento para PDAs também é decidível: basta converter o PDA para uma GLC e aplicar o algoritmo anterior.

Por fim, como definido por Silva, Oliveira e Santos (2025), toda GLC pode ser representada por uma expressão livre de contexto utilizando PDOp. Assim, a aceitabilidade de uma expressão livre de contexto também é decidível, desde que sua semântica seja bem definida e equivalente a uma GLC.

Neste capítulo foram apresentados os conceitos e resultados essenciais que sustentam o desenvolvimento desta pesquisa. Com essa base conceitual consolidada, o próximo capítulo abordará o procedimento metodológico a ser realizado pela pesquisa.

3 FORMULAÇÃO DE ARQUITETURA DETERMINÍSTICA PARA O CASO BASE DO PDOP

Este capítulo descreve a evolução do modelo de geração de autômatos de pilha a partir de expressões livres de contexto com PDOp. Inicialmente, estabelece-se o contexto teórico e as limitações da abordagem predecessora, fundamentando a necessidade de uma nova proposta algorítmica para o problema enfrentado neste trabalho.

Para atingir os objetivos de eficiência e determinismo, o desenvolvimento da solução seguiu um fluxo sistemático estruturado nas seguintes etapas:

1. **Análise crítica e identificação de lacunas** - Investigação das limitações do modelo predecessor, especificamente quanto ao problema da parada em ciclos vazios e à geração de autômatos não determinísticos.

2. **Reestruturação arquitetural** - Proposição da arquitetura baseada em "Estado Hub" e decomposição ternária do operador (Push, Pivô, Pop) para linearizar o fluxo de processamento.

3. **Redefinição de operadores** - Alteração estrutural nos algoritmos de Fecho de Kleene e União para segregar estados de leitura e estados vazios, mitigando ambiguidades.

4. **Otimização e determinização** - Desenvolvimento de um *pipeline* de algoritmos (remoção de transições não-redutoras, tratamento de conflitos ε-POP e criação de Superestados) para maximizar o determinismo do autômato resultante.

As seções a seguir detalham cada uma dessas etapas, fundamentando as decisões de projeto e apresentando as formalizações necessárias.

3.1 Análise das fontes de não-determinismo e não parada do PDOp

Antes de fundamentar a proposta deste trabalho, torna-se imperativo realizar uma análise rigorosa do caso base do PDOp, proposto originalmente por Silva, Oliveira e Santos (2025) e Oliveira (2024), e seu algoritmo para converter expressões livres de contexto com PDOp em APND.

3.1.1 A necessidade de determinismo

No domínio das linguagens regulares, sabe-se que todo AFN possui um AFD equivalente. Conforme (HOPCROFT; MOTWANI; ULLMAN, 2006), a execução de um AFD ocorre em tempo proporcional ao comprimento da entrada, ou seja, $\Theta(n)$, onde n é o tamanho da palavra w (SIPSER, 2006b).

Entretanto, na das Linguagens livres de contexto, essa equivalência de eficiência desaparece. A literatura estabelece dois cenários distintos:

1. **Linguagens livres de contexto gerais** — Requerem modelos não determinísticos como os APNDs, não existe procedimento de reconhecimento que obtenha desempenho assintótico melhor que o tempo cúbico no pior caso. Resultados clássicos demonstram que o reconhecimento de CFL gerais exige, no pior cenário, $\Theta(n^3)$ (EARLEY, 1970; AHO, 1972; SIPSER, 2006a). Isso signfica que a ambiguidade estrutural e o não determinismo impedem qualquer possibilidade de processamento em tempo linear, podendo inclusive levar à explosão exponencial de derivações em gramáticas ambíguas (AHO, 1972). Tais limitações foram rigorosamente estabelecidas em trabalhos fundamentais como Earley (1970) e Aho (1972).

2. **Linguagens livres de contexto determinísticas** — Esta subclasse, correspondente às linguagens reconhecíveis por APDs, admite procedimentos de reconhecimento em tempo linear $O(n)$ (AHO, 1972; KOZEN, 1997; SIPSER, 2006a). As linguagens dessa categoria, notadamente aquelas descritas por gramáticas LR(k) (KNUTH, 1965b), evitam a explosão combinatória presente nas gramáticas não determinísticas e constituem a base prática para a maioria das linguagens de programação modernas, conforme estabelecido por Knuth (KNUTH, 1965b).

O algoritmo de conversão de expressões livres de contexto com PDOp em autômatos de pilha (OLIVEIRA, 2024), por lidar exatamente com o conjunto das linguagens livres de contexto, produz como resultado um APND, que é a máquina que reconhece exatamente o conjunto das linguagens livres de contexto. Contudo, embora sua corretude seja formalmente estabelecida, do ponto de vista de aplicabilidade prática em máquinas determinísticas, a execução de um APND resulta em processos computacionais não eficientes.

Diante disso, a motivação primária para a reestruturação do algoritmo de conversão de expressões livres de contexto com PDOp em autômatos de pilha reside na distinção fundamental entre diferentes classes de tempo de execução, especialmente no contexto de máquinas determinísticas, nas quais um algoritmo é considerado computacionalmente eficiente quando opera em tempo polinomial no tamanho da entrada. Isso significa que procedimentos eficientes de reconhecimento, em tais máquinas, são alcançáveis por meio de APDs, mas não por APNDs.

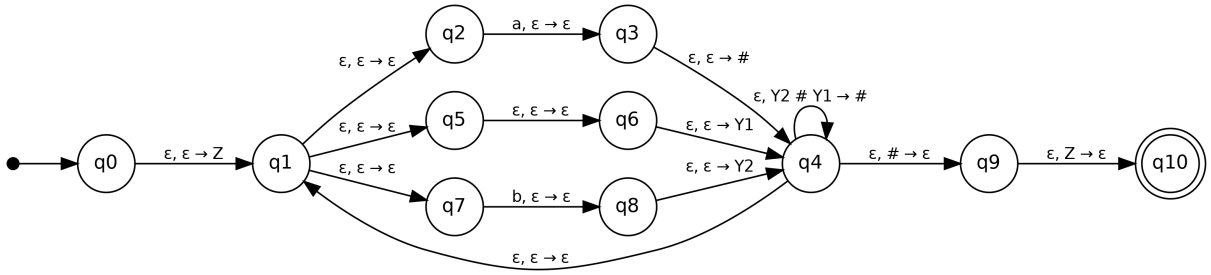
3.2 O problema da parada

A análise do algoritmo apresentado em Oliveira (2024) revelou casos de computação infinita, especificamente situações análogas ao “problema da parada”, em instâncias que envolvem a palavra vazia ε.

3.2.1 O caso base e o ciclo de recursão infinita

Considere a aplicação da proposta original no "caso base" representado pela expressão $a{:}b\#c$. O algoritmo constrói três autômatos independentes (M_a, M_b, M_c), interligados por transições vazias (ε), o que resultado no APND contigo na Figura 14. O controle de fluxo depende de um estado central que gerencia a recursão, retornando ao estado inicial via ε-transição.

Figura 14 – PDA para $a{:}\varepsilon\#b$ (OLIVEIRA, 2024)



O ciclo de recursão infinita ocorre quando a linguagem de qualquer operando do PDOp contém a palavra vazia ($\varepsilon \in L(M_n)$) visto que visto que a aceitação de ε permite que o operador seja aplicado sem consumir entrada nem produzir avanço significativo no estado computacional. Nessa situação, a operação de empilhamento do PDOp pode ser repetida indefinidamente, gerando uma sequência de transições não redutoras que apenas reproduzem o mesmo estado lógico de forma recorrente. Como não há consumo de símbolos nem alteração efetiva da configuração da pilha, o processo não progride e a computação entra em um loop infinito, impedindo a terminação do procedimento de reconhecimento.

Observa-se, nesse cenário, que o autômato resultante executa uma cadeia de transições improdutivas, acumulando símbolos na pilha sem qualquer redução:

$$\text{configuração inicial} \Rightarrow \text{empilhamento} \Rightarrow \text{retorno via } \varepsilon\text{-transição} \Rightarrow \text{novo empilhamento} \Rightarrow \cdots$$

Como o autômato dispõe de uma transição vazia que retorna ao estado inicial para preservar a semântica denotacional do PDOp, e não há consumo da entrada que force o avanço do cabeçote, a máquina permanece nesse ciclo infinito. Além disso, devido ao empilhamento contínuo dentro desse ciclo, ocorre a explosão da pilha (*stack overflow*), agravando o problema.

Tal comportamento viola a propriedade de terminação indispensável a qualquer algoritmo de decisão. Para que um PDA seja funcional, transições ε não redutoras, aquelas que não consomem entrada nem diminuem a altura da pilha, devem ser rigorosamente evitadas ou eliminadas.

3.2.2 Indeterminismo na manipulação da pilha

Outro problema identificado refere-se à gestão das operações de pilha. O algoritmo proposto por Oliveira (2024) não fornece mecanismos para distinguir deterministicamente entre a necessidade de empilhar (operação *push*) ou apenas transitar, nem como tratar conflitos entre leitura de entrada e leitura de pilha (conflitos *shift/reduce* em terminologia de compiladores).

A ausência de tratamento para transições que utilizam simultaneamente a leitura da pilha e a leitura da entrada resulta na geração massiva de não-determinismo. Sendo assim, o que, conforme discutido na Seção 3.1, implica em custos computacionais proibitivos em máquinas determinísticas.

Diante dessas constatações, e visando a transição do campo teórico para aplicações práticas, torna-se evidente a necessidade de uma nova proposta algorítmica eficiente, restrita ao conjunto das linguagens livres de contexto determinísticas. Isso implica que tal proposta deve ser concebida de modo a eliminar recursões improdutivas e a maximizar o determinismo do processo de reconhecimento.

3.3 Nova proposta: caso base e arquitetura hub

Neste trabalho, a nova proposta é delimitada ao "caso base" (SILVA; OLIVEIRA; SANTOS, 2025), definido como uma instância do PDOp cujos operandos imediatos não envolvem múltiplas chamadas nem chamadas alternadas.

3.3.1 O operador pushdown como entidade ternária

Na nova proposta, em vez de tratar o PDOp, em seu caso base, como uma entidade recursiva monolítica, o algoritmo de conversão passa a decompor o processo em uma estrutura linearizada composta por três etapas distintas. Isso significa que no autômato de pilha resultante, o PDOp deixa de atuar como um modificador de fluxo e passa a ser interpretado como uma sequência de operações derivadas de seus operandos, conforme a estrutura ternária presente em seu caso base.

A linguagem denotada pelo PDOp, em seu caso base, é definida como segue (SILVA; OLIVEIRA; SANTOS, 2025):

$$a : b\#c \;=\; \bigcup_{i=0}^{\infty} L(b^i a c^i)$$

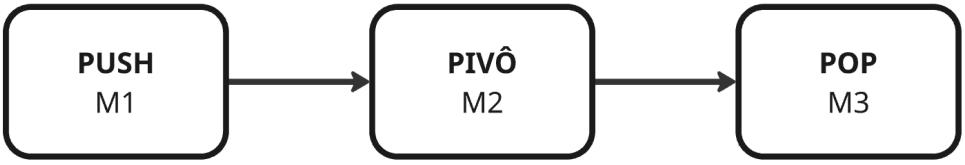
onde a, b, c são expressões regulares ou expressões estendidas pelo PDOp.

Assim, $\Omega(M_1, M_2, M_3)$ pode ser interpretada como a função correspondente ao PDOp em seu caso base, em que M_1, M_2 e M_3 representam os autômatos de pilha que reconhecem os operandos b, a e c, respectivamente. Cada operando tem um papel específico

na construção do autômato resultante:

1. **Operando PUSH (M_1):** Responsável pelo reconhecimento do contexto recursivo (esquerda) e pela geração de símbolos na pilha. Corresponde ao operando b.
2. **Operando PIVÔ (M_2):** Atua como o elemento de transição de contexto, delimitando o fim da fase de empilhamento. Corresponde ao operando a.
3. **Operando POP (M_3):** Responsável pelo reconhecimento da segunda metade da sentença, condicionado ao desempilhamento. Corresponde ao operando c.

Figura 15 – Representação semântica das fases do PDA



Conforme demonstrado pelo diagrama 15, a nova proposta estabelece uma linearidade de fases. Embora o ciclo de empilhamento possa ser iterativo, a transição entre fases é unidirecional.

3.3.2 Redefinição do Fecho de Kleene (Σ^*)

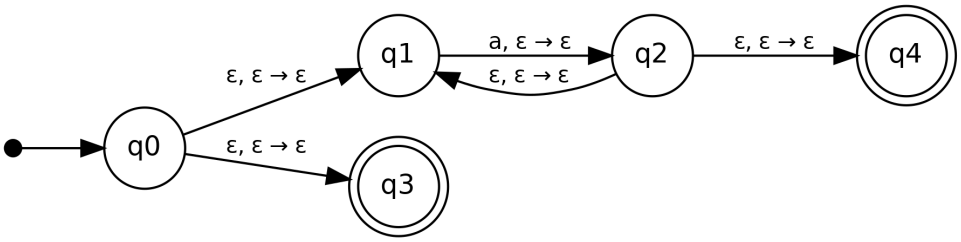
Uma das necessidades desta proposta é a modificação da construção do autômato para o operador Fecho de Kleene ($*$). No algoritmo de Oliveira (2024), derivado do método de Thompson, o autômato obtido para o Fecho de Kleene possui um único estado final. Entretanto, conforme discutido na Seção 3.2.1, o problema de parada ocorre quando a linguagem associada a algum operando contém ε. Para evitar esse comportamento, tornou-se necessário distinguir o estado final do autômato resultante dessa operação em duas categorias, separando o reconhecimento da palavra vazia do reconhecimento das demais cadeias.

Assim, propõe-se que o autômato resultante M^* exponha explicitamente dois conjuntos disjuntos de estados finais:

- **Estado Final ε (F_ε):** Estado dedicado a aceitar exclusivamente a entrada vazia, correspondente ao caso de zero iterações.
- **Estado Final de Leitura (F_{read}):** Conjunto de estados responsáveis por aceitar palavras que envolvem o consumo de pelo menos um símbolo.

A Figura 16 representa o APND para a^*. O estado F_ε é representado por q_3 e o estado F_{read} é representado por q_4.

Figura 16 – Alteração estrutural do Fecho de Kleene



Esta alteração estrutural permite desacoplar a contagem interna do Fecho de Kleene da lógica de empilhamento do operador *pushdown*, garantindo que a ação de empilhar seja binária e baseada no consumo real de entrada.

Contudo, é necessário observar que, nos casos onde o PUSH é definido por um Fecho de Kleene (M^*), embora o algoritmo mitigue ciclos vazios, persistem limitações quanto ao **determinismo da transição de saída**.

A restrição manifesta-se na leitura de símbolos $a \in \Sigma$, onde o autômato carece de capacidade preditiva (*lookahead*) suficiente para discernir se o símbolo corrente compõe uma nova iteração do fecho ou se marca o encerramento da sequência (realizando o empilhamento final antes de transitar para a fase subsequente, o PIVÔ).

Essa incerteza decorre da natureza iterativa do APND na fase de PUSH; isto obriga o autômato a explorar múltiplos caminhos simultaneamente (não-determinismo), levando ao empilhamento de sequências espúrias na pilha nos caminhos incorretos. Consequentemente, tal comportamento degrada a eficiência do reconhecimento e impede a geração de um DPDA puro nestes cenários específicos.

3.3.3 Extensão da redefinição para a união (∪)

A lógica de segregação estende-se ao operador de União. Ao invés de fundir os estados finais dos operandos, o autômato resultante propaga os conjuntos originais:

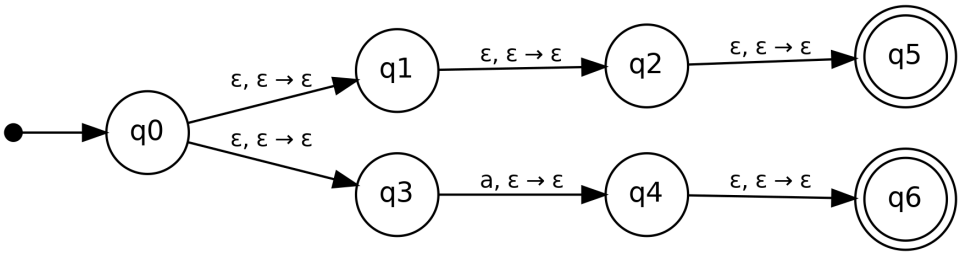
$$F_\varepsilon(M_{uniao}) = F_\varepsilon(M_A) \cup F_\varepsilon(M_B) \tag{2}$$

$$F_{read}(M_{uniao}) = F_{read}(M_A) \cup F_{read}(M_B) \tag{3}$$

Isso preserva a informação topológica necessária para tratar expressões como $a : (b + \varepsilon)\#c$ sem adivinhação não-determinística.

A Figura 17 representa o APND para $(a + \varepsilon)$ o estado F_ε é representado por q_5 enquanto o estado F_{read} é representado por q_6

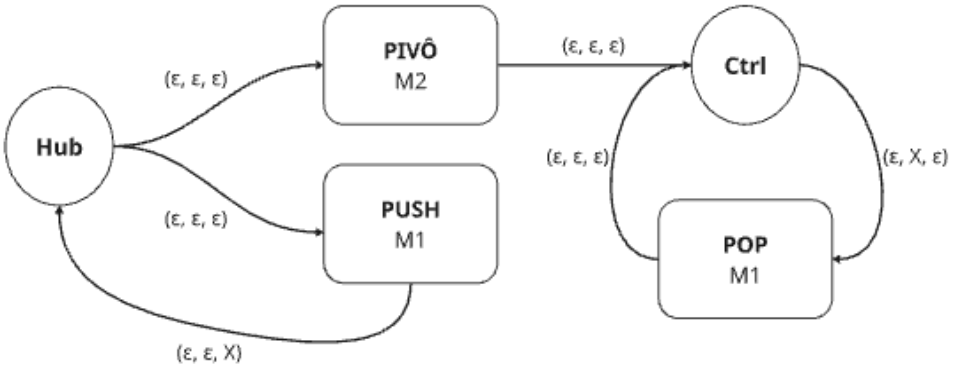
Figura 17 – Propagação de estados na União



3.4 Arquitetura interna

A implementação computacional utiliza uma arquitetura baseada em um estado distribuidor central (denominado "Hub") e um estado gerenciador de desempilhamento ("Estado de Controle"). A estrutura foi projetada para encapsular a lógica de aceitação por pilha vazia, sendo o empilhamento da variável de fundo de pilha realizado antes do estado Hub, dentro de um autômato que opera globalmente por estado final. Para garantir que o esvaziamento da pilha do operador não interfira na pilha global, foi introduzido um marcador de fundo de pilha local (Z) antes do início do estado Hub.

Figura 18 – Diagrama de etapas com Hub e Estado de Controle



A arquitetura do autômato de pilha proposto fundamenta-se no fluxo lógico delineado na Figura 18. Nesta topologia, destaca-se o papel do estado centralizador (*Hub*), responsável pela orquestração do fluxo entre as fases de PIVÔ e PUSH, enquanto o estado de controle (CTRL) governa as operações de desempilhamento (POP).

3.4.1 Definição dos estados e transições

Esse encadeamento sistemático de passos permite organizar o processo de conversão, esclarecer o papel de cada componente do PDOp e assegurar que o autômato resultante seja consistente com a arquitetura proposta.

1. **Inicialização e Isolamento ($q_0 \rightarrow q_{hub}$):** O estado inicial global (q_0) realiza uma

transição vazia para o estado hub (q_{hub}), empilhando o marcador de fundo Z.

$$\delta(q_0, \varepsilon, \varepsilon) = \{(q_{hub}, Z)\}$$

2. **O Distribuidor (q_{hub}):** Este estado atua como um pivô de decisão para a fase de empilhamento. Dele partem duas vias de transição ε:

   - **Via PUSH (M_1):** Conecta-se ao estado inicial de M_1.
   - **Via PIVÔ (M_2):** Conecta-se ao estado inicial de M_2.

3. **Ciclo de Empilhamento (M_1):** Os estados finais de M_1 possuem transições de retorno para q_{hub}. É neste retorno que ocorre a ação semântica de empilhar a variável de controle X.

   $$\forall q_f \in F(M_1), \delta(q_f, \varepsilon, \varepsilon) = \{(q_{hub}, X)\}$$

   Nota de Determinismo: Conforme a redefinição do Fecho de Kleene, apenas os estados finais de M_1 pertencentes ao conjunto F_{read} devem possuir esta transição, evitando o ciclo infinito de empilhamento vazio.

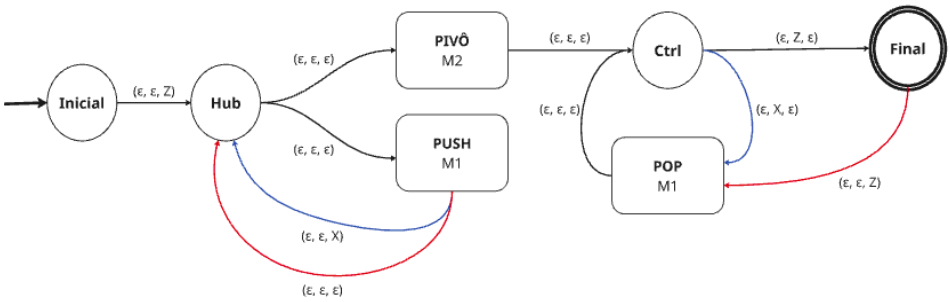
4. **Transição de Fase (M_2):** O autômato M_2 (Pivô) é executado. Seus estados finais conectam-se ao Estado de Controle (q_{ctrl}) por transição ε.

5. **Estado de Controle e Desempilhamento (q_{ctrl}):** Este estado gerencia o consumo da pilha através de um laço com M_3 (POP).

   - **Execução do POP:** Se o topo da pilha contém X, transita para M_3 desempilhando X.
   - **Aceitação:** Se o topo da pilha contém Z, consome Z e aceita (q_{final}).
   - **Caso Sufixo Irrestrito:** Se o PUSH foi vazio, a lógica segue o caminho `INDEPENDENT`. M_3 é executado após a verificação do marcador Z, sendo acessado por uma transição que desempilha e reempilha Z (operação de *Peek*), garantindo a validação do contexto.

Conforme ilustrado na Figura 19, as transições destacadas em vermelho indicam o fluxo correspondente ao caso `INDEPENDENT`, enquanto as transições em azul representam o caso `DEPENDENT` ambos os fluxos estão presentes em casos mistos.

Figura 19 – PDA para caso base reformulado diagrama

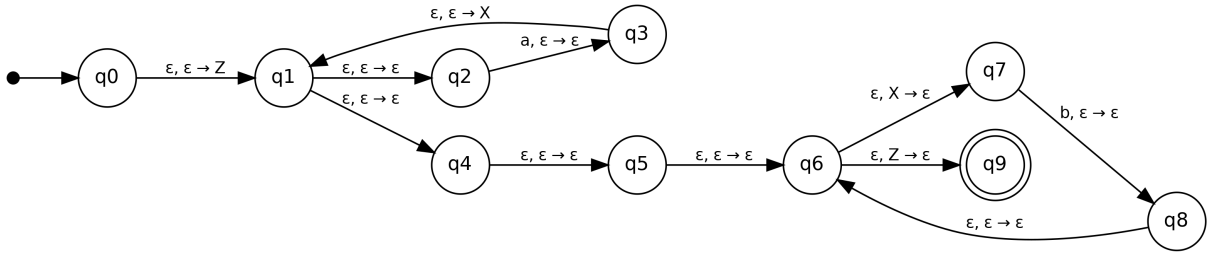


Dado que o PIVÔ funciona estritamente como um mecanismo de transição de fase, sem realizar manipulações diretas na pilha, não há restrições de contagem ou empilhamento associadas a ele. Contudo, há um caso particular referente à propagação de seus estados finais que requer uma restrição arquitetural, a qual será abordada na sequência.

3.4.2 Condição especial para PIVÔ vazio

Assim como no PUSH, se o PIVÔ aceitar vazio, seus estados finais (q_{vazio} e q_{read}) devem propagar essa informação para os estados finais do *pushdown*. Conforme demonstrado na Figura 20, essa condição é verificada no estado inicial, que antecede o empilhamento do marcador de fundo, garantindo que nenhum símbolo ou variável tenha sido processado."

Figura 20 – PDA para $\varepsilon{:}b\#c$



3.5 Lógica de Classificação do Operando PUSH

O fluxo de pilha a ser definido depende estritamente das propriedades de M_1. Classifica-se M_1 em três casos fundamentais baseado na necessidade de empilhamento, apresentados a seguir.

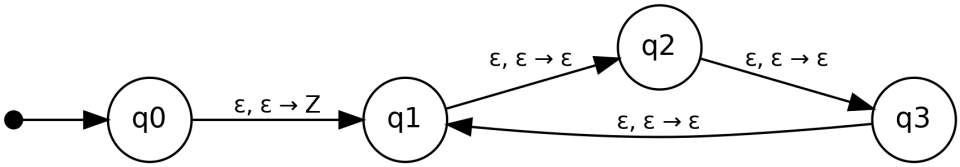
3.5.1 Caso 1: Contagem irrelevante (PUSH nulo)

Ocorre quando $L(M_1) = \{\varepsilon\}$.

- **Conexão:** Estados finais conectam-se ao Hub sem empilhar X conforme demonstrado na Figura 21.

- **Impacto no POP:** Flag `IS_INDEPENDENT` verdadeira.

Figura 21 – Caso 1: PUSH ε

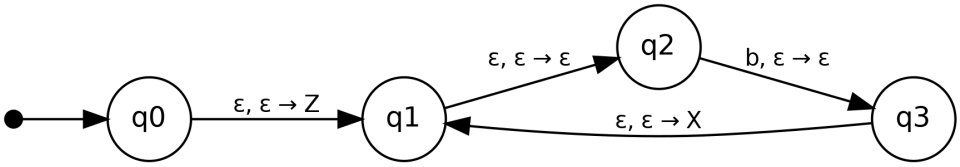


3.5.2 Caso 2: Contagem estrita (PUSH obrigatório)

Ocorre quando $\varepsilon \notin L(M_1)$.

- **Conexão:** Estados finais (F_{read}) retornam ao Hub empilhando X, conforme demonstrado na Figura 22.
- **Impacto no POP:** Flag `IS_DEPENDENT` verdadeira.

Figura 22 – Caso 2: PUSH b



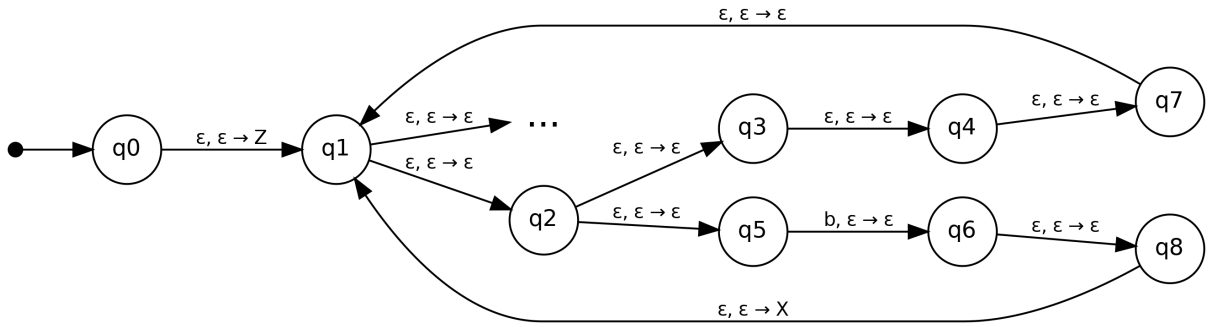
3.5.3 Caso 3: Contagem condicional (PUSH misto)

Ocorre quando $\varepsilon \in L(M_1)$ e $L(M_1) \cap \Sigma^+ \neq \emptyset$.

- **Via F_ε:** Retorna ao q_{hub} via ε **SEM** empilhamento.
- **Via F_{read}:** Retorna ao q_{hub} com ação PUSH(X), conforme demonstrado na Figura 23.
- **Impacto no POP:** O autômato M_3 terá duas vias de execução (Dependente e Independente).

É importante ressaltar que os autômatos ilustrados nas Figuras 21, 22 e 23 representam isoladamente as variações estruturais no operando PUSH (M_1). O restante do APND resultante (PIVÔ e POP) foi omitido nestes diagramas para fins de clareza, estando conectado via transição vazia (ε) a partir do estado q_1. A estrutura completa e a interação entre os componentes restantes serão abordadas detalhadamente na seção seguinte.

Figura 23 – Caso 3: PUSH $(b + \varepsilon)$



3.6 Definição dos componentes PIVÔ e POP

Uma vez estabelecida a interface de saída de M_1 e o comportamento do Hub, os componentes subsequentes são instanciados para completar o fluxo linear de verificação.

3.6.1 O autômato PIVÔ (M_2)

O PIVÔ atua como o delimitador central da expressão (a subexpressão localizada antes de ':', ou seja, primeiro operando do PDOp). **Sua construção é definida pela subexpressão correspondente, podendo esta ser uma expressão regular ou uma outra expressão com PDOp (em casos de aninhamento).**

Apesar de sua complexidade interna variável, no nível atual da arquitetura Hub, ele desempenha o papel de transição de fase. Deve-se ter atenção especial caso a linguagem deste componente aceite a palavra vazia ($\varepsilon \in L(M_2)$). Neste cenário, os estados finais de M_2 devem propagar a distinção entre "leitura realizada" e "leitura vazia" para o Estado de Controle subsequente. Embora o PIVÔ, por definição nesta proposta, não manipule a pilha (apenas transita), essa informação topológica é crucial para que o autômato saiba se houve uma mudança de contexto real ou apenas uma transição nula.

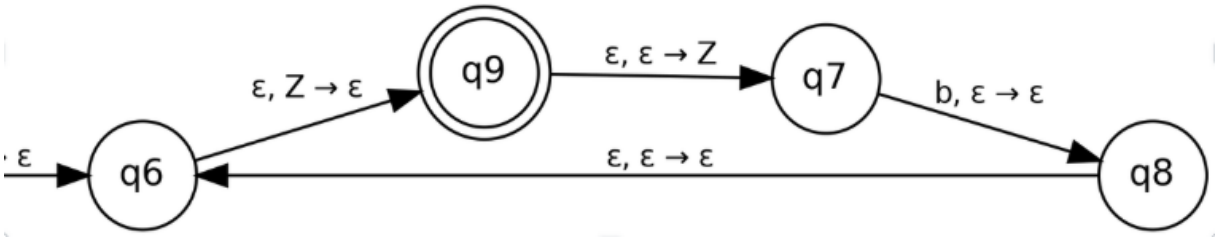
3.6.2 O autômato POP (M_3)

O autômato de desempilhamento é gerado condicionalmente às *flags* definidas pela classificação do PUSH. O Estado de Controle (q_{ctrl}), interface entre PIVÔ e POP, bifurca o fluxo baseando-se na variável presente no topo da pilha (X para contagem ou Z para aceitação).

1. **Se** `IS_INDEPENDENT`**:** O autômato deve permitir o processamento do sufixo mesmo quando a pilha de contagem do PUSH está vazia. Para evitar conflitos de transição no estado de controle (onde a presença de Z indica aceitação), a transição para M_3 é originada a partir do estado final global (q_{final}). Formalmente, o autômato primeiro transita de q_{ctrl} para q_{final} desempilhando Z (aceitação do prefixo). Em seguida,

define-se uma transição de continuidade que reempilha Z e direciona para o início de M_3, conforme demonstrado na Figura 24:

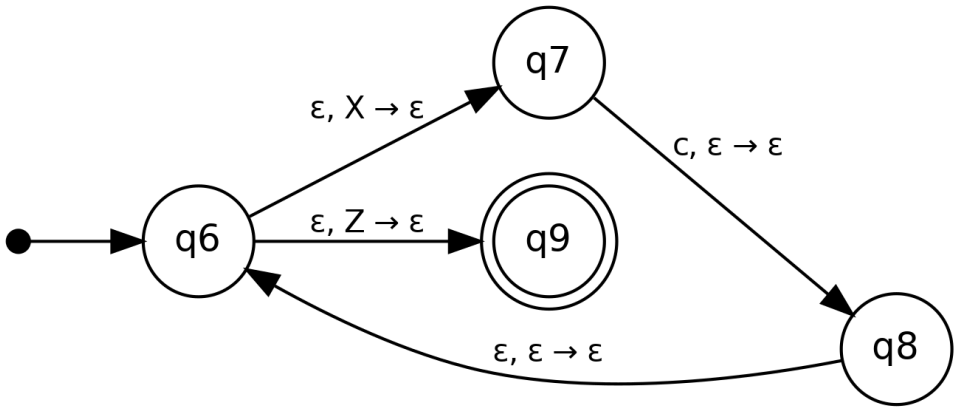
Figura 24 – Caso 1: POP (ε)



Essa abordagem elimina o conflito *Shift/Reduce* no estado de controle, garantindo que a decisão de processar o sufixo ocorra de forma determinística após a validação do contexto anterior. Os estados finais de M_3 retornam com transições ε para q_{ctrl}.

2. **Se** `IS_DEPENDENT`**:** Todas as transições iniciais de M_3 são condicionadas ao consumo da variável de controle. O autômato transita desempilhando efetivamente X, conforme demonstrado na Figura 25:

Figura 25 – Caso 2: POP (b)



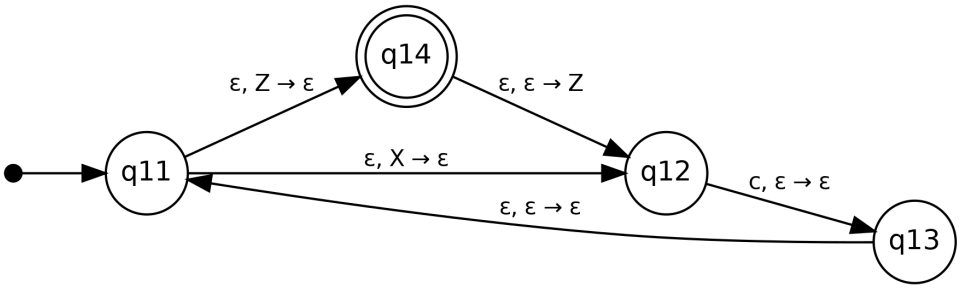
Os estados finais de M_3 retornam com transições ε para q_{ctrl}.

3. **Se** `IS_DEPENDENT/INDEPENDENT`**:** O algoritmo gera duas instâncias (ou caminhos) para M_3, cobrindo ambas as possibilidades lógicas do PUSH:

   - **Caminho A (com pilha):** Acessível se o topo da pilha for X. Segue a lógica `DEPENDENT`, consumindo X e processando a entrada.
   - **Caminho B (sem pilha):** Acessível via estado final (após consumo de Z). Segue a lógica `INDEPENDENT`, permitindo que o POP processe a entrada assumindo que o PUSH optou pelo caminho vazio e a pilha foi esvaziada.

Essas duas transições são aparentes na Figura 26

Figura 26 – Caso 3: POP $(b + \varepsilon)$



Esta estrutura garante que a ambiguidade gerada no PUSH ("empilhou ou não?") seja resolvida deterministicamente no POP através da inspeção do topo da pilha e da segregação dos caminhos de aceitação e continuação.

3.7 Definição operacional dos componentes

A implementação da arquitetura, baseada em um estado distribuidor (q_{hub}), impõe comportamentos específicos aos operandos, conforme explicado a seguir.

3.7.1 O operando PUSH (M_1) como ciclo controlado

Na arquitetura proposta, o PUSH não transita diretamente para o PIVÔ. Sua função é gerar tokens de empilhamento e retornar ao distribuidor para permitir novos empilhamentos ou a transição de fase.

- **Entrada:** Acessado via ε a partir de q_{hub}.
- **Saída:** Estados finais de F_{read} retornam a q_{hub} empilhando X.

3.7.2 O operando PIVÔ (M_2) como transição de fase

O PIVÔ é o mecanismo de escape do ciclo.

- **Entrada:** Acessado via ε a partir de q_{hub}.
- **Saída:** Conecta-se ao Estado de Controle, selando a fase de empilhamento.

3.7.3 O operando POP (M_3) como validador de Sufixo

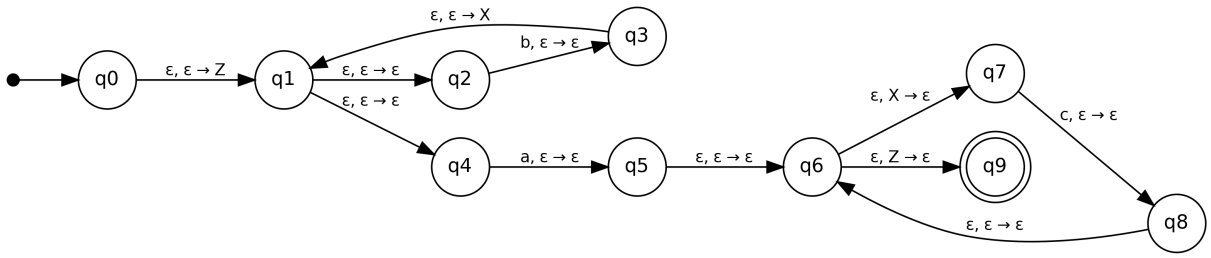
O POP opera sob a gestão estrita do Estado de Controle (q_{ctrl}), funcionando como um consumidor de contexto.

- **Entrada:** Acessado a partir de q_{ctrl} mediante o desempilhamento da variável de controle X (ou validação de Z).

- **Saída:** Seus estados finais retornam a q_{ctrl}, permitindo o processamento iterativo (desempilhamento em laço) até que o marcador de fundo Z seja exposto para a aceitação final.

Com a definição do novo operador *pushdown*, a expressão $a{:}b\#c$ resulta na estrutura apresentada na Figura 27.

Figura 27 – Exemplo de PDA preliminar gerado para $a{:}b\#c$



Embora o autômato ilustrado na Figura 27 ainda seja classificado formalmente como um APND, devido às transições vazias de conexão, a arquitetura segregada por fases proposta neste trabalho é o fator chave que viabiliza a sua determinização. O capítulo a seguir detalhará o *pipeline* de otimização desenvolvido para eliminar essas ambiguidades estruturais e converter o modelo em um APD.

4 OTIMIZAÇÃO E MAXIMIZAÇÃO DO DETERMINISMO

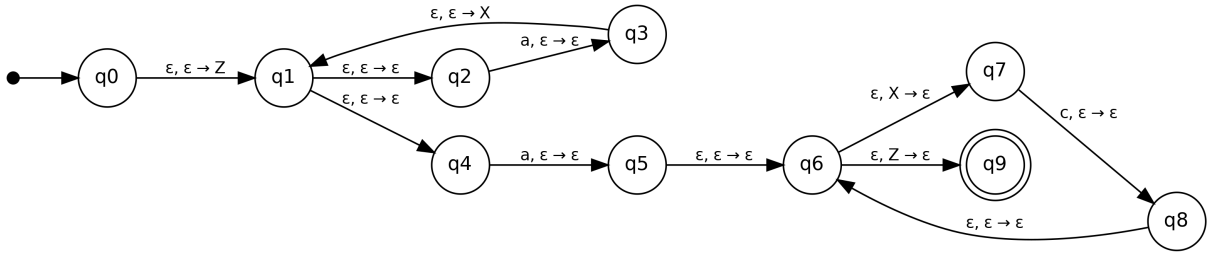
A arquitetura descrita anteriormente gera um Autômato de Pilha funcional, porém não-determinístico por causa do uso extensivo de transições vazias para conectar os componentes.

Visto que a conversão de um APND para um APD não é possível para todas as linguagens livres de contexto, dado que algumas CFL não admitem reconhecimento determinístico, seja por apresentarem ambiguidade inerente, seja por exigirem decisões não locais, este trabalho propõe um *pipeline* de otimização voltado à eliminação do não determinismo estrutural para o caso base do PDOp.

Importante ressaltar que, ao final de cada etapa de otimização descrita a seguir, executa-se um algoritmo auxiliar de limpeza para a remoção de **estados órfãos** (*orphan states*), isto é, estados a partir dos quais não existe qualquer caminho que conduza a um estado final. Essa limpeza garante que o espaço de busca das etapas subsequentes seja minimizado, concentrando-se apenas nos estados efetivamente produtivos.

Para exemplificar a aplicação do pipeline, é utilizado o PDA gerado pela nova proposta para a expressão $a:a\#c$ demonstrado na Figura 28. Trata-se de um exemplo particularmente interessante, pois a linguagem correspondente não é *prefix-free*, isto é, trata-se de uma linguagem em que algumas palavras podem ser prefixo de outras, o que dificulta a determinação determinística do ponto de aceitação. Por essa razão, um DPDA só é obtido após a execução completa de todas as etapas do pipeline.

Figura 28 – PDA Resultante para $a:a\#c$



4.1 Etapa 1: Cálculo e remoção de transições ε não-Redutoras

Uma das principais fontes de não determinismo em autômatos gerados composicionalmente é a presença de transições silenciosas que apenas ajustam o estado interno ou manipulam a pilha sem consumir símbolos da entrada. Conforme a definição clássica de Ginsburg e Greibach (1966):

Definição 1 (Transição ε não-redutora)**.** *Seja* $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ *um autômato de pilha. Uma transição* ε *da forma* $\delta(q, \varepsilon, X) \subseteq Q \times \Gamma^*$ *é dita* não-redutora *quando, para todo par* $(p, \alpha) \in \delta(q, \varepsilon, X)$*, tem-se* $|\alpha| \geq |X|$*.*

Essas transições podem ser classificadas em dois tipos para fins de algoritmo:

- **Tipo 1 (ε-Transição pura):** $(q, \varepsilon, \varepsilon) \to (p, \varepsilon)$. Apenas altera o estado.
- **Tipo 2 (Empilhamento oculto):** $(q, \varepsilon, \varepsilon) \to (p, X)$. Empilha sem ler entrada.

As transições do **Tipo 2** representam um desafio crítico para a determinização. Por permitirem a manipulação da pilha sem o consumo de símbolos da entrada, elas introduzem o risco de crescimento não-limitado da pilha. A presença de ciclos formados exclusivamente por essas transições geraria laços infinitos impossíveis de serem removidos sem alterar a semântica da linguagem.

Diante disso, e assumindo que a construção estrutural do PDOp impede a formação de ciclos de empilhamento vazio, aplica-se uma estratégia de fechamento transitivo modificado. O método visa identificar sequências acíclicas de transições vazias e eliminá-las através da criação de atalhos que 'propaga' o consumo de entrada para os estados anteriores. Essa abordagem expande o conceito clássico de fecho-ε (comum em autômatos finitos) para o contexto de pilha, com a restrição fundamental de acumular e preservar as operações de empilhamento na nova transição direta, garantindo assim a equivalência da linguagem reconhecida.

O procedimento baseia-se na identificação do conjunto de alcance $Reach(q)$ através de caminhos ε. A regra de propagação define que, se $q \xrightarrow{\varepsilon} p$ e $p \xrightarrow{a} r$, cria-se a transição direta $q \xrightarrow{a} r$. O Algoritmo 1 detalha a implementação iterativa deste processo:

Algoritmo 1 Eliminação de Transições ε não-redutoras

Requisitos: PDA $M = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$

Garante: PDA M' otimizado

$M' \leftarrow M$
repita
  para todo Estado $q \in Q$ **faça**
    ▷ Calcula alcance por transições ε não-redutoras (sem ciclos)
    $R \leftarrow$ BuscaProfundidadeTransicoesRuins(q)
    para todo Estado $p \in R$ **faça**
      se $p \neq q$ **então**
        para todo Transição Boa $t : (p, a, \alpha) \rightarrow (r, \beta)$ em M **faça**
          ▷ Cria atalho direto preservando operação de pilha
          Adicionar $\delta'(q, a, \alpha) \rightarrow (r, \beta)$ em M'
        fim para
        se $p \in F$ **então**
          Adicionar q a F' (Herança de aceitação)
        fim se
      fim se
    fim para
  fim para
até não houver novas adições em M'
Remover transições ruins originais onde possível (sem desconectar o grafo)

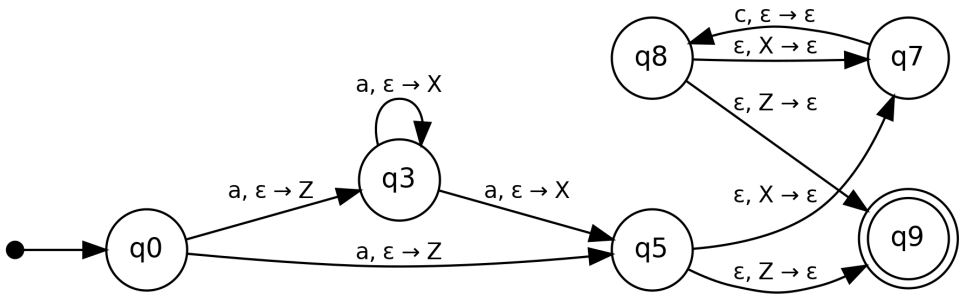
É importante ressaltar que o Algoritmo 1 pressupõe que não existam ciclos de transições puramente do Tipo 2 (empilhamentos infinitos sem leitura), o que causaria divergência. A garantia de convergência é dada pelas propriedades construtivas do PDOp: a redefinição do Fecho de Kleene (Seção 3.3.2) e da União (3.3.3) mitiga a formação desses ciclos na estrutura base.

A aplicação da Etapa 1 à expressão $a : a \# c$ resulta no autômato apresentado na Figura 29, onde caminhos intermediários foram substituídos por transições diretas de leitura.

Observa-se que, em casos onde o componente PUSH é nulo (ex: $a : \varepsilon \# b$), a Etapa 1 é capaz de remover integralmente a fase de empilhamento. Nessa configuração, o autômato reduz-se funcionalmente ao reconhecimento de uma linguagem regular (equivalente a ab^*). No entanto, como a arquitetura preserva o critério de aceitação por pilha vazia, mantém-se uma manipulação de pilha redundante. Isso introduz um custo computacional desnecessário quando comparado à eficiência de um AFD nativo.

Contudo, mesmo após essa otimização estrutural, persistem situações de não-determinismo residual. Estes casos manifestam-se, sobretudo, como conflitos de decisão

Figura 29 – PDA Resultante da Etapa 1 para $a{:}a\#c$



entre o consumo efetivo de um símbolo da entrada e a execução de uma operação de desempilhamento (transição ε-POP).

4.2 Etapa 2: Tratamento de transições ε-POP

Definição 2 (Conflito ε-POP)**.** *Um conflito ε-POP ocorre quando um autômato possui, para um mesmo estado q, uma transição que desempilha sem ler entrada ($\delta(q, \varepsilon, X)$) e outra que consome entrada ($\delta(q, a, \dots)$), criando uma ambiguidade do tipo Shift/Reduce (AHO et al., 2007).*

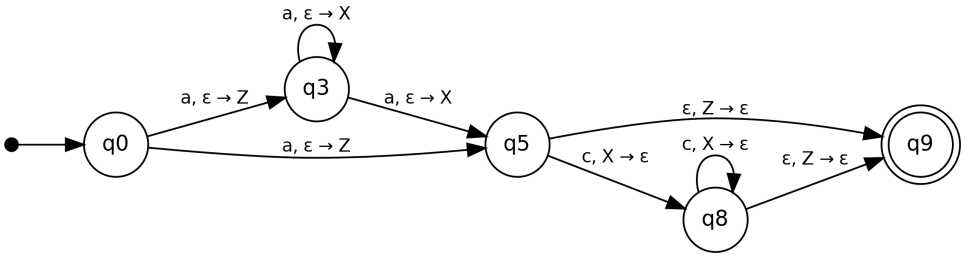
A estratégia adotada é antecipar o desempilhamento para o momento da leitura. Se o autômato desempilha X e *logo em seguida* lê 'a', fundem-se essas operações:

$$\text{Se } (q, \varepsilon, X) \to p \text{ e } (p, a, \varepsilon) \to r \implies \text{Criar } (q, a, X) \to r$$

Esta otimização permite que o autômato de pilha tome a decisão baseando-se no *lookahead* do símbolo 'a', removendo a necessidade de "adivinhar" o momento do POP. Utiliza-se a mesma estratégia de propagação da Etapa 1, porém considerando as transições ε-pop como o alvo da eliminação.

A aplicação da Etapa 2 à expressão $a{:}a\#c$ resulta no autômato apresentado na Figura 30.

Figura 30 – AP Resultante da Etapa 2 para $a{:}a\#c$



Contudo, mesmo após essa otimização, persistem situações de não-determinismo estrutural. Essas situações ocorrem quando o autômato apresenta ambiguidade de transi-

ção por entrada, ou seja, existem dois ou mais caminhos distintos que consomem o mesmo símbolo a partir do mesmo estado ($\delta(q, a, Z)$ possui múltiplos destinos).

Esse cenário impõe a necessidade de observar a Restrição de Disjunção de Símbolos (LL(1)). Para que o autômato resultante seja estritamente determinístico (DPDA), é mandatório que, em qualquer ponto de bifurcação do fluxo de controle, os conjuntos de símbolos iniciais ($FIRST$) de cada caminho alternativo sejam disjuntos ($FIRST(Caminho_A) \cap FIRST(Caminho_B) = \emptyset$). Caso contrário, a máquina exigiria capacidade de *lookahead* (antevisão) superior a 1 ou *backtracking* para resolver a decisão.No entanto, há uma exceção tratável: casos onde a leitura de um mesmo símbolo leva a múltiplas transições isomorfas, isto é, transições que consomem o mesmo símbolo da entrada, desempilham a mesma variável e empilham a mesma sequência, diferindo apenas no estado de destino. Esse tipo específico de não-determinismo (redundância) não viola a lógica da linguagem e pode ser resolvido através da fusão de estados (*superestados*), conforme demonstrado na etapa a seguir.

4.3 Etapa 3: Redução de ambiguidade via superestados

Mesmo após a remoção de transições vazias não-redutoras e ε-POP, o autômato resultante pode apresentar não determinismo local. Isso ocorre quando uma mesma configuração de entrada (estado atual, símbolo lido e topo de pilha) aciona múltiplas transições distintas:

$$\delta(q, a, X) = \{(p_1, \gamma), (p_2, \gamma), \ldots, (p_n, \gamma)\} \tag{4}$$

Note que, neste caso restrito, a operação de pilha (γ) é idêntica para todas as opções de transição, diferindo apenas nos estados de destino (p_i). Para resolver esse conflito, propõe-se a criação de Superestados.

Visto que a manipulação de memória é consistente entre os caminhos, os destinos $\{p_1, \ldots, p_n\}$ podem ser unificados em uma nova entidade única Q_{super}, eliminando a bifurcação sem alterar a linguagem reconhecida.

A execução do Algoritmo 2 reduz significativamente o grau de não-determinismo em expressões complexas (como $ab : ac\#d$), mas sua aplicação é estritamente restrita aos casos onde as operações de manipulação da pilha (símbolo desempilhado e sequência empilhada) são idênticas para todas as transições em conflito. Quando as operações de pilha divergem, a fusão é impossível, restando a análise estrutural das fronteiras entre os componentes do operador.

4.3.1 Análise de fronteiras e condições de determinismo

Existe uma relação crítica entre a garantia de determinismo e o comprimento relativo das sequências de símbolos do PUSH (u) e do PIVÔ (v). A ambiguidade residual

Algoritmo 2 Criação de superestados (determinização parcial)

Requisitos: PDA M' após remoção de transições ε
Garante: PDA M'' com redução de ambiguidade

enquanto Houver conflito determinístico compatível **faça**
    Identificar tripla (q, a, X) que gera destinos múltiplos $P = \{p_1, \ldots, p_n\}$ com mesma operação de pilha γ
    Criar novo Superestado $S_{new} \leftarrow \bigcup P$
    Substituir as transições originais por $\delta(q, a, X) \rightarrow (S_{new}, \gamma)$
    Para cada símbolo b e topo Z, as transições saindo de qualquer $p \in S_{new}$ passam a sair de S_{new}
    se houver conflito nas novas saídas **então**
        marcar para próxima iteração
    fim se
fim enquanto

concentra-se nas fronteiras de decisão onde essas fases se sobrepõem. Identificam-se dois cenários principais que impõem restrições de disjunção ($LL(1)$) específicas, apresentados a seguir.

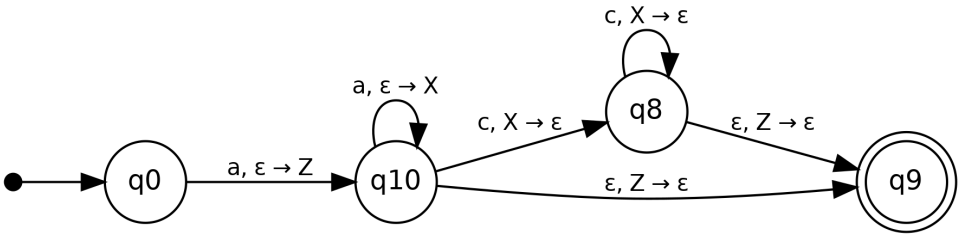
1. **Cenário 1 - PUSH sobrepõe o PIVÔ ($|u| \geq |v|$).** Nesta configuração, a leitura do PIVÔ encerra-se enquanto a sequência de PUSH ainda está sendo processada (ou termina simultaneamente). A ambiguidade surge na decisão entre continuar empilhando o restante de u ou transitar para o desempilhamento (POP).*Exemplo:* Na expressão $aba{:}a\#b$, o pivô a é prefixo do push aba. Ao ler a, o autômato enfrenta um conflito: deve continuar o empilhamento (lendo o próximo b) ou aceitar o pivô? Para garantir o determinismo, o conjunto $FIRST$ da continuação do PUSH deve ser disjunto do $FIRST$ do autômato subsequente (POP). No caso de $a : a\#a$, a interseção dos símbolos (a vs a) impede a decisão determinística sem *lookahead*.

2. **Cenário 2 - PIVÔ estende o PUSH ($|v| > |u|$).** Aqui, a fase de empilhamento termina estritamente dentro da leitura do pivô. O conflito manifesta-se na distinção entre o símbolo que dá continuidade ao PIVÔ e o símbolo que reiniciaria um novo ciclo de PUSH (em casos de iteração).
   Exemplo: Na expressão $aa{:}a\#b$, o PUSH (a) é um prefixo próprio do PIVÔ (aa). Após ler o primeiro a, a máquina deve ler o segundo a para completar o pivô. Se houver um laço externo permitindo o reinício, esse segundo a torna-se ambíguo (parte do pivô atual ou início de um novo push?). A condição de determinismo exige que o símbolo na posição $|u| + 1$ do PIVÔ seja distinto do símbolo inicial do PUSH. A expressão $bab{:}b\#c$ é determinística pois o sufixo do pivô (a) difere do reinício (b).

A otimização de superestados resolve o não-determinismo de redundância, mas a Ambiguidade de Fronteira descrita acima exige que as linguagens componentes respeitem a disjunção de símbolos nos pontos de transição de fase.

Após a aplicação desta etapa e respeitadas as restrições de fronteira, obtém-se o autômato otimizado (Figura 31).

Figura 31 – PDA Otimizado via Superestados para $a{:}a\#c$



Contudo, observa-se que o autômato resultante ainda não se qualifica como um APD. Tal fato decorre da natureza da linguagem denotada por $a{:}a\#c$, que não possui a propriedade de ser livre de prefixo (*prefix-free*). Isso impede que o autômato determine o ponto de aceitação unicamente pela leitura da entrada, exigindo um tratamento específico para a detecção de fim de cadeia, conforme abordado na etapa subsequente.

4.4 Etapa 4: Tratamento de linguagens Não Livre de Prefixo (*Non–Prefix-Free Languages*)

Uma limitação estrutural crítica na geração de APDs com aceitação por pilha vazia reside na propriedade de *Prefix-Freeness*. O operador *pushdown* utiliza o consumo do marcador de fundo de pilha como critério de aceitação. Contudo, se a linguagem L contiver palavras u e v tais que u é prefixo de v (ex: a e aac em $a{:}a\#c$), o critério de pilha vazia gera um conflito insolúvel sem *lookahead* infinito.

Para contornar esta limitação e viabilizar o determinismo, adota-se a transformação da linguagem através de um marcador de fim de sentença (*End-of-File Marker*):

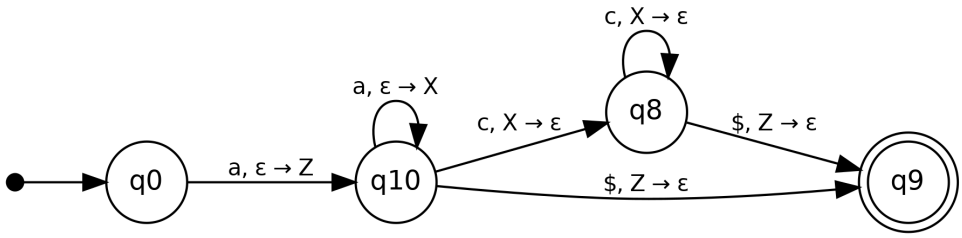
$$L' = L \cdot \{\$\}$$

A transição final de aceitação é alterada para exigir a leitura explícita deste marcador:

$$\delta(q_{ctrl}, \$, Y) \rightarrow (q_{final}, \varepsilon) \tag{5}$$

Desta forma, a ação destrutiva de remover o fundo de pilha torna-se condicionada a um símbolo concreto (\$), eliminando a ambiguidade de parada. O uso do símbolo \$ também auxilia no tratamento de transições pós-estado final. Como o símbolo marca inequivocamente o fim da entrada válida, quaisquer transições definidas após o consumo de \$ representam estados de erro ou extensões da linguagem que não interferem no reconhecimento do prefixo válido já aceito.

Figura 32 – PDA Final com Marcador EOF para $a{:}a\#c$



Após a aplicação de todas as etapas de geração do AP e do *pipeline* de otimização, obtém-se o autômato apresentado na Figura 32. Observa-se que o AP gerado para a expressão $a : a\#c$ opera de forma estritamente determinística, superando as limitações iniciais de definição do pivô e de prefixo. Este resultado evidencia a robustez da proposta de conversão e a eficácia das etapas de determinização, especialmente o uso do marcador de fim de sentença para resolução de conflitos.

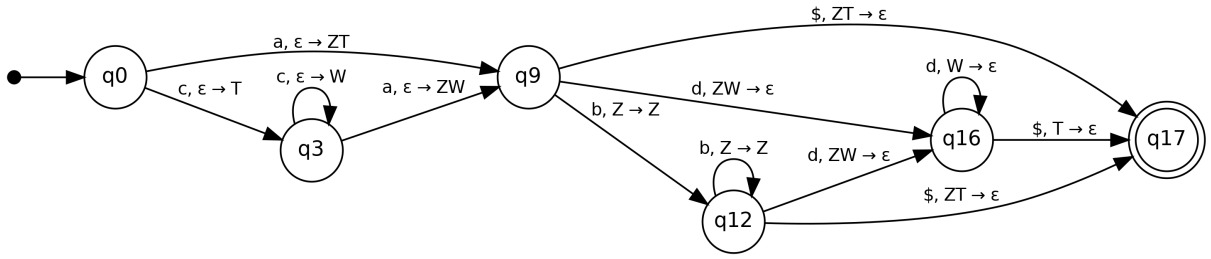
Entretanto, é imperativo ressaltar que a capacidade de gerar um APD não é universal. As limitações observadas em casos mais complexos não decorrem de deficiências algorítmicas, mas refletem as imposições matemáticas intrínsecas à classe das LLCDs. Conforme fundamentado na teoria de autômatos, a classe das LLCDs não é fechada sob operações de concatenação, união, nem sob o Fecho de Kleene. Consequentemente, a composição de duas expressões L_1 e L_2, ainda que individualmente determinísticas, pode resultar em uma linguagem inerentemente ambígua ou que exija mecanismos de *lookahead* superiores à capacidade de um APD padrão.

4.4.1 Análise de dependência em composições aninhadas

Essas restrições teóricas manifestam-se frequentemente em estruturas de recursão aninhada, onde o determinismo local não garante o determinismo global. Um exemplo notável ocorre na expressão $c : (a : a\#b)\#d$. Embora o componente interno $a : a\#b$ e o externo $c : ...\#d$ gerem APDs isoladamente, a sua composição resulta em um APND. Isso ocorre porque a propagação de transições, que atua como mecanismo de antevisão (*lookahead*) no componente interno, perde eficácia ao lidar com a recursão quando esta é utilizada na fase de empilhamento (PUSH) de um operador externo. O contexto de retorno do autômato interno entra em conflito com o fluxo de controle do autômato externo. Outro cenário elucidativo é a expressão $(a : \varepsilon\#b) : c\#d$. Neste caso, a geração de um APD é bem-sucedida, conforme demonstrado na Figura 33, devido à disjunção dos símbolos de fronteira: a transição de saída do componente interno consome b, enquanto a transição subsequente do externo aguarda d. Como $b \neq d$, não há disputa de consumo ($FIRST(POP_{interno}) \cap FIRST(POP_{externo}) = \emptyset$). Contudo, a estabilidade desse determinismo é frágil: se o símbolo de continuidade externa fosse alterado para b (ex: $(a : \varepsilon\#b) : c\#b$), o sistema degradaria imediatamente para um APND devido à

ambiguidade de consumo na fronteira entre os operadores.

Figura 33 – PDA final com chamada aninhada para $(a{:}\varepsilon\#b){:}c\#d$



5 CONSIDERAÇÕES FINAIS

Motivado pela necessidade de eficiência computacional, este trabalho dedicou-se a investigar restrições formais e estruturais aplicáveis às expressões regulares clássicas estendidas com o PDOp, denominadas expressões livres de contexto (SILVA; OLIVEIRA; SANTOS, 2025), com o objetivo de verificar, sob uma perspectiva operacional, em que condições sua interpretação resulta na construção de APDs.

Inicialmente, foram identificadas as causas que levam o algoritmo de conversão de expressões livres de contexto com PDOp em APND, proposto por Oliveira (2024), a enfrentar problemas de parada decorrentes de ciclos vazios. Para tratar essa limitação e analisar a viabilidade operacional da obtenção de APDs a partir de expressões com PDOp, a presente pesquisa delimitou-se ao estudo do caso base do operador.

A partir dessa análise, formulou-se uma nova proposta de conversão, fundamentada no conceito de Estado Hub e na reformulação topológica do autômato de pilha associado ao operador Fecho de Kleene no processo de conversão. Os resultados obtidos indicam que essa abordagem constitui uma estratégia operacionalmente viável para o reconhecimento da subclasse das LLCDs, respeitando o tempo linear de processamento $O(n)$.

A análise dos resultados não configura uma prova dedutiva no sentido formal clássico, mas sim uma **prova operacional**, baseada na construção explícita dos autômatos, na análise estrutural das transições e na validação empírica do comportamento determinístico obtido. Nesse contexto, embora a conversão de uma expressão livre de contexto com PDOp arbitrária para um APD seja indecidível no caso geral, este trabalho contribui ao delimitar subconjuntos bem definidos de expressões para os quais foram encontradas **fortes evidências da obtenção de APDs por construção**.

As evidências observadas indicam que, sob restrições formais específicas, o determinismo emerge como propriedade estrutural do processo de conversão. Essas restrições são sintetizadas a seguir:

- Eliminação do problema da parada – A arquitetura de Estado Hub, combinada à separação dos estados finais do Fecho de Kleene e da União (F_ε e F_{read}), mostrou-se eficaz na prevenção de recursões à esquerda não-redutoras. Essa organização assegura que ciclos de empilhamento ocorram estritamente mediante o consumo de símbolos de entrada.

- Tratamento da ambiguidade – A classificação do operando PUSH permitiu mapear fontes estruturais de não determinismo. A identificação da denominada *Ambiguidade de Fronteira* estabeleceu o critério de que o determinismo é preservado apenas quando os conjuntos de símbolos nas fronteiras de sobreposição entre PUSH e PIVÔ são disjuntos, isto é, quando $FIRST(u) \cap FIRST(v) = \emptyset$.

- Definição estrita de aplicabilidade – O *pipeline* proposto fornece fortes evidências da geração de APDs se, e somente se, as seguintes condições forem satisfeitas:

  1. A linguagem for livre de prefixo, assegurada por meio de marcador de fim de cadeia (\$);
  2. As transições concorrentes forem disjuntas em relação ao símbolo de entrada;
  3. Os conflitos de empilhamento forem estruturalmente isomorfos, permitindo sua resolução por meio de Superestados.

  Essa caracterização estabelece uma fronteira operacional de decidibilidade do algoritmo. Nos casos em que tais condições não são atendidas, o sistema degrada de forma controlada para um APND otimizado.

- Robustez do *pipeline* – O algoritmo de conversão consolidou-se como uma diretriz sintática operacionalmente consistente. A experimentação com casos clássicos e de borda forneceu evidências empíricas de que, respeitadas as restrições estabelecidas, o autômato resultante mantém comportamento determinístico estável durante a execução.

5.1 Limitações da pesquisa

Apesar dos avanços alcançados, reconhecem-se as seguintes limitações desta pesquisa, as quais delimitam o escopo dos resultados obtidos e indicam direções potenciais para investigações futuras.

- Restrição ao caso base - A solução foca na linearização de uma instância única do operador. A interação entre múltiplos operadores *pushdown* aninhados profundos ou concatenados pode introduzir ambiguidades que o atual *pipeline* não resolve sem heurísticas adicionais.

- Antevisão limitada ($k = 1$) - O algoritmo fundamenta suas decisões de otimização na análise do símbolo imediato. Conflitos que exijam *lookahead* $k > 1$ não são resolvidos pela atual implementação.

- Fronteiras das LLCDs - O algoritmo não gera APDs para linguagens inerentemente ambíguas. Nestes casos, a ferramenta mantém a complexidade assintótica típica dos APNDs.

- Eficiência de estados - O foco do algoritmo é a maximização do determinismo, e não a minimização do número de estados, podendo resultar em autômatos com redundâncias estruturais.

- Dependência de marcadores - A garantia da propriedade *prefix-free* depende do uso de marcadores de fim de cadeia (\$).

5.2 Trabalhos futuros

A consolidação deste modelo abre caminhos para investigações subsequentes:

- Investigar a extensão do modelo para contemplar configurações além do caso base (incluindo chamadas múltiplas e chamadas alternadas), refinando os algoritmos para maximizar a geração de APDs.
- Investigar algoritmos de *lookahead* dinâmico ou análise de fluxo global para resolver ambiguidades em aninhamentos complexos, superando a limitação local de $k = 1$.
- Desenvolver uma biblioteca de software que implementa o algoritmo proposto, permitindo que desenvolvedores utilizem a sintaxe PDOp para gerar soluções em APDs e exportar o código resultante (em C, Python ou Java).
- Desenvolver uma biblioteca de software ou de uma ferramenta de linha de comando, análoga ao YACC, que implemente o algoritmo aqui proposto. Essa ferramenta deverá permitir que desenvolvedores utilizem a sintaxe do PDOp e exportem o código do analisador gerado (em C, Python ou Java).
- Realizar estudos empíricos comparando a eficiência dos analisadores gerados pelo método determinístico baseado no PDOp com a de geradores tradicionais de parsers (como YACC/Bison e ANTLR) em cenários reais de compilação.
- Criar uma ferramenta visual para renderizar os autômatos gerados, destacando as fases (PUSH, PIVÔ, POP) e os estados de controle, de modo a auxiliar tanto no ensino quanto na depuração de linguagens livres de contexto.

REFERÊNCIAS

AHO, A. V. Nested stack automata. **Journal of the ACM**, ACM, v. 16, n. 3, p. 383–406, 1972.

AHO, A. V. et al. **Compilers: Principles, Techniques, & Tools**. 2. ed. Boston, MA: Pearson Addison-Wesley, 2007. 159–163 p. ISBN 9780321486813.

AHO, A. V.; SETHI, R.; ULLMAN, J. D. **Compilers: Principles, Techniques, and Tools**. Reading, MA: Addison-Wesley, 1986.

AHO, A. V.; ULLMAN, J. D. **The Design and Analysis of Computer Algorithms**. Reading, MA: Addison-Wesley, 1974.

CHOMSKY, N. Three models for the description of language. **IRE Transactions on Information Theory**, v. 2, p. 113–124, 1956.

EARLEY, J. An efficient context-free parsing algorithm. **Communications of the ACM**, ACM, v. 13, n. 2, p. 94–102, 1970.

GINSBURG, S.; GREIBACH, S. A. Bounded algol-like languages. **Transactions of the American Mathematical Society**, AMS, v. 118, p. 353–389, 1966.

HOPCROFT, J. E.; MOTWANI, R.; ULLMAN, J. D. **Introduction to Automata Theory, Languages, and Computation**. 3. ed. Upper Saddle River, NJ: Pearson, 2006. ISBN 9780321455369.

HOPCROFT, J. E.; ULLMAN, J. D. **Introduction to Automata Theory, Languages, and Computation**. [S.l.]: Addison-Wesley, 1979.

HRUŠA, V. **Regular Expressions with Subpattern Recursion**. Dissertação (Mestrado) — Czech Technical University in Prague, Prague, 2021.

KNUTH, D. E. On the translation of languages from left to right. **Information and Control**, Academic Press, v. 8, n. 6, p. 607–639, 1965.

KNUTH, D. E. On the translation of languages from left to right. **Information and Control**, Elsevier, v. 8, n. 6, p. 607–639, 1965.

KOZEN, D. C. **Automata and Computability**. New York, NY: Springer, 1997. ISBN 9780387949079.

LINZ, P. **An Introduction to Formal Languages and Automata**. 6th. ed. [S.l.]: Jones & Bartlett Learning, 2016. ISBN 9781284077241.

MENEZES, P. B. **Linguagens Formais e Autômatos**. [S.l.]: Sagra Luzzatto, 2000.

OLIVEIRA, G. da C. **Pushdown Operator: An Operator Proposal to Extend Regular Expressions to Denote Context-Free Languages**. Dissertação (Mestrado) — Universidade Federal do Tocantins, Palmas, TO, 2024. Bachelor Thesis, advisor: Alexandre Tadeu Rossini da Silva; co-advisor: Tanilson Dias dos Santos.

SILVA, A. T. R. da; OLIVEIRA, G. da C.; SANTOS, T. D. dos. Introducing the pushdown operator: A proposal to extend regular expressions to context-free languages. In: **Anais do X Encontro de Teoria da Computação**. [S.l.: s.n.], 2025.

SIPSER, M. **Introduction to the Theory of Computation**. 2. ed. Boston, MA: Thomson Course Technology, 2006.

SIPSER, M. **Introduction to the Theory of Computation**. 2. ed. Boston: Thomson Course Technology, 2006.

TURING, A. M. On computable numbers, with an application to the entscheidungsproblem. **Proceedings of the London Mathematical Society**, s2-42, n. 1, p. 230–265, 1936.

TURING, A. M. On computable numbers, with an application to the entscheidungsproblem. **Proceedings of the London Mathematical Society**, Oxford University Press, s2-42, n. 1, p. 230–265, 1936.