



UNIVERSIDADE FEDERAL DO PARÁ
CAMPUS DE TUCURUÍ
GRADUAÇÃO EM ENGENHARIA DE COMPUTAÇÃO

CAMILLE VITÓRIA VIEIRA DE SOUZA
ERICK FRANCYS PORTILHO PAZ
KEVEN KAUÊ GONÇALVES PINTO

RELATÓRIO FINAL - PROJETO DE DESENVOLVIMENTO DE
SOFTWARE

Tucuruí-PA
2026

CAMILLE VITÓRIA VIEIRA DE SOUZA
ERICK FRANCYS PORTILHO PAZ
KEVEN KAUÊ GONÇALVES PINTO

RELATÓRIO FINAL - PROJETO DE DESENVOLVIMENTO DE SOFTWARE

Relatório final apresentado como requisito à obtenção de aprovação na disciplina de Teoria da Computação do Curso de Graduação em Engenharia de Computação da Universidade Federal do Pará.

Prof. Dr. Otávio Noura Teixeira .

Sumário

	Páginas
1 Introdução	5
2 Algoritmos Empregados para Autômatos	5
2.1 Definição de um Autômato Finito Não-Determinístico (AFND) e Determinístico (AFD)	5
2.1.1 Formalização do AFND e do AFD	5
2.1.2 organizar_entrada	6
2.1.3 formalizar_automato	6
2.1.4 printar_formalizacao	6
2.1.5 inicializar_funcao_transicao	6
2.1.6 inserir_transicao	6
2.1.7 printar_tabela_funcao_transicao	7
2.2 Tradução de um AFND em um AFD	7
2.3 Conceitos importantes para tradução	7
2.3.1 criar_funcao_transicao_afd	7
2.4 Minimização de um AFD	9
2.4.1 Conceitos importantes para minimização	9
2.4.2 verificar_se_eh_afd	10
2.4.3 preencher_transicoes_vazias_afd	10
2.4.4 imprimir_tabela_de_pares	11
2.4.5 marcar_pares_trivialmente_nao_equivalentes	11
2.4.6 marcar_pares_nao_equivalentes	11
2.4.7 juntar_pares_equivalentes	11
2.5 Especificação de uma gramática	11
2.5.1 printar_regras_gramaticais	12
2.5.2 verificar_tipo_gramatica	12
3 Algoritmos Empregados para Gramáticas	12
3.1 Conceitos Fundamentais	12
3.2 Processo de Conversão Passo a Passo	12
3.2.1 Exemplo Prático:	13
3.2.2 Importância na Computação	13
3.3 Definições da Gramática Livre de Contexto (GLC)	13
3.3.1 Processo de Simplificação e Aproximação	13
3.3.2 Exemplo de Transformação	14
3.3.3 Limitações	14

4	Testes com palavras	14
4.1	Adição do validador de palavras do AFND	14
4.1.1	<i>percorre_AFND</i>	15
4.1.2	<i>percorre_AFD</i>	16
5	Gerando Árvore de Derivação e Pseudocódigo do Reconhecedor	17
5.1	Geração da Árvore de Derivação	17
5.1.1	Conceitos fundamentais da derivação	17
5.1.2	<i>gerar_arvore_derivacao</i>	17
5.1.3	<i>imprimir_arvore_visual</i>	18
5.2	Geração do Pseudocódigo do Reconhecedor	19
5.3	Conceitos do Analisador Descendente Recursivo	19
5.3.1	<i>gerar_pseudocodigo_reconhecer</i>	19
6	Conclusões	19

1 Introdução

O presente relatório tem por objetivo explicar os conceitos e passos de desenvolvimento do projeto 1 da disciplina de Teoria da Computação. O projeto está pautado na criação de dois reconhecedores de palavras: Automato Finito Não Determinístico (AFND) e o Automato Finito Determinístico (AFD), assim como as suas respectivas lógicas de leitura e validação de uma palavra e a conversão a posteriori de um AFND em AFD. Por fim, o projeto contará também com a respectiva conversão do AFD em sua versão mínima em conjunto com a possibilidade de validação de palavras.

Para alcançar esse objetivo, foi utilizado o ambiente de desenvolvimento *Visual Studio Code*, em conjunto com a linguagem de programação *Python*, versão 3.12.4 e as bibliotecas *Tabulate* e *Pandas* nas versões 0.9.0 e 2.3.0 respectivamente.

2 Algoritmos Empregados para Autômatos

2.1 Definição de um Autômato Finito Não-Determinístico (AFND) e Determinístico (AFD)

2.1.1 Formalização do AFND e do AFD

Dentre os conceitos iniciais da criação de um autômato finito destaca-se a formalização necessária para o pontapé para a definição da máquina, a qual é fruto de um rigor matemático pré-definido, sendo este:

- Σ : Lista de símbolos de um determinado alfabeto
- Q : Conjunto total de estados da máquina
- δ : Função de transição dos estados
- q_0 : Estado inicial da máquina
- F : Conjunto de estados finais

Esses termos configuram a máquina (M) que será desenvolvida: $M = (\Sigma, Q, \delta, q_0, F)$.

Para alcançar essa formalização no software, assim como mostrar as informações foram criadas seis funções no software, sendo essas:

- 1. *organizar_entrada*
- 2. *formalizar_automato*
- 3. *printar_formalizacao*

- 4. *inicializar_funcao_transicao*
- 5. *inserir_transicao*
- 6. *printar_tabela_funcao_transicao*

As funções 5 e 6 possuem certas especificidades em relação ao tipo do automato que está sendo tratado, os quais são informados internamente pelo programa em resposta a escolhas do usuário, essas características que originam os diferentes tratamentos estão pautadas na quantidade de estados que cada tipo de automato pode apresentar para um determinado símbolo AFND (várias transições) AFD (apenas uma).

2.1.2 organizar_entrada

Tem como função exclusiva organizar as informações concedidas pelo usuário em listas. Essa função recebe as informações, retira as vírgulas e espaços em branco e retorna a lista de elementos organizada.

2.1.3 formalizar_automato

Essa função tem como objetivo organizar em uma tupla as informações recebidas após a organização da entrada (alfabeto, estados, tipo, estado inicial e estados finais).

2.1.4 printar_formalizacao

É uma função acessória que possui como entrada a tupla formalizada e entrega como saída a mostragem dos elementos de maneira visualmente acessível para compreensão do usuário.

2.1.5 inicializar_funcao_transicao

Essa função é crucial para a definição da AFND e AFD, haja vista que para iniciar o processo de preenchimento das transições para cada estado do usuário será primordialmente necessária a criação da estrutura por trás, o papel fundamental dessa função, portanto, é após o recolhimento da lista de estados e símbolos e o tipo do automato inicializar a estrutura para receber as transições.

2.1.6 inserir_transicao

Essa função recebe como entrada um estado dentre a lista de estados (criado por organizar_entrada) que foram informados pelo usuário e depois armazenado na lista de estados, seu funcionamento é relativamente simples, pautado novamente na inserção de dados via terminal pelo usuário e armazenagem dessas informações os quais seguem por sua vez o formato geral de um dicionário de estados como chaves em conjunto de sub-dicionários como valores, cujos por sua vez armazenam os símbolos como chaves e os destinos como valores.

2.1.7 `printar_tabela_funcao_transicao`

Essa função tem como objetivo printar a tabela de função de transição de maneira visualmente aceitável, para isso é utilizada a biblioteca `tabulate`.

2.2 Tradução de um AFND em um AFD

2.3 Conceitos importantes para tradução

Conceitualmente, temos que, para realizar a tarefa de tradução de um automato não determinístico em um determinista, ocorre pela verificação dos destinos de um determinado estado lido a partir do estado inicial $\langle q_0 \rangle$; a pergunta primordial deve ser:

"A partir do estado em que eu me encontro, quais outros estados posso alcançar no símbolo lido?"

Os estados possíveis agora serão organizados não em uma lista como no AFND, mas sim em um conjunto unido de estados. Ao analisar novamente a figura3, o estado q_0 ao ler o estado 0 possui o seguinte conjunto de possíveis estados para cada símbolo:

- 1: $\langle q_1q_2q_5 \rangle$
- 2: Nenhum estado

A segunda pergunta a ser feita é:

"Houve o surgimento de algum estado novo?"

No caso analisado, sim, houve o surgimento do estado $\langle q_1q_2q_5 \rangle$, o qual será analisado a partir de cada membro de sua constituição e o resultado da união de todos os destinos de cada membro originará outro estado, que, caso ainda não tenha sido analisado, será posto para repetir o procedimento já mencionado. Até que por fim não haja novos estados para análise.

Para alcançar esse objetivo em código, a implementação na função `criar_funcao_transicao_afd`.

2.3.1 `criar_funcao_transicao_afd`

Essa função tem como objetivo criar a função de transição do AFD a partir de um AFND, como parâmetros foram definidos três:

- `funcao_transicao`: Função de transição do AFND
- `estado_inicial`: Estado de início para verificar os estados de destino
- `lista_simbolos`: Lista para ajudar a percorrer os destinos associados a cada símbolo

A saída da função é dada também por um dicionário de dicionários, a diferença entretanto desse para o dicionário de armazenagem da função de transição do automato não determinístico foi a utilização de um *frozenset* já que a linguagem permite apenas chaves estáticas e como alguns conjuntos de estados (*estado_atual*) podem ser dados por exemplo $\langle q_1q_2q_5 \rangle$ facilitando a iteração sobre essas chaves inibindo possíveis erros relacionados à sintaxe *python*.

O processamento interno da função é baseado em um laço *while*, onde enquanto houver estados em processamento, ou seja, estados novos para serem analisados, uma vez analisados, são armazenados em uma lista chamada *estados_processados*. Há uma verificação para ver também se o estado que está em processamento já foi alguma vez processado, a qual ocorre por meio da análise se esse se encontra no conjunto *estados_processados*, caso analisado, o código avança para a próxima iteração.

Durante as iterações do algoritmo cada estado do conjunto *estado_atual* é selecionado individualmente para recolher os estados de transição para cada símbolo seu, informações essas que são guardadas em uma variável denominada *novos_destinos* que por sua vez são organizadas em um novo estado que será adicionado ao dicionário de dicionários que compõem a função de transição do automato determinístico, cujo final após a análise de todos os estados é retornado ao código principal.

Após esse processamento, o retorno da função será retornado à função de printagem da função de transição que organizará e mostrará ao usuário a função de transição obtida para o AFD, como exemplo do retorno temos 1:

Estado	0	1
$\langle q_0 \rangle$	$\langle q_1q_2q_5 \rangle$	-
$\langle q_1q_2q_5 \rangle$	-	$\langle q_3q_4 \rangle$
$\langle q_3q_4 \rangle$	$\langle q_5q_6 \rangle$	-
$\langle q_5q_6 \rangle$	$\langle q_6 \rangle$	$\langle q_3q_4q_6 \rangle$
$\langle q_6 \rangle$	$\langle q_6 \rangle$	$\langle q_6 \rangle$
$\langle q_3q_4q_6 \rangle$	$\langle q_5q_6 \rangle$	$\langle q_6 \rangle$

Figura 1. Função de Transição do AFD antes da modificação dos nomes

Contudo, essa tabela ainda apresenta os nomes antigos oriundos da conversão, para

Estado	0	1
P0	P1	-
P1	-	P2
P2	P3	-
P3	P4	P5
P4	P4	P4
P5	P3	P4

Figura 2. Função de Transição do AFD após a modificação dos nomes

finalizar a conversão, é necessário trocar tais nomes. Para isso, fora utilizado como convenção o nome de estado "P "seguido da numeração de aparição do estado iniciando pelo 0, como nova tabela teremos a 2:

2.4 Minimização de um AFD

2.4.1 Conceitos importantes para minimização

Conceitualmente temos que a minimização de um AFD é o processo de transformação de um autômato determinístico em outro equivalente, com o menor número possível de estados, nesse processo, é mantida a equivalência da linguagem reconhecida.

Para minimizar a quantidade de estados ao máximo são empregadas estratégias para verificar quais desses estados possuem equivalência, para assim juntá-los em um só.

Para averiguar se um autômato é passível de minimização, antes são necessários alguns passos:

- 1º Verificar se o AF é determinístico
- 2º Verificar se a Função de Transição é total ou completa
- 3º Verificar se todos os estados do AFD são alcançáveis a partir do estado inicial

A terceira condição já é garantida utilizando a conversão do AFND para AFD, uma vez que a verificação de novas transições partem do estado inicial, assim sendo alcançadas.

Para conceber a minimização de um dado AFD no software foram criadas as seguintes funções:

- 1. *verificar_se_eh_afd*
- 2. *preencher_transicoes_vazias_afd*
- 3. *imprimir_tabela_de_pares*
- 4. *marcar_pares_trivialmente_nao_equivalentes*
- 5. *marcar_pares_nao_equivalentes*
- 6. *juntar_pares_equivalentes*

2.4.2 verificar_se_eh_afd

Essa função tem o intuito de sanar a dúvida estabelecida como primeiro pré-requisito para poder iniciar o processo de minimização, basicamente a função tem como entrada a função de transição que será minimizada e a lista de símbolos, como processamento é percorrido cada transição e contado o número de transições de um estado, se nessa contagem houver mais de duas transições para o mesmo símbolo e estado, o autômato não é reconhecido como determinístico.

2.4.3 preencher_transicoes_vazias_afd

Com o objetivo de verificar a existência e preencher possíveis casos vazios em um AFD foi necessária a criação dessa função, que tem como parâmetros a função de transição do AFD, nesse sentido, ela é percorrida a fim de encontrar transições marcadas como *None*, identificador de vazio na linguagem *Python*.

Em caso da ocorrência de algum caso vazio, é anexado à função na célula o caso artificial "A" e ao final mais uma linha de casos artificiais "A". Dessa forma, cumprindo o segundo pré-requisito para minimização.

2.4.4 imprimir_tabela_de_pares

A função tem como objetivo recriar a tabela de pares onde foram marcadas as não equivalências que serão expostas por outras funções. Como entrada será fornecida novamente a função de transição AFD e como saída será mostrada no console a tabela de pares.

As linhas serão dadas pelos estados do segundo ao último. As colunas serão dadas dos estados do primeiro ao penúltimo.

2.4.5 marcar_pares_trivialmente_nao_equivalentes

É uma função responsável pela marcação dos pares trivialmente não equivalentes em razão das características, sendo essas a natureza dos estados: finais e não finais, basicamente a função irá receber além da δ_{AFD} o conjunto de estados finais, internamente foi verificada a paridade dos estados quanto às características mencionadas.

A função retornará as marcações na tabela criada anteriormente.

2.4.6 marcar_pares_nao_equivalentes

A lógica por trás dessa função está em fazer uma busca constante por pares anteriormente já marcados para o par de transições originados pelos símbolos lidos em cada conjunto de pares da tabela e enquanto houver mudanças (novas marcações) o loop continuará sendo alimentado até que todos os pares não equivalentes tenham sido devidamente encontrados, na entrada são adicionados a tabela de minimização já gerada pelas marcações anteriores de pares trivialmente não equivalentes, assim como a δ_{AFD} e a lista de símbolos.

O retorno da função são sucessivos *prints* da mesma tabela sendo marcada após o processamento de cada par de transições.

2.4.7 juntar_pares_equivalentes

A função final de minimização recebe a tabela de minimização atualizada e a δ_{AFD} , onde os estados encontrados como equivalentes são concatenados em um só, o qual recebe as transições do primeiro estado equivalente. Outro ponto a ser destacado é que se uma célula do AFD conter o símbolo "A", tal é substituído por um caso vazio "-", ademais, a linha de casos artificiais é retirada da tabela, o autômato resultante é então formalizado e por fim impresso e são retornados o AFD minimizado e os seus respectivos casos finais.

2.5 Especificação de uma gramática

Além dos reconhecedores de linguagem, outro ponto-chave dos estudos de teoria da computação são os geradores de linguagem, sendo um deles a gramática regular. Dessa forma, é necessário antes do prosseguimento da lógica do software formalizar essa gramática:

- N : Variáveis
- T : Conjunto de símbolos terminais
- P : Conjunto de produções (ou regras de reescrita)
- S : Símbolo inicial

Para facilitar a visualização das regras, P é criada a função *printar_regras_gramaticais*.

2.5.1 printar_regras_gramaticais

Essa função tem o simples intuito de printar as regras fornecidas dado um conjunto de regras P informadas anteriormente.

Para realizar a determinação se a linguagem fornecida é uma Gramática Regular (GR) ou Gramática Livre de Contexto (GLC) foi criado a *funoverificar_tipo_gramatica*.

2.5.2 verificar_tipo_gramatica

Essa função recebe um conjunto de regras de produção P e a partir dela se cada regra de produção possui no máximo um símbolo não-terminal. Por exemplo: A produção $A \rightarrow AB$ resultaria após o processamento da função em uma GLC. Essa função não retorna nada, apenas printa se a determinada gramática informada é uma GR ou GLC.

3 Algoritmos Empregados para Gramáticas

3.1 Conceitos Fundamentais

Gramática Regular (GR): É uma gramática linear (unitária à direita ou à esquerda) que define linguagens regulares. Ela é formalizada pelo conjunto $G = (N, T, P, S)$, onde N são os não-terminais, T os terminais, P as produções e S o símbolo inicial.

Autômato Finito Determinístico (AFD): É um modelo computacional que reconhece linguagens regulares através de estados e transições bem definidos. Para cada par (estado, símbolo), existe exatamente uma transição.

3.2 Processo de Conversão Passo a Passo

Para converter uma GR em um AFD, segue-se a lógica de mapeamento de produções para estados:

- **Mapeamento de Estados:** Cada símbolo não-terminal da gramática (N) torna-se um estado no autômato. Um estado final adicional pode ser necessário para produções que terminam apenas em símbolos terminais.

- **Estado Inicial:** O símbolo inicial da gramática (S) torna-se o estado inicial do AFD. Transições:
* Para produções do tipo $A \rightarrow aB$, cria-se uma transição do estado A para o estado B consumindo o símbolo a . Para produções do tipo $A \rightarrow a$, cria-se uma transição do estado A para o estado final especial consumindo a .
- **Determinização:** Caso a gramática gere um Autômato Finito Não-Determinístico (AFND), deve-se aplicar o algoritmo de conversão para AFD (subconjuntos) para garantir o determinismo.

3.2.1 Exemplo Prático:

Dada a gramática $S \rightarrow aS \mid bA$ e $A \rightarrow \epsilon$:

- O estado S transita para si mesmo com 'a'.
- O estado S transita para A com 'b'.
- Como A produz ϵ , A é um estado de aceitação.

3.2.2 Importância na Computação

Essa conversão é crucial para a construção de analisadores léxicos em compiladores. Gramáticas são fáceis de definir por humanos, enquanto autômatos são eficientes para implementação em hardware ou software.

3.3 Definições da Gramática Livre de Contexto (GLC)

Gramática Livre de Contexto (GLC): Gramáticas onde as produções são do tipo $A \rightarrow \alpha$, onde A é um único não-terminal e α é uma cadeia de terminais e não-terminais. São mais poderosas que as GRs (não lineares).

Simplificação: O processo de remover redundâncias (produções vazias, símbolos inúteis, produções unitárias) para tornar a gramática mais eficiente ou aproximá-la de uma forma linear.

3.3.1 Processo de Simplificação e Aproximação

O objetivo no projeto é simplificar a GLC para aproximá-la ao máximo de uma Gramática Linear Unitária à Direita (GLUD). As etapas incluem:

1. **Remoção de Produções Vazias (ϵ):** Eliminar produções que geram a cadeia vazia, exceto se a linguagem incluir ϵ .
2. **Remoção de Produções Unitárias:** Eliminar regras do tipo $A \rightarrow B$.
3. **Eliminação de Símbolos Inúteis:** Remover símbolos que não são alcançáveis a partir do símbolo inicial ou que não geram cadeias de terminais.

4. **Linearização (Aproximação):** Tentar transformar regras não lineares em lineares, se a linguagem permitir

3.3.2 Exemplo de Transformação

Uma GLC com $S \rightarrow AB$, $A \rightarrow a$ e $B \rightarrow b$ pode ser simplificada/substituída por $S \rightarrow aB$, $B \rightarrow b$, tornando-a linear e, conseqüentemente, passível de conversão para um autômato finito.

3.3.3 Limitações

Nem toda GLC pode ser transformada em uma GR. Se a GLC descreve uma linguagem que exige memória infinita (como o balanceamento de parênteses $a^n b^n$), a simplificação para GR é impossível, pois linguagens regulares são reconhecidas por autômatos com memória finita.

4 Testes com palavras

4.1 Adição do validador de palavras do AFND

A validação teórica do automato não determinístico ocorre por uma leitura recursiva de seus múltiplos estados um a um objetivando encontrar o caminho viável para a validação de uma palavra, como exemplo, temos a 3.

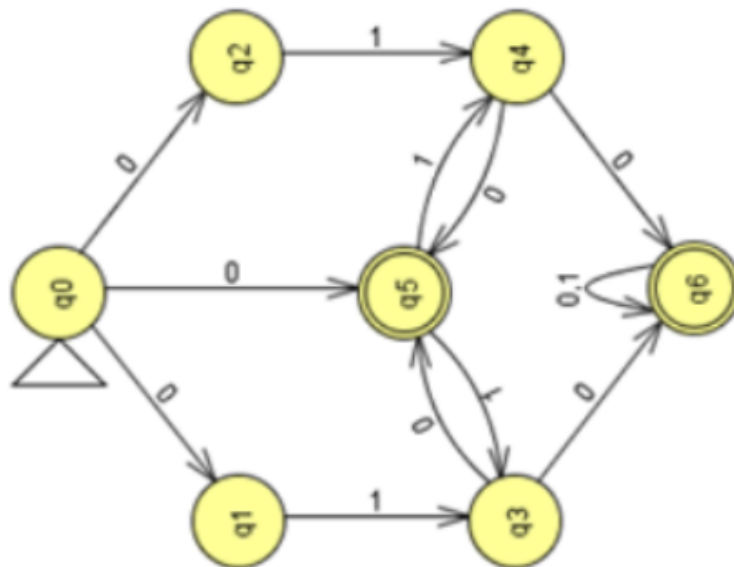


Figura 3. Automato Finito Não Determinístico

Nela podemos observar que o estado q_0 possui múltiplas possibilidades de destinos a partir da leitura do símbolo 0, supondo que vamos validar a palavra 0 no automato, para isso podemos percorrer alguns caminhos para verificar a validade da palavra:

- $q_0 \rightarrow q_1$: Ao chegar ao estado q_1 verifica-se que esse $\notin F$

- $q_0 \rightarrow q_2$: De mesma forma, o estado q_2 não alcança os estados finais, logo $\notin F$
- $q_0 \rightarrow q_5$: Enfim, $q_5 \in F$ validando a palavra

Como pode ser percebido, há uma busca de maneira recursiva em prol da descoberta do caminho que valide a palavra, sempre retornando a um estado posterior ao que foi encontrado um bloqueio para o prosseguimento da validação.

Para poder validar uma certa palavra nesse automato foi criada a função *percorre_AFND* cuja função em síntese é percorrer recursivamente os estados até encontrar um que valide a expressão fornecida.

4.1.1 *percorre_AFND*

A função recebe quatro argumentos:

- estado_atual: Estado onde se está sendo procurado o possível destino
- palavra: Item a ser validado
- cabeca_de_leitura: Elemento lógico para auxiliar na leitura da palavra símbolo a símbolo
- caminho: Trata-se de uma variável que auxilia na construção do *output* desenvolvimento ao longo do percorrimeto dos distintos estados

Em primeira análise, a função faz uma verificação a fim de verificar se a cabeça de leitura já alcançou o limite da palavra através da sua posição em relação ao comprimento do item utilizando $len(palavra)$, momento onde pode ser impressa o caminho dos estados percorridos, caso o estado final alcançado esteja na lista de estados finais inseridos pelo usuário a função retornará que a palavra é aceita pelo AFND, o contrário ocorrerá em caso do não reconhecimento.

Para poder analisar os possíveis caminhos, os símbolos são recolhidos um a um em relação à cabeça de leitura, pegando sua posição no *array* da *string* que forma sentença, um detalhe a ser mencionado trata-se da verificação se o símbolo pertence ao alfabeto ou se o estado atual possui está inserido na lista de estados, caso não, a função retornará um aviso juntamente a *False* ou seja, não reconhecimento.

A recursão mencionada encontra-se ao final da função, chamando a si própria passando os diferentes destinos nos estados de destino para cada símbolo, atualizando-se quanto à cabeça de leitura e ao destino.

Contudo, o desenvolvimento do código não se deu apenas pela escrita manual, houve o uso da *LLM Chat GPT*, para ajudar a percorrer os estados do AFND e facilitar a inclusão da recursividade. Em um primeiro momento, foi empregada uma lógica simples de seguir apenas o primeiro caminho possível (utilizando o primeiro item do vetor de destinos). Contudo essa abordagem simplista reduzia a efetividade em validar uma palavra por meio de outras transições de estados, a dificuldade na construção de uma lógica consistente para montar um aparato que suprisse essa demanda por recursividade levou ao uso do *Chat GPT*, o *prompt* inserido foi:

Crie uma função recursiva que ajude na leitura de todos os caminhos possíveis, como argumentos use (destino, palavra, cabeca_de_leitura, caminho), caminho significa o percurso de estados iniciado pelo estado q0.

Como resultado, foi gerado o seguinte bloco de código:

```
# Explora recursivamente TODOS os estados de destino
for destino in estados_de_destino:
    aceita = percorre_afnd(
        destino,
        palavra,
        indice + 1,
        caminho + [destino]
    ) or aceita

return aceita
```

Figura 4. Resultado do prompt1 - recursividade

Nesse código, podemos observar que o principal diferencial dessa abordagem é percorrer todo o conjunto de *estados_de_destino* e enviá-los à própria função, atualizando, além do destino, o índice para modificar a cabeça de leitura, que avança para o próximo símbolo, e o caminho percorrido.

4.1.2 percorre_AFD

Essa função também recebe cinco argumentos ao todo:

- estado_atual: Estado onde se está sendo procurado o possível destino
- palavra: Item a ser validado
- lista_simbolos: Elemento lógico para auxiliar na leitura da palavra símbolo a símbolo
- estados_finais: Contém os novos estados finais do automato
- funcao_transicao: Toda a função do AFD

Em posse dessas informações, toda a palavra é percorrida recolhendo o símbolo na determinada posição em que a cabeça de leitura está. Ao verificar se o símbolo pertence ao alfabeto, é verificada a transição correspondente para tal símbolo, que posteriormente é armazenada numa lista de caminhos que será impressa.

Vale ressaltar que o mesmo processamento é utilizado para validar palavras no AFD mínimo.

5 Gerando Árvore de Derivação e Pseudocódigo do Reconhecedor

5.1 Geração da Árvore de Derivação

5.1.1 Conceitos fundamentais da derivação

A árvore de derivação é uma representante gráfica que demonstra como uma palavra w pode ser gerada a partir de um símbolo inicial de uma dada gramática $G = (N, T, P, S)$, aplicando sucessivamente suas regras de produção. Para este projeto, optamos por implementar a estratégia de **Derivação a esquerda**, onde o símbolo não-terminal mais à esquerda da cadeia da cadeia atual é sempre o escolhido para ser substituído na proxima etapa.

Para conseguirmos gerar computacionalmente a árvore e encontrar o caminho exato das produções que validam a palavra alvo, foi empregado um algoritmo de **Busca em Profundidade com Backtracking**. O programa explora as possibilidades de substituição e ao encontrar um caminho sem saída ou ultrapassar o tamanho da palavra original, retrocede para testar novas regras.

- 1. *gerar_arvore_derivacao*
- 2. *imprimir_arvore_visual*

5.1.2 gerar_arvore_derivacao

Essa é a função central responsável pela lógica matemática de derivação. Ela recebe como parâmetros o dicionário contendo a gramática formalizada e a *palavra_alvo* que o usuário deseja validar.

Dentro da função possui uma sub-rotina recursiva denominada derivar, que atua rastreando três elementos a cada chamada: *forma_atual* (a cadeia de terminais e não-terminais no momento), os *passos_str* (histórico visual das derivações) e as *regras_usadas* (tuplas exatas das produções escolhidas).

A recursão possui duas condições de parada primordiais para evitar loops infinitos e otimizar o processamento

- **Sucesso:** se a *forma_atual* for estritamente igual à *palavra_alvo*, o algoritmo retorna o histórico de regras com sucesso.
- **Poda:** se o comprimento da *forma_atual* for maior que o da *palavra_alvo*, o caminho é abortado, retornando vazio. pois uma gramática (sem produções vazias desenfreadas) não pode reduzir o tamanho da cadeia em passos subsequentes.

Caso a palavra não tenha sido alcançada, a função localiza o primeiro não-terminal (identificado por caracteres maiúsculos via método *.isupper()*) e aplica um laço de repetição iterando sobre todas as produções disponíveis para aquele estado assim cada tentativa gera uma nova ramificação na recursão. No término do processamento, a função imprime a demonstração da execução passo a passo no console, exibindo transições no formato "*forma anterior => nova forma (usando N -> produção)*".

Assim como na etapa de percorrimeto dos caminhos do AFND, a construção do algoritmo de Busca em Profundidade com *Backtracking* para a árvore de derivação apresentou dificuldade na implementação. A principal dificuldade era em manipular o estado da recursão sem perder o histórico exato das produções utilizadas necessário para a exibição passo a passo e simultaneamente evitar que o algoritmo entrasse em *loops* infinitos ao processar gramáticas recursivas.

Para ajudar na estruturação da sub-rotina *derivar*, fez-se o uso de uma *LLM* (Gemini). O objetivo foi extrair a base lógica de um algoritmo de busca em grafos gramaticais. O *prompt* utilizado como base foi:

"crie uma função recursiva em python que faça a derivação pra esquerda de uma palavra alvo a partir de um dicionario de regras gramaticais"

A partir da estrutura lógica gerada pela IA o código foi adaptado para se integrar na estrutura de dados do projeto (o padrão de dicionários aninhados originados no Passo 6).

A contribuição fundamental da IA neste escopo dividiu-se em dois eixos:

- **Gestão de Estado na Recursão:** A introdução de variáveis de estado contínuas (*passos_str* e *regras_usadas*) passadas como argumento na própria assinatura da função. Isso permitiu que ramificações ruins fossem descartadas sem corromper o histórico do caminho principal.
- **Estratégia de Poda:** A percepção matemática de utilizar o comprimento da cadeia gerada iterativamente ($\text{len}(\text{forma_atual}) > \text{len}(\text{palavra_alvo})$) como condição de bloqueio da recursão, impedindo estouros de pilha em produções cíclicas.

5.1.3 imprimir_arvore_visual

Para atender o requisito de exibição clara da Árvore de Derivação, essa função recursiva atua como um formatador visual no terminal. Ela consome a lista de *regras_usadas* originada pelo backtracking e a desenha hierarquicamente de cima para baixo. O algoritmo calcula a indentação dinâmica (*prefixo*) e verifica se o nó atual é o último filho de uma produção (variável *ultimo*). Com base nisso, emprega os caracteres de formatação | - para galhos intermediários e ' - para galhos finais, gerando uma visualização em formato de diretórios que facilita a compreensão da estrutura gerada.

5.2 Geração do Pseudocódigo do Reconhecedor

5.3 Conceitos do Analisador Descendente Recursivo

O objetivo do passo 10 é construir o código de um *Parser* (Analisador Sintático) capaz de reconhecer cadeias da linguagem gerada pela gramática. A abordagem adotada no programa foi a geração de um **Analisador Descendente Recursivo**.

Nessa arquitetura, cada símbolo não-terminal da gramática é mapeado diretamente para uma função homônica no código gerado. O reconhecimento de um símbolo terminal ocorre por uma função de verificação, enquanto a transição para outros não-terminais ocorre por meio de chamadas de função aproveitando a própria pilha de execução da linguagem para empilhar as regras gramaticais.

5.3.1 gerar_pseudocodigo_reconhecer

A função recebe a estrutura de dados da gramática e inicia a varredura das regras para a escrita do pseudocódigo no console. O processo ocorre em duas etapas lógicas:

1. Função Principal: Inicialmente o gerador imprime a *funcao_principal()*, responsável por inicializar o ponteiro de leitura da fita da palavra e chamar a função referente ao Símbolo Inicial (*S*) da gramática. No final da leitura, verifica-se se toda a palavra foi consumida; caso positivo, a palavra é aceita (SUCESSO).

2. Funções dos Não-Terminais: O algoritmo itera sobre todos os estados não-terminais mapeados. Para cada estado é imprimido a assinatura de uma função respectiva. Dentro de cada função é estabelecido uma estrutura condicional de múltipla escolha baseado no primeiro símbolo da produção.

- Se a regra diz que o símbolo lido na fita combina com o terminal exigido, imprime a instrução *casar('simbolo')*.
- Se a regra aponta para outro não-terminal, imprime a instrução de salto *chamar_Variavel()*.

O pseudocódigo gerado também engloba tratamento de erros, adicionando um caso *padro* que retorna *ERRO_DE_SINTAXE* caso o símbolo lido pela cabeça de leitura não se encaixe em nenhuma das produções mapeadas para o estado atual. A saída produz um pseudocódigo legível, formatado e logicamente coeso com a teoria de construção de compiladores.

6 Conclusões

O trabalho alcança o objetivo final na medida em que atende aos 10 pré-requisitos estabelecidos primordialmente pelo docente, os quais incluem a criação de um autômato finito não determinístico e sua simplificação em um AFD mínimo, os testes com palavras, assim como a criação e definição de gramáticas e suas respectivas conversões e tarefas. Assim como

o trabalho com as gramáticas regulares e livres de contexto e seus respectivos processos de simplificação e aproximação para outros modelos.