UNIVERSIDADE DO ESTADO DE SANTA CATARINA – UDESC

CENTRO DE CIÊNCIAS TECNOLÓGICAS (CCT)

Relatório

Sofia Petrykowski Soares e Elane Souza de Oliveira

T1: Fractais em máquina de Mealy

Joinville - SC

2025

Sumário

1. Objetivos do trabalho

A. Construir **três Máquinas de Mealy** que implementem as três ERs fornecidas.

B. Implementar um **simulador de Máquina de Mealy em C** que:

   a. lê arquivo MM.txt (descrição da máquina);

   b. lê arquivo w.txt (palavra linearizada com separadores '.' e N);

   c. produz arquivo *.ppm (P1) com a matriz binária de saída.

C. Validar o simulador com entradas de tamanhos 8×8, 16×16 e 512×512 (arquivos w8.txt, w16.txt, w512.txt).

2. Expressões regulares e interpretação

As ERs dadas e suas interpretações usadas para modelagem:

A. **ER1:** $(1+3)(2+4)(2+3+4)(1+3+4)(1+2+4)(1+2+3)(1+2+3+4)^*$

   a. Essa expressão impõe uma sequência de classes por posição (ou seja: primeiro símbolo deve ser 1 ou 3; segundo símbolo 2 ou 4; terceiro símbolo 2,3 ou 4; etc).

   b. **Interpretação**: aceita palavras com comprimento >= 6 que satisfaçam essas restrições por posição, e depois qualquer sequência (posições adicionais) de $\{1,2,3,4\}$.

   c. Para máquinas de Mealy que processam cada célula, isso significa: verificar cada posição da subpalavra em relação à classe exigida; ao final da subpalavra, emitir 1 se todas as posições respeitarem o padrão, senão 0.

B. **ER2:** $(12+14+32+34+21+23+41+43)^*$

   a. Essa expressão demonstra uma linguagem de pares (blocos de tamanho 2) aceitando somente os pares listados. **Observação**: cada bloco aceito é um par de símbolos (ex.: 12, 14, 32, ...).

   b. **Interpretação** para a grade: se cada célula (subpalavra de comprimento l) tem comprimento 2, então aceita se a subpalavra for um dos pares listados. Se l for maior que 2, a máquina deve ser construída para verificar pares contíguos ou deve-se interpretar que a célula tem comprimento 2 (depende da l).

C. **ER1:** $(1+2+3+4)^* 12 (1+2+3+4)^*$

a. É a linguagem de todas as palavras sobre {1,2,3,4} que contêm a substring 12. Para a aplicação: uma célula dá saída 1 se a sua subpalavra contém 12 (em qualquer posição); caso contrário 0.

3. Projeto das Máquinas de Mealy

Formato do arquivo MM.txt adotado (obrigatório para o simulador):

<lista de estados>

<estado inicial>

<estados finais>

<alfabeto de entrada> # ex: 1 2 3 4 . N

<alfabeto de saída> # ex: 0 1 \n

<transição 1: estado simbolo estado_destino saída>

Observações gerais, necessárias para todas as MMs:

A. **Entrada linearizada**: w contém subpalavras (cada célula) concatenadas e separadas por "." ; linhas separadas por N. Seu simulador lê caractere a caractere; a máquina deve produzir uma saída (0 ou 1) por subpalavra, possivelmente imprimindo ε (nenhum símbolo) enquanto consome símbolos intermediários, e imprimindo \n ao ler N.

B. **Estratégia prática:** a MM vai manter um estado que representa o progresso de verificação dentro do subpalavra. Ao ler "." a MM deve emitir 1 (se aceitou) ou 0 (se não aceitou) — isso exige que a máquina tenha acumulado suficiente informação durante a leitura do subpalavra. Para a saída ao encontrar N, a máquina deve imprimir \n (o seu simulador interpreta \n como quebra de linha).

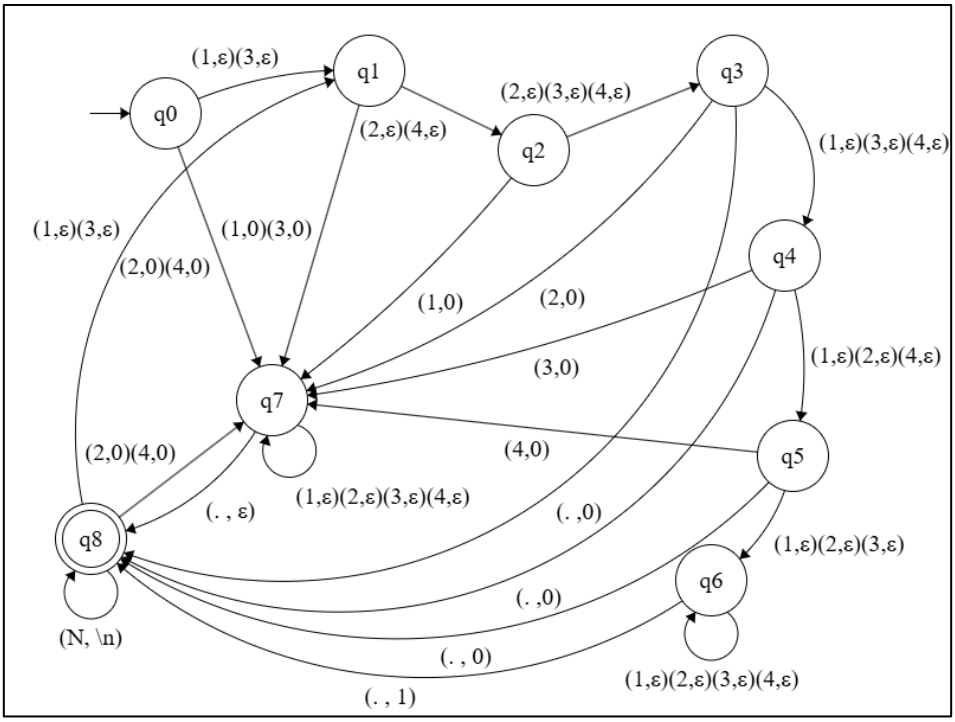
C. **Implementação:** cada transição pode imprimir ε (string vazia), 0, 1 ou \n. O simulador trata **e** como não-impressão e interpreta \n na string de saída como quebra de linha.

D. **TÓPICO RELEVANTE**: É importante ressaltar que **consideramos que '.' apenas separam subpalavras e 'N' só aparecem após pontos**, ou seja, nos autômatos não fazemos verificações de casos específicos de erros para 'N' ou '.'. Decidimos que não verificaríamos as quebras de linha em cada estado pois no autômato fornecido de exemplo na especificação do trabalho não houve essa preocupação.

3.1. MM1 - pontos principais

Observação: a expressão fixa restrições por posição para as primeiras 6 posições; após isso, qualquer símbolo é permitido. Portanto a máquina precisa verificar cada das primeiras 6 posições; se alguma falha — marcar rejeição. Após a sétima posição, não há mais restrição.

Figura 1: Autômato para expressão regular 1



Fonte: Elaborado pelas autoras (2025)

A. **Estados**: q0 (início da célula), q1..q5 (leituras 1..5), q6 (célula válida/completa), q7 (erro), q8 (final após ponto).
B. **Saídas**: e (epsilon), 0, 1, \n.
C. **Política**: Nos estados q1 a q5, se for lido algo diferente do que é pedido então registrará zero e irá para o estado q7, além disso, q6 ao ler 6º símbolo correto vai para q8 e emite 1, '.' em q8 reinicia para q0.
D. **Estratégia**: Ao ler ".", decidir saída: se estado atual é q7 então imprimir 0; caso esteja em q6 imprimir 1. Em N emitir \n e reset para q0 (linha seguinte).
E. **Observação**: Como nas especificações do trabalho foi definido que o tamanho mínimo de cada subpalavra é 3, então nos estados q3, q4 e q5 fazemos uma verificação, visto que como a expressão aceita apenas palavras com no mínimo 6 dígitos na sequência correta, então se houver "." antes disso deve ser registrado 0.

Tabela 1: Adjacências para MM1

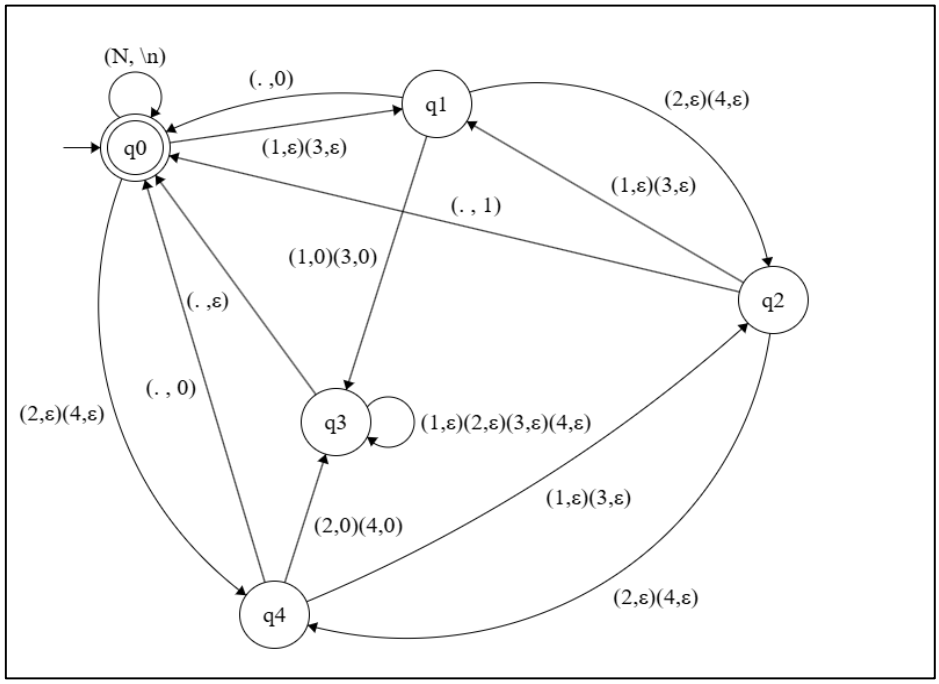
δ	1	Saída p/ 1	2	Saída p/2	3	Saída p/3	4	Saída p/4	N	Saída p/ N	.	Saída p/ .
q0	q1	e	q7	0	q1	e	q7	0				
q1	q7	0	q2	e	q7	0	q2	e				
q2	q7	0	q3	e	q3	e	q3	e			q8	0
q3	q4	e	q7	0	q4	e	q4	e			q8	0
q4	q5	e	q5	e	q7	o	q5	e			q8	0
q5	q6	e	q6	e	q6	e	q7	0			q8	0
q6	q6	e	q6	e	q6	e	q6	e			q8	1
q7	q7	e	q7	e	q7	e	q7	e			q8	e
q8									q8	\n		

Fonte: Elaborado pelas autoras (2025)

3.2. MM2 - pontos principais

Observação: A implementação prática precisa manter qual foi o primeiro símbolo para decidir a transição do segundo.

Figura 2: Autômato para expressão regular 2



Fonte: Elaborado pelas autoras (2025)

A. **Estados**: q0(inicial e final), ->q1->q2 para pares que começam com 1 ou 3 e procedem de 2 ou 4, ->q4->q2 para pares que começam com 2 ou 4 e procedem de 1 ou 3.

B. **Política**: Se chegou em q2 e ler ponto será printado 1, pois será um par válido, agora, caso esteja em q3 significa que obteve uma dupla com números repetidos que não é válida (como 11, 22, 33 e 44), então se ler ponto em q3 registrará 0. Além

disso, se estiver em q1 ou q4, significa que está com um número ímpar de tamanho de subpalavra (o que é inválido), então se ler "." registrará 0 também.

Tabela 2: Adjacências para MM2

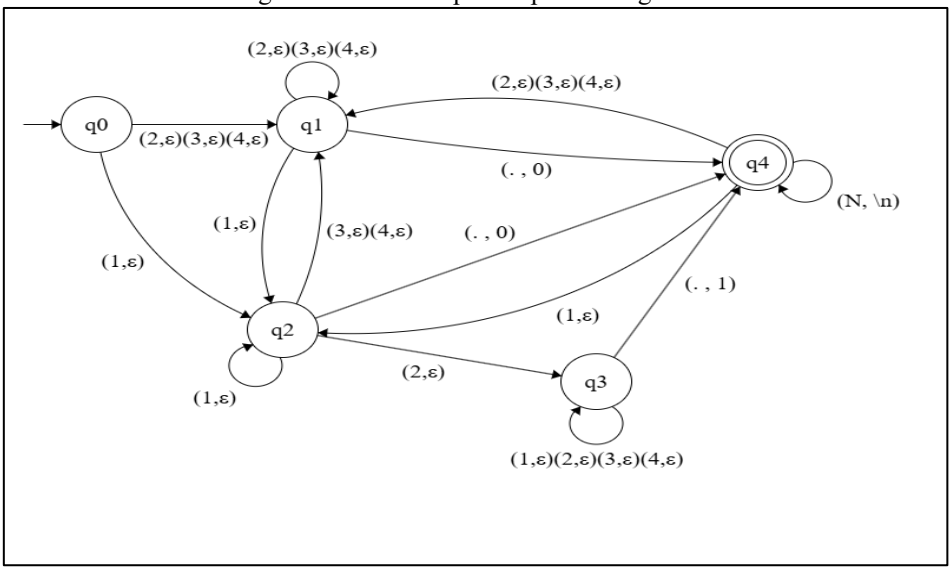
δ	1	Saída p/ 1	2	Saída p/2	3	Saída p/3	4	Saída p/4	N	Saída p/ N	.	Saída p/ .
q0	q1	e	q4	e	q1	e	q4	e	q0	\n		
q1	q3	0	q2	e	q3	0	q2	e			q0	0
q2	q1	e	q4	e	q1	e	q4	e			q0	1
q3	q3	e	q3	e	q3	e	q3	e			q0	e
q4	q2	e	q3	0	q2	e	q3	0			q0	0

Fonte: Elaborado pelas autoras (2025)

3.3. MM3 - pontos principais

Objetivo: emitir 1 ao final de um subpalavra se em algum ponto dentro dele houve a substring 12.

Figura 3: Autômato para expressão regular 3



Fonte: Elaborado pelas autoras (2025)

A. **Estados**: q0 (inicial), q1(erro), ->q2->q3 leu a substring 12, q4 (final após ponto).
B. **Estratégia**: Caso leia 1, vai para q2, se ler 2 vai para q3 e conterá a substrign 12 logo se ler "." registrará 1, agora, caso esteja nos estados q1 ou q2, ao ler "." registrará 0.

Tabela 3: Adjacências para MM3

δ	1	Saída p/ 1	2	Saída p/2	3	Saída p/3	4	Saída p/4	N	Saída p/ N	.	Saída p/ .
q0	q2	e	q1	e	q1	e	q1	e				
q1	q2	e	q1	e	q1	e	q1	e			q4	0
q2	q2	e	q3	e	q1	e	q1	e			q4	0
q3	q3	e	q3	e	q3	e	q3	e			q4	1
q4	q2	e	q1	e	q1	e	q1	e	q4	\n		

Fonte: Elaborado pelas autoras (2025)

4. Especificação do simulador em C - visão geral

O simulador foi escrito em C (padrão ISO C) devido à familiaridade com a linguagem. Requisitos funcionais atendidos:

A. aceita qualquer nome de arquivo para a máquina (1º argumento) e para a palavra (2º argumento);
B. detecta automaticamente largura e altura a partir de w.txt (conta símbolos por célula antes do primeiro N - largura; conta N - altura);
C. lê a definição da MM conforme formato descrito;
D. simula a MM consumindo símbolo a símbolo e concatenando a saída;
E. gera arquivo <nome_da_maquina>.ppm no formato **P1** (PBM ASCII);
F. trata e (epsilon) e \n embutido na saída.

4.1. Estruturas de dados principais

- Transicao - representa uma linha [*origem simbolo destino saida*]; *saida* é *string* (pode ser "e", "0", "1", "\\n", etc).

Figura 4: *Transicao* guarda uma transição legível por *strings*.

```
typedef struct {
    char origem[TAM_STRING];
    char simbolo[TAM_STRING];
    char destino[TAM_STRING];
    char saida[TAM_STRING];
} Transicao;
```

Fonte: Elaborado pelas autoras (2025)

- MaquinaMealy - armazena vetores de estados, estado inicial, estados finais, alfabetos de entrada/saída, vetor de *Transicao* e contadores.

Figura 5: MaquinaMealy guarda a configuração da máquina

```c
typedef struct {
    char estados[MAX_ESTADOS][TAM_STRING];
    int qtd_estados;
    char inicial[TAM_STRING];
    char finais[MAX_ESTADOS][TAM_STRING];
    int qtd_finais;
    char simbolos_entrada[50][TAM_STRING];
    int qtd_entrada;
    char simbolos_saida[50][TAM_STRING];
    int qtd_saida;
    Transicao transicoes[MAX_TRANSICOES];
    int qtd_transicoes;
} MaquinaMealy;
```

Fonte: Elaborado pelas autoras (2025)

- Decidimos adotar valores máximos para alguns tipos de variáveis. Como não houve uma especificação, escolhemos números hipotéticos que achamos pertinentes:

  #define TAM_LINHA 1024

  #define MAX_ESTADOS 100

  #define MAX_TRANSICOES 500

  #define TAM_STRING 20

4.2. Explicação das funções em C (arquivo main.c)

A seguir, a finalidade e análise de cada função.

A. int ler_tokens(FILE *arquivo, char matriz[MAX_ESTADOS][TAM_STRING])

Figura 6: Trecho de código

```c
// Le uma linha de tokens separados por espaco
int ler_tokens(FILE *arquivo, char matriz[MAX_ESTADOS][TAM_STRING]) {
    char linha[TAM_LINHA];
    if (!fgets(linha, sizeof(linha), arquivo))
        return 0;

    int contador = 0;
    char *token = strtok(linha, " \t\r\n");
    while (token) {
        strcpy(matriz[contador++], token);
        token = strtok(NULL, " \t\r\n");
    }
    return contador;
}
```

Fonte: Elaborado pelas autoras (2025)

a. **Objetivo:** lê uma linha do *arquivo* e separa em *tokens* (pela primeira quebra de linha) usando *strtok*, preenchendo a matriz de *strings*.

b. **Entrada:** FILE *arquivo já aberto e apontando para a linha desejada.

c. **Saída:** número de *tokens* lidos (usado para contar estados, símbolos etc.).

d. Usada por *ler_maquina* para ler a primeira linha (estados), terceira (finais), quarta (alfabeto entrada), quinta (alfabeto saída).

e. **Detalhe:** assume que a linha não ultrapassa TAM_LINHA e que cada *token* cabe em TAM_STRING. Uso repetido para: lista de estados, lista de estados finais, alfabeto de entrada e alfabeto de saída.

B. Transicao *buscar_transicao(MaquinaMealy *M, const char *estado, const char *simbolo)

Figura 7: Trecho de código

```c
// Procura a transicao correspondente (estado, simbolo)
Transicao *buscar_transicao(MaquinaMealy *M, const char *estado, const char *simbolo) {
    for (int i = 0; i < M->qtd_transicoes; i++) {
        if (strcmp(M->transicoes[i].origem, estado) == 0 &&
            strcmp(M->transicoes[i].simbolo, simbolo) == 0) {
            return &M->transicoes[i];
        }
    }
    return NULL;
}
```

Fonte: Elaborado pelas autoras (2025)

a. **Objetivo:** busca linear por transição com *origem* == *estado* e *simbolo* == *simbolo*.

b. **Comportamento:** retorna ponteiro para *Transicao* ou NULL se não encontrada.

C. void ler_maquina(MaquinaMealy *M, const char *nome_arquivo)

Figura 8: Trecho de código

```c
// Le o arquivo da Maquina de Mealy
void ler_maquina(MaquinaMealy *M, const char *nome_arquivo) {
    FILE *arq = fopen(nome_arquivo, "r");
    if (!arq) {
        fprintf(stderr, "Erro ao abrir o arquivo da maquina: %s\n", nome_arquivo);
        exit(1);
    }

    M->qtd_estados = ler_tokens(arq, M->estados);
    fgets(M->inicial, sizeof(M->inicial), arq);
    strtok(M->inicial, "\r\n");  // remove quebra de linha

    M->qtd_finais = ler_tokens(arq, M->finais);
    M->qtd_entrada = ler_tokens(arq, M->simbolos_entrada);
    M->qtd_saida = ler_tokens(arq, M->simbolos_saida);

    M->qtd_transicoes = 0;
    char origem[TAM_STRING], simbolo[TAM_STRING], destino[TAM_STRING], saida[TAM_STRING];

    while (fscanf(arq, "%s %s %s %s", origem, simbolo, destino, saida) == 4) {
        strcpy(M->transicoes[M->qtd_transicoes].origem, origem);
        strcpy(M->transicoes[M->qtd_transicoes].simbolo, simbolo);
        strcpy(M->transicoes[M->qtd_transicoes].destino, destino);
        strcpy(M->transicoes[M->qtd_transicoes].saida, saida);
        M->qtd_transicoes++;
    }
    fclose(arq);
}
```

Fonte: Elaborado pelas autoras (2025)

a. **Objetivo:** carrega toda a definição textual da MM para a *struct*.

b. **Passos:**

   i. lê estados (M->estados) via *ler_tokens*.

   ii. lê a linha do estado inicial (fgets(M->inicial,...)).

   iii. lê estados finais (*ler_tokens*).

   iv. lê alfabetos de entrada e saída (l*er_tokens*).

   v. lê o bloco final com transições: *origem simbolo destino saida* via *fscanf*. Cada transição é armazenada em M->transicoes.

c. **Validação:** o código assume formato correto.

d. **Observação**: exige que o arquivo siga exatamente o formato especificado (linhas com espaços).

D. void detectar_tamanho_matriz(const char *nome_arquivo, int *largura, int *altura)

Figura 9: Trecho de código

```c
void detectar_tamanho_matriz(const char *nome_arquivo, int *largura, int *altura) {
    FILE *arq = fopen(nome_arquivo, "r");
    if (!arq) {
        fprintf(stderr, "Erro ao abrir o arquivo de entrada: %s\n", nome_arquivo);
        exit(1);
    }

    *largura = 0;
    *altura = 0;

    int contador = 0;
    char c;
    int primeira_linha = 1;

    while ((c = fgetc(arq)) != EOF) {
        if (c == 'N') {
            // Novo linha (fim de linha)
            if (primeira_linha) {
                *largura = contador; // largura definida pelo número de células (.)
                primeira_linha = 0;
            }
            (*altura)++;
            contador = 0;
        }
        else if (c == '.') {
            contador++; // conta apenas células
        }
    }

    // Caso o arquivo não termine com 'N'
    if (contador > 0) {
        if (primeira_linha)
            *largura = contador;
        (*altura)++;
    }

    fclose(arq);
}
```

Fonte: Elaborado pelas autoras (2025)

a. **Objetivo:** determinar automaticamente as dimensões (largura e altura) da matriz de saída que será gerada pelo simulador da Máquina de Mealy, com base na estrutura do arquivo de entrada (w.txt).

b. **Fluxo principal:** varre o w contando símbolos até N para inferir largura (número de células da primeira linha) e altura (número de linhas). Ignora espaços/linhas. Essa função assume que a primeira linha (antes do primeiro N) contém exatamente largura células (contagem de símbolos relevantes).

   i. Leitura dos caracteres:

      1. A função percorre todo o arquivo com *fgetc()*, analisando cada caractere para identificar:

      2. ‘.’ indica fim de uma célula - incrementa o contador de largura.

      3. ‘N’ indica fim de uma linha - incrementa a altura e, se for a primeira linha, define a largura baseada no contador acumulado até aquele ponto.

4. Os demais caracteres (dígitos 1, 2, 3, 4) são ignorados para o cálculo de tamanho, pois representam o conteúdo interno das células, não a estrutura da matriz.

ii. Tratamento da última linha (sem N):

1. Caso o arquivo termine sem um N final, mas ainda contenha células pendentes, a função considera essa linha final como válida e ajusta altura e largura conforme necessário.

E. void simular_maquina(MaquinaMealy *M, const char *arquivo_entrada, const char *arquivo_maquina) (sem figura devido ao tamanho da função)

a. **Objetivo:** funcionalidade central: simular a MM sobre *arquivo_entrada* e escrever o .ppm.

b. **Fluxo principal:**

i. abre entrada e arquivo de saída (<nome_maquina>.ppm);

ii. detecta largura e altura; escreve cabeçalho P1\n<largura> <altura>\n;

iii. inicializa *estado_atual* = M->inicial;

iv. para cada caractere lido do arquivo de entrada (*fgetc*):

1. ignora \n e espaços; monta *simbolo[0] = c*;

2. busca transição com *buscar_transicao*; se não achar, imprime erro e para;

3. **imprime a saída da transição interpretando 'e' e \n embutido**;

4. atualiza *estado_atual* = t->destino.

c. **Interpretação de saída:**

i. percorre t->saida caractere a caractere; ignora 'e', interpreta "\n" como quebra de linha real, escreve 0/1 caso apareçam; isto garante que o .ppm conterá apenas 0 e 1 com quebras de linha no lugar certo.

d. **Importante**: O simulador executa ***fprintf(stderr, "[ERRO] Não existe transição para (%s, %s)\n", estado_atual, simbolo);*** caso haja transição faltante no arquivo .txt .

F. main

a. valida argumentos;

b. chama *ler_maquina* e *simular_maquina*.

4.3. Trecho essencial do código (padrão de impressão da saída)

Figura 10: Trecho da função simular_maquina.

```
196         char *s = t->saida;
197         for (int i = 0; s[i] != '\0'; ++i) {
198             // Ignora epsilon
199             if (s[i] == 'e') continue;
200 
201             // Interpreta \n como quebra de linha real
202             if (s[i] == '\\' && s[i + 1] == 'n') {
203                 fprintf(saida, "\n");
204                 ++i; // pula o 'n' apos '\'
205             } else {
206                 fprintf(saida, "%c", s[i]); // imprime 0 ou 1
207             }
208         }
```

Fonte: Elaborado pelas autoras (2025)

Explicação: percorre a *string saida* da transição; ignora **e**; se encontra a sequência \n converte em nova linha; caso contrário imprime 0 ou 1.

5. Casos de teste realizados

Este campo é destinado aos resultados visuais gerados pelo simulador executando cada uma das Máquinas de Mealy apresentadas, com os arquivos de entrada de exemplo.

5.1. Arquivos de entrada:

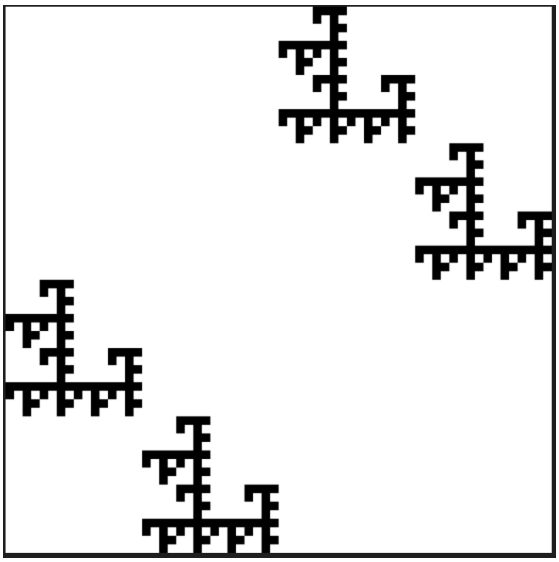
5.1.1. w8.txt

5.1.2. w16.txt

5.1.3. w512.txt.

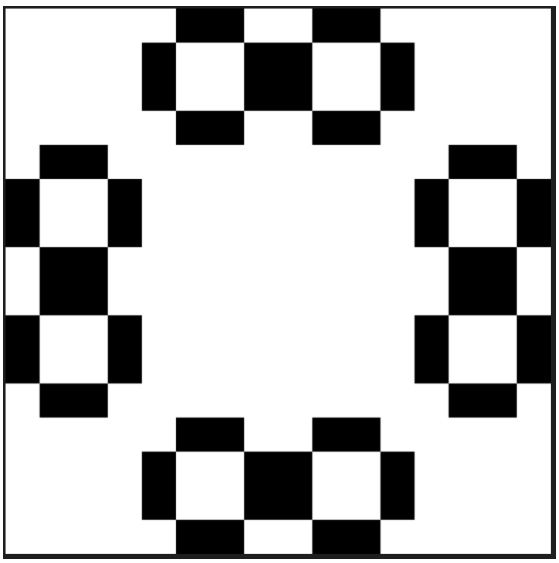
5.2. Exemplos e resultados:

Figura 11: Leitura de w512.txt com MM1.txt



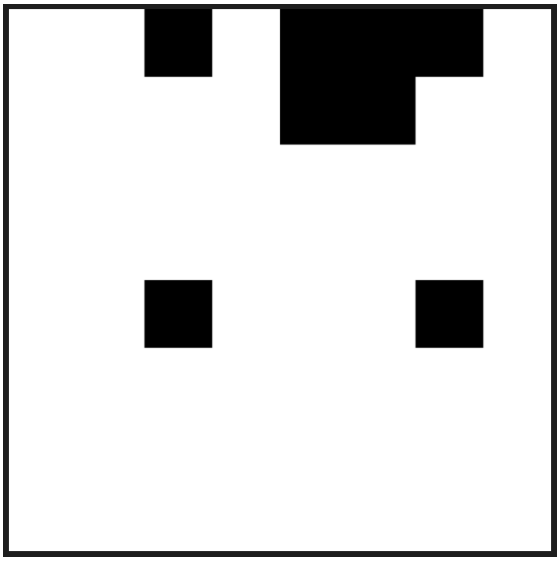
Fonte: Elaborado pelas autoras (2025)

Figura 12: Leitura de w16.txt com MM2.txt



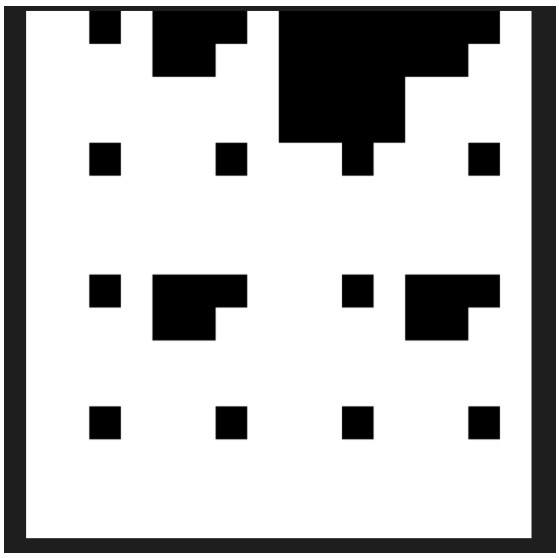
Fonte: Elaborado pelas autoras (2025)

Figura 13: Leitura de w8.txt com MM3.txt



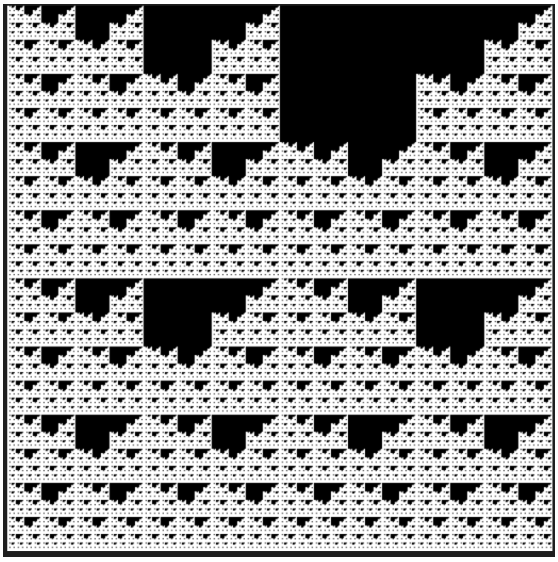
Fonte: Elaborado pelas autoras (2025)

Figura 14: Leitura de w16.txt com MM3.txt



Fonte: Elaborado pelas autoras (2025)

Figura 15: Leitura de w512.txt com MM3.txt



Fonte: Elaborado pelas autoras (2025)

A. **Observação:** Para as seguintes leituras, as matrizes binárias foram nulas e geraram imagens .ppm vazias (em branco) e por isso não registramos por aqui: Leituras de w8.txt e w16.txt com MM1.txt e leituras de w8.txt e w512.txt com MM2.txt.

6. Dificuldades e decisões de implementação

6.1. Quando emitir 0/1 — escolha de política

A. **Problema:** emitir no momento do reconhecimento (transição) ou apenas no separador '.'?

B. **Decisão:** definimos políticas por máquina (ER1 emite ao completar 6ª posição; ER2/MM3 emitem ao completar par ou ao ver '.' a partir de 3 leituras, se célula estiver incompleta), mas padronizamos que **não** haverá saídas compostas.

6.2. Detecção de dimensões da imagem

A. **Problema**: usamos *detectar_tamanho_matriz*, que considera o primeiro N para definir largura e conta quantos N para altura, porém há um risco, se há linhas inconsistentes ou espaços em branco, a detecção pode falhar.

B. **Decisão**: manter essa heurística, mas listar como melhoria (validar retas, permitir especificar dimensões manualmente).

6.3. Parsing e robustez

A. **Problema:** formato incorreto de MM.txt pode quebrar leitura com *fscanf*.

B. **Decisão:** o código assume conformidade de entrada (especificação exigida); recomendada melhoria: validações e mensagens mais precisas.

6.4. Performance em arquivos grandes

A. **Problema:** w512 contém muitas células (512×512 ≈ 262k células) — escrever em texto P1 pode ser pesado.

B. **Decisão:** algoritmo é suficientemente rápido em C.

7. Aspectos que deveriam constar na documentação de especificação

A. **Semântica de saída por célula**: quando a máquina deve emitir 1/0 (na transição que determina o resultado ou no separador).

B. **Comportamento do símbolo N** (fim de linha) em cada estado (se N pode aparecer em qualquer ponto).

C. **Requisitos do w.txt**: alfabeto permitido, separadores, exemplos para dimensões.

D. **Requisitos de validação**: o que o simulador faz em caso de transição faltante (parar com erro? tratar como 0?).

E. **Critérios de avaliação**: saída correta, robustez, documentação e testes.

8. Melhorias que poderiam ser aplicadas futuramente

A. **Validação de arquivo MM.txt:** checagem de formato, *tokens* inválidos, estados/alfabeto inexistente nas transições.

B. **Modo verbose / debug**: para imprimir passos (estado atual, símbolo, saída), útil para relatório.

C. **Conversor de ER → AFD → MM**: ferramenta para automatizar criação de máquinas a partir de expressões regulares.

9. Conclusão

O trabalho implementado atende aos requisitos centrais solicitados: modelagem formal de autômatos (Máquinas de Mealy) para ERs específicas, implementação prática de um simulador em C, e geração de saída visual (PPM) compatível com avaliação. Foram tomadas decisões técnicas para garantir comportamento previsível e documentável (tratamento de e, \n, e '.').

10. Instruções de uso

O programa foi desenvolvido em C e a execução foi testada nos sistemas operacionais Windows 11 e Ubuntu 24.04.1 LTS conforme solicitado nas especificações, utilizando o editor de código-fonte Visual Studio Code (e para a visualização das imagens utilizamos a extensão "PBM/PPM/PGM Viewer for Visual Studio Code") e também pelo próprio terminal Ubunto com as mesmas instruções. Após a execução, o arquivo *.ppm será gerado no mesmo diretório. Vale ressaltar que para cada máquina, ao testar palavras diferentes, **o arquivo sobrescreve a leitura anterior, ou seja, é sempre um arquivo de saída para cada MM**. Para executar o simulador, utilize os seguintes comandos no terminal:

10.1. Compilar:

A. gcc main.c -o main

10.2. Executar:

A. ./main MM1.txt w8.txt

B. ./main MM2.txt w8.txt

C. ./main MM3.txt w8.txt