# Relatório de Implementação

Este trabalho tem por objetivo desenvolver um algorítmo que gere um Autômato Finito Determinístico a partir de um Autômato Finito Não-Determinístico, com ou sem transições epsilon, através dos conhecimentos adquiridos em sala na disciplina de Teoria da Computação.

A primeira escolha do trabalho, foi determinar o uso da linguagem. Como a ideia principal do trabalho é formalizar o aprendizado, junto com um algoritmo eficiente. Python tornou-se a candidata ideal. Além do código fonte ser de fácil entendimento, ela é muito intuitiva e eficiente para certas ações, como por exemplo, percorrer por valores e suas respectivas chaves num hash table, ou verificar se existe um elemento numa lista.

Além disso, ela possui uma API que facilita a manipulação de arquivos JSON: uma representação de objetos usando a notação da linguagem JavaScript, que foi utilizada para descrever as máquinas. Como essa notação é bem conhecida e já tem um parser implementado, o desenvolvimento geral foi muito facilitado.

Sem contar que é possível integrar esse trabalho com qualquer plataforma. Basta fornecer um JSON com a máquina descrita não determinística, que ele devolverá outra máquina determinística, também no mesmo formato.

Para descrever a máquina, observe o exemplo de teste1, junto aos códigos fontes:

```
{
 "usa_epsilon": false,
 "alfabeto": ["a", "b"],
 "estado_inicial": "q1",
 "q1": {"a": ["q1", "q2"], "b": ["*q3"]},
 "q2": {"a": [], "b": ["q1"]},
 "*q3": {"a": ["q1"], "b": ["q2"]}
}
```

O objeto JSON acima funciona como um hash: o simbolo da esquerda é a chave e o da direita é o valor associado, de modo que o “:” separam eles. Todos esses conjuntos chave-valor fornecem os parâmetros da máquina necessários para determinizá-la.

O primeiro parâmetro, pergunta se há ou não transições epsilon. É interessante colocar isso no arquivo. Caso contrário, uma varredura pela máquina, procurando pelo menos um estado com pelo uma transição epsilon deveria ser feita, podendo tornar-se algo custoso.

O segundo parâmetro é o alfabeto, que com certeza, não precisa de muitas palavras para explicar sua importância, já que dependendo do símbolo que vier na entrada, uma determinada transição ocorrerá.

O terceiro parâmetro é o estado inicial. Criou-se uma chave para esse estado, pois ele é usado algumas vezes, como por exemplo, para obter o seu fecho em uma AFND com epsilon e assim descobrir o estado da AFD resultante, obtida através da AFND, já que a partir do estado inicial, derivam-se todos os outros.

E o restante dos parâmetros representam os estados, e suas transições. O estado q1, como se pode observar, ele possui duas transições: se vier um “b” na entrada, ele vai para o estado “q3”; e se vier um “a”, o autômato vai para q1 e q2.

Para finalizar, há três convenções importantes: a letra usada para representar o epsilon é a letra “e”. Na AFD gerada, é usado símbolo $ para representar o estado inicial.

E por fim, repare que no exemplo, o estado q3 possui um asterisco. Isso significa que ele é um estado final. Todo estado final do autômato deve possuir esse asterisco. Além de ser fácil visualizar, o processo de geração dos estados finais da AFD ficou mais simples. Note que na representação da AFD, pode acontecer de dois estados finais juntaram-se. Por exemplo, “*q0*q1q2”: tanto o q0 como o q1 aparecem com asterisco. Embora não haja necessidade dos dois asteriscos, deixou-se a representação assim, para facilitar a implementação.

A primeira coisa a ser feita, quando o programa é carregado, é preparar certas variáveis para facilitar o algoritmo de determinização. Assim, quando um objeto Maquina é instanciado, as ações abaixo são realizadas:

```
metodo preparar_maquina(arquivo):
    automato = carregar_arquivo_json(arquivo)

    usa_epsilon = automato['usa_epsilon']
    deletar automato['usa_epsilon']

    alfabeto = automato['alfabeto']
    deletar automato['alfabeto']

    estado_inicial = automato['estado_inicial']
    deletar automato['estado_inicial']

    if usa_epsilon == True:
      fecho_global = {}
      fecho_estado_inicial = fecho_estado(estado_inicial)
      fecho_global[estado_inicial] = fecho_estado_inicial
      estado_inicial = fecho_estado_inicial
```

Antes de qualquer coisa, o arquivo com as especificações da máquina é carregado. Nessa hora entrar o parser do JSON, que cuida de toda essa parte "burocrática", permitindo que o autômato seja visto como um hash, contendo o mapeamento de cada estado para suas transições.

Depois, a partir do Hash, é obtido o parâmetro usa_epsilon, para verificar se há ou não transições epsilon na máquina. Logo depois vem o alfabeto, seguido pelo estado inicial. Os três são deletados do hash, para facilitar a varredura da estrutura, na hora de determinizar.

E por último, é feito uma verificação se há transições epsilon ou não, pois há duas situações: quando não há transições, todos os estados e transições já existentes permanecem na máquina, de modo que novos estados são derivados. Mas quando há transições epsilon, boa parte dos estados e transições existentes viram estados-mortos. Dessa forma, calcula-se todos os estados e transições a partir do estado inicial, que poderá ser novo, se o fecho dele for um conjunto que não contenha apenas a si mesmo.

Feito essa inicialização, chama-se o método para gerar a determinização:

```
metodo gerar_automato():
  if usa_epsilon:
     automato = {}
     automato[estado_inicial] = estado_com_transicoes(estado_inicial)
     determinizar(automato)
  else:
     determinizar(automato_carregado)
  identificar_estado_inicial()
  gerar_resultado()
```

No trecho de código acima, verifica-se novamente se há transições epsilon. Se houver, as transições do novo estado inicial (o fecho do estado inicial fornecido) são geradas, e a partir delas, os novos estados e transições são obtidos, conforme já mencionado acima.

No entanto, caso a máquina não tenha as transições, todos os estados e transições já descritos já são embutidos no novo autômato, pois basta derivar os estados novos.

Continuando a explicação, abaixo está o método determinizar. Primeiro, armazena numa lista todas as transições. A cada iteração do while, uma transição é obtida, e para cada transição, verifica-se os estados destinos (disparados por certas entradas). Caso um estado destino não esteja ainda no autômato, suas transições são calculadas (primeira linha do if) e o estado com suas transições é inserido no autômato (segunda linha do if). A terceira linha é para inserir na lista de controle. Como era preciso saber quando todos os estados alcançáveis foram percorridos, essa lista vai descartando todos os que foram visitados. Assim, quando ela ficar vazia, é porque todas as transições e estados foram gerados. E a última linha é para associar o autômato gerado a um atributo do objeto, de modo que ele possa ser recuperado posteriormente. Algumas linhas do

código-fonte foram omitidas por simplicidade, como por exemplo, transformar um estado ou um conjunto de estados numa string.

```
metodo determinizar(automato):
    lista = automato.transicoes
    while len(lista) != 0:
     transicoes = lista.pop(0)
     for simbolo, estado_destino in transicoes:
      if (not estado_destino in automato):
        estado_com_transicoes = estado_com_transicoes(estado_destino)
        automato[estado_destino] = estado_com_transicoes
        lista.adicionar(estado_com_transicoes)

    automato_determinizado = automato
```

Note que no método acima, existe outro chamado estado_com_transicoes. O objetivo dele é: dado um estado, quais são suas transições? O primeiro for existe, para o caso da variável estado_destino ser um conjunto de estados (ex: {q1, q2}). Dessa forma, é necessário obter para todos os estados as suas transições. Feito isso, para cada letra do alfabeto, verifica-se para cada transição do estado_destino, qual o estado seguinte. E então, faz-se a união de todos esses estados seguintes, que se tornam um único estado. Repare que dependendo do valor de usa_epsilon, existe uma atribuição diferente. Caso haja transições epsilon, é preciso calcular o fecho dos estados, e não somente os estados advindos da transição gerada pela entrada. E ao final, faz a associação do simbolo com seus estados resultantes obtidos, e limpa a lista para as transições das próximas letras.

```
 metodo estado_com_transicoes(estado_destino):
    estado_com_transicoes = { }
    transicoes_estado_destino = [ ]

    for estado in estado_destino:
     transicoes_estado_destino.adicionar(automato[estado])

    estados_resultantes = [ ]
    for letra in alfabeto:
     for transicao in transicoes_estado_destino:
      if not transicao[letra] == [ ]:
```

```
        if usa_epsilon == True:
          estados_resultantes += fecho_estados(transicao[letra])
        else:
          estados_resultantes += transicao[letra]

      estado_com_transicoes[letra] = estados_resultantes
      estados_resultantes = [ ]
      return estado_com_transicoes
```

E um dos últimos métodos, formente associado a ideia do algoritmo, é o que calcula o fecho de um estado. Primeira coisa, é adicionar ao fecho o próprio estado. Em seguida, verifica-se para quais estados ocorre a transição, a partir do simbolo epsilon. Com isso, você tem o fecho do próprio estado, que é um conjunto de estados. Mas para esse conjunto de estados, deve-se obter também o fecho de cada um, que é o que a estrutura de repetição for realiza, e o estado só será adicionado, caso ele ainda não tenha sido adicionado ao fecho.

```
 metodo fecho_estado(estado):
    transicao = automato[estado]
    fecho = [ ]
    fecho.adicionar(estado)

    index = 0
    while index < len(fecho):
      estado_corrente = fecho[index]
      fecho_estado_corrente = automato[estado_corrente]["e"]

      for estado_fecho in fecho_estado_corrente:
        if not estado_fecho in fecho:
          fecho.adicionar(estado_fecho)

      index = index + 1

    return fecho
```

O método abaixo é um complemento do de cima. Nele, é possível passar um conjunto de estados por parâmetro, enquanto que noutro é obtido o fecho apenas para um estado e não um conjunto de estados. Note que nesse código, os fechos são calculados dinamicamente, a partir do momento que se precisa do fecho de um estado específico. E na atribuição antes do return,

embora possa ter estados duplicados, foram usados artíficios da linguagem Python para evitar esse comportamento.

```
metodo fecho_estados(estado):
    fecho = [ ]
    for e in estado:
      if e not in fecho_global:
        fecho_global[e] = fecho_estado(e)

      fecho += fecho_global[e]

    return fecho
```

Depois de todas essas ações, o programa gera um arquivo JSON na saída, conforme já comentado antes. Há outras chamadas de métodos e instruções auxiliares no código fonte para permitir o desenvolvimento com as estruturas escolhidas, mas nada que venha a interferir na descrição do algoritmo.