UNIVERSIDADE FEDERAL DA FRONTEIRA SUL
CAMPUS DE CHAPECÓ
CURSO DE CIÊNCIAS DA COMPUTAÇÃO

JEAN CARLOS CANOVA GONDOREK

TRABALHO PRÁTICO DE LINGUAGENS FORMAIS E AUTÔMATOS

CHAPECÓ
2025

Resumo:
No artigo iremos apresentar e detalhar o projeto prático desenvolvido na matéria de Linguagens Formais e Autômatos da UFFS. O trabalho consiste na implementação de um autômato finito e determinístico com estado de erros, onde a minimização foi vetada, sendo essa implementação a partir de um arquivo de texto onde serão dadas palavras reservadas e gramáticas regulares de uma linguagem qualquer, tendo como saída 2 arquivos CSV, sendo um o AFND e o outro o AFD com estados de erros.

1. Introdução

Na teoria dos autômatos, um autômato finito determinístico pode ser visto como uma máquina composta por três elementos, sendo eles uma unidade de entrada de informações a serem processadas, uma unidade de controle que gerencia o número finito e predefinido de estados e uma função de transição que define o novo estado da máquina com base no estado atual. A característica mais limitadora é a memória, o que faz com que todas as operações devam ser processadas como um estado (Menezes, 2000).
Com isso, o presente trabalho busca uma implementação em uma linguagem moderna de uma máquina de estados finita para reconhecimento de uma linguagem qualquer, onde pode reconhecer tokens e gramáticas regulares.

2. Trabalhos correlatos

Foram verificados alguns trabalhos para entendimento da estrutura de código e estratégias que poderiam ser utilizadas, com isso pude me basear no TrabalhoLFA (STEFANELLO, Eduardo; SARETA, Laurivan; 2018) e Automato-Finito-Deterministico (RONSONI, Igor; 2023), onde pude verificar códigos e estratégias utilizadas para o desenvolvimento da aplicação.

3. Implementação

A aplicação foi desenvolvida em Java com Quarkus, o único critério para a escolha foi familiaridade com o desenvolvimento Java. Na construção do código temos a implementação de uma arquivo Main.java que é o arquivo que gerencia e inicializa a aplicação, esse arquivo tem uma chamada de função ao InitializeProgram.java, arquivo onde há a implementação de chamadas para processamento dos autômatos e também um funcçao converterParaCSVSomenteEstados, que é utilizada para criar arquivos .csv com a saída do programa, esse arquivo InitializeProgram.java faz a chamada para as

funçoes que estão no controller (AFDController.java), uma função que serve para transformar a gramática e os tokens em algo mais fácil de manipular pela aplicação. Com isso adentramos nas funções de geração dos automatos, sendo elas a função de processFileAfnd onde temos afndGenerator.generate() que é a chamada para a geração do autômato finito não determinístico, e processFileAfd com determinizer.determinize() para transformar o autômato finito não determinístico em autômato finito determinístico.

Agora temos os dois arquivos de maior importância, sendo eles o AFNDGenerator.java que gera o autômato finito não determinístico e o Determinizer.java que gera o autômato finito determinístico. Em AFNDGenerator.java foi necessário utilizar estruturas complexas, sendo elas Automaton onde foram incluídas uma lista de estados `Set<State> states`, um lista de strings `Set<String> alphabet`, para ser o alfabeto, um estrutura para controlar o estado inicial `State initialState`, e por fim uma lista para controlar as transações entre cada estado, dada por `List<Transition> transitions`, mas como vimos, ainda existem objetos não explorados, sendo eles State e Transition, nesse caso, State temos 3 objetos, 2 strings e um booleano, onde o booleano controla se o estado é final ou não, e temos a label e o tokenName, sendo label o item do alfabeto que pode ser acessado e tokenName o estado que está acessando, e dentro de Transition temos três campos, sendo eles source, target e symbol, onde symbol é dado ao item do alfabeto que pode ser acessado e source o estado atual, enquanto target o estado destino ao acessar esse símbolo. Em resumo, temos três objetos para gerenciamento do programa, sendo eles Automaton para o nosso autómato, State para nossos estados e Transitions para nossas transições entre estados.

Abaixo iremos detalhar as estratégias de geração do AFND (autômato finito não determinístico) e do AFD (autômato finito determinístico):

- AFNDGenerator.java: Nessa função do sistemas temos a geração do AFND, que se resume em percorrer todas as palavras reservadas criando seus estados, depois cada ponto da gramática. Para o programa foram implementadas funções auxiliares sendo createState para criação de estados, generateStateName para gerenciamento dos nomes de estados e tokenCreate para a criação dos tokens.
    - Em tokenCreate, responsável pelo processamento de estados das palavras reservadas, temos a aplicação das regras para a criação do estado inicial e do próximo estado, que é montado dinamicamente para a próxima letra do alfabeto, nesse caso S sempre sendo inicial, não será utilizado na geração, somente no primeiro estado S existe. Também podemos ver que os dois for

encadeados gerenciam os valores que estarão presentes para a geração do token, sendo importante ressaltar a função de countState, quando esse igual a zero, irá gerar o estado inicial, que será o estado utilizado por base para se iniciar a verificação de uma palavra reservada.

- generateStateName como já comentado, gerencia a criação de nomes de estados novos quando necessário, ele faz o controle para criação em ordem alfabética desconsiderando S (estado inicial), e quando terminado o alfabeto inicia a criação de novos estados adicionando o valor 2, após 3 sucessivamente até o processamento completo. Note que criamos a limitação de exigir o estado inicial como S, mas por maior comodidade isso não será alterado.
- Já em tokenCreate basicamente faz a criação de cada estado, basicamente é um construtor de estados, foi criada na intenção de fazer uma aplicação mais modularizada (menor repetição de uma função de construção de estados), porém o restante do código não seguiu esse princípio.
- No processamento de tokens temos dois laços de repetição (for) aninhados, sendo que o primeiro for percorre cada token, enquanto o segundo for percorre cada letra do token para interpretação e entendimento. Temos que cada transição nesse meio tempo é adicionada ao autômato como uma transição de estados. Ao final do primeiro for, é adicionado um estado final. Por exemplo, teremos S::=s<A>, A::=<B> e B::=ϵ. A cada novo token, o estado de encerramento do token anterior (o estado B no exemplo) é marcado como final. A notação de "épsilon produção" (B::=ϵ) é utilizada unicamente na representação da gramática para sinalizar que o estado é final (isFinal = true), e não para indicar uma transição com o símbolo ϵ (transição épsilon) no autômato.
- No processamento das gramáticas temos dois laços de repetição (for) aninhados, sendo um deles percorrendo cada item, da gramática e outro percorrendo cada parte da gramática, nesse caso temos o controle de ter que verificar se o estado final é dado por ε (épsilon) dentro da produção, além disso podemos ver que o nosso sistema divide a produção da iteração com o próximo estado. Temos também a implementação de algumas regras para criação de novos estados, onde a aplicação irá verificar a possibilidade de ter um estado compartilhado por ambos, ou seja, podemos ter a saída

desse estado S (inicial) pelas letras do alfabeto a,e,i,o,u e em todos os casos iremos para um mesmo estado destino, em exemplo o M, que será final por ou nao por conter ε (épsilon) produção. Com isso a geração da gramática pode criar alguns estados que saem de uma mesma produção com destino a uma mesma produção, o que torna mais fácil a análise e se adequa às regras estudadas em trabalhos correlatos. Além disso, temos o controle das transições, onde para os tokens, a cada final de for é adicionado às suas transições, válidos para os dois laços de repetição (for) aninhados, e para a gramática a cada for interno, é adicionada a transição com controle de adicionar corretamente o estado inicial (antecessor) a sentença (alfabeto) que leva ao próximo estado (sucessor), tornando possível rastrear de forma correta as transições dos sistema e após gerar um csv corretamente.

    - Se cabe um resumo podemos afirmar que em ambos os casos teremos linhas com “frases”, onde o sistema deve percorrer cada sentença de cada frase (cada letra de cada linha) para análise de sua gramática e palavras reservadas. Com isso foi

- Determinizer.java
    - Essa função adota a estratégia de um algoritmo de construção de subconjunto para transformar o AFND em AFD. O princípio fundamental é que cada estado do AFD representa um conjunto de estados do AFND, resolvendo assim o não-determinismo. Inicialmente na função deve ser inicializado o AFD, onde mantemos o alfabeto vindo do AFND sem alterações, pois o alfabeto não muda na conversão. Com isso seguimos para a criação de estruturas para mapeamento de conjuntos (Map<Set<State>, State> mappedStates) que associa conjuntos de estados do AFND a estados únicos do AFD, e uma fila (Queue<Set<State>> queue) para adicionar os estados pendentes de processamento. O estado inicial do AFD é criado como um conjunto contendo apenas o estado inicial do AFND (conjunto {S}), sendo registrado no mapa e adicionado à fila. Feito isso e incluído o primeiro registro na fila, iremos utilizar o algoritmo BFS (Breadth-First Search ou busca em largura) para processar todos os estados alcançáveis. O loop principal remove estados da fila enquanto ela não estiver vazia, garantindo que todos os estados sejam processados.

Adentrando no BFS iremos verificar para cada símbolo do alfabeto para onde é possível executar a transição de estado. Com as transições possíveis, o algoritmo calcula a união de todos os estados alcançáveis no AFND através do método auxiliar getTargets(), que percorre todas as transições do AFND coletando os estados destino para o par (estado atual, símbolo). Esta operação resolve o não-determinismo ao considerar todas as possibilidades simultaneamente, por exemplo, se estamos no conjunto {A, B} com símbolo 'a', e A vai para {C, D} e B vai para {D, E}, a união resulta em {C, D, E}.

- Quando o conjunto de estados alcançáveis está vazio, significa que não há transição válida no AFND, então criamos uma transição para o estado de erro (ERROR) através do método getOrCreateSinkState(). Este estado especial nunca é marcado como final e todas as suas transições apontam para ele mesmo, caracterizando um sumidouro do qual não é possível sair. O estado ERROR é representado internamente como um conjunto vazio e é criado apenas uma vez, sendo reutilizado através do mapa de estados. Quando criado, automaticamente gera transições para si mesmo com todos os símbolos do alfabeto, garantindo o comportamento de loop infinito.
- Quando o conjunto não está vazio, verificamos se já existe um estado do AFD correspondente através do mapa mappedStates. Se o estado ainda não existe, criamos um novo através do método createStateForSet(), que combina os nomes dos estados do AFND ordenados alfabeticamente separados por hífen, por exemplo o conjunto {C, A, H} gera o estado "A-C-H". O estado do AFD é marcado como final se pelo menos um estado do conjunto original for final, implementando a regra de herança de aceitação. Após criar o estado, ele é adicionado ao AFD, registrado no mapa e inserido na fila para que suas próprias transições sejam processadas posteriormente. Se o estado já existe no mapa, simplesmente o reutilizamos, evitando duplicação.
- Para cada par (estado, símbolo) processado, uma transição determinística é criada no AFD ligando o estado origem ao estado destino calculado. Este processo continua até que a fila esteja vazia, momento em que todos os estados alcançáveis foram processados e o AFD está completo. Como o algoritmo só adiciona à fila estados que são alcançáveis a partir do estado

inicial através de transições válidas, automaticamente elimina estados inalcançáveis do AFND, implementando uma minimização implícita sem necessidade de algoritmos adicionais de remoção.

- Não foi implementada a função de minimização completa através de algoritmos como Hopcroft ou Moore, nem exclusão de ε-transições, pois o AFND gerado não contém transições épsilon, a implementação feita não contempla essas transições de forma explícita, ele simplesmente adicionar como uma notação no estado, sendo validado no método getTargets() que ignora símbolos vazios. Os estados inalcançáveis são naturalmente eliminados pois o AFD é gerado com base no que pode ser acessado partindo do estado inicial S, criando apenas estados descobertos durante a exploração BFS, o que já remove estados inacessíveis automaticamente. Estados mortos (que não levam a estados finais) são criados se alcançáveis mas não são identificados ou removidos posteriormente. Também não foi aplicada a remoção de estados equivalentes, resultando em um AFD funcional porém não necessariamente mínimo em número de estados.

Demais implementações não foram detalhadas por não serem o foco do presente trabalho, e foram geradas auxílio IA para maior produtividade e foco na criação das regras de geração de autômatos, as classes de geração de AFD e AFND também contaram com auxílio de IA para sua geração, porém com um uso muito mais consciente e discreto, somente como provedora de respostas a dúvidas e auxílio em momentos críticos do desenvolvimento.

4. Indeterminismo

Temos um indeterminismo sendo a generalização de um autômato finito determinístico, onde o autômato não finito não determinístico permite o acesso a diversos estados a partir do estado de origem, o que gera esse indeterminismo. (Rabin, M. and Scott, 1959)

5. Determinização

A determinização pode ser descrita como um processo de encapsular a incerteza inerente ao AFND. O AFND pode alcançar um conjunto de estados a partir de um único símbolo, pois a função de transição mapeia para subconjuntos de estados. Já o AFD equivalente opera de maneira determinística. Ele alcança isso definindo seus estados T como sendo os subconjuntos de todos os estados do AFND. Exemplificando: O AFND é como um

sistema que, ao usar uma chave (símbolo de entrada), pode potencialmente abrir diversas portas simultaneamente (chegar a vários estados). O AFD equivalente não usa várias chaves nem abre várias portas diretamente; ele usa uma única chave que o leva a um único super-estado (o estado t do AFD). Este super-estado, por sua vez, representa o conjunto unificado de todas as portas que o AFND poderia ter aberto. Se qualquer porta dentro desse super-estado for uma porta final do AFND, então o super-estado é um estado final do AFD. (Rabin, M. and Scott, 1959)

6. Conclusão

A aplicação foi desenvolvida usando por base trabalhos correlatos já citados, principalmente na determinização do AFND, onde utilizamos o método de busca em largura (BFS) para a construção de subconjuntos, que seria a ideia da unificação de dois ou mais estados possíveis em um super-estado com características herdadas dos estados anteriores. O trabalho demonstrou a viabilidade da implementação do algoritmo de Rabin-Scott em uma linguagem moderna, gerando autômatos funcionais capazes de reconhecer tanto palavras reservadas quanto gramáticas regulares.

Os resultados obtidos validam a equivalência entre o AFND gerado e o AFD resultante, mantendo a mesma linguagem reconhecida enquanto eliminam o não-determinismo inerente ao primeiro. A estratégia de exploração por BFS garantiu o processamento completo de todos os estados alcançáveis, implementando implicitamente a remoção de estados inalcançáveis sem necessidade de algoritmos adicionais. O estado sumidouro ERROR assegurou a completude do AFD, atendendo à exigência de transições definidas para todo par (estado, símbolo).

Como limitações identificadas, o AFD resultante não passou por minimização através de algoritmos como Hopcroft ou Moore, podendo conter estados equivalentes que aumentam desnecessariamente o número total de estados. Estados mortos também não foram removidos explicitamente, permanecendo no autômato final se alcançáveis. Estas decisões de design priorizaram simplicidade de implementação e corretude funcional sobre otimalidade no número de estados.

Para trabalhos futuros, sugere-se a implementação de algoritmos de minimização para reduzir o AFD ao número mínimo de estados teoricamente necessário, além da inclusão de suporte a ε transições com algoritmo de fecho-épsilon, permitindo maior flexibilidade na definição de gramáticas de entrada. Também seria interessante implementar otimizações de desempenho para lidar com autômatos de maior escala, explorando

estruturas de dados mais eficientes e técnicas de compactação de estados compostos. Por fim, a aplicação poderia ser estendida para gerar código executável de analisadores léxicos a partir dos autômatos gerados, aproximando o trabalho de ferramentas práticas como Lex e Flex utilizadas em compiladores reais.

Para conferir o código pode ser consultado em https://github.com/jeangondorek/geradorAFD.

Referências

RONSONI, Igor. Automato-Finito-Deterministico. 2023. [S.l.]: GitHub. Disponível em: https://github.com/Igorronsoni/Automato-Finito-Deterministico. Acesso em: 15 nov. 2025.

STEFANELLO, Eduardo; SARETA, Laurivan. TrabalhoLFA. 2018. [S.l.]: GitHub. Disponível em: https://github.com/laurivansareta/TrabalhoLFA. Acesso em: 15 nov. 2025.

MENEZES, Paulo Fernando Blauth. Linguagens formais e autômatos. Porto Alegre: Sagras Luzzatto, 2000. Acesso em: 15 nov. 2025.

M. O. Rabin and D. Scott, “Finite Automata and their Decision Problems”, IBM Journal of Research and Development, 3:2 (1959) pp. 115–125. Acesso em: 15 nov. 2025.