Universidade Federal da Fronteira Sul
Bacharelado em Ciência da Computação
Linguagens Formais e Autômatos

GERADOR DE AUTÔMATO FINITO DETERMINÍSTICO

Pedro Zawadzki Dutra

CHAPECÓ, 2022

Resumo

Este trabalho descreve o desenvolvimento de uma aplicação para a manipulação de gramáticas e sentenças de uma linguagem com o objetivo de gerar autômatos finitos determinísticos e mínimos. O projeto implementa algoritmos para a conversão de gramáticas regulares em autômatos finitos determinísticos, além de eliminar épsilon transições, estados mortos e inalcançáveis. A aplicação utiliza bibliotecas padrão do Python para manipulação de arquivos e construção de autômatos finitos, utilizando o padrão de orientação a objetos, visando abstrair ao máximo os teoremas de linguagens formais e autômatos de forma aplicada. O projeto apresenta uma solução prática para a geração de autômatos finitos determinísticos, livre de épsilon transições, estados mortos e inalcançáveis, para diversas aplicações em processamento de linguagem natural.

Sumário

1. Introdução

Este relatório descreve o desenvolvimento de um projeto que tem como objetivo a manipulação de gramáticas e sentenças para a geração de autômatos finitos determinísticos e mínimos. O projeto visa implementar algoritmos para a conversão de gramáticas regulares em autômatos finitos determinísticos, além de eliminar épsilon transições, estados mortos e inalcançáveis. O projeto foi desenvolvido como parte da disciplina de Linguagens Formais e Autômatos, cujo objetivo é estudar e aplicar conceitos teóricos sobre linguagens formais e autômatos para a resolução de problemas práticos.

O projeto foi desenvolvido em Python, uma linguagem de programação amplamente utilizada na área de ciência de dados e processamento de linguagem natural. A implementação foi realizada seguindo os princípios de orientação a objetos, visando a modularização e a reutilização de código, além da aplicação prática da teoria de linguagens e autômatos devido à completa abstração dos conceitos. O projeto utiliza bibliotecas padrão do Python para manipulação de arquivos e para a construção dos elementos da linguagem.

O relatório apresentará uma descrição detalhada do projeto, incluindo um referencial teórico com conceitos básicos do assunto, funcionalidades implementadas, a especificação dos comportamentos esperados da aplicação, detalhes sobre a implementação e uma avaliação do desempenho do projeto. O projeto tem como objetivo demonstrar a aplicação prática de conceitos teóricos aprendidos na disciplina de Linguagens Formais e Autômatos, bem como contribuir para a formação de habilidades em programação orientada a objetos e em processamento de linguagem natural.

2. Referencial Teórico

2.1. Símbolos

Em linguagens formais e autômatos, um símbolo é um elemento básico usado para construir uma expressão da linguagem. Os símbolos podem ser caracteres, dígitos, operadores, palavras reservadas, símbolos especiais ou qualquer outro elemento que compõe a linguagem. Assim, uma palavra ou sentença é definida como uma sequência finita de símbolos do alfabeto da linguagem.

Em um autômato finito, os símbolos são usados para definir como o autômato transita de um estado para outro. Cada estado do autômato tem um conjunto de símbolos associados, que são usados para definir as regras de transição. Quando uma cadeia de símbolos é fornecida como entrada, o autômato segue essas regras de transição para se mover de um estado para outro e verificar se a sentença pertence à linguagem.

Existem duas classificações de símbolos para a teoria de linguagens formais e autômatos, os símbolos terminais e não-terminais. Os símbolos terminais representam os elementos básicos de uma linguagem, como letras, números e caracteres especiais. Já os símbolos não-terminais são utilizados para representar estruturas mais complexas, em uma gramática formal, os símbolos não-terminais são substituídos por sequências de símbolos terminais, resultando na formação de cadeias de símbolos terminais que pertencentes à linguagem.

2.2. Gramáticas Regulares

Uma gramática regular (GR) é uma classe de gramáticas formais que geram linguagens regulares, reconhecidas por autômatos finitos determinísticos ou não. As gramáticas regulares são caracterizadas por terem regras de produção com formas restritas, onde a parte esquerda é sempre um único não-terminal e a parte direita é um único não-terminal, ou terminal seguido ou não de um único não-terminal.

2.3. Backus-Naur Form

Durante a implementação deste projeto, a representação utilizada para representar as gramáticas estará na notação BNF (*Backus-Naur Form*). Uma representação formal para descrever gramáticas formais, ela é composta por um conjunto de regras de produção que definem como as expressões de uma linguagem podem ser construídas.

Em síntese, cada regra de produção é composta por um símbolo não-terminal que dá nome às regras de produção. São escritas na forma "<A> ::= R1 | R2 ... Rn", onde A é um símbolo não-terminal e R1, R2, ..., Rn são símbolos não-terminais ou terminais. Essa notação significa que A pode ser substituído pela sequência R1, R2, ..., Rn.

2.4. Autômato Finito

Um autômato finito nada mais é que outra representação para as gramáticas da linguagem. Ele consiste em um conjunto finito de estados, os quais são compostos por diversas transições por meio de símbolos terminais que sempre levam a um novo estado ou ao final da sentença.

O autômato pode ter um ou mais estados finais, que indicam se uma cadeia de entrada foi aceita ou rejeitada pelo autômato. Os autômatos finitos podem ser divididos em duas categorias: autômatos finitos determinísticos (AFD) e autômatos finitos não determinísticos (AFND).

Os AFDs possuem uma única transição possível para cada símbolo de entrada, enquanto que os AFNDs podem ter várias transições possíveis para um mesmo símbolo de entrada. Assim, a conversão de um AFND para um AFD, torna o autômato mais simples e computável visto que só há um caminho possível em cada transição.

2.4.1. Épsilon Transições

Épsilon transições são transições especiais que não geram nenhum símbolo mas mesmo assim são capazes de transitar para outro estado. Em tese, na implementação ideal de um Autômato Finito Determinístico, cada uma das transições realizadas deve

gerar um símbolo, por isso, serão aplicadas transformações no autômato para que o mesmo esteja livre de transições como essa.

2.4.2. Estados Mortos e Inalcançáveis

Em autômatos finitos, estados mortos são estados que não levam a nenhum estado final. Ou seja, quando o autômato atinge um estado morto, por meio de conjunto algum de transições é possível finalizar a sentença, assim, não é possível reconhecer se ela faz parte ou não da linguagem.

Já os estados inalcançáveis são aqueles que não podem ser alcançados a partir do estado inicial do autômato. Esses estados são considerados desnecessários, pois nunca são acessados durante o reconhecimento da linguagem de entrada.

A eliminação de estados mortos e inalcançáveis é importante para simplificar o autômato e para melhorar o desempenho de sua implementação, além de eliminar possíveis erros no momento da construção das gramáticas da linguagem.

3. Especificação

3.1. Funcionamento

O algoritmo recebe como entrada um arquivo de texto com a relação de Tokens e/ou Gramáticas Regulares de uma linguagem. Após isso, faz a carga de tokens para a construção de um Autômato Finito Não Determinístico (AFND). Por fim, aplicando teoremas de Linguagens Formais e Autômatos, transforma a estrutura em um Autômato Finito Determinístico (AFD), livre de épsilon-transições, estados inalcançáveis e mortos.

3.2. Entrada

O usuário deve fornecer no arquivo de entrada os tokens e gramáticas regulares da linguagem para que o algoritmo possa fazer a sua interpretação, determinação e minimização. A inserção destes dados pode ser feita de duas formas:

- A descrição das gramáticas, no formato *BNF (Backus-Naur Form),* respeitando os espaçamentos e caracteres necessários;
- Sentenças em linguagem natural porém sem a utilização de caracteres especiais, com exceção do épsilon que deve ser representado pelo símbolo **ε**.

É possível trabalhar com as duas formas no mesmo arquivo de texto, desde que seja separado por uma quebra de linha, o fim de um modo de leitura para o outro, bem como o fim de uma gramática para outra.

As gramáticas devem ser informadas no seguinte formato:

```
<S> ::= a<A> | e<A> | i<A> | o<A> | u<A>
<A> ::= a<A> | e<A> | i<A> | o<A> | u<A> | ε
```

Já as sentenças, devem ser separadas por quebras de linhas, como:

```
se
senao
entao
```

Um exemplo de arquivo composto por ambos os tipos de leitura e múltiplas gramáticas, seria:

```
<S> ::= a<A> | e<A> | i<A> | o<A> | u<A>
<A> ::= a<A> | e<A> | i<A> | o<A> | u<A> | ε

<S> ::= a<B> | c<A> | b<B>
<A> ::= c<S> | d<B> | c<C> | ε
<B> ::= c<S> | a<D> | ε
<C> ::= b<B> | b<A>
<D> ::= ε

se
senao
entao
```

3.2.1. Comentários

Para facilitar os testes e desenvolvimento da aplicação foi implementado o reconhecimento de comentário no arquivo de carga da linguagem. Comentários começam com dois caracteres consecutivos de barra ‘//’ e estende até o final da linha. Um comentário pode aparecer no início da linha ou após espaço em branco ou código, mas não dentro de um token ou gramática. Como os comentários são para esclarecer as gramáticas e não são interpretados pelo reconhecedor, eles podem ser utilizados para ignorar sentenças ou gramáticas.

Exemplos de comentários aceitos seriam:

```
// Este é um comentário
<S> ::= a<A> | e<A> | i<A> | o<A> | u<A> // | a<B>
// <A> ::= a<A> | e<A> | i<A> | o<A> | u<A> | ε

se
senao // Olá!
// entao
```

Neste caso, o reconhecedor iria interpretar apenas:

```
<S> ::= a<A> | e<A> | i<A> | o<A> | u<A>

se
senao
```

3.3. Carga

A carga destes dados é feita em partes, utilizando como separadores de cada uma das gramáticas da linguagem a quebra de linha, e criando uma nova gramática para cada uma delas. Assim, evitando que gramáticas com os mesmos símbolos terminais ou conflitem durante sua unificação.

Para o caso das sentenças carregadas, um novo símbolo é criado para cada um dos tokens, incluindo um estado final para o épsilon, evitando estados compartilhados e, atingindo assim, um estado final diferente para cada uma das sentenças descritas no arquivo de carga.

Um exemplo de carga de um arquivo composto por, sentenças e gramáticas seria:

```
<S>::= s<A>
<A>::= e<B>
<B>::= ε

<S>::= e<A>
<A>::= n<B>
<B>::= t<C>
<C>::= a<D>
<D>::= o<E>
<E>::= ε

<S>::= s<A>
<A>::= e<B>
<B>::= n<C>
<C>::= a<D>
<D>::= o<E>
<E>::= ε

<S>::= a<A> | e<A> | i<A> | o<A> | u<A> | z<B>
<A>::= a<A> | e<A> | i<A> | o<A> | u<A> | ε
<B>::= a<B>
```

Após isso, a aplicação unifica todas estas gramáticas, gerando uma grande gramática em que apenas o estado inicial é compartilhado e os demais estados são exclusivos para as transições dos demais símbolos dos tokens e/ou estados das GRs.

O resultado da unificação para as gramáticas anteriormente mencionadas seria a seguinte linguagem:

```
<S>::= s<A> | e<C> | s<H> | a<M> | e<M> | i<M> | o<M> | u<M> | z<N>
<A>::= e<B>
<B>::= ε
<C>::= n<D>
<D>::= t<E>
<E>::= a<F>
<F>::= o<G>
<G>::= ε
<H>::= e<I>
<I>::= n<J>
<J>::= a<K>
<K>::= o<L>
<L>::= ε
<M>::= a<M> | e<M> | i<M> | o<M> | u<M> | ε
<N>::= a<N>
```

3.4. Autômato Finito

Com uma gramática agora unificada, a aplicação é capaz de transformá-la em um autômato finito, que inicialmente, poderá ser composto por estados não determinísticos uma vez que podem ocorrer uma ou mais situações em que dois tokens ou sentenças definidas por gramática regulares iniciam pelo mesmo símbolo.

A demonstração visual do autômato finito construído até o momento pela aplicação seria:

```
+------+-----+-------+-----+-----+-----+-------+-----+-----+-----+
|  -   |  a  |   e   |  i  |  n  |  o  |   s   |  t  |  u  |  z  |
+------+-----+-------+-----+-----+-----+-------+-----+-----+-----+
| >[S] | [M] | [C,M] | [M] |     | [M] | [A,H] |     | [M] | [N] |
| [A]  |     |       |     |     |     |       |     |     |     |
| [B]  |     |       |     |     |     |       |     |     |     |
| [C]  |     |       |     |     |     |       |     |     |     |
| [D]  |     |       |     |     |     |       | [E] |     |     |
| [E]  | [F] |       |     |     |     |       |     |     |     |
| [F]  |     |       |     |     | [G] |       |     |     |     |
| [G]* |     |       |     |     |     |       |     |     |     |
| [H]  |     |  [I]  |     |     |     |       |     |     |     |
| [I]  |     |       |     | [J] |     |       |     |     |     |
| [J]  | [K] |       |     |     |     |       |     |     |     |
| [K]  |     |       |     |     | [L] |       |     |     |     |
| [L]* |     |       |     |     |     |       |     |     |     |
| [M]* | [M] |  [M]  | [M] |     | [M] |       |     | [M] |     |
| [N]  | [N] |       |     |     |     |       |     |     |     |
+------+-----+-------+-----+-----+-----+-------+-----+-----+-----+
```

3.5. Autômato Finito Determinístico

Partindo do Autômato já construído, agora é possível determinizá-lo, ou seja, obter um autômato determinístico equivalente que reconheça a mesma linguagem e que ao mesmo tempo tenha transições definidas e livres de ambiguidade.

Para determinizar um autômato, por meio da construção de conjuntos são identificadas ambiguidades e a partir delas, são criados novos estados que contém todas as transições alcançáveis pelos símbolos do conjunto. Esse processo é repetido para cada novo conjunto de estados criado, gerando um novo autômato finito, agora determinístico.

O autômato anteriormente citado, agora determinístico:

-	a	e	i	n	o	s	t	u	z
>[S]	[M]	[C,M]	[M]		[M]	[A,H]		[M]	[N]
[A,H]		[B,I]							
[B,I]*				[J]					
[C,M]*	[M]	[M]	[M]	[D]	[M]			[M]	
[D]							[E]		
[E]	[F]								
[F]					[G]				
[G]*									
[J]	[K]								
[K]					[L]				
[L]*									
[M]*	[M]	[M]	[M]		[M]			[M]	
[N]	[N]								

3.6. Minimização

Além da determinização, outro processo realizado pela aplicação para a obtenção de um autômato otimizado é a minimização. Algoritmo que busca reduzir a quantidade de estados do autômato, sem alterar a linguagem reconhecida. Isso é útil porque quanto menor o número de estados, menor é a complexidade do autômato e, portanto, menor é o tempo necessário para processar uma entrada.

A minimização em questão está diretamente relacionada com a eliminação de estados mortos e inalcançáveis. Para isto, é necessário percorrer o autômato a partir do estado inicial e identificar quais estados podem ser alcançados e quais estados podem alcançar um estado final. Os estados que não são alcançáveis ou que não alcançam um estado final podem ser removidos do autômato, sem afetar a linguagem reconhecida.

Um exemplo de autômato finito resultante do processo de minimização :

```
+--------+-----+-------+-----+-----+-----+-------+-----+-----+
|   -    |  a  |   e   |  i  |  n  |  o  |   s   |  t  |  u  |
+--------+-----+-------+-----+-----+-----+-------+-----+-----+
|  >[S]  | [M] | [C,M] | [M] |     | [M] | [A,H] |     | [M] |
| [A,H]  |     | [B,I] |     |     |     |       |     |     |
| [B,I]* |     |       |     | [J] |     |       |     |     |
| [C,M]* | [M] |  [M]  | [M] | [D] | [M] |       |     | [M] |
|  [D]   |     |       |     |     |     |       | [E] |     |
|  [E]   | [F] |       |     |     |     |       |     |     |
|  [F]   |     |       |     |     | [G] |       |     |     |
|  [G]*  |     |       |     |     |     |       |     |     |
|  [J]   | [K] |       |     |     |     |       |     |     |
|  [K]   |     |       |     |     | [L] |       |     |     |
|  [L]*  |     |       |     |     |     |       |     |     |
|  [M]*  | [M] |  [M]  | [M] |     | [M] |       |     | [M] |
+--------+-----+-------+-----+-----+-----+-------+-----+-----+
```

3.7. Estado de Erro

Por fim, a todas as transições não mapeadas, é atribuído um estado de erro, neste caso representado pelo caractere de sublinhado (*underline*), ao qual não deve ser possível sair por meio de nenhuma transição, assim como no exemplo:

```
+--------+-----+-------+-----+-----+-----+-------+-----+-----+
|   -    |  a  |   e   |  i  |  n  |  o  |   s   |  t  |  u  |
+--------+-----+-------+-----+-----+-----+-------+-----+-----+
|  >[S]  | [M] | [C,M] | [M] | [_] | [M] | [A,H] | [_] | [M] |
| [A,H]  | [_] | [B,I] | [_] | [_] | [_] |  [_]  | [_] | [_] |
| [B,I]* | [_] |  [_]  | [_] | [J] | [_] |  [_]  | [_] | [_] |
| [C,M]* | [M] |  [M]  | [M] | [D] | [M] |  [_]  | [_] | [M] |
|  [D]   | [_] |  [_]  | [_] | [_] | [_] |  [_]  | [E] | [_] |
|  [E]   | [F] |  [_]  | [_] | [_] | [_] |  [_]  | [_] | [_] |
|  [F]   | [_] |  [_]  | [_] | [_] | [G] |  [_]  | [_] | [_] |
|  [G]*  | [_] |  [_]  | [_] | [_] | [_] |  [_]  | [_] | [_] |
|  [J]   | [K] |  [_]  | [_] | [_] | [_] |  [_]  | [_] | [_] |
|  [K]   | [_] |  [_]  | [_] | [_] | [L] |  [_]  | [_] | [_] |
|  [L]*  | [_] |  [_]  | [_] | [_] | [_] |  [_]  | [_] | [_] |
|  [M]*  | [M] |  [M]  | [M] | [_] | [M] |  [_]  | [_] | [M] |
|  [_]*  | [_] |  [_]  | [_] | [_] | [_] |  [_]  | [_] | [_] |
+--------+-----+-------+-----+-----+-----+-------+-----+-----+
```

4. Implementação

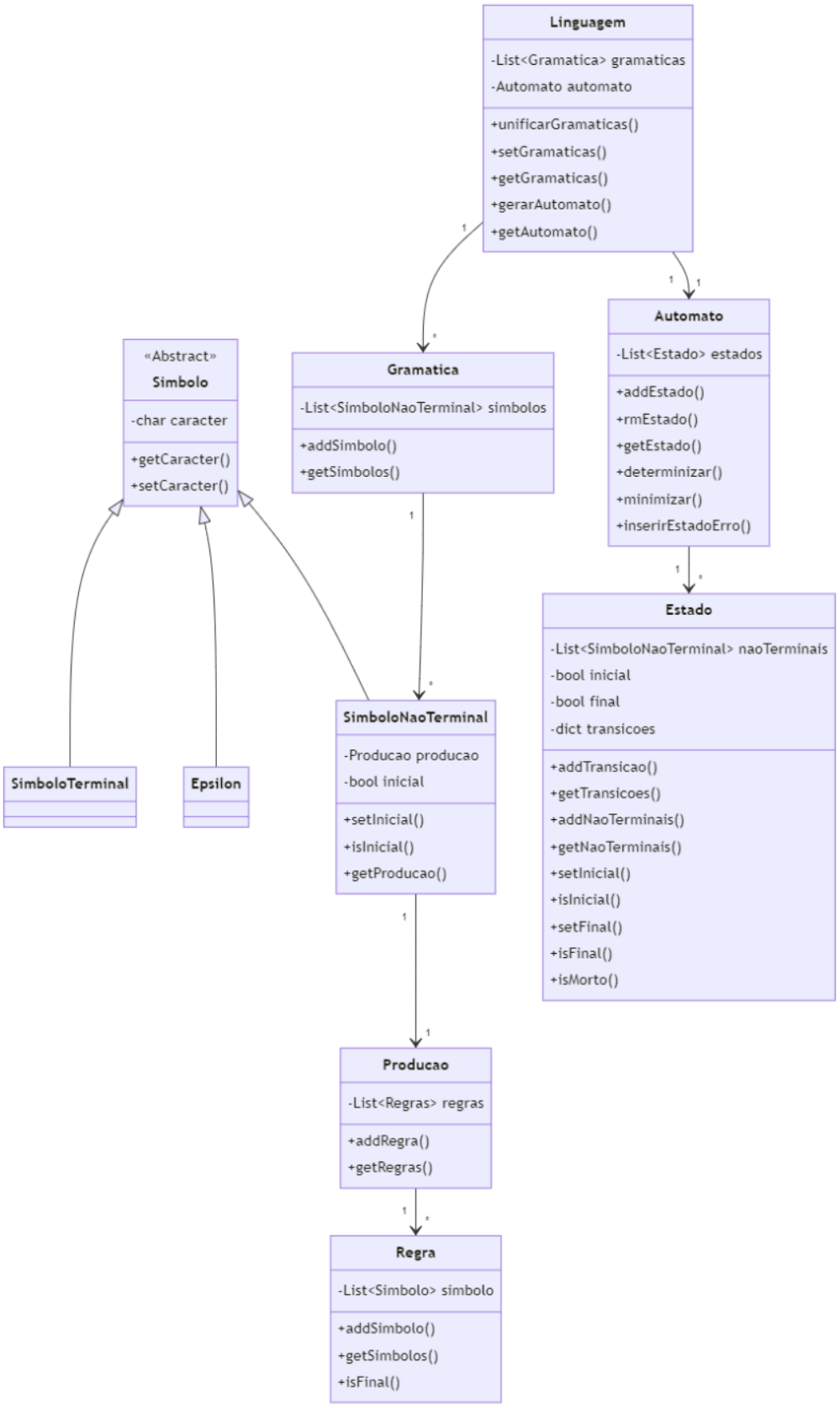
A linguagem de programação utilizada na implementação da aplicação foi Python, apenas com suas bibliotecas básicas, exceto pela biblioteca PrettyTable que auxilia na impressão do autômato finito de forma tabular. A escolha da linguagem se deve à simplicidade do código somado ao nível de abstração que se pode atingir quando trabalhado com orientação a objetos.

O uso do paradigma de orientação a objetos foi escolhido devido à facilidade de transformar os teoremas de linguagens formais e autômatos em algoritmos para a resolução do problema. A estruturação da aplicação em classes permitiu uma melhor organização e clareza do código, facilitando a sua manutenção e entendimento. O projeto foi concebido com a seguinte estrutura:

```
lfa-gerador-afd
|
├──arquivos
|       entrada.txt
|
└──src
    |   arquivo.py
    |   main.py
    |
    └──linguagem
        |   linguagem.py
        |
        ├──automato
        |       automato.py
        |       estado.py
        |
        └──gramatica
                gramatica.py
                producao.py
                regra.py
                simbolo.py
```

Em que todos os arquivos em python descrevem classes e seus comportamentos com a exceção de *"main.py"* e *"arquivo.py"*. O primeiro, apenas inicia o programa e imprime os resultados encontrados pelas classes. Já o segundo é responsável por ler o arquivo de entrada (*"entrada.txt"*) e, a partir dele, instanciar os objetos iniciais da gramática.

As classes criadas para a resolução do problema podem ser descritas pelo diagrama abaixo:



5. Conclusão

De fato, este projeto permitiu a aplicação prática dos conceitos teóricos vistos em aula, além de contribuir para a implementação de algoritmos para processamento de linguagem natural. A manipulação de gramáticas e sentenças e a geração de autômatos finitos determinísticos a partir delas é uma tarefa fundamental em diversos problemas de processamento de linguagem natural e certamente está diretamente conectado ao funcionamento dos compiladores.

O projeto desenvolvido demonstrou a importância da compreensão do conteúdo exposto em um ambiente prático, demonstrando como o reconhecedor de linguagem presente em compiladores e tradutores funciona. Em conclusão, o projeto foi importante para a compreensão dos teoremas estudados em aula e para a aplicação real de um reconhecedor em problemas de processamento de linguagem natural.

6. Referencial Bibliográfico

SCHEFFEL, Roberto M. **Apostila de Linguagens Formais e Autômatos.** Unisul - Universidade do Sul de Santa Catarina, Departamento de Ciências Tecnológicas, Curso de Ciência da Computação, 2021.

RAMOS, Marcus Vinícius Midena. **Linguagens Formais e Autômatos.** Petrolina: Universidade Federal do Vale do São Francisco, Curso de Engenharia de Computação, 22 abr. 2008.