GRAMÁTICAS LIVRES DE CONTEXTO - IMPLEMENTANDO ALGORITMOS DE SIMPLIFICAÇÕES E TRANSFORMAÇÕES EM JAVA

Daniella Martins Vasconcellos[1]
Vinícius Takeo Friedrich Kuwaki[2]

Resumo: Linguagens Livres de Contexto estão presentes em várias ferramentas do dia-a-dia, encontradas em processadores de texto, serviços de e-mail, dentre outros. São geradas por formalismos geradores chamados de Gramáticas Livres de Contexto, que através de regras de produções podem ou não reconhecer palavras. Este trabalho busca implementar algoritmos que simplificam as gramáticas, eliminando produções que não são necessárias, além de implementar também um algoritmo de transformação de gramática, transformando uma gramática qualquer para uma gramática na forma normal de Chomsky. Ademais, foi implementado um algoritmo de reconhecimento de palavra, o algoritmo de Early. Esses algoritmos foram todos implementados utilizando linguagem Java, utilizando apenas recursos nativos da linguagem, além do código ter sido estruturado em pacotes, de forma a propiciar que os algoritmos pudessem ser facilmente conectados a outros sistemas.

Palavras-chave: Linguagens Formais e Autômatos, Simplificações, Algoritmos e Java.

1 INTRODUÇÃO

Uma Gramática Livre de Contexto é um formalismo gerador de uma Linguagem Livre de Contexto, onde as regras de produções são definidas de forma mais livre que em uma Gramática Regular. O campo de aplicação das Linguagens Livres de Contexto é vasto: analisadores sintáticos, tradutores de linguagens e processadores de texto são exemplos de aplicações dessa classe de linguagem, que embora não pareça, mas está presente em muitos softwares do cotidiano, aplicações da Microsoft como: Word, Excel e PowerPoint utilizam analisadores sintáticos, além de serviços de e-mail como Gmail, e outros.

Nesse trabalho utilizamos a linguagem de programação Java para implementar os algoritmos, visto que todos os algoritmos trabalham em cima de strings e a biblioteca nativa da linguagem fornece uma grande quantidade de métodos prontos para lidar com strings. Outro motivo da escolha de Java foi a grande semelhança da linguagem com pseudo-código e também o grande encapsulamento que a linguagem possui, gerando uma maior organização do projeto em camadas e em arquivos separados. Assim, na hora de implementar os algoritmos, caso ocorresse um erro que não havia sido percebido anteriormente, seria fácil corrigi-lo. Os algoritmos implementados nesse trabalho estão no livro (MENEZES, 2001).

2 DESENVOLVIMENTO

Uma Gramática Livre de Contexto é definida como $G = (V, T, P, S)$, onde V é o conjunto das variáveis não terminais, T é o conjunto dos simbolos terminais, P é o conjunto das regras

[1] VASCONCELLOS, danivasco@msn.com
[2] KUWAKI, vtkwki@gmail.com

de produções, e S é a variável inicial, tal que $S \in V$. Durante esse trabalho, escolhemos uma linguagem orientada a objetos para modelar uma gramática, visto que facilita a organização do código.

Na segunda seção, são apresentados os modelos das classes e as estratégias adotadas pelos algoritmos implementados. A terceira seção contém o ambiente de compilação e as ferramentas adotadas. Já na quarta seção é apresentado como utilizar os algoritmos em cima de outro sistema.

2.1 MODELAGEM

O trabalho foi modelado em pacotes, cada qual com um objetivo específico, sendo esses: classes, algoritmos, arquivosExternos, sistema e apresentação. A hierarquia dos pacotes está representada no diagrama de classes abaixo.

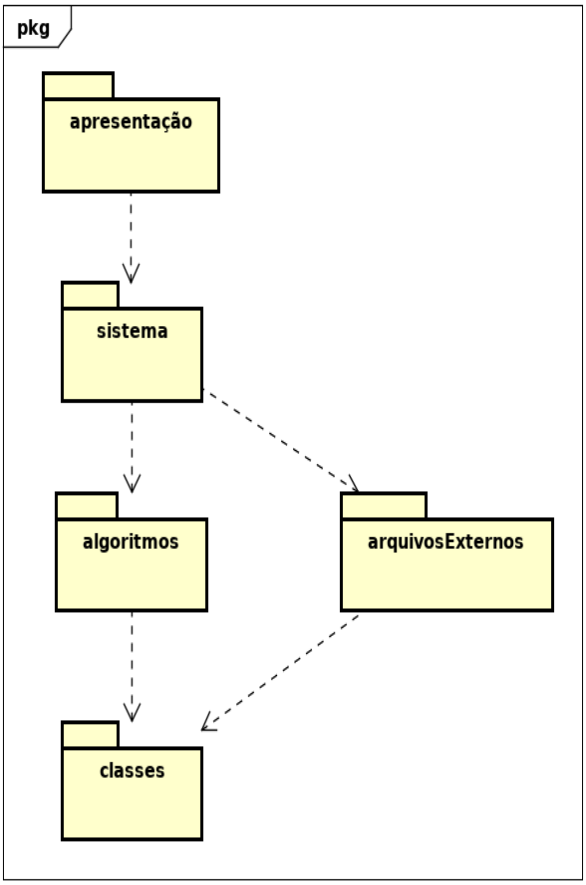

Figura 1 – Diagrama de Classes - Pacotes. Fonte: (VASCONCELLOS; KUWAKI, 2019)

Agora será apresentado as classes presentes dentro desses pacotes e como elas se relacionam entre si:

2.1.1 Pacote classes

O pacote classe contem as classes mais básicas e importantes do trabalho, que são elas: Produções e Gramática, representadas no diagrama UML abaixo:

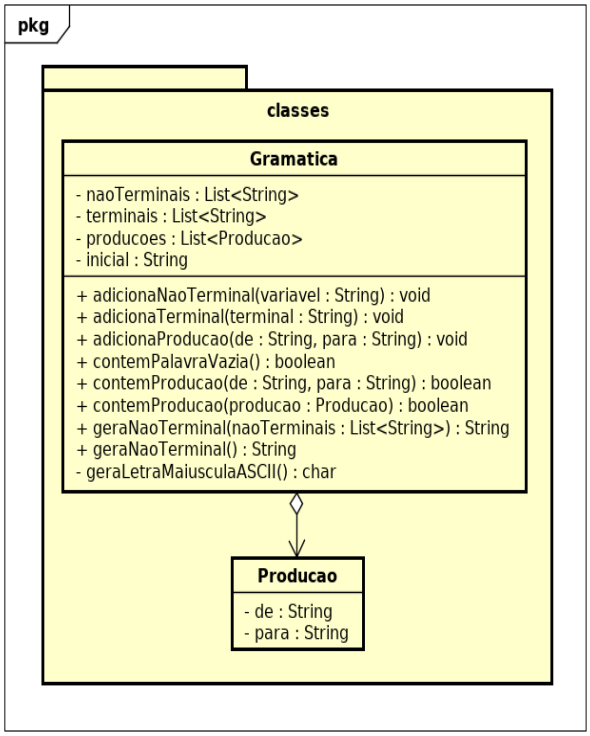

Figura 2 – Diagrama de Classes - Pacote classes. Fonte: (VASCONCELLOS; KUWAKI, 2019)

A classe Producao é composta de duas strings, um "de"e um "para", sua representação na forma matemática é de->para, onde "de"e "para"podem assumir o valor de qualquer terminal ou não-terminal pertencente a Gramática (como estamos lidando com uma GLC, sempre usaremos uma variável não-terminal em "de"). Esse pertencimento é explicado no diagrama com a associação "composição"da Gramática para a Produção, já que toda Gramática vai possuir sua lista de Produções. Ademais, não foi deixado explícito no diagrama de classes, mas todas as classes possuem métodos *getters* e *setters* para todos os atríbutos.

A classe Gramática é composto por uma lista de terminais (ex: a,b,c), uma lista de não-terminais (ex: A,B,C), uma lista de objetos do tipo Producao definidos anteriormente, e uma variável inicial S.

Logo, nossa modelagem está de acordo com o modelo matemático de uma gramática, onde uma gramática é definida como: $G = (V, T, P, S)$, onde V é o conjunto de variáveis não terminais, T é o conjunto de terminais, P é o conjunto das regras de produção e S é o simbolo inicial, note que $S \in V$.

Os métodos *geraNaoTerminal()* possuem duas variações, uma dela precisa de uma lista de simbolos não-terminais como parâmetro e a outra utiliza a própria lista de não-terminais da gramática. O objetivo de ambos é o mesmo: a partir dos não-terminais passados como parâmetro, utiliza-se o método privado *geraLetraMaiusculaASCII()* e retorna uma variável dentro do alfabeto $A, B, ..., Z$, sendo esta uma letra maíuscula, que não pertence a lista de não-terminais.

2.1.2 Pacote algoritmos

O pacote algoritmos possui os algoritmos que foram implementados nesse trabalho. Todos os algoritmos estendem a superclasse Algoritmos que é abstrata (não pode ser intânciada sem ser estendida) e possui uma gramática com a visibilidade protected (isto é, as classes filhas podem acessar e alterar esse atributo) e um método abstrato *executa()*, que executa o tal algoritmo implementado. Todos os algoritmos possuem uma Gramática da qual é trabalhada em cima, e a Gramática possui seus métodos *get* e *set*, para que as classes que os pacotes que se encontram em cima na hierarquia proposta na seção 2.1 possam acessar essas modificações. Todas essas classes funcionam da seguinte forma: é preciso instânciar um objeto da classe do algoritmo que se deseja utilizar, não sendo necessário passar nenhum parãmetro para o construtor. Após instânciar, é necessário usar o método *setGramatica()*, passando como parâmetro a Gramática na qual será trabalhada. Então chama-se o método *executa()*, que realiza o algoritmo em cima dessa gramática. E, por fim, basta usar o método *getGramatica()* para obter a gramática resultante do algoritmo.

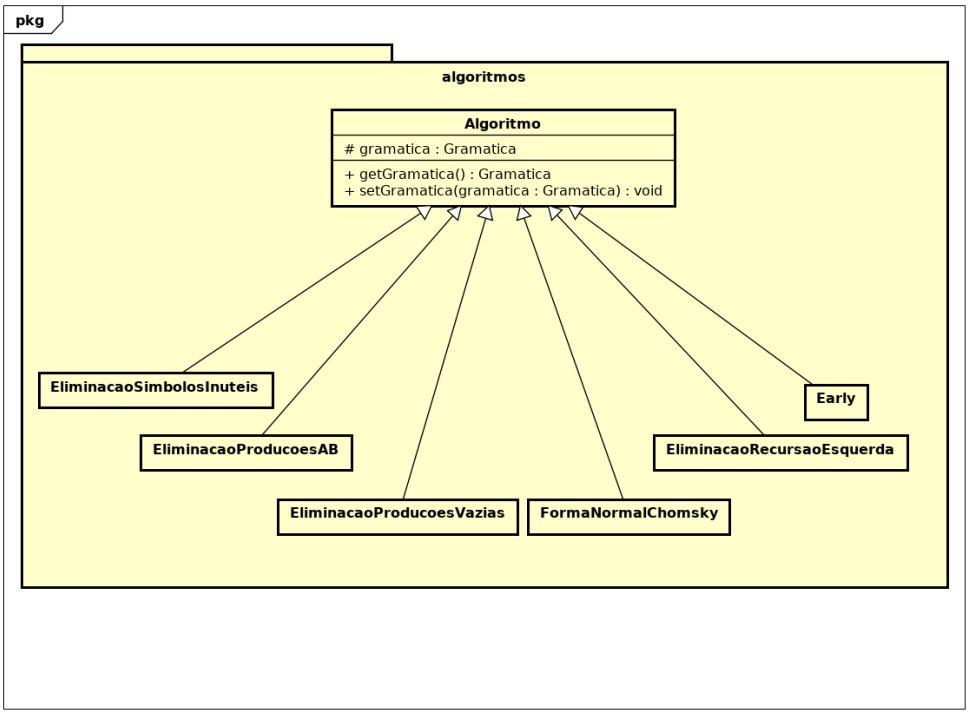

Figura 3 – Diagrama de Classes - Pacote algoritmos. Fonte: (VASCONCELLOS; KUWAKI, 2019)

2.1.3 Pacote arquivosExternos

Esse pacote não possui uma enfâse tão grande nesse trabalho, visto que seu objetivo é simples: a partir de um arquivo texto escrito na forma descrita abaixo, essas classes manipulam as strings e transformam o arquivo em um objeto do tipo Gramática.

A B C D E // Simbolos naoTerminais (obrigatório)
a b c d // Simbolos terminais (obrigatório)
S // Simbolo inicial (obrigatório)
S->A // Regra de produção do simbolo inicial para um não-terminal
A->B // Regra de produção de um não-terminal outro não-terminal
A->a // Regra de produção de um não-terminal para um terminal
A->ε // Regra de produção de um não-terminal para ε

As classes contidas dentro do pacote são:

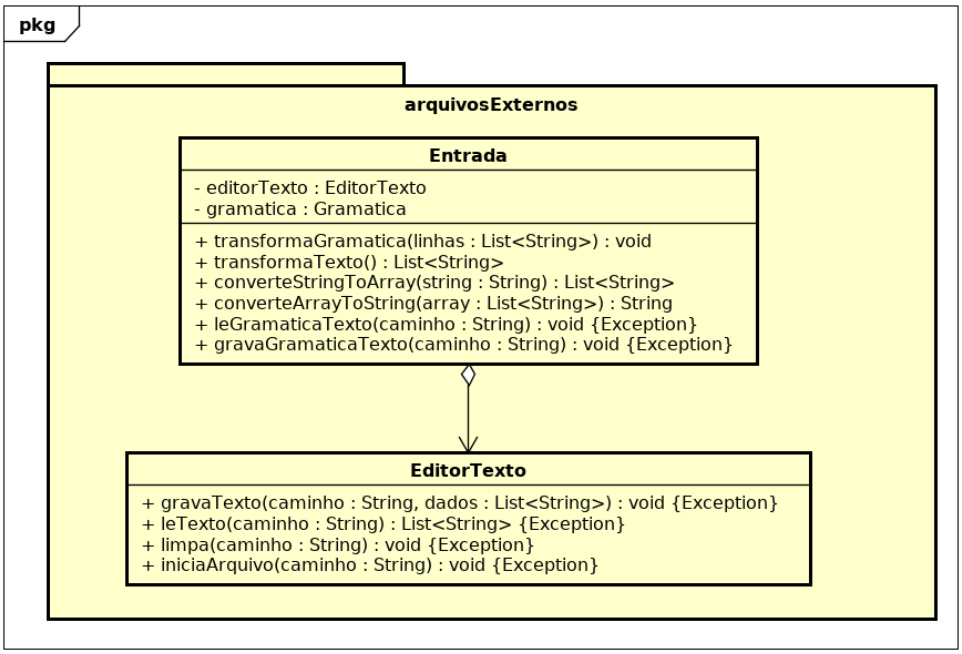

Figura 4 – Diagrama de Classes - Pacote arquivosExternos. Fonte: (VASCONCELLOS; KUWAKI, 2019)

Para usar esse pacote, basta instânciar um objeto do tipo Entrada e usar os métodos *leGramaticaTexto*() e *gravaGramaticaTexto*(). Ao utilizar o método *leGramaticaTexto*(), passando o caminho do arquivo que contém a gramática como parâmetro, a gramática lida do arquivo será salva no atributo gramática dessa classe, então basta utilizar o método *getGramatica*() para obter essa gramática. Para utilizar o método *gravaGramaticaTexto*(), basta fazer o processo inverso. É necessário primeiro existir uma gramática no atributo dessa classe, ou seja, usar o método *setGramatica*() passando a gramática a ser gravada no arquivo como parâmetro dessa função. Depois, basta chamar o método própriamente dito *gravaGramaticaTexto*(), passando o caminho do arquivo no qual sera gravado.

Lembrando que os métodos *getters* e *setters* não foram descritos nos diagramas desse trabalho pois são essenciais em uma implementação orientada a objetos.

2.1.4 Pacote sistema

Esse pacote atua como um intermédio entre o pacote de algoritmos e o pacote de apresentação, promovendo a iteração do usuário com os algoritmos, ou seja, o pacote de apresentação, que será apresentado a frente, apenas envia strings para esse pacote e esse pacote também envia strings para a camada de apresentação.

Dentro desse pacote há apenas uma classe chamada Sistema. Não entraremos em detalhes sobre o funcionamento de cada método dessa classe, visto que como já citado anteriormente, esse não é o foco do trabalho.

2.1.5 Pacote apresentação

Esse pacote contém a interface gráfica entre o usuário e o sistema, toda a entrada e saída é feita através de strings, ou seja, esse pacote pode ser fácilmente reimplementado apenas pelo console, ao invés de uma interface gráfica. Esse pacote não será apresentado aqui pois o foco do trabalho é a implementação dos algoritmos e não a programação da interação com o usuário, serão apresentados apenas alguns componentes da interface.

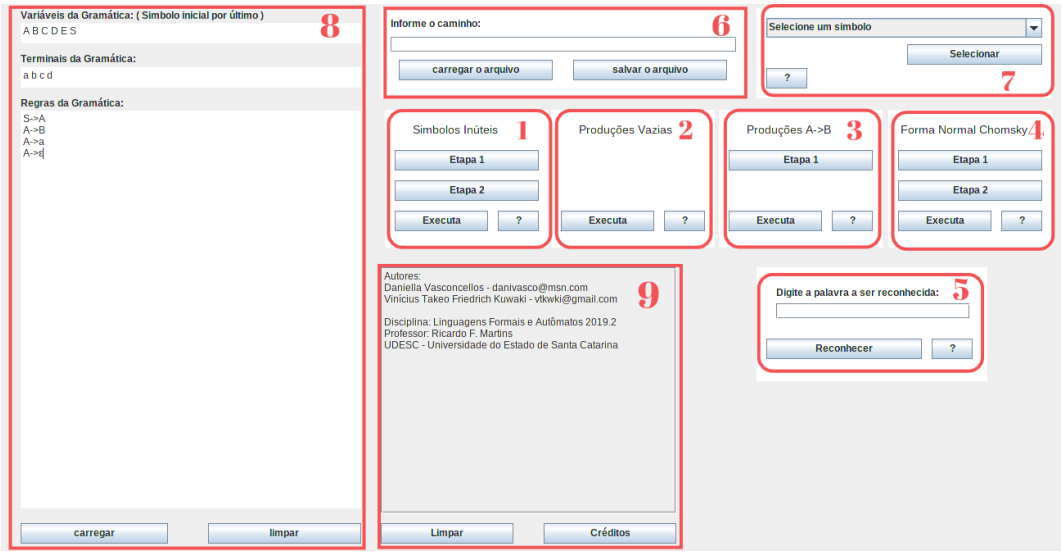

Figura 5 – Screenshot de toda a interface, dividida em oito sessões. Fonte: (VASCONCELLOS; KUWAKI, 2019)

O primeiro deles são os correspondentes às **caixas 1, 2, 3 e 4**, que são os algoritmos implementados no sistema. Cada algoritmo possui um "mini painel"onde o usuário pode escolher o que fazer com a gramática que ele fez a entrada. Há também botões de ajuda especificados pelo simbolo "?", que dao maiores explicações sobre o funcionamento de cada botão dentro das caixas e como o usuário pode utilizá-los.

Na **caixa 1**, o usuário tem as opções *"etapa 1", "etapa 2", "executa", "?"*. A primeira opção irá realizar a primeira etapa do algoritmo de Símbolos Inúteis. A segunda opção irá realizar a primeira etapa do algoritmo de Símbolos Inúteis. A terceira opção irá realizar todo o algoritmo de Símbolos Inúteis, rodando primeiro a etapa 1, seguido da etapa 2. Todos os resultados são

exibidos na **caixa 8**. Os detalhes de cada etapa, e do algoritmo no geral estão descritos na seção 2.2.0.1 desse trabalho. A opção *"?"* é a função de ajuda, que mostra uma breve explicação dos três botões que o usuário pode apertar dentro da caixa.

Na **caixa 2**, o usuário tem a opção *"executa"e "?"*. Ao clicar na opção *"executa"*, o usuário fará o programa rodar o algoritmo de Eliminação de Produções Vazias, mais detalhado na seção 2.2.0.2 deste trabalho. O resultado é exibido na **caixa 8**. A opção *"?"* é a função de ajuda, que mostra uma breve explicação do botão que o usuário pode apertar dentro da caixa.

Na **caixa 3**, o usuário tem as opções *"etapa 1", "executa"e "?"*. Ao clicar na opção *"executa"*, o usuário fará o programa rodar o algoritmo de Eliminação de Produções $A \rightarrow B$, mais detalhado na seção 2.2.0.3 deste trabalho. O resultado é exibido na **caixa 8**. Já se clicar na opção *etapa 1*, ele fará com que o fecho da gramática seja construído, aparecendo na **caixa 9**. A opção *"?"* é a função de ajuda, que mostra uma breve explicação dos dois botões que o usuário pode apertar dentro da caixa.

Na **caixa 4**, o usuário tem a opção *"etapa 1", "etapa 2", "executa", "?"*. A primeira opção irá realizar a primeira etapa do algoritmo de Forma Normal de Chosmky. A segunda opção irá realizar a primeira etapa do algoritmo de Forma Normal de Chosmky. A terceira opção irá realizar todo o algoritmo de Forma Normal de Chomsky, rodando primeiro a etapa 1, seguido da etapa 2. Todos os resultados são exibidos na **caixa 8**. Os detalhes de cada etapa, e do algoritmo no geral estão descritos na seção 2.2.0.4 desse trabalho. A opção *"?"* é a função de ajuda, que mostra uma breve explicação dos três botões que o usuário pode apertar dentro da caixa.

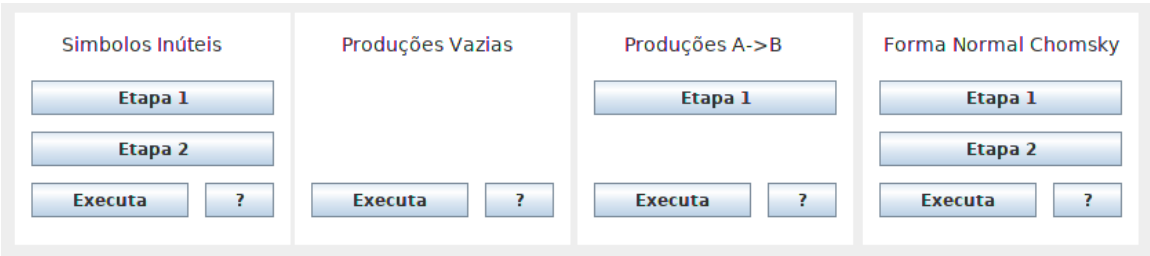

Figura 6 – Algoritmos: Símbolos inúteis, produções vazias, produções A-> B, forma normal Chomsky. Fonte(VASCONCELLOS; KUWAKI, 2019)

Na **caixa 5**, existe um painel para fazer o reconhecimento de uma palavra na gramática especificada. Esse painel contém uma caixa de texto para o usuário especificar a palavra que ele deseja testar se pertence ou não a gramática.

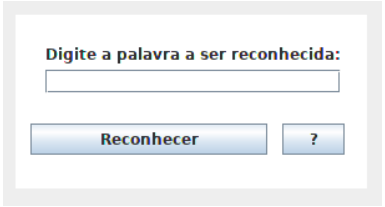

Figura 7 – Caixa de reconhecimento de palavra. Fonte: (VASCONCELLOS; KUWAKI, 2019)

Caso o usuário decida fazer a entrada de dados via arquivos de texto, ele pode escrever tudo em um único *arquivo.txt* (ou qualquer formato que seja de texto) e adicionar o caminho onde o arquivo está salvo para carregá-lo; essa função está relacionada a **caixa 6**. É necessário entrar com a gramática como especificada anteriormente. Também há um painel, correspondente à **caixa 7** onde o usuário pode copiar o símbolo ε ou δ caso seja necessário.

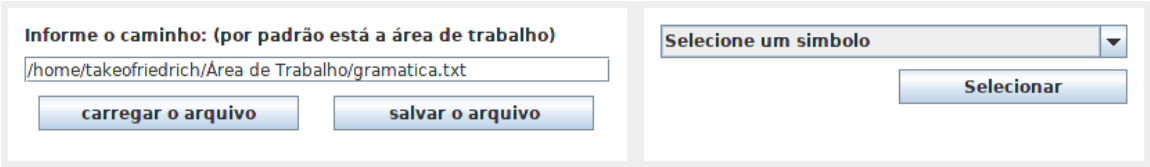

Figura 8 – Arquivos e símbolos. Fonte: (VASCONCELLOS; KUWAKI, 2019)

Na **caixa 8**, há o painel de entrada de dados via interface. Nele, o usuário possui um campo para digitar os simbolos terminais e não terminais da gramática, além de possuir outro campo para escrever as regras de produções. É também nessa caixa que os resultados das funções realizadas pelo usuário serão mostradas. É mais recomendado utilizar a entrada de dados por arquivo, visto que o foco dessa é apresentar o resultado dos algoritmos ao usuário.

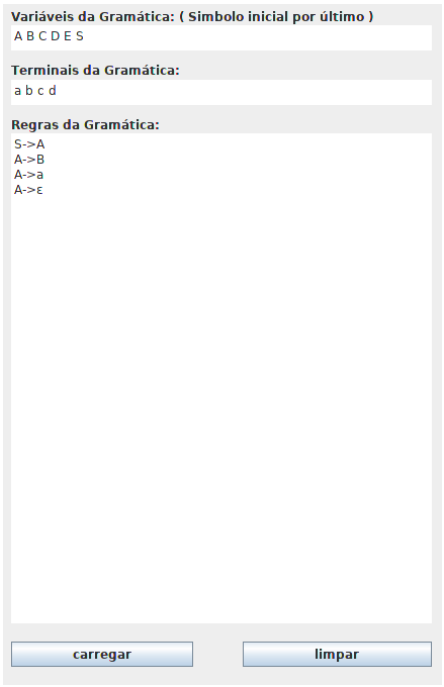

Figura 9 – Entrada de dados. Fonte: (VASCONCELLOS; KUWAKI, 2019)

Por último, na **caixa 9**, há o painel de visualização de dados. Nele, há algumas funções cujas etapas podem ser exibidas ali (como a etapa 1 das Produções $A \rightarrow B$, por exemplo, onde a construção do fecho é exibida nesta caixa, e não na caixa 8). Na caixa, também há os botões de *"limpa"e "créditos"*. O primeiro fará a função de eliminar qualquer coisa escrita dentro da caixa. O segundo irá mostrar um texto que dá crédito aos autores desse trabalho por realizarem a aplicação.

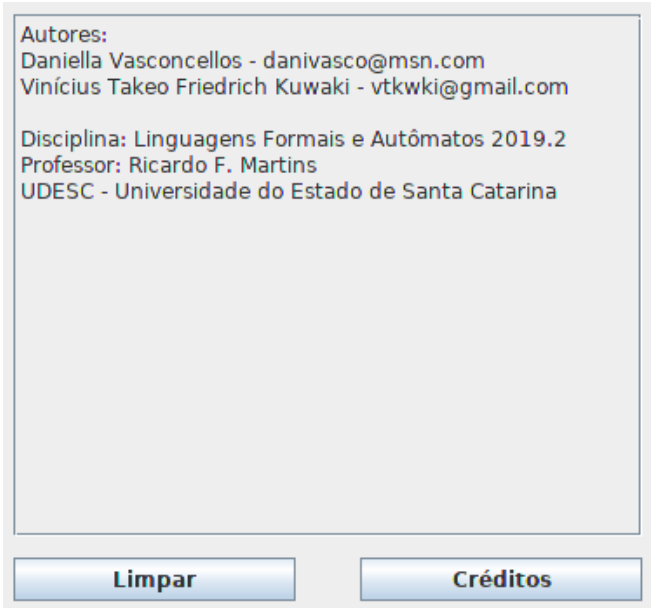

Figura 10 – Créditos. Fonte: (VASCONCELLOS; KUWAKI, 2019)

2.2 IMPLEMENTAÇÕES

Nessa seção discutiremos as estratégias utilizada pelo algoritmo. Cada algoritmo descrito abaixo possui sua própria classe no pacote algoritmos, assim como dito anteriormente. Essas classes são classes filhas da superclasse Algoritmo, e todas possuem um atributo *gramatica*, metódos *getGramatica() e setGramatica()* e também são obrigadas a implementar o método *executa()*

2.2.0.1 Algoritmo de Eliminação de Simbolos Inúteis

Esse algoritmo possui o objetivo de eliminar simbolos ou variáveis que não foram usados na geração de palavras de terminais.

A primeira etapa garante que qualquer variável gera terminais, ou seja, todas as produções precisam estar na forma *A->a ou A->aB*. Ao final do algoritmo, é gerado um novo conjunto de variáveis V_1 e um novo conjunto de produções P_1. A estratégia utilizada por essa etapa é buscar em todas as regras de produções as regras em que o lado direito é uma palavra que contém apenas combinações de terminais ou de simbolos pertencentes ao conjunto V_1. Inicialmente, V_1 é um conjunto vazio, ou seja, durante a primeira busca, o algoritmo procura por produções que contenham apenas palavras de terminais no lado direito, e quando encontra uma, a adiciona a variável do lado esquerdo ao conjunto V_1. Se algo novo foi adicionado ao conjunto V_1, é necessário agora buscar as regra de produções em que o lado direito é uma combinação de terminais e dessa nova variável. A busca acaba quando o algoritmo percebe que o conjunto de novas produções não cresceu desde a iteração anterior. Após isso o algoritmo constrói o conjunto P_1 procurando em todas as produções as que contém variáveis perntecentes a V_1 apenas, descartando as demais. Ao final, é gerado uma nova gramática $G = (V_1, T, P_1, S)$.

A segunda etapa garante que qualquer simbolo terminal é atingida a partir do simbolo inicial, essa etapa possui uma estratégia de construir um novo conjunto de terminais T_2 e um novo conjunto de variáveis V_2, possui o mesmo princípio da etapa anterior, de ficar buscando até que o tamanho dos conjuntos na iteração anterior seja igual ao tamanho do conjunto na iteração em questão. A busca acontece da seguinte forma: primeiro o conjunto T_2 começa vazio, e o conjunto V_2 começa contendo apenas o simbolo inicial. A cada iteração do algoritmo é analisado se a produção em questão contém um terminal ou uma variável no lado direito, se a produção conter uma variável ela adiciona essa a V_2, se ela conter um terminal, adiciona-o em T_2. O processo segue até que a condição de parada seja atendida. Após isso o conjunto P_2 será construido com as produções que contém apenas elementos pertencentes a T_2 e V_2.

2.2.0.2 Algoritmo de Eliminação de Produções Vazias

Esse algoritmo possui o objetivo de eliminar produções na forma $A \rightarrow \varepsilon$, ou se existir uma produção $A \rightarrow B$ e $B \rightarrow \varepsilon$, eliminá-la também.

A primeira etapa consiste em percorrer a lista de produções, procurando por alguma que possui um lado direito igual a ε. Caso encontre um A que leve a ε, então irá percorrer novamente a lista de produções. Se encontrar uma, percorrerá todo o conjunto novamente, procurando pelo símbolo que estava a esquerda do ε para que possa verificar se há outra produção em que ele esteja associado. Sendo assim, alogirtmo só adicionará ao novo conjunto, V_ε, aquelas produções em que não haja A do lado esquerdo.

Na segunda etapa são analisadas as produções em que não são vazias e nem levam a apenas um simbolo pertence ao conjunto das variáveis geradas pela etapa anterior. Para cada produção que tiver um simbolo que referencia um elemento do conjunto V_ε, essa produção é adicionada ao novo conjunto de produções, removendo a referência a essa variável. O processo se repete até que não hajam mais inclusões nesse conjunto.

2.2.0.3 Algoritmo de Eliminação de Produções na forma A->B

Esse algoritmo se baseia no princípio da transitividade, se $X \rightarrow Y$ e $Y \rightarrow Z$, logo $X \rightarrow Z$, funciona a mesma coisa para as regras de produções, a estratégia do algoritmo é primeiro construir o fecho de cada variável da gramática e depois utilizar esse fecho para analisar as produções que precisam ser eliminadas.

A primeira etapa consiste na construção do fecho das variáveis, sendo o fecho definido da seguinte forma: para toda produção $p \in G$ se existe $A \rightarrow B$, onde A *e* B são variáveis, logo $B \in fechoA$ e se existe outra produção onde $B \rightarrow C$, logo $C \in fechoA$ também. A estratégia que implementamos foi percorrer a lista de produções e encontrar as produções na forma $A \rightarrow B$ então o B era adicionado na lista de variáveis do fecho de A, após isso procuravamos producões que contiam B no lado esquerdo e no lado direito possuia apenas uma variável, se isso acontecesse esta era adicionada ao fecho de A também.

Já na segunda etapa o algoritmo primeiro percorre todas as produções e adiciona ao novo conjunto de produções as produções cujo lado direito não pertencem a V, ou seja, as produções que não são compostas somente de variáveis. Após isso o algoritmo percorre a lista de fechos de todas as variáveis, a cada ciclo ele analisa uma variável A, após isso ele percorre todas as produções para verificar se o lado esquerdo pertence ao fecho da variável A e se o lado direito possui terminais, se isso acontecer, então é adicionado uma nova produção ao novo conjunto de produções onde o lado esquerdo é composta por A e o lado direito é composto pela palavra a direita da variável que pertencia ao fecho de A.

2.2.0.4 Algoritmo de Conversão para a Forma Normal de Chomsky

Esse algoritmo é composto de três etapas, a primeira delas é a simplificação da gramática, a segunda é a garantia de que hajam apenas variáveis no lado direito das produções, para que na terceira essas produções sejam divididas em produções com apenas duas variáveis no lado direito.

A estratégia da segunda etapa é simples, primeiro percorremos todas as regras de produções que possuem dois ou mais simbolos do alfabeto no lado direito, então o algoritmo procura no lado direito os simbolos que são terminais, então para cada simbolo terminal é criada uma nova regra de produção na forma *A->a*, onde o *a* será esse terminal encontrado e o *A* será um novo não-terminal criado, após isso essa nova regra de produção é adicionada ao conjunto de regras de produções P2, esse novo não-terminal *A*é adicionado ao conjunto de variáveis V2 e a regra que contia esse terminal é substituida pelo não-terminal. Por exemplo, vamos supor uma regra de produção *A->BcD* onde *A,B e C* são não-terminais e *c* é um terminal, o que o algoritmo faz é criar uma nova regra de produção *E->c* e alterar a regra antiga *A->BcD* para *A->BED*

Já na terceira etapa, o processo consiste em analisar as produções cujo lado direito possui 3 ou mais variáveis, e a partir dessas produções agrupa as variáveis de duas em duas gerando novas regras de produções para elas, por exemplo: *A->BCD* primeiro o algoritmo transforma *CD* em uma nova produção: *E->CD* então *CD* são substituidas na regra de produção original por *E*: *A->BE*, agora a regra de produção está na forma normal de Chomsky.

Esse algoritmo porém tem uma limitação quanto ao número de variáveis novas geradas, como o alfabeto possui apenas 26 letras maiúsculas, esse algoritmo apenas gera $26 - n$ onde n corresponde ao número de variáveis da gramática, esse problema é discutido na subsessão seguinte.

2.2.0.5 Algoritmo de Eliminação de Recursão a Esquerda

O algoritmo de Eliminação a Recursão a Esquerda não foi implementado nesse trabalho, embora constasse na proposta original, pois houve uma falha na modelagem das classes básicas, apenas percebida na hora da implementação. Este algoritmo trabalha com indíces nas variáveis,

isto é: $A_1, A_2, ..., A_n$ e nossas regras de produções foram definidas como sendo uma classe contendo duas Strings, uma String *para*, e outra *de*, onde cada caracter da String correspondia a uma variável ou terminal. Logo é impossível atribuir um índice a uma variável sem uma *"gambiarra"*. Se ele o fosse, seria extremamente limitado. A natureza do algoritmo era, ao encontrar uma produção que poderia ser derivada recursivamente, eliminar a recursão, e a estratégia encontrada foi a de quebrar em mais produções com variáveis associadas a índices. Sendo assim, o algortimo de Eliminação de Recursão à Esquerda até poderia ter sido implementado utilizando nossa modelagem, entretanto, como o nosso alfabeto possui apenas 26 simbolos, teríamos um número limitado de novas variáveis.

Uma solução para esse problema seria a modificação da classe Produção, tendo em vista que agora o atributo *"de"* seria uma lista de Strings em vez de uma String única, a mesma ideia vale para o atributo *"para"*. Outra solução seria implementar um par <String,Integer>, onde a String corresponderia ao simbolo e o inteiro correponderia ao índice.

Esse algoritmo possui três etapas que serão descritas abaixo. A primeira delas consiste em renomear todas as variáveis da gramática em ordem crescente, onde cada variável vai ter um indice variando de 1 a n, onde n é o número de variáveis que existem para serem renomeadas, o algoritmo então renomea todas as variáveis da gramática e também as referências delas nas produções.

Na segunda etapa o algoritmo procura em todas as produções em que o r é maior ou igual ao s, então para estas ele exclui esta produção e percorre novamente todas as produções procurando por produções que possuem o indice do lado esquerdo igual a s, então é adicionada uma nova produção que contém o indice r no lado esquerdo e no lado direito é concatenado o lado direito antigo da produção que possuia o s no lado esquerdo e a lado direito da produção que havia sido excluida, removendo fora a referência ao indice s. Após isso o algoritmo excluia as produções em que $A_r \rightarrow A_r\alpha$ e adicionava as variáveis da gramática uma nova variável com o mesmo indice r, e incluia também as regras de produção duas novas regras, as duas contendo essa nova variável no lado esquerdo, porém uma contendo apenas α e outra contendo α concatenado com essa nova variável e após isso caso alguma produção tivesse sido excluida, o algoritmo percorre novamente as produções procurando por produções em que o lado direito não possui como prefixo o simbolo A_r, encontrando essa produções, elas eram adicionadas concatenando a nova variável com o mesmo indice r.

A terceira etapa mantém apenas um terminal do lado direito de cada produção, reconstruindo o conjunto de produções, percorrendo todas produções em ordem decrescente, de r, e excluindo as produções que o lado direito começavam por indice s, mantendo o restante da palavra guardada como alfa, então percorria novamente as produções procurando por produções onde o lado esquerdo contia o indice s, e para cada uma destas produções guardava o lado direito em beta, então uma nova produção era gerada contendo a variável r no lado esquerdo e no lado direito a concantenação de alfa com beta.

E por fim a última etapa consiste na garantia de que a produções relativas a outra variável iniciam com um terminal no lado direito, para isso a algoritmo percorria todas produções onde

$B_r \rightarrow A_s\beta_r$, excluindo as, e então procurando por produções na forma $A_s \rightarrow a\alpha$, se existirem, estas são adicionadas as produções concatenando B_r ao final da palavra no lado direito e substituindo o A_s por B_r.

2.2.0.6 Algoritmo de Reconhecimento de Early

O Algoritmo de Early é um pouco diferente dos outros algoritmos tratados anteriormente, enquantos os outros tratam de simplificar e transformar a gramática, este é um algoritmo de reconhecimento, dada uma palavra *w* pertencente a linguagem, o algoritmo verifica se a gramática aceita ou não essa palavra. É um algoritmo do tipo Top-Down, isto é, a partir do simbolo inicial, busca reconhecer a palavra construindo uma árvore, cujo reconhecimento depende da chegada até as folhas desta. Este algoritmo possui dois elementos essenciais, um marcador "."e um sufixo "/u"onde u representa o u-ésimo ciclo onde essa produção passou a ser considerada. O ponto antecede a posição a qual será analisada na tentativa de gerar o próximo terminal.

Na primeira etapa o algoritmo apenas contrói a "base"para o resto do processo, é nela que é construido um conjunto de "produções modificadas", isto é, produções com o marcador e o sufixo, sendo estas, construidas em cima das produções da gramática que partem do simbolo inicial e que podem ser aplicadas sucessivas derivações mais a esquerda partindo de S.

Já na etapa seguinte o algoritmo busca reconhecer cada simbolo da palavra em cada ciclo, primeiro ele buscar uma produção que contém $.a_r$, então caso exista essa produção é incluida no conjunto de "produções modificadas"passando o a_r para antes do ponto. Após isso o algoritmo busca não-terminais que estejam após esse ponto, caso encontre, associa uma nova produção modificada a uma produção da gramática que parta desse não-terminal, com o sufixo contendo o indice do ciclo em que está, ou seja $tamanhodapalavra + 1$. Após procurar isso, o algoritmo procura por produções modificadas onde existam palavras antes do marcador concatenado com o sufixo, caso isso ocorra, ele busca no indice do sufixo as produções modificadas associadas aquele nível e que referenciam esse não terminal. Encontradas estas, elas são então adicionadas ao conjunto do nivel em que se estava iterando, passando o ponto para após o não-terminal. O processo se repete até não houver mais inclusões nesses dois procedimentos listados anteriormente.

A terceira etapa consiste no reconhecimento da palavra, caso no ultimo nível, exista uma produção modificada onde há uma palavra da linguagem concatenada com o marcador e com o sufixo do nível 0, a palavra é reconhecida, caso contrário, a palavra não foi reconhecida pela gramática.

3 AMBIENTE DE COMPILAÇÃO

Esse projeto foi compilado utilizando um processador *Intel(R) Core(TM) i5-7200U CPU @ 2.50GHz*, com o sistema operacional *Linux Mint 19.1 Tessa*, utilizando a versão do compilador Java: *javac 11.0.3*.

4 USO DOS ALGORITMOS

Está disponível em (VASCONCELLOS; KUWAKI, 2019) o projeto na ferramenta Eclipse, e também o arquivo *.jar* para uso. Aqui será apresentado como utilizar um das classes do pacote algoritmos.

Será apresentado o uso do algoritmo de Early, para quem deseja obter todos os resultados nível a nível. Primeiro é necessário copiar para dentro do seu projeto na ferramenta Eclipse os pacotes: classes, algoritmos e arquivosExternos. O pacote early na figura abaixo será onde iremos iniciar nosso método *main*.

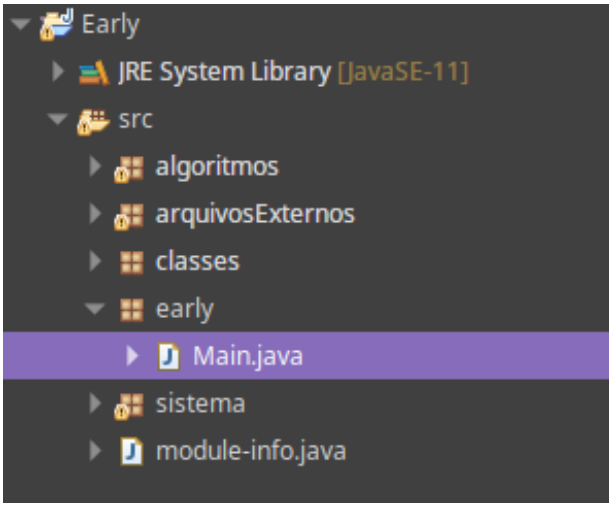

Figura 11 – Projeto na ferramente Eclipse. Fonte: (VASCONCELLOS; KUWAKI, 2019)

Após isso precisamos incluir as classes que serão utilizadas:

```
import algoritmos.Early
import arquivosExternos.Entrada
import classes.Gramatica
```

Primeiro é necessário um método Main:

```
public static void main(String[] args) {

}
```

Dentro desse trecho de código serão incluidas as linhas listadas abaixo:

```
Entrada entrada = new Entrada();
String caminho = "";
```

O objeto entrada é uma instância da classe que fará a leitura do arquivo externo e processará o texto para a forma de um objeto do tipo Gramática. O caminho precisa ser especificado dentro das aspas. Exemplo: *"C:/Documentos/Códigos"*.

```
try {
      entrada.leGramaticaT(caminho);
} catch (Exception e) {

  e.printStackTrace();
}
```

A cima está listado a ação da leitura da gramática do arquivo externo, com o devido tratamento da exceção de caso o arquivo não for encontrado.

```
Early algoritmo = new Early();
algoritmo.setPalavra("x*x");
algoritmo.setGramatica(entrada.getGramatica());
```

No trecho acima é instânciado um objeto da classe que realizará o reconhecimento. Após isso é setado a palavra que o algoritmo busca reconhecer. E por fim é setado a gramática com a qual o algoritmo irá reconhecer essa palavra, sendo a gramática a mesma lida do arquivo externo.

```
try {
      algoritmo.executa();
} catch (Exception e) {

  e.printStackTrace();
}
```

Então o algoritmo é executado e o resultado é salvo na variável early da classe Early.

```
algoritmo.getEarly().forEach((k,v)->{

  System.out.println("N vel: "+k);

  for(int i=0;i<v.size();i++) {
    System.out.println("\t"+v.get(i));
  }

});
```

Então é feito um forEach para percorrer o HashMap<K,V> onde K é o nivel (inteiro) e V é uma lista de "produções modificadas". O forEach percorre todas as chaves K, e para cada chave percorre a sua lista e a imprime.

5 CONCLUSÃO

A escolha da implementação em Java facilitou a implementação dos algoritmos pela natureza dos mesmos, mesmo que uma falha na modelagem do trabalho fez com que o algoritmo de Eliminação de Recursão a Esquerda não pudesse ser implementado. De qualquer forma, esses algoritmos podem ser conectados a outros sistemas facilmente, já que, como citado anteriormente, o campo de aplicação deles é vasto.

REFERÊNCIAS BIBLIOGRÁFICAS

MENEZES, P. B. **Linguagens Formais e Autômatos**. 4. ed. Porto Alegre: Editora Sagra Luzzatto, 2001. 158 p.

VASCONCELLOS, D. M.; KUWAKI, V. T. F. Algoritmos de gramáticas livres de contexto. In: . [S.l.]: Disponível em: <https://github.com/takeofriedrich/glc>. Acesso em: 20 nov. 2019, 2019.