Gramáticas, Autômatos e Expressões Regulares

Primeiro Trabalho Prático

Universidade Federal de Santa Catarina

Departamento de Informática e Estatística - Ciências da Computação

INE5421 - Linguagens Formais e Compiladores

Patricia Dousseau
Mariell Schappo

ÍNDICE

Funcionalidades

Foi desenvolvido um programa para a geração de autômatos, gramáticas e expressões regulares com as seguintes funcionalidades:

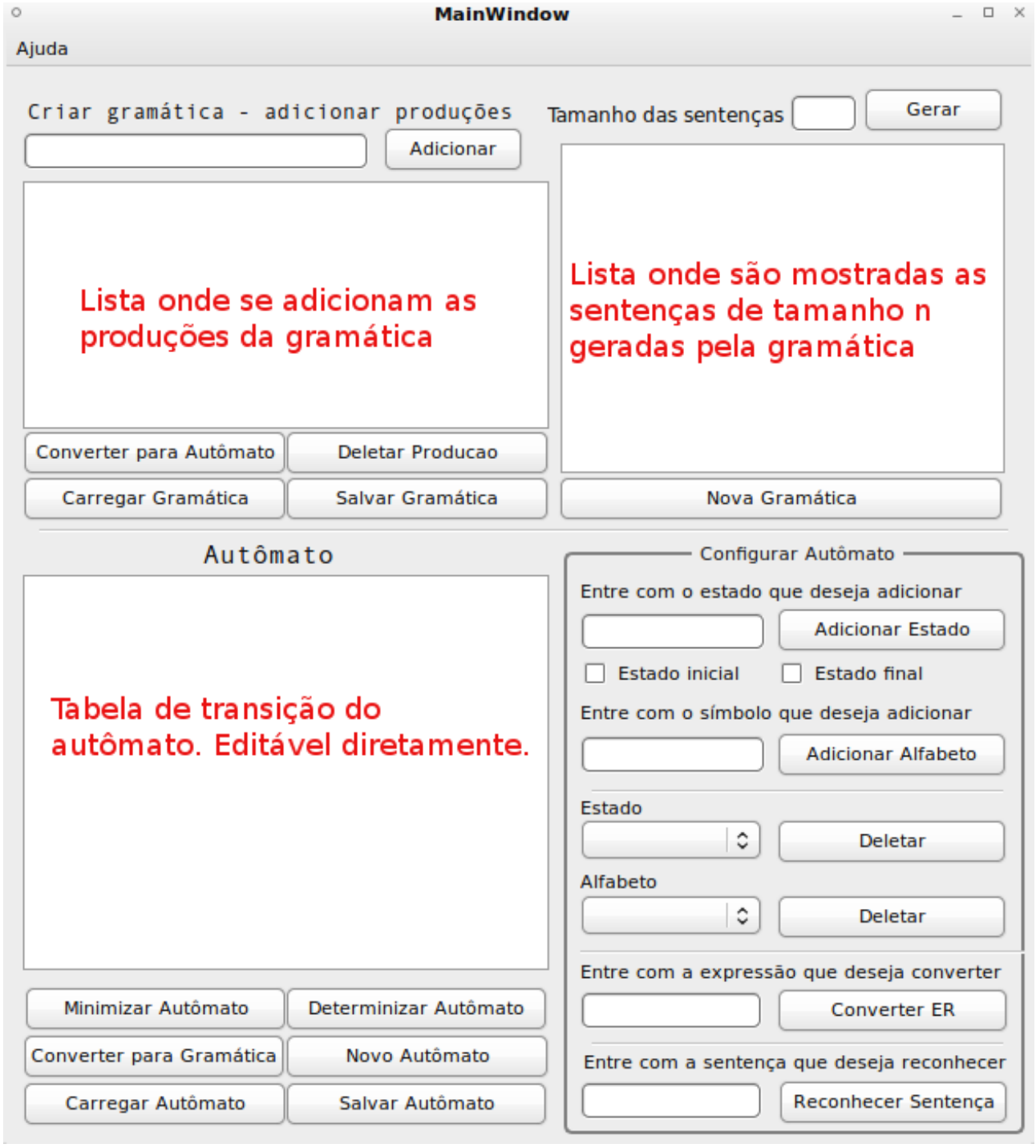
Autômatos
- Minimização e determinização de autômatos
- Conversão de autômato para gramática
- Eliminação de epsilon transições
- Verificação se determinada sentença pertence a linguagem
- Possibilidade de salvar e carregar autômatos

Gramáticas
- Conversão de gramática para autômato
- Geração de sentenças de tamanho n
- Possibilidade de salvar e carregar gramáticas

Expressões Regulares
- Conversão de expressões regulares em autômatos mínimos

Utilização

O programa conta com uma interface gráfica que permite acessar todas as funcionalidades do programa.



Como criar um novo autômato
Para criar um novo autômato, é necessário adicionar os estados e os símbolos do alfabeto antes de poder configurar as transições. Isso é feito na área intitulada “Configurar Autômato” logo a direita. É também possível remover e deletar estados, adicionar, remover e alterar transições.
Para salvar o autômato, é necessário apenas escolher um nome para o arquivo e todos eles são salvos no diretório do projeto na pasta persistência/Autômatos/
Obs: Caso você já tenha especificado um estado inicial, e especifique outro, o anterior será substituido pelo novo estado.

Reconhecendo sentenças
Para reconhecer sentenças é necessário ter especificado um autômato primeiro atráves da tabela de transição. Em seguida coloque a sentença na caixa de sentenças e clique em “Reconhecer Sentença”.

Convertendo Expressões Regulares para Autômatos
Entre com a expressão regular que deseja converter e clique no botão “Converter ER”. Caso haja algum autômato especificado na tabela de transições, ele será deletado. As expressões regulares devem ser especificadas através e unicamente dos seguintes operadores: **+ . ? *** e também com o uso de parênteses **( )** . Por exemplo: **a.b?.(c|g)*** . Caso entrasse com essa expressão na seguinte forma **ab?(c|g)***, uma mensagem de erro apareceria na tela.

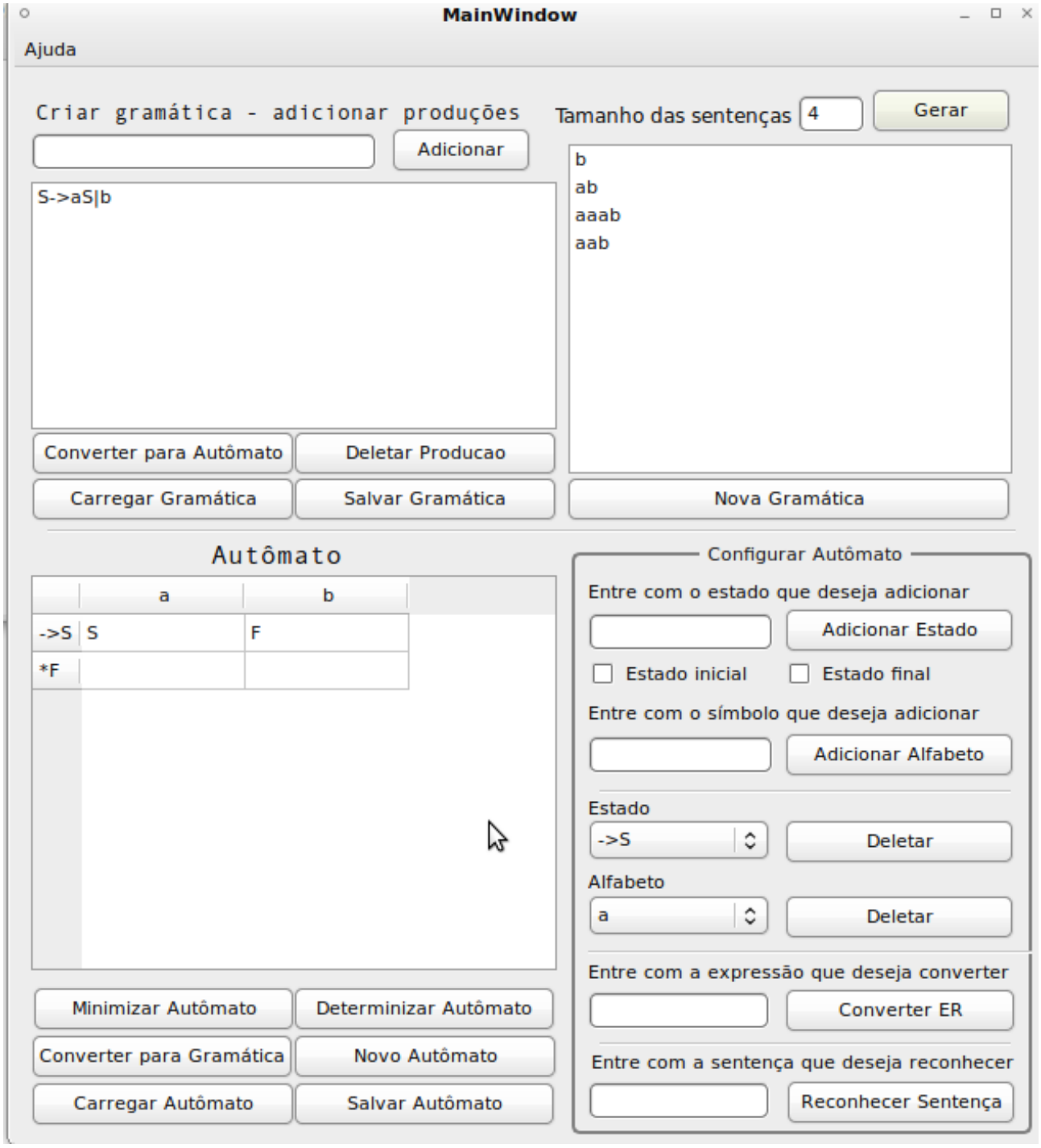
Determinizando e minimizando autômatos
Para determinizar um autômato, primeiro entre com autômato e clique no botão “Determinizar Autômato”. Para minimizar, siga os mesmo passos só que clicando no botão “Minimizar Autômato”, porém o autômato já deve ser determinístico, caso contrário, uma mensagem de erro aparecerá na tela.

Como criar uma nova gramática
Para criar uma nova gramática entre com cada produção na forma S->aS|bB|c na caixa a esquerda do botão “Adicionar Produção” e em seguida clique nesse botão. A produção mais acima na lista, será a produção inicial. É possível salvar e carregar gramáticas, sendo que para salvá-las, é necessário apenas especificar o nome e ela será automaticamente salva na pasta /persistencia/Gramaticas dentro do diretório do projeto.

Gerando as sentenças de tamanho n
Primeiro crie uma nova gramática e entre com um valor positivo na caixa ao lado esquerdo do botão “Gerar”, em seguida clique nesse botão e todas as sentenças serão mostradas na caixa logo abaixo.

Convertendo autômatos para gramáticas e gramáticas para autômatos
Para converter autômatos para gramáticas, especifique o autômato e clique no botão “Converter para Gramática”.
Para converter gramáticas para autômatos, especifique a gramática e clique no botão “Converter para Autômato”.

Exemplo do sistema em utilização

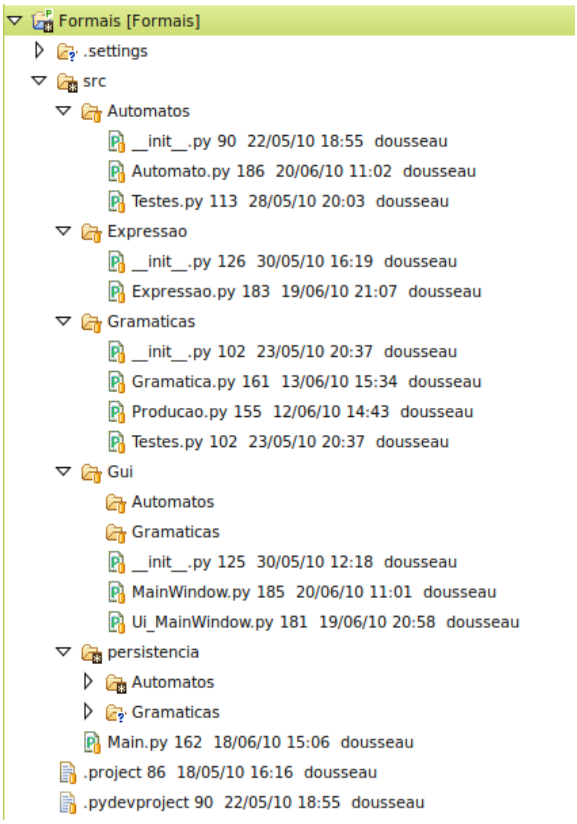


Na figura acima temos um autômato criado e convertido para gramática e sentenças de tamanho 4 geradas por essa gramática.

Desenvolvimento

Ambiente
O projeto foi todo desenvolvido utilizado Python e a biblioteca PyQt para a parte gráfica.
Foi testado apenas na plataforma Ubuntu 9.10
As IDE's utilizadas foram: Eclipse com o plugin PyDev para Python, e QtDesign para desenvolvimento da interface gráfica.

Organização
O projeto foi divido nos seguintes módulos: Autômatos, Gramáticas, Expressões, Gui e Persistência.



Os módulos Autômatos, Gramáticas e Expressões, contêm os arquivos para a construção de cada um desses objetos.
Já o módulo Gui contém a parte gráfica do projeto que faz a interação com o usuário, e a Persistência contém as pastas onde os autômatos e as gramáticas são salvas.

Autômatos
Os autômatos utilizam a seguinte definição:

Tabela de transições - formada por dois dicionários, o primeiro tem por chave o estado do autômato e mapeia para um dicionário que contém as transições desse estado.
Por exemplo:
Para representar a transição S vai para A atráves de a e vai para B atráves de b, e A vai para B através de b, seria o seguinte:

tabelaTransicao = { S:{a:A, b:B}, A:{b:B} }

Estados - conjunto de strings de todos os estados do autômato, usamos uma estrutura em python chamada set (conjunto) por ser mais adequada para o que queríamos representar e por trazer várias características que facilitaram em muito a implementação, como intersecção, união e diferença de conjuntos.
Estado inicial - uma string representando o estado inicial do autômato.
Estados finais - também um conjunto de strings com todos os estados finais do autômato
Alfabeto - conjunto de strings com todos os símbolo de transição do autômato

O autômato possui métodos de determinização, minimização, eliminação de epsílon transições, entre outros métodos.

Método determinizar da classe Autômato:

```
  def minimizarAFD(self):
        """
        Minimiza o automato caso ele seja deterministo, caso contrario
        lanca uma excecao
        """
        #verifica se o automato é determinístico (pre requisito)
        if not self.ehDeterministico():
            raise ExcecaoMinimizarAutomatoNaoDeterministico("Determinize o
automato antes.")
            return

        #elimina estados inalcancaveis, mortos e torna o automato completo
        self.eliminarEstadosInalcancaveis()
        self.eliminarEstadosMortos()
        self.criarEstadosErro()

        #cria os dois conjuntos iniciais, o primeiro com os estados finais,
e o segundo com os outros estados
        conjuntos = [set([est for est in self.estadosFinais if est in
self.estados]),
                     set([est for est in self.tabelaTransicao if est not
in self.estadosFinais]) ]


        #acha as classes de equivalencia e manda montar os automatos
seguindo essas classes
        classesEquivalencia = self.acharEstadosEquivalentes(conjuntos)
        self.criarNovosEstados(classesEquivalencia)

        #atualiza os outros dados do automato, estados iniciais, finais e
manda renomeá-lo
        self.atualizarEstadoInicial()
        self.atualizarEstadosFinais()
        self.renomearAutomato()
```

Expressões Regulares

Para implementar as expressões regulares, utilizamos o algoritmo de Thompson, que gera um autômato com epsílon transições e implementamos na classe do Autômato, um método para remover as transições vazias.
Para implementar o algoritmo de Thompson, criamos um método recursivo que divide a expressão em pedaços e cria um autômato correspondente a cada uma das partes e depois os une corretamente.
Abaixo o trecho do código que analisa a expressão e chama os métodos correspondentes para criar o autômato:

```
def converterParaAF(self, expressaoRegular):

        #Achou um bloco, entao separa o bloco inteiro para analisar
        if expressaoRegular[0] == "(":

            i = 0

            #pegando o bloco inteiro
            while expressaoRegular[i] != ")":
                i += 1

            #Aqui já conseguimos pegar o bloco inteiro, agora mandamos
converter
            automato = self.converterParaAF(expressaoRegular[1:i])

            if i + 1 >= len(expressaoRegular):
                return automato

            #verifica se o proximo simbolo depois do bloco eh *, + ou ?,
para saber o que fazer apos analisar o bloco
            if expressaoRegular[i + 1] == "*":
                automato = self.fechamento(automato)
                i += 1
            elif expressaoRegular[i + 1] == "?":
                automato = self.opcional(automato)
                i += 1
            elif expressaoRegular[i + 1] == "+":
                automato = self.fechamentoPositivo(automato)
                i += 1

            i += 1
            #verifica se a expressao acabou
            if i >= len(expressaoRegular):
                return automato

            #verifica, caso exista, o que tem depois do bloco
            if  expressaoRegular[i] == ".":
                automato2 = self.converterParaAF(expressaoRegular[i+1:])
                return self.concatenacao(automato, automato2)

            elif expressaoRegular[i] == "|":
                automato2 = self.converterParaAF(expressaoRegular[i+1:])
                return self.uniao(automato, automato2)

            else:
                #caso exista algo que nao seja . ou | temos uma expressao
```

```
mal formada
                raise ExcecaoExpressaoInvalida("")

        if expressaoRegular[0] == "*" or expressaoRegular[0] == "?" or
expressaoRegular[0] == "+":
            raise ExcecaoExpressaoInvalida("")

        #caso a expressao seja um simbolo, por exemplo, a
        else:

            #manda montar o automato correspondente
            automato = self._montarAutomato(expressaoRegular[0])

            i = 1

            #caso ja tenha chegado no final da expressao, retorna o
automato
            if i >= len(expressaoRegular):
                return automato

            proximoSimbolo = expressaoRegular[i]

            # caso exista um proximo simbolo, monta o automato
correspondente
            if proximoSimbolo == "?":
                automato = self.opcional(automato)
                i += 1

            elif proximoSimbolo == "*":
                automato = self.fechamento(automato)
                i += 1

            elif proximoSimbolo == "+":
                automato = self.fechamentoPositivo(automato)
                i += 1

            #verifica se a expressao ja acabou
            if i >= len(expressaoRegular):
                return automato

            proximoSimbolo = expressaoRegular[i]

            #verifica se existem mais expressoes para unir ou concatenar
depois dessa
            if proximoSimbolo == ".":
                automato2 = self.converterParaAF(expressaoRegular[i+1:])
                return self.concatenacao(automato, automato2)

            elif proximoSimbolo == "|":
                automato2 = self.converterParaAF(expressaoRegular[i+1:])
                return self.uniao(automato, automato2)

            else:
                raise ExcecaoExpressaoInvalida("")
```

No código acima analisamos a expressão, e para cada parte da expressão, foi montado e unido o

autômato correspondente.

Exemplo prático: convertendo a expressão **(a|b).c**?
Primeiro analisaríamos o que está dentro dos parênteses, ou seja, **a|b**, e mandaríamos montar esse autômato. Para montar esse autômato, ele primeiro mandaria montar o autômato **a**, e depois encontraria que depois de **a** vem um símbolo de união, portanto ele precisa unir **a** com o que vem em seguida, que seria **b**, e pra isso ele manda converter também a expressão b. Com isso, temos a expressão **a** e a expressão **b** já transformadas em autômato. Agora precisamos fazer a união dos dois autômatos. Para isso chamamos o método união e passamos os dois autômatos como parâmetro (código abaixo). Feita a união, já conseguimos construir o autômato correspondente a **(a|b)** e vemos que depois disso encontramos um símbolo de concatenação, o "**.**". Então mandamos converter o resto da expressão, que seria **c?** Para isso ele monta o autômato **c**, e verifica que é seguido por um símbolo **?**, então ele monta o autômato correspondente. Com isso já temos o autômato **(a|b)** e **c?** montados. Precisamos concatená-los. Então chamamos o método de concatenação passando os dois autômatos como parâmetro e verificamos que não existe mais nada para converter. Agora temos o autômato correspondente a expressão **(a|b).c?** montado, porém ele não é mínimo nem determinístico, pois possui epsílon transições. Mas eliminá-las é função do autômato.

Exemplo do código de união do autômato

```
def uniao(self, automato1, automato2):

     #verifica se os automatos nao tem estados com o mesmo nomes, caso tenham, renomeia
        if not self._verificarNomes(automato1, automato2):
            automato1, automato2 = self._renomearAutomatos(automato1, automato2)

        #cria uma nova tabela de transicoes com todas as transicoes do automato 1 e 2
        novaTabelaTransicao = automato1.tabelaTransicao
        novaTabelaTransicao.update(automato2.tabelaTransicao)

        estados = automato1.estados + automato2.estados

        #cria os dois novos estados inicial e final
        estadoInicial, estadoFinal = self._acharNomePossiveisEstados( estados )
        estados += [estadoInicial] + [estadoFinal]

         novaTabelaTransicao[estadoInicial]  =  {"&":automato1.estadoInicial  +  ","  +
automato2.estadoInicial}
        novaTabelaTransicao[estadoFinal] = {}

          #cria uma epsilon transicao de cada antigo estado final de A e B para o novo
estado final
        for estadoFinalA in automato1.estadosFinais:

            #cria a nova transicao para o estado final
            novaTrans = {"&":estadoFinal}

           #agora precisamos verificar se nao estamos perdendo nenhuma transicao antiga
em A
            transAntigas = novaTabelaTransicao[estadoFinalA]

            for trans in transAntigas.keys():
                if trans == "&":
                   #caso ja tenha uma transicao por &, mantem a transicao e adiciona a
nova transicao
                    novaTrans = {"&": transAntigas.get(trans) + "," + estadoFinal }
                elif trans != "&":
                   novaTrans.update( { trans : transAntigas.get(trans) } )

            novaTabelaTransicao[estadoFinalA] = novaTrans
```

```
        for estadoFinalB in automato2.estadosFinais:
            #cria a nova transicao para o estado final
            novaTrans = {"&":estadoFinal}

                #agora precisamos verificar se nao estamos perdendo nenhuma transicao
antiga em B
            transAntigas = novaTabelaTransicao[estadoFinalB]

            for trans in transAntigas.keys():
                if trans == "&":
                                #caso ja tenha uma transicao por &, mantem a transicao e
adiciona a nova transicao
                        novaTrans = {"&": transAntigas.get(trans) + "," + estadoFinal }
                elif trans != "&":
                        novaTrans.update( { trans : transAntigas.get(trans) } )

            novaTabelaTransicao[estadoFinalB] = novaTrans

        alfabeto = automato1.alfabeto.union(automato2.alfabeto)

        #retorna o novo automato com todas as modificacoes criadas
              return Automato(novaTabelaTransicao, alfabeto, estados, estadoInicial,
set([estadoFinal]))
```

Removendo as transições vazias (na classe autômato)
Para remover as transições vazias, verificamos se o estado em questão possui alguma transição vazia para outro estado, caso tenha, copiamos todas as transições do estado de destino para o estado que estamos analisando. É necessário varrer o autômato mais de uma vez, pois podemos ter algo do tipo: **A** vai para **B** com epsílon transições e **B** vai para **C** com epsílon transições. Copiando o estados de **B** para **A**, continuaríamos com transições vazias, só que dessa vez para **C**.
Após isso ter sido feito em todos os estados, chamamos o método de determinização e o de minimização.

Gramáticas

As gramáticas foram definidas da seguinte maneira:
Produção inicial - produção inicial da gramática definida por um objeto do tipo Produção
Produções - lista de todas as Produções da gramática
Terminais - conjunto de todos os terminais da gramática
Não terminais - conjunto de todos os não terminais da gramática

A Produção
O objeto Produção representa as produções da gramática e é definido como sendo uma parte alfa e uma parte beta. Por exemplo, a seguinte produção S->aS|b teria S por alfa e aS|b por beta. Sendo assim, podemos especificar o objeto praticamente como fazemos em aula.

As gramáticas tem a opção de gerar todas as sentenças de determinado tamanho através do seguinte método:

```
  def gerarSentencas(self,n):
        sentencasTerminadas = set([])
        sentencasPorTerminar = set([])
        naoMaisDerivaveis = set([])

        #para cada um dos betas dessa producao
        for beta in self.producaoInicial.obterListaBetas():
            if len(beta) == 1:
                sentencasTerminadas.add(beta)
```

```
            elif len(beta) == 2:
                sentencasPorTerminar.add(beta)

        #depois de incializar as duas sentencas, comeca a derivar as possiveis
        while len(sentencasPorTerminar) > 0:

            #pega uma das sentencas por terminar
            sentenca = sentencasPorTerminar.pop()
            naoMaisDerivaveis.add(sentenca)

            #para cada um dos simbolos dessa sentenca
            for indice in range(len(sentenca)):
                simbolo = sentenca[indice]

                #se for um simbolo nao terminal
                if simbolo in self.naoTerminais:

                    #pega a producao que corresponde a esse simbolo e todas as suas
possiveis transicoes
                    producao = self.obterProducao(simbolo)
                    transPossiveis = producao.obterListaBetas()

                    #para cada uma de suas transicoes
                    for trans in transPossiveis:

                        #verifica se aplicando essa transicao a sentenca ainda esta
dentro do tamanho desejado (-1 pq tira o nao terminal que
                        #pode ser substituido por um terminal )
                        if len(sentenca) - 2 + len(trans) <= n:

                            #faz a substituicao
                            nova = sentenca.replace(simbolo, trans)
                            terminada = True

                            #substitui os epsilons
                            if nova != "&":
                               nova = nova.replace('&','')

                            #verifica se essa sentenca ainda pode ser derivada
                            for s in nova:
                                if s in self.naoTerminais:
                                    terminada = False

                            if terminada and len(nova)<=n:
                                    sentencasTerminadas.add(nova)
                            elif not terminada and nova not in naoMaisDerivaveis:
                                sentencasPorTerminar.add(nova)

        return sentencasTerminadas
```

Para gerar todas as sentenças de uma determinada gramática, criamos duas listas, as sentenças terminadas, de onde não se pode derivar mais nada, ou seja, sentenças apenas com terminais. E uma outra lista, com sentenças deriváveis aindas e que não foram verificadas nenhuma vez (não estão na lista `naoMaisDerivaveis )`. Isso é necessário pois podemos chegar na mesma sentença através de dois caminhos e podemos acabar caindo em um ciclo, analisando sempre a mesma sentença.
Então o primeiro passo é inicializar as duas listas analisando a produção inicial da gramática. A partir daí, pegamos todas as sentenças que estão por terminar, e verificamos todas as suas possibilidades, adicionado as possíveis produções, ou na lista de terminadas ou na lista de não terminadas. Seguimos esses passos até a lista de produções não terminadas estar vazia.

Instalação

Para rodar o programa é necessário ter instalado a versão 2.6 do Python (versão 3.0 não funcionará). E a biblioteca PyQt4.
Para executar o programa é necessário rodar o arquivo Main.py dentro da pasta src do projeto através do comando *python Main.py*

Bibliografia

http://lambda.uta.edu/cse5317/notes/node9.html
http://www.lrde.epita.fr/dload/20070523-Seminar/delmon-eps-removal-vcsn-report.pdf