**1.Provas**

Métodos de prova

Se H então C (H 🡪 C)

* Por contradição: supondo que queremos provar uma proposição P, começamos por assumir ¬P
* Por contraexemplo: mostrar um exemplo que prova que a proposição é falsa
* Por contra positivo: se não C então não H
* Por indução

Prova indutiva

Objetivo: mostrar que a partir de n chego a n + 1

1. **Estrutura indutiva** (se aplicamos a indução a inteiros, árvores, sets, strings, etc.)
2. S(n) = **hipótese de indução** (pergunta / o que quero mostrar)
3. S(primeiro caso) = **caso base** (mostrar que é válido para o primeiro caso)
4. S(n+1) = **caso indutivo** (tendo uma igualdade, começar com o lado direito, substituindo n por n+1, e chegar ao lado direito)

Assim, como se verifica o caso base S(primeiro caso) e, se S(n) se verificar, S(n+1) também se verifica para todo o n \*estrutura indutiva\*. Prova-se, por indução matemática, que \*caso indutivo\*.

**Inventor’s paradox**: em vez de resolver um tipo específico de problema, que pareceria intuitivamente mais fácil, pode ser mais fácil resolver um problema mais geral, que cobre as especificidades da solução procurada

**2.DFAs**

Alfabeto e *String*

Alfabeto (Σ): conjunto de símbolos de entrada

*String*: sequência finita de símbolos formada a partir de um alfabeto

* ε: *string* vazia
* |*string*|: comprimento de uma *string*
* Σk = conjunto de *strings* com comprimento k formada a partir do alfabeto Σ

Linguagem

Σ\*: conjunto de todas as *strings* formadas a partir de Σ

A linguagem L é um *subset* de Σ\*, L Σ\*.

e.g., L = {w | w tem num número igual de 0s e 1s}, isto é, a linguagem aceita *strings*, referidas como w, tal que w tem um número igual de…

Autómatos Finitos Deterministas (DFAs)

Definição: (tuplo) DFA = (Q, Σ, δ, q0, F)

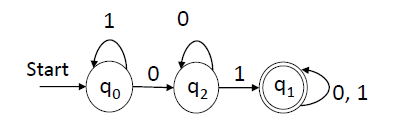
* Q: conjunto de estados
* Σ: alfabeto
* δ: função transição de estados e símbolos para estados, *δ(q, a) = p* ⬄ partindo do estado *q* com o símbolo *a* chegamos ao estado *p*
* q0: estado inicial (q0 pertence a Q)
* F: conjunto de estados finais (F Q)

Determinista: partindo de um estado com um símbolo, chegamos a um e apenas um estado (inclusive o próprio e o estado morto). Se houver um DFA com estados que não têm as transições (se necessárias) para o estado morto, chama-se **DFA incompleto**.

**Estado morto, Ø**

Um estado para as transições não aceites, com auto transições para todos os símbolos do alfabeto.

**Diagramas de transição**



Estado inicial: seta a apontar para, sem nada antes (“*Start*” omitível)

Estados finais: duplo círculo

**Tabelas de transição**

Uma imagem com mesa

Descrição gerada automaticamente

Estado inicial: seta

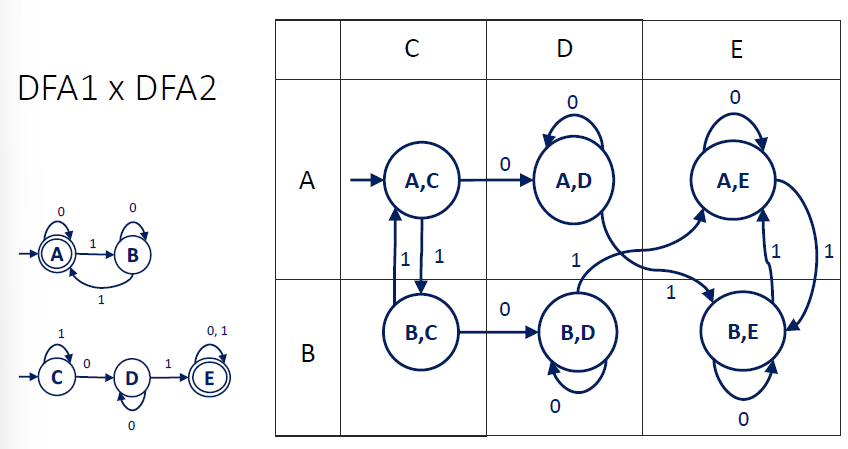
Estados finais: \*

**Função transição estendida**

(q0, ab) = q2

 q0 – a –> q1 – b –> q2

**Operações com DFAs (produto cartesiano)**



**3.NFAs**

Autómatos Finitos Não Deterministas (NFAs)

Não determinista: um determinado símbolo de entrada pode conduzir a mais do que um estado. Chegar a um estado final é o suficiente para um input ser aceite, mesmo que por outro “caminho” não chegue.

Definição: (tuplo) NFA = (Q, Σ, δ, q0, F)  igual a DFA, exceto a função transição δ que retorna um subset {} de Q em vez de um único estado.

**Diagrama e tabela de transição**

Uma imagem com texto, relógio

Descrição gerada automaticamente

Uma imagem com mesa

Descrição gerada automaticamente

Conversão entre FAs (de NFA para DFA)

***Subset construction technique***

1. Partimos do estado inicial da tabela de NFA, mas representando tudo com {}
2. Adicionar linhas com estados = a todos os *subsets* que vão aparecendo na tabela
3. Adicionar linha com vazios (estado morto)
4. Estado inicial = estado inicial tabela NFA

Estados finais = todos os *subsets* que contenham os estados finais da tabela NFA

**Teorema da Linguagem DFA/NFA**

A linguagem L é aceite por um DFA, se e só se L é aceite por um NFA.

**Complemento de uma linguagem**

Transformar em DFA e converter final states em não final states e vice versa.

**4.ε-NFAs**

NFAs com Transições Épsilon

Extensão dos NFAs que permite transições espontâneas entre estados, isto é, sem consumir nenhum símbolo de entrada.

Definição: (tuplo) ε-NFA = (Q, Σ, δ, q0, F) 🡪 diferença: *δ(q, a)* estado *q* pertence a Q e *a* pertence a Σ {ε}.

**ε-close(q)**

Fecho do estado *q*: a que estados consigo chegar a partir de *q* sem consumir nada (próprio + saltos com ε).

Conversão ε-NFA para DFA

***Subset construction tecnique* (*Tabelas* de transição)**

1. Eliminar a coluna épsilon
2. Partir do estado inicial da tabela de ε-NFA
3. Substituir todos os estados pelo seu fecho
4. Adicionar linhas com estados = a todos os *subsets* que vão aparecendo na tabela

(! substituindo sempre os estados pelo fecho)

1. Adicionar linha com vazios (estado morto)
2. Estado inicial = estado inicial tabela ε-NFA

Estados finais = todos os *subsets* que contenham os estados finais da tabela ε-NFA

**5.REs**

Expressões Regulares (REs)

Outra forma de expressar linguagens regulares, sendo uma alternativa e equivalente a NFAs (inclusive ε-NFAs) e DFAs.

Uma expressão regular simples pode ser obtida com um qualquer símbolo do alfabeto ou com o símbolo de cadeia vazia (ε). Adicionalmente, pode ser usado Ø para descrever a linguagem vazia.

**Operações de linguagens**

tendo as linguagens L = {001, 10, 111) e M = {ε, 001}

* União (L M): conjunto de *strings* que pertencem a L, M ou a ambos

L M = {ε, 001, 10, 111}

* Concatenação (LM ou L.M): conjunto de *strings* obtidas pela “conexão” de qualquer *string* em L com qualquer *string* em M

LM = {001, 10, 111, 001001, 10001, 111001}

* Fecho (L\*): conjunto de *strings* obtidas concatenando um número arbitrário de *strings* de L, incluindo repetições

L0 = {ε}

L2 = {001001, 00110, 001111, 10001, 1010, 10111, 111001, 11110, 111111}

L\* = {ε, 001, 10, 111,…}

**Operações de REs**

(ordenando por ordem decrescente de precedência)

1. **\*** 🡪 0 ou mais ocorrências
2. **.** 🡪 concatenação (símbolo omitível)
3. **+** 🡪 união (um ou outro)

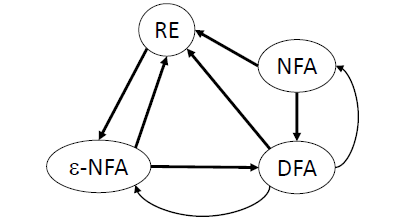
Parêntesis podem ser usados para “forçar” uma certa ordem e + representa uma ou mais ocorrências.

Regras algébricas para REs

|  |  |  |
| --- | --- | --- |
|  | União (+) | Concatenação (.) |
| **Comutatividade** | L + M = M + L | --- |
| **Associatividade** | (L + M) + N = L + (N + M) | (LM)N = L(MN) |
| **Identidade** | Ø + L = L | εL = L |
| **Absorção** | --- | LØ = Ø |
| **Distributiva** | (concatenação sobre união) L (M + N) = LM + LN | |
| **Idempotência** | L + L = L | --- |

Fecho:

* (L\*)\* = L\*
* (ε + 1)\* = 1\*
* Ø\* = ε
* ε\* = ε
* L+ = LL\* = L\*L
* L\* = L+ + ε
* L = ε + L

****Equivalência entre FAs e REs

Podemos converter automaticamente uma RE em ε-NFA.

**Conversão de REs para FAs**

RE 🡪 ε-NFA

* **+** 🡪 duas (ou mais) opções a partir de um estado com junção no final OU separar entre vírgulas com transição direta
* **\*** 🡪 uma seta que volta ao início do envolvente com seta com vazio que avança para o próximo membro OU auto transição
* . 🡪 seta para próximo estado

**Conversão de FAs para REs**

* **Construção de caminhos**

**Rij (k) = Rij (k – 1) + Rik (k – 1) ( Rkk (k – 1) )\* Rkj (k – 1)**

* Cada R é uma RE
* i: nó de entrada
* j: nó de chegada
* k: nós por onde passa, sendo

k = 0: caminhos sem passar por nenhum nó

k = 1: caminhos a passar pelo nó 1 ou nenhum

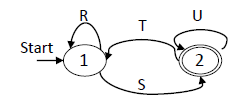
k = 2: caminhos a passar pelo nó 2 e 1, só 2, só 1, ou nenhum (não podendo usar 3, …n como passagem)

Para resolver problemas usando este método, definimos e somamos Rij (k) para todas as combinações do estado inicial (i) com os estados finais (j), sendo k o número de estados. Depois resolvemos com a equação recursivamente: até chegarmos a múltiplos R com k = 0, que conseguimos facilmente resolver, e “voltamos para trás”.

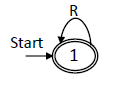
(R110 = ε + auto transição; R120 = só transição direta, caso não exista, Ø)

* **Eliminação de estados**
  1. Eliminar todos os estados, um a um, que não sejam finais nem inicial

(supor que vamos eliminar o estado 2)

* + - Definir q2i, isto é, estados que chegam a 2 com uma transição
    - Definir q2f, isto é, estados a que chega 2 com uma transição
    - Definir as transições para as combinações de q2i 🡪 q2f passando por 2 incluindo o ciclo em 2: **q2i(q2)\*q2f + transição direta de q2i para q2f**
  1. Se houver mais que um estado final, decompor o DFA em tantos DFAs quantos estados finais
  2. Obter RE para cada DFA composto por apenas um estado inicial e um estado final, somando todos para obter o RE final
     + 

RE: (R + S U\* T)\* S U\* = R\* S (U\* + R\* S)\*

* + - a

RE: R\*

**6.RLs**

Linguagens Regulares (RLs)

As linguagens que podem ser expressas por DFAs, NFAs, ε-NFAs ou REs.

Lema da bombagem

* Dada uma linguagem regular infinita L
* Existe um inteiro n (tamanho crítico)
* Para qualquer *string* w L, com tamanho **|w| >= n**
* Podemos escrever **w = xyz**
* Com **|xy| <= n** e **|y| >= 1** (y ≠ ε)
* Tal que**: xykz L**, k = 0, 1, 2, … (k >= 0)

Isto é, para o lema se verificar, conseguimos sempre encontrar uma *string* não vazia y, numa qualquer *string* w = xyz que pertence à linguagem L, tal que y pode ser repetida (*pumped*) um número de vezes arbitrário ou removida, produzindo *strings* que devem pertencer à linguagem.

Para uma linguagem ser regular, então terá que satisfazer o lema. No entanto, uma linguagem pode satisfazer o lema e não ser regular. Isso quer dizer que podemos usar o lema para mostrar que uma linguagem não é regular. Basta para isso que o lema não seja satisfeito. Procedemos do seguinte modo:

1. assumimos que a linguagem é regular (prova por contradição)
2. se a linguagem é regular, então temos uma *string* w = xyz tal que conseguimos repetir ou remover y
3. queremos então mostrar que, para qualquer escolha de x, y e z, se repetirmos ou removermos y (*pumping*), a *string* resultante não pertence à linguagem

Propriedades de fecho

* Operações booleanas: união, interseção, complemento
* Diferença
* Invertido
* Fecho e concatenação
* Homomorfismo e homomorfismo inverso

Se L e M são linguagens regulares, então qualquer operação destas aplicada a L e M gera também uma linguagem / linguagens regulares.

Propriedades de decisão

* Determinar se uma linguagem é vazia:

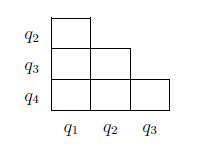
Para a linguagem ser vazia, não poderiam existir no autómato estados finais alcançáveis a partir do estado inicial.

* Determinar se uma cadeia pertence a uma linguagem:

Para verificar quais as cadeias que pertencem à linguagem, basta verificar quais são aceites pelo autómato, simulando o comportamento deste à medida que processa a cadeia de entrada.

* Determinar se duas representações de linguagens correspondem à mesma linguagem:

Minimização de autómatos, verificando se os dois autómatos minimizados são estruturalmente equivalentes (caso o sejam, correspondem à mesma linguagem).

Minimização de autómatos

Temos de detetar quais os estados de um DFA são equivalentes.

1. Desenhar uma tabela triangular
2. Assinalar de imediato os estados não equivalentes: interseção entre estados finais e não finais
3. Preencher as células vazias com as condições necessárias para que sejam equivalentes: equivalência dos estados de destino de cada um dos estados para cada um dos símbolos de linguagem
4. Ir verificando, de forma sistemática, quais os estados com pelo menos uma condição falsa

**7.CFGs**

Gramáticas Livres de Contexto (CFGs)

Uma das formas de representação das Linguagens Livres de Contexto (CFLs). Permitem especificar linguagens (regulares e) não regulares, logo, que REs e autómatos vistos até agora não permitem. Conjunto de regras ou produções no formato H 🡪 B.

Definição: G = (V, T, P, S)

* V: símbolos não terminais, isto é, variáveis
* T: símbolos terminais, isto é, símbolos do alfabeto
* P: produções
* S (pertence a V): variável de arranque

Um CFG é ambíguo quando existe mais que um caminho para a aceitação de uma string (normalmente detetável com a ‘construção’ de strings vazias).

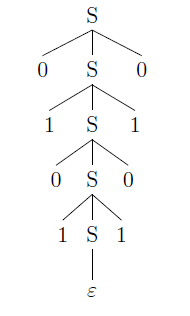
Árvores de análise

Representação de uma derivação de uma *string* a partir de um CFG. Os nós internos representam os símbolos não terminais (variáveis) e as folhas os terminais. Cada expansão de um nó interno nos seus filhos corresponde a um passo da derivação.

Linguagem: L = {vvR : v pertence a {0; 1}\* }

CFG: S 🡪 0S0 | 1S1 | ε

String a produzir: 01011010

derivação: S 🡪 0S0 🡪 01S10 🡪 010S010 🡪 0101S1010 🡪 0101ε1010

árvore:

A colheita da árvore (leitura das folhas da esquerda para a direita) produz a cadeia que a árvore representa.

As derivações podem ser classificadas como:

* ***Leftmost***: derivação mais à esquerda. A cada passo da derivação, é sempre escolhida para substituição a variável mais à esquerda na cadeia intermédia
* ***Rightmost***: derivação mais à direita. A cada passo da derivação, é sempre escolhida para substituição a variável mais à direita na cadeia intermédia

Exemplos de CFGs e *Parsers*

HTML, XML, DTDs

Conversão (ε-)NFA ⬄ CFG

Gramática regular: se for consistentemente linear à esquerda (A 🡪Bx ou A 🡪 x) ou à direita (A 🡪 xB ou A 🡪 x). Linear: todas as produções têm apenas uma variável no seu corpo.

1. Estado inicial ⬄ variável de arranque
2. Estados ⬄ variáveis (não terminais)
3. Símbolos do alfabeto ⬄ símbolos terminais
4. Transição δ ⬄ produção

δ(q1,*a*) = q2 ⬄ Q1 🡪 *a*Q2, podendo *a* ser ε e, nesse caso, omitimos ε na produção, ficando Q1 🡪 Q2

1. Estado final ⬄ nova produção épsilon (q2 é final ⬄ Q2 🡪 ε)

**8.PDAs**

Autómatos de Pilha (PDAs)

Outra forma de representação das Linguagens Livres de Contexto (CFLs). Extensões dos ε-NFAs com uma pilha que permite armazenar informação. Uma transição é descrita da seguinte maneira: **x, z0/x1z1**, tendo, além do símbolo do alfabeto (x) a “consumir”, o símbolo que se encontra no topo da pilha (z0), fazendo *pop* deste, e por qual substituir o topo da pilha (x1z1, x1 fica no topo), fazendo *push* deste.

Definição: P = (Q, Σ, Γ, δ, q0, Z0, F)

* Q: estados
* Σ: alfabeto
* Γ: alfabeto do stack (símbolos terminais e não terminais da gramática)
* δ: função transição. δ(q, a, X) = {(p1, ϒ1), (p2, ϒ2), …}

q, p1, p2 🡪 estados do PDA

a 🡪 símbolo de entrada

X 🡪 símbolo da pilha

ϒ1, ϒ2 🡪 conjuntos de símbolos da pilha, pelos quais X é substituído (pode ser ε

fazendo só *pop* da pilha)

* q0: estado inicial
* Z0: estado inicial do *stack*
* F: estados finais

2 tipos de PDAs

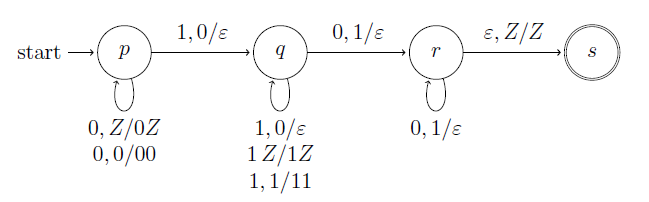
* Aceitação **por estado final**: a aceitação acontece quando o autómato está num estado final e já não existem mais símbolos na cadeia de entrada (independentemente do conteúdo da pilha). Obtidos a partir de um PDA de aceitação por pilha vazia se for adicionado um novo símbolo inicial na pilha antes do processamento, e em cada estado do PDA original for acrescentada uma transição para um estado final fazendo *pop* desse novo símbolo inicial.
* Aceitação **por pilha vazia**. Obtidos a partir de um PDA de aceitação por estado final se, para cada estado final, for realizada uma transição para um novo estado, que faz *pop* de todos os símbolos da pilha, efetivamente esvaziando a pilha.

*Instataneous Description* (ID)

Computação de um PDA: (q, w, ϒ)

* q: estado
* w: *input* reminiscente (cadeia a “consumir”)
* ϒ: conteúdo do stack (esquerda p direita = topo p baixo)

Exemplo: computação da cadeia 011100 (Z0 = Z)

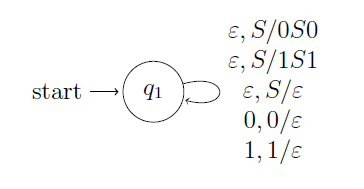
 -

(p, 011100, Z) Ͱ (p, 11100, 0Z) Ͱ (q, 1100, Z) Ͱ (q, 100, 1Z) Ͱ (q, 00, 11Z) Ͱ (r, 0, 1Z) Ͱ (r, ε, Z) Ͱ (s, ε, Z)

CFGs 🡪 PDAs (pilha vazia)

S 🡪 0S0 | 1S1 | ε

1. Um estado, com várias transições para si próprio
2. Inicialmente, a pilha contém apenas o símbolo correspondente à variável de arranque da gramática (S)
3. Variável (não terminal) 🡪 transição, sem consumir símbolo de entrada (ε), que na pilha substitui essa variável (S) pelo corpo de cada uma das suas produções na pilha (0S0 | 1S1 | ε 🡪 ε, S/0S0 ε, S/1S1 ε, S/ε)
4. Símbolo (terminal) 🡪 transição que consome o símbolo (0 e 1) e faz *pop* desse mesmo do topo da pilha (0, 0/ε 1, 1/ε)



Determinismo dos PDAs

Os PDAs vistos até aqui são maioritariamente não determinísticos, isto é, em cada momento, poderia ser realizada mais do que uma transição, dando efetivamente origem não a uma única sequência de descrições instantâneas, mas sim a uma árvore. No entanto, os PDAs podem ser deterministas, o que pode facilitar a sua execução, e análise.

**9.CFLs e propriedades**

Lema da bombagem para CFLs

* Dada um CFL infinito L
* Existe um inteiro n (tamanho crítico)
* Para qualquer *string* z L, com tamanho **|z| >= n**
* Podemos escrever **z = uvwxy**
* Com **|vwx| <= n** e **|vx| >= 1** (vx ≠ ε)
* Tal que**: uviwxiy L**, i = 0, 1, 2, … (k >= 0)

Como para RLs, para o lema se verificar, conseguimos sempre encontrar uma *string* não vazia vwx, numa qualquer *string* z = uvwxy que pertence à linguagem L, tal que v e x podem ser repetidos (*pumped*) um número de vezes arbitrário ou removidos, produzindo *strings* que devem pertencer à linguagem.

Para uma linguagem ser livre de contexto (CFL), então terá que satisfazer o lema. No entanto, uma linguagem pode satisfazer o lema e não ser CFL. Isso quer dizer que podemos usar o lema para mostrar que uma linguagem não é CFL. Basta para isso que o lema não seja satisfeito. Procedemos do seguinte modo:

1. assumimos que a linguagem é CFL (prova por contradição)
2. se a linguagem é regular, então temos uma *string* z = uvwxy tal que conseguimos repetir ou remover v e x
3. queremos então mostrar que, para qualquer escolha de u, v, w, x e y, se repetirmos ou removermos v e x (*pumping*), a *string* resultante não pertence à linguagem

**nota**: dependendo de como decompomos z em uvwxy, podemos concluir que não cumpre o lema, mas não podemos concluir que L não é CFL. Tem de contradizer para todas as decomposições.

Forma normal de Chomsky (CNF)

Esta forma não permite obter a cadeia vazia (ε), pelo que a gramática de uma linguagem que inclua a cadeia vazia irá ter uma correspondente em CNF para L\{ε}.

1. Eliminar produções-ε:
   1. Identificar quais as variáveis anuláveis, isto é, aquelas que podem produzir a cadeia vazia. (A 🡪 ε, B 🡪 ε, C 🡪 AB; A, B e C são anuláveis)
   2. Identificar todas as produções em que as variáveis anuláveis aparecem
   3. Criar novas produções em que se fornece a alternativa em que esta variável não esteja presente. (A anulável e B 🡪 AC transformada em B 🡪 AC | C)
2. Eliminar produções unitárias (formato A 🡪 B):
   1. Identificar os pares unitários. Par (X, X) é unitário; par (X, Y) é unitário se houver uma sequência de produções unitárias que permita gerar Y a partir de X. (A 🡪 B e B 🡪 C; (A, C) é unitário).
   2. Para cada par unitário (A, B), incluir as produções não unitárias de B em A.
3. Eliminar símbolos inúteis:
   1. Identificar e eliminar símbolos não geradores (S 🡪 AB | a, A 🡪 b, B não é gerador) e símbolos não atingíveis (S 🡪 a, A 🡪 b, A não é atingível)

Propriedades de CFLs

* Substituição: se L é CFL e s() uma substituição, que associa a cada símbolo uma CFL, então s(L) é uma CFL
* União
* Concatenação
* Fecho
* Homomorfismo e homomorfismo inverso
* Reverter
* Interseção com uma RL

(uma interseção com outra CFL pode não resultar numa CFL)

CYK *parser algorithm* <https://www.youtube.com/watch?v=VTH1k-xiswM&t=438s>

Testar se uma string pertence a uma CFL. Este método necessita que a gramática da linguagem esteja na CNF.

1. Tabela triangular que tem como base a cadeia. Cada célula Xij vai conter as variáveis que permitem gerar a subcadeia i-j.
2. Preencher a fila de baixo (os elementos Xii): Xi vai ter o subset de variáveis não terminais que podem produzir ai

Uma imagem com mesa

Descrição gerada automaticamenteUma imagem com texto, shoji, palavras cruzadas

Descrição gerada automaticamente

1. Preencher fila a fila

Os elementos da 2ª fila (formato Xi i+1) vão ter o subset de variáveis não terminais que produzem a substrings de 2 letras (formato aiai+1). Por exemplo, para o elemento X12, ver substring *ba*, X12 = variáveis não terminais que produzem o produto cartesiano de {B} com {A, C}.

Uma imagem com mesa

Descrição gerada automaticamente

Os elementos da 3ª fila (formato Xi i+2) vão ter o subset de variáveis não terminais que produzem as combinações de substrings de 3 letras (formatos ai, ai+1ai+2 e aiai+1, ai+2). Por exemplo, para o elemento X13, substrings *b, aa* e *ba, a*. *b* é produzido por B (X11) e *aa* é produzido por B (X23) , produto cartesiano BB não está CFL. *ba* é produzido por SA (X12) e *a* é produzido por AC (X22). Produto cartesiano de {S,A} com {A,C} não está no CFL, pelo que X13 fica vazio.

Uma imagem com mesa

Descrição gerada automaticamente

Assim em diante para as outras filas.

1. Se a variável de arranque estiver presente no topo da tabela (X15 tem S), então a *string* pertence à CFL

**10.TM**

Turing Machine (TM)

Autómatos capazes de reconhecer qualquer tipo de linguagem. Transições do tipo a/B->.

São como uma máquina com uma cabeça de processamento (head) que a cada momento se encontra numa posição de uma fita unidimensional infinita e que se pode deslocar para a esquerda ou para a direita (a/B**->**), mantendo ou alterando o símbolo presente na fita (**a/B**->).

Definição: TM = (Q, Σ, Γ, δ, q0, B, F)

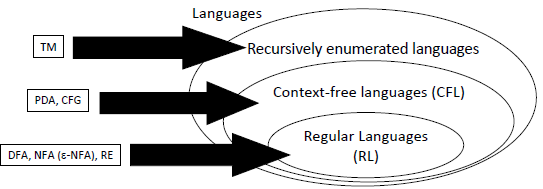
* Q: estados
* Σ: alfabeto
* Γ: símbolos da fita
* δ: função transição. δ(q,X) = (p, Y, D)

q, p pertencem a Q, X, Y que por sua vez pertencem a Γ

D indica a direção do movimento

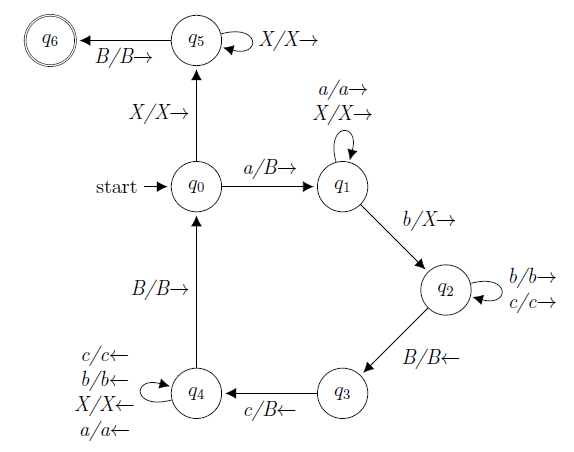
* q0: estado inicial
* B: símbolo (pertencente a Γ) que representa inexistência (Branco)
* F: estados finais

Linguagem de uma TM



Computação

Linguagem anbncn, n >= 1. Computação da cadeia *abc*.

B q0 *abc* B (**temos sempre B antes e depois**) Ͱ

Ͱ BB q1 *bc* B Ͱ B X q2 *c* B Ͱ

Ͱ B X q3 *c* B Ͱ B q4 X BB Ͱ

Ͱ B q4 B X B Ͱ B q0 X B Ͱ

Ͱ B X q5 B Ͱ B X B q6 B