Linguagens Formais e Autômatos

Professores: Dr. Hermes Senger

1. Introdução
2. Gramáticas
3. Autômatos finitos
4. Linguagens livre de contexto
5. Maquinas de Turing
6. Decidibilidade
7. Complexidade computacional
8. Problemas NP completos
9. Introdução às linguagens formais

Bibliografia básica :

1. LEWIS, H.R.; PAPADIMITRIOU, C. Elements of the Theory of Computation. 2nd edition. Prentice Hall. 1997. ISBN: 01-326-2478-8.

2. SIPSER, M. Introduction to the Theory of Computation. Brooks Cole. 1996. ISBN: 05-349-4728-X.

2. HOPCROFT, J.; MOTWANI, R.; ULLMAN, J. Introduction to Automata Theory, Languages, and Computation. 2nd edition. Addison-Wesley Publishing. 2000. ISBN: 02-014-4124-1

Avaliação

1. Prova escrita (1/2);
2. Trabalho prático (1/2).

Organização da disciplina

Conceitos utilizados :

Conceito	Nota
A	8,5 - 10
B	7 – 8,5
C	5 - 7
D	< 5

Organização da disciplina

1. Introdução
2. Gramáticas
3. Autômatos finitos
4. Linguagens livre de contexto
5. Maquinas de Turing
6. Decidibilidade
7. Complexidade computacional
8. Problemas NP completos
9. Introdução às linguagens formais

1. Introdução

Computabilidade e Complexidade

- O que é um computador ?
- Quais são as suas capacidades e limitações ?
- Que classes de problemas são computáveis ?
- Como determinar a complexidade da solução computacional de um problema?

1. Introdução

Histórico

- 1930 - Os matemáticos começam a investigar essas questões.

  => A Teoria Computacional (ou Teoria dos Autômatos) se dedica ao estudo dos dispositivos computacionais abstratos, ou "máquinas".

  - Alan Turing propôs uma máquina abstrata (Máquina de Turing) que tinha o mesmo poder computacional que as máquinas atuais => Determinar o que uma máquina poderia ou não fazer

- 1940's-1950's - Autômatos Finitos – Alvo de estudo de muitos pesquisadores para modelagem do funcionamento do cérebro.

- Final da década de 50 – Gramáticas Formais – Propostas por N.Chomsky, e utilizadas atualmente como base para o desenvolvimento de compiladores, etc.

Histórico (cont')

- 1969 – S.Cook estudou o que poderia ou não ser computado.

  => Problemas que podem ser eficientemente resolvidos

  => Problemas que podem ser resolvidos a priori, mas [illegible]ática levam muito tempo para serem resolvidos, onde comp[illegible]eriam úteis apenas para a solução de uma pequena porç[illegible]lema.

Intratáveis (NP completos)

Os modelos matemáticos (ex: Funções recursivas – Gödel e Kleene, λ-Calculus – Church, Máquinas de Turing – A.Turing, etc.) propostos são ainda aplicados atualmente:

=> Por exemplo, a Máquina de Turing pode ser útil ao depararmos com problemas intratáveis e, se possível, propor soluções para contorná-los (aproximação, heurísticas, timeout para chegar a uma solução, etc.)

O que é Teoria da Computação ?

A Teoria da Computação pode tem ser vista como um guia (um roteiro) que nos orienta no sentido de informar o que pode é o que não pode ser efetivamente computável, explicando porque, de que forma e com que complexidade.

A Teoria da Computação classifica os problemas computacionais em três classes:

1. Problemas Indecidíveis (ou impossíveis de serem solucionados);
2. Problemas Intratáveis (possíveis com recursos ilimitados, porém impossíveis com recursos limitados);
3. Problemas Tratáveis (possíveis de serem solucionadas com recursos limitados).

O que é Teoria da Computação ? (cont')

Essa classificação engloba problemas de toda a natureza (desde problema clássicos que fundamentam a teoria da computação até problemas práticos da ciência da computação, tais como:

1 – Existe programa para solucionar um determinado problema?

2 – Qual o poder de expressão de um determinado modelo de especificação?

3 – Dado um programa qualquer, ele sempre tem parada garantida?

4 – Dois programas P1 e P2 são equivalentes entre si?

5 – Uma determinada solução é a melhor solução para um dado problema? **Complexidade**

6 – Qual o significado de um determinado programa? **Semântica**

7 – Dado um programa qualquer, este programa está correto? **Correção/ Construção**

O que é Teoria da Computação ? (cont')

A teoria da computação pode ser vista como:

- Um conjunto de modelos formais (juntamente com suas propriedades) que fundamentam a ciência da computação => Tais modelos incluem Autômatos (Finitos, de Pilha e Máquinas de Turing) e Gramáticas.
- As propriedades de interesse desses modelos que envolvem questões de decidibilidade, Inter-relacionamento entre modelos (abrangência, equivalência, etc...) e complexidade computacional.

Teoria das Linguagens Formais e Autômatos

Conceitos Fundamentais da Teoria da Computação

- **Procedure** => É um **conjunto finito de passos** (instruções), os quais podem ser **executados mecanicamente** e forma discreta (ex: programa de computador) => Solução em **tempo de execução**

- **Algorítmo** => É uma **procedure que, independentemente de suas entradas, tem parada garantida**.

Conceitos Fundamentais da Teoria da Computação

Exemplos:

Repr. Algorítmica

1 – Dado um número inteiro positivo I, determinar se I é ou não um número primo.

2 – Dado um programa escrito em uma determinada linguagem de programação, determinar se esse programa está sintaticamente correto. **Repr. Algorítmica**

3 – Dado um programa qualquer, determinar se existe alguma entrada para a qual o programa entrará em loop. **Repr. Procedures**

Conceitos Fundamentais da Teoria da Computação

Conjuntos Recursivos e Conjuntos Recursivamente Enumeráveis:

- Um conjunto é dito **Recursivamente Enumerável** se ele pode ser representado (solucionado) por uma procedure,

- Um conjunto é dito **Recursivo** se ele pode ser representado (solucionado) por um algoritmo.

Como **procedures e algoritmos podem ser definidos formalmente através de vários modelos** (gramáticas e autômatos, por exemplo), podemos também definir conjuntos recursivos e recursivamente enumeráveis em função de tais modelos.

Conceitos Fundamentais da Teoria da Computação

Problemas Decidíveis e Indecidíveis X Algoritmos e Procedures

- Um problema é **decidível** (tratável ou não) => **resolvível por um algoritmo**

- **Caso contrário** => ele é um problema **indecidível**.

Podemos concluir que:

- Se um problema é **decidível** => Possui um conjunto de **soluções recursivas**

- Se um problema é **indecidível** => Possui um conjunto de **soluções recursivamente enumeráveis**

A questão da decidibilidade pode ser tratada formalmente através dos modelos que compõem a Teoria das Linguagens Formais e Autômatos !!

Propósitos Fundamentais da Teoria da Computação

- Definição informal e intuitiva de procedures e algoritmos

- Definição mais formal pode ser realizada através de propósitos (ou princípios) da Teoria da Computação.

=> Tais propósitos ou formalismos servem como modelos na solução de diversos problemas práticos. Podemos citar:

- Máquinas de Turing (Turing, 1936);
- Gramáticas (Chomsky, 1959);
- Algoritmos de Markov (Markov, 1951);
- Lambda Calculus (Church, 1941);
- Sistemas Post e Sistemas de Produção (Emil Post, 1936);

Toda procedure (ou algoritmo) descrita por algum destes formalismos, pode também ser descrita através de qualquer um dos demais => equivalência entre os formalismos

Propósitos Fundamentais da Teoria da Computação

Equivalência entre os formalismos:

- Tese de Church => todo processo computável – passível de ser descrito por uma procedure – pode ser realizado por uma Máquina de Turing.

Máquinas de Turing constituem o formalismo mais genérico para a representação de procedure e que qualquer outro formalismo será significativo se for considerado equivalente às máquinas de Turing.

A demonstração formal da equivalência entre os diversos formalismos propostos e máquinas de Turing, reforça a tese de Church.

O que é a Teoria da Linguagem Formal ?

Para respondermos esta questão precisamos primeiro responder o que é Linguagem Formal, e para isto precisamos antes responder o que é Linguagem.

✓ De maneira bastante informal, podemos definir uma linguagem como sendo uma forma de comunicação.

✓ De forma mais elaborada, definimos uma linguagem como sendo "um conjunto de elementos (símbolos) e um conjunto de métodos (regras) para combinar estes elementos, usado e entendido por uma determinada comunidade".

Exemplos:
1 - Linguagens Naturais (ou idiomáticas)
2 - Linguagens de Programação, de Controle, de Consulta
3 – Protoco

Apesar de intuitiva, esta definição não nos permite responder satisfatoriamente as duas primeiras questões;

Precisamos dar um sentido formal para a definição de linguagem !!

Conceitos Básicos

Alfabeto (ou vocabulário): É um conjunto finito, não vazio, de símbolos (elementos). Representaremos um alfabeto por V.

Exemplos: V = {a, b, c,...,z }
V = {0, 1}
V = {a, e, i, o ,u}

Sentenças (String): Uma sentença sobre um alfabeto *V*, é uma seqüência (ou cadeia) finita de símbolos do alfabeto.

Exemplo de sentenças sobre V = { a , b }: a, b, aa, ab, bb, aaa, aab, aba, baa, ...

Conceitos Básicos (cont')

Tamanho de uma sentença: Seja *w* uma sentença.
O tamanho da sentença *w*, denotado por |*w*|, é definido pelo número de símbolos (elementos do alfabeto) que compõem w.

Exemplos: Seja V = { a , b , c }
se x = aba, então |x| = 3
se x = c, então |x| = 1

Sentença vazia: É uma sentença constituída de nenhum símbolo; isto é, uma sentença de tamanho 0 (zero).

Observações: - Representaremos a sentença vazia por ε (épslon).
- Por definição, $|\varepsilon| = 0$

Conceitos Básicos (cont')

Potência de uma sentença: Seja w uma sentença. A n-ésima potência de w, representada por w^n, significa w repetido n vezes.

Exemplos: se x = ab, então x^3 = ababab
Para $\forall$ x, $x^0 = \varepsilon$

Conceitos Básicos (cont')

Fechamento de um Alfabeto: Seja V um alfabeto.

- O fechamento reflexivo (ou simplesmente fechamento) de V, representado por V^*, é dado pelo conjunto de todas as possíveis seqüências que podem ser formadas a partir de V, inclusive a sentença vazia.

- O fechamento transitivo (ou fechamento positivo) de V, representado por V^+, é dado por $V^* - \{ \varepsilon \}$.

Exemplos: Seja V = { 0, 1 }, temos que:

$V^* = \{\varepsilon, 0, 1, 00, 01, 11, 000,...\}$

$V^+ = \{0, 1, 00 ,01, 11, 000,...\}$

O que é uma Linguagem ?

Linguagem: Uma linguagem L sobre um alfabeto V, é um subconjunto de V*; isto é,

$$L \subseteq V^*$$

Linguagens e suas representações

O estudo de linguagens está intimamente relacionado ao estudo das formas de representação dessas linguagens:

- **Linguagem Finita**: É uma Linguagem que pode ser representada por enumeração.

  Exemplo: A linguagem definida como sendo o conjunto dos inteiros positivos pares maiores que 0 e menores que 20, pode ser representado por: L = {2, 4, 6, 8, 10, 12, 14, 16, 18}.

- **Linguagem Infinita**: impossível de enumerar => representação finita

  Exemplo: A linguagem definida como sendo o conjunto dos inteiros pares poderia ser representada por V ={2, 4, 6, 8, 10,...} que, que apesar de intuitiva, não é finita e nem precisa.

Linguagens e suas representações (cont')

As representações finitas de linguagens classificam-se em:

- **Reconhecedores** – São dispositivos formais que nos permitem verificar se uma determinada sentença pertence ou não a uma determinada linguagem.

  Esses dispositivos denominam-se *autômatos*. Por exemplo, autômatos finitos, autômatos de pilha e máquinas de turing.

- **Sistemas Geradores** – São dispositivos formais dotados de mecanismos que permitem a geração sistemática das sentenças de uma linguagem.

  Os principais sistemas geradores disponíveis são as *gramáticas*, dentre as quais, por exemplo, pode-se destacar as gramáticas de CHOMSKY.

Observações: Todo *reconhecedor* e todo *sistema gerador* pode ser representado por *algorítmos* e/ou *procedures.*

Representação Finita de Linguagens

Considerando $L = \{ w \in \{0,1\}^* : $ w possui duas ou três ocorrências de 1, sendo que a primeira e a segunda não são consecutivas}

$\Rightarrow$ Representação da linguagem através dos símbolos $\cup$, $\circ$ e $*$

$L = \{0\}^* \circ \{1\} \circ \{0\}^* \circ \{0\} \circ \{1\} \circ \{0\}^* ((\{1\} \circ \{0\}^*) \cup \varnothing^*)$

<=> $L = 0^*\, 10^*\, 010^*\, (10^* \cup \varnothing^*)$ Expressões Regulares

=> 01001010, 1011, 010101, ...

Algumas regras ...

As expressões regulares (ER) sobre um alfabeto Σ são sentenças (strings) sobre um o alfabeto $\Sigma \cup \{$), (, $\varnothing$, $\cup$, $*\}$. Considera-se:

1. $\varnothing$ e qualquer outro membro de Σ é uma ER.
2. Se α e β são ER's, então $(\alpha\beta)$ é uma ER.
3. Se α e β são ER's, então $(\alpha \cup \beta)$ é uma ER.
4. Se α é uma ER, então α^* é uma ER.
5. Qualquer sentença é uma ER se obedece (1) a (4).

Algumas regras ... (cont')

Toda expressão regular (ER) representa uma linguagem (L)

=> Uma linguagem pode ser definida como:

1. $L(\varnothing) = \varnothing$ e $L(a) = \{a\}$ para cada $a \in \Sigma$.
2. Se α e β são ER's, então $L((\alpha\beta)) = L(\alpha)\,L(\beta)$.
3. Se α e β são ER's, então $L((\alpha \cup \beta)) = L(\alpha) \cup L(\beta)$.
4. Se α é uma ER, então $L(\alpha^*) = L(\alpha)^*$.

Aplicando as regras ...

Exemplo 1: O que representa a linguagem $L(((a \cup b)^*a))$?

$L(((a \cup b)^*a))$ => $L((a \cup b)^*)\,L(a)$ (regra 2)

2. Se α e β são ER's, então $L((\alpha\beta)) = L(\alpha)\,L(\beta)$

=> 1. $L(\varnothing) = \varnothing$ e $L(a) = \{a\}$ para cada $a \in \Sigma$

=> 4. Se α é uma ER, então $L(\alpha^*) = L(\alpha)^*$

3. Se α e β são ER's, então $L((\alpha \cup \beta)) = L(\alpha) \cup L(\beta)$

=> 1. $L(\varnothing) = \varnothing$ e $L(a) = \{a\}$ para cada $a \in \Sigma$

=> $\{ w \in \{a,b\}^* : w$ termina com a$\}$

Exercícios

1) Que linguagem é representada por $(c^*(a \cup (bc^*))^*)$?

=> Expressão regular que representa todas as sentenças sobre {a, b, c} que não possuem a sentença ac.

2) E a linguagem $(0^* \cup (((0^*(1 \cup (11)))((00^*)(1 \cup (11)))^*)0^*))$?

=> Expressão regular que representa todas as sentenças sobre {0, 1} que não possuem a sentença 111.

Mais definições ...

- **Linguagens Formais**: São linguagens que podem ser representadas de maneira <u>finita</u> e <u>precisa</u> através de sistemas com sustentação matemática (dispositivos formais ou modelos matemáticos).
- **Linguagem Recursiva**: Uma linguagem é <u>recursiva</u> se existe um <u>algoritmo</u> capaz de reconhecer ou gerar as sentenças que compõem essa linguagem.
- **Linguagem Recursivamente Enumerável**: É toda a linguagem cujas sentenças podem ser reconhecidas ou geradas por <u>procedures</u>.

Nosso objetivo

Teoria das Linguagens Formais e dos Autômatos

Estudar modelos matemáticos que possibilitam a especificação e o reconhecimento de linguagens (no sentido amplo da palavra), suas classificações, estruturas, propriedades, características e inter-relacionamentos.

=> A importância desta Teoria na Ciência da Computação é dupla:

- Apóia outros aspectos teóricos da Ciência da Computação (ex: decidibilidade, computabilidade, complexidade computacional);
- Fundamenta diversas aplicações computacionais tais como, processamento de linguagens, reconhecimento de padrões, modelagem de sistemas.

Motivação

- Uma linguagem (L) é qualquer conjunto ou subconjunto de sentenças sobre um alfabeto (V*) => $L \subseteq V^*$.

  Como podemos determinar esse subconjunto ??

- A finalidade de uma gramática é definir o subconjunto de V* que forma (define) uma determinada linguagem.

O que é uma gramática ??

Uma gramática define uma estrutura sobre um alfabeto de forma a permitir que apenas determinadas combinações sejam válidas.

=> Realiza a validação de sentenças de uma linguagem.

Uma gramática, de maneira informal, pode ser definida como sendo:

- Um sistema gerador de linguagens;
- Um sistema de reescrita;
- Uma maneira finita de descrever (representar) uma linguagem;
- Um dispositivo formal usado para especificar de maneira finita e precisa uma linguagem potencialmente infinita.

Intuição: um subconjunto da gramática da língua portuguesa

<sentença> :: = <sujeito> <predicado>
<sujeito> :: = <substantivo>
| <artigo> <substantivo>
| <artigo> <adjetivo> <substantivo>
<predicado> :: = <verbo> <objeto>
<substantivo> :: = joão | Maria | cachorro | livro | pão
<artigo> :: = o | a
<adjetivo> :: = pequeno | bom | bela
<verbo> :: = morde | le | olha
<objeto> :: = <substantivo>
| <artigo> <substantivo>
| <artigo> <adjetivo> <substantivo>

Notação utilizada:
< .... > : categoria sintática ou gramatical;
::= : definido por
| : ou (alternativa)
α ::= β : regra de sintaxe (ou regra gramatical ou regra de produção)

Definição Formal de Gramática

Uma gramática G pode ser descrita como uma quádrupla*

G = (Vn, Vt, P, S) onde:

- Vn – É um conjunto finito de símbolos denominados não-terminais. Estes símbolos também são denominados meta variáveis.
- Vt – É um conjunto finito de símbolos denominados terminais. São os símbolos da linguagem que podem ser usados na formação das sentenças da mesma.
- P – É um conjunto finito de pares (α , β) denominado produções (ou regras gramaticais ou regras de sintaxe) => Representada por α :: = β
- S – É o símbolo inicial da gramática => Deve pertencer a Vn, a partir do qual as sentenças da gramática são geradas.

* Sistema formal constituído de quatro elementos

Exemplo

Formalizando o subconjunto da gramática da língua portuguesa :

Gportugues = (Vn, Vt, P, S), onde:

Vn = { <sentença> , <sujeito> , <predicado> , <substantivo> , <artigo> , <adjetivo> , <predicado> , <verbo> , <objeto> }

Vt = { joão, maria , cachorro, livro , pão, o , a , pequeno, bom , bela , morde, le , olha }

P = é o conjunto das regras gramaticais apresentado

S = <sentença>

Fazendo uma analogia entre o exemplo intuitivo e a noção formal de gramática, constatamos que:

- Vn – são as categorias sintáticas ou gramaticais;
- Vt – são as palavras utilizadas como símbolos da linguagem;
- P – são as regras sintáticas (ou gramaticais);
- S - é a categoria gramatical que sintetiza o que será produzi

Notação a ser utilizada:

::= - →

Vn – Letras de “A” a “T” e palavras escritas com letras maiúsculas

Vt – Letras de “a” a “t”, palavras escritas com letras minúsculas, dígitos e caracteres especiais

Vt* - u, v, x, y, w, z

$\{Vn \cup Vt\}$ – U, V, X, Y, W, Z

$\{Vn \cup Vt\}^+$ - $\alpha, \beta, \gamma, \delta, \ldots, \omega$ (exceto ε) => Strings

Derivação e Redução

São operações de substituição que formalizam a utilização de gramáticas, sendo que:

- **Derivação**: É a operação que consiste em substituir um string (ou parte dele) por outro, de acordo com as produções das gramáticas em questão, no sentido símbolo inicial $\rightarrow$ sentença;

  => Operação adequada para geração de sentenças

- **Redução**: É a operação que consiste na substituição de um string (ou parte dele) por outro, de acordo com as produções da gramática, no sentido sentença $\rightarrow$ símbolo inicial.

  => Operação adequada ao reconhecimento de sentenças

Noção Formal de Derivação e Redução

Seja G = (Vn, Vt, P, S) uma gramática.

Seja $\delta\alpha\gamma \in (Vn \cup Vt)^*$.

- Derivação / redução em um passo (ou direta)
- Derivação / redução em zero ou mais passos
- Derivação / redução em um ou mais passos

Noção Formal de Derivação e Redução

Seja G = (Vn, Vt, P, S) uma gramática.

Seja $\delta\alpha\gamma \in (Vn \cup Vt)^*$.

- Derivação / redução em um passo (ou direta): dizemos que $\delta\alpha\gamma$ deriva em um passo (ou deriva diretamente) $\delta\beta\gamma$, se e somente se $\alpha \rightarrow \beta \in P$; indicamos por $\delta\alpha\gamma \Rightarrow \delta\beta\gamma$.

  => Neste caso, dizemos ainda que $\delta\beta\gamma$ reduz-se a $\delta\alpha\gamma$ em um passo (ou diretamente); denotamos por $\delta\alpha\gamma \Leftarrow \delta\beta\gamma$

  Exemplo:

  <sujeito> :: = <substantivo> | ...
  <substantivo> :: = joão | Maria | cachorro | livro | pão

  <sujeito> -> joão => Derivação/Redução direta !!

Noção Formal de Derivação e Redução (cont')

Seja G = (Vn, Vt, P, S) uma gramática.

Seja $\delta\alpha\gamma \in (Vn \cup Vt)^*$.

- Derivação / redução em zero ou mais passos: Por extensão, dizemos que α deriva em zero ou mais passos (ou simplesmente deriva) β, se existirem seqüências $\alpha 1, \alpha 2, ..., \alpha n$ tais que: $\alpha \Rightarrow \alpha 1 \Rightarrow \alpha 2 \Rightarrow ... \Rightarrow \alpha n \Rightarrow \beta$;

  esta forma de derivação é denotada por $\alpha \overset{*}{\Rightarrow} \beta$

  => Analogamente β reduz-se a α em zero ou mais passos (ou simplesmente reduz-se); indicamos por $\alpha \overset{*}{\Leftarrow} \beta$

  Observação: - Se $\alpha \overset{*}{\Rightarrow} \beta$ em 0 (zero) passos, então $\alpha = \beta$.

  Exemplo:

  <predicado> :: = Andar | <objeto>
  <objeto> :: = <substantivo> | <artigo> <substantivo> | ...

  <predicado> -> Andar => 0 passos !!
  <predicado> -> <objeto> => Derivação/Redução (1 ou mais passos!!)

Noção Formal de Derivação e Redução (cont')

Seja G = (Vn, Vt, P, S) uma gramática.

Seja $\delta\alpha\gamma \in (Vn \cup Vt)^*$.

- **Derivação / redução em um ou mais passos:** quando houver certeza de que pelo menos um passo foi necessário para chegar em β a partir de α (ou vice-versa), então teremos uma derivação (redução) em um ou mais passos;

  => indicaremos por: $\alpha \overset{+}{\Rightarrow} \beta$ (ou por $\alpha \overset{+}{\Leftarrow} \beta$).

  Exemplo:

  <sentença> :: = <sujeito> <predicado>
  <sujeito> :: = <substantivo> | ...
  <substantivo> :: = joão | Maria | cachorro | livro | pão

  <sentença> -> joão ... => Redução/Derivação (1 ou mais passos) !!

Sentença, Forma Sentencial e Linguagem

Sentença

É uma seqüência só de terminais produzida (gerada) a partir do símbolo inicial de uma gramática; isto é, se G = (Vn, Vt, P, S) $\wedge$ S $\overset{+}{\Rightarrow}$ x, então x é uma sentença pertencente à linguagem representada por G.

Sentença, Forma Sentencial e Linguagem (cont')

Forma Sentencial

É uma seqüência qualquer (composta por terminais e não-terminais) produzida (gerada) a partir do símbolo inicial de uma gramática; isto é, se G = (Vn, Vt, P, S) $\wedge$ $S \Rightarrow \alpha \Rightarrow \beta \Rightarrow \ldots \Rightarrow \gamma \Rightarrow \ldots$ então $\alpha, \beta, \ldots, \gamma, \ldots$ são formas sentenciais de G.

Sentença, Forma Sentencial e Linguagem (cont')

Linguagem

Formalmente definimos a linguagem gerada por G = (Vn, Vt, P, S), denotada por L(G), como sendo: $L(G) = \{x \mid x \in Vt^* \wedge S \overset{+}{\Rightarrow} x\}$; ou seja, uma linguagem é definida pelo conjunto de sentenças que podem ser derivadas a partir do símbolo inicial da gramática que a representa.

Sentença, Forma Sentencial e Linguagem (cont')

Gramáticas Equivalentes

Duas gramáticas G1 e G2 são equivalentes entre si, se e somente se L(G1) = L(G2).

Formalmente: G1 ≡ G2 ⇔ L(G1) = L(G2).

Derivação utilizando uma gramática

Seja G = (Vn, Vt, P, S) uma gramática, onde :

Vn = {E, I}, E representam as expressões, e I os identificadores

Vt = {a,b,0,1}

S = {E}

P = {P1, P2, P3, P4, P5, P6, P7, P8, P9, P10}

Produções P:

P1.	E -> I
P2.	E -> E + E
P3.	E -> E * E
P4.	E -> (E)
P5.	I -> a
P6.	I -> b
P7.	I -> Ia
P8.	I -> Ib
P9.	I -> I0
P10.	I -> I1

Derivação utilizando uma gramática (cont')

Inferência de strings utilizando a gramática G:

#	String	P/ Vn	ref. Produção	ref.String
i	a	I	I -> a	--
ii	b	I	I -> b	--
iii	b0	I	I -> I0	ii
iv	b00	I	I -> I0	iii
v	a	E	E -> I	i
vi	b00	E	E -> I	iv
vii	a + b00	E	E -> E + E	v, iv
viii	(a + b00)	E	E -> (E)	vii
ix	a * (a + b00)	E	E -> E * E	v, viii

Derivação - Exercício

Seja G = (Vn, Vt, P, S) uma gramática, onde :

Vn = {A, B}

Vt = {0,1}

S = {A1B}

P = {P1, P2}

Produções P:

P1. A -> 0A | ε

P2. B -> 0B | 1B | ε

Quais são as derivações para as strings:

a) 00101

b) 1001

c) 00011

Tipos de Gramáticas (Hierarquia de Chomsky)

Gramática Tipo 0: (ou gramática sem restrições)
G = (Vn, Vt, P, S), onde:
P = {$\alpha \rightarrow \beta$ | $\alpha \in V^*VnV^* \wedge \beta \in V^*$}

Gramática Tipo 1: (ou Gramática Sensível ao Contexto – G.S.C.)
G = (Vn, Vt, P, S), onde:
P = {$\alpha \rightarrow \beta$ | $|\alpha| \leq |\beta|$, $\alpha \in V^*VnV^* \wedge \beta \in V+$}

Gramática Tipo 2: (ou Gramática Livre de Contexto – G.L.C.)
G = (Vn, Vt, P, S), onde:
P = {$A \rightarrow \beta$ | $A \in Vn \wedge \beta \in V^+$}

Gramática Tipo 3: (ou Gramática Regular – G.R.)
G = (Vn, Vt, P, S), onde:
P = {$A \rightarrow a X$ | $A \in Vn$, $a \in Vt \wedge X \in \{Vn \cup \{\varepsilon\}\}$}

Tipos de Gramáticas (Hierarquia de Chomsky)

Tipo 0 - **Gramáticas Irrestritas (GI)**

São definidas pelas seguintes regras de produção:

$$P = \{ \alpha \rightarrow \beta \mid \alpha \in V^+ , \beta \in V^* \}$$

Ou seja, do lado esquerdo da produção pode haver uma sequência de quaisquer símbolos, desde que, entre eles, haja um não-terminal. Do lado direito da produção pode haver qualquer sequência de símbolos, inclusive a sentença vazia.

Tipos de Gramáticas (Hierarquia de Chomsky)

Tipo 1 - **Gramáticas Sensíveis ao Contexto (GSC)**

Para toda produção

$$\alpha \rightarrow \beta \in P$$
$$|\alpha| <= |\beta|$$

Ou seja, o comprimento da sentença do lado esquerdo deve ser menor ou igual ao comprimento da sentença do lado direito da produção. Do lado direito não é aceito a sentença vazia.
Ex: $\alpha_1 A \alpha_2 \rightarrow \alpha_1 B \alpha_2$

Tipos de Gramáticas (Hierarquia de Chomsky)

Tipo 2 - **Gramáticas Livres de Contexto (GLC)**

Quando as regras de produção são todas na seguinte forma:

$$P = \{ \alpha \rightarrow \beta \mid \alpha \in N \text{ e } \beta \neq \varepsilon \}$$

Ou seja, do lado esquerdo da produção deve, sempre, ocorrer um e apenas um não-terminal. A sentença vazia também não é aceita do lado direito da produção.
Ex: $X \rightarrow abcX$ {não interessa o contexto em que X se encontra}

2. Gramáticas

Tipos de Gramáticas (Hierarquia de Chomsky)

Tipo 3 - **Gramáticas Regulares (GR)**

Toda produção é da forma:

$A \rightarrow aB$ ou

$A \rightarrow a$

Ou seja:

$P = \{ A \rightarrow aX \mid A \in N, a \in T, X \in N \cup \{\varepsilon\} \}$

Ou seja, do lado esquerdo da produção deve, sempre, ocorrer um e apenas um não-terminal e do lado direito podem ocorrer ou somente um terminal, ou um terminal seguido de um não-terminal.

2. Gramáticas

Tipos de Gramáticas (Hierarquia de Chomsky)

Observação:

Organização da disciplina

1. **Introdução**
2. **Gramáticas**
3. **Autômatos finitos**
4. **Linguagens livre de contexto**
5. **Maquinas de Turing**
6. **Decidibilidade**
7. **Complexidade computacional**
8. **Problemas NP completos**
9. **Introdução às linguagens formais**

3. Autômatos Finitos

Geradores X Reconhecedores

Gramáticas sem Restr.	→	**Máquinas de Turing**
Gramáticas S.C.	→	**Autômatos Limitados Lineares**
Gramáticas L.C.	→	**Autômatos de Pilha**
Gramáticas Regulares	→	**Autômatos Finitos**

Autômatos Finitos: são reconhecedores de linguagens regulares;

=> Entende-se por reconhecedor de uma linguagem ”L”, um dispositivo que tomando uma seqüência w como entrada, respondem ”SIM” se w $\in$ L e “NÃO” em caso contrario.

Tipos de Autômatos Finitos:

Autômato Finito Determinístico (A.F.D.)

Autômato Finito Não Determinístico (A.F.N.D.)

Autômatos Finitos Determinísticos

Formalmente definimos um A.F.D. como sendo um sistema formal

$M = (K, \Sigma, \delta, qo, F)$, onde:

K → É um conjunto finito não vazio de Estados;

Σ → É um Alfabeto, finito, de entrada;

δ → Função de Mapeamento (ou função de transição) definida em: $K \times \Sigma \rightarrow K$

qo → $\in$ K, é o Estado Inicial

F → $\subseteq$ K, é o conjunto de Estados Finais

Interpretação de δ

A interpretação de uma transição $\delta(q, a) = p$, onde $q \wedge p \in K \wedge a \in \Sigma$, é a seguinte:

=> Se o "Controle de M" esta no estado "q" e o próximo símbolo de entrada é "a", então "a" deve ser reconhecido e o controle passar para o estado "p".

Autômatos Finitos Determinísticos (cont')

Formalmente definimos um A.F.D. como sendo um sistema formal

$M = (K, \Sigma, \delta, qo, F)$, onde:

K → É um conjunto finito não vazio de Estados;

Σ → É um Alfabeto, finito, de entrada;

δ → Função de Mapeamento (ou função de transição) definida em: $K \times \Sigma \rightarrow K$

qo → $\in$ K, é o Estado Inicial

F → $\subseteq$ K, é o conjunto de Estados Finais

Significado Lógico de um Estado

=> Logicamente um estado é uma situação particular no processo de reconhecimento de uma sentença.

Autômatos Finitos Determinísticos (cont')

Formalmente definimos um A.F.D. como sendo um sistema formal

$M = (K, \Sigma, \delta, qo, F)$, onde:

K → É um conjunto finito não vazio de Estados;

Σ → É um Alfabeto, finito, de entrada;

δ → Função de Mapeamento (ou função de transição) definida em: $K \times \Sigma \rightarrow K$

qo → $\in$ K, é o Estado Inicial

F → $\subseteq$ K, é o conjunto de Estados Finais

Sentenças Aceitas por M

=> Uma seqüência x é aceita (reconhecida) por um A.F. $M = (K, \Sigma\ \delta, qo, F)$,

$$\delta(qo, x) = p \mid p \in F.$$

Autômatos Finitos Determinísticos (cont')

Formalmente definimos um A.F.D. como sendo um sistema formal

$M = (K, \Sigma, \delta, qo, F)$, onde:

K → É um conjunto finito não vazio de Estados;

Σ → É um Alfabeto, finito, de entrada;

δ → Função de Mapeamento (ou função de transição) definida em: $K \times \Sigma \rightarrow K$

qo → $\in$ K, é o Estado Inicial

F → $\subseteq$ K, é o conjunto de Estad

Linguagem Aceita por M

=> É o conjunto de todas as sentenças aceitas por M. Formalmente, definimos por:

$$T(M) = \{x \mid \delta(qo, x) = p \wedge p \in F\}$$

OBS.: Todo conjunto aceito por um Autômato Finito é um Conjunto Regular.

Autômatos Finitos Determinísticos (cont')

Diagrama de Transição

=> Um diagrama de transição para um A.F. M é um grafo direcionado e rotulado, onde

- os vértices representam os estados e fisicamente são representados por círculos (sendo que o estado inicial é possui uma seta com rótulo "Inicio" e os estados finais são representados por círculos duplos), e

- as arestas representam as transições (sendo que, entre dois estados "p" e "q", existirá uma aresta direcionada de "p" para "q", com rótulo "a" ($a \in \Sigma$) $\Leftrightarrow \exists\ \delta(p, a) = q$ em M.

Autômatos Finitos Determinísticos (cont')

Tabela de Transições

=> É uma representação tabular de um A.F. Nesta tabela :

- as linhas representam os estados (o inicial é indicado por uma seta e os finais por asteriscos),

- as colunas representam os símbolos de entrada e o conteúdo da posição (q, a) será igual a "p" se existir $\delta(q, a) = p$, senão será indefinido.

Autômatos Finitos Determinísticos

Formalmente definimos um A.F.D. como sendo um sistema formal

$M = (K, \Sigma, \delta, qo, F)$, onde:

K → É um conjunto finito não vazio de Estados;

Σ → É um Alfabeto, finito, de entrada;

δ → Função de Mapeamento (ou função de transição) definida em: $K \times \Sigma \rightarrow K$

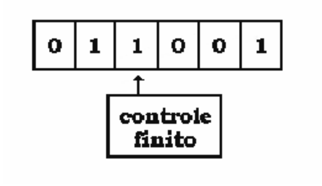
qo → $\in$ K, é o Estado Inicial

F → $\subseteq$ K, é o conjunto de Estados Finais

Autômatos Finitos Determinísticos (cont')

- Um autômato finito é representado através de um controle finito, que tem acesso a uma fita onde está a seqüência a ser analisada.

- O autômato percorre esta fita da esquerda para a direita, lendo um símbolo de cada vez.



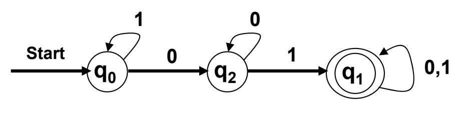
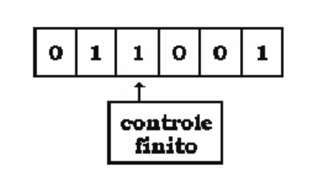


Diagrama de transição

$\delta(qo, x) = p$ => Transição de estados

AFD para reconhecer todas as strings com uma substring 01

Autômatos Finitos Determinísticos (cont')



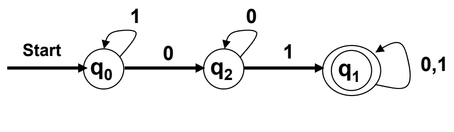


Diagrama de transição

$\delta(qo, x) = p$ => Transição de estados

M = (K, ∑, δ, q0, F)
K = (q0, q1, q2)
∑ = (0, 1)
F = (q1)

δ (q0, 0) = q2
δ (q0, 1) = q0
δ (q1, 0) = q1
δ (q1, 1) = q1
δ (q2, 0) = q2
δ (q2, 1) = q1

	0	1
$\rightarrow q_0$	q_2	q_0
$*q_1$	q_1	q_1
q_2	q_2	q_1

Tabela de Transições

Autômatos Finitos Não Determinísticos

Um A.F.N.D. é um sistema formal

M = (K, Σ, δ, qo, F) , onde:

K, Σ qo, F → possuem a mesma definição dos A.F.D.

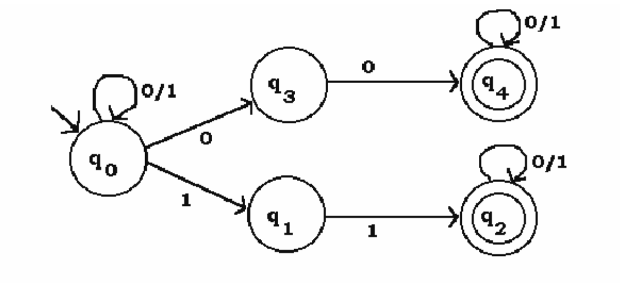
δ → É uma função de mapeamento, definido em K x Σ = ρ(K);

=> sendo que ρ(K) é um subconjunto de K; isto equivale a dizer que δ(q, a) = p1, p2, ...,pn.

A interpretação de δ é que <u>M</u> no estado "q", com o símbolo "a" na entrada pode ir tanto para o estado p1 como para o estado p2, ..., como para o estado pn.

Autômatos Finitos Não Determinísticos (cont')

AFND para reconhecer todas as sentenças que contenham : dois 0's ou dois 1's consecutivos



M = (K, ∑, δ, q0, F)

K = (q0, q1, q2, q3, q4)
∑ = (0, 1)
F = (q2, q4)

δ(q0, 0) = { q0, q3 }
δ(q1, 0) = ∅
δ(q2, 0) = { q2 }
δ(q3, 0) = { q4 }
δ(q4, 0) = { q4 }

δ(q0, 1) = { q0, q1 }
δ(q1, 1) = { q2 }
δ(q2, 1) = { q2 }
δ(q3, 1) = ∅
δ(q4, 1) = { q4 }

AFD X AFND

	Vantagem	Desvantagem
A.F.D.	Implementação Trivial	Não natural na representação de algumas L.R.
A.F.N.D.	Representação mais natural de algumas L.R.	Implementação complexa

Equivalência entre AFD's e AFND's

Teorema: Se L é um conjunto aceito por um autômato finito não-determinístico, então existe um autômato finito determinístico que aceita L.

Seja M = (K, ∑, δ, q0, F) um AFND que aceita L. Definimos um AFD M' = (K', ∑', δ', q0', F') tal que:

- Os estados de M' constituem todos os subconjuntos do conjunto de estados de M: K' = 2^K.
- F' é o conjunto de todos os estados de K' contendo um estado de F.
- Um elemento de K' é denotado por [q1 q2 ... qi] onde q1 q2 ... qi ∈ K.
- q0' = [q0]
- Definimos δ' ([q1 q2 ... qi], a) = [p1 p2 ... pj] se e somente se δ ({ q1, q2, ... qi },a) = { p1, p2, ... pj } = δ (q1, a) ∪ δ (q2, a) ∪ ... ∪ δ (qi, a)

Equivalência entre AFD's e AFND's (cont')

Exemplo: Dado o AFND M = ({ q0, q1 }, { 0, 1 }, δ, q0, { q1 }), onde:

δ (q0, 0) = { q0, q1 }
δ (q0, 1) = { q1 }
δ (q1, 0) = ∅
δ (q1, 1) = { q0, q1 }

Construir um AFD M' que reconheça a mesma linguagem que M.

M' = (K', { 0, 1 }, δ', [q0], F')

K' = { [q0], [q1], [q0 q1], ∅ }
F' = { [q1], [q0 q1] }

δ (q0, 0) = { q0, q1 }	δ' ([q0], 0) = [q0, q1]
δ (q0, 1) = { q1 }	δ' ([q0], 1) = [q1]
δ (q1, 0) = ∅	δ' ([q1], 0) = ∅
δ (q1, 1) = { q0, q1 }	δ' ([q1], 1) = [q0, q1]

Equivalência entre AFD's e AFND's (cont')

δ (q0, 0) = { q0, q1 }	δ' ([q0], 0) = [q0, q1]
δ (q0, 1) = { q1 }	δ' ([q0], 1) = [q1]
δ (q1, 0) = ∅	δ' ([q1], 0) = ∅
δ (q1, 1) = { q0, q1 }	δ' ([q1], 1) = [q0, q1]

δ' ([q0, q1], 0) = [q0, q1] desde que
δ ({q0, q1 }, 0) = δ (q0, 0) ∪ δ (q1, 0) = { q0, q1 } ∪ ∅ = { q0, q1 }

e δ' ([q0, q1], 1) = [q0, q1] desde que
δ ({q0, q1 }, 1) = δ (q0, 1) ∪ δ (q1, 1) = { q1 } ∪ { q0, q1 } = { q0, q1 }

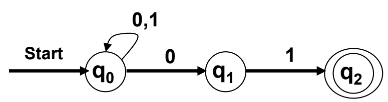
δ (∅, 0) = δ' (∅, 0) = ∅

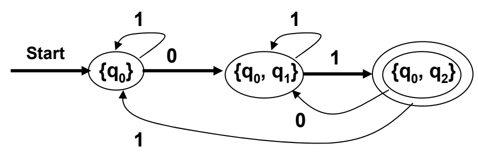
δ	0	1
→q_0	[q_0, q_1]	q_1
* q_1	∅	[q_0,q_1]

δ'	0	1
→[q_0]	[q_0,q_1]	[q_1]
* [q_1]	∅	[q_0,q_1]
* [q_0,q_1]	[q_0,q_1]	[q_0,q_1]

Equivalência entre AFD's e AFND's (cont')

Outro exemplo: Dado o AFND M = ({ q0, q1, q2 }, { 0, 1 }, δ, q0, { q2 })





	0	1
→q_0	{q_0, q1}	{q_0}
q_1	∅	{q_2}
*q_2	∅	∅

K' = 2^3 →

	0	1
∅	∅	∅
→{q_0}	{q_0, q_1}	{q_0}
{q_1}	∅	{q_2}
*{q_2}	∅	∅
{q_0,q_1}	{q_0,q_1}	{q_0,q_2}
*{q_0,q_2}	{q_0,q_1}	{q_0}
*{q_1,q_2}	∅	{q_2}
*{q_0, q_1,q_2}	{q_0,q_1}	{q_0, q_2}

Equivalência entre AFD's e AFND's (cont')

δD ({q0, q1 }, 0) = δN (q0, 0) ∪ δN (q1, 0) = {q0,q1} ∪ ∅ = {q0,q1}

δD ({q0, q1 }, 1) = δN (q0, 1) ∪ δN (q1, 1) = {q0} ∪ {q2} = {q0,q2}

δD ({q0, q2 }, 0) = δN (q0, 0) ∪ δN (q2, 0) = {q0, q1} ∪ ∅ = {q0,q1}

δD ({q0, q2 }, 1) = δN (q0, 1) ∪ δN (q2, 1) = {q0} ∪ ∅ = {q0}

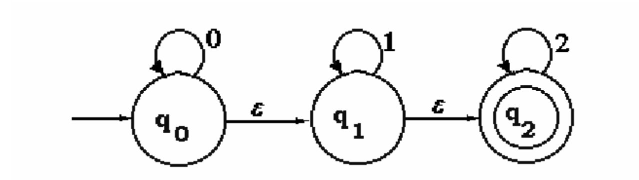
δD ({q1, q2 }, 0) = δN (q1, 0) ∪ δN (q2, 0) = ∅ ∪ ∅ = ∅

δD ({q1, q2 }, 1) = δN (q1, 1) ∪ δN (q2, 1) = ∅ ∪ ∅ = ∅

δD ({q0, q1, q2}, 0) = δN (q0, 0) ∪ δN (q1, 0) ∪ δN (q2, 0) = {q0,q1} ∪ ∅ ∪ ∅ = {q0,q1}

δD ({q0, q1, q2}, 1) = δN (q0, 1) ∪ δN (q1, 1) ∪ δN (q2, 1) = {q0} ∪ {q2} ∪ ∅ = {q0,q2}

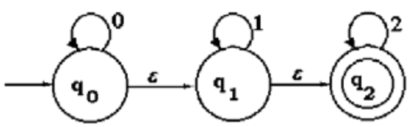
AFND com transições Epsilon (ε)



Define-se um AFND-ε como sendo uma quíntupla (K, ∑, δ, q0, F), onde:

- K é um conjunto finito, não vazio, de estados
- ∑ é um alfabeto finito de entrada
- q0 é o estado inicial e q0 ∈ K
- F é o conjunto de estados finais e F ∈ K
- δ é a função de transição de estados, mapeando K x (∑ ∪ {ε}). A intenção é que δ (q, a) consiste de todos os estados *p* tal que existe uma transição *a* de *q* para *p*, onde *a* é ε ou um símbolo de ∑.

AFND com transições Epsilon (ε) – (cont')



δ	0	1	2	ε
q_0	$\{q_0\}$	∅	∅	$\{q_1\}$
q_1	∅	$\{q_1\}$	∅	$\{q_2\}$
q_2	∅	∅	$\{q_2\}$	∅

Deve-se estender a função de transição δ para a função ξ que mapeia K x ∑* em 2^K.

=> O resultado desta função será todos os estados *p* tal que pode-se ir de *q* para *p* através de um caminho *w*, talvez incluindo-se arcos ε.

=> Na construção de ξ será importante saber o conjunto de estados atingíveis a partir de um determinado estado *q* usando-se somente ε => ε-CLOSURE(q)

No exemplo acima o ε-CLOSURE(q0) = {q0, q1, q2} .

AFND com transições Epsilon (ε) – (cont')

Considera-se ε-CLOSURE(P), onde *P* é um conjunto de estados, e q ∈ ε-CLOSURE(q). Podemos definir ξ como segue:

1) ξ(q, ε) = ε-CLOSURE(q).

2) Para w ∈ ∑* e a ∈ ∑, ξ(q, wa) = ε-CLOSURE(P),

onde P = {p | para algum *r* em ξ(q, w), *p* está em δ(r, a)}

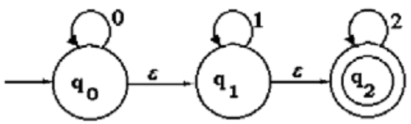
AFND com transições Epsilon (ε) – (cont')

Exemplo: Considere o exemplo anterior.

ξ(q0, ε) = ε-CLOSURE(q0) = {q0, q1, q2}   Então

ξ(q0, 0) = ε-CLOSURE(δ(ξ(q0, ε), 0))
ε-CLOSURE(δ({q0, q1, q2}, 0))
ε-CLOSURE(δ(q0, 0) ∪ δ(q1, 0) ∪ δ(q2, 0))
ε-CLOSURE({q0} ∪ ∅ ∪ ∅)
ε-CLOSURE({q0}) = {q0, q1, q2}
e

ξ(q0, 1) = ε-CLOSURE(δ(ξ(q0, ε), 1))
ε-CLOSURE(δ({q0, q1, q2}, 1))
ε-CLOSURE(δ(q0, 1) ∪ δ(q1, 1) ∪ δ(q2, 1))
ε-CLOSURE(∅ ∪ {q1} ∪ ∅)
ε-CLOSURE({q1}) = {q1, q2}



Relação entre Autômatos Finitos e Gramáticas Regulares

Teorema:

Se G = (N, T, P, S) é uma gramática do tipo regular então existe um autômato finito M = (K, T, δ, S, F) tal que T(M) = L(G).

. M é um AFND (ou AFND-ε)

. Os estados de M são as variáveis (NT) de G, mais um estado adicional A, A ∉ N => K = N ∪ {A}

. O estado inicial de M é S

. Se P tem produção S → ε então F = {S,A}, caso contrário F = {A}

. Transições de M:
1. δ(B,a) = A para cada B → a ∈ P
2. δ(B,a) = C para cada B → aC ∈ P
3. δ(A,a) = ∅ para todo a ∈ T

Relação entre Autômatos Finitos e Gramáticas Regulares (cont')

Exemplo:

G = ({S,B}, {0,1}, P, S)

P: S → 0B
B → 0B | 1S | 0

Construir um autômato finito que aceite L(G):

M = ({S,B,A}, {0,1}, δ, S, {A}) é um AFND (ou AFND-ε)

δ(S,0) = {B}	S → 0B
δ(S,1) = ∅	
δ(B,0) = {B,A}	B → 0B | 0
δ(B,1) = {S}	B → 1S

Equivalência entre Autômatos Finitos e Expressões Regulares

Teorema:

Considere que *r* é uma expressão regular. Então existe um AFND-ε que aceita L(r).

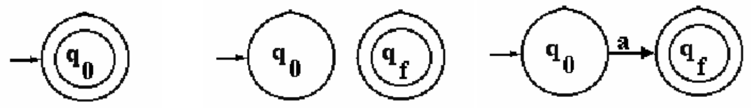
Prova:

Pode-se mostrar por indução no número de operadores da ER que existe um AFND-ε, tendo um estado final que não possui saída a partir dele, tal que L(M) = L(r).

Equivalência entre AF e ER (cont')

Base: (Sem operadores)

A ER r precisa ser ε, ∅, ou *a* para algum *a* ∈ ∑. Os AFND's abaixo claramente satisfazem as condições.



a) r = ε b) r = ϕ c) r = a

Equivalência entre AF e ER (cont')

Indução: (um ou mais operadores)

Considere r tendo i operadores, Existe 3 casos dependendo da forma de r.

1. Se α e β são ER's, então L((α ∪ β)) = L(α) ∪ L(β) => r = r1 + r2
2. Se α e β são ER's, então L((αβ)) = L(α) L(β) => r = r1 r2
3. Se α é uma ER, então L(α*) = L(α)* => $r = r_1{}^*$

Equivalência entre AF e ER (cont')

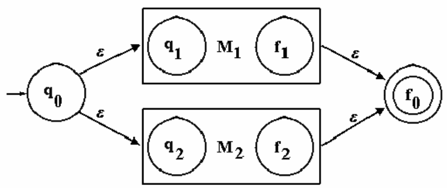
CASO 1: União r = r1 + r2

=> existe AFND-ε's M1 = (Q1, ∑1, δ1, q1, {f1}) e
M2 = (Q2, ∑2, δ2, q2, {f2})

Com L(M1) = L(r1) e L(M2) = L(r2).

Assumindo que Q1 e Q2 são desarticulados => Considere *q0* como sendo um novo estado e *f0* um novo estado final. Construa

M = (Q1 ∪ Q2 ∪ {q0, f0}, ∑1 ∪ ∑2, δ, q0, {f0})

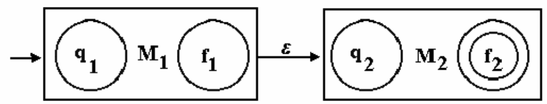


Equivalência entre AF e ER (cont')

CASO 2: Concatenação r = r1 r2

Considere M1 e M2 sendo como em CASO 1 e construa

M = (Q1 ∪ Q2, ∑1 ∪ ∑2, δ, {q0}, { f2})

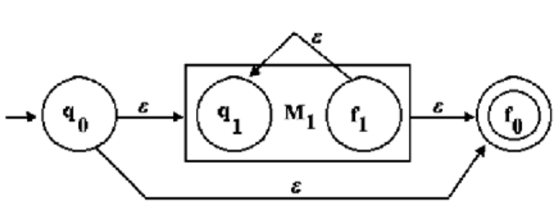


Equivalência entre AF e ER (cont')

CASO 3: Fechamento Reflexivo (Kleene star) $r = r_1^*$

Considere M1 sendo como em CASO 1 e L(M1) = r1, construa

$M = (Q1 \cup \{q0, f0\}, \sum 1, \delta, q0, \{ f0 \})$



Equivalência entre AF e ER (cont')

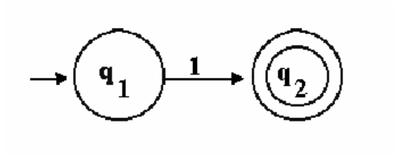
Exemplo:

Construir um AFND-ε para expressão regular 0 1* | 1.

=> Utilizando parênteses (precedência): (0(1*)) | 1

=> Essa expressão é na forma r1 + r2, onde r1 = 01* e r2 = 1.

=> O autômato para r2 é simples, sendo representado como:



r2 = 1

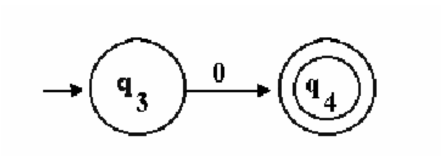
Equivalência entre AF e ER (cont')

Exemplo: (cont')

Construir um AFND-ε para expressão regular 0 1* | 1.

=> Pode-se expressar r1 como r3 r4, onde r3 = 0 e r4 = 1*.

=> O autômato para r3 seria também facilmente construído:



r3 = 0

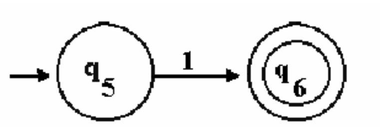
Equivalência entre AF e ER (cont')

Exemplo: (cont')

Construir um AFND-ε para expressão regular 0 1* | 1.

=> Pode-se notar que r4 é r5* onde r5 = 1

=> Um AF para r5 seria



r5 =1

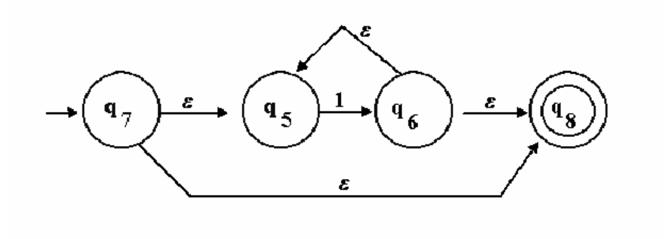
Equivalência entre AF e ER (cont')

Exemplo: (cont')

Construir um AFND-ε para expressão regular 0 1* | 1.

=> Para construir um AF para r5* usa-se o CASO 3 visto anteriormente.

=> O AF resultante é mostrado abaixo.



r5*

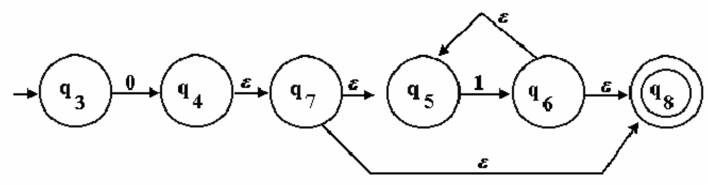
Equivalência entre AF e ER (cont')

Exemplo: (cont')

Construir um AFND-ε para expressão regular 0 1* | 1.

=> r1 = r3 r4 usa-se o CASO 2 visto anteriormente.

=> o AF resultante seria o seguinte:



r1 = r3 r4 => r1 = 01*

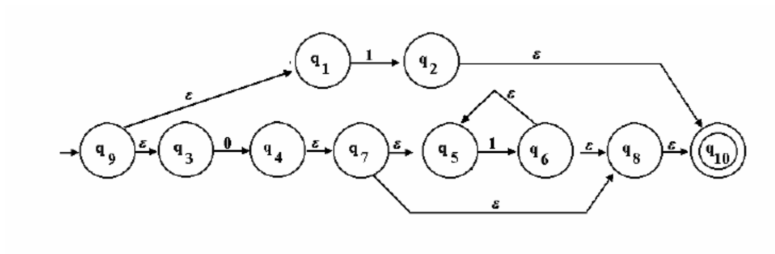
Equivalência entre AF e ER (cont')

Exemplo: (cont')

Construir um AFND-ε para expressão regular 0 1* | 1.

=> utiliza-se o CASO 1 para completar o AF para r = r1 + r2.

=> O AF final é mostrado abaixo.



r = r1 + r2 => r = 0 1* | 1

Autômatos Finitos

- Formalmente definimos um A.F. Determinístico (AFD) como sendo um sistema formal

  M = (K, Σ, δ, qo, F), onde: ...

- Autômatos Finitos Não-Determinísticos (AFND)
- Equivalência entre um AFD e um AFND
- Autômatos Finitos Não-Determinísticos com transições ε (AFNDε)
- Relação entre Autômatos Finitos e Gramáticas Regulares
- Equivalência entre AF e GR

Eficiência de um AF como Algoritmo de Reconhecimento

- **Simulador de qualquer AFD**

  - Fácil implementação => algoritmo que controla a mudança de estado a cada símbolo lido da entrada

- **Tempo de processamento**

  - Para aceitar ou rejeitar => diretamente proporcional ao tamanho da entrada

  - Não depende do AFD => qualquer AFD que reconheça a linguagem terá a mesma eficiência

- **Otimização?**

  - Redução do número de estados
  - Existe um algoritmo para construir um AFD mínimo => AFD com o menor número de estados

Minimização de um Autômato Finito

- **Objetivo**

  • Gerar um AF equivalente com o menor número de estados possível

- **Minimização do número de estados**

  • Adotada na maioria das soluções práticas

  => entretanto, em algumas aplicações minimizar o número de estados pode não implicar no menor custo de implementação

- **Exemplo :desenho de circuitos eletrônicos**

  => pode ser desejável introduzir estados intermediários para melhorar a eficiência ou simplesmente facilitar as ligações físicas

  => nestes casos o algoritmo deve ser modificado prevendo as variáveis específicas da aplicação

Minimização de um Autômato Finito (cont’)

- **O autômato mínimo é único**

  • A minimização de AF distintos que aceitam a mesma linguagem geram o mesmo AF mínimo

- **Idéia básica do algoritmo**

  • Unificar os *estados equivalentes*

Definição: Estados Equivalentes

• q e p são equivalentes sse para qq w, δ(q, w) e δ (p, w)

=> Resultam simultaneamente em estados finais, ou não-finais

=> Ou seja, o processamento de uma entrada qq a partir de estados equivalentes gera o mesmo resultado (aceita/rejeita)

Minimização de um Autômato Finito (cont’)

- **Pré-Requisitos do Algoritmo**

  • AF deve ser determinístico

  • O AF não pode ter estados inacessíveis => não-atingíveis a partir do estado inicial

  • A função programa deve ser total => a partir de qualquer estado são previstas transições para todos os símbolos do alfabeto

Minimização de um Autômato Finito (cont')

➢ Caso o AF não satisfaça algum dos pré-requisitos

a) Gerar um AFD equivalente => Equivalência entre um AFD e um AFND

3. Autômatos Finitos

Minimização de um Autômato Finito (cont')

➢ Caso o AF não satisfaça algum dos pré-requisitos

a) Gerar um AFD equivalente => Equivalência entre um AFD e um AFND

b) Eliminar os estados inacessíveis e suas correspondentes transições

	δ	a	b
$\rightarrow$	q_0	q_1	q_5
	q_1	-	q_2
*	q_2	q_3	q_2
*	q_3	q_3	q_3
	q_4	q_4	q_1

⇨

	δ	a	b
$\rightarrow$	q_0	q_1	q_5
	q_1	-	q_2
*	q_2	q_3	q_2
*	q_3	q_3	q_3

estado inacessível : q4 => Sai da tabela

3. Autômatos Finitos

Minimização de um Autômato Finito (cont')

➢ Caso o AF não satisfaça algum dos pré-requisitos

a) Gerar um AFD equivalente => Equivalência entre um AFD e um AFND

b) Eliminar os estados inacessíveis e suas correspondentes transições

c) A função programa deve ser total

1. Introduzir um novo estado não-final *d*

2. Incluir as transições não-previstas, tendo como resultado o estado d

3. Incluir um ciclo em d para todos os símbolos do alfabeto

3. Autômatos Finitos

Minimização de um Autômato Finito (cont')

Idéia básica do algoritmo

➢ Identificar os estados equivalentes

. por *exclusão*

1. A partir de uma tabela de estados

. são marcados os estados não-equivalentes

2. Ao final do algoritmo

. as referências não-marcadas => representam os estados equivalentes.

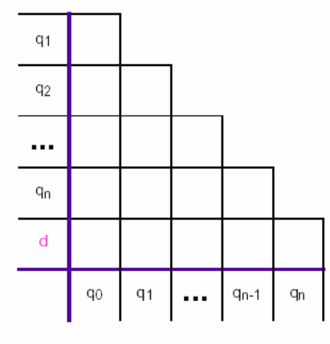
Minimização de um Autômato Finito (cont')

Algoritmo de Minimização

- Seja $M = (K, \Sigma, \delta, q0, F)$ um AFD

  . Satisfaz aos pré-requisitos

(1) Descrever uma tabela => relaciona os estados distintos



Minimização de um Autômato Finito (cont')

Algoritmo de Minimização (cont')

(2) Marcar na tabela os pares não-equivalentes

. {estado final, estado não-final}
. Estados finais não são equivalentes a não-finais

Minimização de um Autômato Finito (cont')

Algoritmo de Minimização (cont')

(3) Verificar os pares não-marcados da tabela

Para $\{q_u, q_v\}$ não-marcado e $a \in \Sigma$

- Suponha $\delta(q_u, a) = p_u$ e $\delta(q_v, a) = p_v$
- Se $p_u = p_v$

  . q_u é equivalente a q_v

  . Para o símbolo a => não marcar

- Se pu ≠ pv e o par $\{p_u, p_v\}$ é não-marcado

  . $\{q_u, q_v\}$ é incluído em uma lista a partir de $\{p_u, p_v\}$ para posterior análise

Minimização de um Autômato Finito (cont')

Algoritmo de Minimização (cont')

(3) Verificar os pares não-marcados da tabela (cont')

- Se $p_u \neq p_v$ e o par $\{p_u, p_v\}$ é marcado

  . $\{q_u, q_v\}$ é não-equivalente => marcar

  . Se $\{q_u, q_v\}$ encabeça uma lista => marcar todos os pares da lista e, recursivamente, se algum par da lista encabeça outra lista

Minimização de um Autômato Finito (cont')

Algoritmo de Minimização (cont')

(4) Unificar os pares equivalentes

- **Pares não-marcados são equivalentes => unificar**
  - **A equivalência de estados é transitiva**
    - **=> Se os estados p e q são equivalentes, e os estados q e r são equivalentes, então p e r também são equivalentes.**
  - **Pares de estados equivalentes =>unificados como um único estado**
  - **Se algum dos estados equivalentes é inicial => estado unificado é inicial**

Minimização de um Autômato Finito (cont')

Algoritmo de Minimização (cont')

(5) Excluir os estados inúteis

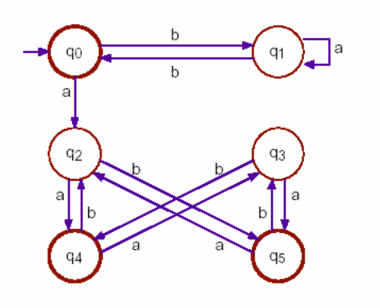
- **Um estado q é inútil**
  - **. Se é não-final => a partir de q não é possível atingir um estado final**
- **O estado d**
  - **. Se incluído => sempre é inútil**

Minimização de um Autômato Finito (cont')

***Exemplo*: Considere o AFD**

M = ({q0,q1,q2,q3,q4,q5}, {a,b}, δ, {q0}, {q0, q4, q5})

• satisfaz os pré-requisitos de minimização



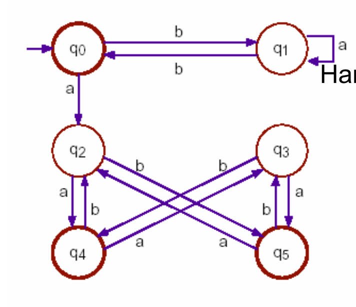
q1					
q2					
q3					
q4					
q5					
	q0	q1	q2	q3	q4

Passo 1: Construção da tabela

Minimização de um Autômato Finito (cont')

***Exemplo*: Considere o AFD (cont')**

M = ({q0,q1,q2,q3,q4,q5}, {a,b}, δ, {q0}, {q0, q4, q5})



Hamilton Luiz

q1					
q2					
q3					
q4					
q5					
	q0	q1	q2	q3	q4

Passo 2: Marcar pares não-equivalentes {estado final, estado não-final}

Minimização de um Autômato Finito (cont')

Exemplo: Considere o AFD (cont')

M = ({q0,q1,q2,q3,q4,q5}, {a,b}, δ, {q0}, {q0, q4, q5})

q1	×				
q2	×				
q3	×				
q4	☺	×	×	×	
q5		×	×	×	
	q0	q1	q2	q3	q4

Passo 3: Verificar os pares não-marcados

- $\{q_0, q_4\}$

  $\delta(q_0, a) = q_2$ $\delta(q_4, a) = q_3$

  $\delta(q_0, b) = q_1$ $\delta(q_4, b) = q_2$

  $\{q_1, q_2\}$ e $\{q_2, q_3\}$ são não-marcados:

  $\{q_0, q_4\}$ é *incluído* nas listas de $\{q_1, q_2\}$ e $\{q_2, q_3\}$

Listas anotadas:
L1 => {q1,q2}, {q0,q4}
L2 => {q2,q3}, {q0,q4}

➢ Se pu ≠ pv e o par $\{p_u, p_v\}$ é não-marcado
. $\{q_u, q_v\}$ é incluído em uma lista a partir de $\{p_u, p_v\}$
para posterior análise

3. Autômatos Finitos

Minimização de um Autômato Finito (cont')

Exemplo: Considere o AFD (cont')

M = ({q0,q1,q2,q3,q4,q5}, {a,b}, δ, {q0}, {q0, q4, q5})

q1	×				
q2	×				
q3	×				
q4		×	×	×	
q5	☺	×	×	×	
	q0	q1	q2	q3	q4

Passo 3: Verificar os pares não-marcados (cont')

- $\{q_0, q_5\}$

  $\delta(q_0, a) = q_2$ $\delta(q_5, a) = q_2$

  $\delta(q_0, b) = q_1$ $\delta(q_5, b) = q_3$

  $\{q_1, q_3\}$ é não-marcado (e como $\{q_2, q_2\}$ é trivialmente equivalente):

  $\{q_0, q_5\}$ é *incluído* na lista de $\{q_1, q_3\}$

Listas anotadas:
L1 => {q1,q2}, {q0,q4}
L2 => {q2,q3}, {q0,q4}
L3 => {q1,q3}, {q0,q5}

➢ Se pu ≠ pv e o par $\{p_u, p_v\}$ é não-marcado

. $\{q_u, q_v\}$ é incluído em uma lista a partir de $\{p_u, p_v\}$
para posterior análise

3. Autômatos Finitos

Minimização de um Autômato Finito (cont')

Exemplo: Considere o AFD (cont')

M = ({q0,q1,q2,q3,q4,q5}, {a,b}, δ, {q0}, {q0, q4, q5})

q1	×				
q2	×	⊗			
q3	×				
q4	⊗	×	×	×	
q5		×	×	×	
	q0	q1	q2	q3	q4

Passo 3: Verificar os pares não-marcados (cont')

- $\{q_1, q_2\}$

  $\delta(q_1, a) = q_1$ $\delta(q_2, a) = q_4$

  $\delta(q_1, b) = q_0$ $\delta(q_2, b) = q_5$

  $\{q_1, q_4\}$ é marcado: $\{q_1, q_2\}$ é *marcado*

  $\{q_1, q_2\}$ encabeça uma lista: $\{q_0, q_4\}$ é *marcado*

Listas anotadas:
L1 => {q1,q2}, {q0,q4}
L2 => {q2,q3}, {q0,q4}
L3 => {q1,q3}, {q0,q5}

➢ Se $p_u \neq p_v$ e o par $\{p_u, p_v\}$ é marcado
. $\{q_u, q_v\}$ é não-equivalente => marcar
. Se $\{q_u, q_v\}$ encabeça uma lista => marcar todos os pares da lista e, recursivamente, se algum par da lista encabeça outra lista

3. Autômatos Finitos

Minimização de um Autômato Finito (cont')

Exemplo: Considere o AFD (cont')

M = ({q0,q1,q2,q3,q4,q5}, {a,b}, δ, {q0}, {q0, q4, q5})

q1	×				
q2	×	⊗			
q3	×	⊗			
q4	⊗	×	×	×	
q5	⊗	×	×	×	
	q0	q1	q2	q3	q4

Passo 3: Verificar os pares não-marcados (cont')

- $\{q_1, q_3\}$

  $\delta(q_1, a) = q_1$ $\delta(q_3, a) = q_5$

  $\delta(q_1, b) = q_0$ $\delta(q_3, b) = q_4$

  $\{q_1, q_5\}$ e $\{q_0, q_4\}$ são marcados: $\{q_1, q_3\}$ é *marcado*

  $\{q_1, q_3\}$ encabeça uma lista: $\{q_0, q_5\}$ é *marcado*

Listas anotadas:
L1 => {q1,q2}, {q0,q4}
L2 => {q2,q3}, {q0,q4}
L3 => {q1,q3}, {q0,q5}

➢ Se $p_u \neq p_v$ e o par $\{p_u, p_v\}$ é marcado
. $\{q_u, q_v\}$ é não-equivalente => marcar
. Se $\{q_u, q_v\}$ encabeça uma lista => marcar todos os pares da lista e, recursivamente, se algum par da lista encabeça outra lista

Minimização de um Autômato Finito (cont')

***Exemplo*: Considere o AFD (cont')**

M = ({q0,q1,q2,q3,q4,q5}, {a,b}, δ, {q0}, {q0, q4, q5})

q_1	×				
q_2	×	⊗			
q_3	×	⊗	☺		
q_4	⊗	×	×	×	
q_5	⊗	×	×	×	
	q_0	q_1	q_2	q_3	q_4

Passo 3: Verificar os pares não-marcados (cont')

- $\{q_2, q_3\}$

  $\delta(q_2, a) = q_4$ $\delta(q_3, a) = q_5$

  $\delta(q_2, b) = q_5$ $\delta(q_3, b) = q_4$

  $\{q_4, q_5\}$ é não-marcado:

  $\{q_2, q_3\}$ é *incluído* na lista de $\{q_4, q_5\}$

Listas anotadas:
L1 => {q1,q2}, {q0,q4}
L2 => {q2,q3}, {q0,q4}
L3 => {q1,q3}, {q0,q5}
L4 => {q4,q5}, {q2,q3}

➢ **Se pu ≠ pv e o par $\{p_u, p_v\}$ é não-marcado**

. $\{q_u, q_v\}$ é incluído em uma lista a partir de $\{p_u, p_v\}$ para posterior análise

Minimização de um Autômato Finito (cont')

***Exemplo*: Considere o AFD (cont')**

M = ({q0,q1,q2,q3,q4,q5}, {a,b}, δ, {q0}, {q0, q4, q5})

q_1	×				
q_2	×	⊗			
q_3	×	⊗			
q_4	⊗	×	×	×	
q_5	⊗	×	×	×	☺
	q_0	q_1	q_2	q_3	q_4

Passo 3: Verificar os pares não-marcados (cont')

- $\{q_4, q_5\}$

  $\delta(q_4, a) = q_3$ $\delta(q_5, a) = q_2$

  $\delta(q_4, b) = q_2$ $\delta(q_5, b) = q_3$

  Como $\{q_2, q_3\}$ é não-marcado:

  $\{q_4, q_5\}$ é *incluído* na lista de $\{q_2, q_3\}$

Listas anotadas:
L1 => {q1,q2}, {q0,q4}
L2 => {q2,q3}, {q0,q4}, {q4,q5}
L3 => {q1,q3}, {q0,q5}
L4 => {q4,q5}, {q2,q3}

➢ **Se pu ≠ pv e o par $\{p_u, p_v\}$ é não-marcado**

. $\{q_u, q_v\}$ é incluído em uma lista a partir de $\{p_u, p_v\}$ para posterior análise

Minimização de um Autômato Finito (cont')

***Exemplo*: Considere o AFD (cont')**

M = ({q0,q1,q2,q3,q4,q5}, {a,b}, δ, {q0}, {q0, q4, q5})

q_1	×					$\{q_0, q_5\}$
q_2	×	⊗				$\{q_0, q_4\}$
q_3	×	⊗	✓			$\{q_0, q_4\}$ → $\{q_4,$
q_4	⊗	×	×	×		
q_5	⊗	×	×	×	✓	$\{q_2, q_3\}$
	q_0	q_1	q_2	q_3	q_4	

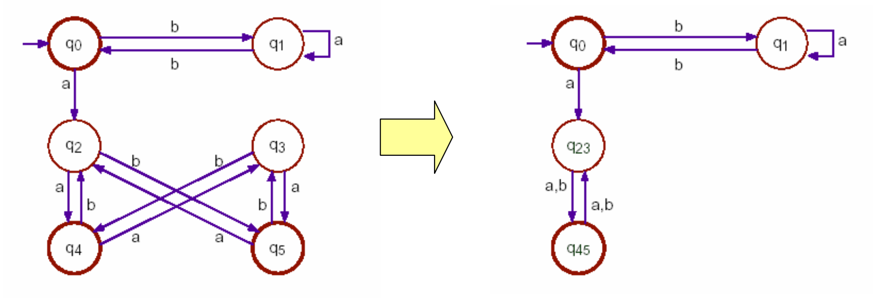
Passo 4: Unificar os estados equivalentes

- Como os pares $\{q_2, q_3\}$ e $\{q_4, q_5\}$ são não-marcados
  - q_{23}: unificação dos estados não-finais q_2 e q_3;
  - q_{45}: unificação dos estados finais q_4 e q_5.

Minimização de um Autômato Finito (cont')

***Exemplo*: Considere o AFD (cont')**

M = ({q0,q1,q2,q3,q4,q5}, {a,b}, δ, {q0}, {q0, q4, q5})



Minimização de um Autômato Finito (cont')

Conclusão:

- O AFD construído usando o algoritmo de minimização
- O AFD mínimo é o autômato com menor número de estados para a linguagem
- O *AFD mínimo* de uma linguagem é único

1. Introdução
2. Gramáticas
3. Autômatos finitos
4. Linguagens livre de contexto
5. Maquinas de Turing
6. Decidibilidade
7. Complexidade computacional
8. Problemas NP completos
9. Introdução às linguagens formais

Intuição

As Gramáticas Livres de Contexto tem grande importância dentro do estudo das Linguagens Formais

=> Através delas pode ser descrita a maior parte das construções sintáticas das linguagens de programação.

- Permite tratar questões como:
  - a) Parenteses Balanceados,
  - b) Construções Bloco-Estruturadas
  - c) Outras estruturas próprias de linguagens como C, Pascal, etc.
- Os algoritmos que as implementam são simples e eficientes.
- Aplicações: analisadores sintáticos, tradutores de linguagens e processadores de texto etc.

Intuição (cont')

O estudo das LLC é desenvolvido a partir de um formalismo gerador (gramática) e um formalismo reconhecedor (autômato), como segue:

- Gramáticas Livres de Contexto: São gramáticas onde as regras de produção são definidas de forma mais livre do que nas gramáticas regulares,
- Autômato com Pilha: Possui a estrutura básica de um AFD ao qual é associado uma memória auxiliar na forma de pilha e a facilidade de não-determinismo.

Obs: As LLCs são desenvolvidas a partir das GLCs.

Definição

Pode-se estender a definição de GLC para permitir quaisquer produções da forma:

$A \to \varepsilon$

=> Estas produções, cujo lado direito contém somente a sentença vazia, são chamadas de ε-produções.

Definição : Gramática Livre de Contexto

Definição:

Uma Gramática Livre de Contexto (GLC) G é uma gramática

$G = (V, T, P, S)$,

com a restrição de que qualquer regras de produção em P é da forma

$P = \{ A \to \beta \mid A \in N, \beta \in (N \cup T)^* \}$

=> Portanto uma GLC é uma gramática onde o lado esquerdo das produções possui exatamente uma variável.

GLC ε-livre

Uma GLC é ε-livre quando

=> não possui ε-produções ou

=> quando possui uma única ε-produção,

$S \to \varepsilon$,

onde S é o símbolo inicial da gramática e S não aparece do lado direito de nenhuma regra de produção.

Definição : Linguagem Livre de Contexto

Uma linguagem é uma LLC (ou do Tipo 2 na Classificação de Chomsky), se for gerada por uma GLC.

=> A expressão "livre de contexto" significa que para tais linguagens, cuja produção é da forma $A \to \alpha$, em uma derivação a variável A deriva α sem depender (livre) de qualquer análise dos símbolos que antecedem ou seguem A (contexto).

=> Assim claramente toda LR é também LLC.

Universo de Todas as Linguagens $\supset$ *LLC* $\supset$ *LR*

Exemplo1 : Linguagem Livre de Contexto

A linguagem $L = \{a^nb^n \mid n \geq 0\}$ é gerada pela seguinte GLC:

$G = (\{S\}, \{a, b\}, P, S)$, onde $P = \{S \rightarrow aSb \mid \varepsilon \}$.

Por exemplo, a palavra aabb pode ser gerada pela seguinte seqüência de derivações:

$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aa\varepsilon bb \Rightarrow aabb$

=> Esta linguagem é um exemplo clássico e de fundamental importância no estudo das LLC, pois permite estabelecer analogia entre a^nb^n e as linguagens bloco-estruturadas do tipo $BEGIN^nEND^n$, ou com expressões com parênteses balanceados na forma $(^n)^n$.

Exemplo2 : Linguagem Livre de Contexto

A linguagem gerada pela GLC abaixo é composta por expressões aritméticas contendo colchetes balanceados, dois operandos e um operador:

$G = (\{E\}, \{+, *, [,], x\}, P, E)$, onde $P = \{E \rightarrow E+E \mid E*E \mid [E] \mid x\}$.

Por exemplo, a expressão [x+x]*x pode ser gerada pela seguinte seqüência de derivações:

$E \Rightarrow E*E \Rightarrow [E]*E \Rightarrow [E+E]*E \Rightarrow [x+E]*E \Rightarrow [x+x]*E \Rightarrow [x+x]*x$

Árvores de Derivação para GLC

=> As árvores de derivação são representações gráficas para as derivações nas GLC.

Através destas, temos representada explicitamente a estrutura hierárquica que está implícita na linguagem.

Formalmente, consideremos que G = (N, T, P, S) seja uma GLC.

Uma árvore é uma *árvore de derivação* para G se:

1. Todo nodo tem um rótulo que é um símbolo de $N \cup T \cup \{\varepsilon\}$;
2. O rótulo da raiz é S;
3. Se um nodo A tem um ou mais descendentes, então A é um elemento de N;
4. Se A1, A2, ..., An são descendentes diretos de A, da esquerda para a direita, então $A \rightarrow A1\ A2 \ldots An$ é uma produção de P;
5. Se D é a única subárvore da raiz e tem rótulo ε, então a regra $S \rightarrow \varepsilon \in P$.

Exemplo: Árvores de Derivação para GLC

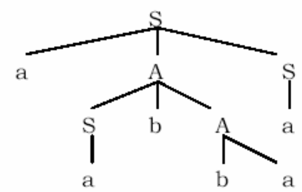
Considere a gramática G = ({S, A}, {a, b}, P, S), onde P consiste

$S \rightarrow aAS \mid a$
$A \rightarrow SbA \mid SS \mid ba$

a derivação da sentença *aabbaa* é dada por:

$S \rightarrow aAS \rightarrow aSbAS \rightarrow aabAS \rightarrow aabbaS \rightarrow aabbaa$

A árvore de derivação correspondente a essa sentença seria:



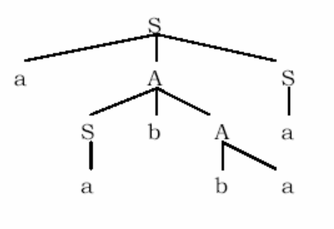
Mais Definições ...

➢ **Profundidade da Árvore de Derivação**

É o comprimento do maior caminho entre a raiz e um nodo terminal. No exemplo anterior, a árvore de derivação da sentença tem profundidade 3.

➢ **Limite de uma Árvore de Derivação**

É a seqüência formada pela concatenação, da esquerda para a direita, das folhas da árvore de derivação. (aabbaa)

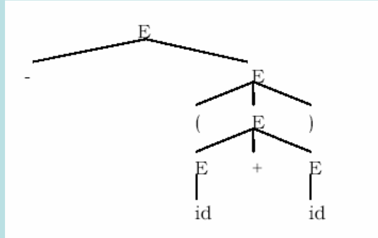


Derivação mais à esquerda e mais à direita

Uma árvore de derivação ignora variações na ordem em que os símbolos foram substituídos na derivação. Por exemplo:

G = ({E}, {+, *. (,), -, id}, P, E)

Onde P= E → E + E
E → E * E
E → (E)
E → - E
E → id



a sentença "- (id * id)" pode ser derivada de dois modos diferentes:

E → - E → - (E) → - (E + E) → - (id + E) → - (id + id)

ou

E → - E → - (E) → - (E + E) → - (E + id) → - (id + id)

As derivações acima correspondem à mesma árvore de derivação:

Derivação mais à esquerda e mais à direita (cont')

➢ Uma derivação é chamada de mais à esquerda quando o símbolo substituído for o não-terminal mais à esquerda da forma sentencial.

➢ Na derivação mais à direita, o símbolo substituído é o não-terminal mais à direita.

Nas duas derivações da sentença "- (id + id)" mostradas acima, a primeira é mais à esquerda e a segunda mais à direita.

Derivação mais à esquerda e mais à direita (cont')

Exemplo:

Para a mesma gramática anterior, obtenha as derivações mais à esquerda e mais à direita da sentença

id + id * id

Solução:

➢ **Derivação mais à esquerda**

E → E + E → id + E → id + E * E → id + id * E → id + id * id

➢ **Derivação mais à direita**

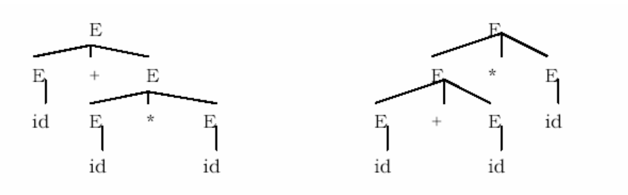
E → E + E → E + E * E → E + E * id → E + id * id → id + id * id

Gramática Ambígua

=> Uma GLC é ambígua quando, para alguma sentença da linguagem gerada, existe mais de uma árvore de derivação.

Exemplo:

A gramática de expressão aritmética apresentada antes é ambígua. Isto pode ser visto através de duas árvores de derivação diferentes para a sentença vista: id + id * id



As duas derivações analisadas no exemplo anterior estão representadas na primeira árvore de derivação.

Linguagens Inerentemente Ambíguas

É uma linguagem para a qual todas as GLC que a geram são ambíguas.

Exemplo de linguagem inerentemente ambígua:

$L = \{ a^n b^n c^m d^m \mid n >= 1, m >= 1\} \cup \{ a^n b^m c^m d^n \mid n >= 1, m >= 1\}$

Transformações em GLC

Algumas transformações podem efetuadas em GLC's com o objetivo de torná-las mais simples ou de prepará-las para posteriores aplicações.

=> É importante notar que, qualquer que seja a transformação efetuada, a linguagem gerada deverá ser sempre a mesma.

1) Eliminação de Símbolos Inúteis
2) Transformação de uma GLC qualquer para uma GLC ε-Livre
3) Remoção de Produções Simples (unitárias)
4) Fatoração de GLC
5) Eliminação de Recursão à Esquerda (e a Direita)

1. Eliminação de Símbolos Inúteis

Em uma GLC, um símbolo (terminal ou não-terminal) é inútil se ele não aparece na derivação de nenhuma sentença.

=> Um símbolo é inútil se ele é:

estéril => não gera nenhuma seqüência de terminais pertencente a uma sentença ou também chamado

inalcançável => não aparece em nenhuma forma sentencial da gramática.

1. Eliminação de Símbolos Inúteis (cont')

Determinação do conjunto de símbolos férteis

Pode ser efetuada através do seguinte algoritmo:

a) Construir o conjunto $N_0 = \varnothing$ e fazer $i = 1$

b) Repetir

$N_i = N_{i-1} \cup \{ A \mid A \rightarrow \alpha \in P \text{ e } \alpha \in (N_{i-1} \cup T)^* \}$

$i = i + 1$

até que $N_i = N_{i-1}$

c) N_i é o conjunto de símbolos férteis.

=> Se o símbolo inicial não fizer parte do conjunto de símbolos férteis, a linguagem gerada pela gramática é vazia.

1. Eliminação de Símbolos Inúteis (cont')

Determinação do conjunto de símbolos férteis (cont')

Exemplo:

Retirar os símbolos estéreis da gramática:

G = ({S,A,B,C,D}, {a,b,c,d}, P, S)

P: S → a A
A → a | b B
B → b | d D
C → c C | c
D → d D

Solução:

G = ({S,A,B,C,D}, {a,b,c,d}, P, S)

N0 = ∅

P/ i=1 :

S→aA => N1 = No ∪ {S → a A | S → aA ∈ P e aA ∈ (N0 ∪ T)*} => N1 = ∅

A→a|bB => N1 = No ∪ {A→a|bB | A→a|bB ∈ P e a|bB ∈ (N0 ∪ T)*} => N1 = {A}

B→b|dD => N1 = No ∪ {B→b|dD | B→b|dD ∈ P e b|dD ∈ (N0 ∪ T)*} => N1 = {A, B}

C→cC|c => N1 = No ∪ {C→cC|c | C→cC|c ∈ P e cC|c ∈ (N0 ∪ T)*} => N1 = {A, B, C}

D→dD => N1 = No ∪ {D→dD | D→dD ∈ P e dD ∈ (N0 ∪ T)*} => N1 = {A, B, C}

1. Eliminação de Símbolos Inúteis (cont')

Determinação do conjunto de símbolos férteis (cont')

Exemplo: (cont')
Retirar os símbolos estéreis da gramática:

G = ({S,A,B,C,D}, {a,b,c,d}, P, S)

P: S → a A
A → a | b B
B → b | d D
C → c C | c
D → d D

Solução: (cont')

G = ({S,A,B,C,D}, {a,b,c,d}, P, S)

N0 = ∅, N1 = {A, B, C}

P/ i=2 :

S→aA => N2 = N1 ∪ {S → a A | S → aA ∈ P e aA ∈ (N1 ∪ T)*} => N2 = {S,A, B,C}

A→a|bB => N2 = N1 ∪ {A→a|bB | A→a|bB ∈ P e a|bB ∈ (N1 ∪ T)*} => N2 = {S,A, B,C}

B→b|dD => N2 = N1 ∪ {B→b|dD | B→b|dD ∈ P e b|dD ∈ (N1 ∪ T)*} => N2 = {S,A, B,C}

C→cC|c => N2 = N1 ∪ {C→cC|c | C→cC|c ∈ P e cC|c ∈ (N1 ∪ T)*} => N2 = {S,A, B,C}

1. Eliminação de Símbolos Inúteis (cont')

Determinação do conjunto de símbolos férteis (cont')

Exemplo: (cont')

Reti

G =

Solução: (cont')

G = ({S,A,B,C,D}, {a,b,c,d}, P, S)

N0 = ∅, N1 = {A, B, C}, N2 = {S, A, B, C}

P/ i=3 :

S→aA => N3 = N2 ∪ {S → a A | S → aA ∈ P e aA ∈ (N2 ∪ T)*} => N3 = {S,A, B,C}

A→a|bB => N3 = N2 ∪ {A→a|bB | A→a|bB ∈ P e a|bB ∈ (N2 ∪ T)*} => N3 = {S,A, B,C}

B→b|dD => N3 = N2 ∪ {B→b|dD | B→b|dD ∈ P e b|dD ∈ (N2 ∪ T)*} => N3 = {S,A, B,C}

C→cC|c => N3 = N2 ∪ {C→cC|c | C→cC|c ∈ P e cC|c ∈ (N2 ∪ T)*} => N3 = {S,A, B,C}

D→dD => N3 = N2 ∪ {D→dD | D→dD ∈ P e dD ∈ (N2 ∪ T)*} => N3 = {S, A, B, C}

1. Eliminação de Símbolos Inúteis (cont')

Determinação do conjunto de símbolos férteis (cont')

Exemplo:

Retirar os símbolos estéreis da gramática:

G = ({S,A,B,C,D}, {a,b,c,d}, P, S)

P: S → a A
A → a | b B
B → b | d D
C → c C | c
D → d D

Solução:

N0 = ∅
N1 = {A,B,C}
N2 = {S,A,B,C}
N3 = {S,A,B,C} = N2

Conjunto de símbolos férteis: {S,A,B,C}
Gramática simplificada:

G' = ({S,A,B,C}, {a,b,c}, P', S)
P': S → a A
A → a | b B
B → b
C → cC | c

1. Eliminação de Símbolos Inúteis (cont')

Determinação do conjunto de símbolos alcançáveis

Pode ser efetuada através do seguinte algoritmo:

a) Construir o conjunto V_0 = {S} (S = símbolo inicial) e fazer i = 1

b) Repetir
$V_i = V_{i-1} \cup \{ X |$ existe algum $A \rightarrow \alpha X \beta$ e $A \in V_{i-1}$ e $\alpha, \beta \in (N \cup T)^* \}$
i = i + 1
até que $V_i = V_{i-1}$

c) V_i é o conjunto de símbolos alcançáveis.

1. Eliminação de Símbolos Inúteis (cont')

Determinação do conjunto de símbolos alcançáveis (cont')

Exemplo: Simplificar a gramática G' do exemplo anterior, retirando os símbolos inalcançáveis.

Solução:

V0 = {S}
V1 = {S, a, A}
V2 = {S, a, A, b, B}
V3 = {S, a, A, b, B} = V2

Conjunto de símbolos alcançáveis: {S, a, A, b, B}

Gramática simplificada:
G' = ({S,A,B}, {a,b}, P'', S)
P'': S → a A
A → a | b B
B → b

2. Transformação de uma GLC qualquer para uma GLC ε-Livre (cont')

Esta transformação sempre é possível e pode ser efetuada pelo seguinte algoritmo:

a) Reunir em um conjunto os não-terminais que derivam direta ou indiretamente a sentença vazia:
Ne = {A | A ∈ N e A +→ ε}

b) Construir o conjunto de regras P' como segue:

b1) incluir em P' todas as regras de P, com exceção daquelas da forma A → ε

b2) para cada ocorrência de um símbolo Ne do lado direito de alguma regra de P, incluir em P' mais uma regra, substituindo este símbolo por ε.

=> Isto é, para regra de P do tipo A → αBβ, B ∈Ne e α, β ∈V*
incluir em P' a regra A → α β

2. Transformação de uma GLC qualquer para uma GLC ε-Livre (cont')

Esta transformação sempre é possível e pode ser efetuada pelo seguinte algoritmo: (cont')

c) Se S ∈ Ne, adicionar a P' as regras S' → S e S' → ε, sendo que N' ficará igual a N ∪ S'.

=> Caso contrário trocar os nomes de S por S' e N por N'.

d) A nova gramática será definida por:

G' = (N', T, P', S')

2. Transformação de uma GLC qualquer para uma GLC ε-Livre (cont')

Transformar as GLC abaixo, definidas pelo respectivo conjunto de regras de produção P, para GLC ε-Livres.

G = ({S, B}, {a,b}, P, S)

P: S → a B
B → b B | ε

Solução:

Ne = {B}

a) Reunir em um conjunto os não-terminais que derivam direta ou indiretamente a sentença vazia:

Ne = {A | A ∈ N e A +→ ε}

2. Transformação de uma GLC qualquer para uma GLC ε-Livre (cont')

Transformar as GLC abaixo, definidas pelo respectivo conjunto de regras de produção P, para GLC ε-Livres.

G = ({S, B}, {a,b}, P, S)

P: S → a B
B → b B | ε

Solução:

Ne = {B}

P': S → a B
B → b B

b1) incluir em P' todas as regras de P, com exceção daquelas da forma A → ε

2. Transformação de uma GLC qualquer para uma GLC ε-Livre (cont')

Transformar as GLC abaixo, definidas pelo respectivo conjunto de regras de produção P, para GLC ε-Livres.

G = ({S, B}, {a,b}, P, S)

P: S → a B
B → b B | ε

Solução:

Ne = {B}

P': S → a B | a
B → b B | b

b2) para cada ocorrência de um símbolo Ne do lado direito de alguma regra de P, incluir em P' mais uma regra, substituindo este símbolo por ε.

=> Isto é, para regra de P do tipo A → αBβ, B ∈Ne e α, β ∈V*
incluir em P' a regra A → α β

2. Transformação de uma GLC qualquer para uma GLC ε-Livre (cont')

Transformar as GLC abaixo, definidas pelo respectivo conjunto de regras de produção P, para GLC ε-Livres.

G = ({S, B}, {a,b}, P, S)

P: S → a B
B → b B | ε

Solução:

Ne = {B}

P': S' → a B | a
B → b B | b

c) Se S ∈ Ne, adicionar a P' as regras S' → S e S' → ε, sendo que N' ficará igual a N ∪ S'.

=> Caso contrário trocar os nomes de S por S' e N por N'.

2. Transformação de uma GLC qualquer para uma GLC ε-Livre (cont')

Transformar as GLC abaixo, definidas pelo respectivo conjunto de regras de produção P, para GLC ε-Livres.

G = ({S, D, C}, {b,c,d,e}, P, S)

P: S → b D C e
D → d D | ε
C → c C | ε

2. Transformação de uma GLC qualquer para uma GLC ε-Livre (cont')

Transformar as GLC abaixo, definidas pelo respectivo conjunto de regras de produção P, para GLC ε-Livres.

G = ({S, D, C}, {b,c,d,e}, P, S)

P: S → b D C e
D → d D | ε
C → c C | ε

Solução:

Ne = {D, C}

a) Reunir em um conjunto os não-terminais que derivam direta ou indiretamente a sentença vazia:
Ne = {A | A ∈ N e A +→ ε}

2. Transformação de uma GLC qualquer para uma GLC ε-Livre (cont')

Transformar as GLC abaixo, definidas pelo respectivo conjunto de regras de produção P, para GLC ε-Livres.

G = ({S, D, C}, {b,c,d,e}, P, S)

P: S → b D C e
D → d D | ε
C → c C | ε

Solução:

Ne = {D, C}

P': S → b D C e
D → d D
C → c C

b1) incluir em P' todas as regras de P, com exceção daquelas da forma A → ε

2. Transformação de uma GLC qualquer para uma GLC ε-Livre (cont')

Transformar as GLC abaixo, definidas pelo respectivo conjunto de regras de produção P, para GLC ε-Livres.

G = ({S, D, C}, {b,c,d,e}, P, S)

P: S → b D C e
D → d D | ε
C → c C | ε

Solução:

Ne = {D, C}

P': S → b D C e | b C e | b D e | b e
D → d D | d
C → c C | c

b2) para cada ocorrência de um símbolo Ne do lado direito de alguma regra de P, incluir em P' mais uma regra, substituindo este símbolo por ε.

=> Isto é, para regra de P do tipo A → αBβ, B ∈Ne e α, β ∈V*
incluir em P' a regra A → α β

4. Linguagens livre de contexto

2. Transformação de uma GLC qualquer para uma GLC ε-Livre (cont')

Transformar as GLC abaixo, definidas pelo respectivo conjunto de regras de produção P, para GLC ε-Livres.

G = ({S}, {a}, P, S)

P: S → a S | ε

Solução:

Ne = {S}

P ': S' → S | ε
S → a S | a

Ling. Formais e AUtômatos

4. Linguagens livre de contexto

2. Transformação de uma GLC qualquer para uma GLC ε-Livre (cont')

Transformar as GLC abaixo, definidas pelo respectivo conjunto de regras de produção P, para GLC ε-Livres.

G = ({S, D, C}, {b,c,d,e}, P, S)

P: S → b D C e
D → d D | ε
C → c C | ε

Solução:

Ne = {D, C}

P': S' → b D C e | b C e | b D e | b e
D → d D | d
C → c C | c

c) Se S ∈ Ne, adicionar a P' as regras S' → S e S' → ε, sendo que N' ficará igual a N ∪ S'.

=> Caso contrário trocar os nomes de S por S' e N por N'.

Ling. Formais e AUtômatos

4. Linguagens livre de contexto

3. Remoção de Produções Simples (Unitárias)

Produções simples são produções da forma A → B onde A e B ∈ N.

Estas produções podem ser removidas de uma GLC através do seguinte algoritmo:

a) Transformar a GLC em uma GLC ε-livre, se necessário

b) Para todo não-terminal de N, construir um conjunto com os não-terminais que ele pode derivar, em um ou mais passos. Isto é, para todo A ∈ N, construir NA = { B | A *→ B }

c) Construir P' como segue:
se B →α ∈ P e não é uma produção simples, adicione a P' as produções:
A → α para todo A | B ∈ NA

d) A GLC equivalente, sem produções simples, será definida por:
G' = (N, T, P', S)

Ling. Formais e AUtômatos

3. Remoção de Produções Simples (Unitárias) – (cont')

Transformar as GLC abaixo em gramáticas equivalentes que não apresentem produções simples.

a) G = ({S, A}, {a,b}, P, S)

P: $S \rightarrow b\ S \mid A$
$A \rightarrow a\ A \mid a$

a) Transformar a GLC em uma GLC ε-livre, se necessário

3. Remoção de Produções Simples (Unitárias) – (cont')

Transformar as GLC abaixo em gramáticas equivalentes que não apresentem produções simples.

a) G = ({S, A}, {a,b}, P, S)

P: $S \rightarrow b\ S \mid A$
$A \rightarrow a\ A \mid a$

b) Para todo não-terminal de N, construir um conjunto com os não-terminais que ele pode derivar, em um ou mais passos. Isto é, para todo $A \in N$, construir $NA = \{ B \mid A \overset{*}{\rightarrow} B \}$

Ns = {A}
NA = {}

3. Remoção de Produções Simples (Unitárias) – (cont')

Transformar as GLC abaixo em gramáticas equivalentes que não apresentem produções simples.

a) G = ({S, A}, {a,b}, P, S)

P: $S \rightarrow b\ S \mid A$
$A \rightarrow a\ A \mid a$

Ns = {A}, NA = {}

c) Construir P' como segue:
se $B \rightarrow \alpha \in P$ e não é uma produção simples (PS), adicione a P' as produções:
$A \rightarrow \alpha$ para todo $A \mid B \in NA$

S->bS => ∈ P e não é PS => A -> bS se S ∈ NA
S->A => ∈ P e não é PS
A->aA => ∈ P e não é PS => S -> aA se A ∈ NS
A->a => ∈ P e não é PS => S -> a se A ∈ NS

3. Remoção de Produções Simples (Unitárias) – (cont')

Transformar as GLC abaixo em gramáticas equivalentes que não apresentem produções simples.

a) G = ({S, A}, {a,b}, P, S)

P: $S \rightarrow b\ S \mid A$
$A \rightarrow a\ A \mid a$

Solução:

Ns = {A}
NA = {}

P ': $S \rightarrow b\ S \mid a\ A \mid a$
$A \rightarrow a\ A \mid a$

3. Remoção de Produções Simples (Unitárias) – (cont')

Transformar as GLC abaixo em gramáticas equivalentes que não apresentem produções simples.

b) G= ({S, A, B}, {a,b,c}, P, S)

P: $S \rightarrow a\,S\,b \mid A$
$A \rightarrow a\,A \mid B$
$B \rightarrow b\,B\,c \mid b\,c$

a) Transformar a GLC em uma GLC ε-livre, se necessário

3. Remoção de Produções Simples (Unitárias) – (cont')

Transformar as GLC abaixo em gramáticas equivalentes que não apresentem produções simples.

b) G= ({S, A, B}, {a,b,c}, P, S)

P: $S \rightarrow a\,S\,b \mid A$
$A \rightarrow a\,A \mid B$
$B \rightarrow b\,B\,c \mid b\,c$

b) Para todo não-terminal de N, construir um conjunto com os não-terminais que ele pode derivar, em um ou mais passos. Isto é, para todo $A \in N$, construir $NA = \{ B \mid A \overset{*}{\rightarrow} B \}$

Ns = {A, B}
NA = {B}
NB = {}

3. Remoção de Produções Simples (Unitárias) – (cont')

Transformar as GLC abaixo em gramáticas equivalentes que não apresentem produções simples.

b) G= ({S, A, B}, {a,b,c}, P, S)

P: $S \rightarrow a\,S\,b \mid A$
$A \rightarrow a\,A \mid B$
$B \rightarrow b\,B\,c \mid b\,c$

Ns = {A, B}, NA = {B}, NB = {}

c) Construir P' como segue: se $B \rightarrow \alpha \in P$ e não é uma produção simples (PS), adicione a P' as produções: $A \rightarrow \alpha$ para todo $A \mid B \in NA$

S->aSb => ∈ P e não é PS => A -> aSb se S ∈ NA
S->aSb => ∈ P e não é PS => B -> aSb se S ∈ NB
S->A => ∈ P e não é PS

3. Remoção de Produções Simples (Unitárias) – (cont')

Transformar as GLC abaixo em gramáticas equivalentes que não apresentem produções simples.

b) G= ({S, A, B}, {a,b,c}, P, S)

P: $S \rightarrow a\,S\,b \mid A$
$A \rightarrow a\,A \mid B$
$B \rightarrow b\,B\,c \mid b\,c$

Ns = {A, B}, NA = {B}, NB = {}

c) Construir P' como segue: se $B \rightarrow \alpha \in P$ e não é uma produção simples (PS), adicione a P' as produções: $A \rightarrow \alpha$ para todo $A \mid B \in NA$

A->aA => ∈ P e não é PS => S -> aA se A ∈ NS
A->aA => ∈ P e não é PS => B ->aS se A ∈ NB
A->B => ∈ P e não é PS

3. Remoção de Produções Simples (Unitárias) – (cont')

Transformar as GLC abaixo em gramáticas equivalentes que não apresentem produções simples.

b) G= ({S, A, B}, {a,b,c}, P, S)

P: S → a S b | A
A → a A | B
B → b B c | b c

Ns = {A, B}, NA = {B}, NB = {}

c) Construir P' como segue: se B →α ∈ P e não é uma produção simples (PS), adicione a P' as produções: A → α para todo A | B ∈ NA

B->bBc => ∈ P e não é PS => S ->bBc se B ∈ NS
B->bBc => ∈ P e não é PS => A ->bBc se B ∈ NA
B->bc => ∈ P e não é PS => S ->bc se B ∈ NS
B->bc => ∈ P e não é PS => A ->bc se B ∈ NA

3. Remoção de Produções Simples (Unitárias) – (cont')

Transformar as GLC abaixo em gramáticas equivalentes que não apresentem produções simples.

b) G= ({S, A, B}, {a,b,c}, P, S)

P: S → a S b | A
A → a A | B
B → b B c | b c

Solução:
Ns = {A, B}
NA = {B}
NB = {}

P ': S → a S b | a A | b B c | b c
A → a A | b B c | b c
B → b B c | b c

4. Fatoração de GLC

Uma GLC está fatorada se ela é determinística, isto é, não possui produções cujo lado direito:

- inicie com o mesmo conjunto de símbolos ou

- com símbolos (terminais e não-terminais) que gerem seqüências que iniciem com o mesmo conjunto de símbolos.

Por exemplo, a gramática fatorada não deverá apresentar as seguintes regras:

A → a B | a C

pois as duas iniciam com o mesmo terminal a.

4. Fatoração de GLC (cont')

Outro exemplo de gramática não-fatorada é o seguinte:

S → A | B
A → a c
B → a b

4. Fatoração de GLC (cont')

Para fatorar uma GLC devemos alterar as produções envolvidas no não-determinismo da seguinte maneira:

a) as produções que apresentam não-determinismo direto, da forma
$A \rightarrow \alpha \beta \mid \alpha \delta$

serão substituídas por

$A \rightarrow \alpha A'$
$A' \rightarrow \beta \mid \delta$

sendo A' um novo não-terminal

b) O não-determinismo indireto é retirado fazendo, nas regras de produção, as derivações necessárias para torná-lo um determinismo direto, resolvido posteriormente como no item anterior.

4. Fatoração de GLC (cont')

Fatorar as GLC abaixo

a) G = ({S, A, B}, {a,b}, P, S)

P: $S \rightarrow a\,A \mid a\,B$
$A \rightarrow a\,A \mid a$
$B \rightarrow b$

Solução:

S-> aA | aB => S -> aS´
S´ -> A | B

A-> aA | a => A -> aA´
A´ -> a | ε

4. Fatoração de GLC (cont')

Fatorar as GLC abaixo

a) G = ({S, A, B}, {a,b}, P, S)

P: $S \rightarrow a\,A \mid a\,B$
$A \rightarrow a\,A \mid a$
$B \rightarrow b$

Solução:
P ': $S \rightarrow a\,S'$
$S' \rightarrow A \mid B$
$A \rightarrow a\,A'$
$A' \rightarrow A \mid \varepsilon$
$B \rightarrow b$

4. Fatoração de GLC (cont')

Fatorar as GLC abaixo

b) G = ({S, A}, {a,b}, P, S)

P: $S \rightarrow A\,b \mid a\,b \mid b\,a\,A$
$A \rightarrow a\,a\,b \mid b$

Solução:

P ': $S \rightarrow a\,a\,b\,b \mid b\,b \mid a\,b \mid b\,a\,A$
$A \rightarrow a\,a\,b \mid b$

P'': $S \rightarrow a\,S' \mid b\,S''$
$S' \rightarrow a\,b\,b \mid b$
$S'' \rightarrow b \mid a\,A$
$A \rightarrow a\,a\,b \mid b$

5. Eliminação de Recursão à Esquerda

- Uma gramática G = (N, T, P, S) tem **recursão à esquerda** se existe $A \in N$ tal que $A \overset{+}{\rightarrow} A\,\alpha$, $\alpha \in (N \cup T)^*$
- Uma gramática G = (N, T, P, S) tem **recursão à direita** se existe $A \in N$ tal que $A \overset{+}{\rightarrow} \alpha\,A$, $\alpha \in (N \cup T)^*$
- A **recursão** é dita **direta** se a derivação acima for em **um passo**,

  - G tem recursão **direta à esquerda** se existe produção $A \rightarrow A\,\alpha \in P$
  - G tem recursão **direta à direita** se existe produção $A \rightarrow \alpha\,A \in P$

5. Eliminação de Recursão à Esquerda – (cont')

Para eliminar a recursão direta à esquerda usa-se o seguinte algoritmo:

Seja G = (N, T, P, S) uma GLC **sem produções ε** e **sem produções do tipo $A \rightarrow A$**.

a) Para eliminar recursão direta à esquerda envolvendo um símbolo não-terminal A **dividimos inicialmente o conjunto das produções de P do tipo $A \rightarrow \alpha$ em subconjuntos**:

C1 = **conjunto das produções** $A \rightarrow \alpha$ que apresentam **recursão direta à esquerda**, ou seja, $A \rightarrow A\,\alpha \in P$ onde α $(N \cup T)^*$.

C2 = **conjunto das produções** $A \rightarrow \beta$ **que não apresentam recursão direta à esquerda**, ou seja, $A \rightarrow \beta \in P \mid \beta \neq A\,\alpha$ para qualquer α.

5. Eliminação de Recursão à Esquerda – (cont')

Para eliminar a recursão direta à esquerda usa-se o seguinte algoritmo: (cont')

Seja G = (N, T, P, S) uma GLC **sem produções ε** e **sem produções do tipo $A \rightarrow A$**.

b) **Cria-se um novo não-terminal B**

c) **Substitua as produções $A \rightarrow \alpha$** da gramática original de acordo com os seguintes passos:

c.1) para cada produção $A \rightarrow \beta i \in C2$ criar a produção $A \rightarrow \beta i\,B$

c.2) para cada produção $A \rightarrow A\alpha i \in C1$ criar as produções
$B \rightarrow \alpha i\,B$
$B \rightarrow \varepsilon$

5. Eliminação de Recursão à Esquerda – (cont')

Obs1: A recursão direta à esquerda foi transferida para recursão à direita.

Obs2: Se a gramática original tiver produções $A \rightarrow A$ poderão surgir produções ε após a execução do algoritmo. Além disso, podem aparecer produções simples após a execução do algoritmo.

Obs3: O algoritmo para eliminação de recursão direta à direita é análogo.

5. Eliminação de Recursão à Esquerda – (cont')

Elimine as recursões à esquerda da GLC abaixo

G = (N, T, P, S)

P: S → A a
A → S b | c A | a

Solução:

Derivar as recursões a esquerda:

P': S → A a
A → A a b | c A | a

5. Eliminação de Recursão à Esquerda – (cont')

Elimine as recursões à esquerda da GLC abaixo

G = (N, T, P, S)

P: S → A a
A → S b | c A | a

Solução:

P': S → A a
A → A a b | c A | a

a) Para eliminar recursão direta à esquerda envolvendo um símbolo não-terminal A dividimos inicialmente o conjunto das produções de P do tipo A → α em subconjuntos:

C1 = conjunto das produções A → α que apresentam recursão direta à esquerda, ou seja, A → A α ∈ P onde α (N ∪ T)*.

C1 : A → A a b

5. Eliminação de Recursão à Esquerda – (cont')

Elimine as recursões à esquerda da GLC abaixo

G = (N, T, P, S)

P: S → A a
A → S b | c A | a

Solução:

P': S → A a
A → A a b | c A | a

a) Para eliminar recursão direta à esquerda envolvendo um símbolo não-terminal A dividimos inicialmente o conjunto das produções de P do tipo A → α em subconjuntos:

C2 = conjunto das produções A → β que não apresentam recursão direta à esquerda, ou seja, A → β ∈ P | β ≠ A α para qualquer α.

C2 : A → c A | a

5. Eliminação de Recursão à Esquerda – (cont')

Elimine as recursões à esquerda da GLC abaixo

G = (N, T, P, S)

P: S → A a
A → S b | c A | a

Solução:

P': S → A a
A → A a b | c A | a

C1 : A → A a b
C2 : A → c A | a

b) Cria-se um novo não-terminal B

5. Eliminação de Recursão à Esquerda – (cont')

Elimine as recursões à esquerda da GLC abaixo

G = (N, T, P, S)

P: $S \rightarrow A\ a$
$A \rightarrow S\ b \mid c\ A \mid a$

Solução:

P': $S \rightarrow A\ a$
$A \rightarrow A\ a\ b \mid c\ A \mid a$

C1 : $A \rightarrow A\ a\ b$
C2 : $A \rightarrow c\ A \mid a$

c) Substitua as produções $A \rightarrow \alpha$ da gramática original de acordo com os seguintes passos:

c.1) para cada produção $A \rightarrow \beta i \in C2$ criar a produção $A \rightarrow \beta i\ B$

$A \rightarrow c\ A \Rightarrow A \rightarrow c\ A\ A'$
$A \rightarrow a \Rightarrow A \rightarrow a\ A'$

4. Linguagens livre de contexto

5. Eliminação de Recursão à Esquerda – (cont')

Elimine as recursões à esquerda da GLC abaixo

G = (N, T, P, S)

P: $S \rightarrow A\ a$
$A \rightarrow S\ b \mid c\ A \mid a$

Solução:

P': $S \rightarrow A\ a$
$A \rightarrow A\ a\ b \mid c\ A \mid a$

C1 : $A \rightarrow A\ a\ b$
C2 : $A \rightarrow c\ A \mid a$

c) Substitua as produções $A \rightarrow \alpha$ da gramática original de acordo com os seguintes passos:

c.2) para cada produção $A \rightarrow A\alpha i \in C1$ criar as produções
$B \rightarrow \alpha i\ B$
$B \rightarrow \varepsilon$

$A \rightarrow A\ a\ b \Rightarrow A' \rightarrow a\ b\ A' \mid \varepsilon$

Ling. Formais e AUtômatos

4. Linguagens livre de contexto

5. Eliminação de Recursão à Esquerda – (cont')

Elimine as recursões à esquerda da GLC abaixo

G = (N, T, P, S)

P: $S \rightarrow A\ a$
$A \rightarrow S\ b \mid c\ A \mid a$

Solução:

P': $S \rightarrow A\ a$
$A \rightarrow A\ a\ b \mid c\ A \mid a$

P'': $S \rightarrow A\ a$
$A \rightarrow c\ A\ A' \mid a\ A'$
$A' \rightarrow a\ b\ A' \mid \varepsilon$

Ling. Formais e AUtômatos

4. Linguagens livre de contexto

Forma Normal de Chomsky (CNF)

Uma GLC está na Forma Normal de Chomsky se ela é ε-livre e apresenta todas as produções da forma

$A \rightarrow B\ C$
ou $A \rightarrow a$

$A, B, C \in N$ e $a \in T$.

➢ Prova-se que toda Linguagem Livre de Contexto ε-livre pode ser gerada por uma GLC na Forma Normal de Chomsky.

Ling. Formais e AUtômatos

Forma Normal de Chomsky (CNF) – (cont')

Uma GLC ε-livre G = (N, T, P, S) pode ser colocada na CNF através do seguinte algoritmo:

a) obter G' = (N', T, P', S) a partir de G, removendo de G as suas produções simples, de modo que L(G') = L(G)

b) nas regras de G' cujo lado direito apresenta mais de um termo, substituir cada terminal (por exemplo, a ∈ T) por um novo não-terminal (Aa), incluindo para cada um destes novos não-terminais uma nova regra, Aa → a, resultando G'' = (N'', T, P'', S)

c) substituir cada regra do tipo A → B1 B2 ... Bm, m >= 3 onde A, B1, B2, ..., Bm são não-terminais, pelo conjunto de regras

A → B1 B'1
B'1 → B2 B'2
...
B'm-2 → Bm-1 Bm

onde B'1, B'2, ..., Bm-2 são novos não-terminais

d) a gramática gerada está na CNF e é dada por G''' = (N''', T, P''', S)

Forma Normal de Chomsky (CNF) – (cont')

Dada a GLC abaixo, ache a gramática equivalente na CNF

G = ({S, A, B}, {a, b}, P, S)

P: S → A | A B A
A → a A | a
B → b B | b

Solução

a) obter G' = (N', T, P', S) a partir de G, removendo de G as suas produções simples, de modo que L(G') = L(G)

P': S → a A | a | A B A
A → a A | a
B → b B | b

Forma Normal de Chomsky (CNF) – (cont')

Dada a GLC abaixo, ache a gramática equivalente na CNF

G = ({S, A, B}, {a, b}, P, S)

P: S → A | A B A
A → a A | a
B → b B | b

Solução

b) nas regras de G' cujo lado direito apresenta mais de um termo, substituir cada terminal (por exemplo, a ∈ T) por um novo não-terminal (Aa), incluindo para cada um destes novos não-terminais uma nova regra, Aa → a, resultando G'' = (N'', T, P'', S)

P':		P'':	
	S → a A | a | A B A		S → Aa A | a | A B A
	A → a A | a		A → Aa A | a
	B → b B | b		B → Ab B | b
			Aa → a
			Ab → b

Forma Normal de Chomsky (CNF) – (cont')

Dada a GLC abaixo, ache a gramática equivalente na CNF

G = ({S, A, B}, {a, b}, P, S)

P: S → A | A B A
A → a A | a
B → b B | b

Solução

c) substituir cada regra do tipo A → B1 B2 ... Bm, m >= 3 onde A, B1, B2, ..., Bm são não-terminais, pelo conjunto de regras

A → B1 B'1, B'1 → B2 B'2, .., B'm-2 → Bm-1 Bm

P'':		P''':	
	S → Aa A | a | A B A		S → Aa A | a | A B'
	A → Aa A | a		B' → B A
	B → Ab B | b		A → Aa A | a
	Aa → a		B → Ab B | b
	Ab → b		Aa → a
			Ab → b

Forma Normal de Greibach (GNF)

Uma GLC está na Forma Normal de Greibach se ela é ε-livre e apresenta todas as produções na forma:

$A \rightarrow a\ \alpha$

onde $A \in N$, $a \in T$ e $\alpha \in N^*$.

➢ Prova-se que toda a Linguagem Livre de Contexto ε-livre pode ser gerada por uma GLC na Forma Normal de Greibach.

Forma Normal de Greibach (GNF) – (cont')

Para achar a gramática equivalente a G = (N, T, P, S), na GNF, deve-se seguir os seguintes passos:

a) achar G' = (N', T, P', S) tal que L(G') = L(G) e que G' esteja na CNF

b) ordenar e renomear os não-terminais de G' em uma ordem quaisquer - por exemplo: N' = { A1, A2, ..., Am }

c) modificar as regras de P' de modo a que, se $Ai \rightarrow Aj\gamma$ é uma regra de P', então $j > i$

d) a gramática obtida do passo anterior, G'', apresentará todas as regras de Am com o lado direito iniciando por um terminal

=> através de substituições sucessivas dos primeiros termos das regras Ai anteriores, coloca-se estas também nessa forma

e) se no item *c* tiverem sido incluídos novos não-terminais Bi (para retirar recursões à esquerda), fazer também para as regras correspondentes a estes, as devidas substituições dos primeiros termos (que serão sempre terminais ou Ai)

f) a gramática final, G''', está na GNF

Forma Normal de Greibach (GNF) – (cont')

Coloque a GLC abaixo na GNF

G = ({S,A}, {a,b}, P, S)

P: $S \rightarrow A\ S \mid a$
$A \rightarrow S\ A \mid b$

Solução

a) G já está na CNF

Forma Normal de Greibach (GNF) – (cont')

Coloque a GLC abaixo na GNF

G = ({S,A}, {a,b}, P, S)

P: $S \rightarrow A\ S \mid a$
$A \rightarrow S\ A \mid b$

Solução

b) Renomear os não-terminais (ordem):

S = A1 e A = A2

P': $A1 \rightarrow A2\ A1 \mid a$
$A2 \rightarrow A1\ A2 \mid b$

Forma Normal de Greibach (GNF) – (cont')

Coloque a GLC abaixo na GNF

G = ({S,A}, {a,b}, P, S)

P: S → A S | a
A → S A | b

Solução

c) modificar as regras de P' de modo a que, se Ai → Ajγ é uma regra de P', então j > i

=> a única regra a modificar é A2 → A1 A2

substituindo A1 nessa regra pelas suas regras temos

A2 → A2 A1 A2 |a A2 | b

para retirar a recursividade à esquerda da 1ª regra, introduzimos um novo não-terminal B2 resultando para G'':

P'': A1 → A2 A1 | a
A2 → a A2 B2 |b B2 |a A2 | b
B2 → A1 A2 B2 | A1 A2 ???

S = A1 e A = A2

P': A1 → A2 A1 | a
A2 → A1 A2 | b

4. Linguagens livre de contexto

Forma Normal de Greibach (GNF) – (cont')

Coloque a GLC abaixo na GNF

G = ({S,A}, {a,b}, P, S)

P: S → A S | a
A → S A | b

Solução

d) a gramática obtida do passo anterior, G'', apresentará todas as regras de Am com o lado direito iniciando por um terminal

=> através de substituições sucessivas dos primeiros termos das regras Ai anteriores, coloca-se estas também nessa forma

P''': A2 → a A2 B2 | b B2|a A2 | b
A1 → a A2 B2 A1 | b B2 A1 | a A2 A1| b A1 | a
B2 → a A2 B2 A1 A2 B2 | b B2 A1 A2 B2 |
a A2 A1 A2 B2 | b A1 A2 B2 |a A2 B2 |
a A2 B2 A1 A2 | b B2 A1 A2 |a A2 A1 A2 |
b A1 A2 | a A2

4. Linguagens livre de contexto

Outros Tipos de GLC

a) Gramática reduzida

É uma GLC que satisfaz às seguintes condições:

1 - L(G) ≠ ∅ e
2 - se A → α ∈ P então A ≠ α e G não possui símbolos inúteis

b) Gramática sem ciclos

É uma GLC que não possui derivação da forma:

A +→ A para A ∈ N

4. Linguagens livre de contexto

Outros Tipos de GLC (cont')

c) Gramática Própria

É uma GLC que:

1 - não possui ciclos
2 - é ε-livre
3 - não possui símbolos inúteis

d) Gramática de Operadores

É uma GLC que não possui produções da forma:

A → ... B C ... onde A, B, C ∈ N

ou seja, é uma GLC na qual não aparecem dois não-terminais juntos em nenhuma regra de produção.

Outros Tipos de GLC (cont')

e) Gramática Linear

É uma GLC na qual todas as produções se apresentam na forma:

$A \rightarrow x\ B\ w$
ou $A \rightarrow x$ onde $A, B \in N$ e $x, w \in T^*$

Principais Aplicações de GLC

1 – Especificação de linguagens de programação;

2 – Formalização de parsing / implementação de parser's;

3 – Esquemas de tradução dirigidos pela sintaxe

4 – Processamento de string's, de modo geral.

Linguagens Formais e Autômatos

Professores: Dr. Hermes Senger