

Importante: não imprima esse documento inteiro, pois haverá atualizações ao longo do período letivo.

Notas de Aula

Este material é apenas um roteiro contendo os tópicos na ordem que são estudados em sala de aula com uma breve explicação dos pontos mais importantes. Portanto, é necessário que você complete seus estudos lendo a bibliografia indicada. Mais importante ainda é a resolução dos exercícios propostos que estão organizados em outro documento.

Essas notas de aula dão um foco para estudos de casos aplicados às linguagens de programação e visam a construção de conhecimentos que são pré-requisitos para a disciplina de Compiladores.

Sobre a ementa, critérios avaliativos, lista de exercícios propostos, materiais de apoio, ferramentas para interação etc, consulte o AVA (ambiente virtual de aprendizagem) da disciplina: <https://ava.ufes.br/>

Tópicos: (clique no tópico desejado para navegar)

1. Bibliografia recomendada
2. Hierarquia de Chomsky, seus elementos, aplicações e exemplos
3. Exemplo de gramática.
4. Estudo de caso 1: número inteiros
5. Conceitos importantes
6. Linguagens Regulares
7. Gramática Regular (GR)
8. Estudo de caso 1: número inteiros (resolvido com gramática regular)
9. Expressão Regular
10. Estudo de caso 1: número inteiros
11. Estudo de caso 2: números reais
12. Autômato Finito Determinístico (AFD)
13. Exemplos de AFD, AFND e AF λ
14. Estudo de caso 1: número inteiros
15. Estudo de caso 2: números reais
16. GR x AFD x ER
17. Algoritmo para converter AFD em GR (sem conflitos à esquerda)
18. Estudo de caso 2: números reais (gramática regular sem conflitos)
19. Estudo de caso 3: nomes de variáveis
20. Estudo de caso 4: números binários sem sinal e com 3 dígitos
21. Estudo de caso 5: números reais com possibilidade de notação científica
22. Estudo de caso 6: números hexadecimais em Java, de 1 a 4 dígitos
23. Estudo de caso 7: números octais em Java
24. Conversão de AFND e AF λ em AFD
25. Minimização de um Autômato Finito
26. Busca de texto: aplicação para um AFND
27. Uso do simulador JFLAP para Linguagens Regulares

28. Máquina de Mealy
29. Máquina de Moore
30. Exemplo de Máquina de Moore para classificar números inteiros sem sinal
31. Exemplo de Máquina de Moore para classificar os operadores relacionais do C/Java
32. Máquina de Moore para construção de Analisador Léxico
33. Gramática Livre de Contexto (GLC)
34. Exemplos de linguagens livres de contexto
35. Uso do simulador JFLAP para Linguagens Livres de Contexto
36. Lema do Bombeamento para Linguagens Regulares
37. Aplicação do Lema do Bombeamento para a Prova de Não Regularidade de Linguagens
38. Aplicações de Linguagens Livres de Contexto
39. Gramáticas ambíguas e o problema do else flutuante
40. Análise Preditiva
41. Fatoração de gramáticas
42. Algoritmo para fatorar gramáticas
43. Recursividade à esquerda em gramáticas
44. Algoritmo para eliminação da recursividade à esquerda
45. Exemplo de Análise Preditiva
46. Análise Preditiva na construção de Analisadores Sintáticos
47. Forma Normal de Chomsky
48. Aplicação da Forma Normal de Chomsky: Algoritmo de Cocke-Younger-Kasami
49. Forma Normal de Greibach
50. Formato BNF para gramáticas livres de contexto
51. Formato EBNF para gramáticas livres de contexto
52. Autômato com Pilha
53. Aplicação da Forma Normal de Greibach: construção de Autômato com Pilha
54. Lema do Bombeamento para Linguagens Livres de Contexto
55. Aplicação do Lema do Bombeamento para Linguagens Livres e Contexto
56. Linguagens Sensíveis ao Contexto
57. Máquina de Turing
58. Gramática Sensível ao Contexto
59. Linguagens Enumeráveis Recursivamente
60. Gramática Irrestrita

APÊNDICE A: Exemplo de uso do Graphviz (Graph Visualization Software)

APÊNDICE B: Exemplo de um Analisador Léxico construído através da Análise Preditiva

APÊNDICE C: Exemplo de um Analisador Sintático construído através da Análise Preditiva

1. Bibliografia recomendada:

MENEZES, Paulo Fernando Blauth. **Linguagens Formais e Autômatos**. 5 ed. Porto Alegre: Sagra Luzzatto, 2008.

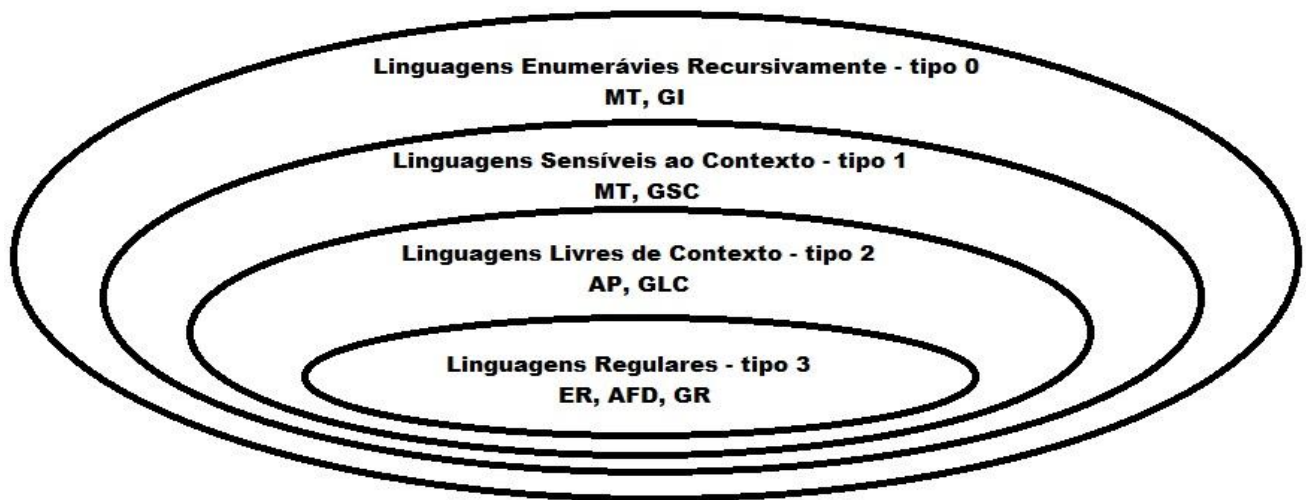
HOPCROFT, John E.; ULLMAN, Jeffrey D.; MOTWANI, Rajeev. **Introdução à teoria de autômatos, linguagens e computação**. 2. ed. Rio de Janeiro: Elsevier, 2002.

ROSA, João Luis Garcia. **Linguagens formais e autômatos**. Rio de Janeiro: LTC, 2010.

SUDKAMP, Thomas A. **Languages and machines: an introduction to the theory of computer science**. 2. ed. Massachusetts: Addison-Wesley Publishing Company, Inc., 1997.

2. Hierarquia de Chomsky, seus elementos, aplicações e exemplos:

O linguista Noam Chomsky criou uma hierarquia de linguagens com 4 níveis, sendo os dois últimos muito utilizados na descrição de linguagens de programação e na implementação de interpretadores e compiladores.



- **Linguagem Regular (tipo 3)**

Formalismo denotacional: expressão regular.

Formalismo gerador (ou axiomático): gramática regular.

Formalismo reconhecedor: autômato finito determinístico.

Aplicações: representação e geração de tokens; construção de analisadores léxicos (scanner).

Exemplos de linguagens: nomes de variáveis de uma linguagem, números inteiros sem sinal, números reais, palavras reservadas de uma linguagem, ponto-e-vírgula, operadores relacionais.

- **Linguagem Livre de Contexto (tipo 2)**

Formalismo gerador: gramática livre de contexto.

Formalismo reconhecedor: autômato com pilha.

Aplicações: representação e geração de programas da maioria das linguagens de programação conhecidas (C++, Pascal, Java, etc.); construção de analisadores sintáticos (parser).

Exemplos de linguagens: Linguagem C, Linguagem SQL.

- **Linguagem Sensível ao Contexto (tipo 1)**

Formalismo gerador: gramática sensível ao contexto.

Formalismo reconhecedor: Máquina de Turing com memória limitada.

Aplicações: auxílio na representação de linguagens naturais.

- **Linguagem Enumerável Recursivamente (tipo 0)**

Formalismo gerador: gramática irrestrita, também conhecida como gramática com estrutura de frase.

Formalismo reconhecedor: Máquina de Turing.

Aplicações: auxílio na representação de linguagens naturais.

3. Exemplo de gramática.

Para ilustrar o conceito de gramática que será dado logo adiante, segue uma gramática divertida e geradora de frases em inglês. Disponível em SUDKAMP, 1997.

```
<sentence> -> <noun-phrase><verb-phrase> | <noun-phrase><verb><direct-object-phrase>
<noun-phrase> -> <adjective-list><proper-noun> | <determiner><adjective-list><common-noun>
<proper-noun> -> John | Jill
<common-noun> -> car | hamburger
<determiner> -> a | the
<verb-phrase> -> <verb><adverb> | <verb>
<verb> -> drives | eats
<adverb> -> slowly | frequently
<adjective-list> -> <adjective><adjective-list> | λ
<adjective> -> big | juicy | brown
<direct-object-phrase> -> <determiner><adjective-list><common-noun>
```

Derivação para a string: “big John drives slowly”

```
<sentence> => <noun-phrase><verb-phrase>
=> <adjective-list><proper-noun><verb-phrase>
=> <adjective-list><proper-noun><verb><adverb>
=> <adjective><adjective-list><proper-noun><verb><adverb>
=> <adjective><proper-noun><verb><adverb>
=> big <proper-noun><verb><adverb>
=> big John <verb><adverb>
=> big John drives <adverb>
=> big John drives slowly
```

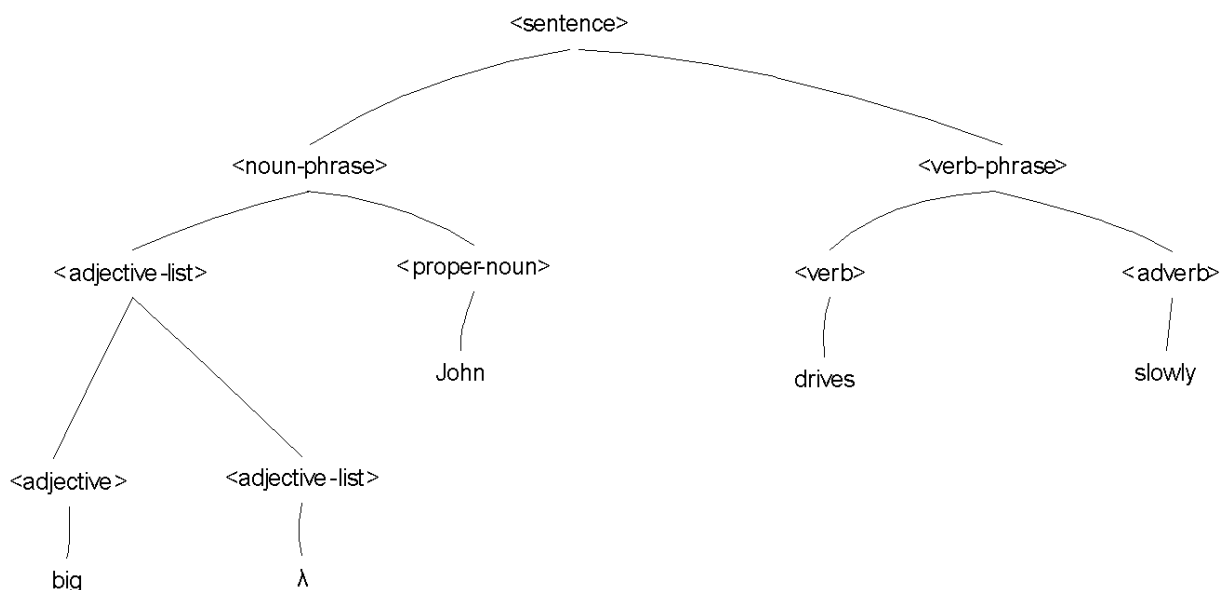
Derivação com notação resumida:

```
<sentence> => <noun-phrase><verb-phrase>
=>2 <adjective-list><proper-noun><verb><adverb>
=>2 <adjective><proper-noun><verb><adverb>
=>4 big John drives slowly
```

ou ainda:

```
<sentence> =>* big John drives slowly
```

Árvore de derivação:



Exemplo de outras duas strings geradas pela gramática:

“big big John eats frequently”

“Jill drives a big car”

Obs.: como a gramática é livre de contexto e está no nível 2 da Hierarquia de Chomsky, ela não tem poder de representar de forma completa uma linguagem natural que é do nível zero, portanto ela também gera frases semanticamente incorretas, como por exemplo:

“the big big car eats a brown car”

Parte dos problemas semânticos podem ser resolvidos computacionalmente, tal como será visto na disciplina de Compiladores no próximo semestre.

A **representação formal** de uma gramática é dada por uma quádrupla de elementos $G = (V, T, P, S)$
Onde:

V = conjunto de símbolos variáveis (ou não terminais)

T = conjunto de símbolos terminais

P = conjunto de regras de produção

S = símbolo que deve iniciar todas as derivações

Sendo assim a gramática anterior é definida:

$G = (V, T, P, \text{<sentence>})$

$V = \{ \text{<sentence>}, \text{<noun-phrase>}, \text{<verb-phrase>}, \text{<verb>}, \text{<direct-object-phrase>}, \text{<proper-noun>}, \text{<determiner>}, \text{<common-noun>}, \text{<adverb>}, \text{<adjective-list>}, \text{<adjective>}, \text{<direct-object-phrase>} \}$

$T = \{ a, the, John, Jill, hamburger, car, drives, eats, slowly, frequently, big, juicy, brown \}$

$P = \{ \begin{array}{l} \text{<sentence>} \rightarrow \text{<noun-phrase>}<\text{verb-phrase}> \mid \text{<noun-phrase>}<\text{verb>}<\text{direct-object-phrase>} \\ \text{<noun-phrase>} \rightarrow \text{<adjective-list>}<\text{proper-noun>} \mid \text{<determiner>}<\text{adjective-list>}<\text{common-noun>} \\ \text{<proper-noun>} \rightarrow John \mid Jill \\ \text{<common-noun>} \rightarrow car \mid hamburger \\ \text{<determiner>} \rightarrow a \mid the \\ \text{<verb-phrase>} \rightarrow \text{<verb>}<\text{adverb>} \mid \text{<verb>} \\ \text{<verb>} \rightarrow drives \mid eats \\ \text{<adverb>} \rightarrow slowly \mid frequently \\ \text{<adjective-list>} \rightarrow \text{<adjective>}<\text{adjective-list>} \mid \lambda \end{array} \}$

```

    <adjective> -> big | juicy | brown
    <direct-object-phrase> -> <determiner><adjective-list><common-noun>
}

```

4. Estudo de caso 1: número inteiros

Gramática para gerar os números inteiros (gramática livre de contexto):

a) sem sinal

```

G = (V, T, P, S)
V = { S, D }
T = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 }
P = { S -> DS | D
      D -> 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
    }

```

b) com possibilidade de sinal:

```

G = (V, T, P, S)
V = { S, A, B, C }
T = { +, -, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 }
P = { S -> AB
      A -> + | - | λ
      B -> DB | B
      D -> 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
    }

```

Observe a modularização feita na variável S, passando a responsabilidade do sinal para A e a responsabilidade do número inteiro sem sinal para B.

5. Conceitos importantes

- **Alfabeto:** é um conjunto finito de símbolos. Normalmente representado por Σ
- String, **palavra**, cadeia de caracteres ou sentença: é uma sequência finita de símbolos do alfabeto justapostos.
- **Palavra vazia:** é a ausência de símbolos. Normalmente é representada por λ ou ϵ
- **Tamanho** de uma string: é a qtd de símbolos que compõe a string. $|w|$ representa o tamanho de w.
- **Prefixo, sufixo e subpalavra** (ou substring): parte inicial, final ou qualquer de uma palavra.
- **Concatenação** de palavras: é a junção das palavras para formação de uma palavra.
- **Concatenação sucessiva:** suponha w uma palavra de uma alfabeto e \underline{n} a quantidade de concatenações sucessivas, então: $w^n = w^{n-1} w$. Quando $n=0$: $w^0 = \lambda$

- **Estrela de Kleene:** seja V um conjunto, então V^* o conjunto de todos elementos que podem ser formados através da concatenação de zero ou mais elementos de V .
- **Linguagem formal:** é um conjunto de palavras sobre um alfabeto. Por exemplo $L \subseteq \Sigma^*$, isto é, a linguagem L é formada pelas palavras que podem ser formadas pelo alfabeto Σ , inclusive o λ .
- Duas **gramáticas** são **equivalentes** se a linguagem gerada de uma é igual a outra.

6. Linguagens Regulares

Tem como aplicação o reconhecimento de tokens num projeto de compiladores.

Possui os seguintes formalismos (serão estudados com detalhes nas próximas seções):

- **Gerador:** gramática regular (GR)
- **Denotacional:** expressão regular (ER)
- **Reconhecedor:** autômato finito determinístico (AFD)

Propriedades:

- Se L é uma linguagem gerada por uma gramática regular, ou representada por uma expressão regular, ou reconhecida por um autômato finito determinístico, então L é uma linguagem regular.
- Se L é uma linguagem regular então existe uma gramática regular que gera L , existe também uma expressão regular que representa L , e existe um autômato finito determinístico que reconhece L .

7. Gramática Regular (GR)

Também definida por uma quádrupla (a mesma apresentada anteriormente), sendo que todas as regras de produção devem se encaixar num dos dois seguintes formatos:

$\begin{array}{l} A \rightarrow \alpha B \\ A \rightarrow \alpha \end{array}$

Onde: $A, B \in V$ (conjunto de variáveis)
 $\alpha \in T^*$

Pelo fato da variável estar posicionada à direita, na regra $A \rightarrow \alpha B$, essa gramática regular é chamada de **linear à direita**. De forma equivalente, as gramáticas regulares também podem ser escritas na forma **linear à esquerda**:

$\begin{array}{l} A \rightarrow B\alpha \\ A \rightarrow \alpha \end{array}$
--

Ainda de forma equivalente, quando acontece a exigência de: $\alpha \in T$, isto é, apenas um terminal na regra, a gramática regular recebe o nome de **linear unitária à esquerda**, ou **linear unitária à direita**, conforme a posição do α .

Obs.:

- As quatro formas de gramática regulares apresentadas são **equivalentes** entre si, isto é, a partir de uma das formas é possível gerar as outras três.
- Gramática **regular** e gramática **linear** são termos equivalentes.

8. Estudo de caso 1: número inteiros (resolvido com gramática regular)

Gramática transformada em regular:

$$\begin{aligned}
 G &= (V, T, P, S) \\
 V &= \{S, A, B\} \\
 T &= \{+, -, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9\} \\
 P &= \{ S \rightarrow +A \mid -A \mid A \\
 &\quad A \rightarrow 0A \mid 1A \mid 2A \mid 3A \mid 4A \mid 5A \mid 6A \mid 7A \mid 8A \mid 9A \mid B \\
 &\quad B \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \\
 &\quad \}
 \end{aligned}$$

Outra solução, com menos símbolos variáveis:

$$\begin{aligned}
 G &= (V, T, P, S) \\
 V &= \{S, A\} \\
 T &= \{+, -, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9\} \\
 P &= \{ S \rightarrow +A \mid -A \mid A \\
 &\quad A \rightarrow 0A \mid 1A \mid 2A \mid 3A \mid 4A \mid 5A \mid 6A \mid 7A \mid 8A \mid 9A \mid 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \\
 &\quad \}
 \end{aligned}$$

9. Expressão Regular

É um formalismo denotacional para linguagens regulares que facilita a comunicação homem x homem, e principalmente entre homem x máquina.

Definição: uma expressão regular (ER) sobre um alfabeto Σ é indutivamente definida como:

- $\{\}$ é uma ER e representa a linguagem vazia;
- λ é uma ER e representa a linguagem contendo a string vazia, isto é, $\{\lambda\}$
- $x \in \Sigma$ é uma ER e representa a linguagem $\{x\}$
- se r e s são ER e representam as linguagens R e S respectivamente, então:
 - $(r \mid s)$ é ER e representa a linguagem $R \cup S$
Obs.: algumas literaturas usam: $r + s$, outras usam: $r \cup s$
 - (rs) é ER e representa a linguagem $RS = \{uv \mid u \in R \text{ e } v \in S\}$
 - (r^*) é ER e representa a linguagem R^*

Precedência: por convenção assume-se a seguinte ordem de execução:

- 1º) parênteses: $()$
- 2º) concatenação sucessiva: $*$
- 3º) concatenação
- 4º) união: $|$

Conjunto de meta-símbolos extendidos usados nas ER e normalmente encontrados na literatura:

Suponha o alfabeto $\Sigma = \{a,b\}$

a^+	= um ou vários a's concatenados $\{a, aa, aaa, aaaa, \dots\}$
$a b$	= a ou b $\{a, b\}$ Obs.: algumas literaturas usam a notação: $a \cup b$, ou ainda: $a + b$
a^*	= zero, um ou vários a's concatenados $\{\lambda, a, aa, aaa, aaaa, \dots\}$
$a?$	= $\{\lambda, a\}$

Exemplos:

Observe exemplos de ER sobre o alfabeto $\Sigma = \{a,b\}$, e suas respectivas linguagens.

ER	Descrição da linguagem, ou exemplos de strings
a^*	$\{\lambda, a, aa, aaa, aaaa, \dots\}$
a^+	$\{a, aa, aaa, aaaa, \dots\}$
$a?$	$\{\lambda, a\}$
$a b$	$\{a, b\}$
ab	$\{ab\}$
$(ab)^*$	$\{\lambda, ab, abab, ababab, \dots\}$ Obs.: ba, aab, aabb não pertencem à linguagem
$b^+ a^*$	$\{\lambda, b, bb, bbb, a, aa, aaa, \dots\}$ Obs.: ba, bba, ab não pertencem à linguagem
$(a b)^*$	todas as strings, inclusive λ
$(a b)^+$	todas as strings, exceto λ
$a(a b)^*$	todas as strings iniciadas por 'a'
$(a b)^*ba$	todas as strings terminadas por 'ba'
$((a b)(a b))^*$	todas as strings com tamanho par
$a(a b)^*b$	strings iniciadas por 'a' e terminadas por 'b'
$bb(a b)^*aaa$	strings iniciadas por 'bb' e terminadas por 'aaa'

10. Estudo de caso 1: número inteiros

ER: $(+|-|\lambda)(0|\dots|9)^+$

11. Estudo de caso 2: números reais

Exemplos de strings da linguagem: $L = \{2.5, +2.5, -2.5, +2, -2, +0, 0, .5, -.5, \dots\}$

ER: $(+|-|\lambda)(0|\dots|9)^*(.)?(0|\dots|9)^+$

12. Autômato Finito Determinístico (AFD)

É um formalismo reconhecedor que tem como objetivo aceitar ou rejeitar strings dadas como entrada de uma determinada linguagem regular. Ou seja é uma máquina abstrata capaz de reconhecer a pertinência de

strings para uma dada linguagem representada pelo autômato através de um grafo orientado.

Definição:

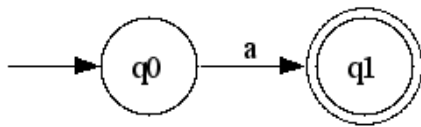
Um autômato finito determinístico (AFD) é uma 5-upla $M = (\Sigma, Q, \delta, q_0, F)$ onde:

- Σ é o alfabeto dos símbolos de entrada
- Q é o conjunto dos estados possíveis
- δ é a função transição, $\delta: Q \times \Sigma \rightarrow Q$
- q_0 é o estado inicial
- F é o conjunto de estados finais. Deve possuir ao menos um elemento.

Representação do AFD por intermédio de um grafo:

O estado inicial é indicado por uma seta sem origem, e os estados finais são representados por dois círculos concêntricos, cada um.

Exemplo 1: AFD para aceitar a linguagem regular formada apenas por 'a'.



$$M = (\Sigma, Q, \delta, q_0, F)$$

$$\Sigma = \{a\}$$

$$Q = \{q_0, q_1\}$$

δ - função transição:

δ	a
q0	q1
q1	-

$$F = \{q_1\}$$

Exemplo 2: AFD para aceitar a linguagem regular denotada por $(ab)^*$



$$M = (\Sigma, Q, \delta, q_0, F)$$

$$\Sigma = \{a, b\}$$

$$Q = \{q_0, q_1\}$$

δ - função transição:

δ	a	b
q0	q1	-
q1	-	q0

$$F = \{q_0\}$$

Algumas observações importantes sobre autômatos finitos:

- em cada estado pode ter no máximo uma transição de saída por símbolo terminal, senão ele é classificado como **Autômato Finito Não Determinístico (AFND)**
- havendo uma transição com λ o autômato é classificado como **Autômato com Movimentos Vazios (AF λ)**
- todo **AFND** e **AF λ** pode ser convertido em **AFD**.

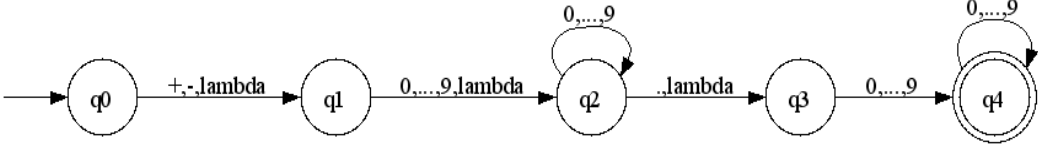
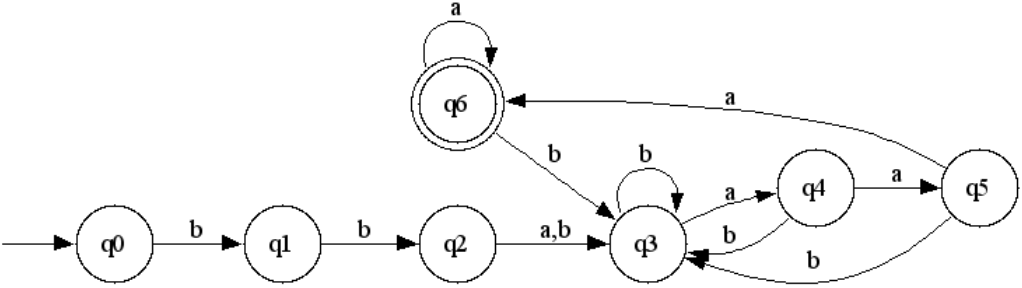
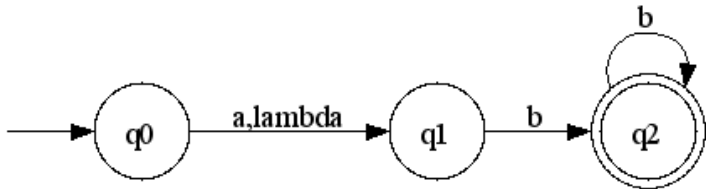
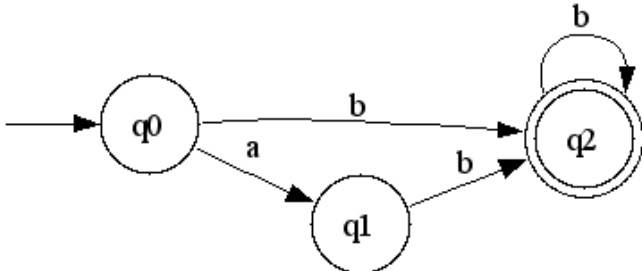
13. Exemplos de AFD, AFND e AF λ

Observe os exemplos (já usados em ER) a a distinção entre AFD, AFND, AF λ .

ER	AFD, AFND, AF λ
a^*	<p>A finite automaton with a single state q_0, which is both the start state (indicated by an incoming arrow) and the final state (indicated by a double circle). There is a self-loop transition on q_0 labeled 'a'.</p>
a^+	<p>A finite automaton with two states: q_0 and q_1. q_0 is the start state. q_1 is the final state. There is a transition from q_0 to q_1 labeled 'a', and a self-loop transition on q_1 labeled 'a'.</p>
$a?$	<p>A finite automaton with two states: q_0 and q_1. Both q_0 and q_1 are final states. q_0 is the start state. There is a transition from q_0 to q_1 labeled 'a'.</p>
$a b$	<p>A finite automaton with three states: q_0, q_1, and q_2. q_0 is the start state. Both q_1 and q_2 are final states. There are two transitions from q_0: one labeled 'a' to q_1 and one labeled 'b' to q_2.</p>
ab	<p>A finite automaton with three states: q_0, q_1, and q_2. q_0 is the start state. q_2 is the final state. There is a transition from q_0 to q_1 labeled 'a', and a transition from q_1 to q_2 labeled 'b'.</p>

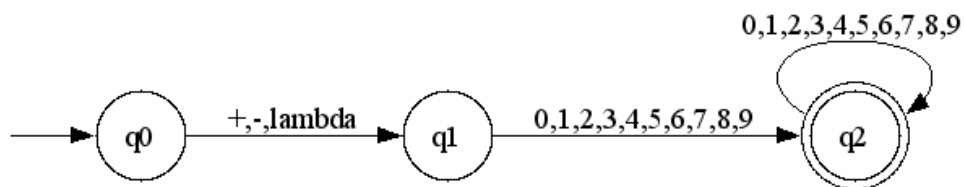
$(ab)^*$	<pre> graph LR start(()) --> q0((q0)) q0 -- a --> q1((q1)) q1 -- b --> q0 style start fill:none,stroke:none style q0 fill:none,stroke:none style q1 fill:none,stroke:none </pre>
$b^* a^*$	<pre> graph LR start(()) --> q0(((q0))) q0 -- b --> q1(((q1))) q1 -- b --> q1 q0 -- a --> q2(((q2))) q2 -- a --> q2 style start fill:none,stroke:none style q0 fill:none,stroke:none style q1 fill:none,stroke:none style q2 fill:none,stroke:none </pre>
$(a b)^*$	<pre> graph LR start(()) --> q0(((q0))) q0 -- "a,b" --> q0 style start fill:none,stroke:none style q0 fill:none,stroke:none </pre>
$(a b)^+$	<pre> graph LR start(()) --> q0((q0)) q0 -- "a,b" --> q1(((q1))) q1 -- "a,b" --> q1 style start fill:none,stroke:none style q0 fill:none,stroke:none style q1 fill:none,stroke:none </pre>
$a(a b)^*$	<pre> graph LR start(()) --> q0((q0)) q0 -- a --> q1(((q1))) q1 -- "a,b" --> q1 style start fill:none,stroke:none style q0 fill:none,stroke:none style q1 fill:none,stroke:none </pre>
$(a b)^*ba$	<p>AFND:</p> <pre> graph LR start(()) --> q0((q0)) q0 -- "a,b" --> q0 q0 -- b --> q1((q1)) q1 -- a --> q2(((q2))) style start fill:none,stroke:none style q0 fill:none,stroke:none style q1 fill:none,stroke:none style q2 fill:none,stroke:none </pre> <p>AFD:</p>

	<pre> graph LR start(()) --> q0((q0)) q0 -- a --> q1((q1)) q1 -- b --> q0 q1 -- a --> q2(((q2))) q2 -- b --> q1 q1 -- a --> q1 q1 -- b --> q1 </pre>
$((a b)(a b))^+$	<pre> graph LR start(()) --> q0((q0)) q0 -- "a,b" --> q1((q1)) q1 -- "a,b" --> q2(((q2))) q2 -- "a,b" --> q1 </pre>
$a(a b)^*b$	<p>AFND:</p> <pre> graph LR start(()) --> q0((q0)) q0 -- a --> q1((q1)) q1 -- "a,b" --> q1 q1 -- b --> q2(((q2))) </pre> <p>AFD:</p> <pre> graph LR start(()) --> q0((q0)) q0 -- a --> q1((q1)) q1 -- a --> q2(((q2))) q2 -- b --> q1 q1 -- a --> q1 q1 -- b --> q1 q2 -- b --> q2 </pre>

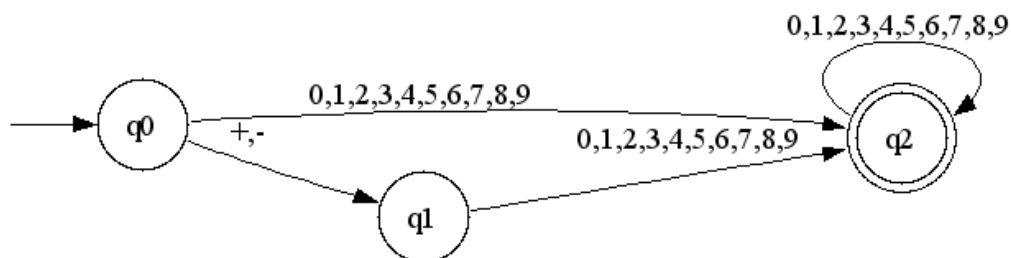
<p>bb (a b) +aaa</p>	<p>AFND:</p>  <p>AFD:</p> 
<p>a?b+</p>	<p>AFλ:</p>  <p>AFD:</p> 

14. Estudo de caso 1: número inteiros

AFλ :



AFD:

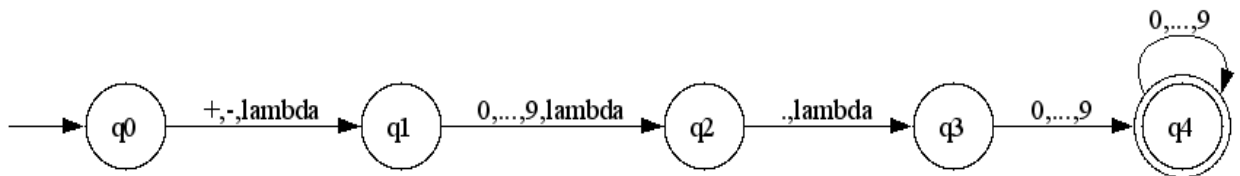


15. Estudo de caso 2: números reais

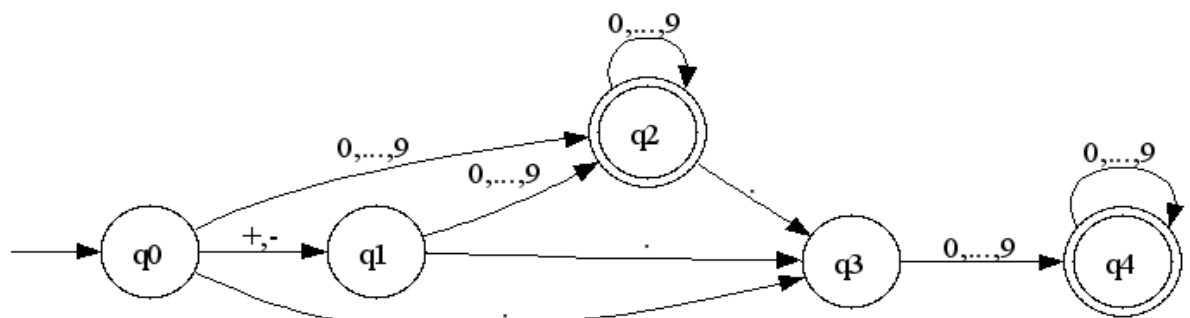
Exemplos de strings da linguagem: $L = \{ 2.5, +2.5, -2.5, +2, -2, +0, 0, .5, -.5, \dots \}$

ER: $(+|-|\lambda)(0|\dots|9)^*(.)?(0|\dots|9)^+$

AF λ :



AFD:



GR:

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

$V = \{S, A, B\}$

$T = \{+, -, ., 0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$

$P = \{$
 $S \rightarrow +A \mid -A \mid A$
 $A \rightarrow 0A \mid \dots \mid 9A \mid .B \mid B$
 $B \rightarrow 0B \mid \dots \mid 9B \mid 0 \mid \dots \mid 9$
 $\}$

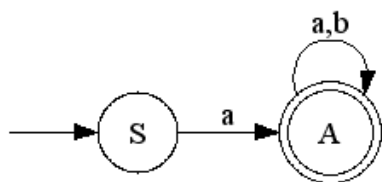
16. GR x AFD x ER

Se uma dada linguagem formal pode ser representada por um dos elementos GR, AFD ou ER, então ela é regular e pode-se encontrar os outros dois elementos.

17. Algoritmo para converter AFD em GR (sem conflitos à esquerda):

Cada estado do AFD se transforma numa variável da gramática, e cada transição numa regra. Os estados finais simbolizam a variável que vai para λ .

Exemplo:



pode ser convertido em:

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

$$V = \{S, A\}$$

$$T = \{a, b\}$$

$$P = \{ S \rightarrow aA$$

$$A \rightarrow aA \mid bA \mid \lambda$$

}

IMPORTANTE: usando um AFD ao invés de um AF λ ou AFND, tem-se uma gramática regular sem conflitos à esquerda.

Observe nos exemplos a seguir que o uso reverso do algoritmo, isto é, partindo-se de uma gramática regular para um autômato é possível chegar a um AFD, AF λ ou AFND:

1. Gramática com conflito à esquerda para a linguagem $a(a^*|b^*)$

$$S \rightarrow aA \mid aB$$

$$A \rightarrow aA \mid \lambda$$

$$B \rightarrow bB \mid \lambda$$

Nesse caso, pelo algoritmo, obtém-se um **AFND**.

2. Gramática fatorada (sem conflito à esquerda) para a mesma linguagem $a(a^*|b^*)$

$$S \rightarrow aC$$

$$C \rightarrow A \mid B$$

$$A \rightarrow aA \mid \lambda$$

$$B \rightarrow bB \mid \lambda$$

Nesse caso, pelo algoritmo obtém-se um **AF λ** .

3. Gramática sem conflito à esquerda para a mesma linguagem $a(a^*|b^*)$

$$S \rightarrow aC$$

$$C \rightarrow aA \mid bB \mid \lambda$$

$$A \rightarrow aA \mid \lambda$$

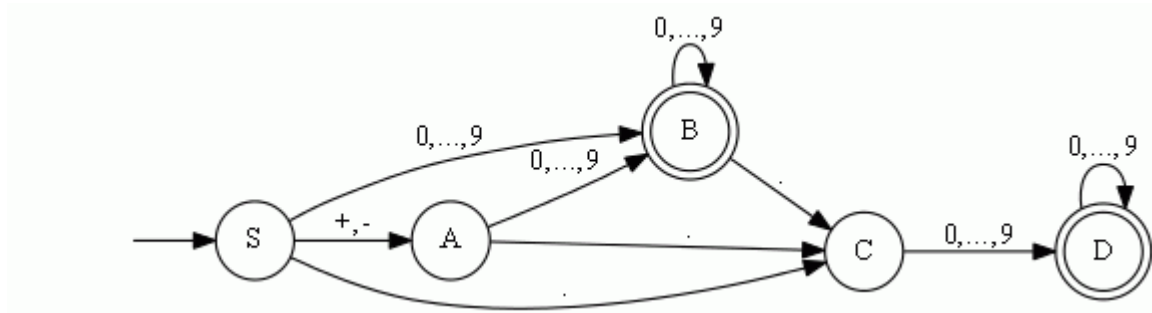
$$B \rightarrow bB \mid \lambda$$

Nesse caso, pelo algoritmo obtém-se um **AFD**.

Apesar desses resultados diferentes para a mesma linguagem é importante lembrar que sempre haverá um AFD que reconhece uma linguagem regular, isto é, que pode ser representada por uma expressão regular e gerada por uma gramática regular.

18. Estudo de caso 2: números reais (gramática regular sem conflitos)

Vamos refazer a gramática regular através do AFD:



GR fatorada (sem conflitos à esquerda):

```
G = (V, T, P, S)
V = {S, A, B, C, D}
T = {+, -, ., 0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
P = {
  S -> +A | -A | .C | 0B | ... | 9B
  A -> 0B | ... | 9B | .C
  B -> .C | 0B | ... | 9B | λ
  C -> 0D | ... | 9D
  D -> 0D | ... | 9D | λ
}
```

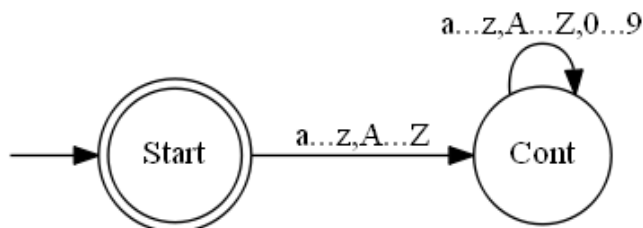
19. Estudo de caso 3: nomes de variáveis

Suponha que estas variáveis só podem ser formadas por uma letra obrigatória e, opcionalmente, seguidas de letras e números.

Exemplos de strings da linguagem: $L = \{x, \text{Valor}, \text{Salario}13, a2b3, \dots\}$

ER: $(a | \dots | Z) (0 | \dots | 9 | a | \dots | Z)^*$

AFD:



GR:

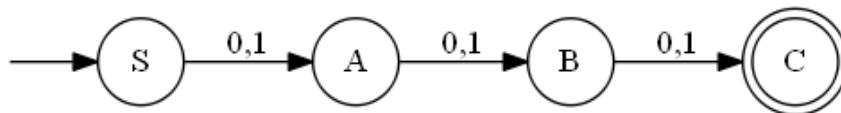
```
G = (V, T, P, <S>)
V = {Start, Cont}
T = {0, ..., 9, a, ..., z, A, ..., Z}
P = {
  Start -> a Cont | ... | Z Cont
  Cont -> a Cont | ... | Z Cont | 0 Cont | ... | 9 Cont | λ
}
```

20. Estudo de caso 4: números binários sem sinal e com 3 dígitos

$L = \{ 000, 001, 010, 011, 100, 101, 110, 111 \}$

ER: $(0|1)(0|1)(0|1)$

AFD:



GR:

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

$V = \{ S, A, B \}$

$T = \{ 0, 1 \}$

$P = \{ S \rightarrow 0A \mid 1A$

$A \rightarrow 0B \mid 1B$

$B \rightarrow 0 \mid 1$

$\}$

ou, simplesmente:

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

$V = \{ S \}$

$T = \{ 0, 1 \}$

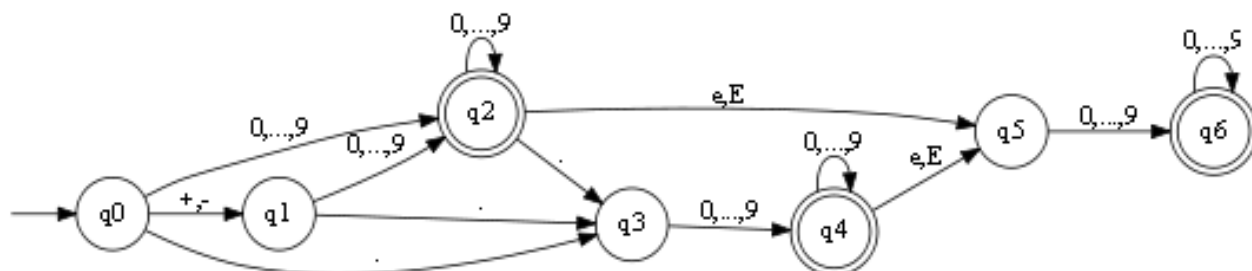
$P = \{ S \rightarrow 000, 001, 010, 011, 100, 101, 110, 111 \}$

21. Estudo de caso 5: números reais com possibilidade de notação científica

Exemplos de strings da linguagem: $L = \{ 2.5E34, +2.5E-45, -2.5e+5, +2e7, .5E100, -.555e555, \dots \}$

ER: $(+|-)?(0|\dots|9)^*(.)?(0|\dots|9)^+((E|e)(+|-)?(0|\dots|9)^+)?$

AFD:



GR:

$G = (V, T, P, \langle S \rangle)$

$V = \{ q0, q1, q2, q3, q4, q5, q6 \}$

$T = \{ +, -, ., 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, e, E \}$

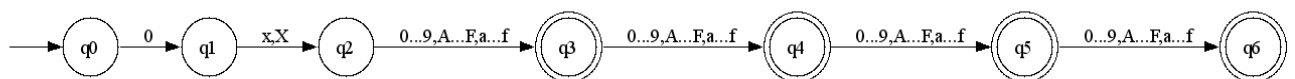
$$\begin{aligned}
 P = \{ & q_0 \rightarrow + q_1 \mid - q_1 \mid . q < C > \mid 0 q_1 \mid \dots \mid 9 q_1 \\
 & q_1 \rightarrow 0 q_2 \mid \dots \mid 9 q_2 \mid . q_2 \\
 & q_2 \rightarrow . q_3 \mid e q_5 \mid E q_5 \mid 0 q_2 \mid \dots \mid 9 q_2 \mid \lambda \\
 & q_3 \rightarrow 0 q_4 \mid \dots \mid 9 q_4 \\
 & q_4 \rightarrow 0 q_4 \mid \dots \mid 9 q_4 \mid \lambda \mid e q_5 \mid E q_5 \\
 & q_5 \rightarrow 0 q_6 \mid \dots \mid 9 q_6 \\
 & q_6 \rightarrow 0 q_6 \mid \dots \mid 9 q_6 \mid \lambda \\
 & \}
 \end{aligned}$$

22. Estudo de caso 6: números hexadecimais em Java, de 1 a 4 dígitos

Exemplos de strings da linguagem: $L = \{ 0xABCD, 0x0000, 0XA, 0X3a3f, 0x25, \dots \}$

ER: $0(X|x)(0|\dots|F)(0|\dots|F)?(0|\dots|F)?(0|\dots|F)?$

AFD:



23. Estudo de caso 7: números octais em Java

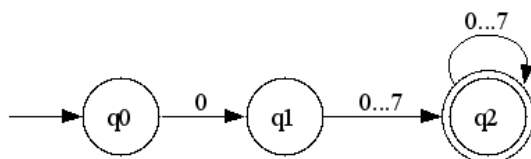
São sempre iniciados por zero e formados por dígitos de 0 a 7. Convencionaremos que apenas um zero não é octal, mas decimal.

Supomos também a inexistência de sinal.

$L = \{ 05, 01234567, 00, 01010, \dots \}$

ER: $0(0|\dots|7)^+$

AFD:



24. Conversão de AFND e AFλ em AFD

Todo AFND e AFλ pode ser convertido num AFD equivalente.

O estudo do algoritmo para a conversão e a sua implementação será avaliado como uma dos trabalhos da disciplina.

25. Minimização de um Autômato Finito

Um autômato mínimo de uma linguagem regular L é um autômato finito determinístico (AFD) que reconhece linguagem L com a menor quantidade possível de estados.

Condições:

- (i) Deve ser determinístico.
- (ii) Não pode ter estados inacessíveis.

Eficiência da busca:

Este tipo de busca perde um tempo inicial na montagem do autômato referente aos termos que serão buscados, porém, no momento da busca propriamente dita, o algoritmo tem complexidade linear $O(n)$. Um modelo de algoritmo pode ser encontrado em

<http://www-igm.univ-mlv.fr/~lecroq/string/node4.html#SECTION0040>

Em comparação com outros algoritmos de busca de texto ele é um dos melhores pois no momento da busca não há repetições, isto é, o ponteiro da busca vai sempre avançando para o próximo caractere, mesmo que estejam sendo procuradas palavras similares ou se há ocorrências parecidas no texto. Mais informações sobre comparativos de métodos de busca, inclusive os algoritmos, fogem do escopo desse material, mas podem ser encontrados em <http://www-igm.univ-mlv.fr/~lecroq/string/index.html>.

Curiosidade: buscas realizadas em uma página de um browser de internet usam normalmente o algoritmo Boyer-Moore, que em alguns casos é mais rápido do que um AFD. Por outro lado, buscas realizadas na internet, tal como a máquina de busca do Google, usam um índice invertido (os documentos são referenciados a partir da lista de palavras existentes) onde há eficiência na busca em detrimento do tempo na montagem dos índices. Dessa forma, devido a dependência da atualização dos índices pelos robôs, elas não são boas para encontrar informações mais novas, que foram recentemente publicadas.

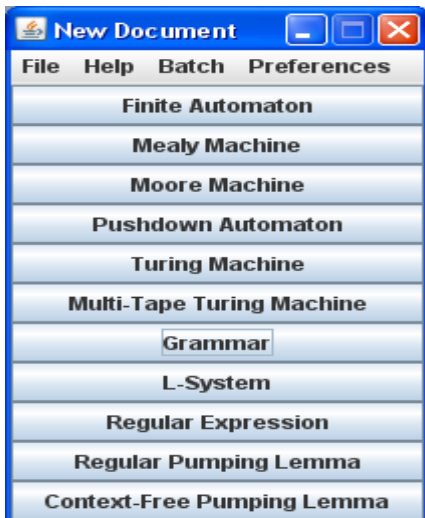
27. Uso do simulador JFLAP para Linguagens Regulares (opcional)

Página principal do JFLAP: <http://www.jflap.org/>

Download: <http://www.jflap.org/jflaptmp/>

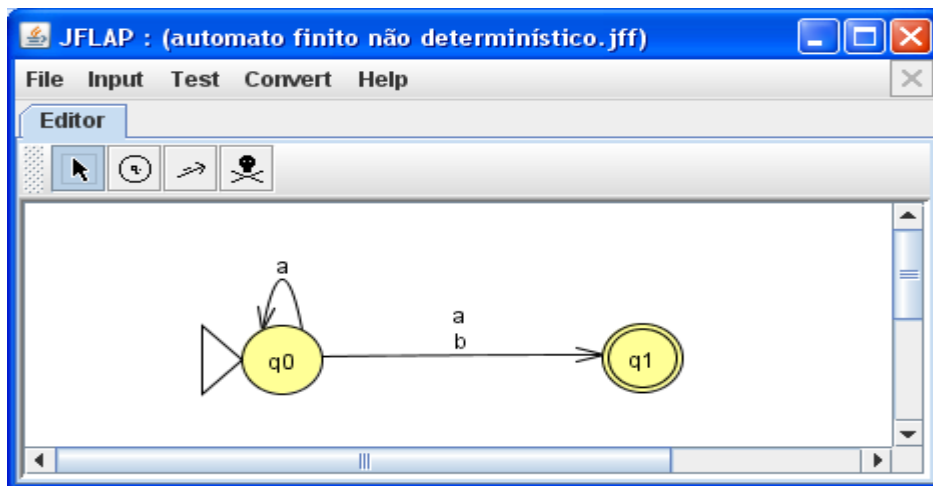
Tutorial: <http://www.jflap.org/tutorial/>

Execute diretamente o arquivo 'JFLAP.jar'





Autômato finito (determinístico e não determinístico)

- 1- Selecione 'Finite Automaton'
- 2- Construa o grafo:



Dicas para construção:

- para criar uma transição arraste o mouse do estado inicial até o estado final.
- o software não aceita a notação 'a,b' como transições para os símbolos 'a' ou 'b', mas devem ser feitas uma transição separada para cada símbolo.
- use o botão direito para selecionar estado inicial e final (selecione o botão ).
- uma transição para o próprio estado é feita com um clique sobre ele (selecione o botão .

3- Teste do autômato:

Selecione 'Input' - 'Multiple Run'

Digite as strings de teste e selecione 'Run Inputs':

Input	Result
	Reject
a	Accept
aa	Accept
aaa	Accept
ab	Accept
aab	Accept
b	Accept
abb	Reject
aabb	Reject
bbb	Reject

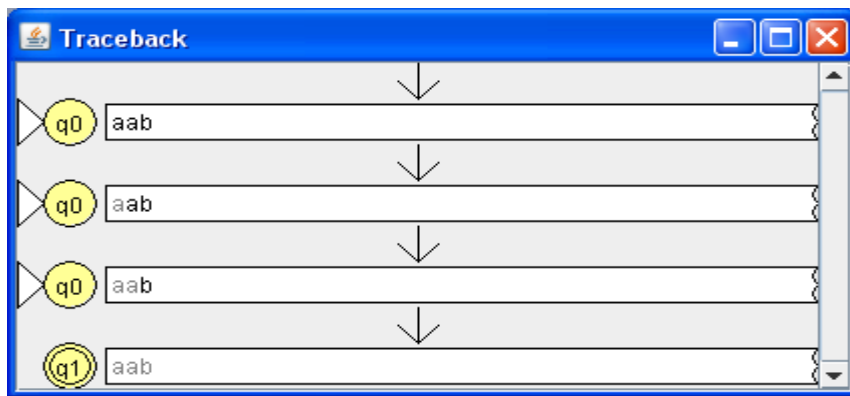
Buttons: Run Inputs, Clear, Enter Lambda, View Trace

Verifique o passo-a-passo de cada string selecionando-a, e

Input	Result
	Reject
a	Accept
aa	Accept
aaa	Accept
ab	Accept
aab	Accept
b	Accept
abb	Reject
aabb	Reject
bbb	Reject

Buttons: Run Inputs, Clear, Enter Lambda, View Trace

em seguida clique 'View Trace':



Cada linha representa um símbolo consumido.

28. Máquina de Mealy

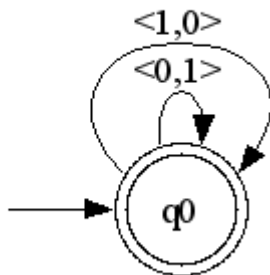
Pode ser considerado como um Autômato Finito Determinístico (AFD) com saídas associadas às transições.

Definição:

Uma Máquina de Mealy é uma 6-upla $M = (\Sigma, Q, \delta, q_0, F, \Delta)$ onde:

- Σ é o alfabeto dos símbolos de entrada
- Q é o conjunto dos estados possíveis
- δ é a função transição, $\delta: Q \times \Sigma \rightarrow Q \times \Delta^*$
- q_0 é o estado inicial
- F é o conjunto de estados finais. Deve possuir ao menos um elemento.
- Δ é o alfabeto de símbolos de saída

Exemplo: uma Máquina de Mealy que lê uma cadeia de 0's e 1's e produz uma saída trocando os caracteres da entrada (0's por 1's e 1's por 0's).



29. Máquina de Moore

É um AFD com saídas associadas aos estados.

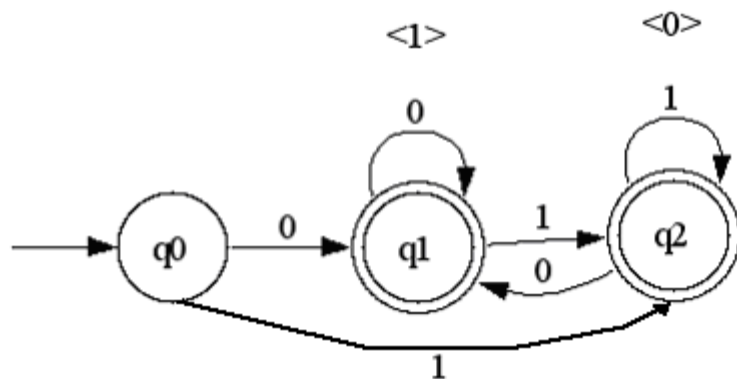
Aplicação típica de Máquinas de Moore é no desenvolvimento de Analisadores Léxicos em Compiladores, onde são classificados os caracteres de entrada de um programa em tokens, ou seja, em unidades léxicas que representam uma categoria de elementos. Exemplos de tokens: números reais, variáveis, operadores relacionais, palavras reservadas, etc.

Definição:

Uma Máquina de Moore é uma 7-upla $M = (\Sigma, Q, \delta, q_0, F, \Delta, G)$ onde:

- Σ é o alfabeto dos símbolos de entrada
- Q é o conjunto dos estados possíveis
- δ é a função transição, $\delta: Q \times \Sigma \rightarrow Q$
- q_0 é o estado inicial
- F é o conjunto de estados finais. Deve possuir ao menos um elemento.
- Δ é o alfabeto de símbolos de saída
- G é a função de saída, $G: Q \rightarrow \Delta^*$

Exemplo: observe uma Máquina de Moore para solucionar o mesmo problema resolvido pela Máquina de Mealy anterior, ou seja, lê e troca os caracteres da entrada para a saída.



30. Exemplo de Máquina de Moore para classificar números inteiros sem sinal

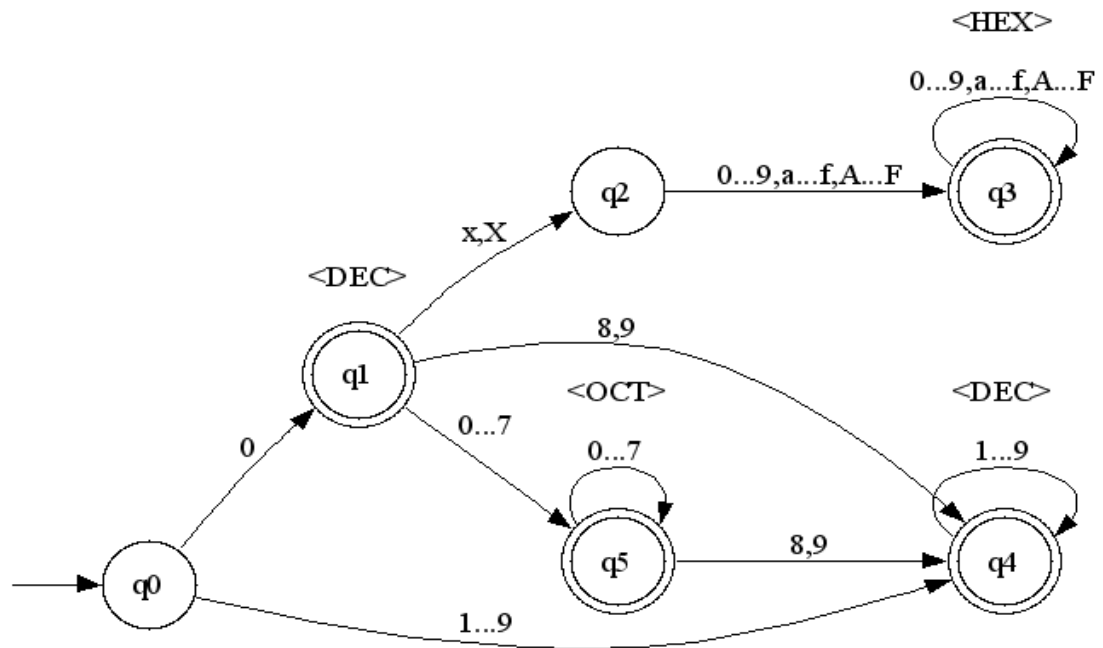
Esta máquina de Moore classifica números inteiros sem sinal de acordo com a sua base: decimal, octal e hexadecimal.

ER para cada saída da máquina:

- ER para números hexadecimais <HEX>: $0(X|x)(0|\dots|F)^+$
- ER para números decimais <DEC>: $(0|\dots|9)^+$
- ER para números octais <OCT>: $0(0|\dots|7)^+$

Serão considerados “0”, “08” e “058” como decimais.

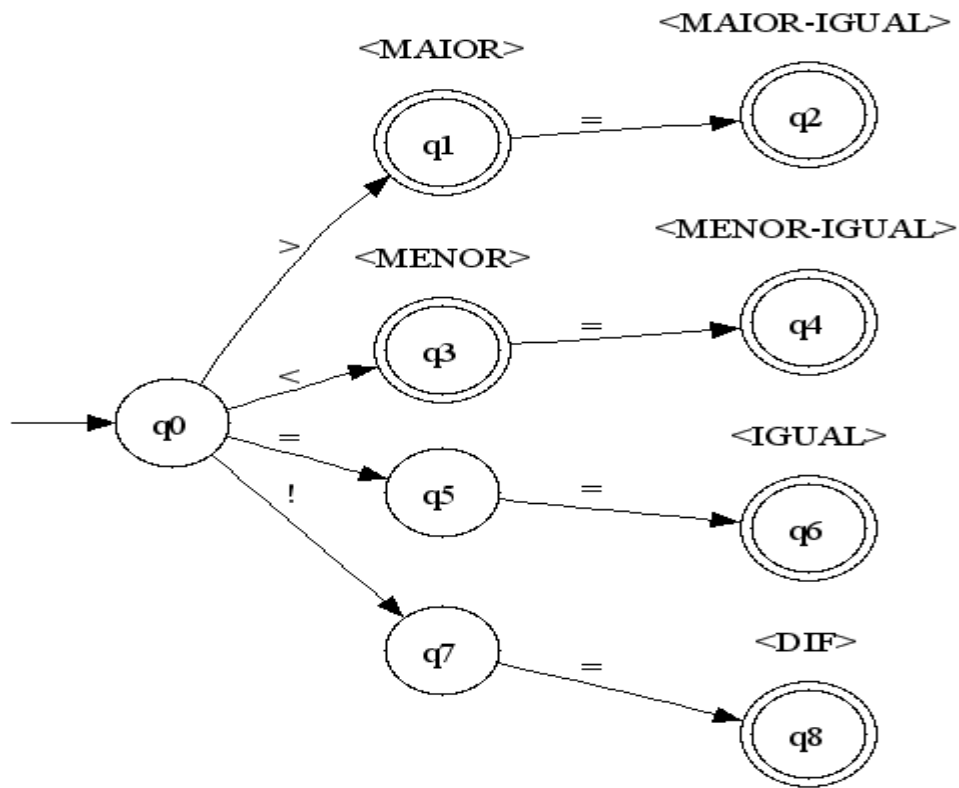
Obs.: as saídas são representadas pelas descrições entre ‘<’ e ‘>’ e em letras maiúsculas.



31. Exemplo de Máquina de Moore para classificar os operadores relacionais do C/Java

ER para cada saída da máquina:

<MAIOR>	>
<MENOR>	<
<MAIOR-IGUAL>	>=
<MENOR-IGUAL>	<=
<IGUAL>	==
<DIF>	!=



32. Máquina de Moore para construção de Analisador Léxico

Uma aplicação interessante para a Máquina de Moore é a construção de analisadores léxicos em compiladores. No apêndice B, encontra-se um exemplo, desenvolvido em Java, de um analisador para uma linguagem simples.

Sugere-se como trabalho da disciplina a implementação de um analisador léxico para um subconjunto de uma linguagem de programação conhecida.

33. Gramática Livre de Contexto (GLC)

É um formalismo gerador para linguagens do tipo 2, na Hierarquia de Chomsky. Numa GLC, as regras não tem limitação do lado direito.

Isto é, elas tem o formato:

$$\boxed{A \rightarrow \beta}$$

Onde: $A \in V$ (conjunto de variáveis)

$\beta \in (T \cup V)^*$ (concatenações de símbolos entre T e V)

34. Exemplos de linguagens livres de contexto

Como elas não são regulares, não existe expressão regular que as represente.

Observe a outra notação usada para a sua representação:

a^n = a's concatenados n vezes

i) $\{ a^i b^i ; i \geq 0 \}$

$$S \rightarrow aSb \mid \lambda$$

ii) $\{ a^{2i} b^{2i} ; i \geq 1 \}$

$$S \rightarrow aaSbb \mid aabb$$

iii) $\{ a^i b^i a^k b^k ; i \geq 0, k \geq 0 \}$

$$S \rightarrow AA$$

$$A \rightarrow aAb \mid \lambda$$

iv) $\{ a^i b^k a^k b^i ; i \geq 1, k \geq 1 \}$

$$S \rightarrow aSb \mid aAb$$

$$A \rightarrow bAa \mid ba$$

v) palíndromos sobre {a,b}

$$S \rightarrow aSa \mid bSb \mid a \mid b \mid \lambda$$

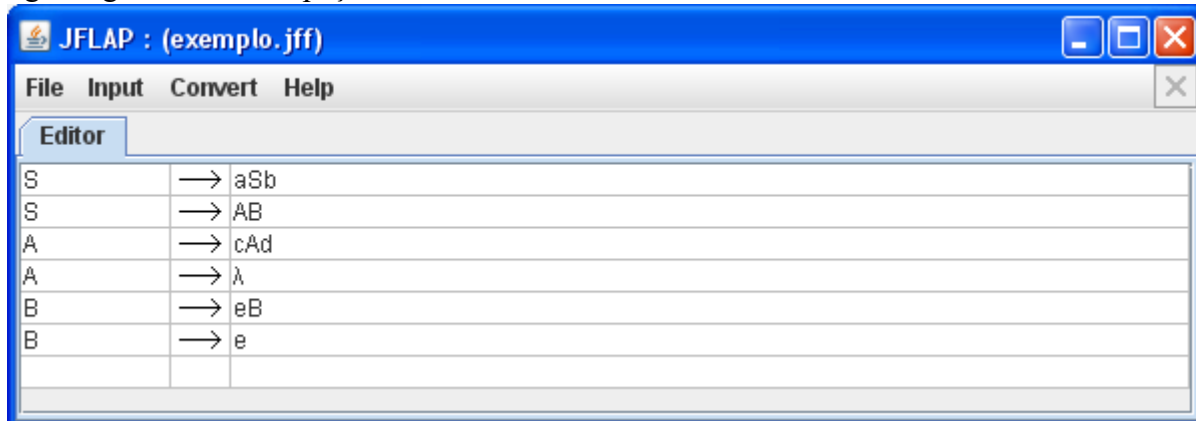
35. Uso do simulador JFLAP para Linguagens Livres de Contexto (opcional)

Página principal do JFLAP: <http://www.jflap.org/>, e download: <http://www.jflap.org/jflaptmp/>

Gramática livre de contexto

1- Selecione 'Grammar'

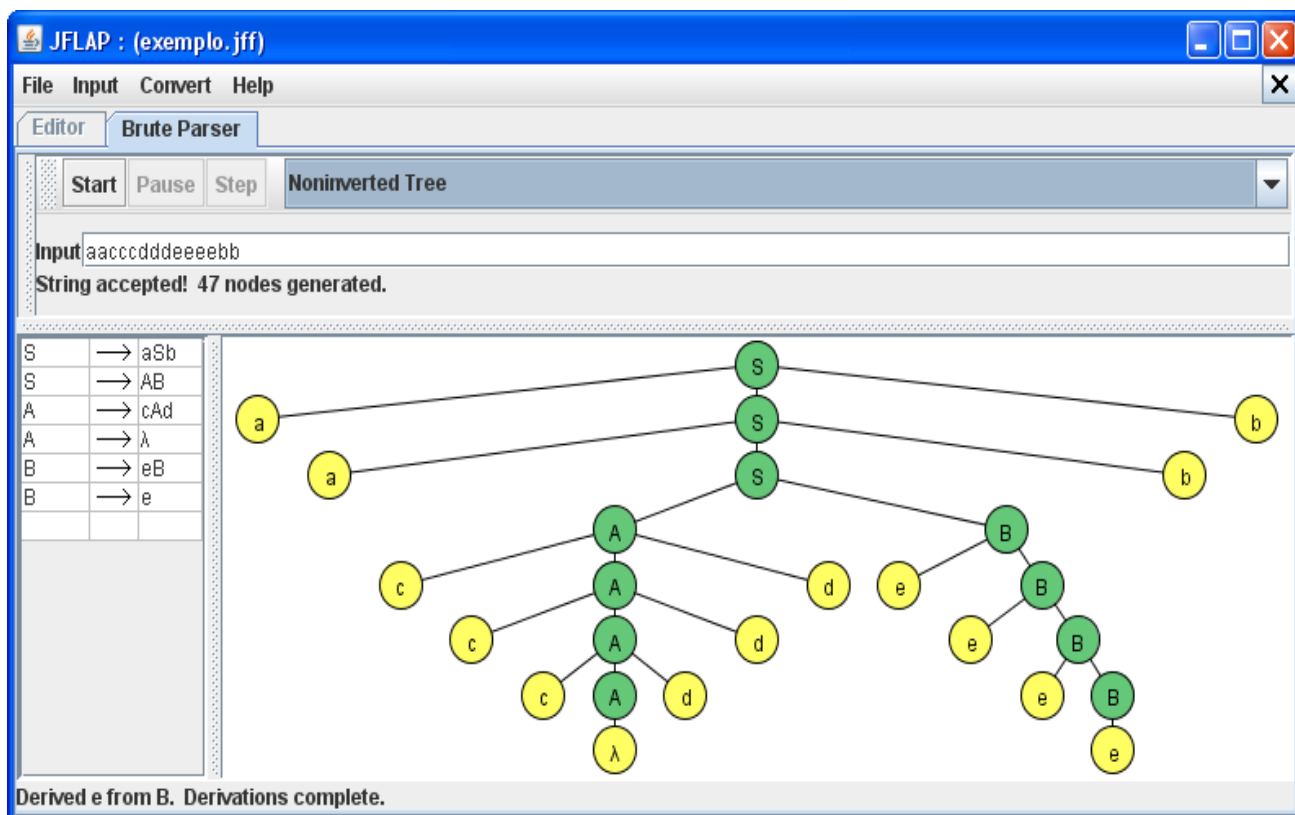
2- Digite a gramática no espaço indicado:



3- Para testar uma string, selecione 'Input' – 'Brute Force Parse'

Digite a string desejada na caixa 'Input' e clique em 'Start'

Se houver indicação que ela foi aceita, pressione 'Step' várias vezes para visualizar a árvore de derivação.



36. Lema do Bombeamento para Linguagens Regulares

Se L é linguagem regular, então $\exists p \geq 1; \forall w \in L, |w| \geq p$ onde $w = xyz$ com:

1. $|y| \geq 1$
2. $|xy| \leq p$
3. $\forall i \geq 0, xy^iz \in L$

Observações:

- y é uma substring que pode ser bombeada (repetida, ou removida quando $i = 0$)
- y precisa estar entre os p primeiros caracteres
- repetindo a substring y (ou omiti-la), xyz continua pertencendo a L

O Lema do Bombeamento descreve que todas as palavras de uma dada linguagem regular possuem uma subpalavra central que, se repetida arbitrariamente, continua pertencendo a linguagem, isto é, essa subcadeia pode ser “bombeada” que ainda irá produzir uma outra que pertence a linguagem original..

Obs.: a constante p pode ser considerada como a qtd de estados do menor AFD que reconhece a linguagem regular (SUDKAMP, 1997, p. 212). Por isso, é recomendado o limite $p \geq 1$, tal como diretamente estabelecido por Lewis e Papadimitriou (2000, p. 91). Tal limite é útil em provas da não regularidade de algumas linguagens, tal como o exemplo 3 no próximo tópico onde há utilização direta dessa condição: $p \geq 1$. Na prova por absurdo desse exemplo, $w = a^{2p} b^p b^p a^p$ continuará não pertencendo a L , já que se fosse considerado $p = 0$, por outro lado se fizéssemos $p=0$, w seria λ e pertenceria a L , invalidando a prova.

Exemplo 1:

Linguagem: $L = ab^*$

L é regular, logo deve atender as condições do Lema:

Como o menor automato que reconhece L possui 2 estados,

$\exists p = 2; \forall w \in L, |w| \geq 2$ onde $w = xyz$ com $|y| \geq 1, |xy| \leq 2$

Testando para cada um w :

$w = ab; x=a, y=b, z=\lambda. \forall i \geq 0, xy^iz \in L$

$w = abb; x=a, y=b, z=b. \forall i \geq 0, xy^iz \in L$

$w = abbb; x=a, y=b, z=bb. \forall i \geq 0, xy^iz \in L$

...

e assim por diante.

Exemplo 2:

Linguagem: $L = ab$

Tal com o lema afirma, se L é regular então deverá atender as suas condições:

L é uma linguagem finita, então .

$\exists p = 3; \forall w \in L, |w| \geq 3$ onde $w = xyz$ com $|y| \geq 1, |xy| \leq 2$

Como não existe nenhum w com tamanho maior ou igual a 3, então não existe contra-exemplo e a linguagem é regular.

Atenção: os exemplos 1 e 2 ilustram que se uma linguagem é regular, ocorre o Lema do Bombeamento. Porém, esse Lema não serve para provar que uma linguagem é regular!

Veja a continuação dessa discussão no próximo tópico

37. Aplicação do Lema do Bombeamento para a Prova de Não Regularidade de Linguagens

Questão motivadora: como descobrir que uma determinada linguagem é regular ou não?

Sabe-se que a prova da sua regularidade é a simples apresentação de um dos três elementos abstratos, ER, AFD ou GR, que, respectivamente, denotam, reconhecem e geram a linguagem.

Por outro lado, para provar que ela não é regular, pode-se usar o Lema do Bombeamento. Observe a lógica inferida a partir do lema:

Se uma linguagem é regular (LR) então atende as condições do lema do bombeamento (LB):

LR \rightarrow LB

Pela lógica proposicional:

LR \rightarrow LB	ok
LB \rightarrow LR	não podemos afirmar
\sim LR \rightarrow \sim LB	não podemos afirmar
\simLB \rightarrow \simLR	ok , pela equivalência lógica da contraposição

Assim, devido a contraposição, usa-se o lema como uma ferramenta para provar que uma linguagem não é regular, isto é, se for mostrado que ela não atende ao Lema do Bombeamento então conclui-se que ela não é regular: **\sim LB \rightarrow \sim LR**

Para isso, usaremos a técnica da prova por absurdo: supomos inicialmente que a linguagem é regular (**LR**), depois verificamos que o Lema não é satisfeito (**\sim LB**) e, finalmente, concluimos, por absurdo, que a linguagem não é regular (**\sim LR**).

Reafirmamos então que o Lema do Bombeamento é bom para provar que uma determinada linguagem L não é regular, porém, observamos que isso nem sempre é possível, ou seja, não podemos afirmar \sim LR \rightarrow \sim LB.

Observe que para buscar a negativa do Lema devemos negar o seu enunciado que é composto pelos quantificadores \exists e \forall . Relembrando da lógica de primeira ordem:

$$\sim(\exists x) \Leftrightarrow \forall x$$

$$\sim(\forall x) \Leftrightarrow \exists x$$

Resumindo, o método empregado pra descobrir que uma linguagem L não é regular é o seguinte:

1º) supomos L regular

2º) tentamos verificar que o Lema não é satisfeito por L.

Isto significa verificar a sua negativa:

$\forall p \geq 1; \exists w \in L, |w| \geq p$ onde $w = xyz$ com: $|y| \geq 1; |xy| \leq p; \exists i \geq 0, xy^iz$ não pertence a L (observe que o tamanho de w é dependente de p, então deve-se encontrar um w que permita o crescimento em função de p)

3º) assim, o lema não é satisfeito e, por absurdo, fica provado que L não é regular.

Exemplos de provas de não regularidade de linguagens:

Exemplo 1: $L = \{ a^n b^n \mid n \geq 0 \}$

Suponha por absurdo que L é regular.

Vamos tentar mostrar que o Lema do Bombeamento não é satisfeito.

Fazendo a negação do Lema:

$\forall p \geq 1;$

$\exists w = xyz \in L, |w| \geq p$ onde $w = a^{p-1} a b^p$ com $x = a^{p-1}, y = a, z = b^p, |y| \geq 1; |xy| \leq p;$

$\exists i = 2, xy^iz = a^{p-1} a^2 b^p$ não pertence a L

Assim, considerando a suposição de que L é regular e considerando que o Lema não foi satisfeito, ocorreu uma contradição!

Portanto, L não é regular.

Exemplo 2: $L = \{ a^j b^k \mid j \geq k \geq 0 \}$

Suponha por absurdo que L é regular.

Vamos tentar mostrar que o Lema do Bombeamento não é satisfeito.

Fazendo a negação do Lema:

$\forall p \geq 1;$

$\exists w = xyz \in L, |w| \geq p$ onde $w = a^p b^p$ com $x = \lambda, y = a^p, z = b^p, |y| \geq 1; |xy| \leq p;$

$\exists i = 0, xy^iz = a^0 b^p$ não pertence a L

Assim, considerando a suposição de que L é regular e considerando que o Lema não foi satisfeito, ocorreu uma contradição!

Portanto, L não é regular.

Exemplo 3: $L = \{ x x^R \mid x \in \{a,b\}^* \}$

Suponha por absurdo que L é regular.

Vamos tentar mostrar que o Lema do Bombeamento não é satisfeito.

Fazendo a negação do Lema:

$\forall p \geq 1;$

$\exists w = xyz \in L, |w| \geq p$ onde $w = a^p b^p b^p a^p$ com $x = \lambda, y = a^p, z = b^p b^p a^p, |y| \geq 1; |xy| \leq p;$

$\exists i = 2, xy^iz = a^{2p} b^p b^p a^p$ não pertence a L

Assim, considerando a suposição de que L é regular e considerando que o Lema não foi satisfeito, ocorreu uma contradição!

Portanto, L não é regular.

Exemplo 4: $L = \{ a^j b^k \mid j \neq k \}$

Suponha por absurdo que L é regular.

Vamos tentar mostrar que o Lema do Bombeamento não é satisfeito.

Fazendo a negação do Lema:

$\forall p \geq 1;$

$\exists w = xyz \in L, |w| \geq p$ onde $w = a^p b^{2p}$ com $x = \lambda, y = a^p, z = b^{2p}, |y| \geq 1; |xy| \leq p;$

$\exists i = 2, xy^iz = a^{2p} b^{2p}$ não pertence a L

Assim, considerando a suposição de que L é regular e considerando que o Lema não foi satisfeito, ocorreu uma contradição!

Portanto, L não é regular.

Exemplo 5: $L = \{ a^n \mid n \text{ é primo} \}$

Suponha por absurdo que L é regular.

Vamos tentar mostrar que o Lema do Bombeamento não é satisfeito. Fazendo a negação do Lema:

$\forall p \geq 1;$

$\exists w = xyz \in L, |w| \geq p$ onde $w = a^{p+k-1}$ com $x = \lambda, y = a, z = a^{p+k-1}, p+k-1 = n^\circ \text{ primo}, |y| \geq 1; |xy| \leq p;$

$\exists i = p+k+1, xy^iz = a^{p+k-1} a^{p+k-1}$ não pertence a L

Assim, considerando a suposição de que L é regular e considerando que o Lema não foi satisfeito, ocorreu uma contradição!

Portanto, L não é regular.

38. Aplicações de Linguagens Livres de Contexto

Os elementos classificados numa máquina de Moore pertencem às linguagens regulares, são representados por expressões regulares, gerados por gramáticas regulares e reconhecidos por autômatos finitos determinísticos ficando limitados em representações simples como números, variáveis, palavras reservadas, operadores etc.

Problemas mais complexos, como trechos de um programa, expressões com parênteses balanceados não podem ser representados por gramáticas regulares, precisando subir um nível na Hierarquia de Chomsky para serem representados, neste caso, as linguagens livres de contexto.

Estudo de caso: GLC para gerar trechos de uma linguagem de programação parecida com o Pascal
Suponha os seguintes tokens já classificados por um analisador léxico através de uma máquina de Moore.

Token	Expressão regular
<NUM>	$(0 \dots 9)^*(.)?(0 \dots 9)^+$
<IDENT>	$(a \dots Z)(0 \dots 9 a \dots Z)^*$
<ATRIB>	$:=$
<OPER-ADIT>	$+ -$
<OPER-MULT>	$* /$
<VIRG>	$,$
<PT-VIRG>	$;$
<ABRE-PAR>	$($
<FECHA-PAR>	$)$
<FOR>	$(f F)(o O)(r R)$
<FUNCTION>	$(f F)(u U)(n N)(c C)(t T)(i I)(o O)(n N)$
<BEGIN>	$(b B)(e E)(g G)(i I)(n N)$
<END>	$(e E)(n N)(d D)$
<DO>	$(d D)(o O)$
<TO>	$(t T)(o O)$
<DOWNTO>	$(d D)(o O)(w W)(n N)(t T)(o O)$

Exemplos de GLCs para gerar as linguagens descritas:

- i) Expressão aritmética com parênteses balanceados, números, variáveis e operadores de soma, subtração, multiplicação e divisão.

Exemplos de 5 strings: $2 ; (+3) ; 3-+6 ; 2*(Valor+4) ;$
 $3-(Cont1+Cont2)*(4-(9-8)+1)$

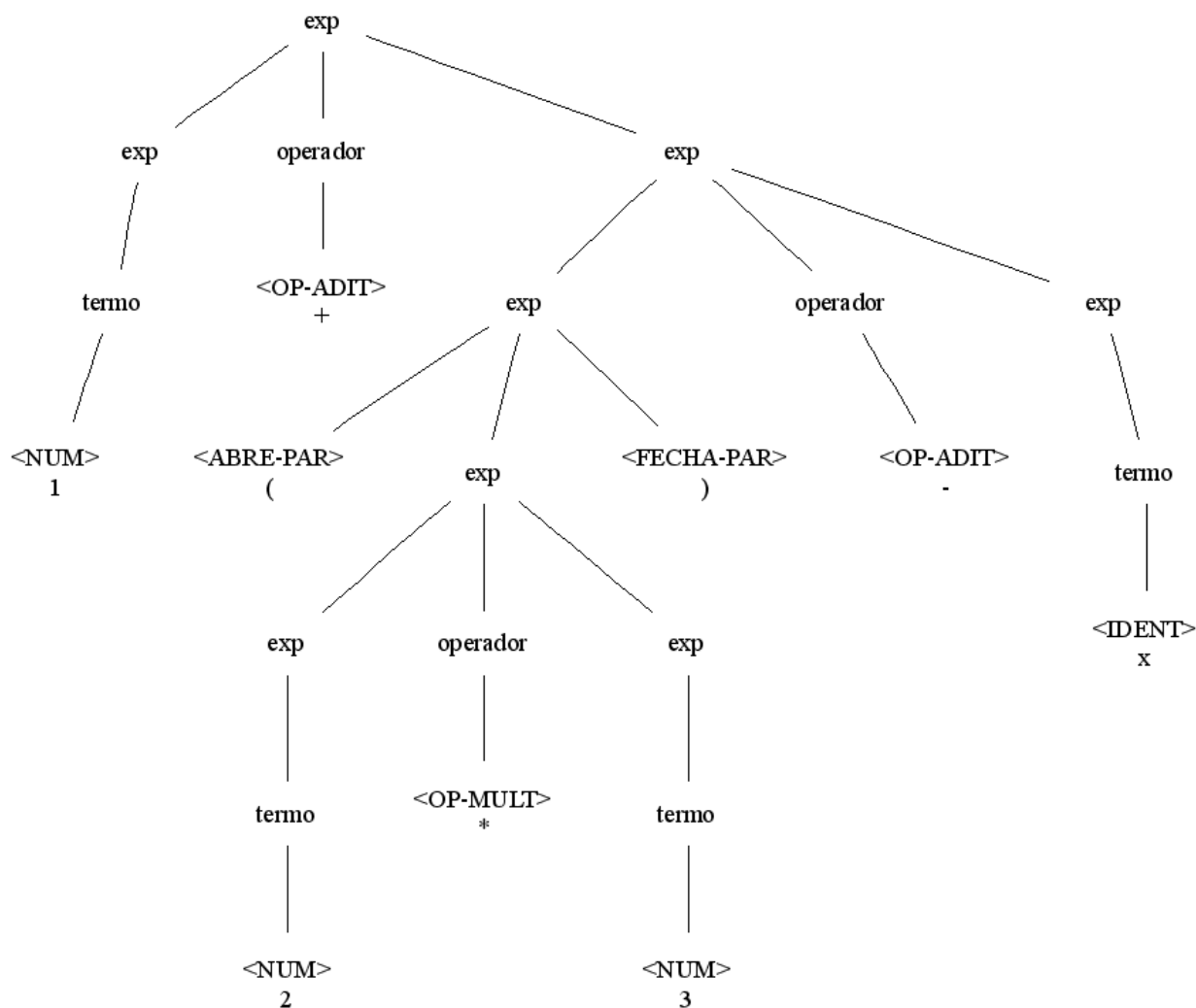
GLC:

```
exp    ->    exp operador exp
          | <ABRE-PAR> exp <FECHA-PAR>
          | termo

termo  ->    <OPER-ADIT> <NUM>
          | <NUM>
          | <IDENT>

operador -> <OPER-ADIT>
          | <OPER-MULT>
```


Exemplo de árvore de derivação para a string: $1 + (2 * 3) - x$



Esta GLC também poderia ser reescrita usando-se um único símbolo não terminal (exp):

```

exp  ->  exp <OPER-ADIT> exp
        | exp <OPER-MULT> exp
        | <ABRE-PAR> exp <FECHA-PAR>
        | <OPER-ADIT> <NUM>
        | <NUM>
        | <IDENT>
  
```

ii) Comando de atribuição.

Exemplos de 2 strings: $x := 3$
 $\text{Valor} := \text{Valor} + 2 * (x+1)$

GLC:

```

atrib -> <IDENT> <ATRIB> exp
  
```

- iii) Bloco de comandos de atribuição. Cada comando deve ser terminado por ponto-e-vírgula.

Exemplo de 1 string:

```
begin
    x := 3;
    Valor := Valor + 2 * (x+1);
end
```

GLC:

```
blocoAtrib -> <BEGIN> listaAtrib <END>
listaAtrib -> atrib <PT-VIRG> listaAtrib |  $\Lambda$ 
```

- iv) Comando de repetição for estilo Pascal.
Considere que o corpo pode ser composto apenas por atribuições e/ou outros comandos for, e o início e fim podem ser expressões aritméticas quaisquer.

Dois exemplos:

1ª string:

```
for x := 0 to 50 do
begin
    x := 3;
    Valor := Valor + 1;
end
```

2ª string:

```
for x := 10 downto 5 do
    for y := 1 to 10 do
        w := w + x + y;
```

GLC:

```
comandoFor -> <FOR> <IDENT> <DOIS-PT> <ATRIB> exp sentido exp
              <DO> corpoFor
sentido -> <TO> | <DOWNTO>
corpoFor -> comandoÚnico | blocoComandos
comandoÚnico -> atrib <PT-VIRG> | comandoFor
blocoComandos -> <BEGIN> listaComandos <END>
listaComandos -> comandoÚnico listaComandos |  $\Lambda$ 
```

- v) Definição de funções no estilo Pascal.
O corpo é formado por comandos de atribuição e/ou for.

Exemplo 1:

```
function Media(Nota1, Nota2: Real; var Peso: Integer): Real;
var
    Soma, Resultado : Real;
    Status : Integer;
begin
```

```

    Soma := Nota1 + Nota2;
    Resultado := Soma/2;
    Status := 1;
    Media := Resultado;
end

```

Exemplo 2:

```

function Constante:Real;
begin
    Constante := 100;
end

```

GLC:

```

declaraFunção -> <FUNCTION> <IDENT> parâmetro <DOIS-PT>
                <PT-VIRG> declaraVar corpoFunção
parâmetro -> <ABRE-PAR> listaArgumentos <FECHA-PAR> | λ
listaArgumentos -> argumento <PT-VIRG> listaArgumentos | argumento
argumento -> referência defVar
referência -> <VAR> | λ
defVar -> listaIdent <DOIS-PT> tipo
tipo -> <IDENT>
listaIdent -> <IDENT> <VIRG> listaIdent | <IDENT>
declaraVar -> λ | <VAR> listaDeclara
listaDeclara -> defVar <PT-VIRG> listaDeclara | defVar
corpoFunção -> blocoComandos

```

39. Gramáticas ambíguas e o problema do else flutuante

Uma gramática é ambígua quando gera a mesma string por mais de uma derivação. Um situação típica onde isto acontece é na gramática que gera a estrutura condicional sem os tokens de marcação de bloco, como por exemplo nas Linguagens C e Java:

```

if (x>5)
if (y>8)
comando1;
else
comando2;

```

Para resolver tal conflito, o `else` sempre pertence ao `if` mais próximo, como mostra a indentação:

```

if (x>5)
    if (y>8)
        comando1;

```

```
else
    comando2;
```

Uma primeira tentativa de produzir a gramática:

```
comando -> if | ...
if ->    <IF> <ABRE-PAR> expressão <FECHA-PAR> comando
        | <IF> <ABRE-PAR> expressão <FECHA-PAR> comando <ELSE> comando
```

Esta gramática é ambígua pois gera o código anterior através de duas árvores de derivação distintas.

Sugere-se como exercício a construção de uma gramática não ambígua para resolver o problema.

40. Análise Preditiva

É uma técnica algorítmica que transforma uma gramática em um algoritmo reconhecedor.

Pré-requisitos para a gramática:

- fatorada (sem conflitos à esquerda)
- sem recursividade à esquerda
- sem ambigüidade

Procedimentos básicos:

- cada variável da gramática se transforma num procedimento.

- cada ‘|’ encontrado numa regra deve ser escrito como uma estrutura condicional onde cada seleção deve verificar a ocorrência de todos os possíveis primeiros terminais.

Os próximos tópicos irão mostrar técnicas de preparação de uma gramática, com a retirada de conflito e recursividade à esquerda.

41. Fatoração de gramáticas

Uma gramática deve ser fatorada quando possui conflitos à esquerda. A fatoração mantém a mesma linguagem gerada pela gramática.

Exemplo de gramática com conflito à esquerda, no símbolo terminal ‘a’ da primeira regra::

```
S -> aS | aA
A -> bA | λ
```

Tentativa de implementação (problema: as duas condicionais são iguais):

```
S() {
    se (prox = 'a')
        reconhece('a'); S();
    senão se (prox = 'a')
        reconhece('a'); A();
}
```

```

A() {
    se(prox = 'b')
        reconhece('b'); A();
}

```

A gramática pode ser reescrita colocando-se o elemento conflitante 'a' em evidência:

```

S -> aR
R -> S | A
A -> bA | λ

```

Implementação da gramática sem conflitos, através da análise preditiva:

```

S() {
    reconhece('a'); R();
}
R() {
    se(prox = 'a')
        S();
    senão se(prox = 'b')
        A();
}
A() {
    se(prox = 'b')
        reconhece('b'); A();
}

```

42. Algoritmo para fatorar gramáticas

Caso geral da eliminação de conflitos à esquerda:

Gramática com conflito à esquerda:

$A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \dots \mid \alpha\beta_n \mid \gamma$
--

Onde: $A \in V$
 $\alpha, \beta_n, \gamma \in (TuV)^*$

Gramática sem conflito à esquerda (fatorada):

$A \rightarrow \alpha R \mid \gamma$ $R \rightarrow \beta_1 \mid \beta_2 \mid \dots \mid \beta_n$
--

Onde: $A, R \in V$
 $\alpha, \beta_n, \gamma \in (TuV)^*$

43. Recursividade à esquerda em gramáticas

Exemplo de gramática com recursividade à esquerda:

```

S -> Sa | b

```

A variável S possui recursividade à esquerda.

Tentativa de implementação (problema: loop infinito):

```

S() {

```

```

se (prox = 'b')
    S(); reconhece('a');
senão se (prox = 'b')
    reconhece('b');
}

```

A gramática, com a mesma linguagem gerada, porém sem recursividade à esquerda:

```

S -> bR
R -> aR | λ

```

Implementação:

```

S() {
    reconhece('b'); R();
}
R() {
    se (prox = 'a')
        reconhece('a'); R();
}

```

44. Algoritmo para eliminação da recursividade à esquerda

Caso geral:

Gramática com recursividade à esquerda:

$A \rightarrow A\beta_1 \mid A\beta_2 \mid \dots \mid A\beta_n \mid \gamma_1 \mid \gamma_2 \mid \dots \mid \gamma_n$
--

Onde: $A \in V$

$\beta_n, \gamma_n \in (TuV)^*$

Gramática sem recursividade à esquerda:

$A \rightarrow \gamma_1 R \mid \gamma_2 R \mid \dots \mid \gamma_n R$
$R \rightarrow \beta_1 R \mid \beta_2 R \mid \dots \mid \beta_n R \mid \lambda$

Onde: $A, R \in V$

$\beta_n, \gamma_n \in (TuV)^*$

45. Exemplo de Análise Preditiva

É uma técnica algorítmica que transforma uma gramática em um algoritmo reconhecedor.

Pré-requisitos para a gramática:

- fatorada (sem conflitos à esquerda)
- sem recursividade à esquerda
- sem ambigüidade

Procedimentos básicos:

- cada variável da gramática se transforma num procedimento.
- cada ']' encontrado numa regra deve ser escrito como uma estrutura condicional onde cada seleção deve

verificar a ocorrência de todos os possíveis primeiros terminais.

Exemplo:

Suponha a seguinte linguagem regular:

$$((a|b)^*c^+d^*(e|f)g)?\langle\text{EOF}\rangle$$

Uma possível gramática para esta linguagem (neste caso, a gramática não é regular):

```
G = (V, T, P, S)
V = {S, A, B, C, D, E}
T = {a, b, c, d, e, f, g, <EOF>}
P = {S -> A <EOF> | <EOF>
      A -> aA | bA | Bg
      B -> cC
      C -> B | DE
      D -> dD | λ
      E -> e | f
    }
```

Como esta gramática não tem conflito, ambigüidade e recursividade à esquerda, aplicaremos a técnica da análise preditiva para geração do algoritmo:

```
função S() {
    se(próximoCaracter=='a' ou próximoCaracter=='b' ou próximoCaracter=='c')
        A(); reconhece(<EOF>);
    senão se(próximoCaracter==<EOF>)
        reconhece(<EOF>);
    senão
        imprime("erro"); // era esperado um dos seguintes caracteres: a, b, c, <EOF>
}
```

```

função A() {
    se(próxCaracter=='a')
        reconhece('a'); A();
    senão se(próxCaracter=='b')
        reconhece('b'); A();
    senão se(próxCaracter=='c')
        B(); reconhece('g');
    senão
        imprime("erro"); // era esperado um dos seguintes caracteres: a,b,c
}

função B() {
    reconhece('c'); C();
}

função C() {
    se(próxCaracter=='c')
        B();
    senão se(próxCaracter=='d' ou próxCaracter=='e' ou próxCaracter=='f')
        D(); E();
    senão
        imprime("erro"); // era esperado um dos seguintes caracteres: c,d,e,f
}

função D() {
    se(próxCaracter=='d')
        reconhece('d'); D();
    senão
        ;
}

função E() {
    se(próxCaracter=='e')
        reconhece('e');
    senão se(próxCaracter=='f')
        reconhece('f');
    senão
        imprime("erro"); // era esperado um dos seguintes caracteres: e,f
}

```

Obs.: A função `reconhece` tem o objetivo de verificar se o próximo caractere é de fato o que foi passado como argumento. Se sim então lê o próximo caractere põe em `próxCaracter`, senão dispara uma mensagem de erro.

```

função reconhece(caracter c) {
    se(próxCaracter == c)
        próxCaracter = lêPróximoCaracterDoArquivo();
    senão
        imprime("erro"); // era esperado o caractere representado por c
        sai_do_programa;
}

```


46. Análise Preditiva na construção de Analisadores Sintáticos

Desde que a gramática de uma determinada linguagem esteja fatorada, sem recursividade à esquerda e sem ambiguidade, constrói-se facilmente um analisador sintático por meio da técnica da análise preditiva explicada no tópico anterior. Neste caso, ele se comporta como um reconhecedor top-down.

No apêndice C encontra-se um exemplo, desenvolvido em Java, de um analisador sintático construído por meio da técnica da análise preditiva.

Este exemplo é a continuidade do analisador léxico apresentado no apêndice B.

Sugere-se como trabalho da disciplina a implementação de um analisador sintático para um subconjunto de uma linguagem de programação conhecida.

47. Forma Normal de Chomsky

Uma gramática livre de contexto está na Forma Normal de Chomsky (FNC) se cada produção segue uma das seguintes formas:

- i) $A \rightarrow BC$
- ii) $A \rightarrow a$

onde:

$A, B, C \in V$

$a \in \Sigma$

Exemplo:

A gramática formada pelas regras:

$S \rightarrow aSb \mid c$

Pode ser reescrita na FNC como:

$S \rightarrow AB \mid c$

$A \rightarrow a$

$B \rightarrow Sb$

$C \rightarrow b$

48. Aplicação da Forma Normal de Chomsky: Algoritmo de Cocke-Younger-Kasami

O algoritmo de Cocke-Younger-Kasami foi desenvolvido por J. Cocke, D. H. Younger e T. Kasami, em 1965. Ele usa uma gramática livre de contexto na FNC e é classificado como bottom-up, ou seja, faz o processamento a partir das folhas da árvore de derivação. Desta forma, ele aceita gramáticas com conflitos à esquerda e recursividade à esquerda.

Sugere-se como trabalho da disciplina o estudo e a implementação deste algoritmo para a construção de um analisador sintático aplicado a um subconjunto de uma linguagem de programação.

49. Forma Normal de Greibach

Uma gramática livre de contexto está na Forma Normal de Greibach (FNG) se toda produção possui a seguinte forma:

$$A \rightarrow a\alpha$$

onde:

$$A \in V$$

$$a \in \Sigma \cup \{\lambda\}$$

$$\alpha \in V^*$$

Exemplo:

A gramática formada pelas regras:

$$S \rightarrow aSb \mid c$$

Pode ser reescrita na FNG como:

$$S \rightarrow aSB \mid c$$

$$B \rightarrow b$$

Uma aplicação interessante da FNG é a construção de autômatos com pilha, que serão estudados mais adiante.

50. Formato BNF para gramáticas livres de contexto

É um formato de gramática usado para simplificar escritas de gramáticas livres de contexto. BNF: Backus Naur Form foi escrito pela primeira vez por John Backus and Peter Naur para a sintaxe da linguagem de programação Algol 60.

O formato BNF consiste em, simplesmente, reescrever regra do tipo:

$$A \rightarrow uA \mid \lambda$$

em:

$$A \rightarrow \{u\}$$

Exemplo:

Suponha uma gramática para gerar declarações de variáveis em Java, como:

```
int a,b;  
double c;
```

suponha também os tokens:

<TIPO>, <IDENT>, <VIRG>, <PT-VIRG>

Uma gramática convencional seria escrita desta forma:

```
listaDeclara -> declara <PT-VIRG> listaDeclara |  $\lambda$   
declara -> <TIPO> listaIdent declara |  $\lambda$ 
```

```
listaIdent -> <IDENT> <VIRG> listaIdent | <IDENT>
```

No formato **BNF** ela ficaria:

```
listaDeclarar -> { declara <PT-VIRG> }  
declara -> { <TIPO> listaIdent }  
listaIdent -> <IDENT> { <VIRG> <IDENT> }
```

51. Formato EBNF para gramáticas livres de contexto

É uma ‘E’xtensão do BNF. É usado também na simplificação da escrita de gramáticas livres de contexto, sendo bem mais poderoso do que o formato BNF.

Usa os mesmos meta-símbolos da expressão regular: *, +, ?, |, além de permitir a associação de subexpressões com parênteses.

Exemplo:

Suponha uma gramática para gerar declarações de variáveis em Java, como:

```
int a,b;  
double c;
```

suponha também os tokens:

```
<TIPO>, <IDENT>, <VIRG>, <PT-VIRG>
```

Uma gramática convencional seria escrita desta forma:

```
listaDeclarar -> declara <PT-VIRG> listaDeclarar |  $\lambda$   
declara -> <TIPO> listaIdent declara |  $\lambda$   
listaIdent -> <IDENT> <VIRG> listaIdent | <IDENT>
```

No formato **EBNF** ela ficaria:

```
listaDeclarar -> ( <TIPO> <IDENT> ( <VIRG> <IDENT> ) * <PT-VIRG> ) *
```

52. Autômato com Pilha

É uma máquina abstrata para reconhecimento de linguagens livres de contexto.
É um AFND com o poder de uma pilha associada.

Definição:

Um autômato com pilha (AP) é uma 6-upla $M = (\Sigma, Q, \Delta, \delta, q_0, F)$ onde:

- Σ é o alfabeto dos símbolos de entrada
- Q é o conjunto finito dos estados possíveis
- Δ é o conjunto de símbolos de pilha
- δ é a função transição $\delta : Q \times (\Sigma \cup \{\lambda\}) \times (\Delta \cup \{\lambda\}) \rightarrow (Q \times \Delta \cup \{\lambda\})$
- q_0 é o estado inicial

- F é o conjunto de estados finais. Deve possuir ao menos um elemento.

Representação do AP por intermédio de um grafo:

Tal como no AFND, o estado inicial é indicado por uma seta sem origem, e os estados finais são representados por dois círculos concêntricos, cada um.

É possível classificar Autômatos com Pilha (ou Autômatos de Pilha - em algumas literaturas) como **Determinístico** ou **Não Determinístico**. Aqui nesse material considera-se não determinístico quando não for mencionado.

Para que ocorra a aceitação da palavra de entrada é necessário que, além de terminar a leitura da palavra e o autômato estar num estado final, a pilha deve estar vazia.

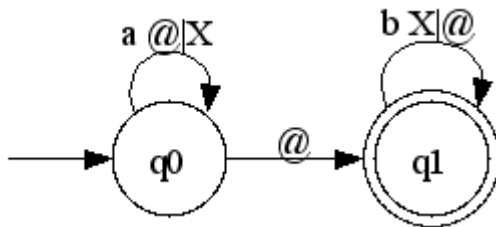
Cada transição é formada pelo símbolo de leitura seguido do símbolo que é desempilhado e o símbolo que é empilhado.

Exemplos:

Obs.: devido a limitação do software que fez o desenho, será considerado '@' como ' λ '

i) $L = \{ a^n b^n \ ; \ i \geq 0 \}$

$$S \rightarrow aSb \mid \lambda$$



$$M = (\Sigma, Q, \Delta, \delta, q_0, F)$$

onde:

$$\Sigma = \{ a, b \}$$

$$Q = \{ q_0, q_1 \}$$

$$\Delta = \{ X \}$$

$$\delta: (q_0, a, \lambda) \rightarrow (q_0, X)$$

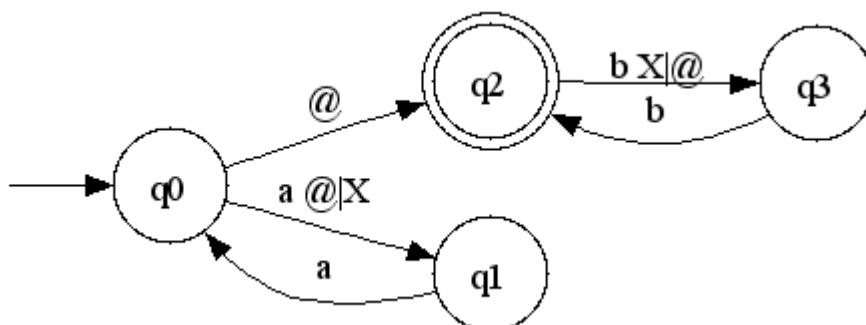
$$\delta: (q_0, \lambda, \lambda) \rightarrow (q_1, \lambda)$$

$$\delta: (q_1, b, X) \rightarrow (q_1, \lambda)$$

$$F = \{ q_1 \}$$

ii) $L = \{ a^{2n} b^{2n} \ ; \ n \geq 0 \}$

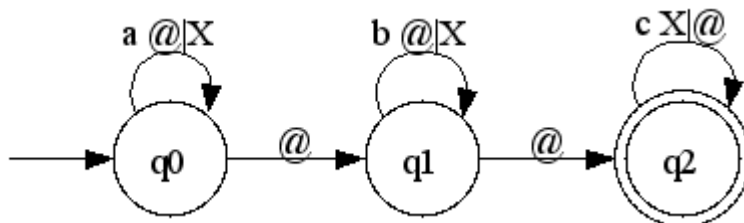
$$S \rightarrow aaSbb \mid \lambda$$



iii) $L = \{ a^n b^k c^i \ ; \ i = n + k \}$

$$S \rightarrow aSc \mid A$$

$$A \rightarrow bAc \mid \lambda$$

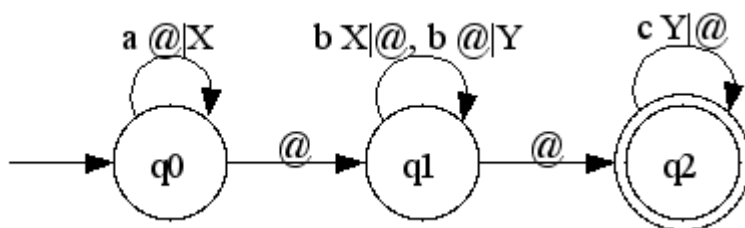


iv) $L = \{ a^n b^k c^i \ ; \ k = n + i \}$

$$S \rightarrow AB$$

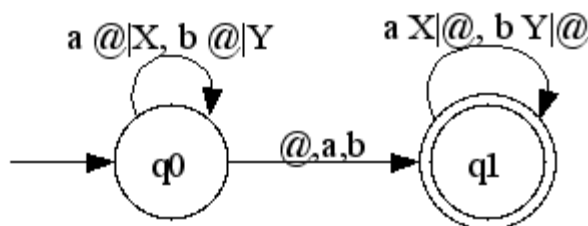
$$A \rightarrow aAb \mid \lambda$$

$$B \rightarrow bBc \mid \lambda$$



v) $L = \text{palíndromos sobre } \{a,b\}$

$$S \rightarrow aSa \mid bSb \mid a \mid b \mid \lambda$$



53. Aplicação da Forma Normal de Greibach: construção de Autômato com Pilha

A partir de uma gramática livre de contexto na FNG, que gera a linguagem L , constrói-se com relativa facilidade um autômato com pilha que reconhece a linguagem L . O autômato com pilha gerado pode depois ser usado, por exemplo, para a construção de analisadores sintáticos.

Sugere-se como trabalho da disciplina o estudo e a implementação deste algoritmo, capaz de receber uma gramática na FNG e fornecer como saída o autômato com pilha equivalente.

54. Lema do Bombeamento para Linguagens Livres de Contexto

Motivação:

Como descobrir que uma determinada linguagem é livre de contexto ou não? Sabemos que se for apresentado um autômato com pilha que a reconheça ou uma gramática livre de contexto que a gere então esta linguagem é livre de contexto.

Por outro lado, para provar que ela não é livre de contexto, usaremos o Lema do Bombeamento como explicado a seguir.

Se uma linguagem é livre de contexto (LLC) então atende ao lema do bombeamento (LB): **LLC \rightarrow LB**

Pela lógica proposicional podemos representar:

LLC \rightarrow LB	ok
LB \rightarrow LLC	não podemos afirmar
\sim LLC \rightarrow \sim LB	não podemos afirmar
\simLB \rightarrow \simLLC	ok , pela equivalência lógica da contraposição

Assim, devido a contraposição, conseguimos transformar o lema numa ferramenta valiosa para provar que uma linguagem não é livre de contexto, bastando para isto, mostrar que ela não atende ao lema do bombeamento.

Lema:

Se **L** é linguagem livre de contexto, então $\exists p \geq 1 ; \forall w \in L, |w| \geq p$ onde $w = uxvyz$ com $|xy| \geq 1$ e $|xvy| \leq p$ e $\forall i \geq 0, ux^i v y^i z \in L$

55. Aplicação do Lema do Bombeamento para Linguagens Livres e Contexto

Como já foi dito, o Lema do Bombeamento é bom para provar que uma determinada linguagem **L** não é livre de contexto. Isto nem sempre é possível, mas, de uma maneira geral, usamos o seguinte modelo de prova, pela técnica do absurdo:

1º) supomos **L** livre de contexto.

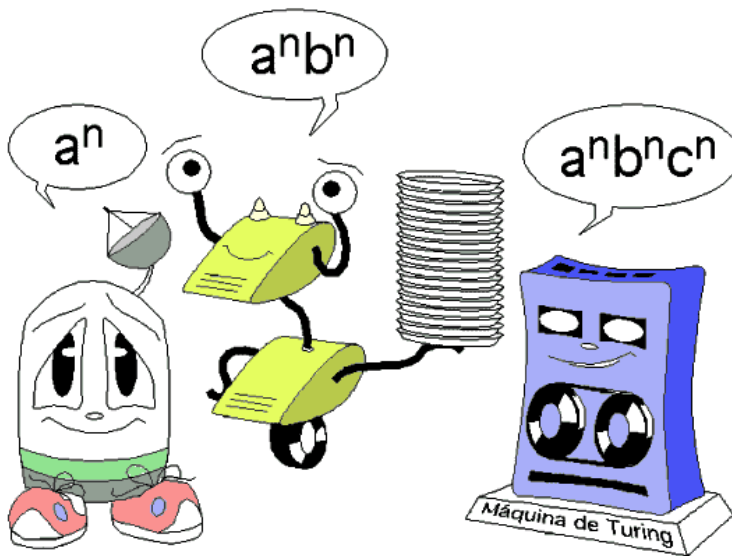
2º) procuramos uma palavra **w** que represente **L**, escrita em função de **p**, de tal forma que $|w| \geq p$ e seja formada por cinco partes **uxvyz** com $|xy| \geq 1$ e $|xvy| \leq p$, e de tal forma que quando **x** e **y** forem bombeados, isto é, escolhido um $i \geq 0$ para **uxⁱvyⁱz**, verificamos que **uxⁱvyⁱz** deixa de pertencer a **L**.

3º) assim, o lema não é satisfeito e, por absurdo, fica provado que **L** não é livre de contexto.

56. Linguagens Sensíveis ao Contexto

Na Hierarquia de Chomsky essas linguagens são de tipo 1.

Elas conseguem representar problemas mais complexos do que as de tipo 3 e 2.



Fonte: MENEZES, 2008

As abstrações usadas neste nível são: gramática sensível ao contexto como geradora e máquina de Turing como reconhecedora.

57. Máquina de Turing

É um dispositivo teórico, conhecido como *máquina universal*, que foi concebido pelo matemático britânico [Alan Turing](#) (1912-1954), muitos anos antes de existirem os modernos computadores digitais (o artigo de referência foi publicado em 1936). Num sentido preciso, é um modelo abstrato de um computador, que se restringe apenas aos aspectos lógicos do seu funcionamento (memória, estados e transições) e não à sua implementação física. Numa máquina de Turing pode-se modelar qualquer computador digital. (Wikipedia).

Na Hierarquia de Chomsky ela é usada para reconhecer linguagens sensíveis ao contexto (nível 1).

A **Tese de Church**, de acordo com as palavras do próprio Turing, pode ser enunciada como:

Toda 'função que seria naturalmente considerada computável' pode ser computada por uma Máquina de Turing.

Devido à imprecisão do conceito de uma "função que seria naturalmente considerada computável", a tese não pode ser nem provada nem refutada formalmente.

Qualquer programa de computador pode ser traduzido em uma máquina de Turing, e qualquer máquina de Turing pode ser traduzida para uma linguagem de programação de propósito geral; assim, a tese é equivalente a dizer que qualquer linguagem de programação de propósito geral é suficiente para expressar qualquer algoritmo.

A máquina de Turing é composta de um cabeçote gravador/leitor que opera sobre uma fita de entrada dividida em células sucessivas. Cada célula contém um único símbolo e elas são preenchidas a priori com espaço em branco.

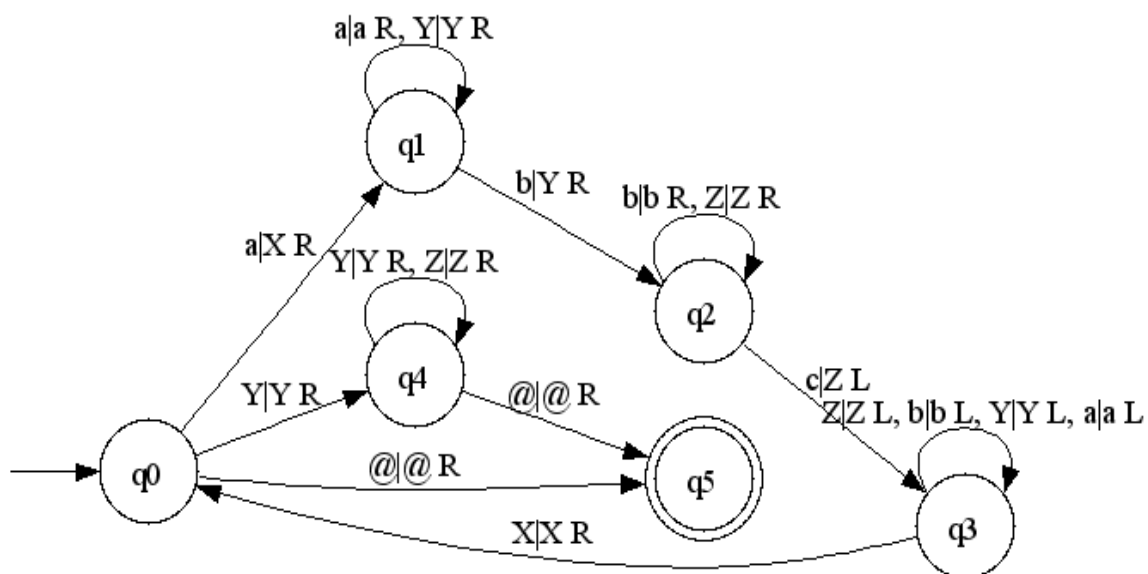
Cada computação sobre a máquina (transição no grafo) faz sempre:

- 1º. lê um símbolo na posição do cabeçote
- 2º. grava um símbolo na posição do cabeçote
- 3º. move o cabeçote para esquerda (L) ou direito (R).



Exemplos:

i) $L = \{ a^n b^n c^n \ ; \ i \geq 0 \}$



ii) $L = \{ w c w \ ; \ w \text{ pertence a } \{a,b\}^* \}$

Este exemplo é sugerido como exercício.

58. Gramática Sensível ao Contexto

Na Hierarquia de Chomsky, as linguagens sensíveis ao contexto são geradas por uma gramática sensível ao contexto.

A sensibilidade ao contexto é devido ao fato de permitir que o lado esquerdo de suas regras de produção seja formado por mais de um símbolo, gerando, assim, um contexto obrigatório para a substituição.

Ela é representada por produções do tipo:

$$\alpha A \beta \rightarrow \alpha' \gamma \beta'$$

onde: $A \in V$

$$\alpha, \beta, \alpha', \beta' \in (T \cup V)^*$$

$$\gamma \in (T \cup V)^+$$

ou:

$$S \rightarrow \lambda$$

desde que S não apareça no lado direito de nenhuma produção.

Obs.: algumas definições encontradas na literatura acrescentam a restrição do lado direito de uma regra não poder ser menor do que o seu lado esquerdo.

Exemplos:

i) $L = \{ a^n b^n c^n \ ; \ i \geq 1 \}$

Regra	Exemplo de geração pela aplicação da regra
$S \rightarrow aSBc$	aaSBcBc
$S \rightarrow abc$	aaabcBcBc
$cB \rightarrow Bc$	aaabBBccc
$bB \rightarrow bb$	aaabbbccc

Logo, a gramática é:

$$S \rightarrow aSBc \mid abc$$

$$cB \rightarrow Bc$$

$$bB \rightarrow bb$$

ii) $L = \{ w c w \ ; \ w \text{ pertence a } \{a,b\}^* \}$

$$S \rightarrow aAS \mid bBS \mid c$$

$$Aa \rightarrow aA$$

$$Ab \rightarrow bA$$

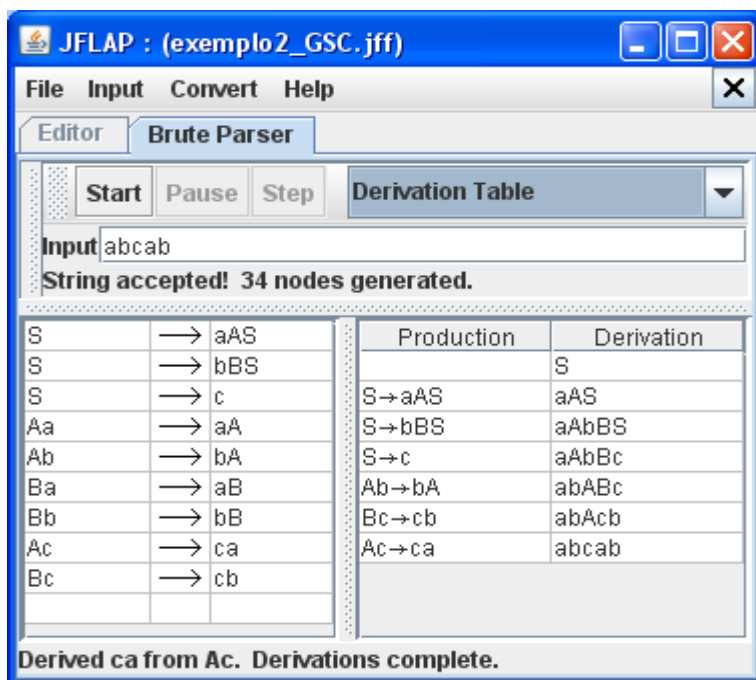
$$Ba \rightarrow aB$$

$$Bb \rightarrow bB$$

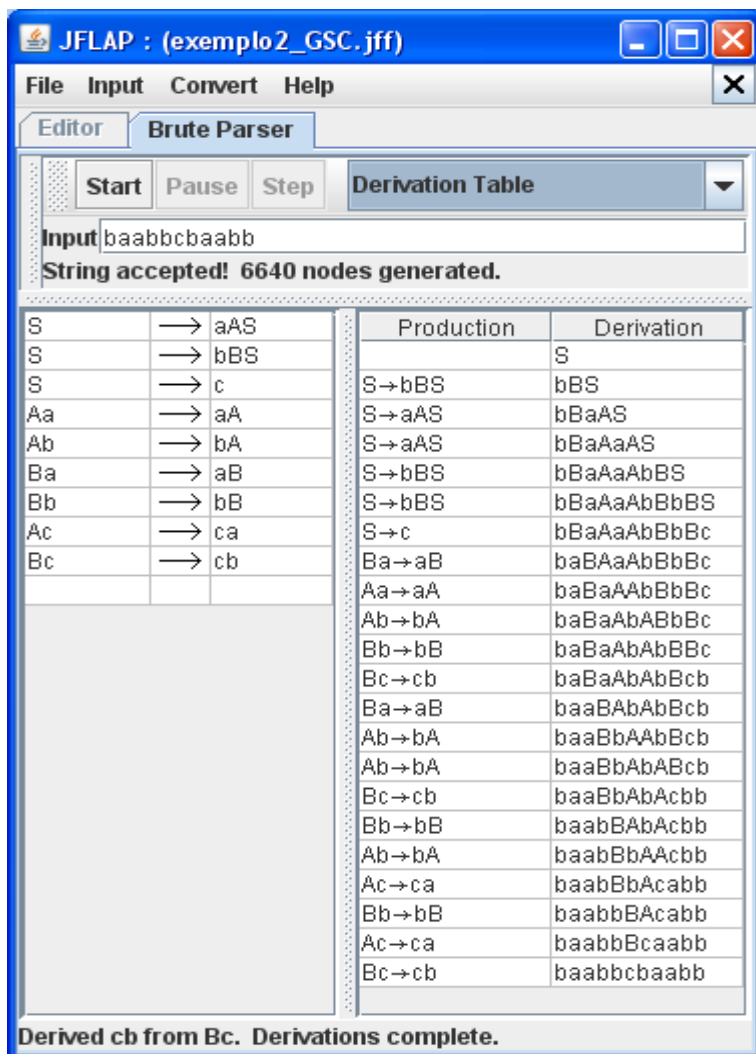
$$Ac \rightarrow ca$$

$$Bc \rightarrow cb$$

Conferindo no simulador JFLAP:



Outro teste:



59. Linguagens Enumeráveis Recursivamente

Uma linguagem que é aceita por uma Máquina de Turing é dita como uma Linguagem Enumerável

Recursivamente, que é do tipo 0. Lembre-se que o conjunto das linguagens de nível 0 contém as linguagens de nível 1 que contém as de nível 2 e que, finalmente, contém as de nível 3.

Se existe uma Máquina de Turing que aceita todas as strings de uma linguagem, e que não aceita as strings que não pertencem a linguagem, essa linguagem também é considerada Enumerável Recursivamente. Isto é, a parte do conjunto das de nível 0 que não contém as de nível 1.

É importante observar que ‘não aceita’ não é mesmo que ‘rejeita’, pois a Máquina de Turing poderia entrar num loop infinito e nunca parar para aceitar ou rejeitar a string.

Além da Máquina de Turing, que trabalha como reconhecedora desse tipo de linguagem, existem as **gramáticas irrestritas** que geram essas linguagens do nível 0.

60. Gramática Irrestrita

A **gramática irrestrita** é ainda mais flexível do que a gramática sensível ao contexto. Ela é representada por produções do tipo:

$$\alpha \rightarrow \beta$$

$$\text{onde: } \alpha \in (TUV)^+ \\ \beta \in (TUV)^*$$

Ou seja, as suas regras não possuem restrição, exceto pela exigência de pelo menos um símbolo (terminal ou não terminal) no lado esquerdo da regra de produção.

Obs.: a gramática irrestrita também é conhecida na literatura por gramática com estrutura de frase.

Exemplo:

$$L = \{ w w \ ; \ w \text{ pertence a } \{a,b\}^* \}$$

$$\begin{aligned} S &\rightarrow aAS \mid bBS \mid F \\ Aa &\rightarrow aA \\ Ab &\rightarrow bA \\ Ba &\rightarrow aB \\ Bb &\rightarrow bB \\ AF &\rightarrow Fa \\ BF &\rightarrow Fb \\ F &\rightarrow \lambda \end{aligned}$$

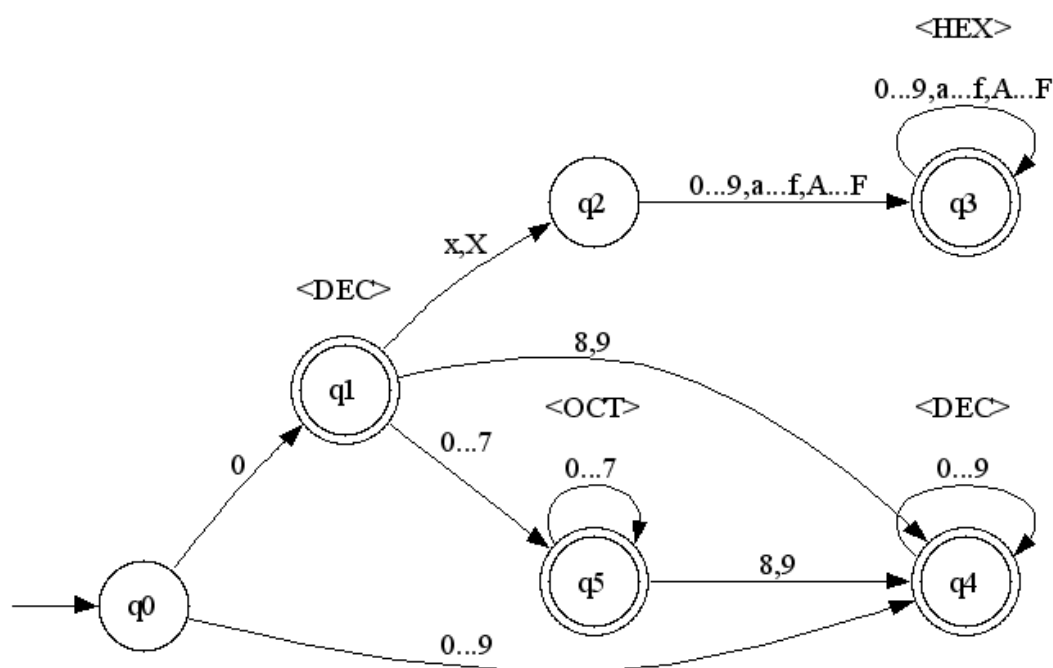
Obs.: vários outros elementos que dão continuidade a esse assunto serão estudados na disciplina Teoria da Computação, tais como outros tipos de Máquinas de Turing e seus formalismos, decidibilidade, o problema da parada e da indecidibilidade, etc.

APÊNDICE A: Exemplo de uso do Graphviz (Graph Visualization Software)

Instalação e uso:

- Faça o download em <http://www.graphviz.org/>
- Após a instalação execute o aplicativo “Gvedit”.
- Selecione “File” – “New”, escreva o código na janela em branco.
- Selecione o ícone “Settings”, e determine o tipo, o nome e o local do arquivo de saída.
Após “Ok”, aparecerá o desenho numa nova janela.
- Para compilar novamente basta usar o ícone “Layout”.

Exemplo de saída:



Código fonte para geração do desenho anterior:

```
/* Máquina de Moore para classificar números decimais, octais e  
haxadecimais (inteiros sem sinal)
```

```
ER de números hexadecimais: 0(X|x)(0|...|F)+
```

```
ER de números decimais: (0|...|9)+
```

```
ER de números octais: 0(0|...|7)+
```

São decimais: 0, 08, 058

```
*/
```

```
/* Obs.: aceita nomes dos estados grandes: basta colocá-lo entre aspas */
```

```
digraph AFD {  
    rankdir=LR;  
    size="6"  
    node [shape = circle];  
  
    /* determinação dos estados de classificação (estados  
    finais) da máquina de Moore */  
    subgraph cluster1 { /* numere sequencialmente... */  
        node [shape = doublecircle];  
        q1          /* mude esta linha: nome do estado */  
        label = "<DEC>"; /* mude esta linha: descrição do token*/  
        color=white  
    }  
    subgraph cluster2 {  
        node [shape = doublecircle];  
        q3          /* mude esta linha: nome do estado */  
        label = "<HEX>"; /* mude esta linha: descrição do token*/  
        color=white  
    }  
    subgraph cluster3 {  
        node [shape = doublecircle];  
        q4          /* mude esta linha: nome do estado */  
        label = "<DEC>"; /* mude esta linha: descrição do token*/  
        color=white  
    }  
    subgraph cluster4 {  
        node [shape = doublecircle];  
        q5          /* mude esta linha: nome do estado */  
        label = "<OCT>"; /* mude esta linha: descrição do token*/  
        color=white  
    }  
}
```

```

/* Nesta seção escreva todas as transições... */
q0 -> q1 [ label = "0" ];
q1 -> q2 [ label = "x,X" ];
q2 -> q3 -> q3 [ label = "0...9,a...f,A...F" ];
q0 -> q4 -> q4 [ label = "0...9" ];
q1 -> q5 -> q5 [ label = "0...7" ];
q1 -> q4 [ label = "8,9" ];
q5 -> q4 [ label = "8,9" ];

/* determine aqui o estado inicial */
node [shape = none, label=""];
s -> q0 ;

}

```

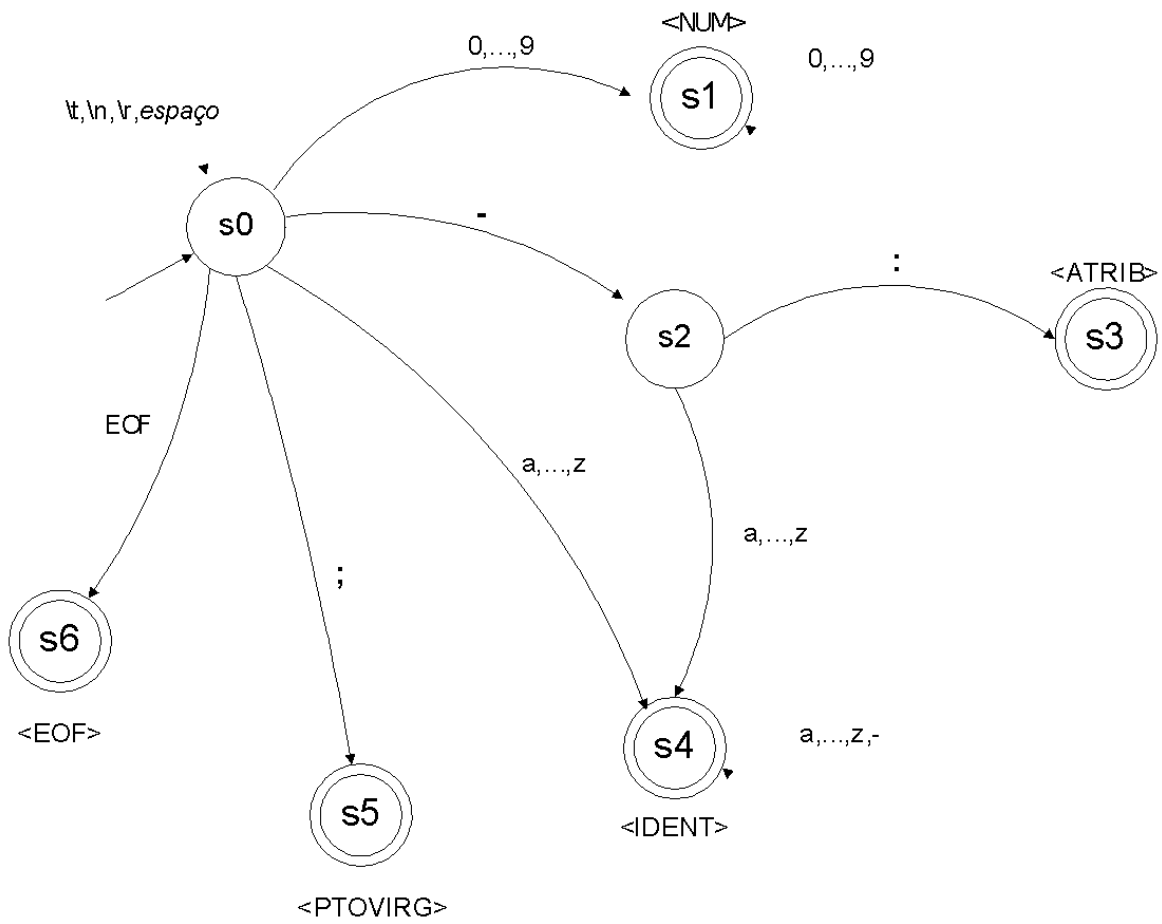
APÊNDICE B: Exemplo de um Analisador Léxico construído através da Análise Preditiva

Suponha uma linguagem de programação composta somente de comandos de atribuição cujo lado direito é formado por um número inteiro sem sinal ou uma variável (composta por letras minúsculas e hífens, sendo que se o hífen for o primeiro caractere então deve ser acompanhado por uma letra na segunda posição)

Expressões regulares para os tokens:

Token	ER	Exemplos de lexemas
<NUM>	$(0 \dots 9)^+$	0, 25, 999, 0005
<ATRIB>	$- :$	$- :$
<IDENT>	$(- \lambda) (a \dots z) (a \dots z -)^*$	x, x-, -a, -a---, def-cod, x-a
<PTOVIRG> >	$;$	$;$
<EOF>	<i>caracter que representa o fim de arquivo</i>	

Máquina de Moore:



Testes do analisador léxico – implementação da máquina de Moore:

(execute o arquivo “TesteAnalisadorLexico.java” do exemplo)

Teste 1:

```
x -: 5;
valor -: valor-total ; -soma -: total ;
```

Console de saída:

```
IDENT
ATRIB
NUM
PTOVIRG
IDENT
ATRIB
IDENT
PTOVIRG
IDENT
ATRIB
IDENT
PTOVIRG
EOF
```

Análise léxica realizada com sucesso no arquivo entrada.txt

Teste 2:

```
x -: 5;
valor -: total;
```

Console de saída:

```
IDENT
Erro léxico: caractere encontrado: :
era(m) esperado(s): 0123456789abcdefghijklmnopqrstuvwxyz;-
```

Obs.: após a classificação de “x-”, como não existe previsão para “:” em S4, a máquina de Moore é iniciada novamente em S0 para classificar o próximo token.

Teste 3:

```
x -: 5;
valor - total;
```

Console de saída:

```
IDENT
ATRIB
NUM
PTOVIRG
IDENT
Erro léxico: caractere encontrado:
era(m) esperado(s): abcdefghijklmnopqrstuvwxyz:
```

Teste 4:

```
x -; 5;
```

Console de saída:

```
IDENT
Erro léxico: caractere encontrado: ;
era(m) esperado(s): abcdefghijklmnopqrstuvwxyz:
```

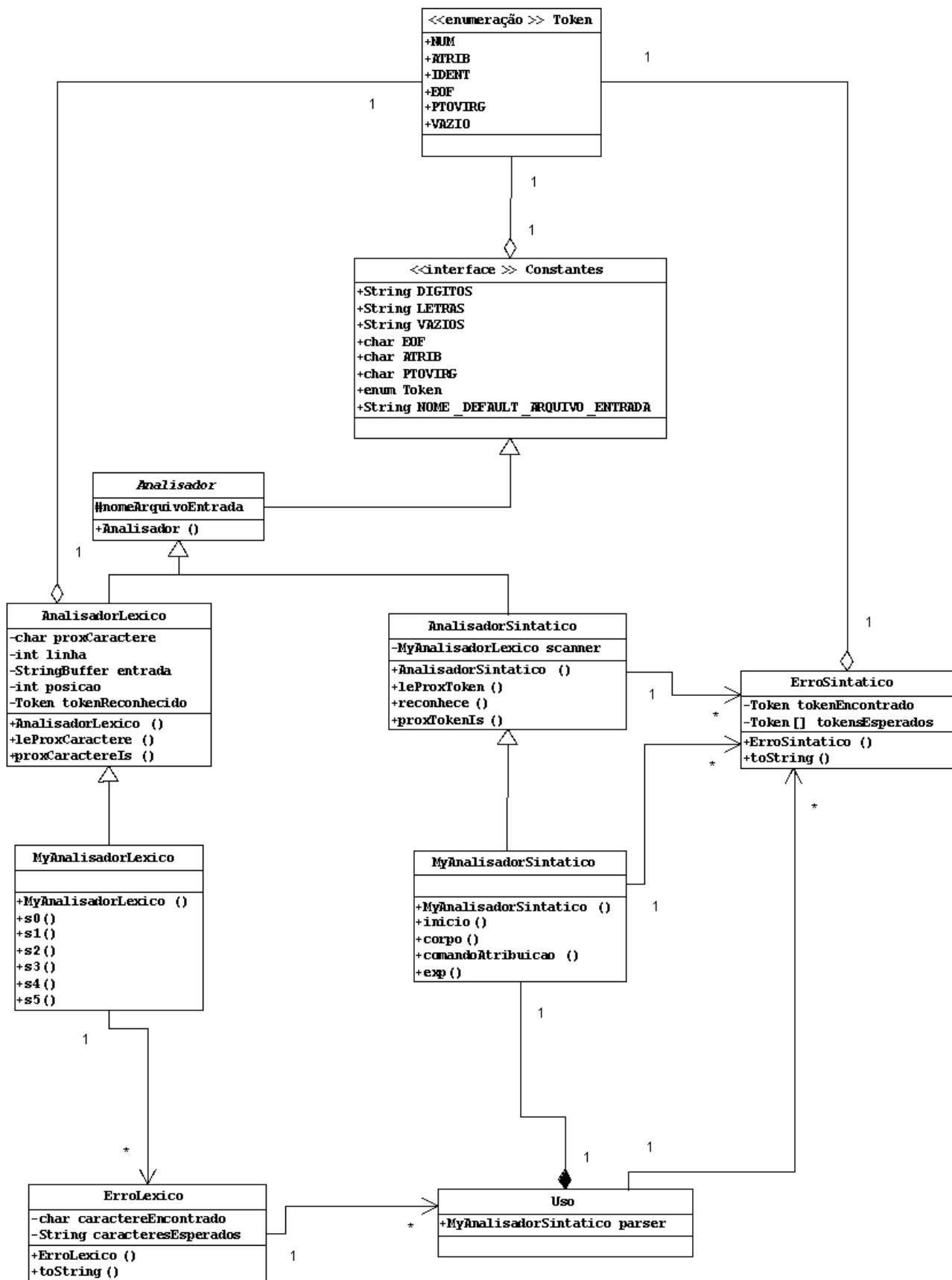
Teste 5:

```
valor@2 -: 5;
```

Console de saída:

```
IDENT
Erro léxico: caractere encontrado: @
era(m) esperado(s): abcdefghijklmnopqrstuvwxyz-0123456789;<EOF>
```


Diagrama UML de classes:



Implementação do analisador léxico por intermédio da análise preditiva:

Obs.: para rodar este exemplo, separe cada classe num arquivo, pois elas são **public**.

```
//-----  
public interface Constantes {  
  
    enum Token { NUM, ATRIB, IDENT, EOF, PTOVIRG, VAZIO };  
  
    String          DIGITOS= "0123456789",  
                   LETRAS  = "abcdefghijklmnopqrstuvwxyz",  
                   ATRIB   = "-:",  
                   VAZIOS  = " \r\n\t";  
  
    char  EOF          = 0,  
         HIFEN        = '-',  
         PTOVIRG      = ';',  
         DOISPONTOS    = ':';  
  
    String  NOME_DEFAULT_ARQUIVO_ENTRADA = "entrada.txt";  
}  
  
  
//-----  
public class ErroLexico extends RuntimeException {  
    private char caractereEncontrado;  
    private String caracteresEsperados;  
  
    public ErroLexico(char _caracterEncontrado, String _caracteresEsperados) {  
        this.caractereEncontrado = _caracterEncontrado;  
        this.caracteresEsperados = _caracteresEsperados;  
    }  
    public String toString() {  
        return "caractere encontrado: " + ((char) this.caractereEncontrado) +  
            "\nera(m) esperado(s): " + this.caracteresEsperados;  
    }  
}  
  
  
//-----  
public abstract class Analisador implements Constantes {  
    protected String nomeArquivoEntrada;  
    public Analisador(String _nomeArquivoEntrada) {  
        this.nomeArquivoEntrada = _nomeArquivoEntrada;  
    }  
    public Analisador() {  
        this.nomeArquivoEntrada = NOME_DEFAULT_ARQUIVO_ENTRADA;  
    }  
}
```

```
//-----
import java.io.FileReader;
import java.io.IOException;

public class AnalisadorLexico extends Analisador {
    protected char proxCaractere; // caractere disponível no cabeçote de leitura
    protected int linha = 1; // linha atual do arquivo fonte
    protected StringBuffer entrada = new StringBuffer(); // armazena o conteúdo do arquivo
    protected int posicao = 0; // posição do caractere a ser lido na entrada
    protected Token tokenReconhecido; // último token lido

    // transfere o arquivo para o buffer 'entrada'
    public AnalisadorLexico(String _nomeArquivoEntrada) {
        super(_nomeArquivoEntrada);
        try {
            FileReader file = new FileReader(_nomeArquivoEntrada);
            int c;
            while((c = file.read()) != -1) {
                this.entrada.append((char)c);
            }
            file.close();
            leProxCaractere();
        }
        catch (IOException e) {
            throw new RuntimeException("Erro de leitura no arquivo "+_nomeArquivoEntrada);
        }
    }

    // lê o próximo caractere do buffer. Se fim, retorna EOF
    // avança o ponteiro de leitura 1 posição
    public void leProxCaractere() {
        try {
            this.proxCaractere = this.entrada.charAt(this.posicao++);
        }
        catch (IndexOutOfBoundsException e) {
            this.proxCaractere = EOF;
        }
    }

    // verifica se o próximo caractere é um dos que estão em 's'
    // NÃO avança o ponteiro de leitura
    public boolean proxCaractereIs(String s) {
        if (s.indexOf(this.proxCaractere) != -1)
            return true;
        else
            return false;
    }
}

```

```
//-----
import java.io.IOException;

public class MyAnalizadorLexico extends AnalizadorLexico {
    public MyAnalizadorLexico(String _nomeArquivoEntrada) {
        super(_nomeArquivoEntrada);
    }
    public void s0() {
        if(this.proxCaractereIs(DIGITOS)) {
            leProxCaractere();
            s1();
        }
        else if(this.proxCaractere == HIFEN) {
            leProxCaractere();
            s2();
        }
        else if(this.proxCaractereIs(LETRAS)) {
            leProxCaractere();
            s4();
        }
        else if(this.proxCaractere == PTOVIRG) {
            leProxCaractere();
            s5();
        }
        else if(this.proxCaractere == EOF) {
            leProxCaractere();
            s6();
        }
        else if(this.proxCaractereIs(VAZIOS)) {
            leProxCaractere();
            s0();
        }
        else {
            throw new ErroLexico(this.proxCaractere, DIGITOS+LETRAS+VAZIOS+PTOVIRG+HIFEN);
        }
    }
    public void s1() {
        this.tokenReconhecido = Token.NUM;
        if(this.proxCaractereIs(DIGITOS)) {
            leProxCaractere();
            s1();
        }
    }
    public void s2() {
        if(this.proxCaractere == DOISPONTOS) {
            leProxCaractere();
            s3();
        }
        else if(this.proxCaractereIs(LETRAS)) {
            leProxCaractere();
            s4();
        }
        else
            throw new ErroLexico(this.proxCaractere, LETRAS+DOISPONTOS);
    }
    public void s3() {
        this.tokenReconhecido = Token.ATRIB;
    }
    public void s4() {
        this.tokenReconhecido = Token.IDENT;
        if(this.proxCaractereIs(LETRAS+HIFEN)) {
            leProxCaractere();
            s4();
        }
    }
    public void s5() {
```

```

        this.tokenReconhecido = Token.PTOVIRG;
    }
    public void s6() {
        this.tokenReconhecido = Token.EOF;
    }
}

//-----
public class TesteAnalizadorLexico {
    static public MyAnalizadorLexico scanner;
    public static void main(String[] args) {
        try {
            if(args.length != 1)
                throw new RuntimeException("esqueceu de escrever o nome do arquivo
                de entrada! \n" + "No Eclipse insira em: Run - Open Run Dialog
                - Arguments");
            scanner = new MyAnalizadorLexico(args[0]);

            // chama a máquina de Moore várias vezes até encontrar o fim de arquivo
            do {
                scanner.s0();
                System.out.println(scanner.tokenReconhecido);
            }
            while(scanner.tokenReconhecido != Constantes.Token.EOF);

            System.out.println("Análise lexica realizada com sucesso
            no arquivo "+scanner.nomeArquivoEntrada);
        }
        catch(ErroLexico e) {
            System.out.println("Erro léxico: "+e.toString());
        }
        catch(RuntimeException e) {
            System.out.println("Erro: "+e.getMessage());
        }
    }
}

```

APÊNDICE C: Exemplo de um Analisador Sintático construído através da Análise Preditiva

O analisador iniciado no apêndice B, através da construção da análise léxica, será agora complementado com o analisador sintático, por meio da análise preditiva classificada como uma técnica de reconhecimento top-down.

Relembrando, o exemplo é uma linguagem composta somente de comandos de atribuição cujo lado direito é formado por um número inteiro sem sinal ou uma variável (composta por letras minúsculas e hífens, sendo que se o hífen for o primeiro caractere então deve ser acompanhado por uma letra na segunda posição)

Gramática livre de contexto:

inicio	-> corpo <EOF>
corpo	-> comandoAtribuicao <PTOVIRG> corpo λ
comandoAtribuicao	-> <IDENT> <ATRIB> exp
exp	-> <NUM> <IDENT>

Obs.: esta gramática já está fatorada (sem conflitos à esquerda) e também não possui recursividade à esquerda.

Testes dos analisadores léxico e sintático integrados:

(execute o arquivo “Uso.java” do exemplo)

Teste 1:

```
x -: 5;
valor-mensal -: 123; s -: -total ;
```

Console de saída:

Análise realizada com sucesso no arquivo entrada.txt

Teste 2:

```
x -:
valor -: 123; soma -: total ;
```

Console de saída:

Erro sintático: token encontrado: IDENT
era(m) esperado(s): PTOVIRG

Teste 3:

```
X 5;
valor -: 123; soma -: total ;
```

Console de saída:

Erro sintático: token encontrado: NUM
era(m) esperado(s): ATRIB

Teste 4:

```
x -: 5;
valor -: 123;;soma -: total ;
```

Console de saída:

Erro sintático: token encontrado: PTOVIRG
era(m) esperado(s): IDENT EOF

Teste 5:

```
x -: 5;
valor -: 123; soma -: ;
```

Console de saída:

Erro sintático: token encontrado: PTOVIRG
era(m) esperado(s): NUM IDENT

Teste 6:

```
x -: 5#;  
valor -: 123; soma -: total ;
```

Console de saída:

Erro léxico: caractere encontrado: #
era(m) esperado(s): 0123456789;

Teste 7:

```
x -: 5;  
valor -: @ ; soma -: total ;
```

Console de saída:

Erro léxico: caractere encontrado: @
era(m) esperado(s): 0123456789abcdefghijklmnopqrstuvwxyz-

Implementação por intermédio da análise preditiva:

Obs.: basta inserir estes arquivos no código do exemplo do analisador léxico.

```
//-----  
public class ErroSintatico extends RuntimeException implements Constantes {  
    private Token tokenEncontrado;  
    private Token[] tokensEsperados;  
  
    public ErroSintatico(Token _tokenEncontrado, Token[] _tokensEsperados) {  
        this.tokenEncontrado = _tokenEncontrado;  
        this.tokensEsperados = _tokensEsperados;  
    }  
    public ErroSintatico(Token _tokenEncontrado, Token _tokenEsperado) {  
        this.tokenEncontrado = _tokenEncontrado;  
        this.tokensEsperados = new Token[1];  
        tokensEsperados[0] = _tokenEsperado;  
    }  
    public String toString() {  
        String listaDeTokensEsperados = "";  
        for(int i=0; i<this.tokensEsperados.length; i++)  
            listaDeTokensEsperados += this.tokensEsperados[i] + " ";  
        return "token encontrado: "+this.tokenEncontrado+  
            "\nera(m) esperado(s): "+listaDeTokensEsperados;  
    }  
}  
  
//-----  
public class AnalisadorSintatico extends Analisador implements Constantes {  
    protected MyAnalisadorLexico scanner;  
  
    public AnalisadorSintatico(String _nomeArquivoEntrada) {  
        this.scanner = new MyAnalisadorLexico(_nomeArquivoEntrada);  
        // lê o primeiro token e o coloca no campo tokenReconhecido  
        this.leProxToken();  
    }  
    public AnalisadorSintatico() {  
        super();  
    }  
}
```

```

// executa 1 vez a máquina de Moore
public void leProxToken() {
    this.scanner.s0();
}

// verifica se o próximo token é t
// avança o ponteiro para o próximo token
public void reconhece(Token t) {
    if(t == this.scanner.tokenReconhecido)
        this.leProxToken();
    else
        throw new ErroSintatico(this.scanner.tokenReconhecido, t);
}

// verifica se o próximo token é t
// NÃO avança o ponteiro de leitura
public boolean proxTokenIs(Token t) {
    if(t == this.scanner.tokenReconhecido)
        return true;
    else
        return false;
}
}

//-----
public class MyAnalizadorSintatico extends AnalizadorSintatico {

    public MyAnalizadorSintatico(String _nomeArquivoEntrada) {
        super(_nomeArquivoEntrada);
    }
    public void inicio() {
        corpo();
        reconhece(Token.EOF);
    }
    public void corpo() {
        if(proxTokenIs(Token.IDENT)) {
            comandoAtribuicao();
            reconhece(Token.PTOVIRG);
            corpo();
        }
        else if(proxTokenIs(Token.EOF))
            ;
        else {
            Token[] tokensEsperados = {Token.IDENT,Token.EOF};
            throw new ErroSintatico(this.scanner.tokenReconhecido,tokensEsperados);
        }
    }
    public void comandoAtribuicao() {
        reconhece(Token.IDENT);
        reconhece(Token.ATRIB);
        exp();
    }
    public void exp() {
        if(proxTokenIs(Token.NUM))
            leProxToken();
        else if(proxTokenIs(Token.IDENT))
            leProxToken();
        else {
            Token[] tokensEsperados = {Token.NUM,Token.IDENT};
            throw new ErroSintatico(this.scanner.tokenReconhecido,tokensEsperados);
        }
    }
}
}

```



```
//-----
public class Uso {
    static public MyAnalizadorSintatico parser;
    public static void main(String[] args) {
        try {
            if(args.length != 1)
                throw new RuntimeException("esqueceu de escrever o nome do arquivo
                    de entrada! \n" + "No Eclipse insira em: Run - Open Run Dialog
                    - Arguments");
            parser = new MyAnalizadorSintatico(args[0]);
            parser.inicio();
            System.out.println("Análise realizada com sucesso no
                arquivo "+parser.nomeArquivoEntrada);
        }
        catch (ErroLexico e) {
            System.out.println("Erro léxico: "+e.toString());
        }
        catch (ErroSintatico e) {
            System.out.println("Erro sintático: "+e.toString());
        }
        catch (RuntimeException e) {
            System.out.println("Erro: "+e.getMessage());
        }
    }
}
```