UFES – CEUNES - Engenharia de Computação e Ciência da Computação
Linguagens Formais e Autômatos – 2017/1
Prof. *Henrique Monteiro Cristovão*

Aviso: não imprima o documento inteiro, pois ele está sendo atualizado simultaneamente com as aulas dadas.

Notas de Aula

Este material é apenas um roteiro contento 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 em outro documento.

Há um foco para estudos de casos aplicados às linguagens de programação visando uma construção sólida de 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 da disciplina.

Tópicos: (clique para navegar)

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.

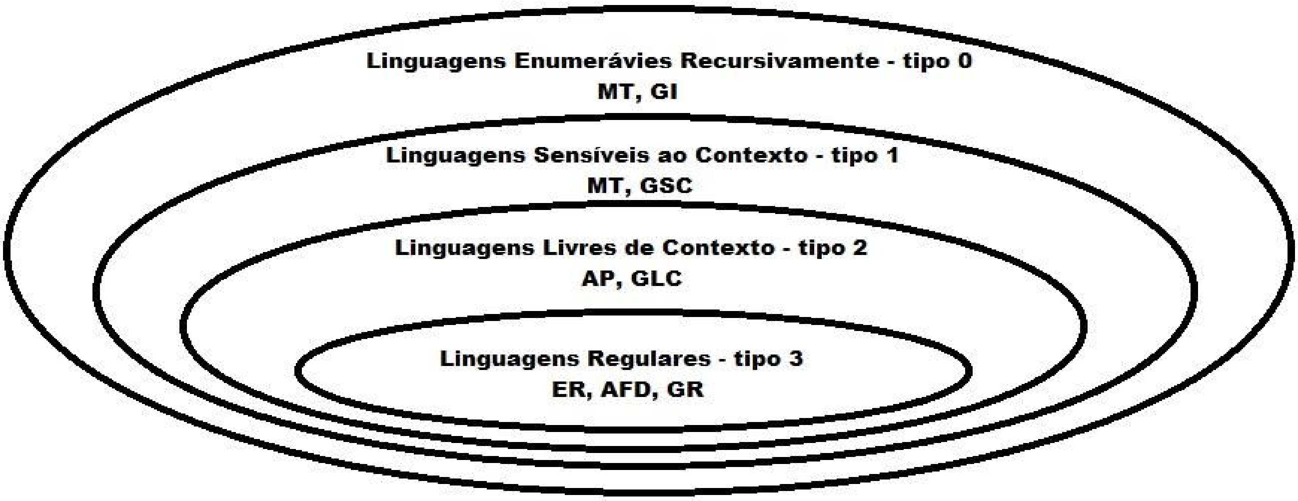
Massachusets: Addison-Wesley Publishing Company, Inc., 1997.

GERSTING, Judith L**. Fundamentos matemáticos para a ciência da computação**. 4. ed. Porto Alegre: Sagra Luzzatto, 2001.

LEWIS, Harry R.; PAPADIMITRIOU, Christos H. **Elementos de teoria da computação**. 2. ed. Porto Alegre: Artmed, 2000.

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

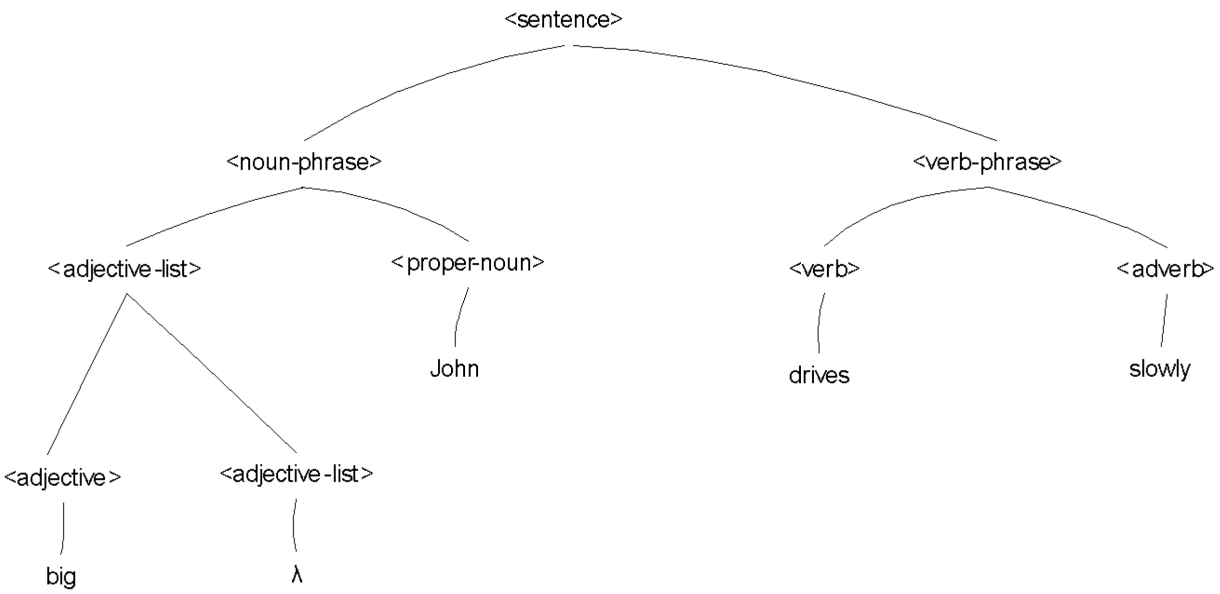
Derivação com notação resumida:

<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 gramátia 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, <sentence>)

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

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

P = { <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>
}

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, passando a responsabilidade do sinal para A e a responsabilidade do número inteiro sem sinal para B. Desta forma a expressão resultante ficou sendo a concatenação de A com B.

5. Conceitos importantes

- **Alfabeto**: é um conjunto finito de símbolos. Normalmente representado por $\sum$
- String, **palavra** ou cadeia de caracteres: é uma sequencia 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 qtde de símbolos que compõe a string. |w| representa o tamanho de w.
- **Prefixo**, **sufixo** e **subpalavra** (ou substring).
- **Concatenação**.
- **Concatenação sucessiva**: suponha w uma palavra de uma alfabeto e 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.
- 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 gega 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:

```
A -> αB
A -> α
```

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

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

Gramática transformada em regular:

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

Outra solução, com menos símbolos variáveis:

```
G = (V,T,P,S)
V = {S,A}
```

```
T = {+,-,0,1,2,3,4,5,6,7,8,9}
P = { S -> +A | -A | A
         A -> 0A|1A|2A|3A|4A|5A|6A|7A|8A|9A|0|1|2|3|4|5|6|7|8|9
      }
```

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 ∑ é indutivamente definida como:

a) {} é uma ER e representa a linguagem vazia;

b) λ é uma ER e representa a linguagem contendo a string vazia;

c) x Є ∑ é uma ER e representa a linguagem {x}

d) se r e s são ER e representam as linguagens R e S respectivamente, então:

d.1) (r | s) é ER e representa a linguagem R U S

Obs.: algumas literaturas usam: r + s, outras usam: r U s

d.2) (rs) é ER e representa a linguagem RS = {uv | u Є R e v Є S}

d.3) (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 ∑ = {a,b}

a^+	=	um ou vários a's concatenados { a, aa, aaa, aaaa, …}
a | b	=	a ou b { a, b } Obs.: algumas literaturas usam a notação: **a U b**, ou ainda: **a + b**
a*	=	zero, um ou vários a's concatenados { λ, a, aa, aaa, aaaa, …}
a?	=	{ λ, a }

Exemplos:
Observe exemplos de ER sobre o alfabeto ∑ = {a,b}, e suas respectivas linguagens.

ER	Descrição da linguagem, ou exemplos de strings
`a*`	{ λ, a, aa, aaa, aaaa, …}

`a+`	{ a, aa, aaa, aaaa, …}
`a?`	{ λ, a }
`a|b`	{ a, b }
`ab`	{ ab}
`(ab)*`	{ λ, ab, abab, ababab, ...} Obs.: ba, aab, aabb não pertencem à linguagem
`b+|a*`	{ λ, b, bb, bbb, a, aa, aaa, ...} 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: `(+|-|λ)(0|...|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, ...}

ER: `(+|-|λ)(0|...|9)*(.)?(0|...|9)+`

12. Autômato Finito Determinístico (AFD)

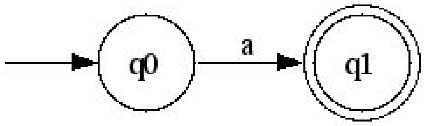
É um formalismo reconhecedor que tem como objetivo aceitar ou rejeitar strings dadas como entrada de uma determinada linguagem regular. Ou seja é uma máquina abstrata capaz de reconhecer a pertinência de strings para uma dada linguagem representada pelo autômato através de um grafo orientado.

<u>Definição</u>:
Um autômato finito determinístico (AFD) é uma 5-upla M = (∑, Q, δ, q0, F) onde:

- ∑ é o alfabeto dos símbolos de entrada
- Q é o conjunto dos estados possíveis
- δ é a função transição, $\delta: Q x \sum \rightarrow Q$
- q0 é o estado inicial
- F é o conjunto de estados finais. Deve possuir ao menos um elemento.

<u>Representação do AFD por intermédio de um grafo</u>:
O estado inicial é indicado por uma seta sem origem, e os estados finais são representados por dois círculos concêntricos, cada um.

<u>Exemplo 1</u>: AFD para aceitar a linguagem regular formada apenas por ‘a’.



M = (∑, Q, δ, q0, F)

∑ = {a}

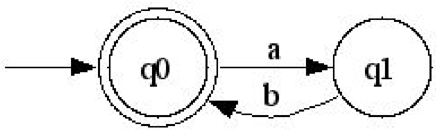
Q = {q0, q1}

δ - função transição:

δ	a
q0	q1
q1	-

F = {q1}

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



M = (∑, Q, δ, q0, F)

∑ = {a,b}

Q = {q0, q1}

δ - função transição:

δ	a	b
q0	q1	-
q1	-	q0

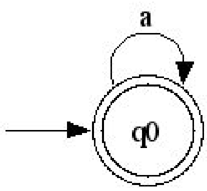
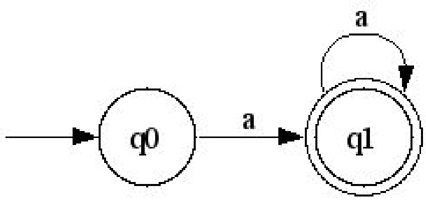
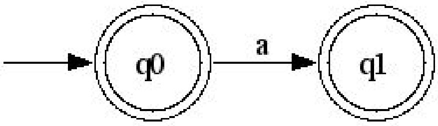
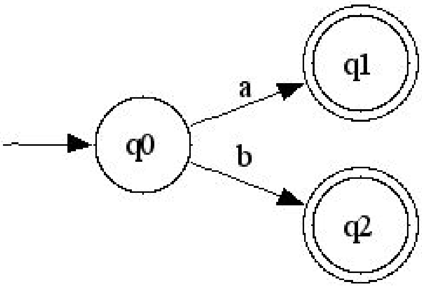
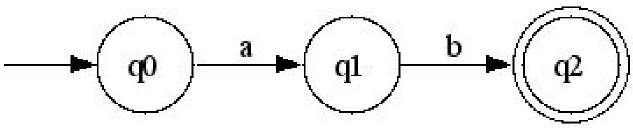
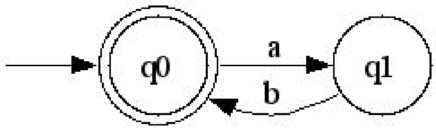
F = {q0}

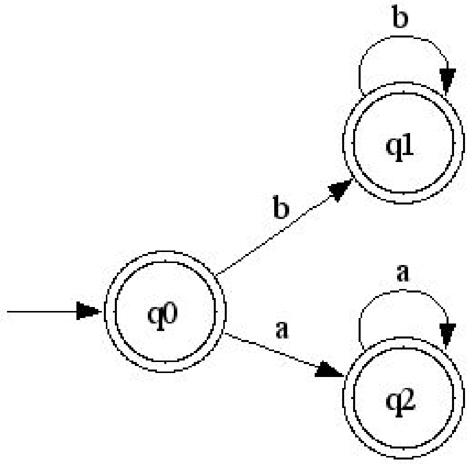
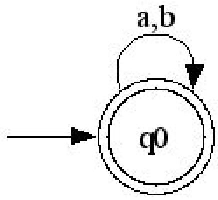
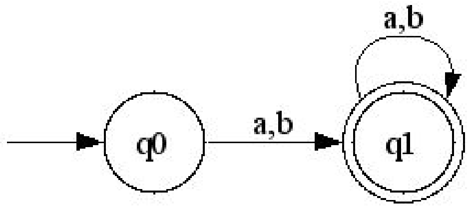
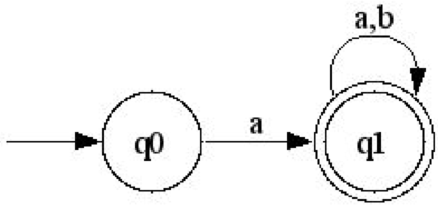
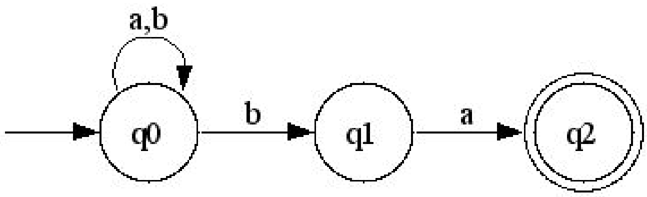
Algumas observações importantes sobre autômatos finitos:

- 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 (AFλ)**
- todo **AFND** e **AFλ** pode ser convertido em **AFD**.

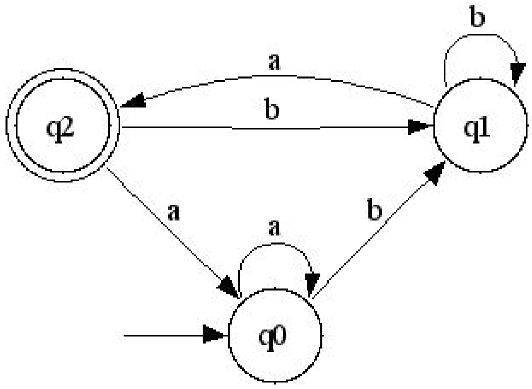
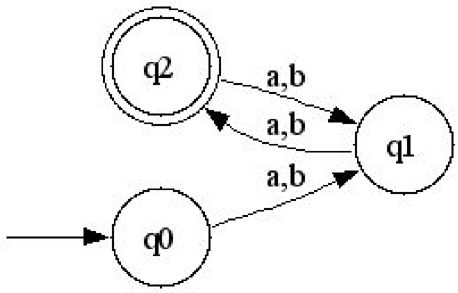
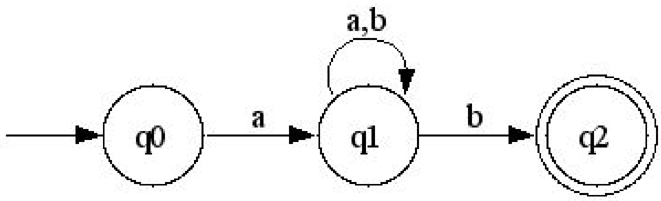
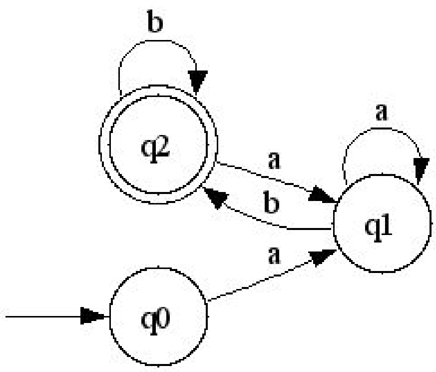
13. Exemplos de AFD, AFND e AFλ

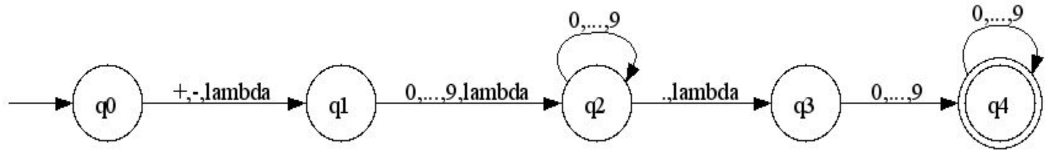
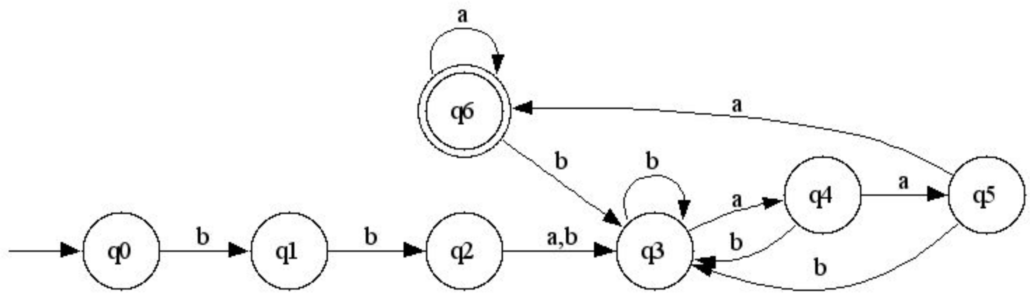
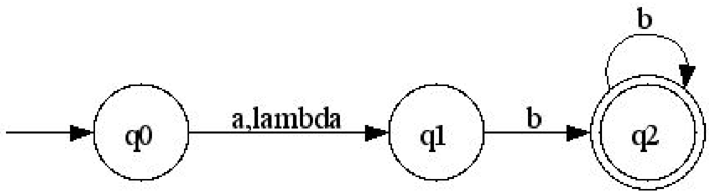
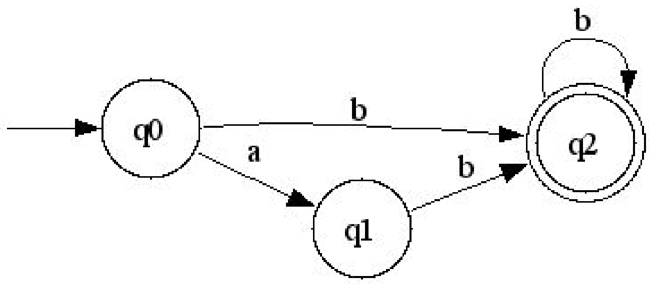
Observe os exemplos (já usados em ER) a a distinção entre AFD, AFND, AFλ.

ER	AFD, AFND, AFλ
a*	
a+	
a?	
a|b	
ab	
(ab)*	

<table>
<tr><td>b*|a*</td><td>
</td></tr>
<tr><td>(a|b)*</td><td>
</td></tr>
<tr><td>(a|b)+</td><td>
</td></tr>
<tr><td>a(a|b)*</td><td>
</td></tr>
<tr><td>(a|b)*ba</td><td>AFND:


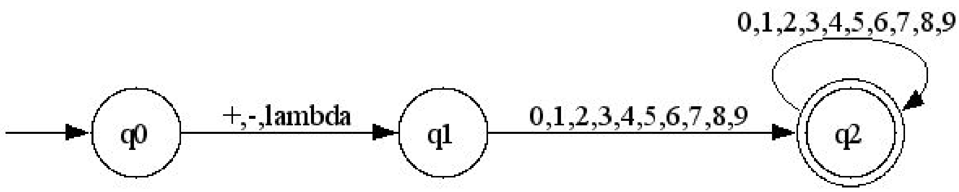
AFD:</td></tr>
</table>

	
((a|b)(a|b))+	
a(a|b)*b	AFND:  AFD: 

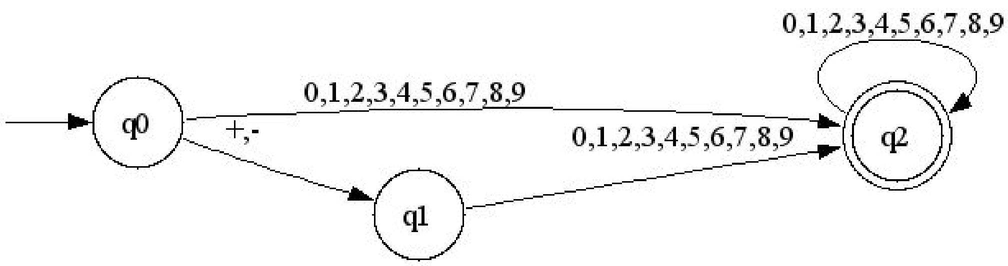
bb(a|b)+aaa	AFND:  AFD: 
a?b+	AFλ**:**  AFD: 

14. Estudo de caso 1: número inteiros

AFλ :



AFD:

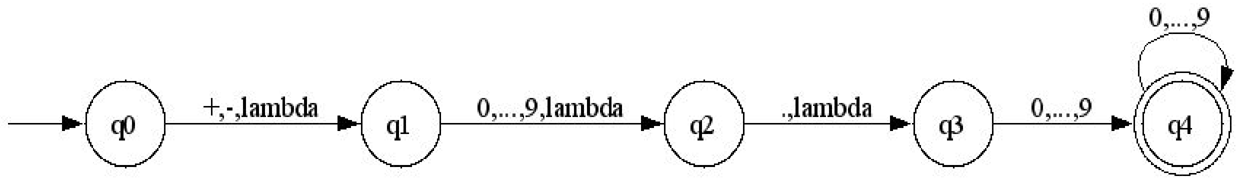


15. Estudo de caso 2: números reais

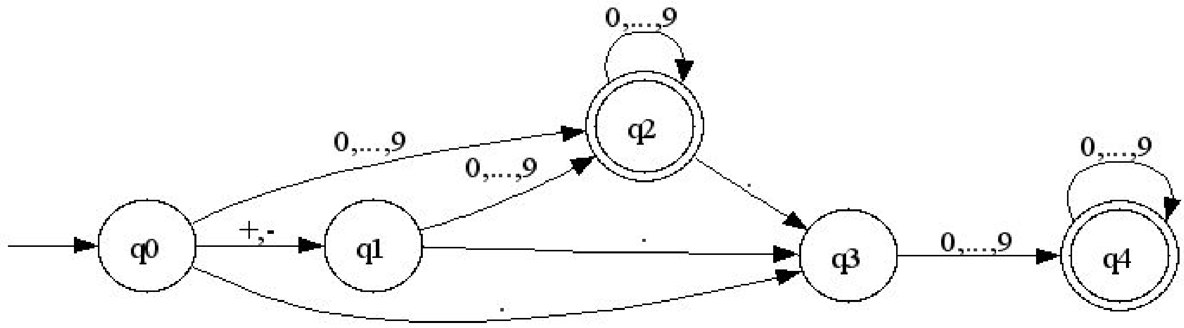
Exemplos de strings da linguagem: L= { 2.5, +2.5, -2.5, +2, -2, +0, 0, .5, -.5, ...}

ER: `(+|-|λ)(0|...|9)*(.)?(0|...|9)+`

AFλ:



AFD:



GR:

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

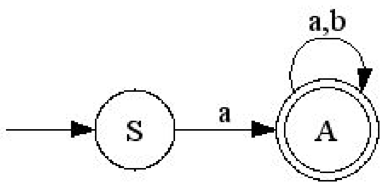
16. GR x AFD x ER

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 -> aA
        A -> aA | bA | λ
      }
```

Quando usamos um AFD, ao invés de um AFλ ou AFND, temos a garantia de obter uma gramática sem conflitos à esquerda. É considerado conflito em terminais á esquerda para uma mesma variável.

Exemplo de gramática com conflito à esquerda para a linguagem a(a*|b*)

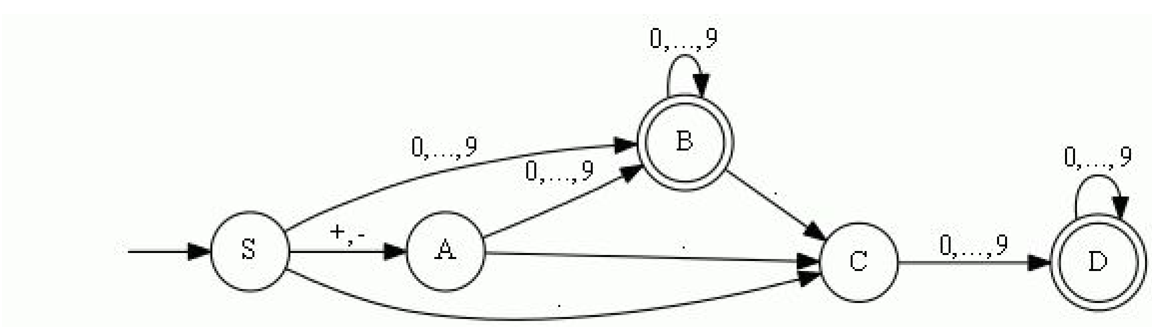
```
S -> aA | aB
A -> aA | λ
B -> bB | λ
```

Gramática sem conflito à esquerda (fatorada) para a mesma linguagem a(a*|b*)

```
S -> aC
C -> A | B
A -> aA | λ
B -> bB | λ
```

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)
```

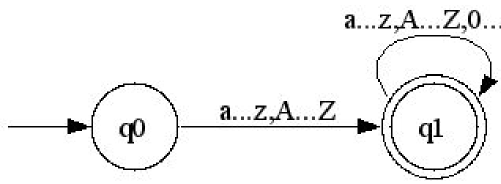
```
V = {S,A,B,C,D}

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, Valor, Salario13, a2b3, ...}

ER: `(a|...|Z)(0|...|9|a|...|Z)*`

AFD:



GR:

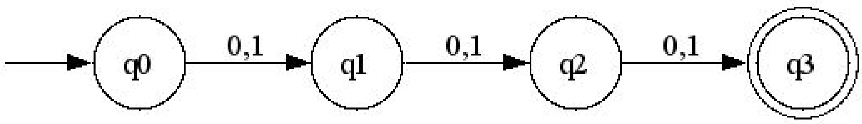
```
G = (V,T,P,<S>)

V = {<S>,<A>}

T = {0,...,9,a,...,z,A,...,Z}

P = {  <S> -> a<A> |...| Z<A>

       <A> -> a<A> |...| Z<A> | 0<A> |...| 9<A> | 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 -> 0A | 1A
       A -> 0B | 1B
       B -> 0 | 1
    }
```

ou, simplesmente:

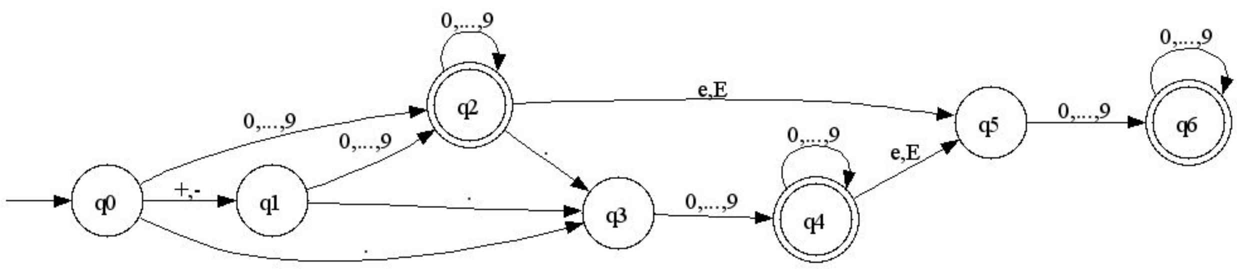
```
G = (V,T,P,S)
V = {S}
T = {0,1}
P = {  S ->  000, 001, 010, 011, 100, 101, 110, 111 }
```

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, ...}

ER: `(+|-)?(0|...|9)*(.)?(0|...|9)+ ((E|e)(+|-)?(0|...|9)+)?`

AFD:



GR:

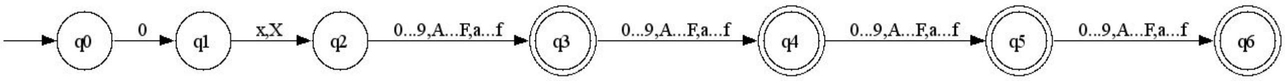
```
G = (V,T,P,<S>)
V = {<S>,<A>,<B>,<C>,<D>,<E>,<F>}
T = {+,-,.,0,1,2,3,4,5,6,7,8,9,e,E}
P = {  <S> -> +<A> | -<A> | .<C> | 0<B> |...| 9<B>
       <A> -> 0<B> |...| 9<B> | .<C>
       <B> -> .<C> | e<E> | E<E> | 0<B> |...| 9<B> | λ
       <C> -> 0<D> |...| 9<D>
       <D> -> 0<D> |...| 9<D> | λ | e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

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, ...}

ER: `0(X|x)(0|...|F)(0|...|F)?(0|...|F)?(0|...|F)?`

AFD:

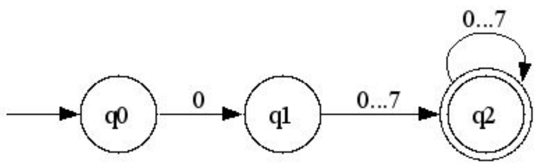


23. Estudo de caso 7: números octais em Java

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, ...}

ER: `0(0|...|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.
(iii) A função transição deve ser total.

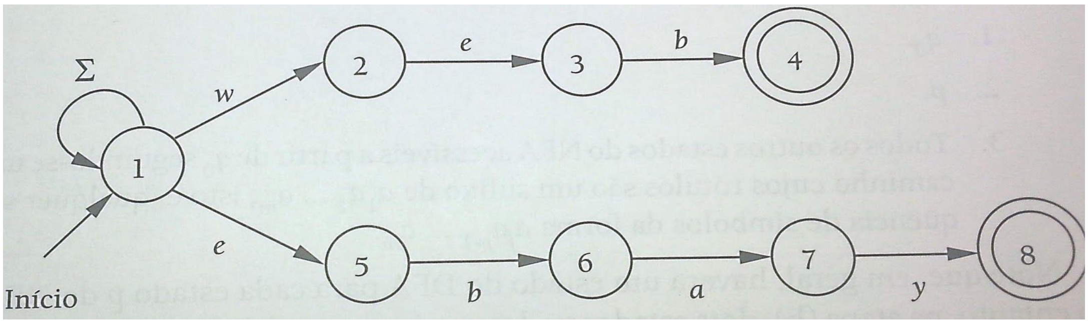
O estudo do algoritmo para a minimização bem como a sua implementação será avaliado como um dos trabalhos da disciplina.

26. Busca de texto: aplicação para um AFND

Uma boa aplicação de um AFND é a busca de palavras dentro de um texto, isto é, considerando que ele seja, primeiramente, transformado para um AFD equivalente. Este último é mais rápido devido a sua determinação na transição dos estados.

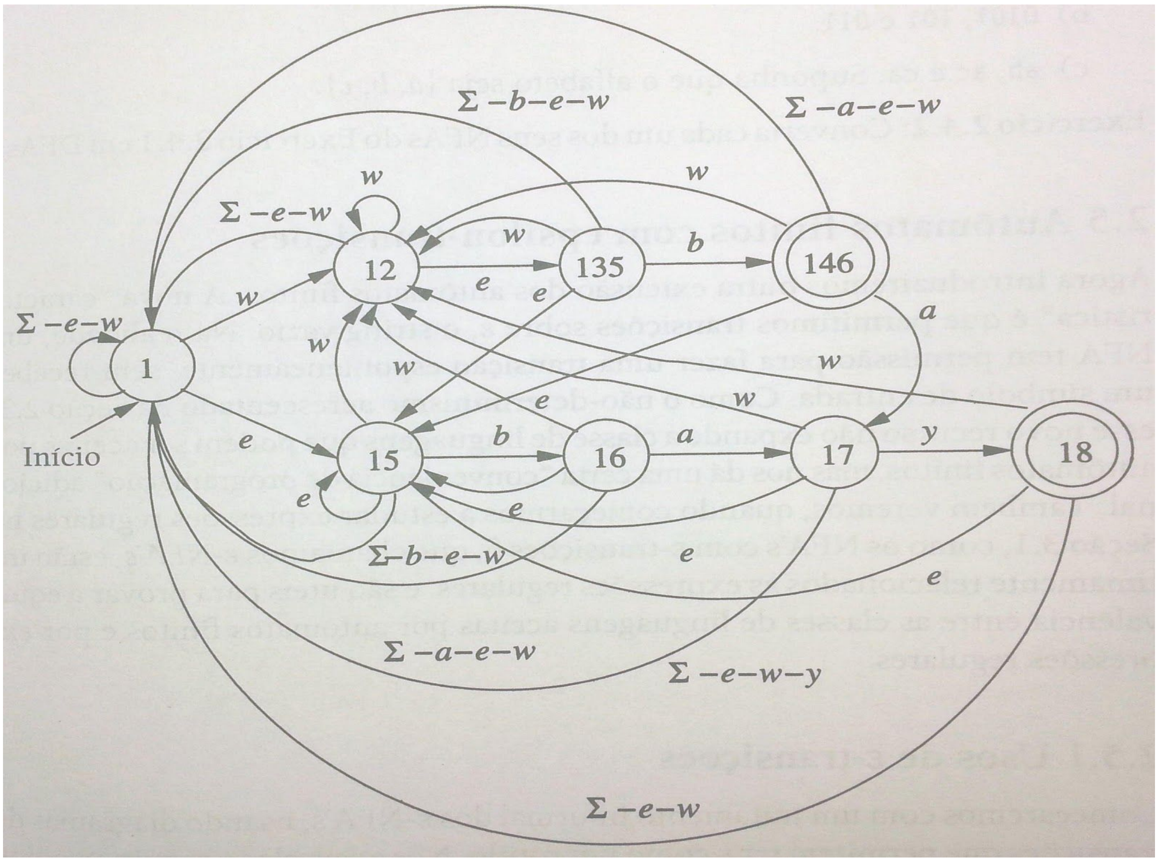
Obs.: este tipo de busca é apropriado quando a base de dados sofre modificações com frequência. Em outras situações, existem métodos de busca mais adequados, por exemplo, na internet, costuma-se usar a busca de índice invertido (os documentos são referenciados a partir da lista de palavras existentes) onde há eficiência na busca em detrimento do tempo na montagem dos índices. Já a busca feita pelo AFD não depende de índices, e o algoritmo da máquina é rapidamente montado.

O exemplo seguinte demonstra bem uma situação de busca feita a partir de um AFND: palavras "web" e "ebay" sobre um texto constituído de símbolos do alfabeto ∑.



Fonte: HOPCROFT, ULLMAN, MOTWANI, 2002.

O AFND é transformado para seu equivalente AFD:

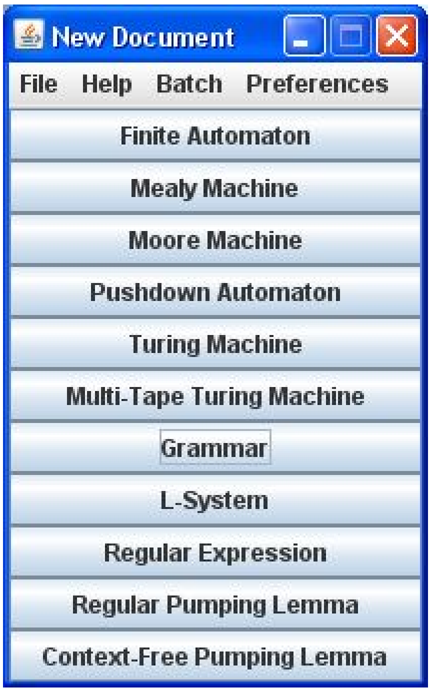


Fonte: HOPCROFT, ULLMAN, MOTWANI, 2002.

27. Uso do simulador JFLAP para Linguagens Regulares

Página principal do JFLAP: http://www.jflap.org/
Download: http://www.jflap.org/jflaptmp/
Tutorial: http://www.jflap.org/tutorial/

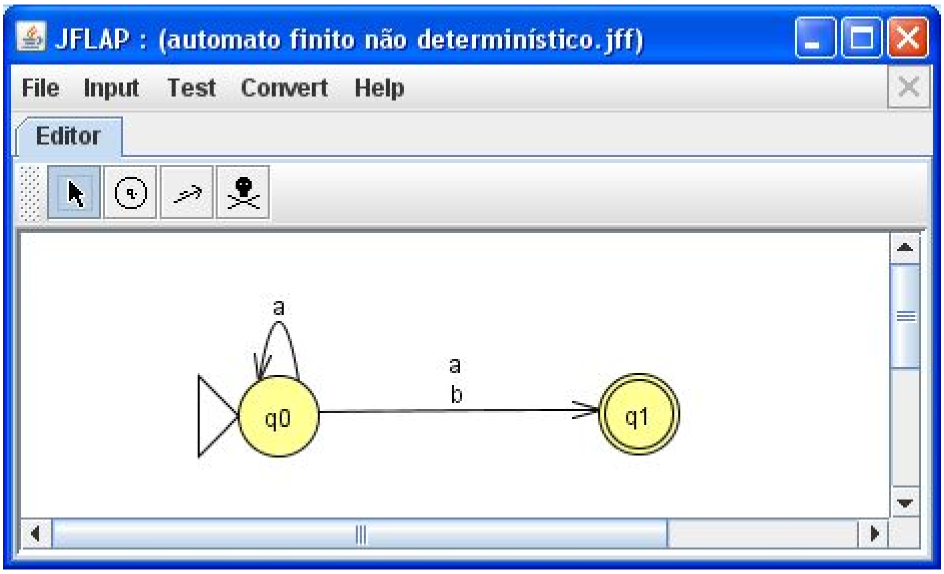
Execute diretamente o arquivo 'JFLAP.jar'



Autômato finito (determinístico e não determinístico)

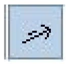
1- Selecione 'Finite Automaton'

2- Construa o grafo:

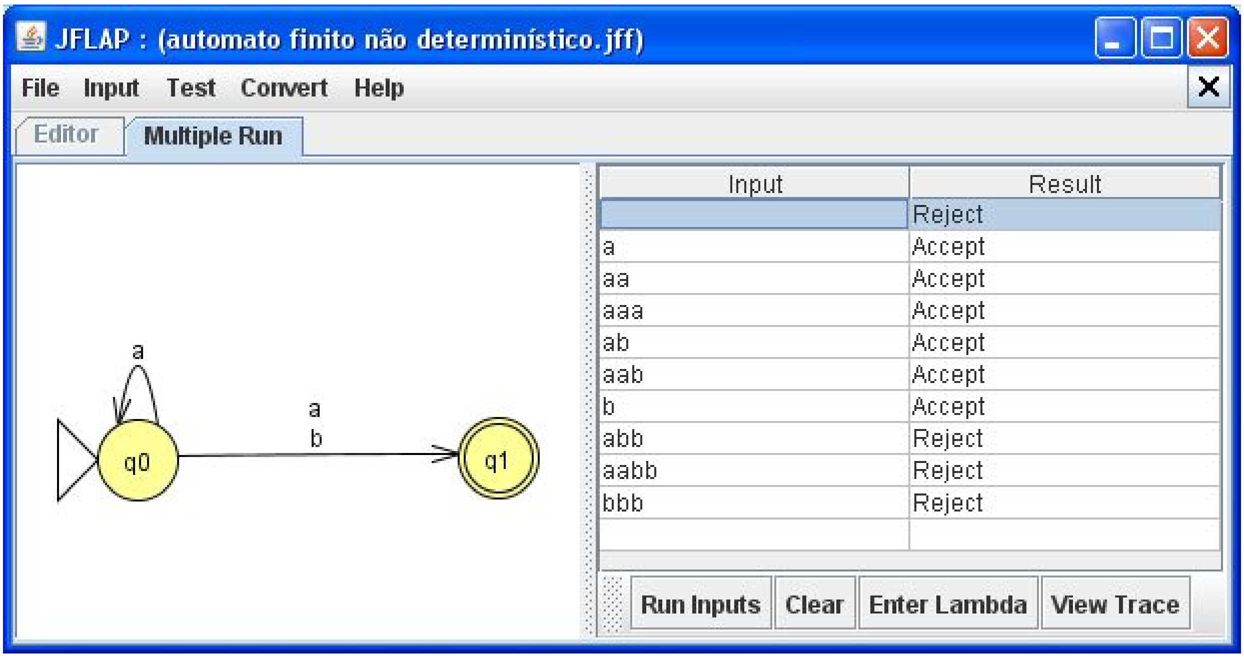


Dicas para construção:

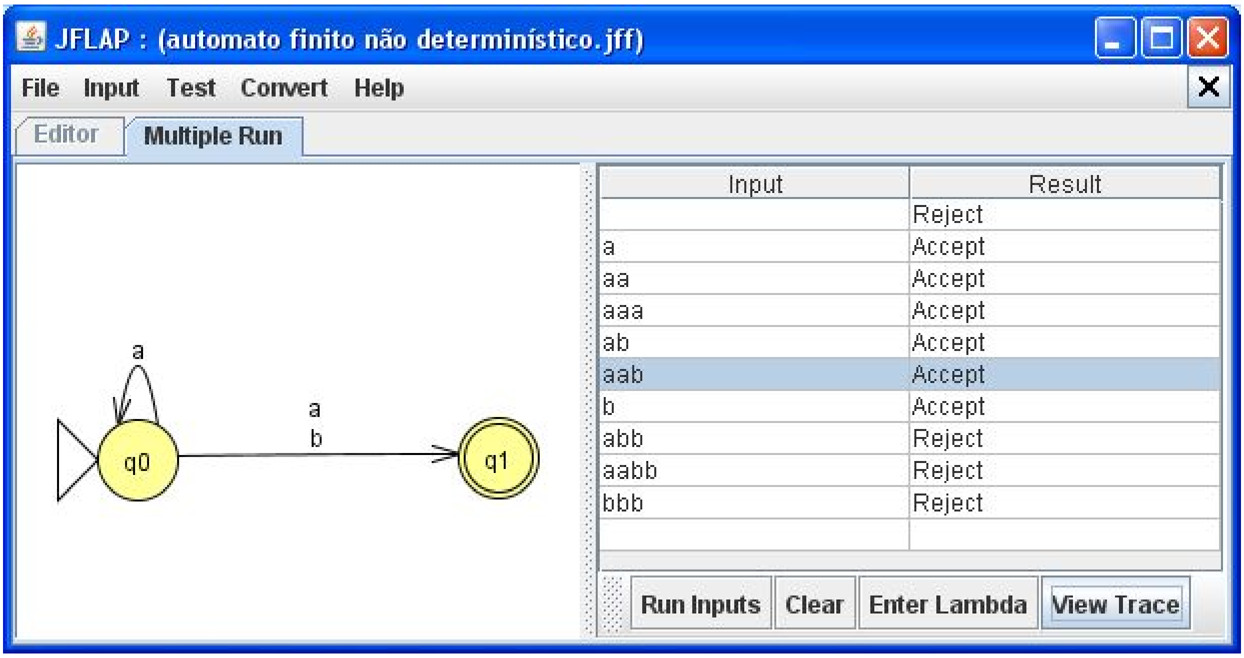
- para criar uma transição arraste o mouse do estado inicial até o estado final.

- 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 ).

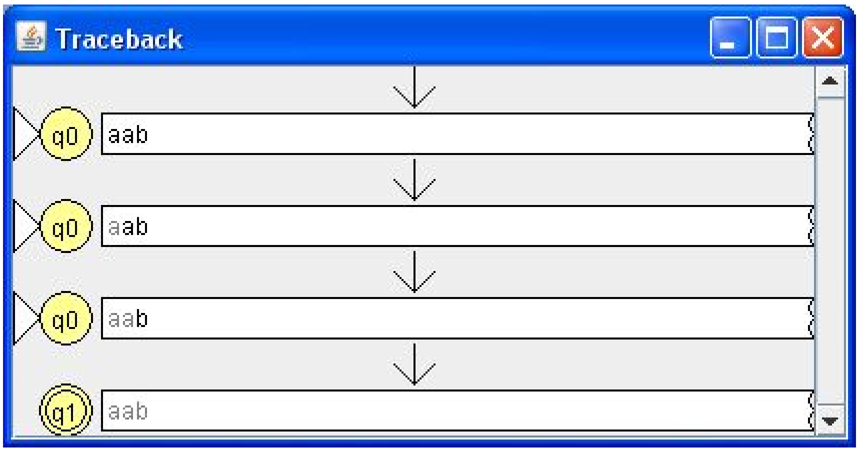
3- Teste do autômato:
Selecione ‘Input’ - ‘Multiple Run’
Digite as strings de teste e selecione ‘Run Inputs’:



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



em seguida clique 'View Trace':



Cada linha representa um símbolo consumido.

28. Lema do Bombeamento para Linguagens Regulares

Se **L** é linguagem regular, então $\exists$ **m** ≥ 1; $\forall$ **w** ϵ **L**, $|\mathbf{w}| \geq \mathbf{m}$ onde **w** = **xyz** com:
$|\mathbf{y}| \geq 1$ e $|\mathbf{xy}| \leq \mathbf{m}$ e $\forall$ $\mathbf{i} \geq 0$, $\mathbf{xy^iz}\ \epsilon\ \mathbf{L}$

De forma geral, 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 **m** pode ser considerada como a qtde de estados do menor AFD que reconhece a linguagem regular (SUDKAMP, 1997, p. 212). Por isso, é recomendado o limite $\mathbf{m} \geq 1$, tal como diretamente estabelecido por Lewis e Papadimitriou (2000, p. 91). Tal limite é útil em provas da não regularidade de linguagens. Veja exemplo 3 no próximo tópico onde, na prova por absurdo, considerar $\forall$ $\mathbf{m} \geq 1$ permite que $\mathbf{w = a^{2m}\ b^m\ b^m\ a^m}$ continue não pertencendo a L, por outro lado se fizéssemos **m**=0, w pertenceria a L, invalidando a prova.

<u>Exemplo 1:</u>

Linguagem: L = **ab***

L é regular, logo deve atender as condições do Lema:
$\exists$ **m** = 2 ; $\forall$ **w** ϵ **L**, $|\mathbf{w}| \geq 2$ onde **w** = **xyz** com $|\mathbf{y}| \geq 1$, $|\mathbf{xy}| \leq \mathbf{2}$
Testando para cada um w:
w = ab; x=a, y=b, z=λ. $\forall$ $\mathbf{i} \geq 0$, $\mathbf{xy^iz}\ \epsilon\ \mathbf{L}$
w = abb; x=a, y=b, z=b. $\forall$ $\mathbf{i} \geq 0$, $\mathbf{xy^iz}\ \epsilon\ \mathbf{L}$
w = abbb; x=a, y=b, z=bb. $\forall$ $\mathbf{i} \geq 0$, $\mathbf{xy^iz}\ \epsilon\ \mathbf{L}$
…
e assim por diante.

<u>Exemplo 2:</u>

Linguagem: L = **ab**

Tal com o lema afirma, se L é regular então deverá atender as condições do Lema:
$\exists$ **m** = 3 ; $\forall$ **w** ϵ **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 $\forall$ **i** $\geq$ 0, $\mathbf{xy^iz}$ ϵ **L**

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 continuação dessa discussão no próximo tópico,

29. 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 → LB

Pela lógica proposicional:

LR	→ LB	ok
LB	→ LR	não podemos afirmar
~LR	→ ~LB	não podemos afirmar
~LB	→ **~LR**	**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. Bastando, para isto, provar por absurdo que se ela não atende as condições do lema do bombeamento então ela não é regular. Em outras palavras, supomos que a linguagem se quer analisar seja regular e, então mostramos que o Lema não acontece (**LR → ~LB**)
Dessa forma, fica provado, por absurdo, que a linguagem não é regular (**~LR**).

Como já observado, o Lema do Bombeamento é bom para provar que uma determinada linguagem L não é regular, mesmo que isso nem sempre seja possível.

De uma maneira geral, usamos o seguinte modelo de prova por absurdo:

1) supomos L regular

2) procuramos uma palavra **w** ϵ **L** dependente de **m** de tal forma que $\forall$ **m**, |**w**| $\geq$ **m** e satisfazendo **w** = **xyz** com |**y**| $\geq$ 1 e |**xy**| $\leq$ **m**

3) finalmente, exibimos um valor de **i** $\geq$ 0 tal que $\mathbf{xy^iz}$ não pertence a **L**, desta forma fica provado por absurdo 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** seja regular,
então existe um **w** escolhido como $\mathbf{a^{m-1}\, a\, b^m}$, $\forall$ **m**, pertence a **L**
onde **w** = **xyz** com **x**= $\mathbf{a^{m-1}}$ e **y**=**a** e **z**=$\mathbf{b^m}$ satisfazendo |**w**| $\geq$ **m**, |**y**| $\geq$ 1 e |**xy**| $\leq$ **m**
Assim, fazendo i=0 em **w** = $\mathbf{xy^iz}$ teremos **w** = $\mathbf{a^{m-1}\, b^m}$ não pertencente a L, o que é uma contradição!
Logo, **L** não é regular.

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

Suponha por absurdo que **L** seja regular,
então existe um **w** escolhido como $\mathbf{a^{m-1}\, b\, b^{m-3}}$, $\forall$ **m**, pertence a **L**
onde **w** = **xyz** com **x**= $\mathbf{a^{m-1}}$ e **y**= **b** e **z**=$\mathbf{b^{m-3}}$ satisfazendo |**w**| $\geq$ **m**, |**y**| $\geq$ 1 e |**xy**| $\leq$ **m**
Assim, fazendo i=2 em **w** = $\mathbf{xy^iz}$ teremos **w** = $\mathbf{a^{m-1}\, b^2\, b^{m-3}} = \mathbf{a^{m+1}\, b^{m-1}}$ não pertencente a L, o que é uma contradição!
Logo, **L** não é regular.

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

Suponha por absurdo que **L** seja regular,
então existe um **w** escolhido como $\mathbf{a^m\, b^m\, b^m\, a^m}$, $\forall$ **m**, pertence a **L**
onde **w** = **xyz** com **x**= λ e **y**= $\mathbf{a^m}$ e **z**= $\mathbf{b^m\, b^m\, a^m}$ satisfazendo |**w**| $\geq$ **m**, |**y**| $\geq$ 1 e |**xy**| $\leq$ **m**
Assim, fazendo i=2 em **w** = $\mathbf{xy^iz}$ teremos **w** = $\mathbf{a^{2m}\, b^m\, b^m\, a^m}$ não pertencente a L, o que é uma contradição!
Logo, **L** não é regular.

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

Suponha por absurdo que **L** seja regular,
então existe um **w** escolhido como $\mathbf{a^m\, b^{m+m}}$, $\forall$ **m**, pertence a **L**
onde **w** = **xyz** com **x**= λ e **y**= $\mathbf{a^m}$ e **z**= $\mathbf{b^{m+m}}$ satisfazendo |**w**| $\geq$ **m**, |**y**| $\geq$ 1 e |**xy**| $\leq$ **m**
Assim, fazendo i=2 em **w** = $\mathbf{xy^iz}$ teremos **w** = $\mathbf{a^{2m}\, b^{m+m}} = \mathbf{a^{2m}\, b^{2m}}$ não pertencente a L, o que é uma contradição!
Logo, **L** não é regular.

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

Suponha por absurdo que **L** seja regular,
então existe um **w** escolhido como $\mathbf{a\, a^{m+1}}$, $\forall$ **m**, pertence a **L**
onde **w** = **xyz** com **x**= λ e **y**= **a** e **z**=$\mathbf{a^{m+1}}$ satisfazendo |**w**| $\geq$ **m**, |**y**| $\geq$ 1 e |**xy**| $\leq$ **m**
Assim, fazendo i=m+1 em **w** = $\mathbf{xy^iz}$ teremos **w** = $\mathbf{a^{m+1}\, a^{m+1}}$ não pertencente a L, pois **2(m+1)** não é sempre primo, mas, para **m** $\geq$ 1 **será** um número par e maior ou igual a 4, o que é uma contradição!
Logo, **L** não é regular.

Explicações adicionais e exemplos de aplicação do Lema do Bombeamento:
https://pt.wikipedia.org/wiki/Lema_do_bombeamento_para_linguagens_regulares
http://www-di.inf.puc-rio.br/~hermann//bombreg/bombreg.html

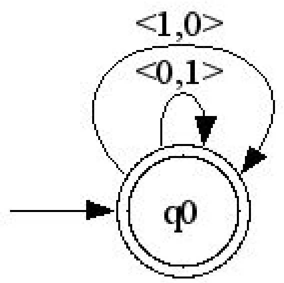
30. Máquina de Mealy

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

Definição:
Uma Máquina de Mealy é uma 6-upla $M = (\sum, Q, \delta, q0, F, \Delta)$ onde:

- $\sum$ é o alfabeto dos símbolos de entrada
- Q é o conjunto dos estados possíveis
- δ é a função transição, $\delta: Q x \sum \rightarrow Q x \Delta^*$
- q0 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).



31. Máquina de Moore

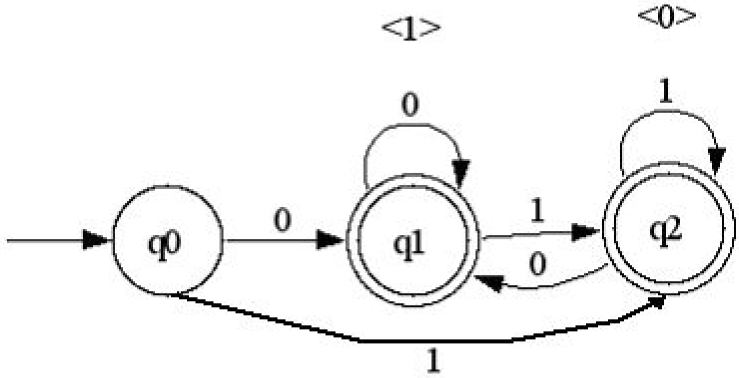
É um AFD com saídas associadas aos estados.

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: numeros reais, variáveis, operadores relacionais, palavras reservadas, etc.

Definição:
Uma Máquina de Moore é uma 7-upla $M = (\sum, Q, \delta, q0, F, \Delta, G)$ onde:

- $\sum$ é o alfabeto dos símbolos de entrada
- Q é o conjunto dos estados possíveis
- δ é a função transição, $\delta: Q x \sum \rightarrow Q$
- q0 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 Máquna de Mealy anterior, ou seja, lê e troca os caracteres da entrada para a saída.



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

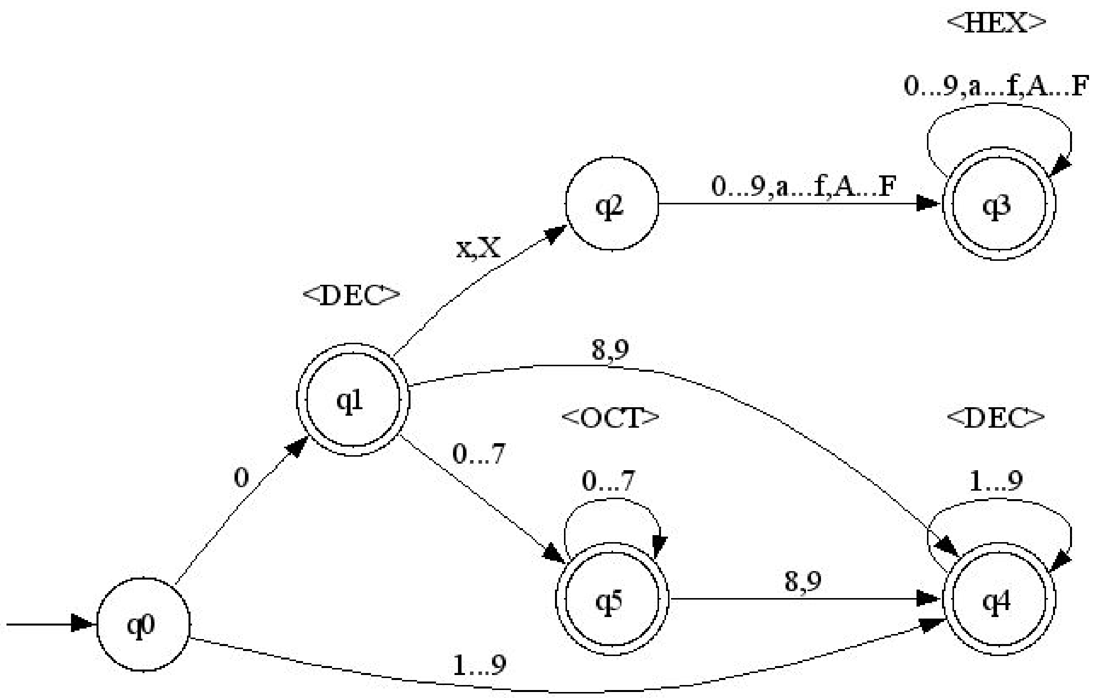
Esta máquina de Moore classifica números inteiros <u>sem sinal</u> de acordo com a sua base: decimal, octal e hexadecimal.

ER para cada saída da máquina:

- ER para números hexadecimais <HEX>: `0(X|x)(0|...|F)+`
- ER para números decimais <DEC>: `(0|...|9)+`
- ER para números octais <OCT>: `0(0|...|7)+`

Serão considerados "0", "08" e "058" como decimais.

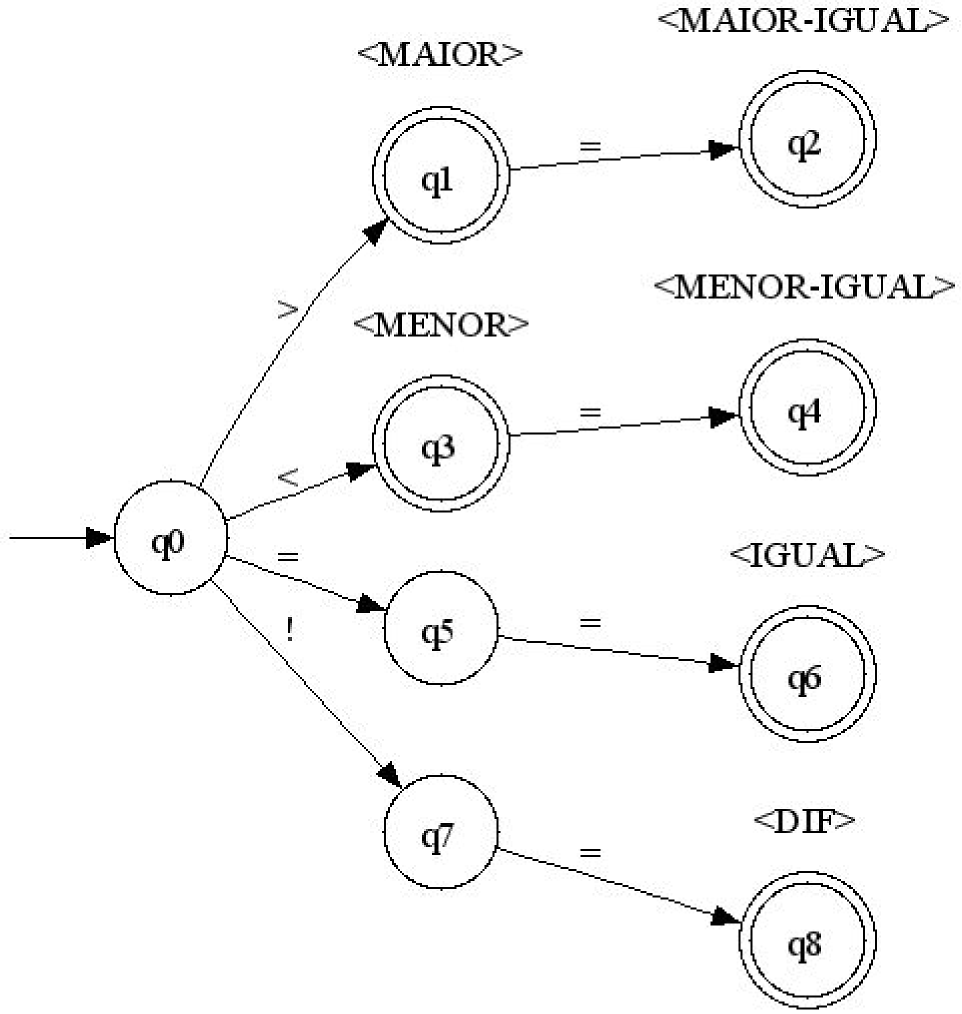
Obs.: as saídas são representadas pelas descrições entre '<' e '>' e em letras maiúsculas.



33. 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>	! =



34. 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.

35. 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)

36. 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 -> aSb | λ
```

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

```
S -> aaSbb | aabb
```

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

```
S -> AA
A -> aAb | λ
```

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

```
S -> aSb | aAb
A -> bAa | ba
```

v) palíndromos sobre {a,b}

```
S -> aSa | bSb | a | b | λ
```

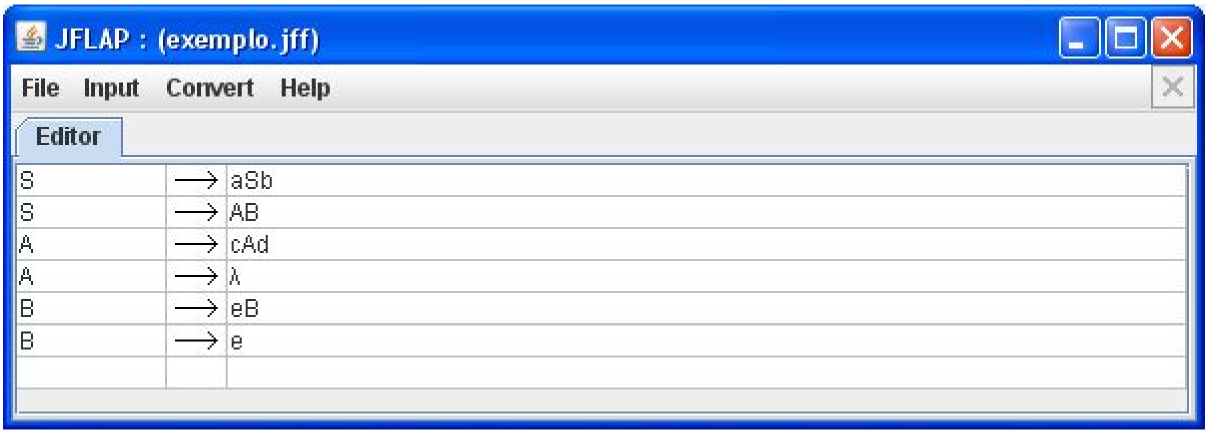
37. Uso do simulador JFLAP para Linguagens Livres de Contexto

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

Gramática livre de contexto

1- Selecione 'Grammar'

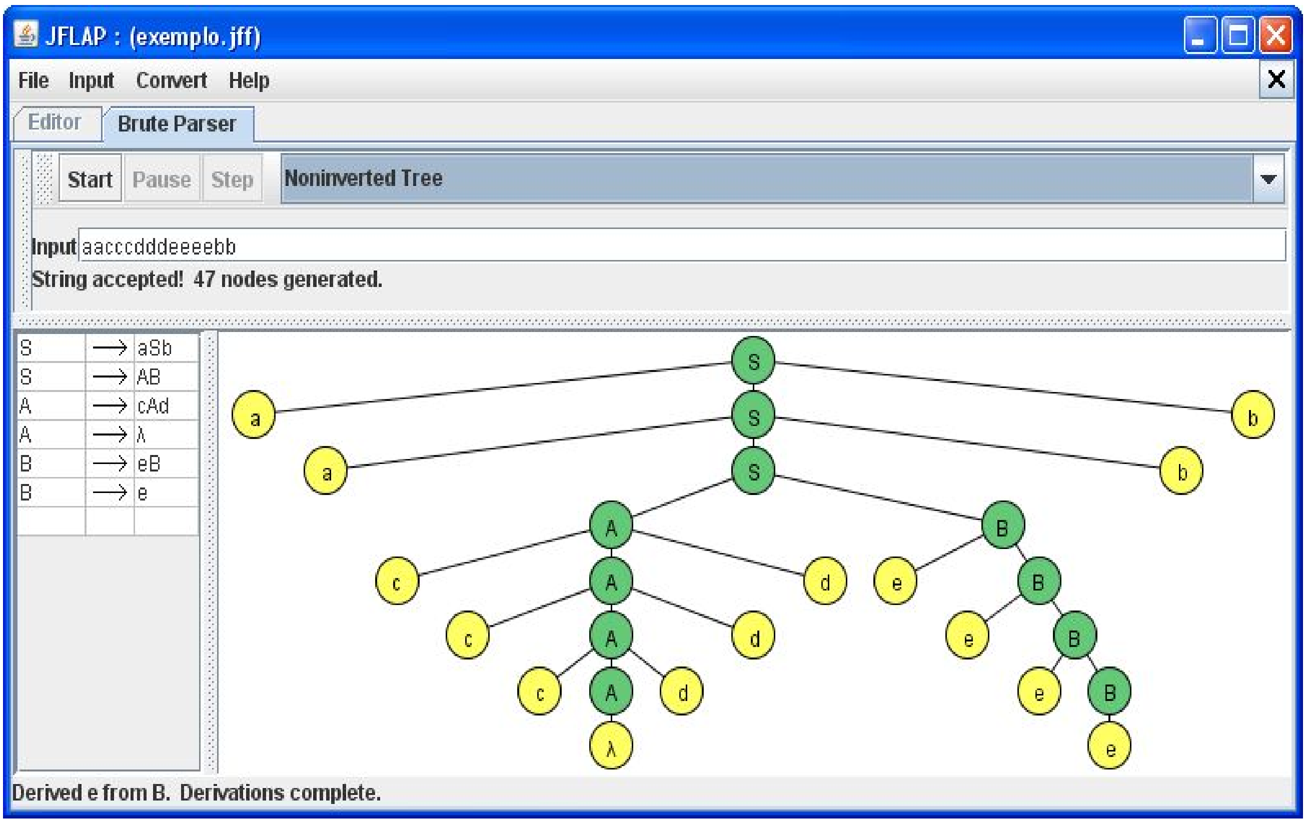
2- Digite a gramática no espaço indicado:



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.



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 fincando 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.

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

Exemplos de GLCs para gerar as linguagens descritas:

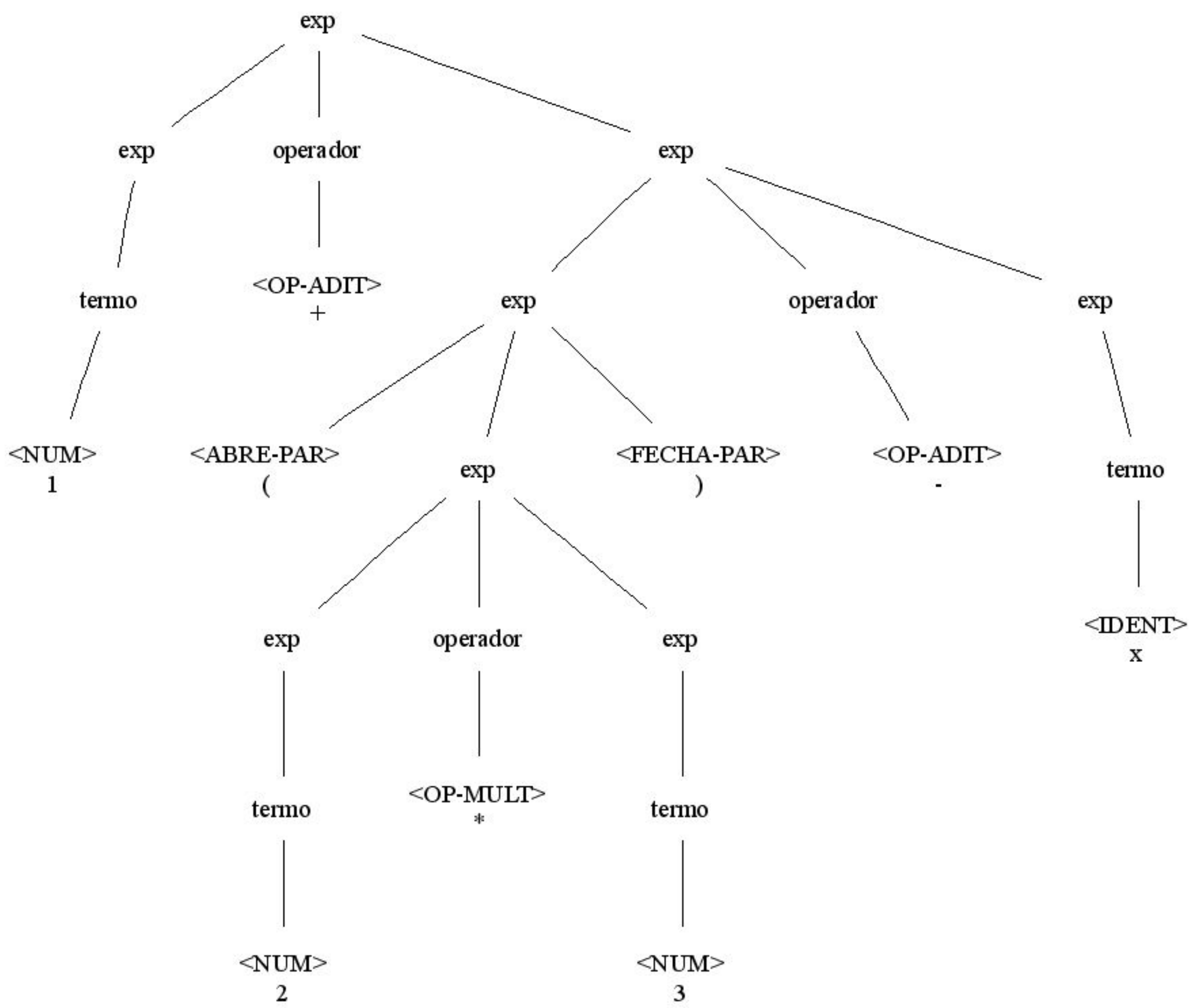
i) Expressão aritmética com parênteses balanceados, números, variáveis e operadores de soma, subtração, multiplicação e divisão.

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
Valor := 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 | λ
```

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 | λ
```

v) Definição de funções no estilo Pascal.
O corpo e 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
corpoFunçao -> 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 identação:

```
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 algoritmica 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 tecnicas de prepração 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:

$$A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \ldots \mid \alpha\beta_n \mid \gamma$$

Onde: $A \in V$

$\alpha, \beta_n, \gamma \in (T \cup V)^*$

Gramática sem conflito à esquerda (fatorada):

$$A \rightarrow \alpha R \mid \gamma$$
$$R \rightarrow \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$$

Onde: $A, R \in V$

$\alpha, \beta_n, \gamma \in (T \cup V)^*$

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:

$A \to A\beta_1 \mid A\beta_2 \mid \ldots \mid A\beta_n \mid \gamma_1 \mid \gamma_2 \mid \ldots \mid \gamma_n$

Onde: $A \in V$

$\beta_n, \gamma_n \in (T \cup V)^*$

Gramática sem recursividade à esquerda:

$A \to \gamma_1 R \mid \gamma_2 R \mid \ldots \mid \gamma_n R$
$R \to \beta_1 R \mid \beta_2 R \mid \ldots \mid \beta_n R \mid \lambda$

Onde: $A, R \in V$

$\beta_n, \gamma_n \in (T \cup V)^*$

45. Exemplo de Análise Preditiva

É uma técnica algoritmica 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.

<u>Exemplo</u>:

Suponha a seguinte linguagem regular:

$((a|b)^*c^+d^*(e|f)g)?$<EOF>

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(próxCaracter=='a' ou próxCaracter=='b' ou próxCaracter=='c')
      A(); reconhece(<EOF>);
   senão se(próxCaracter==<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 -> BC`
ii) `A -> a`

onde:
`A, B, C` ∊ V
`a` ∊ ∑

Exemplo:

A gramática formada pelas regras:

```
S -> aSb | c
```

Pode ser reescrita na FNC como:

```
S -> AB | c
A -> a
B -> SC
C -> 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 -> aα
```

onde:
A ϵ V
a ϵ ∑ ∪ {λ}
α ϵ V*

Exemplo:

A gramática formada pelas regras:

```
S -> aSb | c
```

Pode ser reescrita na FNG como:

```
S -> aSB | c
B -> 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 -> uA | λ
```

em:

```
A -> {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 | λ
declara -> <TIPO> listaIdent declara | λ
listaIdent -> <IDENT> <VIRG> listaIdent | <IDENT>
```

No formato **BNF** ela ficaria:

```
listaDeclara ->  { 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:

```
listaDeclara ->  declara <PT-VIRG> listaDeclara | λ
declara -> <TIPO> listaIdent declara | λ
listaIdent -> <IDENT> <VIRG> listaIdent | <IDENT>
```

No formato **EBNF** ela ficaria:

```
listaDeclara ->  ( <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.

<u>Definição</u>:
Um autômato com pilha (AP) é uma 6-upla $M = (\sum, Q, \Delta, \delta, q0, F)$ onde:

- ∑ 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 δ : Q x (∑ U {λ}) x (Δ U {λ}) → (Q x Δ U {λ})
- q0 é o estado inicial
- F é o conjunto de estados finais. Deve possuir ao menos um elemento.

<u>Representação do AP por intermédio de um grafo</u>:
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.

Não precisa de ser determinístico.

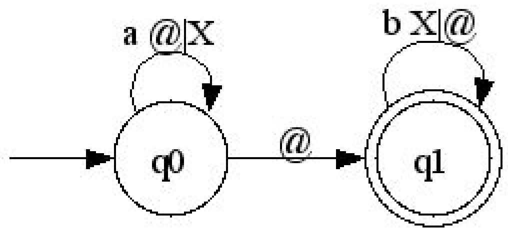
Para que ocorra a aceitação da palavra de entrada é necessário que, além de terminar a leitura da palavra e o autômato estar num estado final, a pilha deve estar vazia.
Cada transição é formada pelo símbolo de leitura seguido do símbolo que é desempilhado e o símbolo que é empilhado.

<u>Exemplos:</u>

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

i) L = { a^n b^n ; i ≥ 0 }

```
S -> aSb | λ
```



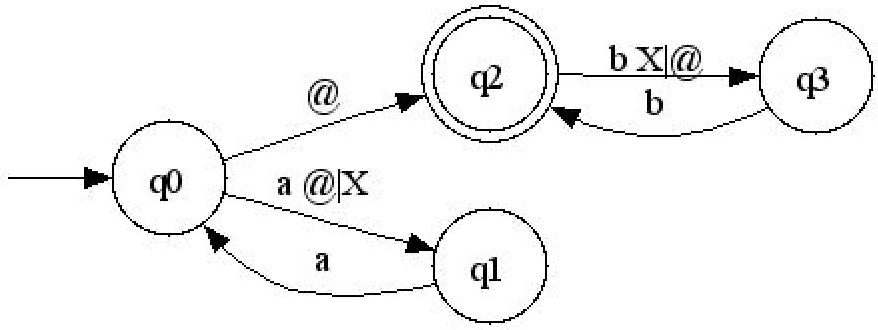
M = (∑, Q, Δ , δ, q0, F)

onde:

∑ = { a, b }
Q = { q0, q1 }
Δ = { X }
δ: (q0, a, λ) → (q0, X)
δ: (q0, λ, λ) → (q1, λ)
δ: (q1, b, X) → (q1, λ)
F = { q1 }

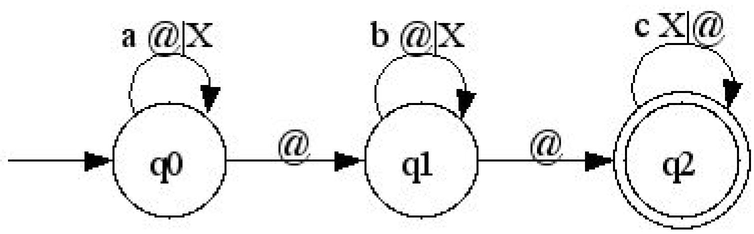
ii) L = { a^{2n} b^{2n} ; n ≥ 0 }

```
S -> aaSbb |  λ
```



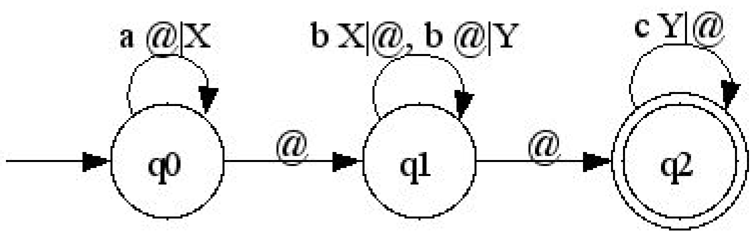
iii) L = { $a^n b^k c^i$; $i = n + k$ }

```
S -> aSc | A
A -> bAc | λ
```



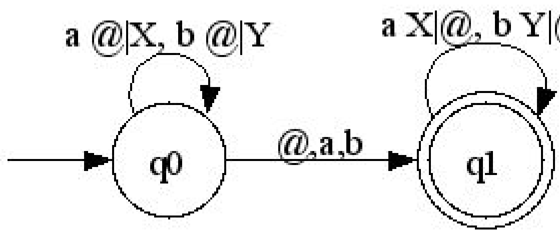
iv) L = { $a^n b^k c^i$; $k = n + i$ }

```
S -> AB
A -> aAb | λ
B -> bBc | λ
```



v) L = palíndromos sobre {a,b}

```
S -> aSa | bSb | a | b | λ
```



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 linhagem 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 → LB`**

Pela lógica proposicional podemos representar:

`LLC`	→	`LB`	ok
`LB`	→	`LLC`	não podemos afirmar
`~LLC`	→	`~LB`	não podemos afirmar
`~LB`	→	**`~LLC`**	**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 $\exists$ **m** ≥ 1 ; $\forall$ **w** ϵ **L**, $|\mathbf{w}| \geq \mathbf{m}$ onde $\mathbf{w} = \mathbf{uxvyz}$ com $|\mathbf{xy}| \geq 1$ e $|\mathbf{xvy}| \leq \mathbf{m}$ e $\forall$ **i** ≥ 0, $\mathbf{ux^ivy^iz}\ \epsilon\ \mathbf{L}$

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

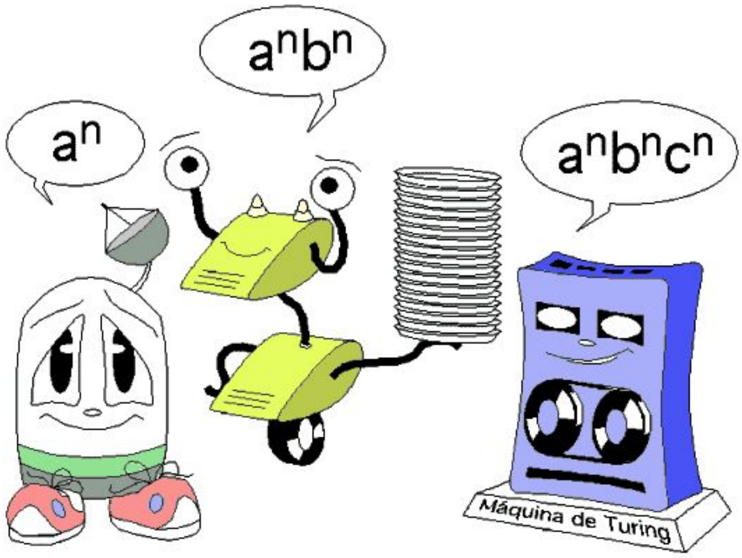
Como já foi dito, o Lema do Bombeamento é bom para provar que uma determinada linguagem L não é livre de contexto. Isto nem sempre é possível, mas, de uma maneira geral, usamos o seguinte modelo de prova, pela técnica do absurdo:

1) supomos L livre de contexto
2) procuramos uma palavra **w** ϵ **L** dependente de **m** de tal forma que $\forall$ **m**, $|\mathbf{w}| \geq \mathbf{m}$ e satisfazendo $\mathbf{w} = \mathbf{uxvyz}$ com $|\mathbf{xy}| \geq 1$ e $|\mathbf{xvy}| \leq \mathbf{m}$
3) finalmente, exibimos um valor de $\mathbf{i} \geq 0$ tal que $\mathbf{ux^ivy^iz}$ não pertence a **L**, desta forma fica provado por absurdo que L não é regular.

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 reconhcedora.

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 Chomsk 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.

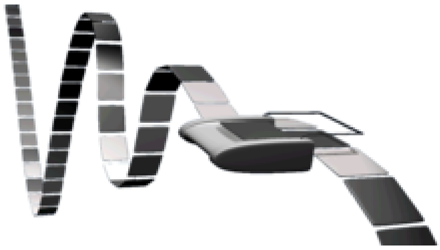
A máquina de Turing é composta de um cabeçote gravador/leitor que opera sobre uma fita de entrada dividida em células sucessivas. Cada célula contém um único símbolo e elas são preenchidas a priori com espaço em branco.

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

1º. lê um símbolo na posição do cabeçote

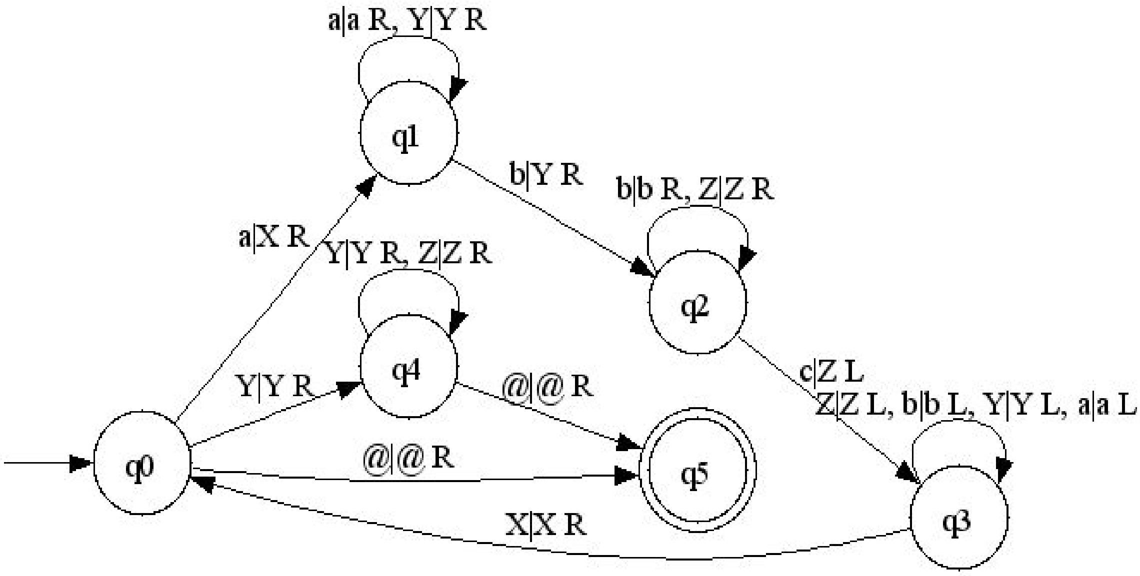
2º. grava um símbolo na posição do cabeçote

3º. move o cabeçote para esquerda (L) ou direito (R).



Exemplos:

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



ii) L = { w c w ; w 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 que, normalmente, o lado esquerdo de suas regras de produção é 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: $A \in V$
$\alpha, \beta \in (T \cup V)^*$
$\gamma \in (T \cup V)^+$

ou:

```
S -> λ
```

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

Exemplos apresentados em aula:

i) L = { a^n b^n c^n ; i ≥ 1 }

Regra	Exemplo de geração pela aplicação da regra
`S  -> aSBc`	`aaSBcBc`
`S  -> abc`	`aaabcBcBc`
`cB -> Bc`	`aaabBBccc`
`bB -> bb`	`aaabbbccc`

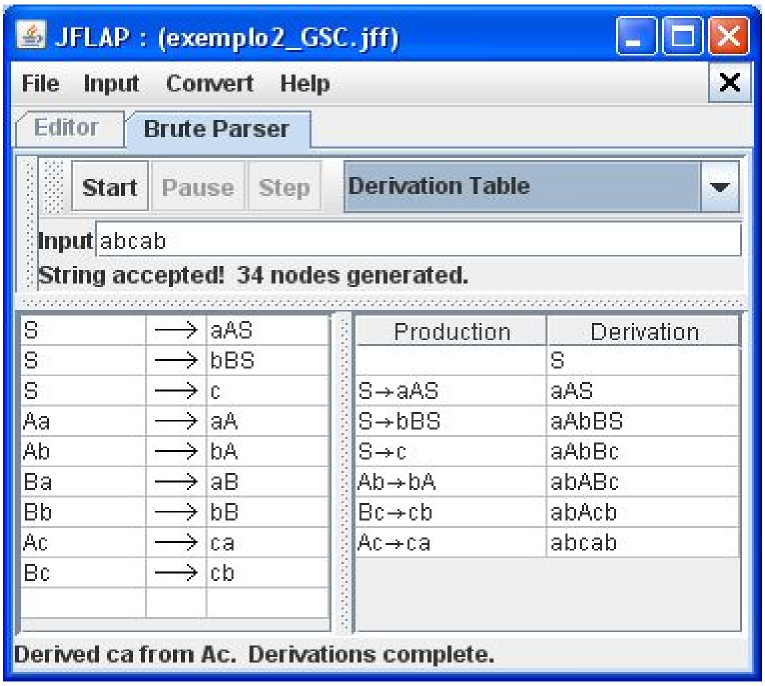
Logo, a gramática é:

```
S  -> aSBc | abc
cB -> Bc
bB -> bb
```

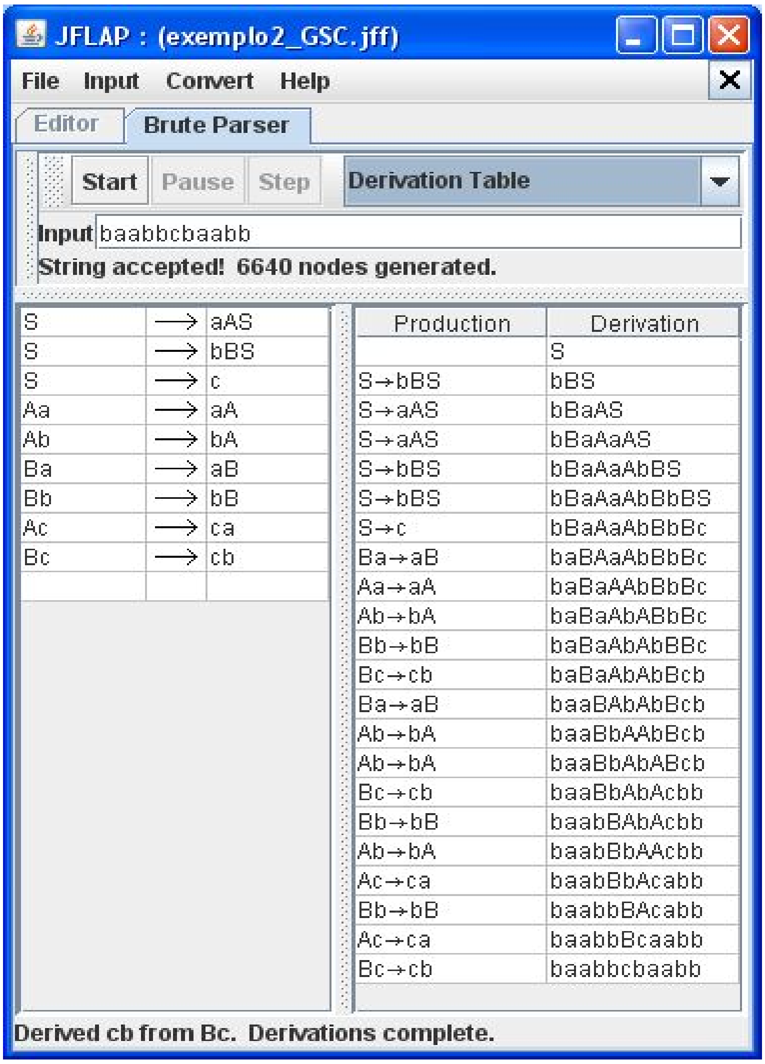
ii) L = { w c w ; w pertence a {a,b}* }

```
S  -> aAS | bBS | c
Aa -> aA
Ab -> bA
Ba -> aB
Bb -> bB
Ac -> ca
Bc -> 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.

Se existe uma Máquina de Turing que aceita todas as strings da linguagem, e que não aceita as strings que não pertencem a linguagem, essa linguagem é Enumerável Recursivamente.

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 deste tipo de linguagem, existem as **gramáticas irrestritas** que gera a linguagem.

A **gramática irrestrita** é representada por produções do tipo:

$$\alpha \rightarrow \beta$$

onde: $\alpha \in (T \cup V)+$
$\beta \in (T \cup V)*$

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.

Exemplo:

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

```
S  -> aSBc | λ
cB -> Bc
aB -> ab
bB -> bb
```

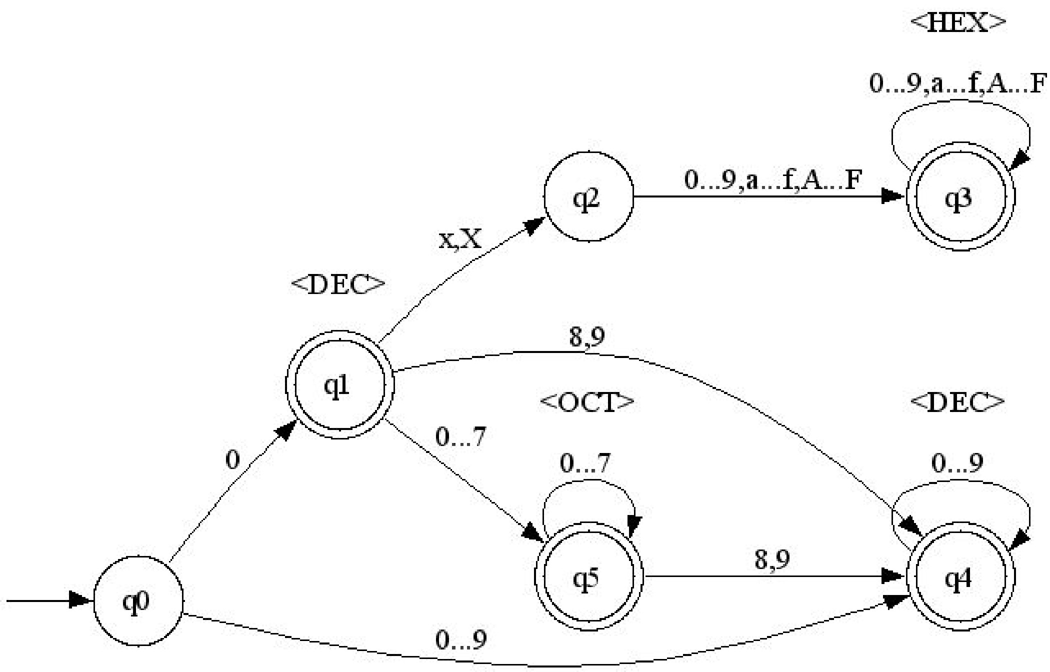
Obs.: vários outros elementos que dão continuidade a esse assunto serão estudados na disciplina Teoria da Computação, tais como outros tipos de Máquinas de Turing e seus formalismos, decidibilidade, o problema da parada e da indecidibilidade, etc.

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

Instalação e uso:

- Faça o download em http://www.graphviz.org/ (freeware – 55MB, v.2.28)
- 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: "Output file Type" (por exemplo GIF), e "Output file Name".
- 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 ;

}
```

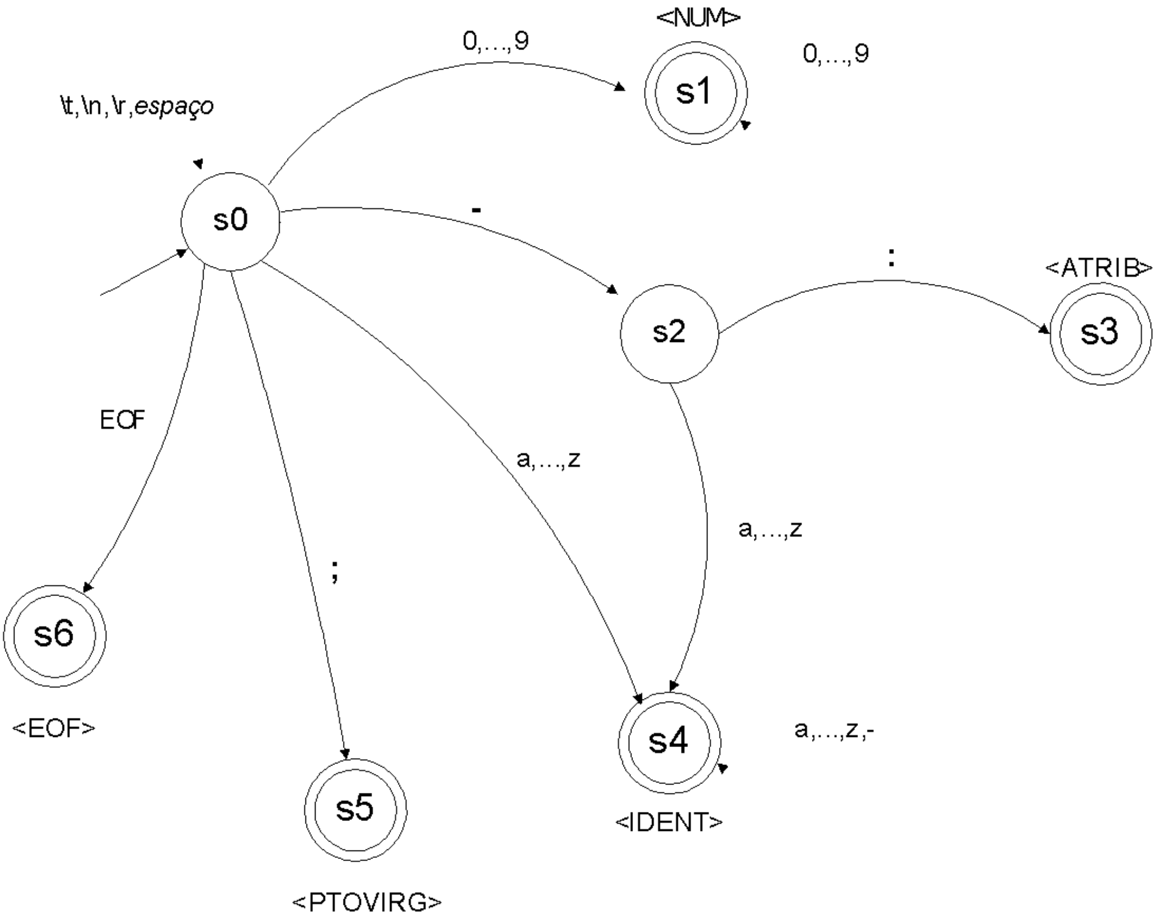
APÊNDICE B: Exemplo de um Analisador Léxico construído através da Análise Preditiva

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|...|9)+	0, 25, 999, 0005
<ATRIB>	-:	-:
<IDENT>	(-|λ)(a|...|z)(a|...|z|-)*	x, x-, -a, -a---, def-cod, x-a
<PTOVIRG>	;	;
<EOF>	*caracter que representa o fim de arquivo*	

Máquina de Moore:


<NUM>
0,...,9
0,...,9
s1
\t,\n,\r,*espaço*
s0
-
:
<ATRIB>
s2
s3
EOF
a,...,z
a,...,z
;
s6
<EOF>
s4
a,...,z,-
s5
<IDENT>
<PTOVIRG>

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:
```

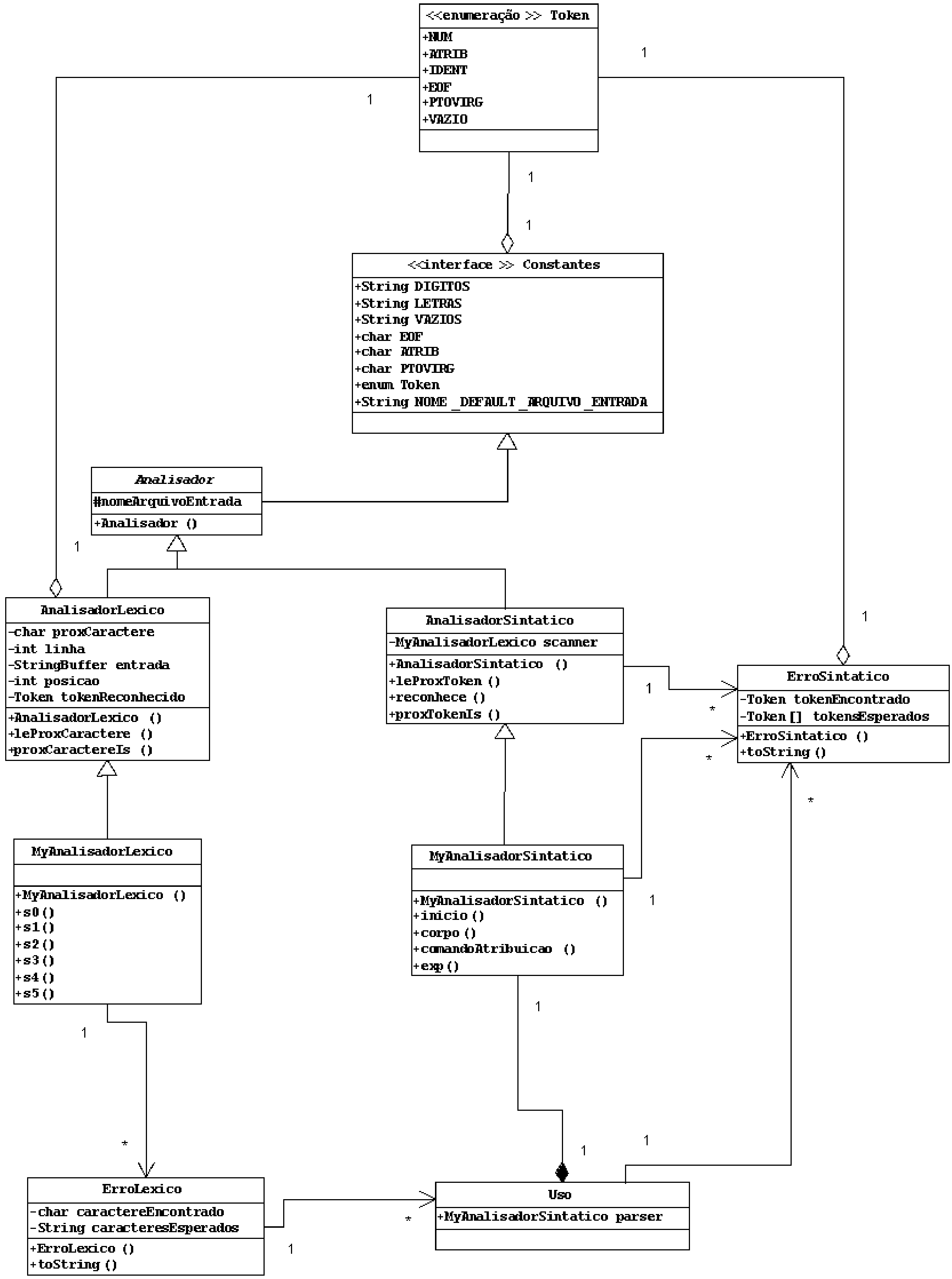
Teste 5:

```
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;
 }
}
```

```java
//--------------------------------------------------------------------------------------------------------------------
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'
   // NÂO avança o ponteiro de leitura
 public boolean proxCaractereIs(String s) {
    if (s.indexOf(this.proxCaractere) != -1)
       return true;
    else
       return false;
 }
}
```

//---

```java
import java.io.IOException;

public class MyAnalisadorLexico extends AnalisadorLexico {
 public MyAnalisadorLexico(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 TesteAnalisadorLexico {
 static public MyAnalisadorLexico 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 MyAnalisadorLexico(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();
 }
```

```java
 // 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
 // NÂO avança o ponteiro de leitura
 public boolean proxTokenIs(Token t) {
    if(t == this.scanner.tokenReconhecido)
       return true;
    else
       return false;
 }
}

//------------------------------------------------------------------------------------------------------------
public class MyAnalisadorSintatico extends AnalisadorSintatico {

 public MyAnalisadorSintatico(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 MyAnalisadorSintatico 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 MyAnalisadorSintatico(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());
    }
 }
}
```