

**Representação de Léxicos  
através de Autômatos Finitos**

**Marcelo Alexandre Couto de Jesus  
Maria das Graças Volpe Nunes**

**Nº 110**

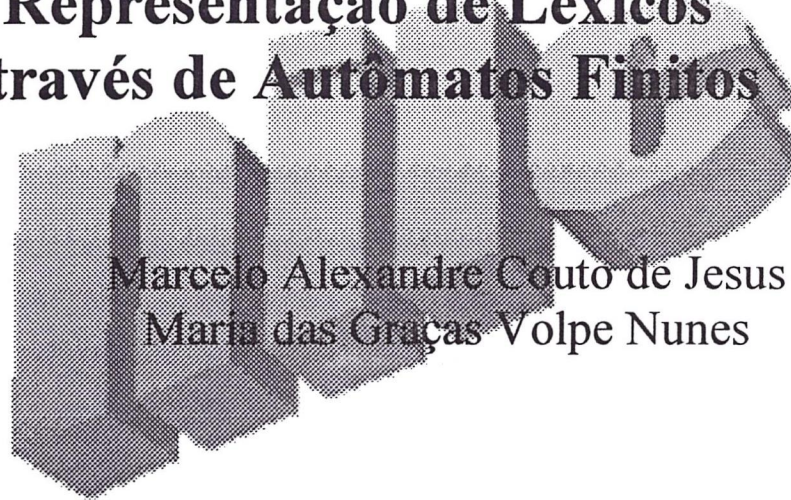
**RELATÓRIOS TÉCNICOS DO ICMC**

**São Carlos  
Mar./2000**

|             |                |
|-------------|----------------|
| SYSNO       | <u>1094882</u> |
| DATA        | <u>1 / 1</u>   |
| ICMC - SBAB |                |

Universidade de São Paulo - USP  
Universidade Federal de São Carlos - UFSCar  
Universidade Estadual Paulista - UNESP

# **Representação de Léxicos através de Autômatos Finitos**



Marcelo Alexandre Couto de Jesus  
Maria das Graças Volpe Nunes

**NILC-TR-00-5**

Março, 2000

Série de Relatórios do Núcleo Interinstitucional de Linguística Computacional  
NILC - ICMC-USP, Caixa Postal 668, 13560-970 São Carlos, SP, Brasil

# Índice

---

|                                                                                    |           |
|------------------------------------------------------------------------------------|-----------|
| <b>LISTA DE FIGURAS.....</b>                                                       | <b>3</b>  |
| <b>FISTA DE TABELAS .....</b>                                                      | <b>4</b>  |
| <b>RESUMO.....</b>                                                                 | <b>5</b>  |
| <b>ABSTRACT.....</b>                                                               | <b>5</b>  |
| <b>1. INTRODUÇÃO .....</b>                                                         | <b>6</b>  |
| 1.1. OBJETIVOS .....                                                               | 6         |
| 1.2. ORGANIZAÇÃO DO TRABALHO.....                                                  | 6         |
| <b>2. LÉXICOS .....</b>                                                            | <b>7</b>  |
| 2.1. ESTRUTURAS .....                                                              | 7         |
| 2.2. O LÉXICO DO NILC .....                                                        | 7         |
| <b>3. AUTÔMATOS FINITOS .....</b>                                                  | <b>10</b> |
| 3.1. NOTAÇÕES E DEFINIÇÕES .....                                                   | 10        |
| 3.2. REPRESENTAÇÃO DE AUTÔMATO FINITO.....                                         | 11        |
| 3.3. MINIMIZAÇÃO DE AUTÔMATO FINITO DETERMINÍSTICO .....                           | 12        |
| 3.4. PACOTES PARA MANIPULAÇÃO DE AUTÔMATOS FINITOS.....                            | 13        |
| 3.4.1. <i>FSA Utilities</i> .....                                                  | 14        |
| 3.4.2. <i>FIRE Lite</i> .....                                                      | 14        |
| 3.4.3. <i>Grail</i> .....                                                          | 15        |
| 3.4.4. <i>FSM Library</i> .....                                                    | 16        |
| <b>4. USO DE AF PARA REPRESENTAÇÃO DE LÉXICOS.....</b>                             | <b>17</b> |
| 4.1. O TRABALHO DE K.L.S. ....                                                     | 17        |
| 4.1.1. <i>Implementação do Autômato</i> .....                                      | 17        |
| 4.1.2. <i>Autômato Binário</i> .....                                               | 18        |
| 4.1.3. <i>Resultados Práticos</i> .....                                            | 20        |
| 4.1.4. <i>Codificação de Informações Lexicais no Autômato</i> .....                | 20        |
| 4.2. ALGORITMO DE MINIMIZAÇÃO DE REVUZ .....                                       | 22        |
| 4.2.1. <i>O Algoritmo de Minimização</i> .....                                     | 22        |
| 4.2.2. <i>O Algoritmo de Ordenação</i> .....                                       | 23        |
| 4.3. OUTROS TRABALHOS RELACIONADOS .....                                           | 23        |
| 4.3.1. <i>“The World’s Fastest Scrabble Program”</i> .....                         | 23        |
| 4.3.2. <i>“Verifica” – Uma Ferramenta de Auxílio à Redação</i> .....               | 23        |
| 4.3.3. <i>AUMAC/NILC</i> .....                                                     | 24        |
| 4.3.4. <i>INTEX</i> .....                                                          | 27        |
| <b>5. UM SISTEMA DE REPRESENTAÇÃO DE GRANDES AF DETERMINÍSTICOS ACÍCLICOS.....</b> | <b>28</b> |
| 5.1. ESTRUTURAS DE DADOS.....                                                      | 28        |

|                                                       |                               |
|-------------------------------------------------------|-------------------------------|
| 5.2. ALGORITMOS PROPOSTOS .....                       | 30                            |
| 5.3. RESULTADOS .....                                 | 35                            |
| 5.4. CONSIDERAÇÕES.....                               | 36                            |
| <b>6. REPRESENTAÇÃO DE ATRIBUTOS LEXICAIS.....</b>    | <b>38</b>                     |
| 6.1. CONSIDERAÇÕES INICIAIS .....                     | 38                            |
| 6.2. PROPOSTA DE FORMATO PARA O LÉXICO DO NILC.....   | 39                            |
| 6.2.1. <i>BNF do Formato Proposto de Léxico</i> ..... | 40                            |
| 6.3. RESULTADOS .....                                 | 41                            |
| 6.4. COMPARAÇÕES E ANÁLISE DOS RESULTADOS .....       | 42                            |
| <b>7. CONCLUSÕES .....</b>                            | <b>45</b>                     |
| 7.1. CONCLUSÕES.....                                  | ERRO! INDICADOR NÃO DEFINIDO. |
| 7.2. RESUMO DAS CONTRIBUIÇÕES .....                   | ERRO! INDICADOR NÃO DEFINIDO. |
| 7.3. TRABALHOS FUTUROS .....                          | ERRO! INDICADOR NÃO DEFINIDO. |
| <b>8. REFERÊNCIAS BIBLIOGRÁFICAS .....</b>            | <b>47</b>                     |

## APÊNDICE A -- B.N.F. DO LÉXICO DO NILC



# Lista de Figuras

---

|                                                                                                                             |    |
|-----------------------------------------------------------------------------------------------------------------------------|----|
| FIGURA 1 – EXEMPLO DE AUTÔMATO FINITO DETERMINÍSTICO .....                                                                  | 10 |
| FIGURA 2 – EXEMPLO DE MATRIZ DE TRANSIÇÕES .....                                                                            | 11 |
| FIGURA 3 – EXEMPLO DE LISTA DE ADJACÊNCIA.....                                                                              | 12 |
| FIGURA 4 – EXEMPLO DE LISTA DE TRANSIÇÕES .....                                                                             | 12 |
| FIGURA 5 – <i>MERGE</i> DE DOIS ESTADOS EQUIVALENTES .....                                                                  | 13 |
| FIGURA 6 – AUTÔMATO PARA TODAS AS FORMAS DOS VERBOS <i>REWORK</i> , <i>REPLAY</i> , <i>OVERWORK</i> E <i>OVERPLAY</i> ..... | 18 |
| FIGURA 7 – AUTÔMATO FINITO DETERMINÍSTICO ACÍCLICO MÍNIMO CRIADO A PARTIR DA FIGURA 6.....                                  | 18 |
| FIGURA 8 – VISÃO GERAL DO SISTEMA.....                                                                                      | 28 |
| FIGURA 9 - PROCESSO DE INSERÇÃO DE UMA PALAVRA NO AUTÔMATO .....                                                            | 31 |
| FIGURA 10 – OPERAÇÃO DE CLONAGEM DE ESTADO .....                                                                            | 32 |
| FIGURA 11 - PROCESSO DE MINIMIZAÇÃO DO AUTÔMATO .....                                                                       | 33 |
| FIGURA 12 - PROCESSO DE CRIAÇÃO DA REPRESENTAÇÃO FINAL DO AUTÔMATO.....                                                     | 34 |
| FIGURA 13 - PROCESSO DE BUSCA DE UMA PALAVRA NA ESTRUTURA FINAL .....                                                       | 35 |
| FIGURA 14 - REUTILIZAÇÃO DE SUFIXOS .....                                                                                   | 38 |
| FIGURA 15 - REPRESENTAÇÃO POR DIFERENÇA DE SUFIXO .....                                                                     | 38 |
| FIGURA 16 - RESULTADOS OBTIDOS POR MOHRI COM O FSM LIBRARY .....                                                            | 43 |

# Lista de Tabelas

---

|                                                                                               |    |
|-----------------------------------------------------------------------------------------------|----|
| TABELA 1 - CLASSES DAS PALAVRAS DO LÉXICO DO NILC .....                                       | 8  |
| TABELA 2 - ESTATÍSTICAS DETALHADAS DO LÉXICO DO NILC.....                                     | 9  |
| TABELA 3 – ESTATÍSTICAS PARA OS VOCABULÁRIOS E OS AUTÔMATOS .....                             | 20 |
| TABELA 4 – EXEMPLO DE ENTRADAS SIMPLES DO LÉXICO .....                                        | 21 |
| TABELA 5 – EXEMPLO DE ENTRADAS DO LÉXICO COM A FORMA CANÔNICA .....                           | 21 |
| TABELA 6 – EXEMPLO DE ENTRADAS COMPLETAS DO LÉXICO.....                                       | 22 |
| TABELA 7 – ESQUEMA DE CODIFICAÇÃO DE ACENTOS.....                                             | 25 |
| TABELA 8 - EXEMPLO DE CODIFICAÇÃO DOS ACENTOS AO FINAL DA PALAVRA.....                        | 25 |
| TABELA 9 - DADOS SOBRE OS LÉXICOS USADOS NOS TESTES .....                                     | 35 |
| TABELA 10 – ENTRADAS DO LÉXICO DO NILC E RESPECTIVAS CONVERSÕES PARA O LÉXICO OTIMIZADO ..... | 39 |
| TABELA 11 - DESCRIÇÃO DOS LÉXICOS UTILIZADOS NOS TESTES DO FORMATO .....                      | 41 |
| TABELA 12 - RESULTADOS DA CONVERSÃO DAS PALAVRAS DE LEX1 E LEX10.....                         | 41 |
| TABELA 13 - RESULTADOS DE LEX1 E LEX10 .....                                                  | 42 |
| TABELA 14 - COMPAÇÃO DE LEX*.DIR E LEX*.FULL COM LEX*.W .....                                 | 42 |

## RESUMO

---

Aplicações de Processamento de Língua Natural (PLN), como revisores ortográficos, gramaticais e tradutores, geralmente precisam consultar extensos dicionários contendo, por exemplo, informações morfossintáticas de várias centenas de milhares de palavras de uma língua. Autômatos finitos, largamente utilizados na construção de eficientes analisadores léxicos para compiladores, apresentam-se como ótimos candidatos para o problema de representação desse tipo de léxico de língua natural. Este trabalho investigou os métodos de codificação de léxicos utilizando autômatos finitos, as técnicas de minimização de autômatos determinísticos acíclicos e as estruturas de dados necessárias para uma representação compacta, resultando em um sistema computacional eficiente e versátil de representação de grandes léxicos de língua natural. Um léxico de mais de 430.000 palavras da língua portuguesa pode ser convertido em um autômato de menos de 220Kb, em menos de 5 minutos usando um computador doméstico.

## ABSTRACT

---

Natural Language Processing (NLP) applications, such as speller and grammar checkers and translation systems, need to search very large dictionaries which contain morphosyntactical and/or semantic information of several hundreds of thousand words of a given language. Finite Automata are often used in efficient scanners for compilers and are also good candidates for representing dictionaries. This work has investigated methods for representing dictionaries using finite automata, techniques for minimizing acyclic deterministic finite automaton, and adequate data structures for a compact representation. The resulting system is able to represent a dictionary of 430,000 Brazilian Portuguese words in a 220Kb automaton, by using a standard home computer and spending less than five minutes.



# Representação de Léxicos através de Autômatos Finitos\*

## 1.Introdução

---

A representação de grandes dicionários de língua natural, principalmente nos casos em que se trabalha com vários milhões (ou dezenas de milhões) de palavras, é um interessante problema computacional a ser tratado dentro da área de Processamento de Língua Natural (PLN). Autômatos finitos (AFs), largamente usados na construção de compiladores [AHO86, HOP79], são excelentes estruturas para representação desses dicionários, permitindo acesso direto aos às palavras e seus possíveis atributos (gênero, número, grau, etc) [APP88; GRO89; KOW93; MOH95; NUN96; KOW98].

Um dicionário contendo mais de 430.000 palavras da língua portuguesa sem atributos, cuja representação em formato texto ocupa mais de 4.5Mb, pode ser convertido em um AF compactado de apenas 218Kb, conforme será apresentado (o processo de conversão para esse caso levou 4:17Seg, rodando em um Pentium166/Linux).

### 1.1. Objetivos

Esse trabalho tem por objetivo investigar métodos eficientes de construção, representação e minimização de AFs de grande porte (dezenas de milhões de estados), a fim de propor um conjunto de estruturas e algoritmos para criação de um sistema eficiente e versátil de representação de grandes léxicos de língua natural.

Outro objetivo é o de investigar métodos de representação para os atributos das entradas léxicas, de forma a permitir a criação de um formato otimizado capaz de ser inserido diretamente no AF, com o menor impacto possível no tamanho final desse último.

Será também desenvolvido um protótipo do sistema, para efeito de testes e confirmação de eficiência do sistema.

### 1.2. Organização do Trabalho

Esse trabalho se encontra dividido da seguinte forma: o capítulo 2 descreve, de forma geral, o que é um léxico e suas formas tradicionais de representação, além de introduzir o léxico do NILC (que será usado no decorrer desse trabalho); o capítulo 3 discorre sobre AFs (definições, forma de representação, minimização e cita alguns pacotes de manipulação disponíveis); no capítulo 4 são resumidos dois trabalhos importantes da área, além de outros trabalhos relacionados; o capítulo 5 descreve o sistema de representação de AFs que está sendo proposto; o capítulo 6 introduz uma alternativa do formato para representação de atributos; e, finalmente, o capítulo 7 apresenta as conclusões, contribuições e trabalhos futuros.

---

\* Este trabalho, que teve o apoio da Fapesp, foi apresentado, como dissertação de mestrado do primeiro autor, ao Programa de Pós-Graduação em Computação do ICMC-USP/São Carlos, em Dezembro de 1999.

## 2. Léxicos

---

O léxico consiste de uma lista de palavras com zero ou mais informações associadas, tais como sua categoria sintática, gênero, número e forma canônica, entre outras. Em PLN, o léxico representa o vocabulário (ou dicionário) que será utilizado pelo sistema, e por isso sua abrangência depende do tipo de aplicação que o utiliza.

### 2.1. Estruturas

Os léxicos podem ser representados de várias formas, de acordo com o seu tamanho e aplicação a que se destinam.

A forma mais simples de representação léxica é através de arquivos texto, que apesar de simples e não otimizada, é bastante usada. Problemas começam a ocorrer com o crescimento do número de entradas lexicais (lexemas), pois passa a ser difícil (senão impossível) manipular esse arquivo. Mas este formato é excelente para importação e exportação de léxicos entre diferentes sistemas, devido à sua inerente simplicidade.

Outra forma de representação léxica muito usada é através de base de dados, onde todo o trabalho de armazenamento e indexação é feito pelo Sistema Gerenciador de Banco de Dados (SGBD). Mesmo não se obtendo uma taxa muito alta de velocidade de acesso, e com uma alta taxa de ocupação em disco, esse sistema é atraente pela capacidade de executar complexas buscas, por possibilitar vários usuários trabalharem ao mesmo tempo no mesmo léxico (através de um SGBD de arquitetura cliente-servidor), pela fácil manutenção do léxico, e pela facilidade no desenvolvimento de softwares para manipulação do léxico (usando ferramentas RAD<sup>1</sup>).

Entretanto, aplicações que necessitem acessar léxicos de forma rápida e otimizada precisam que esses sejam implementados de uma forma bem eficiente. Neste ponto, a questão do tamanho começa a representar um problema, pois passa a ser desejável que o léxico resida totalmente na memória principal da máquina, e as formas de representação citadas acabam não sendo aplicáveis nesse caso. Assim, é necessária a utilização de estruturas de dados especiais para esse tipo de representação. Existem várias alternativas, que podem ser usadas separada ou combinadamente: árvores, *tries*, tabelas *hash*, autômatos, listas, redes neurais, entre outras. Em geral a solução adotada depende muito das características do léxico a ser representado e no tipo de manipulação que será efetuada.

### 2.2. O Léxico do NILC

O Núcleo Interinstitucional de Linguística Computacional (NILC), durante o convênio entre a Universidade de São Paulo (USP) e a Itautec-Philco, criou um léxico da língua portuguesa do Brasil com cerca de 1.500.000 lexemas, gerados (via flexão e derivação) a partir de cerca de 100.000 lemas ou formas canônicas. Cada lexema pode pertencer a uma ou mais classes, de 13 disponíveis, cada uma com atributos distintos [NUN96]. Esse léxico será utilizado nos testes e exemplos citados neste trabalho. Segue um rápido exemplo de algumas entradas desse léxico:

---

<sup>1</sup> RAD – Rapid Application Development, ferramentas de desenvolvimento de software como, por exemplo, o Visual Basic da Microsoft e o Delphi da Inprise (antiga Borland)



---

```

banana=<S.2G.SI.N.[ ]?.3.[banana]0.>
bananinhas=<S.F.PL.D.[ ]??.[banana]0.>
casa=<S.F.SI.N.[para.]?.3.[casa]0.#V.[INT.PRONOM.TD.TI.][IMPER-
AFIRM.TU.PRES.ELE.]N.[ ][casar]0.>

```

---

Neste exemplo, as duas primeiras entradas identificam substantivos (singular normal e plural diminutivo, respectivamente), associados à mesma forma canônica ([*banana*]). A terceira entrada (*casa*) identifica uma palavra que é usada tanto como substantivo (feminino, singular, regido pela preposição *para*, derivado da forma canônica [*casa*]) quanto como verbo (intransitivo, pronominal, transitivo direto e transitivo indireto, que ocorre na segunda pessoa do singular do imperativo afirmativo e na terceira pessoa do singular do presente do verbo *casar*).

A Tabela 1 apresenta as classes possíveis para as palavras do léxico do NILC. E, nos apêndices, pode ser encontrada a notação BNF completa do formato desse léxico.

**A partir de um *snapshot* do léxico consolidado de Fevereiro/1999, foram compiladas algumas estatísticas expostas na**

Tabela 2. Esse léxico contém 1.506.765 entradas compostas distintas (89Mbytes), como as citadas no primeiro exemplo dessa seção.

**Tabela 1 - Classes das palavras do léxico do NILC**

| Classe       | Abreviatura |
|--------------|-------------|
| Substantivo  | S.          |
| Artigo       | ART.        |
| Preposição   | PREP.       |
| Numeral      | NUM.        |
| Pronome      | PRON.       |
| Verbo        | V.          |
| Adjetivo     | ADJ.        |
| Conjunção    | CONJ.       |
| Advérbio     | ADV.        |
| Nome Próprio | NOM.        |
| Prefixo      | PREF.       |
| Sigla        | SIGL.       |
| Abreviatura  | ABREV.      |

**Tabela 2 - Estatísticas Detalhadas do Léxico do NILC**

| <b>Classe</b> | <b>Quantidade<sup>2</sup> / %</b> |         | <b>Derivações<sup>3</sup> / %</b> |         |
|---------------|-----------------------------------|---------|-----------------------------------|---------|
| Substantivo   | 57.862                            | 3,725%  | 105.074                           | 3,443%  |
| Artigo        | 8                                 | 0,001%  | 16                                | 0,001%  |
| Preposição    | 148                               | 0,010%  | 268                               | 0,009%  |
| Numeral       | 235                               | 0,015%  | 430                               | 0,014%  |
| Pronome       | 252                               | 0,016%  | 534                               | 0,017%  |
| Verbo         | 1.426.476                         | 91,839% | 2.817.374                         | 92,314% |
| Adjetivo      | 59.624                            | 3,839%  | 111.202                           | 3,644%  |
| Conjunção     | 52                                | 0,003%  | 192                               | 0,006%  |
| Advérbio      | 2.673                             | 0,172%  | 5.730                             | 0,188%  |
| Nome Próprio  | 5.143                             | 0,331%  | 9.700                             | 0,318%  |
| Prefixo       | 73                                | 0,005%  | 104                               | 0,003%  |
| Sigla         | 459                               | 0,030%  | 912                               | 0,030%  |
| Abreviatura   | 228                               | 0,015%  | 416                               | 0,014%  |
| <b>TOTAL</b>  | <b>1.553.233</b>                  |         | <b>3.051.952</b>                  |         |

<sup>2</sup> O valor total é maior que o número de entradas distintas do léxico devido ao fato de algumas entradas pertencerem a duas ou mais classes diferentes (ex: "casa" que além de substantivo fem. sing. também é verbo na 3ª pessoa e no imperativo).

<sup>3</sup> Na derivação, todas as entradas são separadas pelas suas variações de gênero, número, grau, pessoa, tempo, etc. Um verbo, por exemplo, é derivado em mais de 220 entradas distintas (pouco mais de 50 flexões normais dos verbos, e as outras decorrentes das ênclises e mesóclises).

## 3. Autômatos Finitos

Autômatos finitos são largamente utilizados na construção de compiladores, onde exercem a função de eficientes analisadores léxicos [AHO86]. Nesta seção serão apresentadas as notações e definições básicas, os métodos de representação alguns dos algoritmos de minimização de autômatos finitos.

### 3.1. Notações e Definições

Nesta seção serão informalmente introduzidas algumas notações e definições relevantes a linguagens e autômatos finitos.

#### Alfabeto, Palavras e Linguagens

Seja  $A$  um conjunto finito, chamado *alfabeto*. Seus elementos são chamados *letras* e, por conveniência, serão denotá-los por letras minúsculas ( $a, b, c, \dots$ ). Será assumida a existência de uma ordem no alfabeto.

Uma *palavra* é uma sequência finita de letras. O comprimento de uma palavra  $w$  é denotado por  $|w|$ , e sua  $n$ -ésima letra por  $w_n$ . O *produto* de duas palavras  $u$  e  $v$ ,  $uv$ , representa a palavra obtida escrevendo-se sequencialmente as letras de  $u$  e depois as letras de  $v$ . O conjunto de todas as palavras é representado por  $A^*$  e a palavra vazia por  $\varepsilon$ . Diz-se que uma *linguagem* é qualquer subconjunto de  $A^*$ .

Uma palavra  $f$  é dita um *fator* de uma palavra  $w$  se  $w = xfy$  para algum  $x, y \in A^*$ . É um *fator próprio* de  $w$  se  $w \neq f$ , um *prefixo* de  $w$  se  $x = \varepsilon$ , e um *sufixo* de  $w$  se  $y = \varepsilon$ .

#### Autômato Finito

Um autômato finito  $M$  (Figura 1) é uma quintupla  $(Q, \Sigma, \delta, q_0, F)$  onde

$Q$  é um conjunto finito de *estados*;

$\Sigma$  é o *alfabeto de entrada*;

$\delta: Q \times \Sigma \rightarrow Q$  representa a *função de transição de estados*;

$q_0 \in Q$  é o *estado inicial* do autômato; e

$F \subseteq Q$  é o conjunto de *estados finais*.

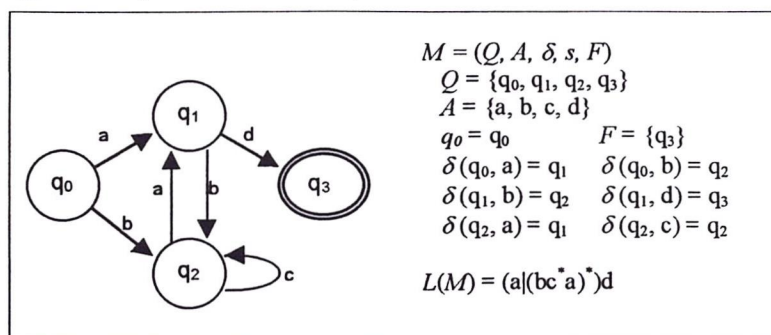


Figura 1 – Exemplo de Autômato Finito Determinístico

A função de transição  $\delta$  não é necessariamente uma função total, isto é, pode não estar definida para todos os pares de estados e símbolos de entrada em  $Q \times \Sigma$ . Se  $\delta$  é uma função total então diremos que  $M$  é um autômato finito *completo*.



Um autômato finito  $M$  é dito *determinístico* se, dado um estado  $q \in Q$  e um símbolo  $a \in \Sigma$ , quaisquer, não existe mais do que uma transição definida pela função  $\delta$  para este par de estado/símbolo. Caso contrário o autômato é dito *não determinístico*.

Diz-se que um autômato finito é *acíclico* quando este não apresenta ciclos, ou seja, a partir de um estado  $q \in Q$  qualquer, a função de transição deve definir valores tais que este estado não seja mais alcançado. Se existe apenas um estado final (sem transições a partir dele) e nenhuma transição leva ao estado inicial, diz-se que o autômato é *normalizado*.

O estado alcançado pela transição do símbolo  $a \in \Sigma$  a partir do estado  $q \in Q$  é denotado por  $q.a = \delta(q, a)$ . Esta notação é transitiva: se  $w \in \Sigma^*$  é uma palavra então  $q.w$  denota o estado alcançado usando-se as transições simbolizadas por cada letra  $w_1, w_2, \dots, w_n$  de  $w$ .

Uma palavra  $w$  é *aceita* por um autômato finito  $M$  se existe um caminho definido pela função de transição para as letras de  $w$ , do estado inicial até algum estado final de  $M$  (ou, de outra forma, se  $q_0.w \in F$ ). A linguagem *aceita* (ou *reconhecida*) por um autômato finito  $M$  é denotada por  $L(M)$ . Diz-se que dois autômatos finitos  $M$  e  $N$  são *equivalentes* se  $L(M) = L(N)$ . Um autômato é dito *mínimo* se ele for determinístico e, se comparado a todos os autômatos equivalentes a ele, possuir o menor número de estados.

## 3.2. Representação de Autômato Finito

De modo geral, o problema de representação de um autômato finito reduz-se ao problema de representar a função de transição  $\delta$  do autômato. Nesta seção serão expostos os métodos de representação tradicionais.

### Matriz de Transições

Este é o método mais simples para se representar um autômato finito, e consiste em se armazenar os valores da função de transição em uma matriz  $\Sigma \times Q$  (Figura 2). Entretanto, este método só deve ser utilizado para autômatos com poucos estados e alfabetos pequenos, pois apesar do tempo de acesso ser  $O(1)$ , o requisito de espaço é  $O(|\Sigma| \times |Q|)$ , para um autômato determinístico.

|   | $q_0$ | $q_1$ | $q_2$ | $q_3$ |
|---|-------|-------|-------|-------|
| a | $q_1$ | ---   | $q_1$ | ---   |
| b | $q_2$ | $q_2$ | ---   | ---   |
| c | ---   | ---   | $q_2$ | ---   |
| d | ---   | $q_3$ | ---   | ---   |

Figura 2 – Exemplo de Matriz de Transições

### Listas de Adjacência

Uma forma tradicional de representação de grafos é utilizar listas de adjacência, e isto também pode ser aplicado a autômatos. Assim, para cada estado  $q \in Q$  do autômato, associa-se um conjunto de pares  $(a, \delta(q, a))$ , para toda  $\delta(q, a)$  definida (Figura 3). O requisito de espaço é  $O(|Q| + |\delta|)$ , e o tempo de acesso é  $O(\log(\min(|Q|, |\delta|)))$ .

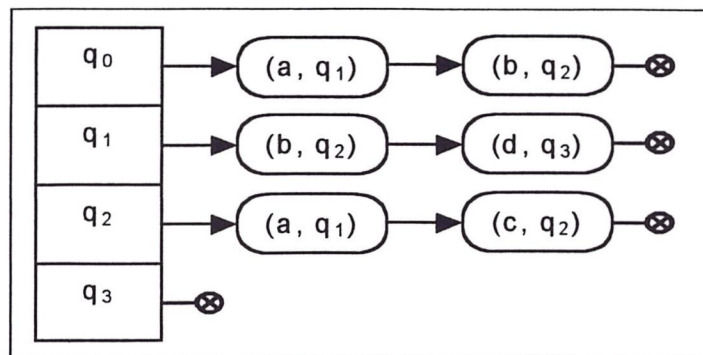


Figura 3 – Exemplo de Lista de Adjacência

### Lista de Transições

Este método consiste em implementar a lista de transições definida pela função  $\delta$  do autômato. Cada  $\delta(p, a) \rightarrow q$  é representado na forma de uma tripla  $(p, a, q)$ , e o requisito total de espaço é  $O(|\delta|)$ . Assume-se que esta lista será armazenada em uma tabela *hash* para que seja possível calcular rapidamente a função de transição. Desta forma, dada uma função *hash* eficiente, na média o tempo de acesso à função de transição será constante.

| <i>p</i>       | <i>a</i> | <i>q</i>       |
|----------------|----------|----------------|
| q <sub>0</sub> | a        | q <sub>1</sub> |
| q <sub>0</sub> | b        | q <sub>2</sub> |
| q <sub>1</sub> | b        | q <sub>2</sub> |
| q <sub>1</sub> | d        | q <sub>3</sub> |
| q <sub>2</sub> | a        | q <sub>1</sub> |
| q <sub>2</sub> | c        | q <sub>2</sub> |

Figura 4 – Exemplo de Lista de Transições

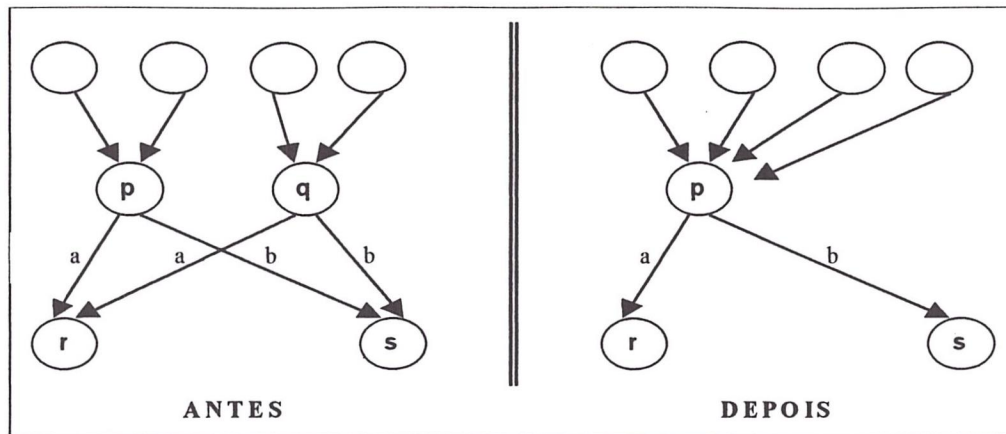
## 3.3. Minimização de Autômato Finito Determinístico

A minimização de autômato finito determinístico é um problema que vem sendo estudado desde os anos 50. Basicamente o problema consiste em encontrar o único autômato finito determinístico mínimo que aceita a mesma linguagem que um dado autômato finito determinístico. Algoritmos para resolver este problema são usados em aplicações desde construção de compiladores até minimização de circuitos de hardware [WAT93a].



De forma geral, algoritmos de minimização de autômatos trabalham identificando conjunto de estados equivalentes e “fundindo-os” em um só, numa técnica conhecida como *merge*. Dois estados  $p$  e  $q \in Q$  são equivalentes quando  $\delta(p, a) = \delta(q, a)$ , para todo  $a \in \Sigma$ . O *merge* dos estados  $p$  e  $q$  é feito adaptando-se a função de transição de tal forma que as transições que antes levavam a  $q$  devem agora levar a  $p$  (

Figura 5).



**Figura 5 – Merge de Dois Estados Equivalentes**

Existem quatro algoritmos clássicos de minimização de autômatos finitos determinísticos baseados na técnica de equivalência de estados:

- Algoritmo de Wood [WOO87] – de fácil entendimento, realiza a minimização do autômato em tempo  $O(|Q|^3)$ .
- Algoritmo de Aho-Setti-Ullman [AHO86] – capaz de realizar a minimização do autômato finito em tempo  $O(|Q|^2)$ , é o método mais utilizado na prática, dada sua boa relação custo-benefício (em termos de implementação). Calcula uma relação de equivalência nos estados, que indica quais estados são indistinguíveis
- Algoritmo de Hopcroft [HOP71; GRI73; WAT93a] – este é o melhor algoritmo conhecido para minimização de um autômato finito determinístico, em termos de tempo de execução:  $O(|Q| \log |Q|)$ . Também é o mais complexo.
- Algoritmo de Hopcroft-Ullman [HOP79] – Calcula o complemento da relação calculada pelo algoritmo de Aho-Sethi-Ullman. Como o algoritmo atravessa as transições do autômato finito determinístico na direção reversa, ele tem uma desvantagem de tempo de execução na maioria das implementações de autômato finito determinístico.

O único método de minimização que não se baseia na equivalência de estados é o Método de *Brzowski* [BRZ62; WAT93a]. Neste método, o autômato (que não precisa ser determinístico) é revertido, determinizado, revertido novamente e mais uma vez determinizado. Sua implementação é relativamente simples, uma vez que consiste apenas nas operações de reversão e determinização do autômato.

### 3.4. Pacotes para Manipulação de Autômatos Finitos

Existem vários pacotes para manipulação de autômatos finitos. Nesta seção serão apresentados alguns deles, que se destacam pela boa cobertura de autômato finito determinístico e pela implementação de algoritmos de minimização de autômatos.



Embora para o problema de representação de grandes léxicos seja mais eficiente utilizar métodos especiais do que os métodos genéricos (devido a todas as particularidades do autômato finito determinístico acíclico), estes pacotes são de grande valor para testes, protótipos e pesquisas sobre autômatos finitos.

### 3.4.1. FSA Utilities

*FSA Utilities* é uma coleção de utilitários para manipular autômatos e *transducers* de estado finito. Manipulações incluem determinização, minimização, composição, complementação, intersecção, etc. A ferramenta é implementada em *SICStus Prolog*, e existem várias ferramentas de visualização para o autômato [NOO97].

#### Representação

No *FSA Utilities*, um autômato finito é definido como um conjunto de cláusulas Prolog, usando as seguintes relações:

`start(State)`. `State` é o estado inicial. Deve haver pelo menos um estado inicial. Múltiplos estados iniciais são permitidos.

`final(State)`. `State` é um estado final. Pode haver qualquer número de estados finais.

`trans(State0, Sym, State)`. Define uma transição do estado `State0` para `State` associado ao símbolo `Sym`. Pode haver qualquer número de transições.

`jump(State0, State)`. Define uma transição vazia ( $\epsilon$ -transição) do estado `State0` para `State`. Pode haver qualquer número de saltos.

#### Minimização

A ferramenta suporta três diferentes implementações da operação de minimização:

1. Algoritmo de *Hopcroft e Ullman* (1979);
2. Método de *Brzozowski* (1962);
3. Algoritmo de *Hopcroft* (1971).

#### Considerações

Esta ferramenta é bastante útil para testes e experimentação de novas técnicas, embora não seja indicada para construção de programas práticos. Por ser implementado em Prolog, facilita a implementação de protótipos a partir de definições teóricas, além de a ferramenta poder ser utilizada na linha de comando, funcionando como filtro.

### 3.4.2. FIRE Lite

*FIRE Lite* é um pacote em C++ que implementa algoritmos de autômato finito e expressões regulares, desenvolvido por Bruce Watson na Universidade de Tecnologia de Eindhoven. O pacote provê classes de objetos e algoritmos de um nível suficientemente baixo tal que eles podem ser usados na maioria das aplicações que requerem autômato finito ou expressões regulares. Quase todos os algoritmos definidos em [WAT93a; WAT93b] são implementados [WAT96].

## Vantagens

A seguir são citadas algumas vantagens e diferenças do uso do pacote *FIRE Lite* em relação a outros pacotes do gênero [WAT96]:

- *FIRE Lite* não provê uma interface de usuário. Ao invés disso, *FIRE Lite* é estritamente uma biblioteca de classes e algoritmos para serem usados em aplicações;
- O pacote é implementado para ser eficiente. Ao contrário de outros, que são implementados com objetivos educacionais, a implementação usando *FIRE Lite* é suficientemente eficiente para poder ser utilizada em aplicativos práticos;
- *FIRE Lite* provê implementações de quase todos os algoritmos conhecidos de construção de autômato finito, ao contrário dos outros que geralmente apresentam apenas um.

## Minimização

O pacote inclui 5 algoritmos de minimização de autômato finito determinístico, a saber:

1. *Algoritmo de Brzozowski*;
2. *Algoritmo de Aho-Sethi-Ullman*;
3. *Algoritmo de Hopcroft-Ullman*;
4. *Algoritmo de Hopcroft* ;
5. *Algoritmo de B.W.* (algoritmo especial do autor [WAT96]);

## Considerações

Este pacote é particularmente interessante por três motivos: primeiramente, ele foi escrito pelo autor de excelentes taxionomias de construção e minimização autômato finito [WAT93a; WAT93b]; segundo, pela ênfase em eficiência, o que o torna atrativo para protótipos mais robustos; e, por fim, por ele ter sido desenvolvido em C++, a mesma linguagem que será usada neste projeto.

### 3.4.3. Grail

O pacote *Grail* [RAY93], desenvolvido na Universidade de Waterloo por Darrell Raymond e Derick Wood, é um ambiente de computação simbólica para máquinas de estados finitos, expressões regulares, e outros objetos de teoria de linguagem formal. Capaz de realizar conversão, minimização, complemento e várias outras operações sobre autômatos, *Grail* é indicado para ensino, além de ser eficiente. Ele pode manipular autômatos com cerca de 100,000 estados e dicionários de 20,000 palavras.

*Grail* apresenta duas interfaces:

1. Um conjunto de *programas filtros*: cada filtro implementa uma operação elementar em autômato finito ou expressão regular. Essas operações incluem conversões de expressões regulares para autômato finito, minimização de autômato finito, etc. Os filtros podem ser combinados para obter-se operações mais complexas. Esta interface satisfaz os dois primeiros objetivos (de três) do *Grail*: prover um mecanismo de pesquisa de algoritmos teóricos, e facilitar o ensino de teoria de linguagem.
2. Uma *biblioteca de classes C++* provendo uma grande variedade de objetos e algoritmos. Esta interface satisfaz o terceiro objetivo de *Grail*: um sistema eficiente para uso em aplicativos práticos.



## **Considerações**

*Grail* é um bom pacote para fins educacionais, mas não é capaz de tratar problemas de autômatos extremamente grandes como os usados em grandes léxicos de língua natural.

### **3.4.4. FSM Library**

Esse é um pacote desenvolvido pela divisão de pesquisa da AT&T para manipulação de autômatos finitos e *transducers*, com o objetivo de lidar com problemas de grande porte de forma eficiente.

Pode ser usado diretamente através de seus utilitários, ou por um programa em C, por exemplo, através de sua interface de programação. Implementa mais de 30 funções de manipulação de AF.

Maiores informações e licença para *download* estão disponíveis no *site* da *AT&T Research* em: <http://www.research.att.com/sw/tools/fsm>.

## **Considerações**

Essa ferramenta é a que mais se aproxima do sistema sendo proposto nesse trabalho, embora seja uma ferramenta de tratamento genérico de autômatos. Apesar da AT&T disponibilizar apenas as ferramentas em formato binário, e não a biblioteca de programação, essa ainda é uma ótima ferramenta para manipular autômatos de grande porte. Alguns resultados de tratamento de léxicos usando essa ferramenta são expostos na seção 6.4 (página 42).



## 4. Uso de AF para Representação de Léxicos

---

Nesse capítulo serão discutidos alguns trabalhos importantes relacionados ao uso de AF para representação de léxicos de língua natural. Na seção 4.1 está descrito o trabalho de Kowaltowski, Lucchesi e Stolfi (KLS), pesquisadores do IC-UNICAMP, que desenvolvem um excelente trabalho nessa área; a seção 4.2 resume o trabalho de Dominique Revuz sobre minimização de AF determinístico acíclico; por fim, a seção 4.3 cita alguns outros trabalhos relacionados, como o sistema INTEX do LADL/University Paris 7.

### 4.1. O Trabalho de K.L.S.

Inicialmente motivados pelo problema de implementar um eficiente aconselhador ortográfico para a Língua Portuguesa, Kowaltowski e Lucchesi (e depois também Stolfi), pesquisadores do IC-Unicamp, desenvolveram um importante trabalho sobre o uso de autômato finito para representação de grandes vocabulários [KOW93; KOW95a; KOW95b; KOW98].

O sistema desenvolvido por estes pesquisadores permitiu a representação de um léxico simples (apenas palavras, sem atributos) de cerca de 200.000 palavras em um autômato de 124Kb [KOW93], e recentemente um léxico completo (palavras com atributos sintáticos e gramaticais) de cerca de 2.000.000 de entradas foi codificado em um autômato de menos de 1,1Mb [KOW98].

#### 4.1.1. Implementação do Autômato

A construção do autômato ocorre de acordo com o seguinte algoritmo [KOW93]:

---

```
Procedimento Construir_Autômato(Vocabulário V)
Início
  A ← Autômato_Vazio;
  Repita
    Enquanto A "não estiver cheio" faça
      "inclua a próxima palavra de V em A";
      A ← Autômato_Mínimo(A);
  Até que "não existam mais palavras em V";
Fim.
```

---

Após a primeira execução do laço *enquanto...faça* o autômato *A* é na verdade uma árvore digital, como pode ser observado na Figura 6. Esta árvore pode crescer muito rapidamente: para o léxico de cerca de 200.000 palavras citado anteriormente, a árvore teria mais de 600.000 vértices. Por isso o laço externo *repita...até* é necessário. O processo de minimização aproveita o fato de o autômato ser acíclico e usa um algoritmo de ordem de tempo linear no tamanho do autômato. Após a minimização, o mesmo autômato toma a forma da Figura 7.

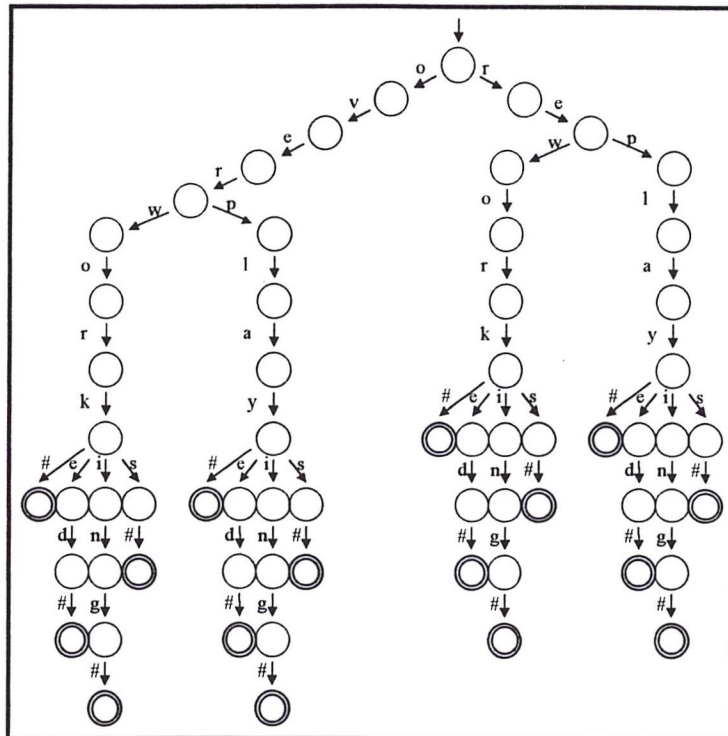


Figura 6 – Autômato para todas as formas dos verbos *rework*, *replay*, *overwork* e *overplay*

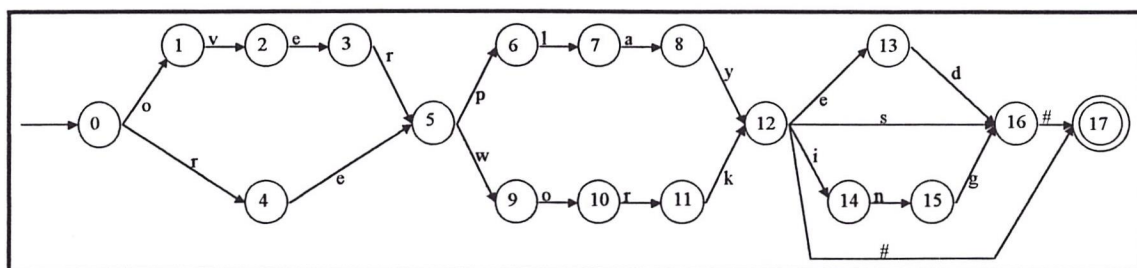


Figura 7 – Autômato Finito Determinístico Acíclico Mínimo criado a partir da Figura 6

Cada estado é representado como um vetor de  $M$  entradas (onde  $M$  é o tamanho do alfabeto), sendo que a maioria das entradas corresponde a transições não existentes. Aproveitando-se deste fato, pode-se definir uma estrutura melhor, onde cada estado está associado a um vetor de  $M$  bits que seleciona as transições válidas para o estado.

O algoritmo de busca para uma dada palavra é muito eficiente. Começando pelo estado inicial, ele percorre o autômato utilizando letras consecutivas da palavra para selecionar as transições, até que um estado final é alcançado ou não existam mais transições válidas (em outras palavras, ou a palavra pertence ao vocabulário ou não pertence).

#### 4.1.2. Autômato Binário

A experiência demonstra que o autômato para um vocabulário de língua natural é muito esparsa, no seguinte sentido: para a grande maioria dos estados em  $q \in Q$  e para quase todas as letras  $a \in \Sigma$ , a função  $\delta(q, a)$  não está definida (ou seja, levam ao estado de rejeição). No caso da Língua Portuguesa, por exemplo, quase metade dos estados têm apenas uma transição válida, e aproximadamente 70% dos estados tem no máximo duas transições válidas. Essa característica leva a um esquema de representação onde cada estado  $q$  é representado por uma lista de valores que representam suas transições válidas [KOW95a].

Com essa representação, pode-se ver os nós das listas de transições como “estados” de uma espécie de autômato, onde cada estado tem apenas uma transição de *sucesso* (associada a



uma letra), e uma transição de *falha* (representada por um ponteiro para o próximo nó). Chama-se este dispositivo de *autômato binário* (ver abaixo). Considera-se também, para esse autômato, a existência de apenas um estado final, denotado  $q_a$  (estado de aceitação), e apenas um estado de rejeição (chamado de  $q_r$ ). Todas as transições de  $q_a$  e  $q_r$  levam a  $q_r$ , sendo este o único estado a admitir ciclos.

---

```

Estado = ponteiro_para Estrutura_Estado;
Estrutura_Estado = início_estrutura
    L : Letra;
    S : Estado;    {transição de sucesso}
    F : Estado;    {transição de falha}
fim_estrutura;
Autômato_Binário = início_estrutura
    Estado_Inicial : Estado;
fim_estrutura;

```

---

Usando a estrutura definida acima, pode-se calcular a função de transição do autômato ( $\delta : Q \times \Sigma \rightarrow \Sigma$ ) da seguinte forma:

---

```

Função delta(q: Estado, a: Letra): Estado;
Início
    Se q = qr então delta := qr
    senão se q.L = a então delta := q.S
    senão delta := delta(q.F, a);
Fim.

```

---

## Representação Vetorial

Em algumas situações é interessante trocar a estrutura dinâmica por uma representação estática baseada em *vetor*. Para isso, enumera-se os estados do autômato binário com números inteiros consecutivos, e armazena-se todos as estruturas *Estrutura\_Estado* em um vetor indexado pelo número do estado.

Neste esquema, é conveniente atribuir o índice 0 para o estado de rejeição ( $q_r$ ), e o índice 1 para o estado de aceitação ( $q_a$ ), e desta forma estas duas entradas podem ser omitidas no vetor. Assim, se o  $i$ -ésimo estado é  $q$ , o  $i$ -ésimo elemento do vetor contém a tripla ( $L, S, F$ ), onde os campos  $S$  e  $F$  agora são números de estados, e não mais ponteiros.

Usando-se o conceito de ordenação topológica, pode-se também assumir a existência de uma ordem total no conjunto de estados  $Q$  do autômato tal que  $\delta(q, a) \leq q$ , para quaisquer  $q \in Q$  e  $a \in \Sigma$ . Para isso, ao construir a representação vetorial do autômato binário, enumera-se os estados de  $Q$  como  $\{0, 1, \dots, n\}$  de forma que 0 é o estado de rejeição, 1 é o estado de aceitação,  $n$  é o estado inicial, e os campos  $S$  e  $F$  na entrada  $i$  do vetor ( $i > 1$ ) são ambos estritamente menores que  $i$ .

## Considerações

Este tipo de autômato é atraente pela sua simplicidade de implementação e manipulação. O algoritmo de minimização pode trabalhar simplesmente identificando entradas repetidas no vetor e realizando um *merge*, embora o uso de um algoritmo de reordenação das listas de transições possa tornar o processo de minimização mais eficiente. Mas encontrar um autômato binário verdadeiramente mínimo parece ser um problema *NP-completo* [KOW95a].



Kowaltowski, Lucchesi e Stolfi sugerem nesse artigo um algoritmo heurístico para minimização de autômato binário.

### 4.1.3. Resultados Práticos

Os testes principais foram aplicados a um vocabulário de 81.000 palavras do Inglês e outro de 206.000 palavras do Português [KOW93]. Apesar deste último ser maior, o autômato que o representa é menor que o autômato para o vocabulário do Inglês, devido ao fato do grande número de formas derivadas na Língua Portuguesa<sup>4</sup>.

A Tabela 3 apresenta algumas estatísticas desses testes, comparando-os em relação ao número de palavras (e tamanho do arquivo texto), e ao número de estados e transições e tamanho dos autômatos correspondentes. Também são indicados os resultados obtidos pela compactação dos dados utilizando os utilitários PKZIP e PKPAK (embora o autômato não seja efetivamente um mecanismo de compactação, mas sim de representação eficiente).

**Tabela 3 – Estatísticas para os Vocabulários e os Autômatos [KOW93]**

|                    | Inglês | Português |
|--------------------|--------|-----------|
| <b>Vocabulário</b> |        |           |
| Palavras           | 81.142 | 206.786   |
| Kbytes             | 858    | 2,389     |
| PKPAK              | 313    | 683       |
| PKZIP              | 253    | 602       |
| <b>Autômato</b>    |        |           |
| Estados            | 29.317 | 17.267    |
| Transições         | 67.709 | 45.838    |
| Kbytes             | 203    | 124       |
| PKPAK              | 173    | 105       |
| PKZIP              | 183    | 109       |

Também foi observado que o crescimento do autômato era quase linear caso as palavras fossem inseridas por ordem alfabética, o que não acontecia caso as palavras fossem inseridas em uma ordem randômica (neste caso, o autômato poderia chegar a ter até o dobro do tamanho, durante as inserções, para no final (após a minimização) se reduzir ao mesmo autômato da inserção alfabética) [KOW93].

### 4.1.4. Codificação de Informações Lexicais no Autômato

A codificação das informações lexicais deve ser bem organizada e estruturada de acordo com uma melhor taxa de aproveitamento do autômato. Considerando cada entrada do léxico como uma *string*  $w$  (contendo a palavra a ser armazenada e sua classificação), quando esta *string* for inserida no autômato haverá uma melhor taxa de aproveitamento se:

1. O prefixo de  $w$  puder ser reutilizado;
2. O sufixo de  $w$  puder ser reutilizado;

A condição 1 está implicitamente definida, pois trabalhamos com um autômato determinístico, então se  $xy$  e  $xz \in \Sigma^*$  são palavras quaisquer que compartilham um mesmo prefixo  $x$  então os estados do autômato relativos às transições para o prefixo  $x$  serão comuns às

<sup>4</sup> Por exemplo, um verbo regular do Português apresenta 51 flexões distintas, cujos sufixos podem ser compartilhados entre todos os outros verbos regulares.

duas palavras. Analogamente, a condição 2 ocorre para o sufixo comum  $z$  de duas palavras  $xz$  e  $yz \in \Sigma^*$  quaisquer, após o autômato ser minimizado.

Os atributos gramaticais de uma palavra podem ser codificados como *tokens* ligados à palavra. Para representar o substantivo *menino* e suas flexões, por exemplo, pode-se usar um esquema de codificação como o exemplificado na Tabela 4 [KOW98].

Desta forma, para se saber qual a classe gramatical da palavra *menina*, por exemplo, basta procurar por “*menina:*” no autômato e depois enumerar todos os sufixos possíveis.

**Tabela 4 – Exemplo de Entradas Simples do Léxico**

| ENTRADA    | SIGNIFICADO                                  |
|------------|----------------------------------------------|
| menino:as  | Substantivo masculino singular ( <i>sa</i> ) |
| menina:sb  | Substantivo feminino singular ( <i>sb</i> )  |
| meninos:sc | Substantivo masculino plural ( <i>sc</i> )   |
| meninas:sd | Substantivo feminino plural ( <i>sd</i> )    |

Entretanto, para algumas pesquisas é preciso incluir mais uma informação às entradas do léxico: a forma canônica. Uma forma eficiente de se codificar esta informação é adicioná-la à *string* da entrada léxica, de forma a distingui-la da palavra da entrada pelos últimos caracteres (Tabela 5). Na entrada “*meninas:sd:2o*”, por exemplo, a forma canônica foi obtida removendo-se os últimos 2 caracteres de “*meninas*” e acrescentando-se “*o*”, e este procedimento é denotado por “*:2o*” na *string* da entrada léxica.

**Tabela 5 – Exemplo de Entradas do Léxico com a Forma Canônica**

| ENTRADA       | FORMAÇÃO DA CANÔNICA      |
|---------------|---------------------------|
| menino:sa:0   | menino – “” + “” = menino |
| menina:sb:1o  | menina – a + o = menino   |
| meninos:sc:1  | meninos – s + “” = menino |
| meninas:sd:2o | meninas – as + o = menino |

Por fim, para podermos responder perguntas como: “*Qual o plural de menina?*”, temos que obter a classificação e forma canônica de “*menina*” (“*:sb*” e “*menino*”, respectivamente), e então procurar pela entrada “*:sd*” (feminino plural) da canônica “*menino*”. Para que esta última operação seja efetuada de forma eficiente, deve-se incluir uma entrada adicional para cada entrada do léxico, a *entrada reversa* (

Tabela 6).

Na entrada reversa, utiliza-se a mesma regra da entrada normal para obtenção da palavra a partir da canônica: subtrai-se zero ou mais letras do final da canônica e acrescenta-se zero ou mais caracteres para obter-se a palavra. O fato de duplicarmos o número de entradas lexicais não faz duplicar o tamanho do autômato, pois os afixos das entradas são altamente reutilizados.



Tabela 6 – Exemplo de Entradas Completas do Léxico

| ENTRADA NORMAL | ENTRADA REVERSA |
|----------------|-----------------|
| menino:sa:0    | menino::sa:0    |
| menina:sb:1o   | menino::sb:1a   |
| meninos:sc:1   | menino::sc:0s   |
| meninas:sd:2o  | menino::sd:1as  |

Seguindo-se este sistema, pode-se definir métodos de representação para vários outros atributos gramaticais, como exposto em [KOW98].

## 4.2. Algoritmo de Minimização de Revuz

Dominique Revuz (Universidade de Paris) apresenta em [REV92] um algoritmo linear para minimização de um autômato finito determinístico acíclico, que pode ser usado, em particular, para minimizar um autômato que representa um léxico.

Cada estado do autômato é associado a uma *string* descrevendo o autômato reduzido iniciado neste estado. Trabalhando nos níveis maiores (onde o nível é a maior distância até um estado final), o algoritmo gera rótulos que são criados a partir dos rótulos de seus estados seguintes. A cada nível, uma ordenação lexicográfica é aplicada à lista de rótulos e todos os estados com rótulos idênticos sofrem *merge* (tais estados são equivalentes). A associação dos rótulos é feita apenas uma vez em cada estado, e a ordenação é linear; assim, a complexidade geral é linear nos números de transições.

### 4.2.1. O Algoritmo de Minimização

Primeiramente é necessário definir a função *peso* em um autômato acíclico: para cada estado  $s$  do autômato, coloca-se  $\text{peso}(s) = \max\{|w| \text{ tal que } s.w \text{ é final}\}$  ( $\text{peso}(s)$  = comprimento do maior caminho iniciando em  $s$  e chegando a um estado final). Esta função cria uma partição  $P$  de  $Q$ ;  $P_i$  irá denotar o conjunto de estados de peso  $i$ . Dizemos ainda que o conjunto  $P_i$  é *distinguível* se não existem estados equivalentes em  $P_i$ .

Assim, se cada partição  $P_j$  com  $j < i$  é distinguível, então dois estados  $p$  e  $q \in P_i$  são ditos equivalentes se e somente se para qualquer letra  $a \in A$  a igualdade  $q.a = p.a$  é válida. Um algoritmo simples de minimização é definido a partir dessa propriedade: primeiro criam-se as partições pelo peso, o que pode ser realizado por uma função de  $O(|\delta|)$ . Segue o algoritmo geral de Revuz:

---

**Calcular  $P$**   
 Para  $i := 0$  até  $\text{peso}(q_0)$  faça  
 Início  
     "ordene os estados de  $P_i$  pelas suas arestas";  
     "faça *merge* para os estados equivalentes";  
 Fim.

---

O ponto crítico deste algoritmo é a rotina de ordenação dos estados, uma vez que o particionamento ocorre apenas uma vez e pode ser realizado em tempo linear no número de transições. Assim, dado um esquema de ordenação de tempo  $O(f(n))$ , o algoritmo é capaz de minimizar um autômato finito acíclico em tempo  $O(|\delta| + \sum f(|P_i|))$ .



### 4.2.2. O Algoritmo de Ordenação

A implementação do algoritmo de minimização pode se aproveitar do fato de que não é necessário ordenar os estados, mas apenas distingui-los dois a dois. A ordenação lexicográfica que será descrita baseia-se neste fato, e é composta basicamente de repetidas *bucket sorts*.

#### **Bucket Sort**

Ordena-se uma sequência  $a_1, a_2, \dots, a_n$  de inteiros ( $1 \leq a_i \leq m$ ) da seguinte forma:

(1) Cria-se um vetor de filas vazias:  $Queue[1..m]$

(2) Rastreia-se a sequência  $a_1, a_2, \dots, a_n$  colocando o elemento  $a_i$  em  $Queue[a_i]$

(3) Concatenam-se as filas de  $Queue$  para se obter a sequência ordenada.

A complexidade de tempo é  $O(n + m)$ , e o espaço necessário é  $O(n + m)$ .

#### **Ordenação Lexicográfica**

Ordena-se uma sequência  $A_1, A_2, \dots, A_n$  de  $k$ -uplas  $(a_{i1}, a_{i2}, \dots, a_{ik})$ , onde  $a_{ij}$  está na faixa de 1 a  $m$ . Primeiramente aplica-se *bucket sort* à sequência baseado no  $k$ -ésimo inteiro de cada  $k$ -upla. A nova sequência, uma permutação da primeira, está então ordenada de acordo com o  $k$ -ésimo inteiro. A próxima ordenação, no  $(k-1)$ -ésimo inteiro, parte dessa ordem. Então, por indução, a sequência final, após a execução de  $k$  *bucket sorts*, é uma permutação ordenada da sequência inicial. A complexidade de tempo é  $O(k \times (n + m))$ , e o espaço necessário é  $O(n + m)$ .

A generalização para sequências de palavras de tamanhos variáveis, na faixa de 1 a  $L_{max}$ , é feita aplicando-se *bucket sort* à palavra pelo seu comprimento e então executando a ordenação lexicográfica como descrito, com a ação adicional de concatenar as palavras de comprimento  $L$  ao início da sequência antes do  $(L_{max} - L)$ -ésimo *bucket sort* (estas palavras estão ordenadas pela inexistência de uma  $(L+1)$ -ésima letra) [REV92]. Dominique Revuz descreve em seu artigo um algoritmo completo para a ordenação lexicográfica.

## 4.3. Outros Trabalhos Relacionados

Aqui serão citados outros trabalhos existentes sobre o uso de autômato finito para representação de léxicos e dicionários.

### 4.3.1. “The World’s Fastest Scrabble Program”

Em 1988, Andrew Appel, um professor assistente da Universidade de Princeton, publicou um artigo na *Communication of the ACM* intitulado “The World’s Fastest Scrabble Program” [APP88], que descreve as estruturas de dados, estratégias de busca, e heurísticas especiais que produzem um programa extremamente rápido para jogar *Scrabble*<sup>5</sup>. Tal rapidez é obtida pela construção de uma *Trie* [LAN96] e depois de um DAWG (*Directed Acyclic Word Graph*, uma forma de autômato finito determinístico acíclico) para armazenar o léxico.

Este artigo é um dos primeiros trabalhos publicados sobre representação de léxicos usando autômatos.

### 4.3.2. “Verifica” – Uma Ferramenta de Auxílio à Redação

Dissertação de mestrado de Hellen Pacheco (DCC-UFMG), sobre o uso de autômato finito para implementação de um verificador ortográfico, chamado *Verifica*, baseia-se em parte nos trabalhos de K.L.S. (seção 0, página 17) e Revuz (seção 4.2, página 22).

---

<sup>5</sup> Um jogo no qual palavras são formadas por letras em padrões semelhantes a um jogo de palavras cruzadas.

“O sistema testa a ocorrência das palavras da entrada em um vocabulário da Língua Portuguesa armazenado em um autômato finito determinístico acíclico. (...) Visto que um verificador ortográfico tradicional apresenta deficiências, foi estudada uma forma de se refinar a análise ortográfica a partir da análise da estrutura da frase. Foi implementado, então, um algoritmo que atribui classes sintáticas às palavras da língua. Esse é o primeiro componente de um analisador sintático para o português segundo uma nova abordagem, a abordagem funcional, que é independente de um léxico.” [PAC96]

O sistema *Verifica* pode ser executado pela linha de comando, através de uma interface gráfica (criada em *Tcl-Tk*), ou ainda por meio de uma interface *on-line* disponível pela WWW (<http://www.dcc.ufmg.br/verifica>).

### 4.3.3. AUMAC/NILC

Projeto de graduação desenvolvido por este aluno como parte dos créditos necessários para obtenção do título de Bacharel em Ciência da Computação, em 1997; foi baseado nos trabalhos de K.L.S. (seção 0, página 17) e Revuz (seção 4.2, página 22), e descreve a implementação de um protótipo para representação de um léxico simples (apenas uma lista de palavras, sem atributos) usando autômato finito. O sistema implementado neste trabalho, chamado *AUMAC/NILC*, foi implementado em C++ ANSI e divide-se em 2 módulos básicos:

1. *O Módulo Gerador*

Responsável pela criação e compactação do autômato. Necessita de bastante recurso computacional (disco, memória e processamento), e é executado apenas uma vez.

2. *O Módulo de Consulta*

Responsável pela utilização do autômato compactado. Os requisitos computacionais são mínimos, possibilitando seu uso eficiente em praticamente qualquer sistema.

Apesar do protótipo desenvolvido ser bastante rudimentar, os testes iniciais comprovaram a eficiência do método utilizado.

### **Considerações sobre os Vocabulários**

A fim de que o processo de “compactação” do autômato seja mais eficiente, deve-se otimizar também o vocabulário de entrada. Estando essa lista de palavras ordenada, alguns algoritmos podem ser simplificados, mas apenas isto não basta.

É necessário definir quantos e quais serão os símbolos distintos antes da formação do autômato. Os símbolos podem ser representados utilizando-se *códigos de Huffman* [LAN96], ou símbolos pouco utilizados podem ser representados como uma combinação de 2 ou mais outros símbolos. Percebe-se assim que uma boa análise do vocabulário a ser representado é essencial.

**Os acentos também podem ser tratados de forma especial. Optando-se por uma representação alternativa das palavras acentuadas, pode-se aumentar a taxa de minimização do autômato. Uma dessas alternativas pode ser codificar os acentos no final da palavra, com um esquema semelhante ao mostrado na Tabela 7. Nessa proposta, os acentos são identificados por um conjunto de duplas (*posição, tipo*), onde *posição* representa a posição da letra acentuada (da direita para a esquerda), e *tipo* representa código do acento, conforme a**

Tabela 8.



Tabela 7 – Esquema de Codificação de Acentos

| Tipo de acento | Código |
|----------------|--------|
| Agudo          | a      |
| Grave (crase)  | g      |
| Til            | t      |
| Trema          | u      |
| Circunflexo    | c      |

Com a codificação percebe-se que houve uma taxa maior de reutilização dos prefixos (demonstrado em *itálico* no exemplo mostrado na

Tabela 8). Este tipo de código pode ser bastante útil no caso das flexões verbais, visto que cada verbo do português apresenta 51 flexões distintas e a remoção dos acentos pode fazer com que todos utilizem um prefixo comum. O esquema de codificação dos acentos ao final da palavra não chega a causar uma sobrecarga no autômato, pois os estados que representam os acentos também serão reutilizados, o que significa que palavras com igual codificação de acentos terão seus estados finais compartilhados (como mostrado em **negrito** nas palavras *dormirá* e *sabiá* do exemplo).

Este tipo de otimização pode ser feito fora do sistema de criação e minimização do autômato, através da aplicação de um filtro de conversão na entrada de dados do sistema.

Tabela 8 - Exemplo de Codificação dos Acentos ao Final da Palavra

| Palavra normal           | palavra codificada                    |
|--------------------------|---------------------------------------|
| <i>dormira</i>           | <i>dormira#</i>                       |
| <i>dormíamos</i>         | <i>dormiamos!5a#</i>                  |
| <i>dormirá</i>           | <i>dormira!<b>1a#</b></i>             |
| <i>sabiá</i>             | <i>sabia!<b>1a#</b></i>               |
| <i>órgão</i>             | <i>orgao!5a2t#</i>                    |
| <i>orgânico</i>          | <i>organico!5c#</i>                   |
| <i>maçã</i> <sup>6</sup> | <i>maca!2c1t#</i>                     |
| <i>qüinqüelíngüe</i>     | <i>quinquelingue!<b>Cu8u5a2u#</b></i> |

### O Módulo Gerador

Este é o módulo principal do sistema, e responsável pela criação, minimização e representação do autômato. Ele necessita de uma máquina de alto desempenho e bastante memória para ser executado, uma vez que ele instancia toda a estrutura da árvore do autômato em memória principal antes de executar o processo de minimização. É formado por 3 unidades básicas:

<sup>6</sup> Neste esquema de representação, optou-se por codificar a cedilha com o código 'c', o mesmo utilizado para os acentos circunflexos, sem implicar em nenhuma confusão.



- *AUTOMATA* – contém a estrutura do autômato, e é responsável pelo processamento e inserção das palavras na estrutura.
- *MINIMIZER* – unidade responsável pela minimização da estrutura *AUTOMATA*.
- *VECTOR* – unidade responsável pela criação de uma estrutura linear vetorial representante do autômato da estrutura *AUTOMATA*.

## O Módulo de Consulta

Este é o módulo que faz a interface entre o autômato binário e o aplicativo, sendo responsável por carregar a estrutura do autômato para a memória e executar as pesquisas. Basicamente este módulo consiste na implementação do algoritmo abaixo:

---

```

VerificaPalavra ( palavra ) {
    a = 1
    p = 1
    enquanto a > 0 {
        se vetor[a].símbolo = palavra[p] {
            a = vetor[a].sucesso
            se ( a = 0 ) e ( p = tamanho(palavra) )
                retorne VERDADEIRO
        } senão {
            a = vetor[a].falha
            se a = 0
                retorne FALHA
        }
        p = p + 1
    }
}

```

---

O módulo de consulta pode ser escrito com extensões específicas para cada aplicação. Para um corretor ortográfico, por exemplo, não bastaria apenas a informação sobre se uma palavra é ou não reconhecida pelo autômato, pois caso ela não seja reconhecida, algumas sugestões devem ser dadas com base nas transições *válidas* do autômato.

## Testes

Foram realizados alguns testes com o protótipo desenvolvido. Para estes testes foi utilizado um subconjunto do vocabulário da língua portuguesa<sup>7</sup>. Serão apresentados a seguir dois dos testes realizados.

Para o primeiro teste foi utilizado um vocabulário de 5.000 palavras da língua portuguesa<sup>8</sup>, e o protótipo apresentou os seguintes resultados:

---

```

AUTOMATA...
5000 words, 17292 states, 17291 transitions, 21 levels
MINIMIZER...
Done. (16461 optimizations -> 831 states) [time = 3574]
>>>> final automata has 2242 clusters.

```

---

<sup>7</sup> Apesar do trabalho ter sido focado na representação de vocabulários da Língua Portuguesa, o sistema implementado pode trabalhar com vocabulários de quaisquer línguas.

<sup>8</sup> O vocabulário consistiu das primeiras 5000 palavras do léxico da língua portuguesa desenvolvido pelo NILC.

Pelo teste, que levou menos de 4 segundos para ser executado, percebe-se a enorme redução do número de transições (de 17.292 para 831, redução de cerca de 95%). O vetor final de representação ficou com apenas 2.242 elementos (11.210 bytes, segundo a forma de representação não otimizada utilizada pelo protótipo<sup>9</sup>), o que representa uma compactação de aproximadamente 80% em relação ao arquivo de entrada (54.634 bytes).

O outro teste foi realizado com as primeiras 1000 palavras do teste anterior:

---

AUTOMATA...

1000 words, 3440 states, 3439 transitions, 18 levels

MINIMIZER...

Done. (3240 optimizations -> 199 states) [time = 2937]

>>>> final automata has 521 clusters.

---

Neste teste, com 20% das palavras do exemplo anterior, a redução no número de estados foi de cerca de 95%, e o vetor final ocupou 2.605 bytes (o arquivo original ocupava 11.025 bytes, portanto a “compactação” foi de 77%).

#### 4.3.4. INTEX

O INTEX é um ambiente de desenvolvimento lingüístico desenvolvido por Max Silberstein (LADL, Jussieu-FR) que permite aos usuários construir AFs descrevendo extensas linguagens e aplicá-los a grandes textos (dezenas de milhões de palavras) em tempo real. Uma característica importante do INTEX é que textos, dicionários e gramáticas são todos representadas por *Finite State Transducers* (FSTs).

Desenvolvido desde 1992, sua atual versão (4.0) é capaz de rodar em máquinas Windows (95/NT). Mais de 30 centros de pesquisa usam o INTEX como ferramenta para pesquisa em várias áreas (lingüística computacional, terminologia, análise de *corpus*, recuperação de informações, etc).

Maiores informações sobre o sistema podem ser obtidas no *site* do LADL: <http://www.ladl.jussieu.fr>.

---

<sup>9</sup> No protótipo são gastos 5 bytes para representar cada elemento do vetor.



## 5. Um Sistema de Representação de Grandes AF Determinísticos Acíclicos

O sistema proposto basicamente recebe um conjunto de palavras como entrada, processa-as e cria um arquivo de saída que contém uma representação otimizada do autômato gerado (

Figura 8). Seu objetivo é ser simples de usar e, ao mesmo tempo, altamente extensível e integrável com outras aplicações e filtros.

O comportamento é simples: as palavras do arquivo de entrada são inseridas uma a uma no autômato, até que o gasto de memória atinja um limite pré-estabelecido, ocorrendo então uma minimização. Isso se repete até que todas as palavras sejam inseridas no autômato. Após cada minimização, e também no final do processamento de todas as palavras, podem ser geradas algumas estatísticas sobre a performance do sistema.

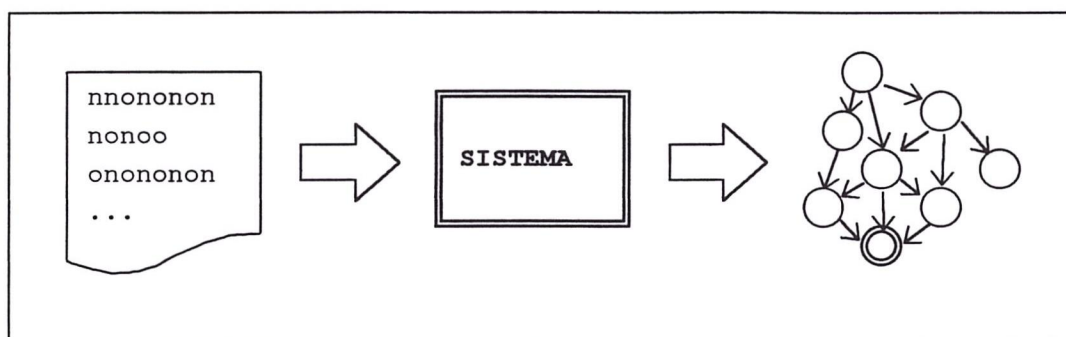


Figura 8 – Visão geral do sistema

Um protótipo foi desenvolvido em C/C++ e testes foram executados em uma estação de trabalho *Sun Ultra-I*, com 128Mb de memória e em um PC Linux com 64Mb de memória.

### 5.1. Estruturas de Dados

A estrutura de dados principal do sistema é a que representa os estados do autômato. Essa estrutura, denominada *State*, e as outras necessárias, são descritas a seguir:

|                   |                     |                                                              |
|-------------------|---------------------|--------------------------------------------------------------|
| <u>Byte</u>       | unsigned char       | // Armazena um valor de 8 bits (0 a 255)                     |
| <u>Symbol</u>     | char                | // Tipo de dado do símbolo ( <i>char</i> , 7 bits + sinal)   |
| <u>TripleByte</u> |                     |                                                              |
| Byte              | byte0, byte1, byte2 | // Armazena um valor de 24 bits (0 a 16.777.215)             |
| <u>State</u>      |                     |                                                              |
| Byte              | flags               | // Flags para estado terminal, visitado, <i>proper</i> , etc |
| TripleByte        | id                  | // Identificador único do estado                             |

|                   |                        |                                                |
|-------------------|------------------------|------------------------------------------------|
| Byte              | height                 | // Altura do estado                            |
| State*            | level_next, level_prev | // Ponteiros para os estados vizinhos de nível |
| Transition*       | trans                  | // Transições do estado                        |
| BinTree*          | ref                    | // Estados que trazem à esse estado            |
| <u>Transition</u> |                        |                                                |
| Symbol            | symbol                 | // Símbolo associado à transição               |
| State*            | destination            | // Estado para onde a transição leva           |
| Transition*       | next                   | // Ponteiro para a próxima transição do estado |
| <u>BinTree</u>    |                        |                                                |
| Word              | key                    | // Chave para o nó                             |
| State*            | state                  | // Dado do nó da árvore binária                |
| BinTree*          | left, right            | // Sub-árvores da esquerda e direita           |

A estrutura *State* ocupa apenas 21 bytes<sup>10</sup>, cada *Transition* ocupa 9 bytes, e cada *BinTree* 14 bytes. Como cada nova *Transition* leva à criação de um novo nó *BinTree*, temos que cada transição do autômato implica em um gasto de 23 bytes (9+14). Isso nos leva à seguinte fórmula de uso de memória<sup>11</sup>, onde  $s$  representa o número de estados do autômato, e  $t$  o número de transições:

$$m = 21s + 23t$$

Como o sistema foi desenvolvido de forma a permitir que as minimizações ocorram de forma intercalada com as inserções, ele pode rodar em estações de menor capacidade de memória, mesmo para grandes conjuntos de dados.

A estrutura de dados que representa o autômato é a *Automata*. Ela e as outras estruturas associadas são descritas a seguir:

|                 |                          |                                                   |
|-----------------|--------------------------|---------------------------------------------------|
| <u>List</u>     |                          |                                                   |
| State*          | state                    | // Referência para o Estado                       |
| List*           | next, prev               | // Links da lista                                 |
| <u>Hash</u>     |                          |                                                   |
| List*           | head[65536], tail[65536] | // Itens da tabela hash (head/tail de cada lista) |
| <u>Automata</u> |                          |                                                   |
| State*          | start                    | // Estado inicial do autômato                     |
| State*          | level[256]               | // Vetor das listas de níveis                     |
| Hash            | hash                     | // Tabela hash usada na minimização               |
| int             | n_states                 | // Número de estados do autômato                  |
| int             | n_trans                  | // Número de transições do autômato               |
| int             | n_words                  | // Número de palavras inseridas                   |
| int             | n_minimize               | // Número de minimizações efetuadas               |

<sup>10</sup> Considerando que cada ponteiro ocupa 32 bits (4 bytes), o que acontece na maioria das plataformas atuais.

<sup>11</sup> A fórmula leva em consideração que os dados serão alinhados byte a byte em memória, sem otimizações de acesso geralmente feitas pelo compilador, e sem *overheads* nas estruturas alocadas pelo sistema.



A estrutura *Automata* ocupa pouco mais que 500Kb de memória. O grande tamanho se deve à tabela *hash* embutida. Durante a fase de minimização, essa tabela é preenchida para cada diferente nível do autômato, criando listas temporárias. Cada nó da lista ocupa 12 bytes, e existirão no máximo  $w$  nós em cada fase da minimização (onde  $w$  é o número de palavras já inseridas no autômato), no pior caso. Assim, um conjunto de 1.000.000 de palavras criará uma estrutura temporária de, no máximo, 12Mb (em condições ideais de gasto de memória).

Por fim, a estrutura final de representação do autômato (na verdade representa apenas as suas transições), é a *Vector*. Essa estrutura é a representação mais compacta criada pelo sistema, e é ela que será utilizada nas posteriores aplicações.

Vector[]

|            |             |                                 |
|------------|-------------|---------------------------------|
| Symbol     | symbol      | // Símbolo que leva à transição |
| TripleByte | destination | // "Estado" de destino          |

A estrutura *Vector* é criada somente após todas as palavras terem sido inseridas no autômato, e este já tiver sido minimizado. Cada estado do autômato é "desmembrado" em suas transições, e estas representadas sequencialmente na estrutura *Vector*. O bit mais significativo do *destination* é usado para marcar a última transição de cada estado.

O tamanho final, em *bytes*, dessa representação do autômato é de 4 vezes o número de transições. Um sistema de compactação externo, como o *gzip*, pode ser usado para compactar essa estrutura, mas quando ela for usada pela aplicação ela deverá ser descompactada. Mesmo assim essa abordagem é interessante, por exemplo, no caso de uma aplicação *Web* que tem que fazer o *download* de seu dicionário.

## 5.2. Algoritmos Propostos

Os principais algoritmos envolvidos no sistema são: inserção de uma nova palavra, minimização do autômato e criação da representação final. Os algoritmos serão expostos no formato de pseudocódigo, a começar pelo de inserção (Figura 9).

---

```

insert( String word ) {                                     // insere word no autômato

    State *s, *actual = start                               // inicializa variáveis
    Int i = 0
    Int h = lenght( word )                                  // comprimento da palavra

    while ( h >= 0 ) {                                       // tenta aproveitar prefixo da palavra
        if ( h > actual.height )                             // acerta altura, se necessário
            actual.height = h
        h --

        s = actual.follow( word[i] )

        if ( s ) {
            while ( s->ref.count() > 1 ) {                   // precisa clonar estado?
                clone_state( s )                             // cria clone(s) do estado destino
                delete_state( s )                             // e apaga o estado original
                s = actual.follow( word[i] )
            }

            actual = s
            i ++
        }
    }

```

```

        } // end if
        else
            break
    } // end while

    while ( word[i] ) {
        s = create_state()
        create_link( actual, word[i], s )
        s.height = h
        actual = s
        i ++
        h --
    } // end while
    n_words ++
    if ( n_states > THRESHOLD )
        minimize()

} // end proc

clone_state( State s, [BinTree bt] ) {
    if ( undefined(bt) ) {
        fix_level_ptrs( s )
        bt = s.ref
    }
    if ( bt.left )
        clone_state( s, bt.left )
    if ( bt.right )
        clone_state( s, bt.right )
    State parent = bt.state
    List list( parent.trans )
    while ( t = list.get() ) {
        State s2 = create_state(s)
        add_level_ptrs( s2 )
        parent.change_trans(t, s2)
    }
} // end proc

```

// cria estados para o radical e/ou sufixo  
 // cria um novo estado  
 // faz as devidas ligações  
 // acerta altura, se necessário  
 // incrementa o número de palavras  
 // precisa minimizar?  
 // então minimiza.  
 // clona o estado em questão  
 // chamada original da função...  
 // inicializa variáveis  
 // chame recursivamente para  
 // todos os estados que têm  
 // transição para o estado em clonagem  
 // "estado pai"  
 // cria lista das transicoes do parent  
 // para todas as transicoes...  
 // cria estado parecido com s  
 // acerta ponteiros do nivel  
 // muda destino da transição

**Figura 9 - Processo de inserção de uma palavra no autômato**

A função de inserção inicialmente verifica quanto do prefixo da palavra pode ser reutilizado na atual estrutura do autômato. Depois, o restante das letras da palavra será representado através de novos estados e transições que serão criadas. Caso o estado que liga o prefixo reutilizado e o restante da palavra seja destino de mais de uma transição, então uma operação de clonagem é necessária. Isso é necessário para se manter o autômato determinístico e não permitir o reconhecimento de palavras inválidas. Assim, a função *clone\_state* faz com que esse estado passe a ser destino de apenas uma transição, como mostra a

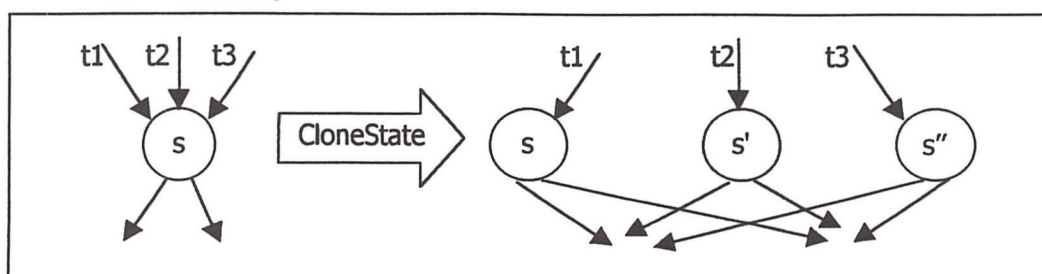


Figura 10. Essa operação consiste basicamente na criação de novos estados e redirecionamento de transições.



**Figura 10 – Operação de clonagem de estado**

O segundo algoritmo, o mais importante de todos, é o responsável pela minimização do autômato (Figura 11). Ele é uma adaptação do algoritmo de Revuz descrito anteriormente. A diferença é que ao invés de uma função de ordenação, é utilizada uma estrutura *hash* para identificação dos estados equivalentes.

---

```
minimize() { // minimiza o autômato

    State *s, *s2, *next // variáveis usadas...
    List *t

    for ( h = 0; h < start.height; h ++ ) { // para todos os níveis (0->maior)
        s = level[ h ]
        while ( s ) { // preenche tabela hash com os
            hash.add( s ) // estados do nível
            s = s.level_next
        }

        s = level[ h ]
        while ( s ) { // para todos os estados do nível
            next = s.level_next
            hash.del( s )
            t = hash.find( s )

            while ( t ) { // procura estado equivalentes
                s2 = t.state
                if ( s.is_equiv( s2 ) ) { // acerta links internos
                    if ( s.level_prev )
                        s.level_prev.level_next = next
                    else
                        level[ h ] = next
                    if ( next )
                        next.level_prev = s.level_prev

                    join_states( s2, s ) // "junta" estados
                    break
                } // end if (equiv)
                else
                    t = t.next // próximo "parecido"
            } // end while (t)

            s = next
        } // end while (s)
    } // end for
} // end proc

join_states(State s1, State s2, [BinTree bt]) { // faz merge de 2 estados equivalentes
```

```

BinTree bt0 = 0
if ( undefined(bt) ) // chamada original da função...
    bt0 = bt = s2.ref // inicializa variáveis

if ( bt.left ) // chame recursivamente para
    join_states( s1, s2, bt.left ) // todos os estados que têm
if ( bt.right ) // transição para o estado que
vai
    join_states( s1, s2, bt.right ) // ser removido (s2)

List list
Transition t = bt.state.trans
while ( t ) { // monta uma lista com todas as
    if ( t.destination == s2 ) // transições que levam à s2
        list.add( t )
    t = t.next
}

while ( t = list.get() ) // muda as transicoes para s1
    s.change_trans( t, s1 )

if ( bt0 )
    delete_state( s2 ) // apaga s2 no final
} // end proc

```

---

**Figura 11 - Processo de minimização do autômato**

O processo de minimização ocorre desde o último nível do autômato, “subindo” até o estado inicial. Para cada nível, são detectados os estados que são equivalentes e esses são “mesclados”. Para distinguir os estados equivalentes, é utilizada uma tabela *hash* que permite a rápida escolha de estados “semelhantes”. A função *hash* é baseada no *id* do estado e nas suas transições, e resulta em um valor de 16 bits (65536 possibilidades).

A função mais importante usada na minimização é a *join\_state*, responsável pela real redução do número de estados e transições do autômato. Após se determinar que dois estados são equivalentes, esses dois estados podem ser representados por apenas um deles, sem “danos” ao autômato. É isso o que faz a função *join\_state(s1, s2)*: todas as transições que levavam à *s2* são redirecionadas para *s1*, e o estado *s2* e suas transições são removidos (conforme visto na Figura 5, página 13)

Por fim, o terceiro algoritmo usado é o que cria a representação física vetorial do autômato (Figura 12).

---

```

vectorize() { // cria vetor

    Int i // variáveis usadas
    State *s
    allocate( n_trans + 1 ) // aloca memória p/ vetor

    remark_id() // remarca ids sequencialmente
    for ( i = start.height; i > 0; i -- ) { // para todos os níveis, do maior ao menor
        s = level[ i ]
        while ( s ) { // adiciona estados do nível ao vetor
            vector_add( s )
            s = s.level_ptr
        }
    }
}

```



```

    }
    } // end for

    save_vector() // faz o que realmente importa
    deallocate() // libera memória do vetor
} // end proc

remark_id( ) {
    State *s // variáveis
    Int , id = 1

    for( i = start.height; i >=0; i-- ) { // para todos os níveis, do maior ao menor
        s = level[ i ]
        while ( s ) { // para todos os estados do nível
            s.id = id // marca id
            id += s.trans.count() // incrementa id adequadamente
            s = s.level_ptr // próximo estado
        } // end while
    } // end for
} // end proc

vector_add( State *state ) { // adiciona transições do estado ao vetor
    Int id = state.id // id do estado
    Transition *t = state.trans.first() // lista de transições do estado

    while ( t ) { // para cada transição
        vector[ id ].symbol = t.symbol // copia dados...
        vector[ id ].destination = t.destination.id
        if ( NOT t.next ) // última transição do estado?
            set_last( vector[ id ] ) // então marca
        id ++ // incrementa identificador
        t = t.next // próxima transição
    } // end while
} // end proc

```

**Figura 12 - Processo de criação da representação final do autômato**

O algoritmo, através da função *remark\_id()*, modifica o valor do campo *id* dos estados, para que esses se tornem sequenciais (uma vez que no processo de minimização são gerados vários “buracos”). Após essa remarcação, a função simplesmente preenche o vetor.

## ALGORITMO DE BUSCA NO VETOR

A aplicação final, como foi dito anteriormente, utilizará apenas a representação vetorial gerada no final da fase de conversão do léxico. O algoritmo para verificar a presença de uma palavra no autômato é bastante simples, e deve ser modificado de acordo com necessidades da aplicação (como obtenção de palavras semelhantes, por exemplo). Segue o pseudo-código:

```

verify ( String word ) { // verifica palavra no automato

    Int p = 1 // p = posição no vetor (1 = raiz)
    Int i = 1 // i = posição na palavra
    Bool ok = TRUE

```

```

while ( ok ) {
    ok = FALSE
    while ( vector[p].symbol <> word[i] ) { // procura transição
        if ( is_last( p ) ) // é transição final do estado?
            break // sai do while
        else
            p ++ // senão pega próx. transição
    }
    if ( vector[p].symbol == word[i] ) { // achou transição?
        p = vector[p].destination // atualiza estado
        i++ // próxima letra da palavra
        if ( i <= lenght(word) ) // não acabaram as letras?
            ok = TRUE // então continua verificando
    }
} // end while

if ( i == lenght(word) + 1 ) // toda a palavra verificada?
    return FOUND
else
    return NOT_FOUND
} // end proc

```

**Figura 13 - Processo de busca de uma palavra na estrutura final**

## 5.3. Resultados

Como foi dito, um protótipo foi desenvolvido em paralelo com a definição dos algoritmos. Esse protótipo mostrou-se eficaz no tratamento em tempo-real de léxicos com centenas de milhares de entradas, rodando em *workstations* relativamente simples<sup>12</sup>. Na Tabela 9 são apresentados os resultados de testes envolvendo léxicos simples (contendo apenas palavras, sem nenhum atributo associado).

O léxico do português utilizado foi obtido a partir do léxico do NILC (seção 2.2), de onde foram extraídas apenas as palavras, e descartadas as que continham hífens (para excluir as ênclises e mesóclises a fim de simplificar o processo). Os outros léxicos foram obtidos em <ftp://ftp.uu.net/doc/dictionaries>.

Para efeito de armazenamento ou transferência por rede de comunicação, por exemplo, o formato compactado do autômato (como sugerido na última coluna da tabela superior) é totalmente aceitável. Entretanto, para uso do autômato ele deve ser descompactado, de forma que o requisito de memória real é o descrito na quarta coluna da mesma tabela.

**Tabela 9 - Dados sobre os léxicos usados nos testes**

| Léxico    | No. Palavras | Tamanho | Tam. Autômato | Aut. Compactado <sup>13</sup> | Redução <sup>14</sup> |
|-----------|--------------|---------|---------------|-------------------------------|-----------------------|
| Sueco     | 14.944       | 123Kb   | 100Kb         | 65Kb                          | 47%                   |
| Inglês    | 45.402       | 399Kb   | 198Kb         | 126Kb                         | 68%                   |
| Italiano  | 61.183       | 560Kb   | 107Kb         | 58Kb                          | 90%                   |
| Português | 431.767      | 4.723Kb | 380Kb         | 218Kb                         | 95%                   |

<sup>12</sup> Foram utilizados nos testes um Pentium 166MHz rodando Linux (kernel 2.2.5-15 + glibc 2.1.1) com 64Mb de memória RAM; e uma Sparc Station Ultra-I rodando Solaris 2.5.1, com 128Mb de RAM.

<sup>13</sup> Essa coluna representa o tamanho do arquivo representante do autômato, após ser submetido ao filtro de compactação *gzip* com parâmetro máximo (-9) de eficiência.

<sup>14</sup> A redução apresentada foi calculada pela razão entre o tamanho do autômato compactado e o tamanho do arquivo original.



| Léxico    | Tot. Estados <sup>15</sup> | Estados <sup>16</sup> | Transições   | Trans./Est. <sup>17</sup> | Tempo <sup>18</sup> | Minim. <sup>19</sup> |
|-----------|----------------------------|-----------------------|--------------|---------------------------|---------------------|----------------------|
| Sueco     | 59.133                     | 13.269 (22%)          | 25.515 (43%) | 1,92                      | 0:04                | 1                    |
| Inglês    | 160.234                    | 23.102 (14%)          | 50.748 (32%) | 2,20                      | 0:25                | 2                    |
| Italiano  | 176.466                    | 10.311 (6%)           | 27.367 (15%) | 2,65                      | 0:32                | 2                    |
| Português | 1.272.983                  | 33.852 (3%)           | 97.277 (8%)  | 2,87                      | 4:17                | 13                   |

A performance de cada um dos quatro léxicos varia devido às características de cada língua, mas mesmo para o primeiro léxico (sueco) o formato ainda é vantajoso. Quanto ao léxico do português a performance final foi excepcionalmente boa (95% de redução no tamanho, 97% no número de estados e 92% no de transições), de forma que o esquema de representação sugerido se torna praticamente mandatário.

Vale lembrar que o acesso à estrutura do autômato é feito de forma direta, sendo necessárias apenas  $n.m$  operações de comparação, onde  $n$  é o número de letras da palavra sendo procurada, e  $m$  é o número médio de transições por estado.

Finalmente, quanto ao uso de memória, o maior léxico (do português) precisou de apenas 14Mb de memória RAM para ser executado, o que mais uma vez demonstra o potencial do sistema em tratar léxicos de grande porte mesmo em *workstations* mais modestas (veja seção de considerações abaixo).

## 5.4. Considerações

Algumas considerações importantes devem ser feitas quanto aos algoritmos e estruturas expostos e quanto à forma de implementação escolhida (linguagem, sistema operacional, máquina, etc).

Os elementos do sistema foram definidos de forma a modelar uma estrutura bem próxima à estrutura teórica de um autômato. Isso facilita tanto o entendimento quanto à implementação de novas funções no sistema, mas compromete um pouco a performance no que diz respeito ao espaço de armazenamento (memória RAM).

Outra questão importante é sobre o gasto teórico e o real de memória. As bibliotecas de alocação dinâmica de memória costumam não ser totalmente configuráveis pelo usuário e, em geral, não apresentam algoritmos de reutilização de memória muito eficientes, o que pode resultar em muita fragmentação na memória alocada. Isso varia para cada sistema operacional e para cada biblioteca de alocação de memória. Uma abordagem interessante é implementar um módulo responsável pelo gerenciamento de memória, que trabalharia em uma enorme área de memória pré-alocada no início do programa. Outra abordagem é reiniciar o programa periodicamente, por exemplo, a cada  $n$  minimizações.

<sup>15</sup> Total de estados representa o número total de estados que esse autômato teria se não sofresse nenhuma minimização.

<sup>16</sup> O percentual entre parênteses representa a relação entre o número final de estados e o número total (da segunda coluna). O mesmo se aplica à quarta coluna, com relação ao número de transições.

<sup>17</sup> Número médio de transições em cada estado do autômato.

<sup>18</sup> Tempo total expresso em minutos:segundos, gasto no processo de criação do autômato, usando um Pentium166/Linux com 55Mb de memória livre e *threshold* de 100.000 estados.

<sup>19</sup> A coluna indica quantas operações de minimização foram necessárias.



Enfim, para que o sistema possa ser capaz de tratar com léxicos de vários milhões de entradas de uma forma aceitável, as limitações físicas não devem ser ignoradas, mas sim contornadas.

## 6. Representação de Atributos Lexicais

Neste capítulo será abordada a questão da associação de atributos lexicais, tais como gênero, número e forma canônica, às entradas do léxico. A abordagem utilizada aqui será a de embutir os atributos à entrada, de forma a centralizar o procedimento de compactação do léxico em um sistema de representação e minimização de autômatos finitos, como o descrito no capítulo anterior.

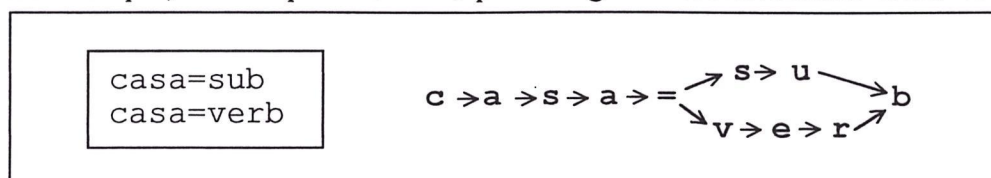
O trabalho aqui descrito usou como base o léxico do NILC (descrito na seção 2.2), cujo formato dos atributos foi alterado para representação mais eficiente no autômato. Esse formato será descrito na seção 6.2, e é baseado na estratégia usada em [KOW98] (seção 4.1.4).

### 6.1. Considerações Iniciais

A representação dos atributos lexicais deve permitir identificar as entradas de forma eficiente através de seu prefixo e embutir os possíveis atributos como sufixos dessa entrada, como no exemplificado na Figura 14.

**Figura 14 - Reutilização de sufixos**

Nesse exemplo, dada a palavra *casa*, pode-se gerar recursivamente todos os possíveis



sufixos, obtendo-se *verbo* e *sub*.

Por outro lado, a ordem e o formato dos atributos nas entradas devem levar em conta a capacidade do autômato de reutilização dos sufixos, de trás para frente, durante a fase de minimização. Nesse aspecto, o atributo representante da forma canônica, por exemplo, merece um tratamento especial, já que poderia comprometer a eficiência do autômato. Um formato possível é representar apenas a diferença de sufixo da palavra, como na Figura 15.

|            |   |                  |
|------------|---|------------------|
| menino:0   | → | menino / menino  |
| menina:1a  | → | menina / menino  |
| meninos:1  | → | meninos / menino |
| meninas:2o | → | meninas / menino |
| somos:4er  | → | somos / ser      |
| é:1ser     | → | é / ser          |

**Figura 15 - Representação por diferença de sufixo**

Nesse formato, o dígito indica quantos caracteres a partir do final da palavra devem ser removidos, e quais devem ser inseridos.

Finalmente, a parte mais complexa é a de se definir quais serão os valores dos atributos, e em que ordem eles vão aparecer na entrada do léxico. A resposta a essa questão depende de cada léxico, caso se queira um sistema realmente eficiente.

## 6.2. Proposta de Formato para o Léxico do NILC

Baseado no esquema de representação apresentado em [KOW98], um formato alternativo de representação do léxico do NILC será descrito nessa seção, levando-se em consideração três pontos importantes:

- Afixos que podem ser facilmente reutilizados;
- Fácil interpretação das entradas do léxico;
- Formato facilmente extensível.

Foi desenvolvido um filtro para converter o léxico do formato original para o formato otimizado que está sendo proposto. O programa gera, para cada entrada do léxico do NILC, uma ou mais entradas simplificadas (já que cada entrada do léxico original pode embutir várias entradas distintas). Também é gerada uma entrada reversa para cada entrada do léxico otimizado, para permitir a busca no autômato pela forma canônica da palavra. A Tabela 10 apresenta alguns exemplos do formato proposto.

**Tabela 10 – Entradas do léxico do NILC e respectivas conversões para o léxico otimizado**

| Léxico original                                                                                                                 | Léxico Otimizado                                                                                                                                                                                                                        |
|---------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| a = <ART.F.SI.DE.?.?.[o]0.#PREP.C.[][a]0.#<br>PRON.F.SI.[DEM.OBL-AT.]3S.?.?.C.[][o]0.#<br>ABREV.M.SI.[a]0.#S.M.SI.N.[]??.[?]0.> | a=1o=ART,F,SI,DE<br>o 0=ART,M,SI,DE<br>a=0=PREP,0<br>a 0=PREP,0<br>a=1o=PRON,F,SI,DEM,1o<br>o 0=PRON,M,SI,DEM,0<br>a=1o=PRON,F,SI,OBL-AT,1o<br>o 0=PRON,M,SI,OBL-AT,0<br>a=0=ABREV,0<br>a 0=ABREV,0<br>a=0=S,M,SI,N,0<br>a 0=S,M,SI,N,0 |
| ab-rogava=<V.[][PRET-IMPERF.EU.<br>PRET-IMPERF.ELE.]N.[][ab-rogar]0.>                                                           | ab-rogava=2r=V,PRET-IMPERF,EU,N<br>ab-rogar 1va=V,PRET-IMPERF,EU,N<br>ab-rogava=2r=V,PRET-IMPERF,ELE,N<br>ab-rogar 1va=V,PRET-IMPERF,ELE,N                                                                                              |
| enquanto=<CONJ.[COORD.][SUBORD.<br>[PROPOR.TEMP.][enquanto]0.>                                                                  | enquanto=0=CONJ,SUBORD-PROPOR<br>enquanto 0=CONJ,SUBORD-PROPOR<br>enquanto=0=CONJ,SUBORD-TEMP<br>enquanto 0=CONJ,SUBORD-TEMP                                                                                                            |

As entradas que contém ‘|’ como primeiro separador são as entradas reversas. Para simplificar o sistema, algumas informações da entrada original estão sendo descartadas (regras de derivação e informações de regência, por exemplo).

A utilização dos *tokens* completos (CONJ, PRET-IMPERF, SUBORD-TEMP, etc.), ao invés de identificadores mais reduzidos, não chega a comprometer o resultado final, pois esses *tokens* serão altamente reutilizados no processo de minimização realizado no autômato (veja seção de resultados, logo adiante).



Testes realizados com o formato que está sendo proposto indicaram uma redução maior que 90% no número de estados/transições do autômato, ou seja, muito próximas às obtidas com o léxico simples na seção 5.3; e uma redução de tamanho maior que 90% (em relação ao tamanho da estrutura final compactada e o arquivo original), também próxima à obtida com os léxicos simples. Isso significa que o *overhead* gerado pelos atributos no formato otimizado sugerido foram muito bem compensados pelo mecanismo implícito de reutilização de afixos do autômato.

### 6.2.1. BNF do Formato Proposto de Léxico

```
<entrada> ::= <entrada_normal> | <entrada_reversa>
<entrada_normal> ::= <palavra> "=" <dif_sufixo> "=" <classe>
<entrada_reversa> ::= <palavra> "|" <dif_sufixo> "=" <classe>
<dif_sufixo> ::= <digito>{<digito>}{<letra>}
<palavra> ::= <letra> {<letra> }
<digito> ::= "0" | .. | "9"
<letra> ::= <letras e caracteres válidos para as palavras>
<classe> ::= <substantivo> | <artigo> | <preposição> | <numeral> | <pronome> | <verbo> |
               <adjetivo> | <conjunção> | <advérbio> | <nome> | <prefixo> | <sigla> |
<abreviatura>
```

```
<substantivo> ::= "SUB," <genero> "," <numero> "," <grau>
<artigo> ::= "ART," <genero> "," <numero> "," <tipo_artigo>
<preposição> ::= "PREP"
<numeral> ::= "NUM," <genero> "," <numero> "," <tipo_numeral>
<pronome> ::= "PRON," <genero> "," <numero> "," <tipo_pronome>
<verbo> ::= "V," <tempo> "," <peessoa> "," <coloc_pron>
<adjetivo> ::= "ADJ," <genero> "," <numero> "," <grau>
<conjunção> ::= "CONJ," <tipo_conjuncao>
<advérbio> ::= "ADV," <tipo_adv> "," <grau>
<nome> ::= "NOM," <genero> "," <numero>
<prefixo> ::= "PREF"
<sigla> ::= "SIGL"
<abreviatura> ::= "ABREV"
```

```
<genero> ::= "M" | "F" | "2G" | "INV"
<numero> ::= "SI" | "PL" | "2N" | "INV"
<grau> ::= "N" | "A" | "D" | "SU"
<tipo_artigo> ::= "DE" | "I"
<tipo_numeral> ::= "ORD" | "MUL" | "FRA" | "FRA-ORD" | "CAR"
<tipo_pronome> ::= "TRAT" | "RET" | "OBL-AT" | "OBL-TO" | "POSS" | "DEM" | "INDE" |
                  "INTE" | "REL" | "REFL"
<coloc_pron> ::= "N" | "ENC" | "MESOC"
<tempo> ::= PRES" | "PRET-IMPERF" | "PRET-PERF" | "PRET-M-Q-P" | "FUT-PRES" |
            FUT-PRET" | "PRES-SUBJ" | "PRET-IMPERF-SUBJ" | "FUT-SUBJ" |
            IMPER-AFIRM" | "INF-PESS"
<peessoa> ::= "EU" | "TU" | "ELE" | "NOS" | "VOS" | "ELES"
<tipo_conjuncao> ::= "COORD-ADIT" | "COORD-ADVE" | "COORD-ALTER" | "COORD-CONCL" |
                    "COORD-EXPL" | "SUBORD-INTE" | "SUBORD-CAUS" | "SUBORD-
COMP" |
                    "SUBORD-CONC" | "SUBORD-COND" | "SUBORD-CONS" | "SUBORD-
FIN" |
                    "SUBORD-TEMP" | "SUBORD-PROPOR" | "SUBORD-CONFOR"
<tipo_adv> ::= "CIR-LUG" | "CIR-TEMP" | "CIR-MOD" | "NEG" | "DUV" | "INT" | "AFIR" |
              "INT-LUG" | "INT-TEMP" | "INT-MOD" | "INT-CAUS"
```

### 6.3. Resultados

Para os testes executados, foram utilizados dois subconjuntos do léxico do NILC (seção, 2.2): o primeiro, contendo cerca de 1% das entradas do léxico, e que será referenciado por LEX1 nessa seção; e o segundo, contendo cerca de 10% das entradas do léxico, que será chamado de LEX10. Os dois léxicos foram obtidos selecionando-se linhas do arquivo original de forma randômica, de forma a se tentar gerar uma amostra representativa do léxico original. Cada léxico foi convertido para o formato otimizado apresentado, gerando quatro arquivos:

- LEX\*.w -- contém apenas as palavras (distintas) do léxico
- LEX\*.full -- contém todas as entradas (diretas e reversas)
- LEX\*.dir -- contém apenas as entradas diretas (pela palavra)
- LEX\*.rev -- contém apenas as entradas reversas (pela canônica)

Dessa forma, foram gerados 8 léxicos, conforme descrito na Tabela 11, que também inclui o léxico original (formato do NILC) como referência.

**Tabela 11 - Descrição dos léxicos utilizados nos testes do formato**

| Léxico         | Entradas | Tamanho    |
|----------------|----------|------------|
| LEX1 Original  | 17.110   | 1.052.012  |
| LEX1.w         | 17.109   | 236.882    |
| LEX1.full      | 34.520   | 1.252.546  |
| LEX1.dir       | 17.260   | 626.273    |
| LEX1.rev       | 17.260   | 626.273    |
| LEX10 Original | 156.580  | 9.842.373  |
| LEX10.w        | 156.532  | 2.166.633  |
| LEX10.full     | 316.602  | 11.479.906 |
| LEX10.dir      | 158.301  | 5.739.953  |
| LEX10.ver      | 158.301  | 5.739.953  |

Inicialmente foram convertidos os léxicos apenas das palavras, para referência e cálculos de performance dos outros testes. Os resultados estão na Tabela 12.

**Tabela 12 - Resultados da conversão das palavras de LEX1 e LEX10**

| Léxico  | Tot. Estados | Estados      | Transições    | Tempo | Min. | Aut. Compact. |
|---------|--------------|--------------|---------------|-------|------|---------------|
| LEX1.w  | 134.213      | 23.784 (82%) | 39.961 (70%)  | 0:07  | 3    | 101Kb         |
| LEX10.w | 833.494      | 75.342 (91%) | 180.599 (78%) | 1:59  | 17   | 459Kb         |

Na Tabela 13 são apresentados os resultados obtidos na conversão dos léxicos que foram convertidos para o formato. O maior processamento (LEX10.full), que durou pouco mais que 10 minutos em um Pentium166/Linux, consumiu 32Mb de memória RAM.



**Tabela 13 - Resultados de LEX1 e LEX10**

| Léxico     | Tot. Estados | Estados      | Transições   | Tempo | Min. | Autômato compactado |
|------------|--------------|--------------|--------------|-------|------|---------------------|
| LEX1.full  | 985.437      | 53.783 (5%)  | 82.218 (9%)  | 1:09  | 10   | 266Kb               |
| LEX1.dir   | 522.098      | 33.206 (6%)  | 49.891 (10%) | 0:39  | 6    | 133Kb               |
| LEX1.rev   | 491.898      | 42.918 (9%)  | 59.493 (12%) | 0:37  | 5    | 156Kb               |
| LEX10.full | 8.271.497    | 176.557 (2%) | 391.416 (5%) | 10:36 | 83   | 993Kb               |
| LEX10.dir  | 4.388.763    | 105.967 (2%) | 220.224 (6%) | 4:48  | 44   | 556Kb               |
| LEX10.rev  | 3.955.594    | 114.297 (3%) | 224.164 (5%) | 4:25  | 40   | 552Kb               |

Comparando os resultados de LEX1\*.full e LEX\*.dir com LEX\*.w, podemos observar melhor o impacto dos atributos nas entradas no léxico (Tabela 14).

**Tabela 14 - Compação de LEX\*.dir e LEX\*.full com LEX\*.w**

|                      | LEX1.dir | LEX10.dir | LEX1.full | LEX10.full |
|----------------------|----------|-----------|-----------|------------|
| Número de estados    | + 40%    | + 41%     | + 126%    | + 134%     |
| Número de transições | + 25%    | + 22%     | + 106%    | + 117%     |
| Tamanho do autômato  | + 32%    | + 21%     | + 163%    | + 116%     |

Analisando LEX\*.dir (entradas diretas, pela palavra), vê-se que as informações sobre os atributos causaram um aumento de pouco mais de 20% no tamanho do autômato final, enquanto que o autômato que inclui também as entradas reversas (LEX\*.full) teve um aumento de mais de 100% em seu tamanho. Assim, dependendo da aplicação, um melhor formato para as entradas reversas pode ser estudado, caso isso represente um problema.

## 6.4. Comparações e Análise dos Resultados

Nesse capítulo será feita uma comparação entre os resultados obtidos com esse trabalho e com resultados obtidos por outros pesquisadores. Para referência, serão utilizados os trabalhos de Mehryar Mohri (AT&T) [MOH95] e de Kowaltowski, Lucchesi e Stolfi (Unicamp) [KOW98].

A Figura 16 mostra os resultados obtidos por Mohri, usando o pacote FSM (introduzido na seção 3.4.4). Trabalhando com léxicos completos (palavras + atributos) de várias línguas, e fazendo a conversão do formato original para um formato mais apropriado, o sistema foi capaz de tratar léxicos de grande porte (+800.000 entradas, +27Mb), mas com tempos às vezes muito altos (+18 horas, em um CrayII com 128Mb de memória).



|              |                   |                      |                     |                   |                   |                   |
|--------------|-------------------|----------------------|---------------------|-------------------|-------------------|-------------------|
| DICTIONARIES | Name              | FDELAF<br>French V.7 | FDELACF<br>Compound | GDELAF<br>General | EDELAF<br>English | IDELAF<br>Italian |
|              | Nb of lines       | 672,000              | 156,000             | 828,000           | 145,000           | 612,000           |
|              | Initial size      | 21.2 Mb              | 5.6 Mb              | 27.9 Mb           | 3.6 Mb            | 20 Mb             |
| AUTOMATA     | Nb of states      | 84,600               | 322,800             | 389,650           | 48,770            | 78,320            |
|              | Nb of transitions | 181,910              | 466,570             | 633,520           | 101,970           | 177,350           |
|              | Size              | 2.4 Mb               | 7.6 Mb              | 10 Mb             | 1.2 Mb            | 2.3 Mb            |
|              | Compacted size    | 818 Kb               | 1.88 Mb             | 2.70 Mb           | 447 Kb            | 806 Kb            |
|              | Nb of codes       | 13,200               | 1,750               | 14,950            | 1,590             | 11,190            |
|              | Size of codes     | 358 Kb               | 445 Kb              | 403 Kb            | 25 Kb             | 257 Kb            |
|              | Total final size  | 1.2 Mb               | 1.9 Mb              | 3.1 Mb            | 470 Kb            | 1.1 Mb            |
| TIME SPENT   | Constr. (CRAY)    | -                    | 12h40               | 18h53             | -                 | -                 |
|              | Constr. (HP)      | 12'30                | -                   | -                 | 4'55"             | 12'30             |
|              | Constr. (NEXT)    | 1h18'                | -                   | -                 | 17'               | 1h20'             |
|              | Look-up (HP)      | 90 w/ms              | 90 w/ms             | 90 w/ms           | 90 w/ms           | 90 w/ms           |

**Figura 16 - Resultados obtidos por Mohri com o FSM Library [MOH95]**

Comparativamente, o sistema proposto nesse trabalho foi capaz de tratar léxicos de mais de 8.000.000 de entradas (11Mb) em pouco mais de 10 minutos e necessitando de apenas 32Mb de memória, gerando uma representação final compactada de menos de 1Mb, como visto na seção 6.3. Entretanto, como os léxicos usados por Mohri apresentavam muitos mais atributos associados, como pode ser notado pela relação entre o tamanho e o número de linhas dos arquivos, uma comparação detalhada dos resultados não pode ser estabelecida.

Um outro trabalho muito importante na área é o de Kowaltowski, Lucchesi e Stolfi (referenciados nesse trabalho de forma resumida por KLS, cujo trabalho está resumido na seção 4.1), cujos resultados podem ser comparados diretamente com os obtidos nesse trabalho devido a utilizarem o mesmo léxico como base para testes (no caso, o léxico do NILC).

Em [KOW98], eles converteram um léxico de 1.490.002 palavras distintas (com seus respectivos atributos), das quais foram geradas 4.500.688 entradas para criar o dicionário. Como resultado, obtiveram um autômato com 190.795 estados e 346.534 transições, representados em um arquivo único arquivo de apenas 1089Kb.

O trabalho de KLS, decorrente de muitos anos de pesquisa na área, é realmente excepcional em seu segmento: trabalha com uma representação final altamente otimizada; usa uma heurística especial em uma fase de pós-minimização; e cria um formato de representação eficiente para cada léxico.

No trabalho aqui apresentado não foi utilizado o léxico completo do NILC nos testes, devido à limitações físicas de memória. Entretanto, trabalhando com 10% do total, os resultados ficaram equivalentes aos resultados obtidos por KLS no tratamento de 100% do léxico! Uma grande diferença se deve ao formato de representação dos atributos (principalmente das entradas reversas), que sem dúvida ainda permitem otimizações.



## 7. Conclusões

---

Esse capítulo apresenta as conclusões e contribuições desse trabalho, assim como a relação dos possíveis trabalhos futuros.

Autômatos finitos se apresentam como excelentes dispositivos para representação de grandes léxicos de língua natural. Aliado ao uso de bons algoritmos e estruturas de dados, pode-se implementar um sistema extremamente eficiente, capaz de converter léxicos de milhões de palavras e trabalhar com autômatos de dezenas de milhões de estados. Dependendo do tamanho do problema, um micro doméstico (PC, 64Mb memória) pode ser suficiente. Já para problemas maiores (autômatos com mais de 15 milhões de estados, por exemplo), uma *workstation* com mais memória se faz necessária.

Os atributos lexicais podem ser codificados eficientemente em um léxico de forma a permitir a melhor taxa possível de otimização pelo sistema de codificação, desde que se leve em consideração as características básicas de re-aproveitamento de estados no autômato. Assim, o tamanho final do autômato compactado incluindo os atributos pode ficar apenas 20% maior (ou até menos) que o autômato equivalente sem os atributos.

Como contribuição principal foi proposto um sistema de codificação eficiente de léxicos em autômatos, incluindo todas as estruturas de dados e algoritmos, além da implementação de um protótipo 100% funcional do sistema<sup>20</sup>. Também foi sugerido um formato otimizado de representação dos atributos léxicos do léxico do NILC, incluindo a descrição BNF completa desse formato.

O sistema proposto nesse trabalho irá possibilitar, juntamente com a base lexical que está sendo organizada no NILC, que o grupo disponha de um ambiente de geração de dicionários específicos para determinadas aplicações, além de funções e bibliotecas de acesso aos mesmos. Isso irá representar uma importante ferramenta para o grupo que trabalha com pesquisa e desenvolvimento de aplicações em PLN.

Um trabalho importante a ser desenvolvido é a melhoria no formato proposto de representação dos atributos do léxico do NILC, de forma a se obter uma representação ainda mais compacta, e também mais detalhada do léxico original (já que alguns dos atributos originais, tais como regra de derivação e regência, por exemplo, foram descartados).

Outro trabalho importante seria um módulo de gerenciamento de memória próprio, para permitir que o sistema possa tratar problemas ainda maiores, mesmo em *workstations* de médio poder computacional. Esse módulo seria muito importante também para quaisquer outros sistemas que necessitem de grandes quantidade de memória RAM para trabalhar.

Um futuro trabalho muito interessante poderia ser a remodelagem do sistema, de forma a se utilizar dois processos concorrentes (ou *pseudo-concorrentes*) comunicando-se por memória compartilhada: um processo responsável pela inserção de palavras e outro pela minimização do autômato. Uma abordagem poderia ser a utilização de *threads*, ou até mesmo o uso de bibliotecas de comunicação como PVM e MPI.

Por fim, mas não menos importante, poderia ser criada uma interface gráfica de acesso ao sistema proposto nesse trabalho, inclusive com possibilidade de visualização gráfica do autômato gerado por partes ou por completo (nesse último caso apenas para pequenos autômatos).

---

<sup>20</sup> Quanto à disponibilidade do protótipo, consultar o *site* do NILC (<http://nilc.icmc.sc.usp.br>), ou diretamente o autor por correio eletrônico em [mcouto@null.net](mailto:mcouto@null.net).



## 8. Referências Bibliográficas

---

- [AHO74] Aho, A.V.; J.E. Hopcroft; Ullman, J.D. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.
- [AHO86] Aho, A.V.; Sethi, R.; Ullman, J.D. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1986.
- [APP88] Appel, A.W.; Jacobson, G. J. *The World's Fastest Scrabble Program*. Communications of ACM, 31(5), 572-578, 585, 1988.
- [BRZ62] Brzozowski, J.A. *Canonical Regular Expressions and Minimal State Graphs for Definite Events*. Mathematical Theory of Automata, MRI Symposia Series, v. 12 (Polytechnic Press, Polytechnic Institute of Brooklyn, NY, 1962), p. 529-561.
- [CPM97] *Combinatorial Pattern Matching '97*. Lecture Notes in Computer Science, vol. 1264, 1997.
- [GRI73] Gries, D. *Describing an Algorithm by Hopcroft*. Acta Inf. 2, 97-109, 1973.
- [GRO89] Gross, M.; Perrin, D. *Electronic Dictionaries and Automata in Computations Linguistics*. Lecture Notes in Computer Science, vol. 377, 1989.
- [HOP79] Hopcroft, J.E.; Ullman, J.D. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, 1979.
- [HOP71] Hopcroft, J.E. *An  $n \log n$  Algorithm for Minimizing the States in a Finite Automaton*. The Theory of Machines and Computations (Academic Press, NY, 1971), p. 189-189.
- [KOW93] Kowaltowski, T.; Lucchesi, C.L. *Applications of Finite Automata Representing Large Vocabularies*. Software-Practice and Experience, 23(1), 15-20, 1993.
- [KOW95a] Kowaltowski, T.; Lucchesi, C.L.; Stolfi, J. *Minimization of Binary Automata*. Journal of the Brazilian Computing Society, 3(1), 36-42, 1995.
- [KOW95b] Kowaltowski, T.; Lucchesi, C.L.; Stolfi, J. *Application of Finite Automata in Debugging Natural Language Vocabularies*. Journal of the Brazilian Computing Society, 3(1), 5-11, 1995.
- [KOW98] Kowaltowski, T.; Lucchesi, C.L.; Stolfi, J. *Finite Automata and Efficient Lexicon Implementation*. Relatório Técnico IC-92-2, DCC/UNICAMP, 1998.
- [LAN96] Langsam, Y.; Augenstein, M.J.; Tenenbaum, A.M. *Data Structures Using C and C++*. Prentice Hall, 2a. ed., 1996.
- [MOH95] Mohri, M. *On Some Applications of Finite-State Automata Theory to Natural Language Processing*. Natural Language Engineering 1, Cambridge University Press, 1995.

- [NOO97] Van Noord, G. *FSA Utilities: A Toolbox to Manipulate Finite-State Automata*, Lecture Notes in Computer Science, vol. 1260, 1997, p.87-108.
- [NUN96] Nunes, M.G.V. et al. *A Construção de um Léxico para a Língua Portuguesa: Lições Aprendidas e Perspectivas*. II Workshop de Processamento Computacional do Português Escrito e Falado, Curitiba, PR, 61-70, Outubro de 1996.
- [PAC96] Pacheco, H.C.F. *Uma Ferramenta de Auxílio à Redação*. Dissertação de Mestrado, DCC-UFMG, 24 de outubro de 1996.
- [RAY93] Raymond, D.R.; Wood, D. *Grail papers*. Department of Computer Science, University of Waterloo, Canada. Internet: <ftp://cs-archive.uwaterloo.edu>
- [REV92] Revuz, D. *Minimization of Acyclic Deterministic Automata in Linear Time*. Theoretical Computer Science, 92(1), 181-189, 1992.
- [WAT93a] Watson, B.W. *A Taxonomy of Finite Automata Minimization Algorithms*. Internet: <http://www.ribbitsoft.com/research/watson>
- [WAT93b] Watson, B.W. *A Taxonomy of Finite Automata Construction Algorithms*. Internet: <http://www.ribbitsoft.com/research/watson>
- [WAT96] Watson, B.W. *FIRE Lite: FAs and REs in C++*. Lecture Notes in Computer Science, vol. 1260, 1997, p.167-188.
- [WIA96] *Workshop on Implementing Automata '96*. Lecture Notes in Computer Science, vol. 1260, 1997.
- [WOO87] Wood, D. *Theory of Computation*. Harper & Row, NY, 1987.



# Apêndice A - B.N.F. do Léxico do NILC

---

**1S.** = primeira pessoa do singular  
**2S.** = segunda pessoa do singular  
**3S.** = terceira pessoa do singular  
**1P.** = primeira pessoa do plural  
**2P.** = segunda pessoa do plural  
**3P.** = terceira pessoa do plural  
**3SP.** = 3ª pessoa do sing. e 3ª pessoa do plural  
**A.** = aumentativo  
**ABREV.** = abreviatura  
**ADIT.** = aditiva  
**ADJ.** = adjetivo  
**ADVE.** = adversativa  
**AFIR.** = afirmação  
**ALTER.** = alternativa  
**ART.** = artigo  
**AUX.** = auxiliar  
**BI.** = bitransitivo  
**C.** = contração  
**CAR.** = cardinal  
**CAUS.** = causal  
**CIR-LUG.** = circunstância lugar  
**CIR-TEMP.** = circunstância tempo  
**CIR-MOD.** = circunstância modo  
**COMP.** = comparativa  
**CONC.** = concessiva  
**CONCL.** = conclusiva  
**COND.** = condicional  
**CONFOR.** = conformativa  
**CONJ.** = conjunção  
**CONS.** = consecutiva  
**COORD.** = coordenativa  
**D.** = diminutivo  
**DE.** = definido  
**DEM.** = demonstrativo  
**DUV.** = dúvida  
**ENC.** = ênclise  
**EXPL.** = explicativa  
**F.** = feminino  
**FIN.** = final  
**FRA.** = fracionário  
**FRA-ORD.** = fracionário e ordinal  
**FUT-PRES.** = futuro do presente  
**FUT-PRET.** = futuro do pretérito  
**FUT-SUBJ.** = futuro do subjuntivo  
**GERUN.** = gerúndio  
**I.** = artigo indefinido  
**IMP-AFIRM.** = imperativo afirmativo  
**INDE.** = pronome indefinido  
**INF-PESS.** = infinitivo pessoal

**INT.** = intransitivo  
**INTE.** = intensidade  
**INTEG.** = integrante  
**INTER.** = interrogativo  
**INT-CAUS.** = interrogativo de causa  
**INT-LUG.** = interrogativo de lugar  
**INT-MOD.** = interrogativo de modo  
**INT-TEMP.** = interrogativo de tempo  
**INV.** = invariável  
**LIG.** = ligação  
**M.** = masculino  
**MESOC.** = mesóclise  
**MUL.** = multiplicativo  
**N.** = nulo  
**NEG.** = negação  
**NOM.** = nome próprio  
**NUM.** = numeral  
**OBL-AT.** = pessoal oblíquo átono  
**OBL-TO.** = pessoal oblíquo tônico  
**ORD.** = ordinal  
**PARTIC.** = participio  
**PL.** = plural  
**POSS.** = possessivo  
**PREF.** = prefixo  
**PREP.** = preposição  
**PRES.** = presente  
**PRES-SUBJ.** = presente do subjuntivo  
**PRET-IMPERF.** = pretérito imperfeito  
**PRET-IMPERF-SUBJ.** = pret. imperfeito do subj.  
**PRET-M-Q-P.** = pretérito mais que perfeito  
**PRET-PERF.** = pretérito perfeito  
**PRON.** = pronome  
**PRONOM.** = pronominal  
**PROPOR.** = proporcional  
**RET.** = pessoal reto  
**REL.** = relativo  
**REFL.** = reflexivo  
**S.** = substantivo  
**SI.** = singular  
**SIGL.** = sigla  
**SU.** = superlativo  
**SUBORD.** = subordinativa  
**TD.** = transitivo direto  
**TEMP.** = temporal  
**TI.** = transitivo indireto  
**TRAT.** = tratamento  
**V.** = verbo  
**2G.** = dois gênero  
**2N.** = dois número

::= Significa "é definido como".  
 | Significa "ou".  
 { } Significa que o conteúdo pode ser repetido nenhuma ou várias vezes.  
 < > Significa que o conteúdo é uma construção sintática BNF.

<verbo> ::= <palavra> =< <classificação> { # <classificação> } >  
 <abreviatura> ::= **ABREV.**  
 <adjetivo> ::= **ADJ.** <gênero-2> <número-2> <grau-2> <regência> <regra-de-derivação>  
 <advérbio> ::= **ADV.** <tipo-do-advérbio> <grau-2>  
 <artigo> ::= **ART.** <gênero-1> <número-1> <tipo-do-artigo> <regra-de-derivação>  
 <canônica> ::= [ <palavra> ]  
 <character> ::= a..z | á | é | í | ó | ú | à | â | ê | ô | ã | õ | ç | ü | - | ' | ¨ | °  
 <classe> ::= <substantivo> | <artigo> | <preposição> | <numeral> | <pronome> | <verbo> |  
 <adjetivo> | <conjunção> | <advérbio> | <nome-próprio> | <prefixo> | <sigla> | <abreviatura>  
 <classificação> ::= <classe> <canônica> <flag-composição>  
 <colocação-pronominal> ::= **N.** | **ENC.** | **MESOC.**  
 <conjugação> ::= [ <tempo> <peessoa-1> | <forma-nominal>  
 { <tempo> <peessoa-1> | <forma-nominal> } ]  
 <conjunção> ::= **CONJ.** <tipo-da-conjunção>  
 <conjunções-coordenadas> ::= [ <tipos-de-conj-coord> { <tipos-de-conj-coord> } ]  
 <conjunções-subordinadas> ::= [ <tipos-de-conj-subord> { <tipos-de-conj-subord> } ]  
 <contração> ::= **C.** <lista-de-palavras>  
 <dígito> ::= **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9**  
 <flag-composição> ::= <dígito>  
 <forma-nominal> ::= **GERUN.** | **PARTIC.** <gênero-2> <número-2>  
 <gênero-1> ::= **M.** | **F.**  
 <gênero-2> ::= **M.** | **F.** | **2G.**  
 <gênero-3> ::= **M.** | **F.** | **2G.** | **INV.**  
 <grau-1> ::= **N.** | **A.** | **D.**  
 <grau-2> ::= **N.** | **A.** | **D.** | **SU.**  
 <lista-de-palavras> ::= [ <palavra> . { <palavra> . } ]  
 <nome-próprio> ::= **NOM.** <gênero-2> <número-2>  
 <numeral> ::= **NUM.** <gênero-3> <número-3> <tipo-do-numeral> <regra-de-derivação>  
 <número-1> ::= **SI.** | **PL.**  
 <número-2> ::= **SI.** | **PL.** | **2N.**  
 <número-3> ::= **SI.** | **PL.** | **2N.** | **INV.**  
 <palavra> ::= <character> { <character> }  
 <peessoa-1> ::= **EU.** | **TU.** | **ELE.** | **NOS.** | **VOS.** | **ELES.**  
 <peessoa-2> ::= **1S.** | **2S.** | **3S.** | **1P.** | **2P.** | **3P.** | **3SP.** | **N.**  
 <predicação> ::= [ <tipo-da-predicação> { <tipo-da-predicação> } ]  
 <prefixo> ::= **PREF.**  
 <preposição> ::= **PREP.** <contração>  
 <pronome> ::= **PRON.** <gênero-3> <número-3> <tipo-do-pronome> <peessoa-2>  
 <regra-de-derivação> <contração>  
 <regência> ::= <lista-de-palavras>  
 <regra-de-derivação> ::= <dígito> | ?  
 <sigla> ::= **SIGL.**  
 <substantivo> ::= **S.** <gênero-2> <número-3> <grau-1> <regência> <regra-de-derivação>  
 <tempo> ::= **PRES.** | **PRET-IMPERF.** | **PRET-PERF.** | **PRET-M-Q-P.** | **FUT-PRES.** |  
**FUT-PRET.** | **PRES-SUBJ.** | **PRET-IMPERF-SUBJ.** | **FUT-SUBJ.** |  
**IMPER-AFIRM.** | **INF-PESS.**  
 <tipo-da-predicação> ::= **INT.** | **TD.** | **TI.** | **BI.** | **LIG.** | **AUX.** | **PRONOM.**  
 <tipo-do-advérbio> ::= **CIR-LUG.** | **CIR-TEMP.** | **CIR-MOD.** | **NEG.** | **DUV.** | **INT.** |  
**AFIR.** | **INT-LUG.** | **INT-TEMP.** | **INT-MOD.** | **INT-CAUS.**  
 <tipo-do-artigo> ::= **DE.** | **I.**



<tipo-da-conjunção> ::= [ **COORD.** <conjunções-coordenadas> |  
                           **SUBORD.** <conjunções-subordinadas>  
                           { **COORD.** <conjunções-coordenadas> |  
                           **SUBORD.** <conjunções-subordinadas> } ]  
 <tipo-do-numeral> ::= **ORD.** | **MUL.** | **FRA.** | **FRA-ORD.** | **CAR.**  
 <tipo-do-pronome> ::= **TRAT.** | **RET.** | **OBL-AT.** | **OBL-TO.** | **POSS.** | **DEM.** | **INDE.** |  
                           **INTE.** | **REL.** | **REFL.**  
 <tipos-de-conj-coord> ::= **ADIT.** | **ADVE.** | **ALTER.** | **CONCL.** | **EXPL.**  
 <tipos-de-conj-subord> ::= **INTE.** | **CAUS.** | **COMP.** | **CONC.** | **COND.** | **CONS.** | **FIN.**  
                                   **TEMP.** | **PROPOR.** | **CONFOR.**  
 <verbo> ::= **V.** <predicação> <conjugação> <colocação-pronominal> <regência>