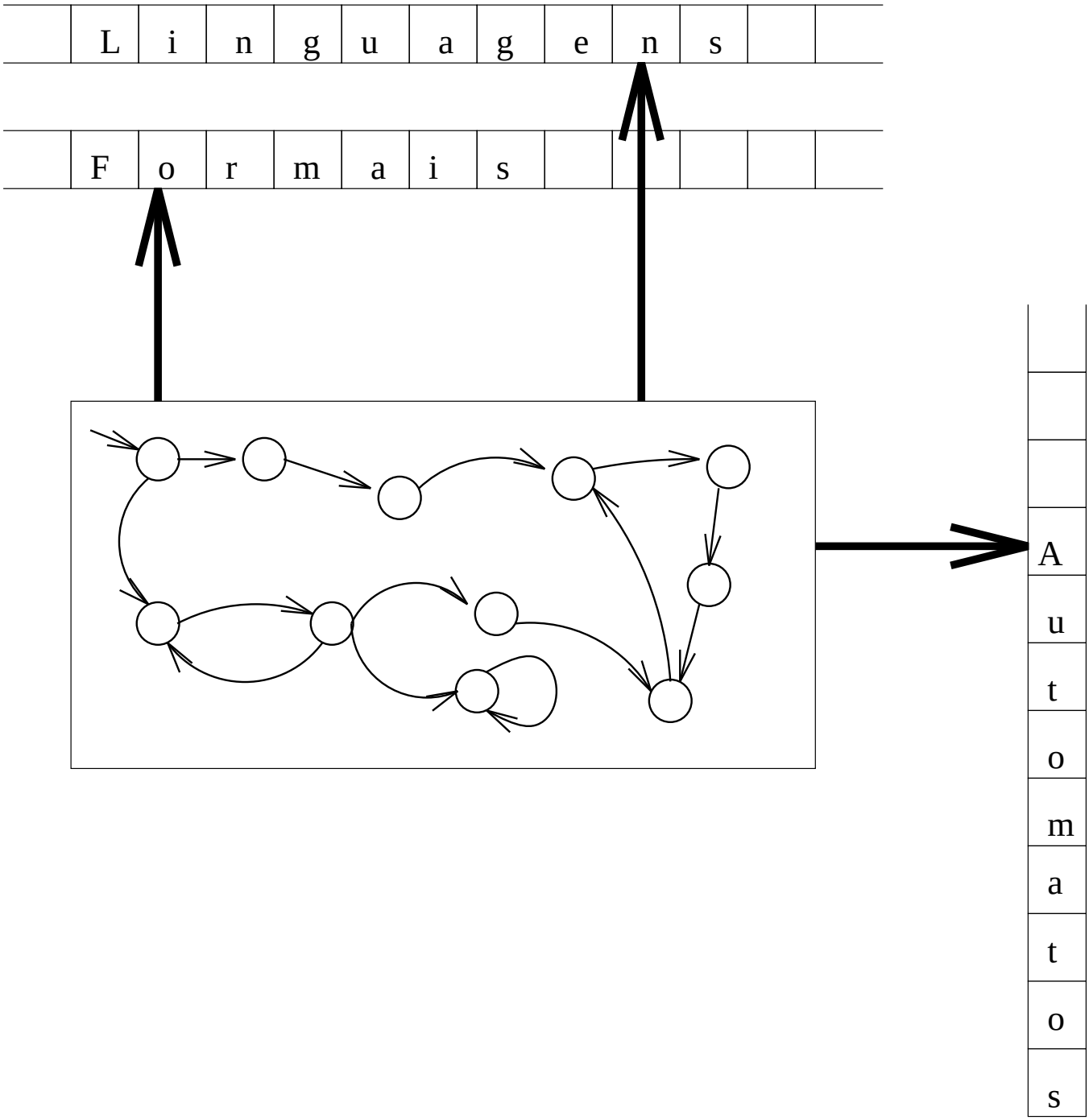

Linguagens
Formais
Autómatos

Índice

1 Introdução **1**

1.1 Alfabetos e Palavras 2

1.2 Linguagens 4

1.3 Exercícios 5

1.4 Implementação 7

2 Autômatos Finitos Determininísticos **11**

2.1 Definições Básicas 13

2.2 Exemplos de Autômatos Finitos Determinísticos 18

2.3 Função de Transição Extendida 20

2.4 Exercícios 22

2.5 Implementação 22

3 Minimização de AFD's **33**

3.1 Equivalência entre AFD's 33

3.2 AFD's não conexos 37

3.3 AFD com Estados Equivalentes 39

3.4 Algoritmo de Minimização 46

3.5 Implementação 47

4 Autômatos Finitos Não Determinísticos **55**

4.1 Definições Básicas 55

4.2 AFND's com transições λ 61

4.3 Equivalência entre AFD's e AFND's 65

4.3.1 Eliminação de Transições λ 66

4.3.2 Transformação de um AFND num AFD 68

4.4 Implementação 71

5 Expressões Regulares **85**

5.1 Definições Básicas 85

5.2 Equivalência entre ER e AF 89

5.2.1 Transformação de uma ER num AF 89

5.2.2 Transformação de um AFND em uma ER 93

6 Gramáticas Regulares **99**

6.1 Definição Básica 99

6.2 Hierarquia de Gramáticas 102

6.3 Exemplos de Linguagens Definidas através de GR's 105

6.4 Equivalência entre AFD's e GR's 107

6.4.1 Tranformação de um AFD numa GLD 107

6.4.2 Transformação de uma GLD num AFND 109

6.5 Expressões Regulares X Gramáticas Regulares 111

6.6 Formas Normais 113

7 Gramáticas Livres de Contexto **117**

7.1 Linguagens Autômato-definíveis 117

7.2 Árvores de Derivação 120

7.3 Ambigüidade 122

7.4 Transformações sobre GLC's 126

7.4.1 Eliminação de Produções Vazias 126

7.4.2 Eliminação de Cadeias de Produções 128

7.4.3 Eliminação de Símbolos Inúteis 129

7.4.4 Forma Normal de Chomsky 132

7.4.5 Eliminação de Recursão à Esquerda 133

7.4.6 Forma Normal de Greibach 134

7.5 Teorema Pumping 138

7.6 Propriedades de Fechamento 139

7.7 Uma GLS para Pascal 140

7.8 Exercícios 140

8 Autômatos de Pilha **143**

8.1 Definições Básicas 143
8.2 Exemplos de AP's 147
8.3 Equivalência entre AP's e GLC's 149
8.3.1 Transformação de GLC para AP 149
8.3.2 Transformação de AP para GLC 151
8.4 Autômato de Pilha Determinístico 153
8.5 Exercícios 154

9 Máquinas de Turing **157**
9.1 Definições Básicas 157
9.2 Exemplos de Máquinas de Turing 159
9.3 Variações de MT's 160
9.4 Enumeração de Linguagens 165
9.5 Computação de Funções 166
9.6 Exercícios 170

10 Decidibilidade **173**
10.1 Problemas de Decisão 173
10.2 Problema da Parada de Máquinas de Turing 175
10.3 Problema da Fita Vazia 178
10.4 Problema da Parada para Σ^* 179
10.5 Outros Problemas 180
10.6 Exercícios 181

Capítulo 1

Introdução

Muitas vezes nos deparamos com o problema de definir um conjunto de cadeias de símbolos. Por exemplo, se quisermos nos referir ao conjunto M dos números binários que têm exatamente dois dígitos, podemos nos expressar da seguinte maneira:

$$M = \{00, 01, 10, 11\}$$

Porém, se quisermos nos referir ao conjunto N de todos os números binários, de qualquer tamanho, não poderíamos utilizar o mesmo método de enumeração de todos os elementos do conjunto, como fizemos acima. Poderíamos tentar a seguinte representação:

$$N = \{0, 1, 00, 01, 10, 11, 000, 001, ...\}$$

Talvez esta seja uma representação suficientemente clara para nós, humanos e instruídos. Por outro lado, ela dificilmente poderia ser utilizada para representar o conjunto N num programa de computador que, dada uma cadeia S, decidisse se S pertence ou não a N. Em outras palavras, a representação acima não é formal o suficiente de modo a poder ser "entendida" por um computador.

Nestes dois casos, os símbolos básicos que compõem os elementos de M e de N são os mesmos – os dígitos 0 e 1. Essas seriam as **letras** de um **alfabeto** e que compõem as **palavras** (00, 01, ...) que formam os conjuntos M e N. E cada um desses conjuntos seria uma **linguagem** sobre esse alfabeto.

Um exemplo mais interessante é o conjunto de todos os programas válidos numa determinada linguagem de programação, digamos, C. Cada programa em C é composto por uma seqüência de símbolos que podem ser:

- identificadores como `x y MyFunction`
- palavras reservadas como `for while do goto`
- operadores como `+ - * / ++ -- > < >> <<`

- outros símbolos especiais como `( ) { } , ;`

Assim, podemos ver um programa em C como uma cadeia composta por elementos menores que são os símbolos descritos nos 4 itens acima. Obviamente, é impossível enumerar quais seriam todos os programas válidos em C, mas ainda assim precisamos descrever de maneira precisa e formal quais seriam esses programas.

É disso que trata este texto. De maneiras para definirem-se formalmente linguagens, em especial aquelas que não podem ser trivialmente enumeradas. Nas próximas seções deste capítulo serão apresentados alguns conceitos básicos – entre eles os de alfabetos e linguagens – que serão utilizados no restante deste texto.

1.1 Alfabetos e Palavras

Linguagens são compostas por palavras, que por sua vez são compostas por letras. Assim, as primeiras definições requeridas são:

Definição 1.1 *Um alfabeto é um conjunto finito, não vazio de símbolos de tamanhos finitos.*

Definição 1.2 *Dado um alfabeto Σ, a é uma letra de Σ sse $a \in \Sigma$.*

Por, exemplo, os seguintes conjuntos são alfabetos:

- $\{0, 1\}$
- $\{a, b\}$
- $\{00, 11\}$
- $\{a, ab, abc, abcd, abcde, abcdef\}$

Essas duas definições são bastante "liberais" no sentido de que quase qualquer conjunto de símbolos pode ser um alfabeto. Embora chamemos de **letra** a cada um desses símbolos, eles não precisam ter um único caracter, nem precisam ter todos o mesmo número de caracteres, como mostram os exemplos acima. A única restrição que se faz é que o tamanho (número de caracteres) de cada letra seja finito.

Ao justaporem-se letras de um alfabeto, obtêm-se **palavras** sobre este alfabeto. Palavras também costumam ser chamadas de **cadeias** ou **strings**.

Definição 1.3 *Dado um alfabeto Σ e uma seqüência de símbolos $x = a_1a_2...a_n$, diz-se que x é uma palavra sobre Σ sse cada $a_i \in \Sigma$, $i = 1, 2, ..., n$.*

Podemos também definir o **tamanho de uma palavra**, que é o número de letras que compõem essa palavra. Então, se $x = a_1a_2...a_n$, temos que o tamanho de x, denotado por $|x|$, é n.

Dado o alfabeto Σ, define-se Σ^* como o conjunto de todos os strings que podem ser derivados a partir das letras de Σ. Além desse, são definidos também os conjuntos Σ^k que são compostos por strings sobre Σ que têm tamanho k. Por exemplo, tomando $\Sigma = \{0, 1\}$, temos:

- $\Sigma^1 = \{0, 1\}$
- $\Sigma^2 = \{00, 01, 10, 11\}$
- $\Sigma^3 = \{000, 001, 010, 011, 100, 101, 110, 111\}$

O conjunto Σ^0 é também definido e é composto por um único string especial chamado de **string vazio** que é denotado pelo símbolo λ. Esse string é o único de tamanho 0, e sempre faz parte de Σ^*, qualquer que seja o alfabeto Σ.

Definição 1.4 *Dado o alfabeto Σ e $x, y \in \Sigma^*$, a* **concatenação** *de x e y, indicada por $x.y$ produz uma palavra formada pelas letras de x seguidas pelas letras de y. Ou seja, se $x = a_1a_2...a_n$ e $y = b_1b_2...b_m$, então $x.y = a_1a_2...a_nb_1b_2...b_m$.*

O string vazio é o elemento identidade da operação de concatenação. Dada qualquer cadeia x, temos que $x.\lambda = \lambda.x = x$. Define-se ainda o conjunto Σ^+ que é o conjunto de todos os strings sobre Σ que têm tamanho maior ou igual a 1. Resumindo, temos:

- $\Sigma^* = \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cup ...$
- $\Sigma^+ = \Sigma^* - \{\lambda\} = \Sigma^1 \cup \Sigma^2 \cup ...$

Vamos agora tomar os alfabeto $\Sigma_1 = \{0, 1\}$ e $\Sigma_2 = \{00, 11\}$. Sobre estes alfabetos, são definidas as seguintes palavras: $x = 0011 \in {\Sigma_1}^*$ e $y = 0011 \in {\Sigma_2}^*$. É importante compreender que, apesar de serem " visualmente semelhantes", eles não são o mesmo string. Cada string deveria, na verdade, ser representado por uma ênupla ordenada composta pelas letras que formam a palavra. Assim, x deveria ser representado com $< 0, 0, 1, 1 >$ e y deveria ser representado como $< 00, 11 >$. Portanto, vemos claramente que $< 0, 0, 1, 1 > \neq < 00, 11 >$. Para afirmar que duas palavras x e y são iguais, deveríamos ter, primeiramente, que ambas são formadas sobre o mesmo alfabeto. Além disso, todas as letras dos dois strings deveriam coincidir. Mais precisamente:

Definição 1.5 *Dados dois strings $x = a_1a_2...a_n$ e $y = b_1b_2...b_m$ sobre o alfabeto Σ, diz que $x = y$ sse $m = n$ e $a_i = b_i$ para todo i de 1 até n.*

Concatenando-se palavras, podem ser criadas outras palavras compostas. Inversamente, podemos pensar em subdividir uma palavra em diversas partes. Podemos então fazer as seguintes definições:

Definição 1.6 *Dado o alfabeto Σ e $x, y \in \Sigma^*$, diz-se que*

- *x é um* **prefixo** *de y sse $\exists w \in \Sigma^* \ni y = xw$*
- *x é um* **sufixo** *de y sse $\exists u \in \Sigma^* \ni y = ux$*
- *x é uma* **subpalavra** *de y sse $\exists u, w \in \Sigma^* \ni y = uxw$*

Dados os strings $x = abcababccbca \in \{a, b, c\}^*$, temos, por exemplo:

- abc e $abcaba$ são prefixos de x
- bca e $ccbca$ são sufixos de x
- abc, bca e bab são subpalavras de x

1.2 Linguagens

Uma linguagem é simplesmente um conjunto de palavras. Ou seja:

Definição 1.7 *Dado o alfabeto Σ, o conjunto de palavras L é uma linguagem sobre Σ, sse $L \subseteq \Sigma^*$.*

Por exemplo, dado $\Sigma = \{0, 1\}$ podemos definir as seguintes linguagens:

- $L_1 = \{\lambda\}$
- $L_2 = \{01, 0110, 010101111\}$
- $L_3 = \{\lambda, 01, 0110, 010101111\}$
- $L_4 = \{x | x = 0^n1^n, n \geq 0\}$

Apesar da semelhança, L_2 e L_3 são linguagens distintas. L_3 possui um elemento a mais, o string vazio, que não faz parte de L_2. As linguagens L_1, L_2 e L_3 são linguagens finitas e por isso utilizamos a enumeração de todos os seus

elementos para descrevê-las. Já a linguagem L_4 é infinita e por isso necessitamos alguma outra maneira de descrever quais palavras estão nela contidas. Nesse caso, utilizou-se a "expressão" $0^n 1^n$, indicando todas as palavras que têm um certo número n de letras 0 seguidas pelo mesmo número de letras 1, para qualquer n inteiro maior que ou igual a 0 (portanto, λ está presente nesta linguagem).

Assim como fazemos com conjuntos quaisquer, podemos definir certas operações sobre as linguagens.

Definição 1.8 *Dadas as linguagens L_1 e L_2 sobre o alfabeto Σ, definem-se as seguintes operações:*

- *União:* $L_1 \cup L_2 = \{x | x \in L_1 \vee x \in L_2\}$
- *Intersecção:* $L_1 \cap L_2 = \{x | x \in L_1 \wedge x \in L_2\}$
- *Diferença:* $L_1 - L_2 = \{x | x \in L_1 \wedge x \notin L_2\}$
- *Concatenação:* $L_1.L_2 = \{x | x = y.z, y \in L_1 \wedge z \in L_2\}$
- *Complemento:* $\overline{L_1} = \{x | x \in \Sigma^* \wedge x \notin L_1\}$

Todas essas operações produzem conjuntos que estão contidos em Σ^*, ou seja, produzem outras linguagens sobre Σ. Isso equivale a dizer que Σ^* é **fechado** sob essas operações. Vejamos algumas exemplos. Sejam $L_1 = \{0, 11\}$ e $L_2 = \{0, 1, 00\}$, linguagens sobre $\{0, 1\}$. Temos:

- $L_1 \cup L_2 = \{0, 11, 1, 00\}$
- $L_1 \cap L_2 = \{0\}$
- $L_1 - L_2 = \{11\}$
- $L_1.L_2 = \{00, 01, 000, 110, 111, 1100\}$

1.3 Exercícios

1.1 *Dados x, y e $z \in \Sigma^*$, mostre que:*

a. se x é prefixo de y e y é prefixo de x, então $x = y$

b. se x é prefixo de y e y é prefixo de z, então x é prefixo de z

1.2 *Dada a linguagem L, mostre que $\overline{L}$ é infinita se L é finita. E se L for infinita?*

1.3 *Dados x, y, u, e $v \in \Sigma^*$, tal que $x.y = u.v$, mostre que*

- *x é prefixo de u e v é sufixo de y ou*
- *u é prefixo de x e y é sufixo de v*

1.4 *Dados* $L_1 = \{a, ab\}$ *e* $L_2 = \{\lambda, a, ba\}$*, linguagens sobre* $\{a, b\}$*, determine*

a. $L_1 \cup L_2$

b. $L_1 \cap L_2$

c. $L_1 - L_2$

d. $L_2 - L_1$

e. $L_1.L_2$

f. $L_2.L1$

g. ${L_1}^2 = L_1.L1$

h. ${L_2}^2 = L_2.L2$

i. $\overline{L_1}$

1.5 *Dada uma palavra* $x = a_1a_2...a_n$*, define-se a reversa (ou transposta) de* x*, denotada por* x^r *como sendo* $x^r = a_na_{n-1}...a_1$*. Dada a linguagem* L*, define-se a operação de reversão (ou transposição) de* L *como sendo*

$$L^r = \{x | x^r \in L\}$$

Usando L_1 *e* L_2 *do exercício anterior, calcule* ${L_1}^r$ *e* ${L_2}^r$*.*

1.6 *Dadas* L_1*,* L_2 *e* L_3*, linguagens sobre o alfabeto* Σ*, mostre que:*

a. $(L_1 \cup L_2)^r = {L_1}^r \cup {L_2}^r$

b. $(L_1.L_2)^r = {L_2}^r.{L_1}^r$

c. $(L_1 \cup L_2).L_3 = (L_1.L_3) \cup (L_2.L_3)$

d. $(L_1 \cup L_2)^2 = {L_1}^2 \cup (L_1.L_2) \cup (L_2.L1) \cup {L_2}^2$

e. $L_1.\emptyset = \emptyset$

f. $L_1.\{\lambda\} = L_1$

1.4 Implementação

Nos algoritmos que serão apresentados nos próximos capítulos, precisaremos representar alfabetos que serão associados aos dispositivos como os Autômatos Finitos. Como os alfabetos que trabalhamos costumam ter cardinalidade reduzida, escolhemos uma simples tabela sequencial onde as letras do alfabeto são armazenadas. O leitor pode alterar essa implementação, utilizando soluções mais eficientes, caso necessite trabalhar com alfabetos extensos.

A interface desse módulo é definida no arquivo "alfabeto.h" pelas seguintes funções, que devem ser mantidas, caso o leitor resolva efetuar mudanças no módulo.

```
/*******************************************************
                        alfabeto.h

Data: 2/3/98
Autor: Marcio Delamaro
*******************************************************/
typedef struct alfabeto ALFABETO;

ALFABETO *ALFACria(char *);
int       ALFACard(ALFABETO *);
int       ALFAOrdem(ALFABETO *, char *);
char     *ALFALetra(ALFABETO *, int);
void      ALFADestroi(ALFABETO *);
```

Nesse arquivo é declarado o tipo **ALFABETO**, que será mostrado adiante, no arquivo "alfabeto.c". As funções que caracterizam este tipo de dado são as seguintes:

- `ALFABETO *ALFACria(char *)`

  Essa função cria um novo alfabeto. Recebe como parâmetro uma cadeia de caracteres que contém as letras do alfabeto, separadas por brancos ou tab's. Devolve como resultado o endereço do alfabeto criado. Esse endereço é utilizado como parâmetro para todas as demais funções deste módulo. Dessa forma, podem ser criados diversos alfabetos distintos durante a execução de um programa.

- `int ALFACard(ALFABETO *)`

  Essa função recebe como parâmetro o endereço de um alfabeto que foi criado anteriormente. Retorna o número de letras que o alfabeto contém.

- `int ALFAOrdem(ALFABETO *, char *)`

  Recebe como parâmetro o endereço de um alfabeto e um string que deve ser uma das letras do alfabeto. Retorna um número inteiro que vai de 0 a $ALFACard - 1$ e que corresponde à ordem que aquela letra ocupa dentro do alfabeto. Se o string passado não é uma letra do alfabeto, retorna -1;

- `char *ALFALetra(ALFABETO *, int)`

Essa função realiza a operação inversa da função `ALFAOrdem`, ou seja, dado um número de ordem, devolve qual é a letra correspondente. Se o número passado como parâmetro estiver fora do intervalo de 0 a $ALFACard-1$, então a função devolve o valor NULL ou `( char *) 0`.

- `void ALFADestroi(ALFABETO *)`

  Recebe como parâmetro o endereço de um alfabeto. Libera a memória utilizada pelo alfabeto, que deixa de existir e não pode mais ser utilizado.

A seguir, é mostrado o arquivo "alfabeto.c" que contém o código dessas funções,

```
/*****************************************************
                          alfabeto.c
Data: 2/3/98
Autor: Marcio Delamaro
*****************************************************/

#include    <malloc.h>
#include    <string.h>


struct alfabeto {        /* Estrutura que armazena um alfabeto */
    char    **letras;        /* apontador para as letras do alfabeto */
    int    nletras;        /* numero de letras armazenadas */
   };

#include    "alfabeto.h"

/*-----------------------------------------------------
                          ALFACria
Aloca memoria e armazena um alfabeto.
Parametros:
char *x   -    String que contem todas as letras do
               alfabeto separadas por espaco ou por tab
Retorna:
ALFABETO * - Ponteiro para o objeto do tipo ALFABETO
             que foi criado.
             NULL se nao conseguiu criar o objeto
-----------------------------------------------------*/

ALFABETO *ALFACria(char *x)
{
char *q, *r;
ALFABETO *alfa;
int fim, nletras;

   nletras = 0;                /* numero de letras do alfabeto */
   while ( isspace(*x)) x++; /* despreza brancos no comeco do string */
   q = x;
   fim = (q == '\0');
   while ( ! fim )          /* conta numero de letras do alfabeto */
   {
        while (! isspace(*q) && *q != '\0') q++;
        fim = (*q == '\0');
        *q++ = '\0';
        nletras++;
        while ( !fim && isspace(*q)) q++;
   }
```

```
   alfa = malloc(sizeof *alfa); /*aloca memoria para a struct alfabeto */
   if (alfa == NULL)
       return NULL;

   /* aloca memoria para o vetor de (char *) para armazenar as letras */
   alfa->letras = malloc(nletras * sizeof (char *));
   if (alfa ->letras == NULL)
   {
        free(alfa);
        return NULL;
   }
   alfa->nletras = 0;

   q = x;
   while ( alfa->nletras < nletras ) /* armazena cada letra no vetor */
   {
      alfa->letras[alfa->nletras++] = r = malloc( strlen(q) +1);
      if (r == NULL)
      {
         alfa->nletras--;
         ALFADestroi(alfa);
         return NULL;
      }
      strcpy(r, q);
      while ( *q != '\0') q++;
      q++;
      while ( alfa->nletras < nletras && isspace(*q)) q++;
   }
   return alfa;
}


/*-----------------------------------------------------
                       ALFACard
Retorna a cardinalidade do alfabeto
-----------------------------------------------------*/

int ALFACard(ALFABETO *alfa)
{
   return alfa->nletras;        /* retorna o numero de letras do alfabeto */
}

/*-----------------------------------------------------
                     ALFAOrdem
Retorna um numero que corresponde a posicao da letra
dentro do alfabeto
-----------------------------------------------------*/
int ALFAOrdem(ALFABETO *alfa, char *x)
{
int i;

    for (i = 0; i < alfa->nletras; i++)  /* procura letra */
       if (strcmp(alfa->letras[i], x) == 0)
            return i;   /* retorna o indice da letra no vetor */
    return -1;        /* nao faz parte do alfabeto*/
}

/*-----------------------------------------------------
                     ALFALetra
Retorna a i-esima letra do alfabeto
-----------------------------------------------------*/
char *ALFALetra(ALFABETO *alfa, int i)
```

```
{
   /* se i < nletras, devolve o string que esta na posicao i do vetor,
      senao NULL */
   return (i < alfa->nletras)? alfa->letras[i]: NULL;
}


/*-----------------------------------------------------
                      ALFADestroi
Libera o alfabeto
-----------------------------------------------------*/
void ALFADestroi(ALFABETO *alfa)
{
int i;

    for (i = 0; i < alfa->nletras; i++) /* libera letras */
        free(alfa->letras[i]);
    free(alfa->letras);        /* libera vetor */
    free(alfa);                /* libera alfabeto */
}
```

Capítulo 2

Autômatos Finitos Determininísticos

Um Autômato Finito Determinístico (AFD) é um modelo matemático que permite que uma linguagem possa ser definida de maneira formal. Além disso, AFD's são usados para modelar sistemas com entradas e saídas discretas e que operam internamente através de vários estados que mudam à medida que a entrada é processada. Cada estado estabelece um modo de operação, possivelmente distinto. Em outras palavras, cada estado estabelece uma maneira de tratar os próximos eventos que compõem a entrada.

Um exemplo clássico de tais sistemas é o problema envolvendo um homem, um lobo, uma cabra e um repolho (HLCR), sugerido por Hopcroft e Ullman [HOP79]. Nesse problema, um homem tem um lobo, uma cabra e um repolho na margem esquerda de um rio e precisa atravessá-los para a margem direita utilizando um pequeno bote onde só cabem ele e mais um dos outros três. Além disso, ele não pode deixar sozinhos nas margens o lobo e a cabra (o lobo comeria a cabra) nem a cabra e o repolho (a cabra comeria o repolho). O estado em que o sistema se encontra pode ser representado pela informação de quais são os ocupantes das duas margens. Por exemplo:

- $< HLCR - \emptyset >$ - todos estão na margem esquerda
- $< L - HCR >$ - lobo na margem esquerda, homem, cabra e repolho na margem direita

As "entradas" do sistema são as ações tomadas pelo homem, que fazem com que o estado corrente do sistema seja alterado. Essas ações são:

h - homem atravessa o rio sozinho

l - homem atravessa com o lobo

c - homem atravessa com a cabra

r - homem atravessa com o repolho

O objetivo é estabelecer uma seqüência de ações que levem do estado inicial $< HLCR - \emptyset >$ ao estado final $< \emptyset - HLCR >$.

O comportamento desse sistema pode ser modelado pelo diagrama da Figura 2.1, onde cada círculo representa um estado e cada arco uma ação, ou transição de um estado para outro. Qualquer seqüência de transições que leve ao estado final, marcado por um círculo duplo, como **chrclhc**, é uma resposta para o problema.

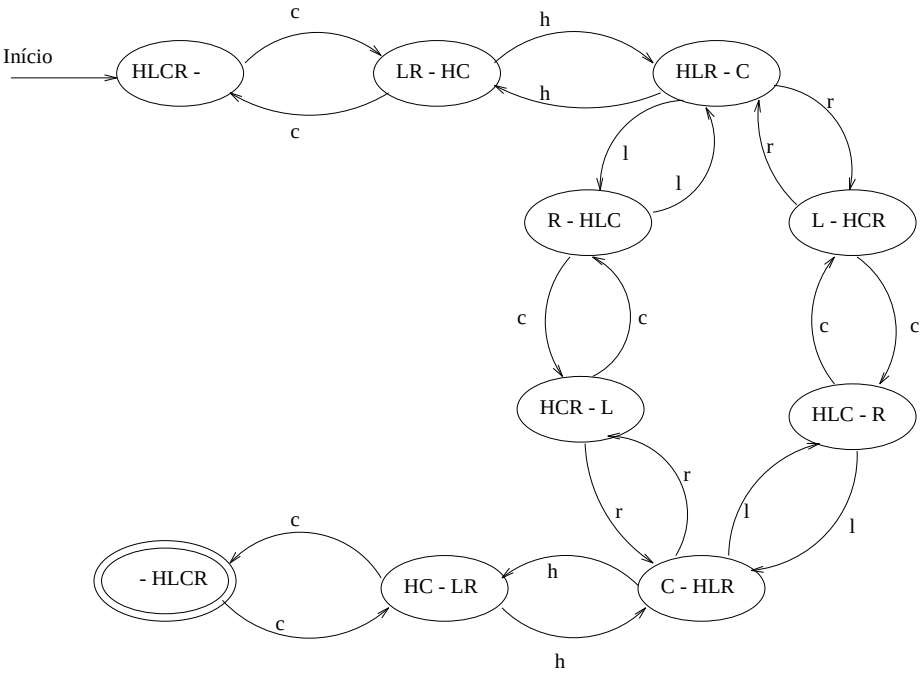


Figura 2.1: Diagrama representando o problema HLCR

Além desse exemplo – puramente acadêmico – encontramos diversos outros sistemas que podem ser representados dessa maneira. Um forno micro-ondas, por exemplo, processa entradas como: o sensor da porta aberta/fechada; os comandos fornecidos pelo(a) cozinheiro(a) através do seu painel; o sinal do "timer" que expira; etc. Cada um desses eventos ou ações faz com que o micro-ondas assuma diferentes estados como: aberto; esperando por comandos; cozinhando; desligado. Assim como este, vários outros sistemas com os quais estamos acostumados a conviver cotidianamente poderiam ser modelados através de estados e transições.

Também um Engenheiro de Software, ao projetar a interface de seu sistema determina quais são os estados que ela deve assumir à medida que certas ações são tomadas, como um tecla pressionada, um botão selecionado pelo mouse ou um item de menu escolhido. No resto deste capítulo são apresentadas as definições básicas sobre Autômatos Finitos Determinísticos.

2.1 Definições Básicas

Um AFD A é um dispositivo que define formalmente uma linguagem $L(A)$ sobre um dado alfabeto Σ. Mais precisamente, dada uma cadeia $x \in \Sigma^*$, o autômato A deve ser capaz de responder se x pertence ou não à linguagem $L(A)$. Ou seja, um AFD tem um caráter de **reconhecedor** de cadeias, ao contrário de outras formas de representarem-se linguagens que serão estudadas posteriormente.

Uma abstração que se pode fazer de um AFD é a da Figura 2.2. O AFD possui um fita sobre a qual está a palavra que se deseja analisar. Esta fita é dividida em células, cada uma contendo uma letra da palavra. Sobre esta fita existe uma cabeça de leitura que extrai, seqüencialmente, o conteúdo de uma célula da fita. Há também um Controle Finito de Estados que reage a cada letra lida da fita pela cabeça de leitura. Finalmente, existe uma luz de aceitação que ao final do processamento da cadeia, ou seja, quando a cabeça de leitura passou por todas as letras da palavra analisada, acende somente se a cadeia pertence à linguagem representada pelo AFD. Se a cadeia não pertence à linguagem, a lâmpada permanece apagada.

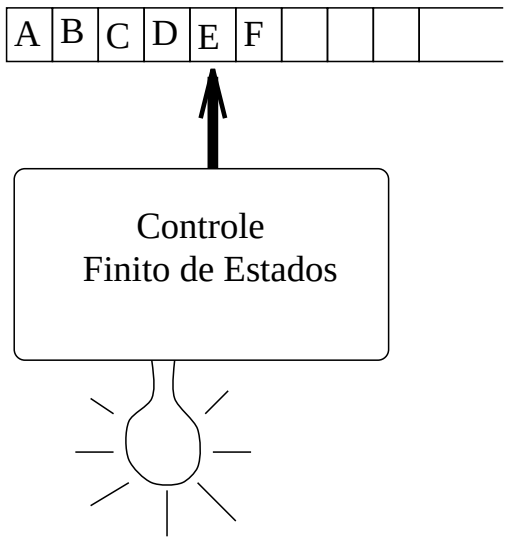


Figura 2.2: Abstração de um AFD como reconhecedor de cadeias

No caso do problema HLCR, por exemplo, a linguagem que se deseja representar é aquela formada pelos strings que levem à solução do problema, ou seja, que levem do estado inicial até (exatamente) ao estado final, como **chrclhc** ou **ccchllrcllllhccc**.

A definição matemática de um AFD é:

Definição 2.1 *Um Autômato Finito Determinístico é uma quíntupla* $< \Sigma, S, S_0, \delta, F >$, *onde*

Σ *é o alfabeto de entrada*

S *é um conjunto finito não vazio de estados*

S_0 *é o estado inicial,* $S_0 \in S$

δ *é a função transição de estados, definida* $\delta : S \times \Sigma \to S$

F *é o conjunto de estados finais,* $F \subseteq S$

Σ é o alfabeto sobre o qual a palavra que está na fita é definidas, ou seja, as palavras que podem aparecer na fita pertencem a Σ^*. O Conjunto S é o conjunto de estados que o autômato pode assumir e o estado S_0 é o estado inicial, no qual o autômato é colocado ao iniciar o processamento de um string. O conjunto F indica quais dos estados de S são estados finais. Um string x para ser aceito deve levar o autômato do estado S_0 até um dos estados pertencentes a F ao terminar de ser processado.

A função δ determina como o AFD muda de um estado para outro. Ela é definida em $\delta : S \times \Sigma \to S$, ou seja, leva um par $< s, a >$, onde s é um estado e a é uma letra do alfabeto num estado s'. O significado de $\delta(s, a) = s'$ é que o autômato, estando no estado s, ao processar a letra a da fita, irá passar para o estado s'.

Resumindo, dado o AFD $A =< \Sigma, S, S_0, \delta, F >$ e o string $x = a_1 a_2 ... a_n \in \Sigma^*$, o autômato inicial sua execução no estado S_0. Ao processar a letra a_1 ele passa para o estado $\delta(S_0, a_1)$. Ao processar a_2 ele passa ao próximo estado $\delta(\delta(S_0, a_1), a_2)$, e assim por diante até que a_n seja processado. Nesse ponto o autômato estará num estado qualquer R. Se $R \in F$, então o string x pertence à linguagem definida por A, ou seja, $x \in L(A)$. Se $R \notin F$ então $x \notin L(A)$.

O termo **determinístico** é utilizado para estabelecer que, para qualquer cadeia $x \in \Sigma^*$, só existe uma única seqüência de estados no autômato para processar x. O termo **finito** indica que o número de estados do autômato tem que ser finito.

Vamos então ver um exemplo da utilização de AFD, conforme sua definição. Uma "vending machine" aceita moedas de 5, 10 e 25 centavos. O preço do produto que ela entrega é 30 centavos. Um AFD que modela o funcionamento dessa máquina deve registrar quantos centavos foram depositados, até chegar num valor suficiente para pagar o produto, ou seja, maior que ou igual a 30 centavos. Nesse caso, atingiu-se um estado que indica que o produto pode ser liberado. Vamos então definir um AFD que modela esta máquina. Temos $V = < \Sigma, S, S_0, \delta, F \}$, onde

- $\Sigma = \{5, 10, 25\}$. Cada uma das letras indica, respectivamente, a ação de colocar uma moeda de 5, 10 ou 25 centavos.

- $S = \{< 0c >, < 5c >, < 10c >, < 15c >, < 20c >, < 25c >, < 30c >, < 35c >, < 40c >, < 45c >, < 50c >\}$. Cada estado indica quantos centavos foram depositados.

- $S_0 = < 0c >$. A operação da máquina inicia sem nada depositado, ou seja, no estado $< 0c >$.

- $F = \{< 30c >, < 35c >, < 40c >, < 45c >, < 50c >\}$. Ao atingir qualquer estado que indique um valor maior do que 30 centavos, deve-se considerar que a seqüência de entrada é valida, e o produto pode ser liberado.

- *delta* é definida como:

$$\begin{array}{ll}
\delta(< 0c >, 5) =< 5c > & \delta(< 0c >, 10 >=< 10c > \\
\delta(< 0c >, 25) =< 25c > & \delta(< 5c >, 5) =< 10c > \\
\delta(< 5c >, 10 >=< 15c > & \delta(< 5c >, 25) =< 30c > \\
\delta(< 10c >, 5) =< 15c > & \delta(< 10c >, 10 >=< 20c > \\
\delta(< 10c >, 25) =< 35c > & \delta(< 15c >, 5) =< 20c > \\
\delta(< 15c >, 10 >=< 25c > & \delta(< 15c >, 25) =< 40c > \\
\delta(< 20c >, 5) =< 25c > & \delta(< 20c >, 10 >=< 30c > \\
\delta(< 20c >, 25) =< 45c > & \delta(< 25c >, 5) =< 30c > \\
\delta(< 25c >, 10 >=< 35c > & \delta(< 25c >, 25) =< 50c > \\
\delta(< 30c >, 5) =< 35c > & \delta(< 30c >, 10 >=< 40c > \\
\delta(< 30c >, 25) =< 50c > & \delta(< 35c >, 5) =< 40c > \\
\delta(< 35c >, 10 >=< 45c > & \delta(< 35c >, 25) =< 50c > \\
\delta(< 40c >, 5) =< 45c > & \delta(< 40c >, 10 >=< 50c > \\
\delta(< 40c >, 25) =< 50c > & \delta(< 45c >, 5) =< 50c > \\
\delta(< 45c >, 10 >=< 50c > & \delta(< 45c >, 25) =< 50c > \\
\delta(< 50c >, 5) =< 50c > & \delta(< 50c >, 10 >=< 50c > \\
\delta(< 50c >, 25) =< 50c > &
\end{array}$$

Então, se o usuário de vending machine deposita 5 centavos e depois mais cinco centavos e depois 25 centavos teríamos a cadeia **5525** a ser analisada pelo nosso AFD. O comportamento do AFD seria o representado na Figura 2.3. O estado $< 35c >$, atingido ao final do processamento da cadeia (Figura 2.3d) é um estado final, indicando que **5525** é uma cadeia válida e que o produto pode ser liberado.

Por outro lado, se o usuário depositasse apenas 20 centavos, com duas moedas de 5 e uma moeda de dez, nessa ordem, teríamos a cadeia 5510 a ser analisada e o comportamento do autômato seria o da Figura 2.4.

Existem maneiras menos tediosas de descrever-se a função δ do que a vista no exemplo da vending machine. Uma delas é através de uma **tabela de transição de estados**. Tal tabela relaciona em cada linha um estado e em cada coluna uma letra do alfabeto. No encontro de linha x coluna é colocado o valor da função δ. A tabela abaixo mostra a função de transição do AFD da vending machine.

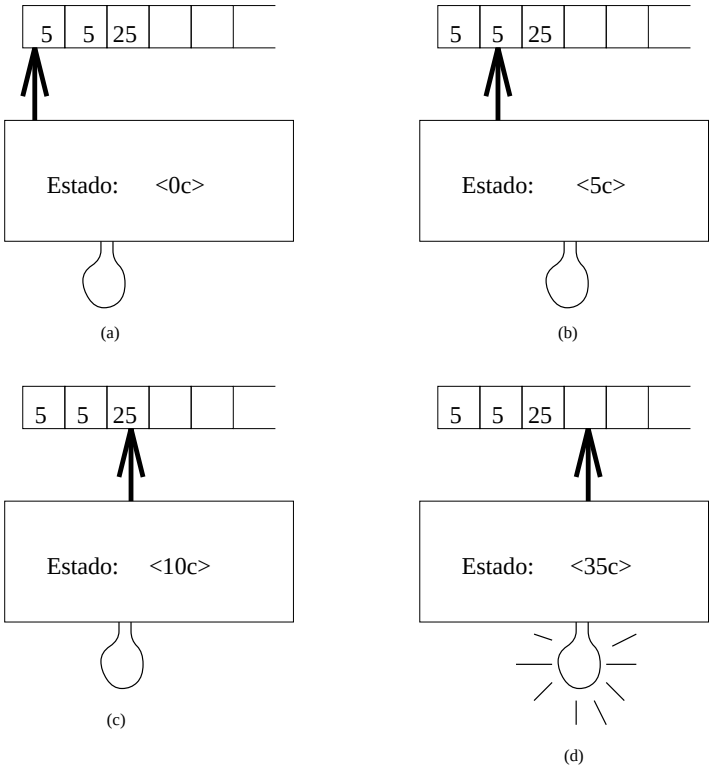


Figura 2.3: Operação da vending machine para a cadeia **5525**

δ	5	10	25
$< 0c >$	$< 5c >$	$< 10c >$	$< 25c >$
$< 5c >$	$< 10c >$	$< 15c >$	$< 30c >$
$< 10c >$	$< 15c >$	$< 20c >$	$< 35c >$
$< 15c >$	$< 20c >$	$< 25c >$	$< 40c >$
$< 20c >$	$< 25c >$	$< 30c >$	$< 45c >$
$< 25c >$	$< 30c >$	$< 35c >$	$< 50c >$
$< 30c >$	$< 40c >$	$< 40c >$	$< 50c >$
$< 35c >$	$< 45c >$	$< 45c >$	$< 50c >$
$< 40c >$	$< 50c >$	$< 50c >$	$< 50c >$
$< 45c >$	$< 50c >$	$< 50c >$	$< 50c >$
$< 50c >$	$< 50c >$	$< 50c >$	$< 50c >$

Mas a maneira mais utilizada é o **diagrama de transição de estados**, semelhante ao usado no exemplo do problema HLCR. Nesse diagrama, cada estado é representado por um círculo rotulado com o nome do estado e cada transição é representada por uma seta ligando dois estados. Esta seta é também rotulada, com a letra do alfabeto que faz com que a transição aconteça. Por exemplo, se tivermos a seguinte transição definida: $\delta(R, a) = S$, teríamos no diagrama o seguinte:

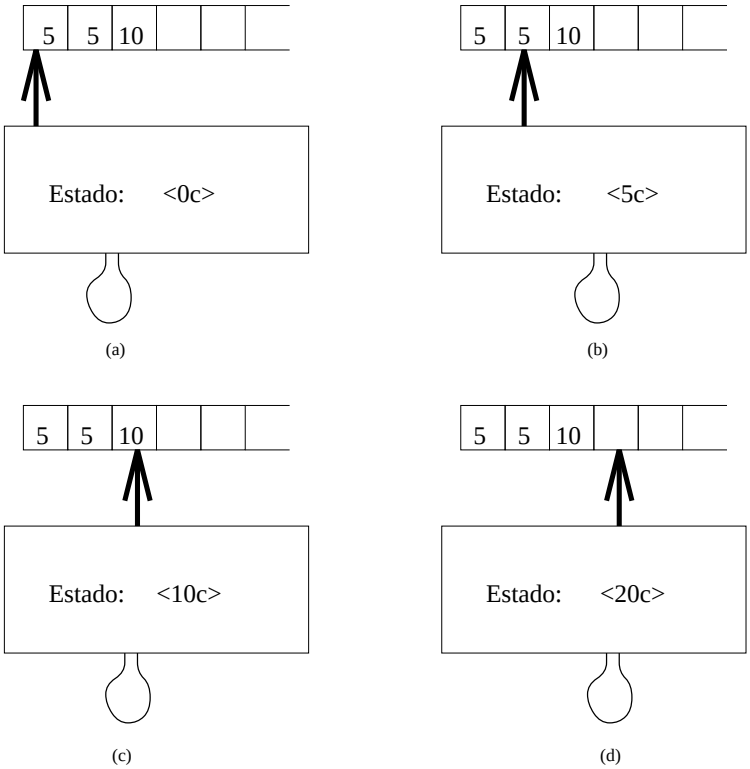
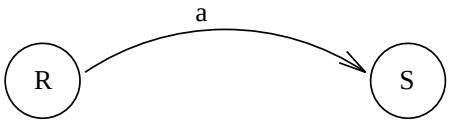
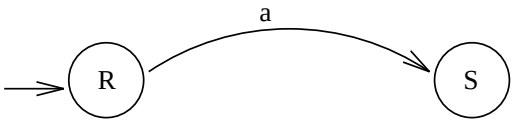


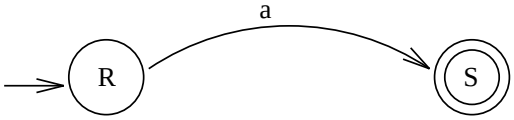
Figura 2.4: Operação da vending machine para a cadeia **5510**



No diagrama de transições, o estado inicial é indicado por uma seta chegando nesse estado, sem um estado de origem. Se R é o estado inicial temos:

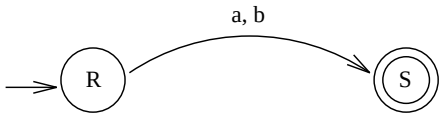


Os estados finais são marcados com círculos duplos.



Se existem mais do que uma transição de um estado R para um estado S, ou seja, $\delta(R, a) = \delta(R, b) = S$, podemos simplificar a representação do diagrama

da seguinte forma:



Assim, a função de transição de estados do AFD V, da vending machine, pode ser representada pelo diagrama de transição de estados da Figura 2.5.

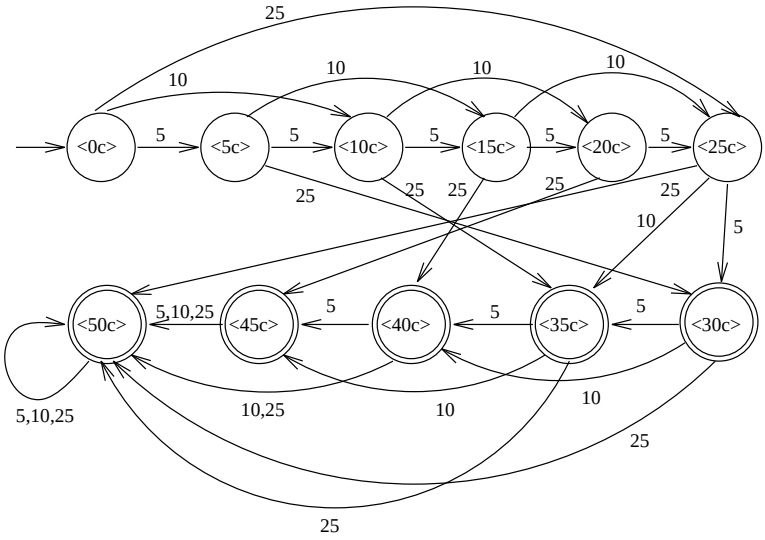


Figura 2.5: Diagrama de transição de estados para o AFD V da vending machine

Essa representação torna mais fácil a visualização de como o AFD reconhece (ou não reconhece) um string, através da visualização do "caminho" percorrido ao se analisar um string.

Voltando ao exemplo do HLCR, ao analisarmos o diagrama da Figura 2.1 vemos que ele não representa um AFD, de acordo com a nossa definição. (Por que?). Ao estudarmos os Autômato Finito não Determinístico iremos ver que aquele diagrama se adequa melhor àquele tipo de autômatos.

2.2 Exemplos de Autômatos Finitos Determinísticos

Vamos nesta seção mostrar alguns exemplos de linguagens definidas por AFD, procurando habituar o leitor a esse tipo de dispositivo. Iniciamos com dois exemplos triviais: as linguagens $L_1 = \emptyset$ e a linguagem $L_2 = \{\lambda\}$. Vamos supor que trabalhamos com o alfabeto $\Sigma = \{a, b\}$. Então, um AFD A_1 que gera L_1 seria $A_1 =< \Sigma, \{S\}, S, \delta, \emptyset >$, onde δ pode ser dada pelo diagrama da Figura 2.6a. Na verdade, qualquer que seja AFD que possua $F = \emptyset$ irá gerar L_1. (Por que?) O AFD A_1 é apenas o menor deles, em número de estados, uma vez que, por definição $|S| \geq 1$.

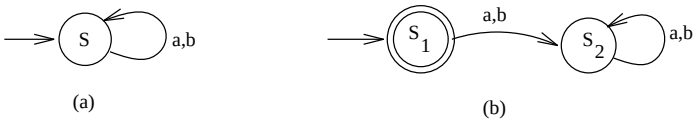


Figura 2.6: AFD's para a)$L_1 = \emptyset$ e b)$L_2 = \{\lambda\}$

Para reconhecer L_2 podemos definir $A_2 =< \Sigma, \{S_1, S_2\}, S_1, \delta, \{S_1\} >$, onde δ é dada pelo diagrama da Figura 2.6b. Nesse caso, o estado inicial S_1 é também o único estado final. O AFD só assume esse estado quando nenhuma letra foi consumida. Qualquer letra irá levar o AFD ao estado S_2. Assim, temos que $L(A_2) = L_2$.

Sobre esse mesmo alfabeto $\{a, b\}$ queremos agora definir a linguagem composta por todos os strings que têm um número ímpar de letras a e um número ímpar de letras b. Queremos então $L_3 = \{x \in \{a, b\}^* \mid |x|_a \equiv 1 \ mod\ 2 \wedge |x|_b \equiv 1 \ mod\ 2\}$[1]. A primeira ideia seria construir um AFD que contasse o número de cada uma das letras já lidas, colocando como final aqueles estados em que os números de a's e de b's sejam ímpares. Seria algo como o representado na Figura 2.7.

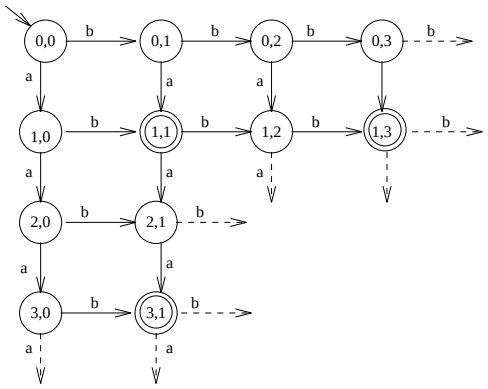


Figura 2.7: AFD's para contar o número de a's e b's

O problema óbvio dessa solução é que teríamos que ter um número infinito de estados, o que não é permitido pela definição de AFD's. Ao invés disso, vamos construir um AFD com 4 estados apenas, cada um representando as possíveis combinações de ímpar/par para cada uma das letras. O AFD seria $A_3 =< \{a, b\}, \{PP, PI, IP, II\}, PP, \delta, \{II\} >$, onde δ é expressa na Figura 2.8.

Uma aplicação onde AFD's são muito utilizados é na análise léxica de compiladores. O analisador léxico é responsável por separar e identificar cada parte de um programa, como: constantes, palavras reservadas, nomes de variáveis, etc. Para descrever a linguagem formada por esses itens léxicos, costumam-se utilizar AFD's. Vamos tomar, por exemplo, o conjunto das constantes numéricas de ponto flutuante. São exemplos válidos dessas constantes:

- 1.0

[1] Costuma-se utilizar a notação $|x|_a$ para denotar o número de letras a no string x.

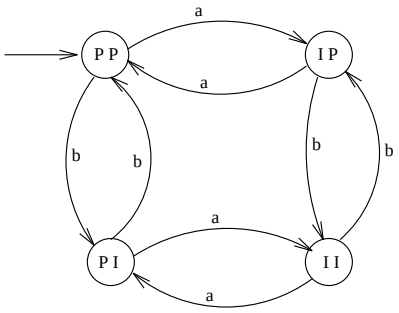


Figura 2.8: AFD para $L_3 = \{x \in \{a,b\}^* \mid |x|_a \equiv 1 \ mod\ 2 \wedge |x|_b \equiv 1 \ mod\ 2\}$

- 1
- +1.0E-3
- -15E+32
- -15E32
- 15.E32

Pana analisar um string e verificar se ele é uma constante válida, podemos definir o AFD $A_4 =< \{0,1,...9,+,-,E,.\}, \{S_0,S_1,S_2,S_3,S_4,S_5,S_6,S_7,S_8\}, S_0, \delta, \{S_2,S_3,S_5,S_8\} >$ onde a função δ é representada no diagrama da Figura 2.9. Nesse AFD, utiliza-se o estado S_7 como um "estado de erro". Em qualquer ponto do processamento da cadeia em que se encontre uma letra inválida, por exemplo, um **E** logo em seguida a um **+** inicial, passa-se ao estado S_7. Naquele estado, todas as letras restantes são consumidas, sem que haja possibilidade de se atingir um estado final. Dessa forma, ao transicionar o AFD para o estado S_7 deseja-se garantir que o string sendo analisado não será aceito.

2.3 Função de Transição Extendida

A função δ, é aplicada sobre um par $< estado\ X\ letra >$. Para definir diversas características de AFD's iremos necessitar de uma **Função de Transição Extendida**, denotada $\overline{\delta}$, que aplica-se a um par $< estado\ X\ cadeia >$, levando também a um estado. Se $\overline{\delta}(S,x) = R$, isso indica que, estando o autômato no estado S e processando o string x a partir desse estado, alcança-se o estado R. Mais precisamente, se $x = a_1a_2...a_n$, então $\overline{\delta}(S,x)$ é o resultado da aplicação de δ sobre $< S,a_1 >$, depois sobre $< \delta(S,a_1),a_2 >$, e assim por diante.

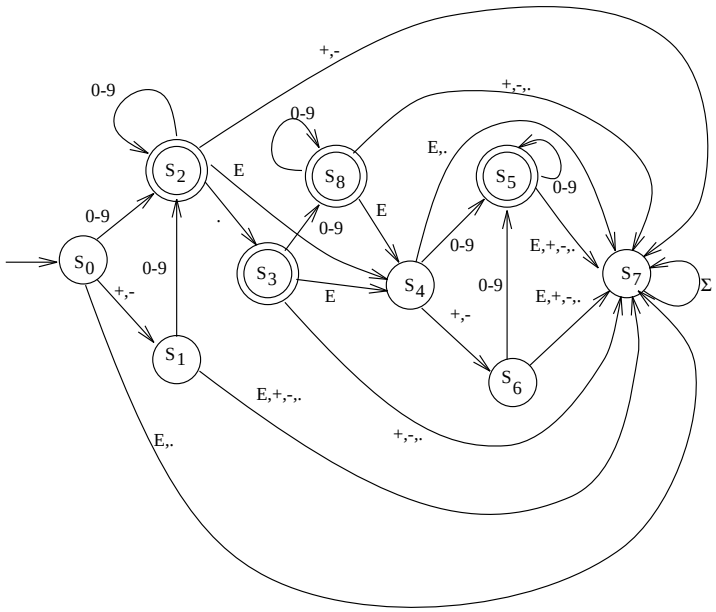


Figura 2.9: AFD para constantes numéricas de ponto flutuante

Definição 2.2 *Dado o AFD* $A = < \Sigma, S, S_0, \delta, F >$*, a Função de Transição Extendida de* A *é definida em* $\overline{\delta} : S \times \Sigma^* \to S$ *tal que:*

$$\begin{array}{ll} (\forall s \in S) & \overline{\delta}(s, \lambda) = s \\ (\forall s \in S)(\forall a \in \Sigma) & \overline{\delta}(s, a) = \delta(s, a) \\ (\forall s \in S)(\forall x \in \Sigma^*)(\forall a \in \Sigma) & \overline{\delta}(s, ax) = \overline{\delta}(\delta(s, a), x) \end{array}$$

Intuitivamente, $\overline{\delta}$ é a aplicação, repetidas vezes, da função δ sobre cada letra da palavra x. Com o auxílio de $\overline{\delta}$, podemos fazer as primeiras definições, vistas informalmente anteriormente:

Definição 2.3 *Dado o AFD* $A = < \Sigma, S, S_0, \delta, F >$ *e a cadeia* $x \in \Sigma^*$*, diz que:*

- A **aceita** x *sse* $\overline{\delta}(S_0, x) \in F$
- A **rejeita** x *sse* $\overline{\delta}(S_0, x) \notin F$

A linguagem $L(A)$, definida pela AFD A é o conjunto de todas as palavras reconhecidas por esse autômato, ou seja:

Definição 2.4 *Dado o AFD* $A = < \Sigma, S, S_0, \delta, F >$, *define-se a linguagem* $L(A)$, *reconhecida por esse autômato como sendo*

$$L(A) = \{x \in \Sigma^* \mid \overline{\delta}(S_0, x) \in F\}$$

2.4 Exercícios

2.1 *Dado o alfabeto* $\Sigma = \{a, b\}$, *construa um Autômato Finito Determinístico para cada uma das seguintes linguagens:*

a. $\{b(ab)^n b | n \geq 0\}$

b. $\{ba^n ba | n \geq 0\}$

c. $\{a^m b^n | m, n \geq 0 \wedge m + n \text{ é par}\}$

d. $\{ab^m ba(ab)^n | m, n \geq 0\}$

2.2 *Seja* $\Sigma = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$. *Construa AFD's que reconheçam as seguintes linguagens:*

a. $\{x \in \Sigma^* \mid$ *o número representado por* x *é divisível por 7*$\}$

b. $\{x \in \Sigma^* \mid$ *o número representado por* x *é divisível por 3*$\}$

c. $\{x \in \Sigma^* \mid$ *o número representado por* x *é divisível por 5*$\}$

2.5 Implementação

Antes da implementação dos AFD's propriamente ditos, vamos ver um módulo auxiliar, chamado "SET32". Ele é utilizado para representar pequenos conjuntos de números inteiro. Cada conjunto é representado através de um **unsigned int**. Cada bit desse **int** representa um número entre 0 e 31. Assim, se o bit 0 está ligado, isso significa que o número 0 está presente no conjunto, se o bit 1 está ligado, o número 1 está presente, e assim por diante. Por exemplo, o conjunto $\{2, 5, 7, 20, 30\}$ seria representado pelo valor 0x401000A4. Abaixo estão os arquivos set32.h e set32.c que implemantam esse tipo de dados.

```
/*********************************************************
                          set32.h

Interface para o tipo de dado SET32, que representa um
conjunto de numeros inteiro no intervalo 0-31

Data: 23/3/98
Autor: Marcio Delamaro
```

```
*********************************************************/

typedef unsigned long int SET32;

SET32 SET32Set(SET32, int , int );
int SET32Get(SET32, int );
SET32 SET32Union(SET32, SET32) ;
SET32 SET32Inter(SET32, SET32);
SET32 SET32Next(SET32, int);


/*********************************************************
                          set32.c

Implementacao do tipo de dado SET32, que representa um
conjunto de numeros inteiro no intervalo 0-31

Data: 23/3/98
Autor: Marcio Delamaro
*********************************************************/
#include       "set32.h"


SET32    SET32Set(SET32 x, int posic, int value)
{ /* inclue ou retira um determinado valor do conjunto */
   value = (value != 0) << posic;
   x &= ~(1 << posic);
   x |= value;
   return x;
}

int    SET32Get(SET32 x, int posic)
{      /* verifica se um determinado valor esta´ no conjunto */
SET32 i;
   i = x & (1 << posic);
   return i != 0;
}

SET32    SET32Union(SET32 x, SET32 y)
{      /* calcula a uniao de dois conjuntos */
    return x | y;
}

SET32    SET32Inter(SET32 x, SET32 y)
{     /* calcula a interseccao de dois conjuntos */
    return x & y;
}

SET32    SET32Next(SET32 x, int i)
{     /* devolve o proximo elemento, a partir do valor i */
int j;

   x >>= i;
   for (j = i; (x & 1) == 0 && j <= 32; j++)
        x >>= 1;
   if (  j <= 32 )
       return j;
   return -1;       /* nenhum elemento apos i */
}
```

O arquivo afd.h abaixo mostra a definicao do tipo de dado **AUTOFIN**, que armazena um Autômato Finito Determinístico. Na estrutura AUTOFIN há um

apontador para o alfabeto de entrada do AFD, criado junto com o AFD, pela função `AFCria`. Há também uma tabela com os estados definidos e uma tabela com as transições definidas. Inicialmente, deve-se chamar a função `AFCria` para criar um AFD sem nenhum estado ou transição, passando apenas um string que representa o alfabeto correspondente. Essa função retorna o endereço do AFD criado. Esse endereço deve ser usado nas chamadas às demais funções de AUTOFIN. A função `AFDestroi` libera a estrutura alocada para um AFD.

Cada estado deve ser adicionado através de uma chamada a `AFAddEstado`, que recebe como parametro, além do endereço do AFD, o nome do estado e uma indicação se ele é inicial, final, ambos ou nenhum. A função `AFAddTransi` adiciona uma transição. Recebe como parâmetros o nome dos estados origem e destino, e a letra do alfabeto correspondente à transição. Os dois estados devem ter sido previamente adicionados ao AFD. A tabela de transição contém uma linha para cada estado. Cada coluna representa qual é o estado a ser atingido, para uma das letras do alfabeto. Por exemplo, com um AFD com 3 estados e um alfabeto de 4 letras, teríamos:

	letra 1	letra 2	letra 3	letra 4
estado 1				
estado 2				
estado 3				

Se a é uma varivél que contém o endereço de um AFD, então a transição correspondente ao estado i com a letra j é encontrada em

```
a->transition[i * a->alfalen + j]
```

As funções `AFDelta` e `AFDeltaBarra` correspondem às funções de transição de estados e função de transição de estados extendida, respectivamente. A função `AFReconhece` retorna verdadeiro (não zero) ou falso (zero) se uma determinada palavra pertence ou não à linguagem definida pelo AFD. A variável *caminho* é utilizada para armazenar os estados percorridos ao se analisar uma palavra.

```
/*********************************************************
                          afd.h

Interface para o tipo de dado AUTOFIN, que representa um
um automato finito deterministico

Data: 23/3/98
Autor: Marcio Delamaro
*********************************************************/

#include    "set32.h"
#include    "alfabeto.h"


#define    MAXSTATE    32   /* numero maximo de estados */

#define    LABELSIZE    50  /* tamanho maximo do nome de um estado */
```

```
#define    INITIAL    1     /* indicacao de que um estado eh inicial */

#define    FINAL      2     /* indicacao de que um estado eh final */

typedef struct STATE {      /* armazena um estado */
    char    label[LABELSIZE+1];
    int    type;
    } STATE;

typedef struct AUTOFIN {     /* armazena um AFD */
    ALFABETO *alfa;      /* apontador para o alfabeto */
    int       nstates;   /* numero de estados */
    int       alfalen;   /* cardinalidade do alfabeto */
    STATE     states[MAXSTATE]; /* tabela de estados */
    int       *transitions;     /* tabela de transicoes */
    int       initial;    /* numero do estado inicial */
    SET32     final;      /* conjunto de estados finais */
    } AUTOFIN;

AUTOFIN    *AFCria(char *sigma);
int         AFAddEstado(AUTOFIN *a, char *label, int final);
int         AFAddTransi(AUTOFIN *a, char *state1, char *state2, char *letra);
int         AFDelta(AUTOFIN *a, char *state, char *letra);
int         AFDeltaBarra(AUTOFIN *a, char *state, char *palavra, char *caminho);
int         AFReconhece(AUTOFIN *a, char *palavra, char *caminho);
void        AFDestroi(AUTOFIN *a);



/*********************************************************
                          afd.c

Implementacao do tipo de dado AUTOFIN, que representa um
um automato finito deterministico

Data: 23/3/98
Autor: Marcio Delamaro
*********************************************************/

#include    <stdio.h>
#include    <string.h>
#include    <ctype.h>
#include    <malloc.h>
#include    "afd.h"

#define        skip_blank(x)    while(isspace(*x))x++;


int     pegaestado(AUTOFIN *, char *);


/*-------------------------------------------------------
           AFCria
Cria a estrututa re um automato finito, vazio sem nenhum
estado ou transicao

Parametros:
   char *sigma:    alfabeto que compoe as letras do automato

Retorna:
   AUTOFIN * :        endereco do automato criado
-------------------------------------------------------------*/
```

```
AUTOFIN    *AFCria(char *sigma)
{
AUTOFIN *p;

   p = (AUTOFIN *) malloc(sizeof(AUTOFIN));
   if (p == NULL)
        return NULL;
   memset(p, 0, sizeof(*p));
   p->alfa = ALFACria(sigma);
   if (p->alfa == NULL)
   {
       free(p);
       return NULL;
   }
   p->alfalen = ALFACard(p->alfa);
   p->initial = -1;
   return p;
}

/*-------------------------------------------------------
            AFAddEstado
Adiciona um stado "vazio" ao AF, isto eh, um estado sem nenhuma
transicao

Parametros:
    char *label      Nome do estado
    int final        Flag indicando se estado eh fina/inicial

Retorno:
    < 0 se houve alguma falha
-------------------------------------------------------------*/
int    AFAddEstado(AUTOFIN *a, char *label, int final)
{
STATE p;

   if ( a->nstates >= MAXSTATE)
    return -1;

   p.type = final;
   strncpy(p.label, label, LABELSIZE);
   if (final & INITIAL)
     a->initial = a->nstates;
   if (final & FINAL)
     a->final = SET32Set(a->final, a->nstates, 1);
   a->states[a->nstates++] = p;
   if (a->transitions != NULL)    /* resize a->transitions se jah existe */
   {
       a->transitions = realloc(a->transitions, a->alfalen * a->nstates *
                         sizeof(*a->transitions));
       memset(&a->transitions[(a->nstates-1)*a->alfalen], -1,
                     a->alfalen * sizeof(*a->transitions));
   }
   return 0;
}

/*-------------------------------------------------------
                AFAddTransi
Adiciona um atransicao a um estado

Parametros:
    AUTOFIN a:      O automato
    char *state1    O nome do estado origem
```

```
    char *state2    O nome do estado destino
    char *letra     O simbolo do alfabeto correspondente a transicao
-------------------------------------------------------------*/

int    AFAddTransi(AUTOFIN *a, char *state1, char *state2,
                   char *letra)
{
int i, j, k;

   if ( a->transitions == NULL )
   {
       a->transitions = malloc(k = a->alfalen * a->nstates *
                         sizeof(*a->transitions));
       if (a->transitions == NULL)
          return -1;
       memset(a->transitions, -1, k);
   }
   i = pegaestado(a, state1); /* descobre o numero do estado origem */
   k = pegaestado(a, state2); /* descobre o numero do estado destino */
   j = ALFAOrdem(a->alfa, letra); /* descobre o numero da letra do alfabeto */

   if (i < 0 || k < 0 || j < 0)
       return -1;

   if (a->transitions[i*a->alfalen+j] != -1)
      return -1;
   a->transitions[i*a->alfalen+j] = k; /* inclui na tabela de transicoes */
   return 0;
}

/*-----------------------------------------------------------
            AFDelta
Essa funcao retorna o valor da funcao de transicao para um
estado e uma letra.

Parametros:
AUTOFIN *a         automato finito
char    *state     estado
char    *letra     letra para dalcular delta(estado,letra)

Retorna:
int                numero do proximo estado
                   -1 se automato invalido
                   -2 parametro invalido
-------------------------------------------------------------*/
int    AFDelta(AUTOFIN *a, char *state, char *letra)
{
int j, k;

   k = pegaestado(a, state);
   j = ALFAOrdem(a->alfa, letra);
   if (k < 0 || j < 0)
       return -2;
   return a->transitions[k*a->alfalen+j];
}


/*-----------------------------------------------------------
            AFDeltaBarra
Essa funcao retorna o valor da funcao de transicao Extendida
para um estado e uma cadeia.
```

```
Parametros:
AUTOFIN *a          automato finito
char    *state      estado
char    *letra      letra para dalcular delta(estado,letra)
char    *caminho    buffer para colocar proximo estado

Retorna:
int                 numero do proximo estado
                    -1 se automato invalido
                    -2 parametro invalido
------------------------------------------------------------*/
int    AFDeltaBarra(AUTOFIN *a, char *state, char *palavra, char *caminho)
{
int i;
static char buf[100];
int k;

   if (caminho != NULL)
   {
       strcat(caminho, " ");
       strcat(caminho,state);
   }
    skip_blank(palavra);
    i = sscanf(palavra, "%s", &buf); /* pega primeira letra */
    if (i != 1)     /* terminaram as letras */
        return pegaestado(a,state); /* retorna estado corrente */
   if ((k = AFDelta(a, state, buf)) < 0)
        return k;
   palavra += strlen(buf);
   return AFDeltaBarra(a, a->states[k].label, palavra, caminho);
}



/*------------------------------------------------------------
           AFReconhece
Essa funcao tenta reconhecer um string

Parametros:
AUTOFIN *a:         automato finito
char    *palavra    string a ser analisado (letras separadas por brancos)
char    caminho     sequencia de estados percorridos

Retorna:
int                 1 reconheceu
                    0 nao reconheceu
                    -1 automato invalido
                    -2 parametro invalido
------------------------------------------------------------*/
int    AFReconhece(AUTOFIN *a, char *palavra, char *caminho)
{
int    next;

   *caminho = '\0';
   if (a->initial < 0 )
      return -1;
   next = AFDeltaBarra(a, a->states[a->initial].label, palavra, caminho);
   if (next < 0)
      return next;
   return SET32Get(a->final, next);
}
```

```
/*-------------------------------------------------------
                AFDestroi
Libera memoria de um AF
-------------------------------------------------------------*/
void    AFDestroi(AUTOFIN *a)
{
   free(a->transitions);
   ALFADestroi(a->alfa);
}


/*-------------------------------------------------------
Dado o nome de um estado, retorna seu numero
-------------------------------------------------------------*/
int pegaestado(AUTOFIN *a, char *s)
{
int i;

   for (i = 0; i < a->nstates; i++)
    if (strcmp(a->states[i].label, s) == 0)
       return i;
   return -1;
}
```

Finalmante, o arquivo `main.c` abaixo é utilizado para testar a implementação de nosso AFD. Ele lê a descrição de um arquivo como:

```
a b c
estado S0 inicial
estado S1 final
transicao S0 S1 a b c
transicao S1 S1 b c
transicao S1 S0 a
```

e cria o AFD correspondente. Depois disso, passa a ler cadeias fornecidas pelo usuário e as analisa, indicando se pertencem ou não ao AFD e quais os estados visitados ao processar aquela cadeia. Para criar um programa executável **afd** deve-se utilizar:

```
gcc afd.c alfabeto.c main.c set32.c -o afd
```

```
/*********************************************************
                          main.c

Teste para afd.

Data: 23/3/98
Autor: Marcio Delamaro
*********************************************************/
#include    <stdio.h>
#include    <string.h>
#include    <ctype.h>


#include    "afd.h"
```

```
char    buf[1024], buf2[1024];
char    label[50], label2[50], final[30], letra[30], *apbuf;

int    init;

void    msg(char *);
int    getsimb(char *);

main(int argc, char *argv[])
{
FILE *fp;
AUTOFIN *a, *b;
int    i,j,k, n;
int    Menosp, Menosnolambda, Menosdfa, debug;

   if (argc != 2)
   {
        r0:
        msg("Uso: af <filename>");
        return -1;
   }


   fp = fopen(argv[argc-1], "r");
   if (fp == NULL)
   {
        msg("Erro ao abrir arquivo com AFD");
        return -1;
   }
   if (fgets(buf, 1024, fp) == NULL)
   {
        r1:
        msg("Erro ao ler arquivo com AFD");
        return -1;
   }
   i = strlen(buf);
   if (buf[i-1] == '\n')
        buf[i-1] = '\0';
   a = AFCria(buf);

   if (a == NULL)
   {
        msg("Erro ao criar AFD");
        return -1;
   }

   for (i = fscanf(fp, "%s", buf); i == 1; i = fscanf(fp, "%s", buf))
   {
        if (strcasecmp(buf, "estado") == 0) /* leu uma linha de estado */
        {
           if (fgets(buf, 1024, fp) == NULL)
              goto r1;
           apbuf = buf;

           j = getsimb(label); /* separa o nome do estado */
           if ( j != 1)
           {
                r2:
                msg("Invalid description file");
                return -1;
           }
```

```
        k = getsimb(final); /* tenta pegar final ou inicial */
        init = 0;
        if ( k == 1)
        {
            if (strcasecmp(final, "final") == 0)
              init += FINAL;
             if (strcasecmp(final, "inicial") == 0)
              init += INITIAL;
        }
        k = getsimb(final); /* tenta pegar final ou inicial */
        if ( k == 1)
        {
            if (strcasecmp(final, "final") == 0)
              init += FINAL;
             if (strcasecmp(final, "inicial") == 0)
              init += INITIAL;
        }
        if (AFAddEstado(a, label, init) < 0) /* adiciona o estado */
        {
             msg("Erro ao adicionar estado");
             return -1;
        }
     }
     else
    if (strcasecmp(buf, "transicao") == 0) /*achou uma linha de transicao */
    {
        if (fgets(buf, 1024, fp) == NULL)
          goto r1;
        apbuf = buf;

        j = getsimb(label); /* separa estado origem */
        if ( j != 1)
          goto r2;

        j = getsimb(label2); /* separa estado destino */
        if ( j != 1)
           goto r2;

        /* adicina todas as transicoes para esses dois estados */
        for (j = getsimb(letra); j == 1; j = getsimb(letra))
           if (AFAddTransi(a, label, label2, letra) < 0)
           {
                  msg("Erro ao adicionar transicao");
                  return -1;
           }
     }
     else goto r2;
}

/* le string e analisa */
for ( printf("\n\nEntre novo string: "), apbuf = gets(buf); apbuf != NULL;
   printf("\n\nEntre novo string: "), apbuf = gets(buf))
{
    j = strlen(buf);
    if (buf[j-1] == '\n')
         buf[j-1] = '\0';
    j = AFReconhece(a, buf, buf2);
    if (j == -2)
         msg("String invalido");
    else
    if (j == -1)
        msg("Automato invalido");
```

```
        else
        if (j == 0)
            msg("String nao pertence a linguagem.");
        else
            msg("String pertence a linguagem.");
        msg("Caminho percorrido:");
        msg(buf2);
    }
    AFDestroi(a);
    return 0;
}


void msg(char *x)
{
    printf("\n%s\n", x);
}


int getsimb(char *x)
{
int i;

    while (isspace(*apbuf) && *apbuf != '\0')
     apbuf++;
    i = sscanf(apbuf, "%s", x);
    if (i == 1)
     apbuf += strlen(x);
    return i;
}
```

Capítulo 3

Minimização de AFD's

É fácil perceber que para uma dada linguagem L, existem mais do que um AFD para representar L. Em geral, ao construir um AFD A, tal que L(A) = L, buscamos um autômato que seja o mais simples possível, ou seja, que possua o menor número possível de estados. Esses AFD's que chamamos de **minimais**, são o centro desse capítulo. Estudaremos inicialmente algumas definições pertinentes ao assunto e em seguida veremos como obter AFD's minimais.

3.1 Equivalência entre AFD's

Inicialmente precisamos definir o conceito de AFD's iguais, ou que realizam a mesma tarefa:

Definição 3.1 *Dados dois AFD's A e B, dizemos que A é* **equivalente** *a B, denotado por $A \equiv B$, sse $L(A) = L(B)$.*

Isso significa que dois AFD's são equivalentes quando a linguagem que eles reconhecem é a mesma. Por exemplo, dados os AFD's $A = < \Sigma, \{S_1, S_2, S_3, S_4\}, S_1, \delta_A, \{S_3\} >$ e $B = < \Sigma, \{R_0, R_1, R_2, R_3\}, R_1, \delta_B, \{R_3\} >$ cujas funções de transição são dadas pelos diagramas da Figura 3.1, temos que $L(A) = L(B)$ e portanto $A \equiv B$.

De certa forma, podemos dizer que B é mais simples do que A pois possui um número inferior de estados. Assim, estaremos, em geral, empenhados em construir AFD's que sejam os menores possíveis, ou em outras palavras, que sejam minimais, conforme a definição abaixo.

Definição 3.2 *Um AFD $A = < \Sigma, S_A, S_{0A}, \delta_A, F_A >$ é dito minimal se para qualquer AFD $B = < \Sigma, S_B, S_{0B}, \delta_B, F_B >$ tal que $A \equiv B$, temos que $|S_A| \leq |S_B|$.*

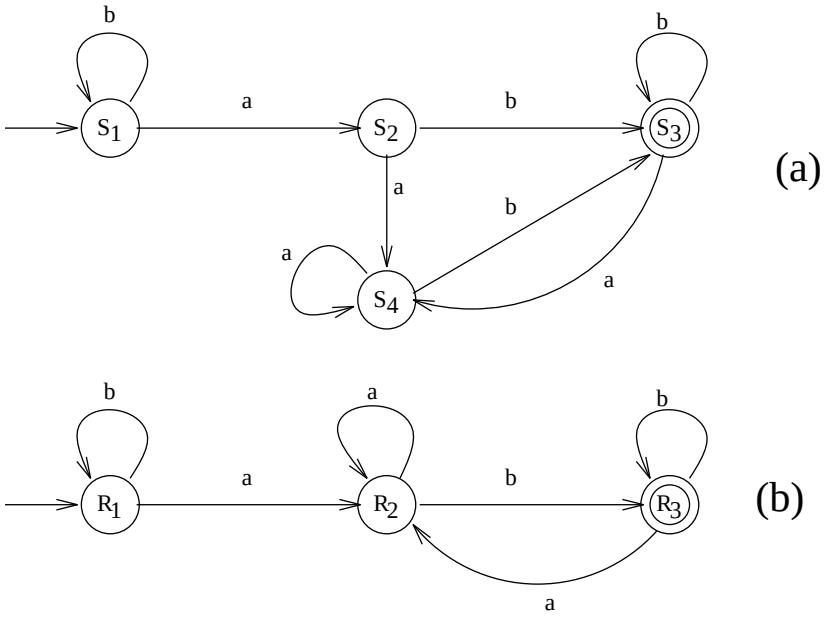


Figura 3.1: Exemplo de dois AFD's equivalentes

Dada a linguagem L, veremos que existe apenas um AFD A minimal que reconhece L. Melhor dizendo, existe uma classe de AFD's minimais, na qual cada AFD possui a mesma "forma" e que apenas diferem nos nomes dados aos estados. Para definir mais precisamente qual é a relação entre os autômatos dessa classe, definimos **homomorfismos** e **isomorfismos** entre autômatos.

Definição 3.3 *Dados dois AFD's* $A = < \Sigma, S_A, S_{0A}, \delta_A, F_A >$ *e* $B = < \Sigma, S_B, S_{0B}, \delta_B, F_B >$, *diz-se que existe um homomorfismo* μ *de* A *para* B *sse* μ *é uma função definida* $\mu : S_A \rightarrow S_B$ *tal que:*

$$\begin{array}{ll} & \mu(S_{0A}) = S_{0B} \\ (\forall s \in S_A) & s \in F_A \Leftrightarrow \mu(s) \in F_B \\ (\forall s \in S_A)(\forall a \in \Sigma) & \mu(\delta_A(s,a)) = \delta_B(\mu(s), a) \end{array}$$

Essa definição indica que cada estado s do autômato A possui um estado s' correspondente no autômato B, que se comporta de maneira idêntica a s. Se s é um estado inicial ou final, também o é o estado s'. Além disso, para cada a do alfabeto Σ, $\delta(s', a)$ é o correspondente em B do estado $\delta(s, a)$ do autômato A. Por exemplo, para os autômatos representados pelos diagramas de transição das Figuras 3.2a e 3.2b, existe um homomorfismo $\mu : S_A \rightarrow S_B$, de A para B, tal que $\mu(S_0) = Q_1$ e $\mu(S_1) = Q_0$.

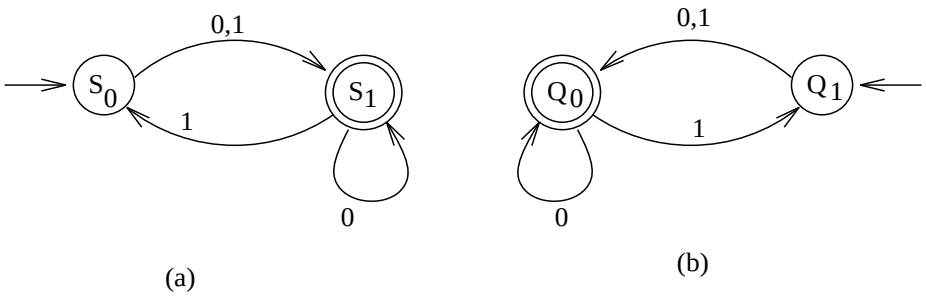


Figura 3.2: Exemplo de homomorfismo de $A = < \Sigma, \{S_0, S_1\}, S_0, \delta_A, \{S_1\} >$ para $B = < \Sigma, \{Q_0, Q_1\}, Q_1, \delta_B, \{Q_0\} >$

Teorema 3.1 *Dados os AFD's A e B e o homomorfismo $\mu : S_A \to S_B$, então $A \equiv B$.*

Prova: Supondo $A = < \Sigma, S_A, S_{0A}, \delta_A, F_A >$, $B = < \Sigma, S_B, S_{0B}, \delta_B, F_B >$ e o homomorfismo $\mu : S_A \to S_B$, vamos inicialmente mostrar que

$$(\forall s \in S_A)(\forall x \in \Sigma^*) \quad \mu(\overline{\delta}_A(s, x)) = \overline{\delta}_B(\mu(s), x) \tag{3.1}$$

Vamos utilizar indução sobre o tamanho de x

Base: $x = \lambda$

$$\mu(\overline{\delta}_A(s, \lambda)) = \mu(s) = \overline{\delta}_B(\mu(s), \lambda)$$

Passo de indução: Supondo que nossa hipótese 3.1 vale para qualquer string x de tamanho $n \geq 1$, vamos mostrar que vale também para qualquer string de tamanho $n + 1$
Sejam $a \in \Sigma$ e $y = a.x$, então

$$\begin{aligned}
\mu(\overline{\delta}_A(s, ax)) = & \quad \text{(pela definição de } \overline{\delta}) \\
\mu(\overline{\delta}_A(\delta_A(s, a), x)) = & \quad \text{(pela hipótese da indução)} \\
\overline{\delta}_B(\mu(\delta_A(s, a)), x) = & \quad \text{(pela definição de homomorfismo)} \\
\overline{\delta}_B(\delta_B(\mu(s), a), x) = & \quad \text{(pela definição de } \overline{\delta}) \\
\overline{\delta}_B(\mu(s), ax) & \quad \text{(C.Q.D)}
\end{aligned}$$

Usando o fato 3.1, vamos demonstrar agora que $x \in L(A) \Leftrightarrow x \in L(B)$, o que demonstra que $A \equiv B$.

$$\begin{aligned}
x \in L(A) \Leftrightarrow & \quad \text{(pela definição de aceitação)} \\
\overline{\delta}_A(S_{0A}, x) \in F_A \Leftrightarrow & \quad \text{(pela definição de homomorfismo)} \\
\mu(\overline{\delta}_A(S_{0A}, x)) \in F_B \Leftrightarrow & \quad \text{(por (3.1))} \\
\overline{\delta}_B(\mu(S_{0A}), x) \in F_B \Leftrightarrow & \quad \text{(pela definição de homomorfismo)} \\
\overline{\delta}_B(S_{0B}, x) \in F_B \Leftrightarrow & \quad \text{(pela definição de aceitação)} \\
x \in L(B) & \quad \text{(C.Q.D)}
\end{aligned}$$

A definição de homomorfismo pode sugerir à primeira vista que, se existe um homomorfismo de A para B, então A e B devem ter a mesma forma. Mas isso nem sempre acontece, como mostram os AFD's das Figuras 3.3a e 3.3b. Nesse

caso existe um homomorfismo $\mu : S_A \rightarrow S_B$, de A para B, tal que $\mu(S_0) = Q_1$, $\mu(S_1) = Q_0$ e $\mu(S_2) = Q_1$.

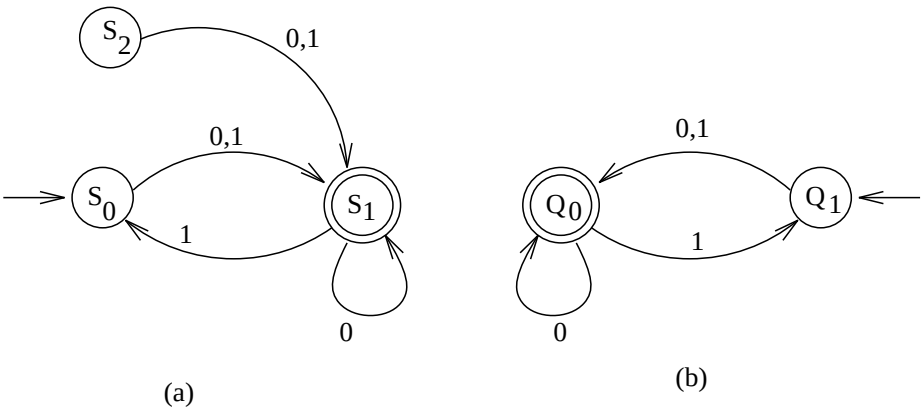


Figura 3.3: Outro exemplo de homomorfismo de $A = < \Sigma, \{S_0, S_1, S_2\}, S_0, \delta_A, \{S_1\} >$ para $B = < \Sigma, \{Q_0, Q_1\}, Q_1, \delta_B, \{Q_0\} >$

Para estabelecer uma relação entre AFD's que tenham a mesma forma e sejam compostos apenas pela troca de nomes dos estados, utilizaremos a seguinte definição:

Definição 3.4 *Dados dois AFD's* $A = < \Sigma, S_A, S_{0A}, \delta_A, F_A >$ *e* $B = < \Sigma, S_B, S_{0B}, \delta_B, F_B >$, *diz-se que existe um isomorfismo* μ *de* A *para* B *sse* μ *é uma função definida* $\mu : S_A \rightarrow S_B$ *tal que:*

- μ *é um homomorfismo*
- μ *é bijetora*

Teorema 3.2 *Dados os AFD's* A *e* B *e o isomorfismo* $\mu : S_A \rightarrow S_B$, *então* $A \equiv B$.

Prova: Pelo teorema 3.1.

É fácil verificar que $\mu^{-1} : S_B \rightarrow S_A$ também é um isomorfismo e podemos dizer que A e B são isomorfos, denotado por $A \cong B$. Além disso, a relação de isomorfismo entre dois AFD's é uma relação de equivalência e por isso pode-se falar numa classe de AFD's isomorfos.

Iremos estudar adiante como transformar qualquer AFD A num autômato minimal equivalente. Vale ressaltar que o AFD obtido é "único" no sentido de que quaisquer AFD's que reconhecem $L(A)$ e que sejam minimais, são isomorfos.

3.2 AFD's não conexos

Para obtermos um AFD minimal é preciso que sejam eliminados todos aqueles estados que são supérfluos. Uma das maneiras pelas quais um estado pode ser supérfluo é mostrado na Figura 3.2a. Naquele caso, o autômato A possui o estado S_2 que não desempenha qualquer papel no funcionamento do autômato. Isso porque S_2 nunca vai se tornar o estado ativo, ou seja, nunca será alcançado a partir do estado inicial.

Definição 3.5 *Dado o AFD* $A = < \Sigma, S, S_0, \delta, F >$*, diz-se que um estado* $R \in S$ *é* **acessível** *ou* **alcançável** *sse para algum string* $x \in \Sigma^*$, $\overline{\delta}(S_0, x) = R$. *Um estado que não seja acessível é chamado* **inacessível**, **inalcançável** *ou* **desconexo**. *Um AFD em que todos os estados são alcançáveis é chamado de* **conexo**.

É fácil verificar que, se quizermos um AFD com o menor número possível de estados, temos que nos livrar dos estados que são inacessíveis. Vamos então definir como, dado um AFD A, achar um AFD A^c que seja conexo e equivalente a A.

Definição 3.6 *Dado um AFD* $A = < \Sigma, S, S_0, \delta, F >$*, temos que* $S = S^c \cup S^d$*, onde* S^c *é o conjunto dos estados alcançáveis e* $S^d = S - S^c$ *é o conjunto dos estados não alcançáveis. Podemos definir um novo AFD* $A^c = < \Sigma, S^c, S_0, \delta^c, F^c >$ *tal que:*

$$F^c = F \cap S^c$$

$$(\forall a \in \Sigma)(\forall s \in S^c) \quad \delta^c(s, a) = \delta(s, a)$$

Assim, esse autômato é obtido descartando-se simplesmente aqueles estados que não são alcançáveis e restringindo a função δ somente àqueles estados que são alcançáveis. Assim, por definição, A^c é um AFD conexo, pois seus estados são todos alcançáveis. Resta mostrar o seguinte teorema:

Teorema 3.3 *A e A^c são equivalentes*

Prova: Primeiramente vamos mostrar por indução que

$$(\forall s \in S^c)(\forall x \in \Sigma^*)\ \overline{\delta}(s, x) = \overline{\delta^c}(s, x). \tag{3.2}$$

Base: $x = \lambda$. Pela definição da função extendida:

$$\overline{\delta}(s, \lambda) = \overline{\delta^c}(s, \lambda) = s$$

Passo de indução: Supondo que 3.2 vale para $y \in \Sigma^*$, tal que $|y| \geq 1$, tomamos $x = a.y$, $a \in \Sigma$. Então temos:

$$\begin{aligned}
\overline{\delta}(s,x) = & \quad \text{(pela escolha de } x\text{)} \\
\overline{\delta}(s,ay) = & \quad \text{(pela definição de } \overline{\delta}\text{)} \\
\overline{\delta}(\delta(s,a),y) = & \quad \text{(pelo teorema 3.4 e pela hipótese da indução)} \\
\overline{\delta^c}(\delta(s,a),y) = & \quad \text{(pela definição de } A^c\text{)} \\
\overline{\delta^c}(\delta^c(s,a),y) = & \quad \text{(pela definição de } \overline{\delta}\text{)} \\
\overline{\delta^c}(s,ay) = & \quad \text{(pela escolha de } x\text{)} \\
\overline{\delta^c}(s,x) & \quad \text{(C.Q.D.)}
\end{aligned}$$

A segunda parte da demostração é provar que $L(A) = L(A^c)$. Tomando $x \in \Sigma^*$, temos que:

$$\begin{aligned}
x \in L(A) \Leftrightarrow & \quad \text{(pela definição de aceitação)} \\
\overline{\delta}(S_0,x) \in F \Leftrightarrow & \quad \text{(pela definição de } A^c\text{)} \\
\overline{\delta}(S_0,x) \in F^c \Leftrightarrow & \quad \text{(por (3.2))} \\
\overline{\delta^c}(S_0,x) \in F^c \Leftrightarrow & \quad \text{(pela definição de aceitação)} \\
x \in L(A^c) & \quad \text{(C.Q.D.)}
\end{aligned}$$

Dessa maneira pudemos verificar como construir um um AFD conexo a partir de um AFD qualquer. O requisito para que isso seja feito é que possamos reconhecer quais são os estados alcançáveis do AFD original. Essa é porém a parte "difícil" do processo. Para calcularmos quais são esses estados, usaremos o seguinte teorema:

Teorema 3.4 *Dado o AFD* $A = < \Sigma, S, S_0, \delta, F >$,

$$(\forall s \in S^c)(\forall a \in \Sigma)\ \delta(s,a) \in S^c$$

Prova: Se s é alcançável, então $\exists x \in \Sigma^*$ tal que:

$$\delta(s,a) = \delta(\overline{\delta}(S_0,x),a) = \overline{\delta}(S_0,x.a)$$

e portanto $\delta(s,a)$ é alcançável.

A idéia para se calcular S^c é utilizar um subconjunto C_i de S^c e ir incrementando esse subconjunto incluindo outros estados que podem ser alcançados a partir dos estados desse subconjunto, até que todo S^c tenha sido calculado. O ponto inicial desse processo é o conjunto unitario $\{S_0\}$, que sabemos é alcançável por definição. O final desse processo é quando nenhum estado pode ser adicionado ao conjunto.

Definição 3.7 *Dado o AFD* $A = < \Sigma, S, S_0, \delta, F >$, *definem-se os* **conjuntos parciais de estados** *de* A, *denotados por* C_i *como sendo:*

$$C_0 = \{S_0\}$$

$$C_{i+1} = C_i \cup \bigcup_{s \in C_i, a \in \Sigma} \{\delta(s, a)\}$$

Tomemos como exemplo o AFD $A = < \{0,1\}, \{A, B, C, D, E, F, G, H, I, J\}, A, \delta, \{B, E, G\} >$ representado na Figura 3.4. Iniciamos com $C_0 = \{A\}$. Vamos calcular C_1 verificando quais são os estados alcançáveis a partir de A processando-se apenas uma letra da entrada, ou seja, queremos achar os estados s_a tais que $\delta(A, a) = s_a$, para todo $a \in \Sigma$. Vemos que $\delta(A, 0) = B$ e $\delta(A, 1) = D$. Portanto

$$C_1 = \{A\} \cup \{B, D\} = \{A, B, D\}$$

Para calcularmos C_2 deveríamos verificar todos os estados alcançáveis a partir de A, B e D. Porém os alcançáveis a partir de A já são conhecidos e já fazem parte de C_1 e por isso só precisamos nos preocupar com os estados recém colocados no conjunto, ou seja, B e D. Teremos então:

$$C_2 = \{A, B, D, C, E\}$$

$$C_3 = \{A, B, D, C, E, F\}$$

Para calcularmos C_4 devemos incluir no conjunto os sucessores de F que são $\delta(F, 0) = C$ e $\delta(F, 1) = E$, ambos já presentes em C_3. Temos portanto que $C_3 = C_4$. Ao calcularmos C_5 notamos que, como nenhum novo estado foi introduzido, todos os alcançáveis a partir dos estados de C_4 já estão no conjunto. O mesmo acontece para $C_6, C_7, ...$ Isso indica que obtivemos um conjunto que já possui todos os estados alcançáveis. Ou seja,

$$C_3 = C_4 = ... = S^c$$

Outro ponto importante a notar é que $|S_A| = n$, então teremos que calcular, no máximo, C_{n-1} para chegarmos até S^c, uma vez que $|S^c| \leq |S_A|$.

3.3 AFD com Estados Equivalentes

O segundo motivo pelo qual um AFD pode conter estados a mais do que o necessário é quando dois estados desempenham o mesmo papel no reconhecimento de qualquer string. É o que acontece no exemplo da Figura 3.1a com os estados S_2 e S_4. Uma vez no estado S_2 ou no estado S_4 não há como diferenciá-los externamente, ou seja, tudo que for reconhecido a partir do estado S_2 também o será a partir de S_4 e tudo que não for reconhecido a partir de S_2 também não o será a partir de S_4 e vice-versa. Mais precisamente, temos a seguinte definição:

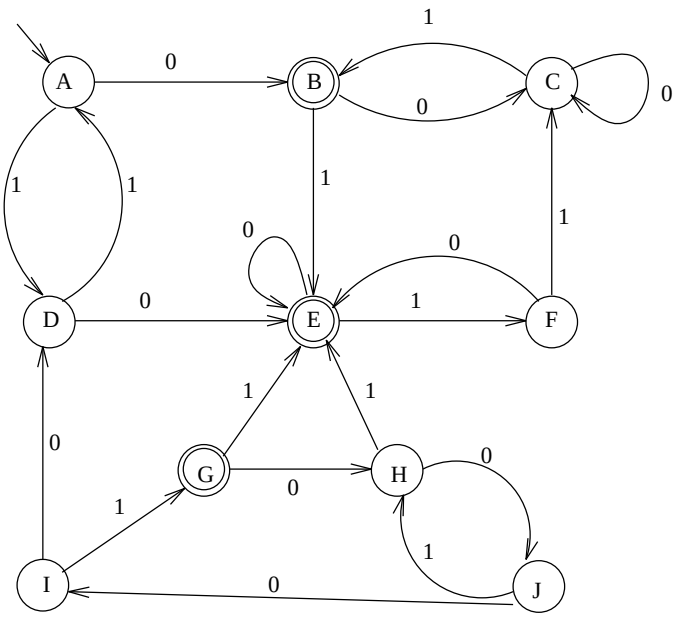


Figura 3.4: AFD exemplo para cálculo de S^c

Definição 3.8 *Dado o AFD* $A = < \Sigma, S, S_0, \delta, F >$*, diz-se que dois estados* s *e* t *são equivalentes, denotando-se* $s \equiv t$ *sse*

$$(\forall x \in \Sigma^*) \quad \overline{\delta}(s, x) \in F \Leftrightarrow \overline{\delta}(t, x) \in F$$

Esta relação $\equiv$ é uma **relação de equivalência** entre os estados de S, ou seja, ela é:

- reflexiva $(\forall s \in S \quad s \equiv s)$
- simétrica $(\forall s, t \in S \quad s \equiv t \Leftrightarrow t \equiv s)$
- transitiva $(\forall s, t, r \in S \quad s \equiv t \wedge t \equiv r \Rightarrow s \equiv r)$

Assim, podemos falar em **classes de equivalência** entre estados de S.

Definição 3.9 *Dado o AFD* $A = < \Sigma, S, S_0, \delta, F >$ *e o estado* $s \in S$*, define-se a classe de equivalência* $[s]$ *de* s *como sendo*

$$[s] = \{t \in S | t \equiv s\}$$

Ou seja, uma classe de estados equivalentes é formada por aqueles estados que não podem ser externamente distinguidos. Note-se que qualquer $s \in S$ faz parte de uma classe de equivalência não vazia pois sempre $s \in [s]$. Além disso, como $\equiv$ é uma relação de equivalência, as classes que ela determina são disjuntas, formando uma **partição** de S, ou seja, cada $s \in S$ pertence a uma única classe de equivalência.

Vejamos um outro exemplo, da Figura 3.5. O AFD $A = < \{a, b, c, d\}, \{S_0, S_1, S_2, S_3, S_4, S_5\}, S_0, \delta, \{S_2, S_5\} >$ reconhece a linguagem $\{x \in \Sigma^* | aa$ é uma subpalavra de $x \vee bca$ é uma subpalavra de $x\}$. Podemos identificar as seguintes classes de equivalência nesse AFD:

$$[S_0] = \{S_0\}$$
$$[S_1] = [S_4] = \{S_1, S_4\}$$
$$[S_2] = [S_5] = \{S_2, S_5\}$$
$$[S_3] = \{S_3\}$$

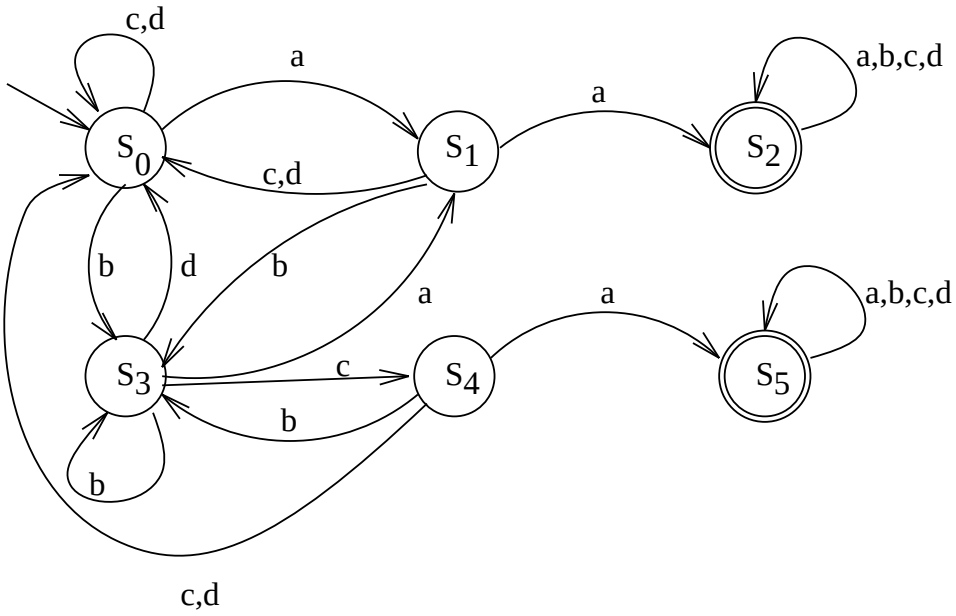


Figura 3.5: Exemplo de estados equivalentes

O que desejamos é eliminar dos nossos AFD estados que desempenhem exatamente as mesmas funções, ou seja, estados equivalentes. Queremos um AFD conforme a definição abaixo:

Definição 3.10 *Um AFD* $A = < \Sigma, S, S_0, \delta, F >$ *é dito* **reduzido** *sse* $(\forall s \in S)\ |[s]| = 1$.

Num AFD reduzido todas as classes de equivalência são unitárias o que implica que $s \equiv t$ sse $s = t$, ou seja, não existem dois estados distintos que sejam

equivalentes. Assim como fizemos na seção anterior, vamos estabelecer como transformar um AFD A qualquer num AFD equivalente e que seja reduzido.

Definição 3.11 *Dado o AFD* $A = < \Sigma, S, S_0, \delta, F >$, *define-se* $A^r = < \Sigma, S^r, S_0^r, \delta^r, F^r >$ *tal que*

$$S^r = \{[s] | s \in S\}$$

$$S_0^r = [S_0]$$

$$F^r = \{[s] | s \in F\}$$

$$(\forall a \in \Sigma)(\forall s \in S) \quad \delta^r([s], a) = [\delta(s, a)]$$

Nesse AFD A^r, cada classe de equivalência de A foi transformada num único estado. O estado inicial S_0^r corresponde à classe de equivalência a que pertence S_0. Os estados finais correspondem às classes a que pertencem os estados finais de A. E a função de transição δ^r no ponto $< [s], a >$ leva na classe de equivalência a que pertence o estado $\delta(s, a)$ do AFD A.

Para o exemplo da Figura 3.5, definiriamos o autômato $A^r = < \{a, b, c, d\}, S^r, S_0^r, \delta^r, F^r >$ onde

$$S^r = \{[S_0], [S_1], [S_2], [S_3]\}$$

$$S_0^r = [S_0]$$

$$F^r = \{[S_2]\}$$

e δ^r é dada pela tabela abaixo e pelo diagrama da Figura 3.6.

δ^r	a	b	c	d
$[S_0]$	$[S_1]$	$[S_3]$	$[S_0]$	$[S_0]$
$[S_1]$	$[S_2]$	$[S_3]$	$[S_0]$	$[S_0]$
$[S_2]$	$[S_2]$	$[S_2]$	$[S_2]$	$[S_2]$
$[S_3]$	$[S_1]$	$[S_3]$	$[S_1]$	$[S_0]$

Teorema 3.5 *Dado o AFD* $A = < \Sigma, S, S_0, \delta, F >$, $A^r \equiv A$.

Prova: Para provarmos o teorema vamos inicialmente mostrar que

$$(\forall s \in S)(\forall x \in \Sigma^*) \ [\overline{\delta}(s, x)] = \overline{\delta^r}([s], x) \tag{3.3}$$

Usando indução no tamanho de x:

Base: $|x| = 0$

$$[\overline{\delta}(s, \lambda)] = [s] = \overline{\delta^r}([s], \lambda)$$

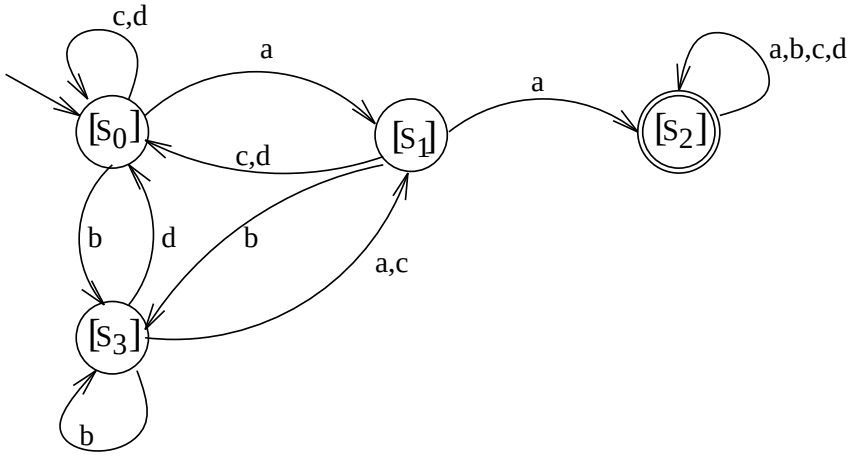


Figura 3.6: Exemplo de AFD reduzido

Supondo que a hipótese 3.3 vale para y, vamos tomar $x = a.y$. Temos:

$$\begin{aligned}
[\overline{\delta}(s,x)] = &\quad \text{(pela escolha de } x\text{)}\\
[\overline{\delta}(s,ay)] = &\quad \text{(dela definição de } \overline{\delta}\text{)}\\
[\overline{\delta}(\delta(s,a),y)] = &\quad \text{(pela hipótese (3.3))}\\
\overline{\delta^r}([\delta(s,a)],y) = &\quad \text{(pela definição de } A^r\text{)}\\
\overline{\delta^r}(\delta^r([s],a),y) = &\quad \text{(pela definição de } \overline{\delta}\text{)}\\
\overline{\delta^r}([s],ay) = &\quad \text{(pela escolha de } x\text{)}\\
\overline{\delta^r}([s],x) &\quad \text{(C.Q.D)}
\end{aligned}$$

Usando (3.3) vamos agora mostrar que $A^r \equiv A$, mostrando que qualquer $x \in L(A) \Leftrightarrow x \in L(A^r)$.

$$\begin{aligned}
x \in L(A) \Leftrightarrow &\quad \text{(pela definição de aceitação)}\\
\overline{\delta}(S_0,x) \in F \Leftrightarrow &\quad \text{(pela definição de } F^r\text{)}\\
[\overline{\delta}(S_0,x)] \in F^r \Leftrightarrow &\quad \text{(por (3.3))}\\
\overline{\delta^r}([S_0],x) \in F^r \Leftrightarrow &\quad \text{(pela definição de } A^r\text{)}\\
\overline{\delta^r}(S_0^r,x) \in F^r \Leftrightarrow &\quad \text{(pela definição de aceitação)}\\
x \in L(A^r) &\quad \text{(C.Q.D)}
\end{aligned}$$

Teorema 3.6 *Dado a AFD* $A = <\Sigma, S, S_0, \delta, F>$, A^r *é reduzido.*

Prova: Temos que mostrar que

$$(\forall s,t \in S^r) \quad S \equiv t \Leftrightarrow s = t$$

Vamos tomar $s = [s']$ e $t = [t']$, onde $s', t' \in S$.

$$
\begin{aligned}
s \equiv t \Leftrightarrow & \quad \text{(pela definição de } s \text{ e } t\text{)} \\
[s'] \equiv [t'] \Leftrightarrow & \quad \text{(pela definição de } \equiv\text{)} \\
(\forall x \in \Sigma^*)\ (\overline{\delta^r}([s'], x) \in F^r \Leftrightarrow \overline{\delta^r}([t'], x) \in F^r) \Leftrightarrow & \quad \text{(por (3.3))} \\
(\forall x \in \Sigma^*)\ ([\overline{\delta}(s', x)] \in F^r \Leftrightarrow [\overline{\delta}(t', x)] \in F^r) \Leftrightarrow & \quad \text{(pela definição de } F^r\text{)} \\
(\forall x \in \Sigma^*)\ (\overline{\delta}(s', x) \in F \Leftrightarrow \overline{\delta}(t', x) \in F) \Leftrightarrow & \quad \text{(pela definição de } \equiv\ \text{)} \\
s' \equiv t' \Leftrightarrow & \quad \text{(pela definição de [ ])} \\
[s'] = [t'] \Leftrightarrow & \quad \text{(pela definição de } s \text{ e } t\text{)} \\
s = t & \quad \text{(C.Q.D.)}
\end{aligned}
$$

Dado o AFD $A = < \Sigma, S, S_0, \delta, F >$, com os Teoremas 3.5 e 3.6 podemos construir um AFD equivalente a A e que é reduzido. Mas para que isso possa ser feito é necessário que conheçamos as classes de estados equivalentes de A. Para calcular a partição de equivalência de A vamos definir as seguintes relações:

Definição 3.12 *Dados o AFD $A = < \Sigma, S, S_0, \delta, F >$ e um número inteiro i, define-se a i-ésima relação parcial de equivalência $\equiv_i$ como:*

$$(\forall s, t \in S)\ s \equiv_i t \Leftrightarrow ((\forall x \in \Sigma^* \ni |x| \leq i)\ \overline{\delta}(s, x) \in F \Leftrightarrow \overline{\delta}(t, x) \in F)$$

Essas relações indicam que s e t são **i-equivalentes** se eles não podem ser distinguidos por nenhum string com tamanho menor que ou igual a i. O que queremos é calcular as classes de estados 0-equivalentes, depois as 1-equivalentes, etc. Até que cheguemos aos estados que são equivalentes para qualquer string, de qualquer tamanho.

Vamos chamar de E_i a partição determinada por $\equiv_i$. Iniciamos com E_0. Nessa partição as classes são compostas por estados que não podem ser diferenciados ao processarem o string vazio. Podemos então identificar duas classes em E_0 que são F e $S - F$.

Para calcularmos E_{i+1} vamos utilizar E_i e a função de transição de estados. Primeiro, é fácil verificar, pela própria definição de $\equiv_i$, que

a. $s \equiv_{i+1} t \Rightarrow s \equiv_i t$

b. $s \not\equiv_i t \Rightarrow s \not\equiv_{i+1} t$

Isso significa que as classes de E_i são subdivididas para formar as classes de E_{i+1} mas que elementos de duas classes distintas de E_i não podem estar numa mesma classe de E_{i+1}. Em outras palavras, E_{i+1} é um **refinamento** de E_i. Precisamos ainda do seguinte teorema para para calcular E_{i+1}:

Teorema 3.7 *Dados o AFD $A = < \Sigma, S, S_0, \delta, F >$ e $s, t \in S$ então*

$$s \equiv_{i+1} t \Leftrightarrow s \equiv_i t \wedge (\forall a \in \Sigma)\ \delta(s, a) \equiv_i \delta(t, a)$$

Prova: Sejam $s, t \in S$. Temos:

$$s \equiv_i t \wedge (\forall a \in \Sigma)\ \delta(s,a) \equiv_i \delta(t,a) \quad \Leftrightarrow$$

$$(\forall x \in \Sigma^* \ni |x| \leq i)\ \overline{\delta}(s,x) \in F \Leftrightarrow \overline{\delta}(t.x) \in F \wedge$$
$$(\forall a \in \Sigma)(\forall y \in \Sigma^* \ni |y| \leq i)\ \overline{\delta}(\delta(s,a),y) \in F \Leftrightarrow \overline{\delta}(\delta(t,a),y) \in F \quad \Leftrightarrow$$

$$(\forall x \in \Sigma^* \ni |x| \leq i)\ \overline{\delta}(s,x) \in F \Leftrightarrow \overline{\delta}(t.x) \in F \wedge$$
$$(\forall a \in \Sigma)(\forall y \in \Sigma^* \ni |y| \leq i)\ \overline{\delta}(s,a.y) \in F \Leftrightarrow \overline{\delta}(t,a.y) \in F \quad \Leftrightarrow$$

$$(\forall x \in \Sigma^* \ni |x| \leq i)\ \overline{\delta}(s,x) \in F \Leftrightarrow \overline{\delta}(t.x) \in F \wedge$$
$$(\forall z \in \Sigma^* \ni 1 \leq |z| \leq i+1)\ \overline{\delta}(s,z) \in F \Leftrightarrow \overline{\delta}(t,z) \in F \quad \Leftrightarrow$$

$$(\forall x \in \Sigma^* \ni |x| \leq i+1)\ \overline{\delta}(s,x) \in F \Leftrightarrow \overline{\delta}(t.x) \in F \wedge \quad \Leftrightarrow$$

$$s \equiv_{i+1} t$$

Em outras palavras, se para dois estados s e t, que são i-equivalentes, tivermos que a $\delta(s,a)$ e $\delta(t,a)$ levam sempre a dois estados também i-equivalentes, para todas as letras do alfabeto, então esses estados s e t são $i+1$-equivalentes. No exemplo da Figura 3.5 teríamos:

$E_0 : E_{0,1} = S - F = \{S_0, S_1, S_3, S_4\}$
$\quad\ E_{0,2} = F = \{S_2, S_5\}$

Para calcular E_1 tomamos primeiro os estados de $E_{0,1}$ e verificamos que:

- $\delta(S_0,a) \not\equiv_0 \delta(S_1,a)$ e portanto, S_0 e S_1 não são 1-equivalentes e devem aparecer em classes distintas em E_1
- $\delta(S_0,k) \equiv_0 \delta(S_3,k)$ para qualquer $k \in \Sigma$ e portanto S_0 e S_3 são 1-equivalentes e devem continuar na mesma classe em E_1
- $\delta(S_0,a) \not\equiv_0 \delta(S_4,a)$ e portanto, S_0 e S_4 não são 1-equivalentes e devem aparecer em classes distintas em E_1
- $\delta(S_1,a) \not\equiv_0 \delta(S_3,a)$ e portanto, S_1 e S_3 não são 1-equivalentes e devem aparecer em classes distintas em E_1
- $\delta(S_1,k) \equiv_0 \delta(S_4,k)$ para qualquer $k \in \Sigma$ e portanto S_1 e S_4 são 1-equivalentes e devem continuar na mesma classe em E_1
- $\delta(S_3,a) \not\equiv_0 \delta(S_4,a)$ e portanto, S_3 e S_4 não são 1-equivalentes e devem aparecer em classes distintas em E_1

Em seguida tomamos os estados de $E_{0,2}$ e verificamos que:

- $\delta(S_2,k) \equiv_0 \delta(S_5,k)$ para qualquer $k \in \Sigma$ e portanto S_2 e S_5 são 1-equivalentes e devem continuar na mesma classe em E_1

Assim, teremos:

$E_1 : E_{1,1} = \{S_0, S_3\}$
$E_{1,2} = \{S_1, S_4\}$
$E_{1,3} = \{S_2, S_5\}$

e repetindo o processo:

$E_2 : E_{2,1} = \{S_0\}$
$E_{2,2} = \{S_3\}$
$E_{2,3} = \{S_1, S_4\}$
$E_{2,4} = \{S_2, S_5\}$

Constatamos então que $E_2 = E_3 = E_4 \ldots$. Ou seja, para qualquer $n \geq 2$ temos que $E_n = E_{n+1}$. Isso nos indica que, se chegarmos a um ponto em que $E_i = E_{i+1}$, podemos parar o processo pois já chegamos à partição de estados equivalentes que procuravamos. Isso é mostrado pelo Teorema 3.8. Além disso, sabemos que isso sempre acontece; no pior caso, $E_i = E_{i+1}$ ocorre quando $i = |S|$, e cada classe de E_i possui apenas um estado do AFD.

Teorema 3.8 *Dado a AFD* $A = < \Sigma, S, S_0, \delta, F >$,

$$E_i = E_{i+1} \Rightarrow (\forall k \in \mathcal{N})\ E_i = E_{i+k}$$

Prova:

3.4 Algoritmo de Minimização

Com o que vimos nas duas seções anteriores podemos determinar então um algoritmo para minimizar um AFD qualquer. Esse algoritmo pode ser resumido da seguinte maneira:
Dado o AFD $A = < \Sigma, S, S_0, \delta, F >$,

a. Achar os estados não alcançáveis de A

b. Eliminar os estados não alcançáveis, conforme a definição 3.6, obtendo A^c

c. Calcular as classes de estados equivalentes

d. Calcular $(A^c)^r$ conforme a definição 3.11

e. $(A^c)^r$ é o AFD minimal que procuramos

Para mostrar que esse algoritmo realmente calcula o AFD minimal que desejamos, é preciso ainda mostrar que, uma vez obtido um AFD em que todos os estados são alcançáveis, nos passos a e b, essa característica continua valendo, após aplicação da segunda parte do algoritmo que é o redução do AFD.

Teorema 3.9 *Dado o AFD* $A = < \Sigma, S, S_0, \delta, F >$*, se* A *é conexo, então* A^r *também é conexo*

Prova: Vamos tomar $s \in S^r$ e $s' \in S$, tal que $s = [s']$. Então

$$\begin{array}{rl} (\exists x \in \Sigma^*)\ \overline{\delta}(S_0, x) = s' \Leftrightarrow & \text{(pela definição de [ ])} \\ (\exists x \in \Sigma^*)\ [\overline{\delta}(S_0, x)] = [s'] \Leftrightarrow & \text{(por (3.3))} \\ (\exists x \in \Sigma^*)\ \overline{\delta^r}([S_0], x) = [s'] \Leftrightarrow & \text{(pela definição de } S_0^r\text{)} \\ (\exists x \in \Sigma^*)\ \overline{\delta^r}(S_0^r, x) = [s'] & \text{(C.Q.D.)} \end{array}$$

3.5 Implementação

Neste capítulo vamos adicionar à nossa implementação duas novas funções `AFReduz` e `AFConecta` que criam, respectivamente, um AFD reduzido e um AFD conecto a partir de um AFD dado. Foi introduzida também a função `AFPrint` que imprime um AFD, armazenado na nossa estrutura, num arquivo que tem o mesmo formato do arquivo de entrada, lido pela função `main` e descrito no capítulo anterior. Como funções auxiliares foram implementadas as funções:

- `nome_classe`, que dado um conjunto (SET32) que representa uma classe de estados, escolhe um nome para essa classe no formato $[S]$, onde S é o nome do primeiro estado encontrado na partição.
- `AFDeltaCompara` que verifica se dois estados devem continuar na mesma classe de equivalência, no processo de se calcular as classes de estados equivalentes.

Finalmente, foi precisa uma pequena alteração no módulo `alfabeto.c`. Foi introduzida a função `ALFADup` que cria um alfabeto exatamente igual a um alfabeto existente. Essa função é utilizada pelo módulo `afd` pois ao criar-se o autômato reduzido ou conexo é preciso fornecer a esse autômato um alfabeto, igual ao do AFD original. Ao invés de criar um novo alfabeto, introduziu-se um contador que indica quantas vezes o alfabeto está sendo referenciado. A função `ALFADup` incrementa esse contador e a função `ALFADestri` decrementa esse contador. Quando esse contador atinge o valor 0, então `ALFADestroi`, realmente libera o alfabeto.

A seguir é apresentada a parte do código dos arquivos `alfabeto.c` e `afd.c` que foi alterado em relação ao capítulo anterior.

```
/*****************************************************
                          alfabeto.c
Data: 4/5/98
Autor: Marcio Delamaro
*****************************************************/
struct alfabeto {        /* Estrutura que armazena um alfabeto */
    char    **letras;    /* apontdor para as letras do alfabeto */
    int    nletras;      /* numero de letras armazenadas */
    int    cont;         /* numero de usuarios do alfabeto */
```

```
   };

#include    "alfabeto.h"

char buf[1024];


/*-----------------------------------------------------
                          ALFACria
Aloca memoria e armazena um alfabeto.
Parametros:
char *x   -    String que contem todas as letras do
               alfabeto separadas por espaco ou por tab
Retorna:
ALFABETO * - Ponteiro para o objeto do tipo ALFABETO
             que foi criado.
             NULL se nao conseguiu criar o objeto
-----------------------------------------------------*/

ALFABETO *ALFACria(char *s)
{
char *q, *r, *x;
ALFABETO *alfa;
int fim, nletras;

   nletras = 0;                /* numero de letras do alfabeto */
   strcpy(buf,s);
   x = buf;
   while ( isspace(*x)) x++; /* despreza brancos no comeco do string */
   q = x;
   fim = (q == '\0');
   while ( ! fim )          /* conta numero de letras do alfabeto */
   {
        while (! isspace(*q) && *q != '\0') q++;
        fim = (*q == '\0');
        *q++ = '\0';
        nletras++;
        while ( !fim && isspace(*q)) q++;
   }
   alfa = malloc(sizeof *alfa); /*aloca memoria para a struct alfabeto */
   if (alfa == NULL)
       return NULL;

   /* aloca memoria para o vetor de (char *) para armazenar as letras */
   alfa->letras = malloc(nletras * sizeof (char *));
   if (alfa ->letras == NULL)
   {
        free(alfa);
        return NULL;
   }
   alfa->nletras = 0;

   q = x;
   while ( alfa->nletras < nletras ) /* armazena cada letra no vetor */
   {
      alfa->letras[alfa->nletras++] = r = malloc( strlen(q) +1);
      if (r == NULL)
      {
         alfa->nletras--;
         ALFADestroi(alfa);
         return NULL;
      }
```

```
      strcpy(r, q);
      while ( *q != '\0') q++;
      q++;
      while ( alfa->nletras < nletras && isspace(*q)) q++;
   }
   alfa->cont = 1;
   return alfa;
}


/*-----------------------------------------------
                      ALFADestroi
Libera o alfabeto
-----------------------------------------------*/
void ALFADestroi(ALFABETO *alfa)
{
int i;

    if ( --alfa->cont > 0) /* decrementa numero de usuarios */
        return;

    for (i = 0; i < alfa->nletras; i++) /* libera letras */
        free(alfa->letras[i]);
    free(alfa->letras);        /* libera vetor */
    free(alfa);                /* libera alfabeto */
}

/*-----------------------------------------------
                      ALFADup
Incrementa contador de usuarios do alfabeto
-----------------------------------------------*/
ALFABETO *ALFADup(ALFABETO *alfa)
{
    alfa->cont++;
    return alfa;
}
```

Em seguida é apresentado o código nove, introduzido em `afd.c`.

```
/*********************************************************
                          afd.c

Implementacao do tipo de dado AUTOFIN, que representa um
um automato finito deterministico

Data: 23/3/98
Autor: Marcio Delamaro
*********************************************************/

void    nome_classe(AUTOFIN *, SET32 , char *);

/*-------------------------------------------------------
              AFConecta
Cria um novo AFD que eh conexo
Parametros:
AUTOFIN *a: automato finito a transformar

Retorna:
AUTOFIN *acon   automato conexo equivalente ao AUTOFIN a
                NULL em caso de erro
-------------------------------------------------------*/
```

```
AUTOFIN *AFConecta(AUTOFIN *a)
{
SET32 ci;        /* conjunto C_i */
SET32 new;      /* novos estados incluidos no ultimo C_i*/
SET32 next;     /* proximos estados a serem incluidos */
int i, j, k;
AUTOFIN *acon;

   ci = SET32Set(0, a->initial, 1);
   new = ci;
   while ( new != 0 )   /* enquanto novo estado foi incluido */
   {
next = 0;
        for (i = SET32Next(new,0); i >= 0; i = SET32Next(new, i+1))
{
            for ( j = 0; j < a->alfalen; j++)
            {
/* k <-- estado destino da transicao do estado
                    i com a j-esima letra */
                k = a->transitions[i*a->alfalen+j];
                if (k < 0)
                   return NULL; /* AFD invalido */
                if ( SET32Get(ci,k) == 0)
                    next = SET32Set(next, k, 1);
             }
         }
         new = next;
         ci = SET32Union(ci,new);
    }

    /* ci contem os estados conexos. elimina os demais */
    acon = AFCria("a");
    if (acon == NULL)
          return NULL;
    ALFADestroi(acon->alfa);
    acon->alfa = ALFADup(a->alfa);
    acon->alfalen = a->alfalen;
    for (i = SET32Next(ci,0); i >= 0; i = SET32Next(ci, i+1))
    {  /* adiciona os estados de ci */
        k = (i == a->initial)? INITIAL: 0;
k += (SET32Get(a->final,i))? FINAL: 0;
        if (AFAddEstado(acon, a->states[i].label, k) < 0)
        {
           r0:
           AFDestroi(acon);
           return NULL;
        }
    }

    for (i = SET32Next(ci,0); i >= 0; i = SET32Next(ci, i+1))
    {  /* adiciona as transicoes */
       for (j = 0; j < a->alfalen; j++)
       {
          k = a->transitions[i*a->alfalen+j];
          if ( AFAddTransi(acon, a->states[i].label,
                a->states[k].label, ALFALetra(a->alfa,j)) < 0 )
           {
              goto r0;
           }
       }
    }
    return acon;
```

```
}


/*-------------------------------------------------------------
              AFReduz
Cria um novo AFD que eh reduzido
Parametros:
AUTOFIN *a: automato finito a transformar

Retorna:
AUTOFIN *acon   automato reduzido equivalente ao AUTOFIN a
                NULL em caso de erro
-------------------------------------------------------------*/
AUTOFIN *AFReduz(AUTOFIN *a)
{
int i, j, k, l, m, nparti, np2;
SET32 *particao, *p2, *paux, x;
static char buf1[LABELSIZE+1], buf2[LABELSIZE+1];
AUTOFIN *acon;

   /* cada particao[i] eh uma classe de equivalencia.
      o numero maximo eh o numero de estados do AFD */
   particao = malloc(a->nstates*sizeof(*particao));
   if (particao == NULL)
       return NULL;

   p2 = malloc(a->nstates*sizeof(*particao));
   if (p2 == NULL)
   {
       free(particao);
       return NULL;
   }

   particao[0] = a->final; /* conjunto F */
   x = 0;
   for (i = 0; i < a->nstates; i++)
      x = SET32Set(x,i,1);
   particao[1] = SET32Inter(x, ~ a->final); /* conjunto S - F */
   nparti = 2; /* numero inicial de particoes */

   do {  /* calcula as classes de equivalencia */
    np2 = 0;  /* zera proxima particao */
    for (i = 0; i < a->nstates; i++)
        p2[i] = 0;

    for (i = 0; i < nparti; i++)
    {
      x = particao[i];
      for (j = SET32Next(x,0); j >= 0; j = SET32Next(x,j+1))
      {
         p2[np2] = SET32Set(0, j, 1);
         x = SET32Set(x, j, 0);
         for (k = SET32Next(x,j+1); k >= 0; k = SET32Next(x,k+1))
         {
            if (AFDeltaCompara(a, particao, nparti,j, k))
            {    /* estados j e k devem ficar juntos */
                p2[np2] = SET32Set(p2[np2], k, 1);
                x = SET32Set(x, k, 0);
            }
         }
         np2++;
      }
```

```
    }
     paux = p2;
     p2 = particao;
     particao = paux;
     k = np2;
     np2 = nparti;
     nparti = k;
   } while (np2 < nparti);

    acon = AFCria("a");
    if (acon == NULL)
          return NULL;
    ALFADestroi(acon->alfa);
    acon->alfa = ALFADup(a->alfa);
    acon->alfalen = a->alfalen;

   for (i = 0; i < nparti; i++)
   {   /* insere um estado para cada particao calculada */
      nome_classe(a,particao[i], buf1);
      k = ( SET32Get(particao[i],a->initial) )? INITIAL: 0;
      k += (SET32Inter(particao[i],a->final) )? FINAL: 0;
      if (AFAddEstado(acon, buf1, k) < 0)
      {
         r0:
         free(particao);
         free(p2);
         AFDestroi(acon);
         return NULL;
      }
   }
   for (i = 0; i < nparti; i++)
   {   /* insere as transicoes */
      k = SET32Next(particao[i], 0); /* pega o primeiro estado da particao */
      for (j = 0; j < a->alfalen; j++)
      {
         l = a->transitions[k*a->alfalen+j];
         for (m = 0; m < nparti; m++)
            if ( SET32Get(particao[m], l) )
               break;
         nome_classe(a,particao[i],buf1);
         nome_classe(a,particao[m],buf2);
         if (AFAddTransi(acon, buf1, buf2, ALFALetra(a->alfa, j) ) < 0)
              goto r0;
      }
   }

   free(particao);
   free(p2);
   return acon;
}

/*-------------------------------------------------------------
                        nome_classe
Dado um conjunto de estados, escolhe um nome para esses conjunto
-------------------------------------------------------------*/
void nome_classe(AUTOFIN *a, SET32 x, char *s)
{
int k;

   s[0] = '\0';
   k = SET32Next(x, 0); /* pega primeiro elemento do conjunto */
   if (k < 0)
```

```
      return;
   sprintf(s, "[%s]", a->states[k].label);
}


/*-------------------------------------------------------------
                          AFDeltaCompara
Verifica se dois estados devem permanecer na mesma classe de
equivalencia.
Parametros:
AUTOFIN *a:       o AFD
STE32 particao[]  classes de equivalencia
int npart         numero de classes de equivalencia
int s1,s2         os estados a comparar
Retorna:
int              1 se devem ficar
                 0 se nao devem ficar
-------------------------------------------------------------*/
int AFDeltaCompara(AUTOFIN *a, SET32 particao[], int npart, int s1, int s2)
{
int i, j, k, l, m;

   for (i = 0; i < a->alfalen; i++)
   {
      j = a->transitions[a->alfalen*s1+i];
      k = a->transitions[a->alfalen*s2+i];
      for (l = 0; l < npart; l++)
      {
         if ( SET32Get(particao[l], j) )
            if (! SET32Get(particao[l],k) )
                return 0;
      }
   }
   return 1;
}



/*-------------------------------------------------------------
                         AFPrint
Imprime o automato finito no mesmo formato do arquivo de entrada

Parametrso:
AUTOFIN *a:     o automato
FILE *fp:    arquivo onde deve ser escrito
-------------------------------------------------------------*/
void AFPrint(AUTOFIN *a, FILE *fp)
{
static char buf[1024];
int i, j, k;
SET32 next;

   for (i = 0; i < a->alfalen; i++)
fprintf(fp, "%s ", ALFALetra(a->alfa, i));

   for (i = 0; i < a->nstates; i++)
   {
fprintf(fp, "\nESTADO %s %s %s", a->states[i].label,
(a->initial == i)? "inicial" : "",
SET32Get(a->final, i)? "final": "");
   }
```

```
   for (i = 0; i < a->nstates; i++)
   {
for (j = next = 0; j < a->alfalen; j++)
   next = SET32Set(next, a->transitions[i*a->alfalen+j], 1);
for (j = SET32Next(next, 0); j >= 0; j = SET32Next(next, j+1))
{
   buf[0] = '\0';
   for (k = 0; k < a->alfalen; k++)
   {
if ( a->transitions[i*a->alfalen+k] == j)
{
   strcat(buf, ALFALetra(a->alfa, k));
   strcat(buf, " ");
}
   }
   if ( buf[0] != '\0')
   {
fprintf(fp, "\nTRANSITION %s %s %s", a->states[i].label,
a->states[j].label, buf);
   }
}
   }
   fprintf(fp, "\n");
}
```

Capítulo 4

Autômatos Finitos Não Determinísticos

Um Autômato Finito Não Determinístico (AFND) é um modelo matemático semelhante a um AFD. Um AFND porém "relaxa" algumas das condições impostas por um AFD. Sua principal característica é que podem existir múltiplos caminhos ou nenhum caminho para processar um determinado string, ao contrário dos AFD's, nos quais sabemos existe uma única seqüência de estados que é seguida ao processar-se um determinado string.

Iniciamos o estudo dos AFND com a sua definição formal, permitindo que se compare este modelo com o de AFD's.

4.1 Definições Básicas

Definição 4.1 *Um Autômato Finito Não Determinístico é uma quíntupla* $< \Sigma, S, S_0, \delta, F >$ *onde*

Σ *é o alfabeto de entrada*

S *é um conjunto finito não vazio de estados*

S_0 *é um conjunto não vazio de estados iniciais,* $S_0 \subseteq S$

δ *é a função transição de estados, definida* $\delta : S \times \Sigma \rightarrow \rho(S)$

F *é o conjunto de estados finais,* $F \subseteq S$

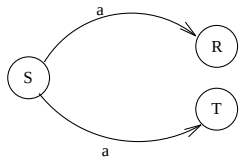
Σ, S e F são caracterizados exatamente como nos AFD's. No caso dos AFD's, tinhamos um único estado inicial, sendo portanto S_0 um elemento de S. Nos AFND's podemos ter diversos (mas pelo menos um) estados iniciais, fazendo

de S_0 um subconjunto de S. Essa mudança leva a uma diferença significativa no comportamento de um AFND, que é o fato de que ele pode ter mais do que um estado "ativo" ou corrente ao mesmo tempo. Inicialmente, todos os estados pertencentes ao conjunto S_0 são ativados, antes de processar-se qualquer letra da cadeia de entrada.

Outra forma de se interpretar essa multiplicidade de estados iniciais é considerando que existem diversas "alternativas" para se tentar reconhecer um string. Tenta-se a partir de um estado inicial R. Se não se chegar a um estado final, ao tentar reconhecer o string x, pode-se tentar com outro estado inicial $T \in S_0$ e assim com todos os estados pertencentes a S_0. Se para **algum deles** x levar a um estado final, então x é parte da linguagem definida pelo autômato. O string x só é rejeitado se a partir de nenhum estado inicial é possível atingir-se um estado final. No Diagrama de Transição de Estados de um AFND, representamos cada estado inicial com uma seta chegando no estado, sem nenhum estado de origem.

A função δ não é mais definida $\delta : S \times \Sigma \to S$ e sim $\delta : S \times \Sigma \to \rho(S)$, onde $\rho(S)$ é o conjunto potência de S, ou seja, o conjunto de todos os subconjuntos de S. Assim, cada elemento do contra-domínio de δ é um conjunto de estados e não mais um um único estado. Por exemplo, podemos ter que $\delta(S, a) = \{R, T\}$, indicando que, se o estado S estiver ativo e uma letra a aparecer na entrada, então ambos os estados, R e T pasarão a ser ativos (lembrando que diversos estados podem estar simultaneamente ativos).

Como no caso dos estados iniciais, pode-se interpretar o fato de $\delta(S, a)$ levar a dois estados distintos como uma alternativa de processamento. Se seguirmos o caminho através de R e não chegarmos a um estado final podemos tentar através de T. Se, para alguma das alternativas, chegarmos a um estado final, então a cadeia é reconhecida. Ela só é rejeitada se nenhuma das alternativas leva a um estado final. No Diagrama de Transição de Estados de um AFND, representamos essa situação como:



De forma semelhante aos AFD's, uma cadeia x é aceita por um AFND se, ao término do seu processamento, **algum** dos estados ativos pertence ao conjunto F. Como já foi dito, o comportamento dos AFND's difere do comportamento dos AFD's pois ao processar uma dada letra, todas as transições rotuladas com aquela letra, a partir de todos os estados ativos serão efetuadas. Por exemplo, na Figura 4.1a temos os estados S_0 e S_1 ativos. Ao processar a letra a passaríamos à situação mostrada na Figura 4.1b com S_2, S_3 e S_4 ativos. E daí, ao processar um b, passaríamos à situação da Figura 4.1c e ao processar outro a à situação da Figura 4.1c.

A definição de δ permite que $\delta(s, a) = \emptyset$. Assim, ao processar-se a letra a a partir da situação mostrada na Figura 4.1c, passamos a uma situação em que nenhum estado está ativo, mostrada na Figura 4.1d. Visto que $\delta(S_3, a) =$

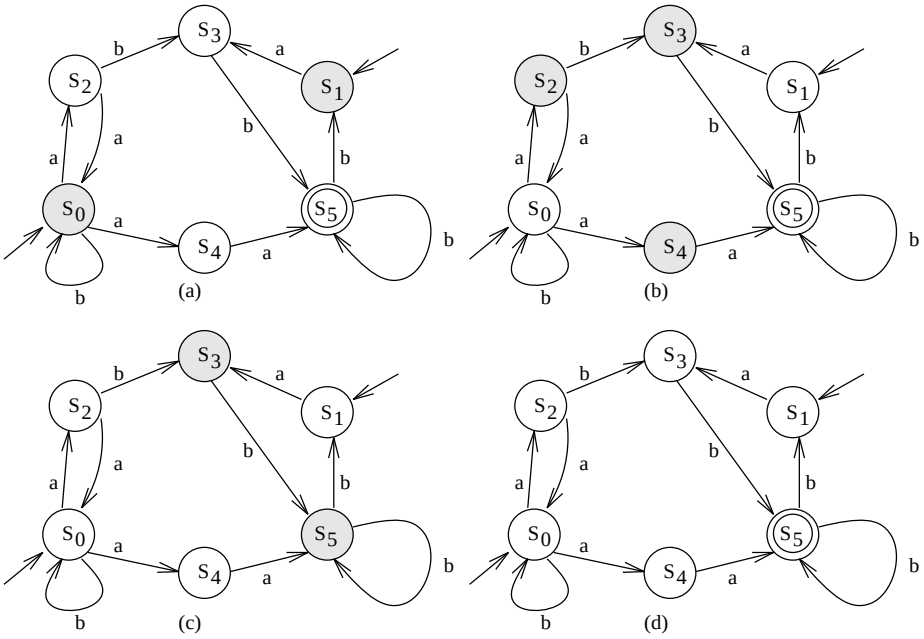


Figura 4.1: Comportamento de um AFND

$\delta(S_5, a) = \emptyset$, – fato mostrado pela falta de arco rotulado com a letra a saindo dos estados S_3 e S_5 – o processamento dessa letra simplesmente "desativa" esses estados, levando a uma situação em que não se pode continuar o processamento da cadeia de entrada. Isso indica que tal cadeia não é reconhecida pelo AFND (pois a partir desse ponto não se chegaria a nenhum estado pertencente a F).

No exemplo do problema HLCR, visto no Capítulo 2, dissemos que o diagrama apresentado não se encaixa na definição dada para um AFD. Isso acontece porque as ações inválidas, que levariam o sistema a um estado ilegal, não aparecem no diagrama. Aquele diagrama poderia ser utilizado para representar um AFND, conforme a definição vista. Já o diagrama da Figura 2.9 poderia ser simplificado, eliminando-se o estado S_7 que foi utilizado como um "estado de erro" ao se reconhecer prematuramente que um string não é uma constante de ponto flutuante. O Diagrama correspondente seria o mostrado na Figura 4.2. Note-se que nos diagramas do problema HLCR e da Figura 4.2 não existe na realidade nenhum caso de não determinismo, como veremos adiante. Iremos entretanto considerar tais autômatos como AFND's pois eles se encaixam na definição de um AFND mas não na de um AFD.

A característica de não determinismo surge num Autômato Finito quando pode existir mais do que um estado ativo ao mesmo tempo, ou seja, quando existe mais de uma possível transição para uma mesma letra num mesmo estado ou mais do que um estado inicial, indicando que uma cadeia pode ser processada de formas alternativas. Por exemplo, supondo o alfabeto $\{0, 1\}$, queremos um Autômato Finito que reconheça a linguagem formada pelos strings que contenham 1011. O Autômato Finito Não Determinístico $A = < \Sigma, \{A, B, C, D, E\}, \{A\}, \delta, \{E\}$, onde δ é dada pelo diagrama da Figura 4.3, realiza essa tarefa. O surgimento

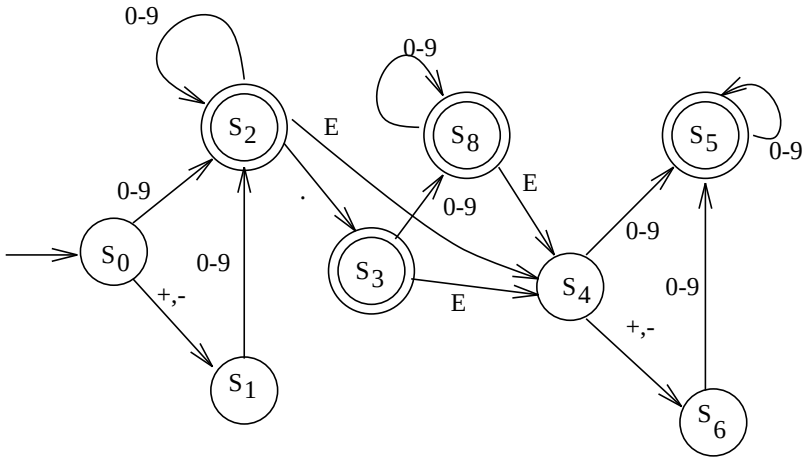


Figura 4.2: AFND para constantes numéricas de ponto flutuante

de uma letra 1 indica o possível início do substring procurado. Já a transição que permanece no estado A serve para o caso da letra 1 não ser parte do substring procurado. Nesse caso, novas letras 1 iniciam o processo de tentativa de "casamento" outra vez.

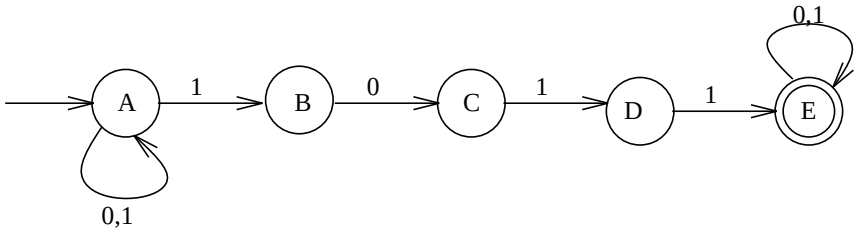


Figura 4.3: AFND para $L = \{x \in \{0,1\}^* \mid 1011 \text{ é subpalavra de } x\}$

Outro exemplo de AFND é $A = < \{0,1\}, \{S_0, S_1, S_2, S_3, S_4, S_5\}, \{S_0\}, \delta, \{S_4, S_5\} >$, onde δ é dada na Tabela abaixo e na Figura 4.4.

δ	0	1
S_0	$\{S_1, S_5\}$	$\{S_2, S_5\}$
S_1	$\{S_1\}$	$\{S_1, S_2\}$
S_2	$\{S_3\}$	$\emptyset$
S_3	$\emptyset$	$\{S_4\}$
S_4	$\emptyset$	$\emptyset$
S_5	$\{S_6\}$	$\{S_6\}$
S_6	$\{S_5\}$	$\{S_5\}$

No diagrama podemos identificar duas partes distintas, cada uma responsável por reconhecer um conjunto de strings. A parte superior reconhece os strings que terminam com 101. A parte de baixo reconhece todos os strings com número ímpar de letras. Vamos analisar a seqüência de estados ativos para três cadeias:

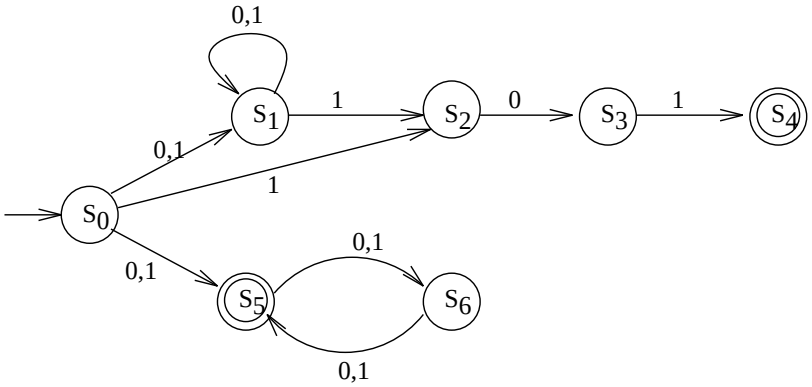


Figura 4.4: AFND para $L = \{x \in \{0,1\}^* \mid 101 \text{ é sufixo de } x \ \vee \ |x| \text{ é ímpar }\}$

001101: $\{S_0\} \xrightarrow{0} \{S_1, S_5\} \xrightarrow{0} \{S_1, S_6\} \xrightarrow{1} \{S_1, S_2, S_5\} \xrightarrow{1} \{S_1, S_2, S_6\} \xrightarrow{0} \{S_1, S_3, S_5\} \xrightarrow{1} \{S_1, S_2, S_4, S_6\}$

100: $\{S_0\} \xrightarrow{1} \{S_1, S_2, S_5\} \xrightarrow{0} \{S_1, S_3, S_6\} \xrightarrow{0} \{S_1, S_5\}$

10101: $\{S_0\} \xrightarrow{1} \{S_1, S_2, S_5\} \xrightarrow{0} \{S_1, S_3, S_6\} \xrightarrow{1} \{S_1, S_2, S_4, S_5\} \xrightarrow{0} \{S_1, S_3, S_6\} \xrightarrow{1} \{S_1, S_4, S_5\}$

Esses três strings são reconhecidos pelo AFND. Para que uma cadeia seja reconhecida, é necessário que (pelo menos) um dos estados ativos no final do processamento do string pertença ao conjunto F. No caso do string 001101, esse estado é o S_4, ou seja, o string é reconhecido na parte "superior" do diagrama. No caso do string 100, ele não é reconhecido na parte superior mas o é na parte inferior pois o estado S_5 está ativo. No caso do string 10101, ambos os estados estão ativos, $\{S_4, S_5\}$, fazendo com que o string seja também reconhecido.

Um exemplo de string que não é reconhecido pelo AFND é:

1001: $\{S_0\} \xrightarrow{1} \{S_1, S_2, S_5\} \xrightarrow{0} \{S_1, S_3, S_6\} \xrightarrow{0} \{S_1, S_5\} \xrightarrow{1} \{S_1, S_6\}$

Nesse caso, nenhum dos estados de $\{S_1, S_6\}$ pertence ao conjunto F, ou seja, $\{S_1, S_6\} \cap F = \emptyset$ e por isso o string não é reconhecido.

Outra forma de representar essa mesma linguagem através de AFND é mostrada na Figura 4.5. Nela, dois grupos distintos e não conexos de estados foram utilizados para formar o AFND, cada um com um estado inicial próprio. Esse tipo de autômato é perfeitamente aceitável, quando se trata de AFND. Para esse novo AFND, teríamos a seguinte seqüência de estados para os strings 001101 e 1001:

001101: $\{S_1, S_6\} \xrightarrow{0} \{S_1, S_5\} \xrightarrow{0} \{S_1, S_6\} \xrightarrow{1} \{S_1, S_2, S_5\} \xrightarrow{1} \{S_1, S_2, S_6\} \xrightarrow{0} \{S_1, S_3\} \xrightarrow{1} \{S_1, S_2, S_4\}$

1001: $\{S_1, S_6\} \xrightarrow{1} \{S_1, S_2, S_5\} \xrightarrow{0} \{S_1, S_3, S_6\} \xrightarrow{0} \{S_1, S_5\} \xrightarrow{1} \{S_1, S_2, S_6\}$

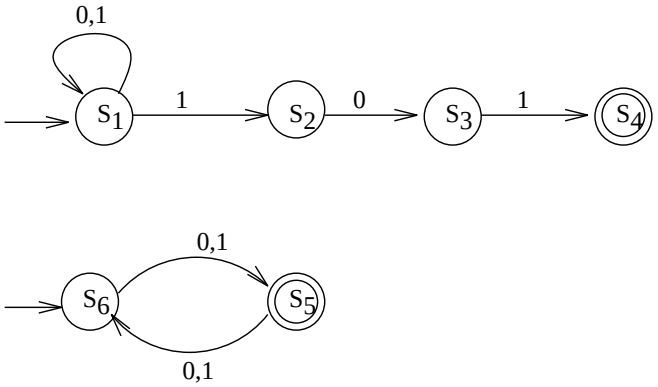


Figura 4.5: AFND também para $L = \{x \in \{0,1\}^* \mid 101$ é sufixo de $x \ \vee \ |x|$ é ímpar $\}$

Para definirmos formalmente quando um string é ou não reconhecido por um AFND, precisamos definir $\overline{\delta}$, a função de transição extendida, assim como fizemos para os AFD's.

Definição 4.2 *Dado o AFND* $A = < \Sigma, S, S_0, \delta, F >$, *a função de transição extendida associada a* A, *definida* $\overline{\delta} : S \times \Sigma^* \to \rho(S)$ *é dada por:*

$$(\forall s \in S)\ \overline{\delta}(s, \lambda) = \{s\}$$

$$(\forall s \in S)(\forall a \in \Sigma)(\forall x \in \Sigma^*)\ \overline{\delta}(s, ax) = \bigcup_{q \in \delta(s,a)} \overline{\delta}(q, x)$$

Essa definição é recursiva e nos diz que o valor de $\overline{\delta}$ para qualquer estado e o string λ é o próprio estado. Para outros strings ax pertencentes a Σ^*, aplica-se a função δ sobre o par $< s, a >$, achando um conjunto de estados $\{q \in S | \delta(s,a) = q\}$ e então aplica-se recursivamente a definição de $\overline{\delta}$ para cada um desses estados q. Obtem-se assim um conjunto de estados que são o valor de $\overline{\delta}(s, ax)$. Tendo definido e função de transição extendida, podemos então definir o que é aceitação para um AFND.

Definição 4.3 *Dado o AFND* $A = < \Sigma, S, S_0, \delta, F >$ *e a cadeia* $x \in \Sigma^*$, *diz-se que:*

- A **aceita** x *sse* $(\bigcup_{q \in S_0} \overline{\delta}(q, x)) \cap F \neq \emptyset$
- A **rejeita** x *sse* $(\bigcup_{q \in S_0} \overline{\delta}(q, x)) \cap F = \emptyset$

A linguagem $L(A)$, definida pelo AFND A é o conjunto de todas as palavras reconhecidas por esse autômato, ou seja:

Definição 4.4 *Dado o AFND* $A = < \Sigma, S, S_0, \delta, F >$, *define-se a linguagem* $L(A)$, *reconhecida por esse autômato como sendo*

$$L(A) = \{x \in \Sigma^* \mid (\bigcup_{q \in S_0} \overline{\delta}(q, x)) \cap F \neq \emptyset\}$$

As definições 4.3 e 4.4 utilizam o conjunto de estados $(\bigcup_{q \in S_0} \overline{\delta}(q, x))$ que é o conjunto de todos os estados alcançados a partir de todos todos os estados inicias $q \in S_0$. Como foi dito anteriormente, podemos ver a existência de diversos estados iniciais como a possibilidade de diversas alternativas diferentes para processar um determinado string. Dentro desse espírito, podemos alterar as definições 4.3 e 4.4 para obtermos:

Definição 4.5 *Dado o AFND* $A = < \Sigma, S, S_0, \delta, F >$ *e a cadeia* $x \in \Sigma^*$, *diz que:*

- A **aceita** x *sse* $(\exists q \in S_0) \ni \overline{\delta}(q, x)) \cap F \neq \emptyset$
- A **rejeita** x *sse* $(\neg\exists q \in S_0) \ni \overline{\delta}(q, x)) \cap F = \emptyset$

Definição 4.6 *Dado o AFND* $A = < \Sigma, S, S_0, \delta, F >$, *define-se a linguagem* $L(A)$, *reconhecida por esse autômato como sendo*

$$L(A) = \{x \in \Sigma^* \mid (\exists q \in S_0) \ni \overline{\delta}(q, x)) \cap F \neq \emptyset\}$$

A definição de equivalência entre AFND's é exatamanete a mesma daquela utilizada para AFD's. Ou seja, dados dois AFND's A e B, diz que $A \equiv B \Leftrightarrow L(A) = L(B)$.

4.2 AFND's com transições λ

O modelo de AFND pode ser estendido ainda mais através da inclusão de transições que ocorrem sem que nenhuma letra da entrada seja processada. Vamos tomar como exemplo o AFND $A = < \{a, b, c\}, \{S_0, S_1, S_2\}, \{S_0\}, \delta, \{S_2\} >$, onde δ é dada pelo diagrama da Figura 4.6. Esse AFND reconhce qualquer

string que possua uma ou mais seqüências bc como sub-palavras, mas sempre que um b aparecer, ele deve vir seguido de um c. O objetivo da transição rotulada λ é fazer com que, sempre que o estado S_2 se tornar ativo, o estado S_0 torne-se ativo também. Dessa forma, um string que possua mais do que um bc pode ser reconhecido pelo AFND.

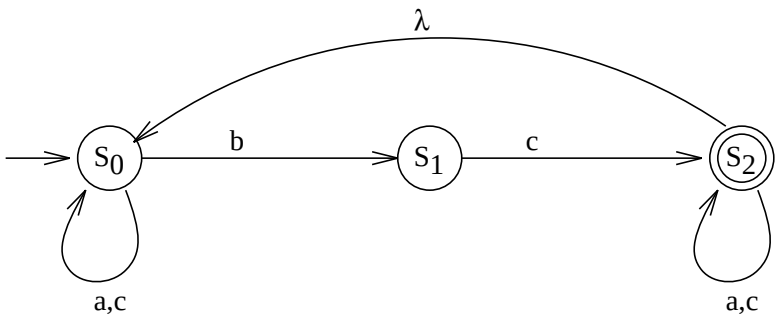


Figura 4.6: AFND com transição λ

Vejamos a seqüência de estados ativos ao processarem-se alguns strings:

bc: $\{S_0\} \stackrel{b}{\rightarrow} \{S_1\} \stackrel{c}{\rightarrow} \{S_2, S_0\}$ (aceito)

abccba: $\{S_0\} \stackrel{a}{\rightarrow} \{S_0\} \stackrel{b}{\rightarrow} \{S_1\} \stackrel{c}{\rightarrow} \{S_0, S_2\} \stackrel{c}{\rightarrow} \{S_0, S_2\} \stackrel{b}{\rightarrow} \{S_1\} \stackrel{a}{\rightarrow} \{\}$ (não aceito)

O modelo matemático utilizado para definirem-se os AFND com transições λ é dado na definição seguinte:

Definição 4.7 *Um Autômato Finito Não Determinístico com transições λ (ou transições vazias) é uma quíntupla $< \Sigma, S, S_0, \delta, F >$ onde*

Σ *é o alfabeto de entrada*

S *é um conjunto finito não vazio de estados*

S_0 *é um conjunto não vazio de estados iniciais, $S_0 \subseteq S$*

δ *é a função transição de estados, definida $\delta : S \times (\Sigma \cup \{\lambda\}) \rightarrow \rho(S)$*

F *é o conjunto de estados finais, $F \subseteq S$*

Note que somente a definição de δ muda em relação aos AFND. Nesse caso, δ é definida para cada estado com cada letra de Σ e para cada estado com o string vazio λ. Quando acorre que $\delta(s, \lambda) \neq \emptyset$, deve-se considerar que o AFND é capaz de realizar uma transição espontaneamente para todos os estados $\delta(s, \lambda)$ sem consumir nenhuma letra da entrada (e sem que s deixe de ser ativo). É o que acontece, por exemplo no exemplo da Figura 4.6, onde $\delta(S_1, c) = \{S_2\}$ e $\delta(S_2, \lambda) = \{S_0\}$. Ao processar, no estado S_1, uma letra c, passa-se ao estado

S_2 que por sua vez, sem processar qualquer letra da entrada, ativa o estado S_0 através da transição vazia.

Outro exemplo de AFND com transições λ é dado por $A = < \{a, b, c, d\}, \{S_0, S_1, S_2, S_3\}, \{S_0\}, \delta, \{S_2\} >$, onde δ é dada pela Figura 4.7. Vejamos abaixo, a seqüência de estados ao processarem-se algumas cadeias nesse autômato.

λ: $\{S_0, S_1, S_2\}$ (aceito)

a: $\{S_0, S_1, S_2\} \xrightarrow{a} \{S_1, S_2\}$ (aceito)

c: $\{S_0, S_1, S_2\} \xrightarrow{c} \{S_2\}$ (aceito)

ddd: $\{S_0, S_1, S_2\} \xrightarrow{d} \{S_2\} \xrightarrow{d} \{S_2\} \xrightarrow{d} \{S_2\}$ (aceito)

ab: $\{S_0, S_1, S_2\} \xrightarrow{a} \{S_1, S_2\} \xrightarrow{b} \{S_3\}$ (não aceito)

abb: $\{S_0, S_1, S_2\} \xrightarrow{a} \{S_1 i, S_2\} \xrightarrow{b} \{S_3\} \xrightarrow{b} \{S_1, S_2\}$ (aceito)

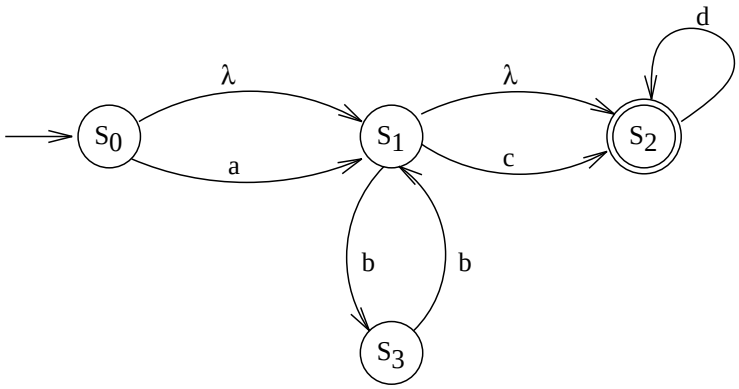


Figura 4.7: Mais um AFND com transição λ

A definição de aceitação de um string para um AFND com transição λ continua sendo a mesma que para um Autômato Finito Não Determinístico. A diferença é que precisamos alterar a definição da função de transição extendida, o $\overline{\delta}$. Vamos tomar por exemplo o estado S_0 da Figura 4.7. O valor da função extendida $\overline{\delta}(S_1, bb)$ deve fornecer o conjunto de estados ativos ao processar-se o string bb a partir de S_1, ou seja, $\{S_1, S_2\}$. Se aplicarmos a definição de $\overline{\delta}$ para um AFND obteremos:

$$\overline{\delta}(S_1, bb) = \overline{\delta}(S_3, b) = \{S_1\}$$

que não corresponde ao resultado desejado. Para definirmos $\overline{\delta}$ corretamente precisamos da seguinte definição:

Definição 4.8 *Dado AFND com transição λ $A = < \Sigma, S, S_0, \delta, F >$, o fecho-$\lambda$ de $t \in S$, denotado por $\Lambda(t)$ é dado pelo conjunto de estados que podem ser alcançados a partir de t sem que nenhuma letra da cadeia de entrada seja processada, ou seja, sempre através de transições λ.*

Essa definição nos diz que o fecho-λ de um estado t é o conjunto de todos os estados que podem ser alcançados a partir de t sem que nenhuma letra da cadeia de entrada seja processada. Assim, para o AFND com transição λ da Figura 4.7 teríamos:

$\Lambda(S_0) = \{S_0\} \cup \Lambda(S_1) = \{S_0, S_1, S_2\}$

$\Lambda(S_1) = \{S_1\} \cup \Lambda(S_2) = \{S_1, S_2\}$

$\Lambda(S_2) = \{S_2\}$

$\Lambda(S_3) = \{S_3\}$

A definição 4.8 pode ser estendida para um conjunto de estados.

Definição 4.9 *Dado um conjunto de estados T, define-se o fecho-λ desse conjunto, denotado por $\Lambda(T)$ como*

$$\Lambda(T) = \bigcup_{t \in T} \Lambda(t)$$

Com a definição do fecho-λ podemos agora definir a função de transição extendida como sendo:

Definição 4.10 *Dado o AFND com transição λ $A = < \Sigma, S, S_0, \delta, F >$, a função de transição extendida associada a A, definida $\overline{\delta} : S \times \Sigma^* \to \rho(S)$ é dada por:*

$$(\forall s \in S) \qquad \overline{\delta}(s, \lambda) = \Lambda(s)$$

$$(\forall s \in S)(\forall a \in \Sigma) \qquad \overline{\delta}(s, a) = \Lambda\Big(\bigcup_{q \in \Lambda(s)} \delta(q, a)\Big)$$

$$(\forall s \in S)(\forall a \in \Sigma)(\forall x \in \Sigma^*) \quad \overline{\delta}(s, ax) = \bigcup_{q \in \overline{\delta}(s,a)} \overline{\delta}(q, x)$$

Deve-se notar que a principal diferença nessa definição é que $\overline{\delta}(s, a) \neq \delta(s, a)$ Por exemplo, no AFND com transição λ representado na Figura 4.7, teríamos:

$$\begin{aligned} \delta(S_0, a) &= \{S_1\} \\ \overline{\delta}(S_0, a) &= \Lambda(\delta(S_0, a) \cup \delta(S_1, a) \cup \delta(S_2, a)) \\ &= \Lambda(\{S_1\} \cup \{\} \cup \{\}) \\ &= \{S_1, S_2\} \end{aligned}$$

E para calcular $\overline{\delta}(S_0, abb)$ teríamos que calcular:

$$
\begin{aligned}
\overline{\delta}(S_0, abb) = \;& \overline{\delta}(S_1, bb) \cup \overline{\delta}(S_2, bb) \\
= \;& \Big(\bigcup_{q \in \overline{\delta}(S_1, b)} \overline{\delta}(q, b) \Big) \cup \Big(\bigcup_{q \in \overline{\delta}(S_2, b)} \overline{\delta}(q, b) \Big) \\
= \;& \Big(\bigcup_{q \in \{S_3\}} \overline{\delta}(q, b) \Big) \cup \Big(\bigcup_{q \in \{\}} \overline{\delta}(q, b) \Big) \\
= \;& \overline{\delta}(S_3, b) \\
= \;& \Lambda(\delta(S_3, b)) \\
= \;& \{S_1, S_2\}
\end{aligned}
$$

4.3 Equivalência entre AFD's e AFND's

A definição de uma linguagem L pode ser facilitada através do uso de um AFND, com ou sem transição λ, ao invés de um AFD. É o caso, por exemplo da linguagem $L = \{x \in \{0,1\}^* \mid 101$ é sufixo de $x \ \vee \ |x|$ é ímpar $\}$ representada através de um AFND na Figura 4.5. Um AFD para tal linguagem seria sensivelmente mais complexo. Sua função de transição é mostrada na Figura 4.8.

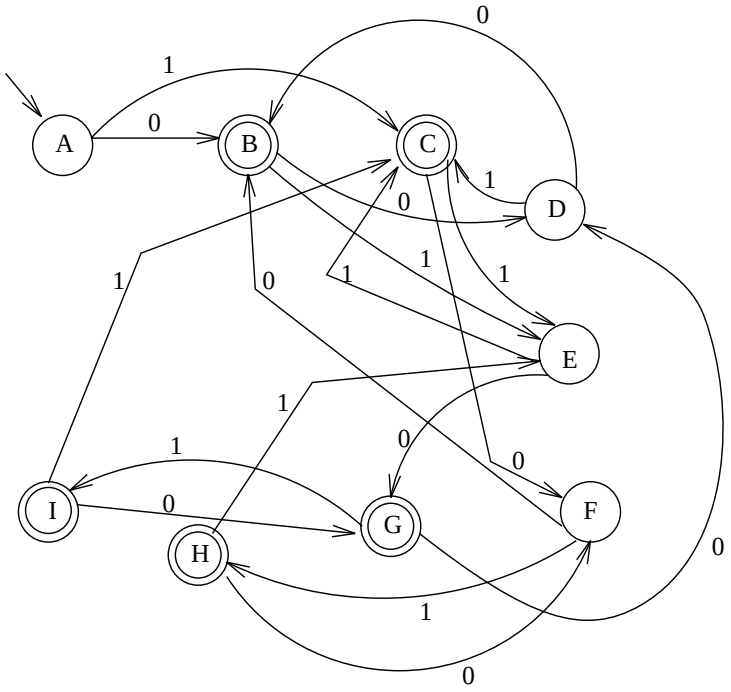


Figura 4.8: AFD para $L = \{x \in \{0,1\}^* \mid 101$ é sufixo de $x \ \vee \ |x|$ é ímpar $\}$

Embora as características de AFND possam facilitar o trabalho de construir autômatos, elas não adicionam nenhum "poder" extra a esses dispositivos. Nessa

sessão iremos mostrar que qualquer linguagem que seja definida através de um AFND com transição λ pode também ser definida através de um AFND sem transição λ ou de um AFD e vice-versa.

Para mostrar isso, iremos mostrar que qualquer AFND com transição λ pode ser transformado num AFND sem transição λ e que qualquer AFND sem transição λ pode ser transformado num AFD. O caminho inverso é trivialmente verdadeiro pois qualquer AFD é um AFND sem transição λ em que o conjunto de estados iniciais é unitário, assim como todo valor de δ. Da mesma forma, qualquer AFND sem transição λ é um AFND com transição λ onde $\delta(s, \lambda) = \emptyset$ para todos os estados do autômato. Nas próximas sessões iremos então mostrar como eliminar as transições λ e como transformar um AFND num AFD.

4.3.1 Eliminação de Transições λ

Definição 4.11 *Dado o AFND com transição* λ $A = < \Sigma, S, S_0, \delta, F >$, *define-se* $A^\sigma = < \Sigma, S \cup \{q_0\}, S_0 \cup \{q_0\}, \delta^\sigma, F^\sigma >$, *onde:*

$$F^\sigma = \begin{cases} F \cup \{q_0\} & \text{se } \lambda \in L(A) \\ F & \text{caso contrário} \end{cases}$$

$$\begin{array}{ll} (\forall a \in \Sigma) & \delta^\sigma(q_0, a) = \emptyset \\ (\forall s \in S)(\forall a \in \Sigma) & \delta^\sigma(s, a) = \overline{\delta}(s, a) = \Lambda(\bigcup_{q \in \Lambda(s)} \delta(q, a)) \end{array}$$

O estado q_0 foi introduzido em A^σ para o caso em que λ pertence a $L(A)$. Ele é um estado isolado ($\delta^\sigma(q_0, a)$ é sempre vazio) que é colocado como estado final sse $\lambda \in L(A)$. Casos em que $\lambda \notin L(A)$, q_0 não precisa ser incluído. A definição de δ^σ é feita de tal forma que $\delta^\sigma(s, a)$ leve a todos os estados aos quais $\overline{\delta}(s, a)$ levaria, o que inclui os estados alcançados através de transições λ.

Vamos tomar como exemplo o autômato da Figura 4.7. Construímos a partir de $A^\sigma = < \{a, b, c, d\}, \{S_0, S_1, S_2, S_3, q_0\}, \{S_0, q_0\}, \{S_2, q_0\} >$. Observe que pode-se determinar se $\lambda \in L(A)$ calculando $\Lambda(\{S_0\}) = \{S_0, S_1, S_2\}$ e verificando que $\Lambda(\{S_0\}) \cap F \neq \emptyset$. Precisamos agora calcular δ^σ. Pela definição temos:

$$\begin{array}{ll}
\delta^\sigma(S_0, a) = \overline{\delta}(S_0, a) = \{S_1, S_2\} & \delta^\sigma(S_2, a) = \overline{\delta}(S_2, a) = \{\} \\
\delta^\sigma(S_0, b) = \overline{\delta}(S_0, b) = \{S_3\} & \delta^\sigma(S_2, b) = \overline{\delta}(S_2, b) = \{\} \\
\delta^\sigma(S_0, c) = \overline{\delta}(S_0, c) = \{S_2\} & \delta^\sigma(S_2, c) = \overline{\delta}(S_2, c) = \{\} \\
\delta^\sigma(S_0, d) = \overline{\delta}(S_0, d) = \{S_2\} & \delta^\sigma(S_2, d) = \overline{\delta}(S_2, d) = \{S_2\} \\
\delta^\sigma(S_1, a) = \overline{\delta}(S_1, a) = \{\} & \delta^\sigma(S_3, a) = \overline{\delta}(S_3, a) = \{\} \\
\delta^\sigma(S_1, b) = \overline{\delta}(S_1, b) = \{S_3\} & \delta^\sigma(S_3, b) = \overline{\delta}(S_3, b) = \{S_1, S_2\} \\
\delta^\sigma(S_1, c) = \overline{\delta}(S_1, c) = \{S_2\} & \delta^\sigma(S_3, c) = \overline{\delta}(S_3, c) = \{\} \\
\delta^\sigma(S_1, d) = \overline{\delta}(S_1, d) = \{S_2\} & \delta^\sigma(S_3, d) = \overline{\delta}(S_3, d) = \{\}
\end{array}$$

O diagrama de transição correspondente a δ^σ é apresentado na Figura 4.9.

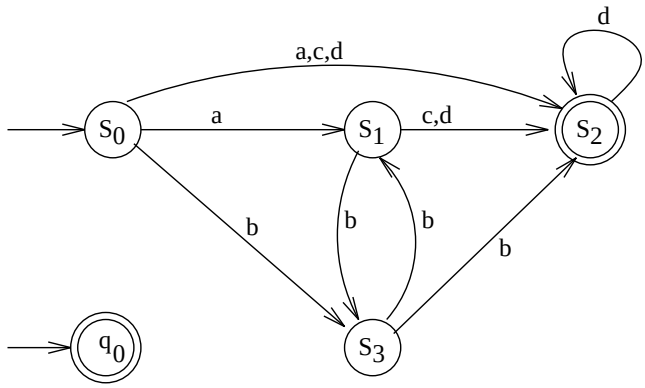


Figura 4.9: AFND sem transição λ equivalente a AFND com transição λ.

Vamos agora mostrar que A^σ, definido acima é realmente equivalente a A. Para isso, iremos antes mostrar que

$$(\forall s \in S)(\forall x \in \Sigma^+) \quad \overline{\delta^\sigma}(s,x) = \overline{\delta}(s,x) \tag{4.1}$$

Por indução no tamanho de x temos:
Base: $|x| = 1$

$$\begin{aligned}(\forall s \in S)(\forall a \in \Sigma) \quad \overline{\delta^\sigma}(s,a) &= \bigcup_{q \in \delta^\sigma(s,a)} \overline{\delta^\sigma}(q,\lambda) \\ &= \overline{\delta^\sigma}(s,a) \\ &= \overline{\delta}(s,a)\end{aligned}$$

Passo de indução: Considerando a hipótese válida para $1 \leq |x| \leq n$ tomamos agora o string ax. Temos:

$$\begin{aligned}\overline{\delta^\sigma}(s,ax) &= \quad \text{(pela definição de } \overline{\delta^\sigma}) \\ \bigcup_{q \in \delta^\sigma(s,a)} \overline{\delta^\sigma}(q,x) &= \quad \text{(pela definição de } \overline{\delta^\sigma}) \\ \bigcup_{q \in \overline{\delta^\sigma}(s,a)} \overline{\delta^\sigma}(q,x) &= \quad \text{(pela hipótese de indução)} \\ \bigcup_{q \in \overline{\delta}(s,a)} \overline{\delta}(q,x) &= \quad \text{(pela definição de } \overline{\delta}) \\ \overline{\delta}(s,ax) &\quad \text{C.Q.D.}\end{aligned}$$

Teorema 4.1 *Dado o AFND com transição* λ $A = <\Sigma, S, S_0, \delta, F>$, $A^\sigma \equiv A$.

Prova: Se $\lambda \in L(A)$, então $\lambda \in L(A^\sigma)$ pois

$$\lambda \in L(A) \Leftrightarrow q_0 \in F^\sigma \Leftrightarrow \lambda \in L(A^\sigma)$$

Tomando agora $x \in \Sigma^+$, podemos desconsiderar q_0 como estado inicial. Temos:

$$\begin{aligned} \lambda \in L(A^\sigma) \Leftrightarrow \\ (\bigcup_{q \in S_0} \overline{\delta^\sigma}(q,x)) \cap F^\sigma \neq \emptyset \Leftrightarrow \\ (\bigcup_{q \in S_0} \overline{\delta}(q,x)) \cap F^\sigma \neq \emptyset \Leftrightarrow \\ (\bigcup_{q \in S_0} \overline{\delta}(q,x)) \cap F \neq \emptyset \Leftrightarrow \\ x \in L(A) \end{aligned}$$

4.3.2 Transformação de um AFND num AFD

Uma vez eliminadas as transições λ, um AFND pode ser transformado num AFD.

Definição 4.12 *Dado o AFND sem transição* λ $A = < \Sigma, S, S_0, \delta, F >$, *define-se o AFD correspondente* $A^d = < \Sigma, S^d, S_0^d, \delta^d, F^d >$ *como:*

$$S^d = \rho(S)$$

$$S_0^d = S_0$$

$$F^d = \{Q \in S^d | Q \cap F \neq \emptyset\}$$

$$(\forall Q \in S^d)(\forall a \in \Sigma) \quad \delta^d(Q,a) = \bigcup_{q \in Q} \delta(q,a)$$

De acordo com essa definição, cada estado de A^d corresponde a um conjunto de estados do AFND A. Por exemplo, o estado inicial de A^d corresponde a todos os estados iniciais de A, ou seja, $S_0^d = S_0$. A função δ^d é definida de maneira que $\delta^d(Q,a)$ irá levar a um estado que corresponde ao subconjunto dos estados que podem ser alcançados ao se processar a letra a a partir de todo estado do conjunto associado com Q. Supondo que, ao terminar de processar um string x, o estado corrente é R. Esse estado corresponde ao conjunto de estados que estariam ativos ao se processar o string x no AFND A. O string x deve ser aceito por A^d sse algum daqueles estados pertence a F e portanto R deve pertencer a F^d sse algum dos estados a ele relacionados pertence a F.

Tomemos como exemplo o AFND $A =< \{a,b\}, \{r,s\}, \{r,s\}, \delta, \{s\} >$ cuja função de transição é dada pelo diagrama da Figura 4.10a. O conjunto de estados de A^d é dado então por $\rho(\{r,s\}) = \{\emptyset, \{r\}, \{s\}, \{r,s\}\}$. O estado inicial

é o subconjunto formado por todos os estados iniciais de A, ou seja $S_0^d = \{r, s\}$. O conjunto de estados finais é formado por todos os subconjuntos de S que possuem pelo menos um estado final de A, ou seja, $F^d = \{\{s\}, \{r, s\}\}$. E a função δ^d é determinada por:

$$\begin{array}{ll} \delta^d(\emptyset, a) = \emptyset & \delta^d(\emptyset, b) = \emptyset \\ \delta^d(\{r\}, a) = \delta(r, a) = \{r, s\} & \delta^d(\{r\}, b) = \delta(r, b) = \{s\} \\ \delta^d(\{s\}, a) = \delta(s, a) = \{r\} & \delta^d(\{s\}, b) = \delta(s, b) = \emptyset \\ \delta^d(\{r, s\}, a) = \delta(r, a) \cup \delta(s, a) = \{r, s\} & \delta^d(\{r, s\}, b) = \delta(r, b) \cup \delta(s, b) = \{s\} \end{array}$$

Obtendo-se o diagrama da Figura 4.10b.

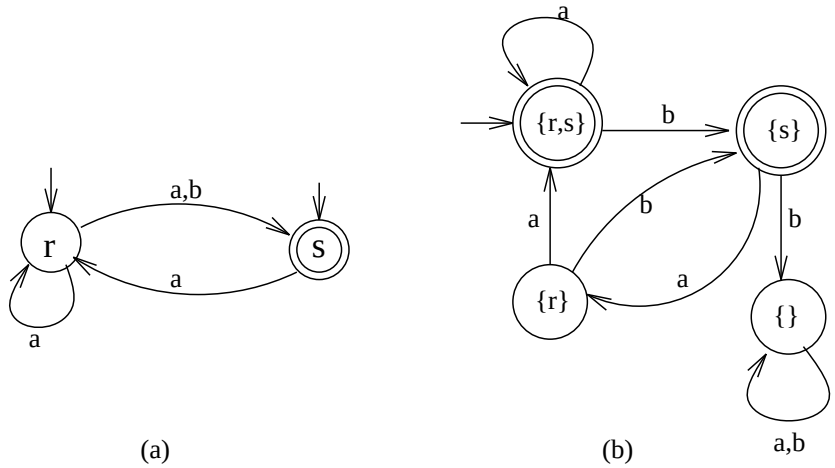


Figura 4.10: AFND com AFD correspondente

É importante notar que $|S^d| = 2^{|S|}$. Ou seja, sendo o número de estados de S^d igual ao número de subconjuntos de S, $|S^d|$ cresce exponencialmente em relação ao tamanho do AFND original. Porém, na maioria das vezes, muitos dos estados de S^d são inacessíveis e podem ser desconsiderados na obtenção de A^d. Para isso, deve-se iniciar o processo a partir de S_0^d e considerar apenas os estados alcançáveis a partir dele. Vamos, por exemplo, achar o AFD correspondente ao AFND da Figura 4.9. Inicialmente, teríamos $S_0^d = \{S_0, q_0\}$. A partir daí calculamos:

$$\begin{array}{ll} \delta(\{S_0, q_0\}, a) = \{S_1, S_2\} & \delta(\{S_0, q_0\}, b) = \{S_3\} \\ \delta(\{S_0, q_0\}, c) = \{S_2\} & \delta(\{S_0, q_0\}, d) = \{S_2\} \end{array}$$

Surgiram três novos estados alcançáveis a partir de $\{S_0, q_0\}$. Vamos então calcular o valor de δ para estes novos estados.

$$\delta(\{S_1,S_2\},a) = \{\ \} \quad \delta(\{S_1,S_2\},b) = \{S_3\}$$
$$\delta(\{S_1,S_2\},c) = \{S_2\} \quad \delta(\{S_1,S_2\},d) = \{S_2\}$$

$$\delta(\{S_3\},a) = \{\ \} \quad \delta(\{S_3\},b) = \{S_1,S_2\}$$
$$\delta(\{S_3\},c) = \{\ \} \quad \delta(\{S_3\},d) = \{\ \}$$

$$\delta(\{S_2\},a) = \{\ \} \quad \delta(\{S_2\},b) = \{\ \}$$
$$\delta(\{S_2\},c) = \{\ \} \quad \delta(\{S_2\},d) = \{S_2\}$$

Achados os valores de δ para esses estados apareceu um único novo estado que é $\{\}$ e que sabemos, para qualquer letra a do alfabeto mapeia $\delta(\{\},a) = \{\}$. Assim, não existem outros estados alcançáveis e podemos limitar os estados de A^d a esses 5 estados alcançáveis.

O diagrama correspondente é dado na Figura 4.11.

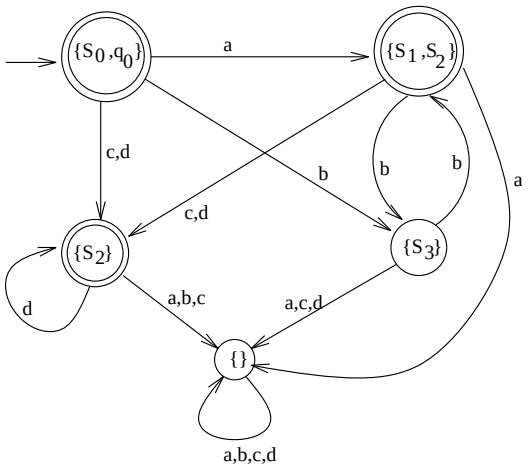


Figura 4.11: AFD correspondente ao AFND da Figura 4.9

Para mostrar que A e A^d são realmente equivalentes, ou seja, que reconhecem a mesma linguagem, vamos inicialmente mostrar por indução que

$$(\forall Q \in S^d)(\forall x \in \Sigma^*)\ \ \overline{\delta^d}(Q,x) = \bigcup_{q \in Q} \overline{\delta}(q,x) \tag{4.2}$$

Base: tomamos $|x| = 0$:

$$\overline{\delta^d}(Q,\lambda) = Q = \bigcup_{q \in Q} \{q\} = \bigcup_{q \in Q} \overline{\delta} q,\lambda)$$

Passo: supondo que a hipótese vale para $|x| = n$, vamos tomar o string ax, $a \in \Sigma$. Temos:

$$\begin{aligned}
\overline{\delta^d}(Q, ax) = &\quad \text{(pela definição de } \overline{\delta}) \\
\overline{\delta^d}(\delta^d(Q, a), x) = &\quad \text{(pela Hipótese de indução)} \\
\bigcup_{q \in \delta^d(q,a)} \overline{\delta}(q, x) = &\quad \text{(pela definição de } \delta^d) \\
\bigcup_{q \in Q} (\bigcup_{p \in \delta(q,a)} \overline{\delta}(p, x)) = &\quad \text{(pela definição de } \overline{\delta}) \\
\bigcup_{q \in Q} \overline{\delta}(q, ax) &\quad \text{C.Q.D.}
\end{aligned}$$

Teorema 4.2 *Dado o AFND* $A = < \Sigma, S, S_0, \delta, F >$, $A^d \equiv A$.

Prova: Dado $x \in \Sigma^*$, temos:

$$\begin{aligned}
x \in L(A) \Leftrightarrow &\quad \text{(pela definição de } L(A)) \\
(\bigcup_{q \in S_0} \overline{\delta}(q, x)) \cap F \neq \emptyset \Leftrightarrow &\quad \text{(pela definição de } A^d) \\
(\bigcup_{q \in S_0} \overline{\delta}(q, x)) \in F^d \Leftrightarrow &\quad \text{(por (4.2) e pela definição de } S_0^d) \\
\overline{\delta^d}(S_0^d, x) \in F^d \Leftrightarrow &\quad \text{(pela definição de } L(A^d)) \\
x \in L(A^d) &\quad \text{C.Q.D.}
\end{aligned}$$

4.4 Implementação

Nessa seção nossa implementação ira permitir que se representem AFND's com ou sem transições λ. Para isso, a estrutura que define um AF foi alterada no arquivo "afd.h". O novo arquivo é dado mostrado a seguir.

```
/*********************************************************
                          afd.h

Interface para o tipo de dado AUTOFIN, que representa um
um automato finito deterministico

Data: 2/6/98
Autor: Marcio Delamaro
*********************************************************/

#include    "set32.h"
#include    "alfabeto.h"



#define    MAXSTATE    32   /* numero maximo de estados */

#define    LABELSIZE    50  /* tamanho maximo do nome de um estado */
```

```
#define    INITIAL    1     /* indicacao de que um estado eh inicial */

#define    FINAL      2     /* indicacao de que um estado eh final */

typedef struct STATE {      /* armazena um estado */
    char    label[LABELSIZE+1];
    int    type;
    } STATE;

typedef struct AUTOFIN {     /* armazena um AFD */
    ALFABETO      *alfa;      /* apontador para o alfabeto */
    int           nstates;   /* numero de estados */
    int           alfalen;   /* cardinalidade do alfabeto */
    unsigned char eafnd:1;   /* indica se eh um AFND */
    unsigned char temlambda:1; /* indica se possui transicoes lambda */
    STATE         states[MAXSTATE]; /* tabela de estados */
    SET32         *transitions;     /* tabela de transicoes */
    SET32         initial;    /* numero do estado inicial */
    SET32         final;      /* conjunto de estados finais */
    } AUTOFIN;

AUTOFIN    *AFCria(char *sigma);
AUTOFIN    *AFConecta(AUTOFIN *a);
AUTOFIN    *AFReduz(AUTOFIN *a);
int         AFAddEstado(AUTOFIN *a, char *label, int final);
int         AFAddTransi(AUTOFIN *a, char *state1, char *state2, char *letra);
SET32     AFFechoL(AUTOFIN *a, SET32 x);
SET32       AFDelta(AUTOFIN *a, int state, char *letra);
SET32       AFDeltaBarra(AUTOFIN *a,SET32 current, char *palavra,char *caminho);
int         AFReconhece(AUTOFIN *a, char *palavra, char *caminho);
void        AFPrint(AUTOFIN *a, FILE *fp);
void        AFDestroi(AUTOFIN *a);
```

A principal mudança é que não se tem mais um único estado inicial. A variável "*initial*" não é mais representada por um inteiro e sim por um conjunto (tipo $SET32$). Da mesma forma, cada posição da tabela de transições é também um conjunto de estados e por isso "*transitions*" é um apontador para $SET32$ e não mais para um valor inteiro. Alem disso, ao criar-se o AF, adiciona-se o string "LAMBDA" como primeira letra do alfabeto para que transições λ sejam tratadas de forma uniforme. Ao adiconar-se então uma transição vazia deve-se utilizar esse string no lugar da letra que rotula a transição.

Tais mudanças determinam alterações sensíveis no código implementado. Fica por conta do leitor verificar tais alterações. Fica por conta do leitor também a criação das funções para retirar transições λ e para transformar um AFND num AFD. As interfaces de tais funções são:

```
/*-------------------------------------------------------------
              AFNoLambda
Cria um novo AFND que nao possui transicoes lambda
Parametros:
AUTOFIN *a: automato finito a transformar

Retorna:
AUTOFIN *       automato sem trans. lambda equivalente ao AUTOFIN a
                NULL em caso de erro
```

```
------------------------------------------------------------*/
AUTOFIN *AFNoLambda(AUTOFIN *a)
{
}
/*------------------------------------------------------------
              AFDeterministico
Cria um novo AFD que eh deterministico
Parametros:
AUTOFIN *a: automato finito a transformar

Retorna:
AUTOFIN *       automato deterministicoequivalente ao AUTOFIN a
                NULL em caso de erro
------------------------------------------------------------*/
AUTOFIN *AFDeterministico(AUTOFIN *a)
{
}
```

O código alterado do arquivo "afd.c" é dado a seguir.

```
/*********************************************************
                          afd.c

Implementacao do tipo de dado AUTOFIN, que representa um
um automato finito deterministico

Data: 23/3/98
Autor: Marcio Delamaro
*********************************************************/

#include    <stdio.h>
#include    <string.h>
#include    <ctype.h>
#include    <malloc.h>
#include    "afd.h"

#define        skip_blank(x)    while(isspace(*x))x++;

SET32 aux_set;

int     pegaestado(AUTOFIN *, char *);
void    nome_classe(AUTOFIN *, SET32 , char *);



/*------------------------------------------------------------
            AFCria
Cria a estrututa re um automato finito, vazio sem nenhum
estado ou transicao

Parametros:
   char *sigma:    alfabeto que compoe as letras do automato

Retorna:
   AUTOFIN * :        endereco do automato criado
------------------------------------------------------------*/
AUTOFIN    *AFCria(char *sigma)
{
AUTOFIN *p;
char *s;
```

```
   s = (char *) malloc(8 + strlen(sigma));
   if ( s == NULL)
return NULL;
   sprintf(s, "LAMBDA ");
   strcat(s, sigma);
   p = (AUTOFIN *) malloc(sizeof(AUTOFIN));
   if (p == NULL)
        return NULL;
   memset(p, 0, sizeof(*p));
   p->alfa = ALFACria(s);
   if (p->alfa == NULL)
   {
       free(p);
       return NULL;
   }
   p->alfalen = ALFACard(p->alfa);
   return p;
}

/*-------------------------------------------------------
            AFAddEstado
Adiciona um stado "vazio" ao AF, isto eh, um estado sem nenhuma
transicao

Parametros:
    char *label      Nome do estado
    int final        Flag indicando se estado eh fina/inicial

Retorno:
    < 0 se houve alguma falha
-------------------------------------------------------------*/
int    AFAddEstado(AUTOFIN *a, char *label, int final)
{
STATE p;
int i;

   if ( a->nstates >= MAXSTATE)
     return -1;
   if (pegaestado(a,label) >= 0)
     return -1;

   p.type = final;
   strncpy(p.label, label, LABELSIZE);
   if (final & INITIAL)
   {
      if (a->initial)
         a->eafnd = 1;
      a->initial = SET32Set(a->initial, a->nstates, 1);
   }
   if (final & FINAL)
     a->final = SET32Set(a->final, a->nstates, 1);
   a->states[a->nstates++] = p;
   if (a->transitions != NULL)    /* resize a->transitions se jah existe */
   {
       a->transitions = realloc(a->transitions, a->alfalen * a->nstates *
                         sizeof(*a->transitions));
       memset(&a->transitions[(a->nstates-1)*a->alfalen], 0,
                    a->alfalen * sizeof(*a->transitions));
   }
   return 0;
}
```

```
/*-------------------------------------------------------
                AFAddTransi
Adiciona um atransicao a um estado

Parametros:
    AUTOFIN a:      O automato
    char *state1    O nome do estado origem
    char *state2    O nome do estado destino
    char *letra     O simbolo do alfabeto correspondente a transicao
-------------------------------------------------------------*/

int    AFAddTransi(AUTOFIN *a, char *state1, char *state2,
                   char *letra)
{
int i, j, k, n;

   if ( a->transitions == NULL )
   {
       a->transitions = malloc(k = a->alfalen * a->nstates *
                        sizeof(*a->transitions));
       if (a->transitions == NULL)
          return -1;
       memset(a->transitions, 0, k);
   }
   i = pegaestado(a, state1); /* descobre o numero do estado origem */
   k = pegaestado(a, state2); /* descobre o numero do estado destino */
   j = ALFAOrdem(a->alfa, letra); /* descobre o numero da letra do alfabeto */

   if (i < 0 || k < 0 || j < 0)
       return -1;
   n = a->transitions[i*a->alfalen+j];
   a->transitions[i*a->alfalen+j] =
SET32Set(a->transitions[i*a->alfalen+j], k, 1);
   if ( n != a->transitions[i*a->alfalen+j])
a->eafnd = 1;
   if (j = 0)
        a->temlambda = a->eafnd = 1;
   return 0;
}


/*-------------------------------------------------------
                AFFechoL
Calcula o fecho lambda de um conjunto de estados

Parametros:
    AUTOFIN a:      O automato
    SET32 x:        Conjunto de estados dos quais se deseja
                    o fecho

Retorno:
    SET32           conjunto de estados correspondente ao
                    fecho lambda
-------------------------------------------------------------*/
SET32 AFFechoL(AUTOFIN *a, SET32 x)
{
int s;

   if (a->transitions == NULL)
return x;
   aux_set = x;
   for (s = SET32Next(x, 0); s >= 0; s = SET32Next(x, s+1))
```

```
   {
slt(a, s);
   }
   return aux_set;
}

slt(AUTOFIN *a, int k)
{
int j;
SET32 s;

   s = a->transitions[k*a->alfalen+0];
   for (j = SET32Next(s, 0); j >= 0; j = SET32Next(s, j+1))
   {
if ( ! SET32Get(aux_set, j))
{
   aux_set = SET32Set(aux_set, j, 1);
   slt(a, j);
}
   }
}



/*-------------------------------------------------------
           AFDelta
Essa funcao retorna o valor da funcao de transicao para um
estado e uma letra.

Parametros:
AUTOFIN *a         automato finito
int     *state     estado
char    *letra     letra para dalcular delta(estado,letra)

Retorna:
SET32               numero do proximo estado
-------------------------------------------------------------*/
SET32    AFDelta(AUTOFIN *a, int state, char *letra)
{
int j;

   if ( a->transitions == NULL)
       return 0;
   j = ALFAOrdem(a->alfa, letra);
   if (state < 0 || j < 0)
       return 0;
   return a->transitions[state*a->alfalen+j];
}


/*-------------------------------------------------------
           AFDeltaBarra
Essa funcao retorna o valor da funcao de transicao Extendida
para um estado e uma cadeia.

Parametros:
AUTOFIN *a         automato finito
SET32    current    estados iniciais correntes
char    *letra     letra para dalcular delta(estado,letra)
char    *caminho   buffer para colocar proximo estado
```

```
Retorna:
SET32               conjunto de estados resultantes
------------------------------------------------------------*/
SET32    AFDeltaBarra(AUTOFIN *a, SET32 current, char *palavra, char *caminho)
{
int i;
static char buf[100];
SET32 k;

   if (caminho != NULL)
   {
       sprint_estados(a,caminho,current);
       strcat(caminho,"\n");
   }
    skip_blank(palavra);
    i = sscanf(palavra, "%s", &buf); /* pega primeira letra */

    current = AFFechoL(a, current);
    if (i != 1)     /* terminaram as letras */
        return current;      /* retorna estados correntes */
    k = 0;
    for ( i = SET32Next(current, 0); i >= 0;
i = SET32Next(current, i+1) )
    {
        k = SET32Union(k, AFDelta(a, i, buf));
    }
    k = AFFechoL(a, k);

    palavra += strlen(buf);
    return AFDeltaBarra(a, k, palavra, caminho);
}



/*------------------------------------------------------------
            AFReconhece
Essa funcao tenta reconhecer um string

Parametros:
AUTOFIN *a:         automato finito
char    *palavra    string a ser analisado (letras separadas por brancos)
char    caminho     sequencia de estados percorridos

Retorna:
int                 1 reconheceu
                    0 nao reconheceu
------------------------------------------------------------*/
int    AFReconhece(AUTOFIN *a, char *palavra, char *caminho)
{
int    next;

   *caminho = '\0';
   if (a->initial == 0 )
      return 0;
   next = AFDeltaBarra(a, a->initial, palavra, caminho);
   next = SET32Inter(a->final, next);
   return next != 0;
}

/*------------------------------------------------------------
               AFDestroi
Libera memoria de um AF
```

```
------------------------------------------------------------*/
void    AFDestroi(AUTOFIN *a)
{
   free(a->transitions);
   ALFADestroi(a->alfa);
   free(a);
}


/*-------------------------------------------------------
              AFConecta
Cria um novo AFD que eh conexo. Nao requer AFD
Parametros:
AUTOFIN *a: automato finito a transformar

Retorna:
AUTOFIN *acon   automato conexo equivalente ao AUTOFIN a
                NULL em caso de erro
------------------------------------------------------------*/
AUTOFIN *AFConecta(AUTOFIN *a)
{
SET32 ci;        /* conjunto C_i */
SET32 new;      /* novos estados incluidos no ultimo C_i*/
SET32 next;     /* proximos estados a serem incluidos */
SET32 k;
int i, j, l, n;
AUTOFIN *acon;

   if ( a->transitions == NULL)
return NULL;
   ci = a->initial;
   new = ci;
   while ( new != 0 )   /* enquanto novo estado foi incluido */
   {
next = 0;
        for (i = SET32Next(new,0); i >= 0; i = SET32Next(new, i+1))
{
            for ( j = 0; j < a->alfalen; j++)
            {
/* k <-- estado destino da transicao do estado
                   i com a j-esima letra */
                k = a->transitions[i*a->alfalen+j];
                k = SET32Inter(~ci,k);
                next = SET32Union(next, k);
             }
         }
         new = next;
         ci = SET32Union(ci,new);
    }

    /* ci contem os estados conexos. elimina os demais */
    acon = AFCria("a");
    if (acon == NULL)
          return NULL;
    ALFADestroi(acon->alfa);
    acon->alfa = ALFADup(a->alfa);
    acon->alfalen = a->alfalen;
    for (i = SET32Next(ci,0); i >= 0; i = SET32Next(ci, i+1))
    {  /* adiciona os estados de ci */
        l = (i == a->initial)? INITIAL: 0;
l += (SET32Get(a->final,i))? FINAL: 0;
        if (AFAddEstado(acon, a->states[i].label, l) < 0)
```

```
        {
           r0:
           AFDestroi(acon);
           return NULL;
        }
    }

    for (i = SET32Next(ci,0); i >= 0; i = SET32Next(ci, i+1))
    {  /* adiciona as transicoes */
       for (j = 0; j < a->alfalen; j++)
       {
          l = a->transitions[i*a->alfalen+j];
          for (n = SET32Next(l,0); n >= 0; n = SET32Next(l, n+1) )
          {
             if ( AFAddTransi(acon, a->states[i].label,
               a->states[n].label, ALFALetra(a->alfa,j)) < 0 )
              {
                 goto r0;
              }
          }
        }
    }
    return acon;
}


/*-------------------------------------------------------------
               AFReduz
Cria um novo AFD que eh reduzido. Requer AFD
Parametros:
AUTOFIN *a: automato finito a transformar

Retorna:
AUTOFIN *acon   automato reduzido equivalente ao AUTOFIN a
                NULL em caso de erro
-------------------------------------------------------------*/
AUTOFIN *AFReduz(AUTOFIN *a)
{
int i, j, k, l, m, nparti, np2;
SET32 *particao, *p2, *paux, x;
static char buf1[LABELSIZE+1], buf2[LABELSIZE+1];
AUTOFIN *acon;

   if (a->eafnd || a->transitions == NULL)
       return NULL;

   /* cada particao[i] eh uma classe de equivalencia.
      o numero maximo eh o numero de estados do AFD */
   particao = malloc(a->nstates*sizeof(*particao));
   if (particao == NULL)
       return NULL;

   p2 = malloc(a->nstates*sizeof(*particao));
   if (p2 == NULL)
   {
       free(particao);
       return NULL;
   }

   particao[0] = a->final; /* conjunto F */
   x = 0;
   for (i = 0; i < a->nstates; i++)
```

```
   x = SET32Set(x,i,1);
particao[1] = SET32Inter(x, ~ a->final); /* conjunto S - F */
nparti = 2; /* numero inicial de particoes */

do {  /* calcula as classes de equivalencia */
 np2 = 0;  /* zera proxima particao */
 for (i = 0; i < a->nstates; i++)
     p2[i] = 0;

 for (i = 0; i < nparti; i++)
 {
   x = particao[i];
   for (j = SET32Next(x,0); j >= 0; j = SET32Next(x,j+1))
   {
      p2[np2] = SET32Set(0, j, 1);
      x = SET32Set(x, j, 0);
      for (k = SET32Next(x,j+1); k >= 0; k = SET32Next(x,k+1))
      {
         if (AFDeltaCompara(a, particao, nparti,j, k))
         {    /* estados j e k devem ficar juntos */
             p2[np2] = SET32Set(p2[np2], k, 1);
             x = SET32Set(x, k, 0);
         }
      }
      np2++;
   }
 }
  paux = p2;
  p2 = particao;
  particao = paux;
  k = np2;
  np2 = nparti;
  nparti = k;
} while (np2 < nparti);

 acon = AFCria("a");
 if (acon == NULL)
       return NULL;
 ALFADestroi(acon->alfa);
 acon->alfa = ALFADup(a->alfa);
 acon->alfalen = a->alfalen;

for (i = 0; i < nparti; i++)
{   /* insere um estado para cada particao calculada */
   nome_classe(a,particao[i], buf1);
   k = ( SET32Get(particao[i],a->initial) )? INITIAL: 0;
   k += (SET32Inter(particao[i],a->final) )? FINAL: 0;
   if (AFAddEstado(acon, buf1, k) < 0)
   {
      r0:
      free(particao);
      free(p2);
      AFDestroi(acon);
      return NULL;
   }
}
for (i = 0; i < nparti; i++)
{   /* insere as transicoes */
   k = SET32Next(particao[i], 0); /* pega o primeiro estado da particao */
   for (j = 0; j < a->alfalen; j++)
   {
      l = a->transitions[k*a->alfalen+j];
```

```
         for (m = 0; m < nparti; m++)
            if ( SET32Get(particao[m], l) )
               break;
         nome_classe(a,particao[i],buf1);
         nome_classe(a,particao[m],buf2);
         if (AFAddTransi(acon, buf1, buf2, ALFALetra(a->alfa, j) ) < 0)
              goto r0;
      }
   }

   free(particao);
   free(p2);
   return acon;
}

/*-------------------------------------------------------------------
                          nome_classe
Dado um conjunto de estados, escolhe um nome para esses conjunto
-------------------------------------------------------------------*/
void nome_classe(AUTOFIN *a, SET32 x, char *s)
{
int k;

   s[0] = '\0';
   k = SET32Next(x, 0); /* pega primeiro elemento do conjunto */
   if (k < 0)
       return;
   sprintf(s, "[%s]", a->states[k].label);
}


/*-------------------------------------------------------------------
                          AFDeltaCompara
Verifica se dois estados devem permanecer na mesma classe de
equivalencia.
Parametros:
AUTOFIN *a:       o AFD
STE32 particao[]  classes de equivalencia
int npart         numero de classes de equivalencia
int s1,s2         os estados a comparar
Retorna:
int              1 se devem ficar
                 0 se nao devem ficar
-------------------------------------------------------------------*/
int AFDeltaCompara(AUTOFIN *a, SET32 particao[], int npart, int s1, int s2)
{
int i, j, k, l, m;

   for (i = 0; i < a->alfalen; i++)
   {
      j = SET32Next(a->transitions[a->alfalen*s1+i],0);
      k = SET32Next(a->transitions[a->alfalen*s2+i],0);
      for (l = 0; l < npart; l++)
      {
         if ( SET32Get(particao[l], j) )
            if (! SET32Get(particao[l],k) )
                return 0;
      }
   }
   return 1;
}
```

```
/*-------------------------------------------------------------------
                        AFPrint
Imprime o automato finito no mesmo formato do arquivo de entrada

Parametros:
AUTOFIN *a:     o automato
FILE *fp:    arquivo onde deve ser escrito
-------------------------------------------------------------------*/
void AFPrint(AUTOFIN *a, FILE *fp)
{
static char buf[1024];
int i, j, k;
SET32 next;

   for (i = 1; i < a->alfalen; i++)
fprintf(fp, "%s ", ALFALetra(a->alfa,i));

   for (i = 0; i < a->nstates; i++)
   {
fprintf(fp, "\nESTADO %s %s %s", a->states[i].label,
SET32Get(a->initial, i)? "inicial" : "",
SET32Get(a->final, i)? "final": "");
   }

   if ( a->transitions != NULL)
     for (i = 0; i < a->nstates; i++)
     {
for (j = next = 0; j < a->alfalen; j++)
   next = SET32Union(next, a->transitions[i*a->alfalen+j]);
for (j = SET32Next(next, 0); j >= 0; j = SET32Next(next, j+1))
{
   buf[0] = '\0';
   for (k = 0; k < a->alfalen; k++)
   {
if ( SET32Get(a->transitions[i*a->alfalen+k], j))
{
   strcat(buf, ALFALetra(a->alfa, k));
   strcat(buf, " ");
}
   }
   if ( buf[0] != '\0')
   {
fprintf(fp, "\nTRANSICAO %s %s %s", a->states[i].label,
a->states[j].label, buf);
   }
}
     }
   fprintf(fp, "\n");
}


/*-------------------------------------------------------------
              AFNoLambda
Cria um novo AFND que nao possui transicoes lambda
Parametros:
AUTOFIN *a: automato finito a transformar

Retorna:
AUTOFIN *       automato sem trans. lambda equivalente ao AUTOFIN a
                NULL em caso de erro
```

```
-------------------------------------------------------------*/
AUTOFIN *AFNoLambda(AUTOFIN *a)
{
}
/*-------------------------------------------------------------
              AFDeterministico
Cria um novo AFD que eh deterministico
Parametros:
AUTOFIN *a: automato finito a transformar

Retorna:
AUTOFIN *       automato deterministicoequivalente ao AUTOFIN a
                NULL em caso de erro
-------------------------------------------------------------*/
AUTOFIN *AFDeterministico(AUTOFIN *a)
{
}

/*-------------------------------------------------------------
Dado o nome de um estado, retorna seu numero
-------------------------------------------------------------*/
int pegaestado(AUTOFIN *a, char *s)
{
int i;

   for (i = 0; i < a->nstates; i++)
    if (strcmp(a->states[i].label, s) == 0)
       return i;
   return -1;
}


/*-------------------------------------------------------------
Pega o nome de um conjunto de estados
-------------------------------------------------------------*/
int sprint_estados(AUTOFIN *a, char *s, SET32 x)
{
int i;

   for (i = SET32Next(x,0); i >= 0; i = SET32Next(x, i+1))
   {
      strcat(s, a->states[i].label);
      strcat(s, " ");
   }
}
```

Capítulo 5

Expressões Regulares

Vimos até agora como representar linguagens através de "máquinas" capazes de identificar cadeias que pertencem ou que não pertencem à linguagem. Neste capítulo iremos estudar uma segunda forma de representar linguagens, chamada de Expressão Regular (ER).

Ao contrário dos Autômatos Finitos, as Expressões Regulares não possuem um caráter de reconhecedor, mas sim de gerador de cadeias. Elas permitem uma melhor "visualização" do tipo de linguagem sendo definida. Por outro lado, a construção de uma Expressão Regular para determinada linguagem pode ser mais difícil do que a construção de um Autômato Finito. Assim, dependendo do tipo de aplicação desejada, pode-se optar por uma ou outra representação. O importante é notar que existe uma equivalência, em relação ao tipo de linguagens que podem ser representadas, entre Autômatos Finitos e Expressões Regulares. Essa equivalência será também abordada neste capítulo. Veremos como um Autômato Finito Determinístico pode ser transformado numa Expressão Regular correspondente e vice-versa.

5.1 Definições Básicas

A abordagem usada numa ER é a de utilizarem-se linguagens simples que são combinadas através de operadores apropriados, para que linguagens mais complexas sejam definidas. Tomando, por exemplo, o alfabeto $\Sigma = \{a, b, c\}$, as linguagens mais simples que se podem definir sobre Σ são:

$\emptyset$ - o conjunto vazio

$\{\lambda\}$ - a linguagem que possui um único string, o string vazio

$\{a\}$, $\{b\}$, $\{c\}$ - os conjuntos unitários compostos por strings formados por uma única letra do alfabeto

A partir dessa linguagens "básicas", utilizamos alguns operadores para definir linguagens mais elaboradas. O primeiro desses operadores é a união ($\cup$), já

vista no Capítulo 1. Recordando, dadas as linguagens L_1 e L_2, define-se:

$$L_1 \cup L_2 = \{x \in \Sigma^* \mid x \in L_1 \vee x \in L_2\}$$

Tomando $\Sigma = \{a, b, c\}$ e as linguagens básicas descritas acima, podemos através do operador de união definir algumas novas linguagens como:

- $\emptyset \cup \{\lambda\} = \{\lambda\}$
- $\{\lambda\} \cup \{\lambda\} = \{\lambda\}$
- $\{a\} \cup \{\lambda\} = \{a, \lambda\}$
- $\{b\} \cup \{a\} \cup \{\lambda\} = \{b, a, \lambda\}$

O segundo operador que utilizaremos é a concatenação $(\cdot)$ também já estudada anteriormente. Para relembrar:

$$L_1 \cdot L_2 = \{y \cdot z \in \Sigma^* \mid y \in L_1 \wedge z \in L_2\}$$

Utilizando as concatenação podemos definir novas linguagens como:

- $\{a\} \cdot \{b\} = \{ab\}$
- $\{a\} \cdot \{\lambda\} = \{a\}$
- $\{a\} \cdot (\{\lambda\} \cup \{b\}) = \{a\} \cdot \{\lambda, b\} = \{a, ab\}$
- $(\{a\} \cup \{b\}) \cdot (\{b\} \cup \{c\}) = \{ab, ac, bb, bc\}$

O terceiro e último operador que utilizaremos é o **Fecho de Kleene**, denotado pelo símbolo *. Sua definição é:

Definição 5.1 *Dada a linguagem L sobre o alfabeto Σ, definem-se:*
$L^0 = \{\lambda\}$
$L^1 = L$
$L^2 = L \cdot L$
$L^3 = L \cdot L^2$
e de forma geral:
$L^n = L \cdot L^{n-1}$

$$L^* = \bigcup_{i=0}^{\infty} L^i = L^0 \cup L^1 \cup L^2 \cup L^3 \cup ...$$

L^* é o Fecho de Kleene de L.

O que o fecho de Kleene representa é a repetição (concatenação) 0 ou mais vezes, dos elementos da linguagem. Por exemplo:

- $\{a\}^* = \{\lambda, a, aa, aaa, aaaa, ...\}$

- $\{\lambda\}^* = \{\lambda\}$
- $\{a\} \cdot \{b\}^* = \{a, ab, abb, abbb, ...\}$
- $(\{a\} \cup \{b\})^* = \{\lambda, a, b, aa, ab, ba, bb, aaa, aab, ...\}$

Todas as linguagens que podem ser representadas a partir dos conjuntos básicos e dos três operadores descritos acima são chamados de **Conjuntos Regulares** ou **Linguagens Regulares**. As expressões usadas para determinar tais conjuntos são as ER, que podem ser definidas mais precisamente como:

Definição 5.2 *Dado o alfabeto* $\Sigma = \{a_1, a_2, ..., a_n\}$, *uma Expressão Regular sobre* Σ *é definida como:*

- $\emptyset$ *é uma Expressão Regular que representa o conjunto vazio*
- ϵ *é uma Expressão Regular que representa o conjunto* $\{\lambda\}$
- a_1, a_2, ..., a_n *são Expressões Regulares que representam os conjuntos* $\{a_1\}$, $\{a_2\}$, ..., $\{a_n\}$ *respectivamente.*
- *dadas as Expressões Regulares* R_1 *e* R_2, *que representam os conjuntos regulares* L_1 *e* L_2 *respectivamente, então* $(R_1 \cdot R_2)$ *é uma Expressão Regular que representa o conjunto* $L_1 \cdot L_2$
- *dadas as Expressões Regulares* R_1 *e* R_2, *que representam os conjuntos regulares* L_1 *e* L_2 *respectivamente, então* $(R_1 \cup R_2)$ *é uma Expressão Regular que representa o conjunto* $L_1 \cup L_2$
- *dada a Expressão Regular* R, *que representa o conjunto regular* L, *então* $(R)^*$ *é uma Expressão Regular que representa o conjunto* L^*

Vejamos então algumas Expressões Regulares e os conjuntos a que elas se referem, considerando $\Sigma = \{a, b, c\}$

- $\epsilon = \{\lambda\}$
- $(b \cup c) = \{b\} \cup \{c\} = \{b, c\}$
- $(a \cdot (b \cup c)) = \{a\} \cdot (\{b\} \cup \{c\}) = \{ab, ac\}$
- $((a \cup \epsilon) \cdot (b)^*) = \{a, \lambda\} \cdot \{\lambda, b, bb, bbb, ...\} = \{\lambda, a, b, ab, bb, abb, bbb, ...\}$

Definindo-se uma precedência entre os operadores podemos eliminar diversos parênteses das ER's. Utilizaremos o Fecho de Kleene como maior precedência, seguido da concatenação e por último a união. Assim, $(a \cdot b \cup c)$ deve ser interpretada como $((a \cdot b) \cup c)$ e $(a \cdot b^*)$ como $(a \cdot (b)^*)$.

Vamos ver um exemplo de uma linguagen definida através de um ER. Tomemos a linguagem definida pelo AFD da Figura 2.9 que representa o conjunto

de constantes de ponto flutuante válidas numa determinada linguagem de programação. Para facilitar, vamos dividir essa ER em diversas partes. Inicialmente dividimos a ER R em duas parte: a primeira que chamaremos de R_1 é a que vem antes do ponto decimal e a segunda, R_2, é composta pelo ponto decimal mais o que vem após o ponto decimal. Teríamos então:

$$R = R_1 \cdot (\epsilon \cup R_2)$$

Note que uma constante que possua somente a parte inteira á válida e por isso utilizou-se $(\epsilon \cup R_2)$ para indicar que após R_1 podemos ter R_2 ou podemos não ter nada. R_1, como vimos no AFD, pode ser composta por um sinal seguido de um número maior que ou igual a 1 de dígitos. Portanto, podemos definir R_1 como:

$$R_1 = (+ \cup - \cup \epsilon) \cdot \mathbf{d} \cdot \mathbf{d}^*$$

Após o ponto decimal iremos dividir a expressão também em duas partes: uma antes do símbolo **E** e a outra a partir do **E**. Note que ambas as partes podem ou não ocorrer, assim definimos R_2 da seguinte maneira:

$$R_2 = (\epsilon \cup R_3) \cdot (\epsilon \cup R_4)$$

E R_3 e R_4 definimos como:

$$R_3 = \text{.} \cdot \mathbf{d}^*$$

$$R_4 = \mathbf{E} \cdot (+ \cup - \cup \epsilon) \cdot \mathbf{d} \cdot \mathbf{d}^*$$

Finalmente, podemos construir R como:

$$R = (+ \cup - \cup \epsilon) \cdot \mathbf{d} \cdot \mathbf{d}^* \cdot (\epsilon \cup (\epsilon \cup \text{.} \cdot \mathbf{d}^*) \cdot (\epsilon \cup \mathbf{E} \cdot (+ \cup - \cup \epsilon) \cdot \mathbf{d} \cdot \mathbf{d}^*))$$

Assim como qualquer expressão algébrica, as ER's podem ser manipuladas de acordo com algumas propriedades dos seus operadores. Algumas dessas propriedades são:

- $R \cup \emptyset = R$
- $R \cdot \epsilon = \epsilon \cdot R = R$
- $R \cdot \emptyset = \emptyset \cdot R = \emptyset$
- $R_1 \cup R_2 = R_2 \cup R_1$
- $R \cup R = R$
- $R_1 \cup (R_2 \cup R_3) = (R_1 \cup R_2) \cup R_3$
- $R_1 \cdot (R_2 \cdot R_3) = (R_1 \cdot R_2) \cdot R_3$
- $R_1 \cdot (R_2 \cup R_3) = (R_1 \cdot R_2) \cup (R_1 \cdot R_3)$
- $\epsilon^* = \epsilon$

- $\emptyset^* = \epsilon$
- $(R_1 \cup R_2)^* = (R_1^* \cup R_2^*)^* = (R_1^* \cdot R_2^*)^*$
- $(R_1^*)^* = R_1^*$

Além disso, existem casos em que

- $R \cup \epsilon \neq R$
- $R_1 \cdot R_2 \neq R_2 \cdot R_1$
- $R \cdot R \neq R$
- $R_1 \cup (R_2 \cdot R_3) \neq (R_1 \cup R_2) \cdot (R_1 \cup R_3)$
- $(R_1 \cdot R_2)^* \neq (R_1^* \cdot R_2^*)^*$
- $(R_1 \cdot R_2)^* \neq (R_1^* \cup R_2^*)^*$

5.2 Equivalência entre ER e AF

Qualquer linguagem definida através de um ER pode também ser representada através de um AFD ou AFND e vice-versa. Nesta seção iremos estudar essa equivalência, mostrando como transformar uma ER dada num AF e como transformar um AF numa ER.

5.2.1 Transformação de uma ER num AF

Para transformar uma ER num AF iremos utilizar o fato de que cada ER é composta por sub-expressões que são sucessivamente combinadas através dos operadores $\cup$, $\cdot$ e $*$, sempre partindo daquelas ER's básicas vistas anteriormente. Iremos então definir AF's que correspondem às ER básicas e a partir deles iremos construir AF's mais complexos e que representam as linguagens desejadas. Por exemplo, se quisermos achar um AF correspondente a $(a \cup b) \cdot c$, iremos construir AF's A_1 e A_2 que reconheçam as linguagens $\{a\}$ e $\{b\}$ respectivamente e iremos combiná-los, construindo um AF A_3 que reconhece $\{a, b\}$. Depois, definimos A_4 que reconhece $\{c\}$ e finalmente combinamos A_3 e A_4 num AF que reconhece $\{a, b\} \cdot \{c\}$.

Iniciamos mostrando como construir AFND's para cada uma das ER's básicas estudadas anteriormente. Temos então:

Definição 5.3 *Dado o alfabeto Σ, define-se o AFND $< \Sigma, \{S_0\}, \{S_0\}, \delta, \emptyset >$ onde δ é definida por*

$$(\forall a \in \Sigma) \quad \delta(S_0, a) = \emptyset$$

como o AFND que reconhece a linguagem $\emptyset$.

Definição 5.4 *Dado o alfabeto* Σ, *define-se o AFND* $< \Sigma, \{S_0\}, \{S_0\}, \delta, \{S_0\} >$ *onde* δ *é definida por*

$$(\forall a \in \Sigma) \quad \delta(S_0, a) = \emptyset$$

como o AFND que reconhece a linguagem $\{\lambda\}$.

Definição 5.5 *Dado o alfabeto* $\Sigma = \{a_1, a_2, ..., a_n\}$, *define-se o AFND* $< \Sigma, \{S_0, S_1\}, \{S_0\}, \delta, \{S_1\} >$ *onde* δ *é definida por*

$$\delta(S_0, a_i) = \{S_1\}$$

$$(\forall a \in \Sigma - \{a_i\}) \quad \delta(S_0, a) = \emptyset$$

$$(\forall a \in \Sigma) \quad \delta(S_1, a) = \emptyset$$

como o AFND que reconhece a linguagem $\{a_i\}$.

Essas definições nos dizem como criar os AFND's para cada uma das ER's básicas, cujos diagramas estão representados na Figura 5.1.

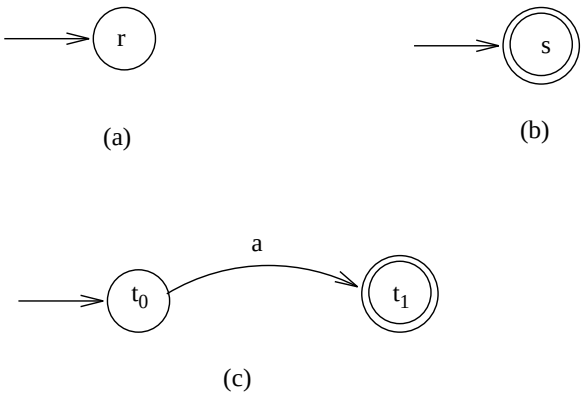


Figura 5.1: AFND's para $\emptyset$, $\{\lambda\}$ e $\{a\}$

Vamos agora supor que queremos representar através de um AF a linguagem definida pela ER $(R_1 \cup R_2)$, onde R_1 e R_2 são também ER's. Então, assumindo que existem AF's A_1 e A_2 tais que $L(A_1) = L(R_1)$ e $L(A_2) = L(R_2)$, iremos utilizar esses AF's para definir um AFND que reconhece a linguagem que desejamos: $L(R_1 \cup R_2) = L(A_1) \cup L(A_2)$.

Definição 5.6 *Dado o alfabeto* Σ*, as ER's* R_1 *e* R_2 *sobre* Σ *e os AFND's* $A_1 = < \Sigma, S_1, S_{01}, \delta_1, F_1 >$ *e* $A_2 = < \Sigma, S_2, S_{02}, \delta_2, F_2 >$ *tais que* $L(A_1) = L(R_1)$ *e* $L(A_2) = L(R_2)$*, define-se o AFND* $A^{\cup} = < \Sigma, S^{\cup}, S_0^{\cup}, \delta^{\cup}, F^{\cup} >$ *onde:*

$$S^{\cup} = S_1 \cup S_2$$

$$S_0^{\cup} = S_{01} \cup S_{02}$$

$$F^{\cup} = F_1 \cup F_2$$

$$(\forall s \in S_1)(\forall a \in \Sigma) \quad \delta^{\cup}(s, a) = \delta_1(s, a)$$

$$(\forall s \in S_2)(\forall a \in \Sigma) \quad \delta^{\cup}(s, a) = \delta_2(s, a)$$

O AFND $A^{\cup}$ reconhece a mesma linguagem definida pela ER $R_1 \cup R_2$. Ele é formado, simplesmente, pela união dos elementos de cada um dos AFND's A_1 e A_2. Tomemos como exemplo a ER $(a \cup b)$, sobre o alfabeto $\{a, b, c\}$. Partindo dos AFND's $A_1 = < \{a, b, c\}, \{r_0, r_1\}, \{r_0\}, \delta_1, \{r1\} >$ e $A_2 = < \{a, b, c\}, \{s_0, s_1\}, \{s_0\}, \delta_2, \{s1\} >$ cujas funções de transição estão representadas nas Figuras 5.2a e 5.2b, construímos, segundo a Definição 5.6, o AFND $A^{\cup} = < \{a, b, c\}, \{r_0, r_1, s_0, s_1\}, \{r_0, s_0\}, \delta^{\cup}, \{r1, s_1\} >$ cuja função de transição é representada na Figura 5.2c e que reconhece a mesma linguagem definida por $(a \cup b)$.

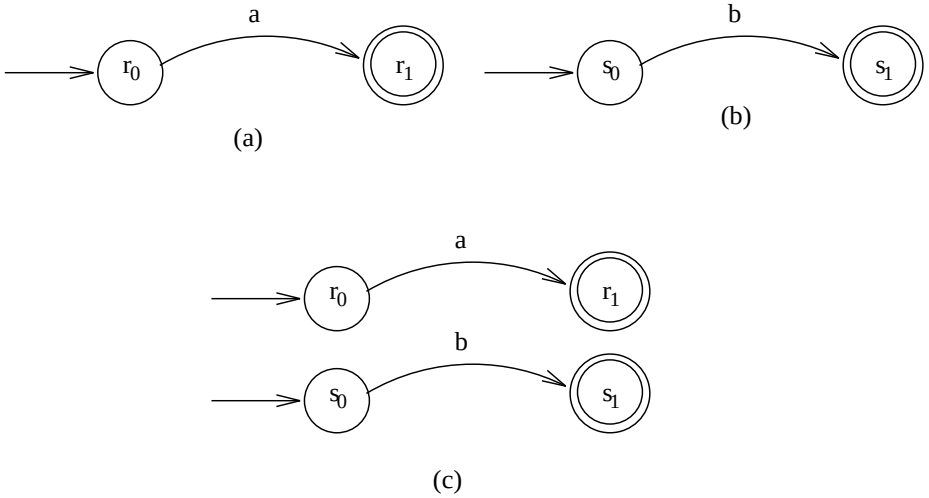


Figura 5.2: União de dois AFND's

Vamos definir agora como construir um AFND para $R_1 \cdot R_2$. A idéia nesse caso é colocar o segundo AFND "no final" do primeiro, de modo que o AFND resultante aceite todos os strings que atinjam os estados finais do primeiro, passem aos estados iniciais do segundo e atinjam os estados finais do segundo. Uma maneira simples de fazê-lo seria criar transições λ de cada estado final do primeiro AFND para cada estado inicial do segundo. Outra maneira, e que dispensa o uso de transições λ seria o seguinte:

Definição 5.7 *Dado o alfabeto Σ, as ER's R_1 e R_2 sobre Σ e os AFND's $A_1 = < \Sigma, S_1, S_{01}, \delta_1, F_1 >$ e $A_2 = < \Sigma, S_2, S_{02}, \delta_2, F_2 >$ tais que $L(A_1) = L(R_1)$ e $L(A_2) = L(R_2)$, define-se o AFND $A^\bullet = < \Sigma, S^\bullet, S_0^\bullet, \delta^\bullet, F^\bullet >$ onde:*

$$S^\bullet = S_1 \cup S_2$$

$$S_0^\bullet = S_{01}$$

$$F^\bullet = \begin{cases} F_2 & se\ \lambda \notin L(A_2) \\ F_1 \cup F_2 & se\ \lambda \in L(A_2) \end{cases}$$

$$(\forall s \in S_1 - F_1)(\forall a \in \Sigma) \quad \delta^\bullet(s,a) = \delta_1(s,a)$$

$$(\forall s \in F_1)(\forall a \in \Sigma) \quad \delta^\bullet(s,a) = \delta_1(s,a) \cup \bigcup_{t \in S_{02}} \delta_2(t,a)$$

$$(\forall s \in S_2)(\forall a \in \Sigma) \quad \delta^\bullet(s,a) = \delta_2(s,a)$$

Vamos então tomar o autômato da Figura 5.2c e $A_3 = < \{a,b,c\}, \{t_0, t_1\}, \{t_0\}, \delta_2, \{t_1\} >$, representado na Figura 5.3a para construir o AFND correspondente à ER $(a \cup b) \cdot c$. Teríamos $A^\bullet = < \{a,b,c\}, \{r_0, r_1, s_0, s_1, t_0, t_1\}, \{r_0, s_0\}, \delta^\bullet, \{t_1\} >$, com $\delta^\bullet$ representada na Figura 5.3b.

Note que neste caso o conjunto $F^\bullet$ é igual ao conjunto de estados finais do AFND A_3 porque $\lambda \notin L(A_3)$. Caso λ pertencesse a $L(A_3)$, então os estados finais do primeiro AFND também deveriam ser incluídos no conjunto $F^\bullet$ indicando que as cadeias aceitas pela primeiro AFND também devem ser aceitas por $A^\bullet$. Outro ponto a ser destacado é que os estados iniciais de A_3 não são estados iniciais de $A^\bullet$. Por isso, o estado t_0 passou a ser inacessível e poderia ser eliminado. Muitas vezes, os AFND obtidos a partir das definições não são os mais eficientes. Por outro lado, isso não deve ser um obstáculo, visto que conhecemos meios para minimizar autômatos e que podem ser empregados nesses casos.

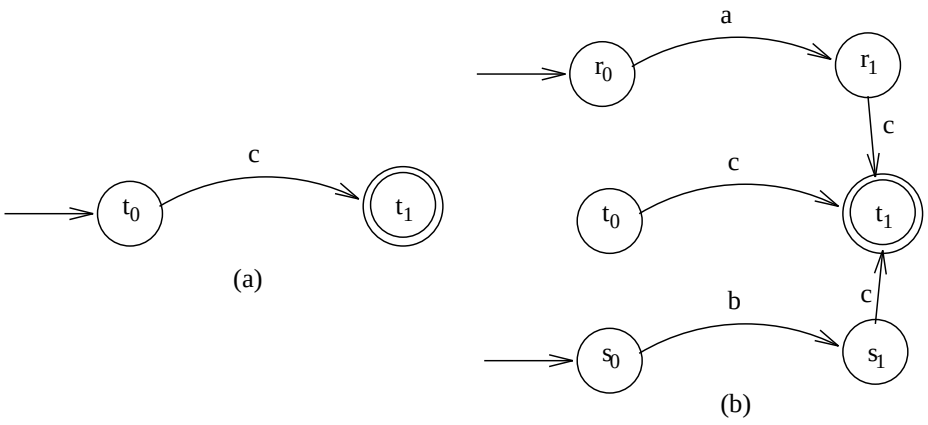


Figura 5.3: Concatenação de dois AFND's

E por último, vamos determinar como construir o AFND para R^*. Semelhante à concatenação, iremos, nesse caso, concatenar o AFND com ele mesmo:

Definição 5.8 *Dado o alfabeto Σ, a ER R sobre Σ e o AFND $A = <\Sigma, S, S_0, \delta, F>$ tais que $L(A) = L(R)$ e define-se o AFND $A^* = <\Sigma, S^*, S_0^*, \delta^*, F^*>$ onde:*

$$S^* = S \cup \{q_0\}$$

$$S_0^* = S_0 \cup \{q_0\}$$

$$F^* = F \cup \{q_0\}$$

$$(\forall a \in \Sigma) \quad \delta^*(q_0, a) = \emptyset$$

$$(\forall s \in F)(\forall a \in \Sigma) \quad \delta^*(s, a) = \delta(s, a) \cup \bigcup_{t \in S_0} \delta(t, a)$$

$$(\forall s \in S - F)(\forall a \in \Sigma) \quad \delta^*(s, a) = \delta(s, a)$$

Para calcular, por exemplo, $((a \cup b) \cdot c)^*$ a partir do AFND da Figura 5.3b, construímos $A^* = <\{a, b, c\}, \{r_0, r_1, s_0, s_1, t_0, t_1, q_0\}, \{r_0, s_0, q_0\}, \delta^*, \{t_1, q_0\}>$, com δ^* dada pelo diagrama da Figura 5.4. O estado q_0 é utilizado para reconhecer a cadeia vazia, que sempre faz parte do Fecho de Kleene de qualquer expressão.

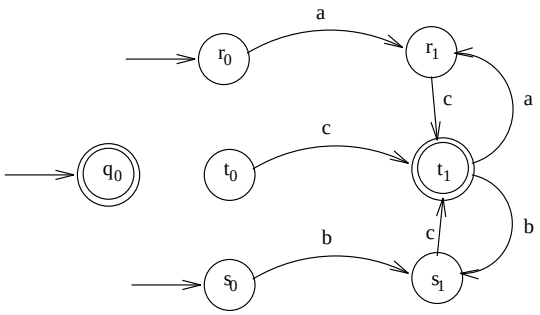


Figura 5.4: Fecho de Kleene de um AFND

5.2.2 Transformação de um AFND em uma ER

Para transformar um AFND numa ER equivalente iremos utilizar **Equações de Linguagens**. Uma equação de linguagem é semelhante a uma equação algébrica que estamos acostumados a utilizar como:

$$2x + 13 = 23$$

Nessa equação desejamos achar o valor de x que faça com que a igualdade seja verdadeira. No caso, o único valor possível para x seria 5. Sabemos também que outras equações podem ter múltiplas soluções ou não ter solução nenhuma.

Equações de linguagens são utilizadas da mesma forma, porém os operandos envolvidos na equação são linguagens e não valores numéricos. Por exemplo, considerando o alfabeto $\{a, b, c, d\}$ podemos ter a equação

$$\{a, b, c\} = \{a, b\} \cup X$$

que indica que desejamos achar o(s) valor(es) da variável X que faça a igualdade ser verdadeira. Nesse caso, poderíamos ter como solução para a equação os valores $\{c\}$ ou $\{a, c\}$, ou $\{b, c\}$ ou $\{a, b, c\}$.

Um tipo de equação que estaremos particularmente interessados são as equações do tipo

$$X = E \cup A \cdot X$$

onde E e A são linguagens. Por exemplo,

$$X = \{b, c\} \cup \{a\}X$$

Ou, utilizando ER's:

$$X = (b \cup c) \cup aX$$

Analisando essas equações iremos encontrar como solução a linguagem $A^* \cdot E$. Ou seja, se substituirmos X na equação por $A^* \cdot E$ teremos a igualdade

$$A^* \cdot E = E \cup A \cdot (A^* \cdot E)$$

ou, no nosso exemplo:

$$\begin{array}{rl} X = & (b \cup c) \cup aX \\ a^* \cdot (b \cup c) = & (b \cup c) \cup a \cdot (a^*(b \cup c)) \end{array}$$

Além disso, se a equação $X = E \cup A \cdot X$ atende o requisito de que $\lambda \notin A$, então a solução $A^* \cdot E$ é única. Vejamos outros exemplos:

$$\begin{array}{rl} X = & (a \cup b)c \cup dX \\ = & d^*(a \cup b).c \end{array}$$

$$\begin{array}{rl} X = & ab \cup (a \cup b)^* X \\ = & ((a \cup b)^*)^* ab \\ = & (a \cup b)^* ab \end{array}$$

$$\begin{array}{rl} X = & (a \cup c)d^* \cup (a \cup c)X \\ = & (a \cup c)^*(a \cup c)^* d \\ = & (a \cup c)^* d \end{array}$$

$$\begin{array}{rl} X = & bX \\ = & \emptyset \cup bX \\ = & b^* \emptyset \\ = & \emptyset \end{array}$$

Para o nosso objetivo de transformar um AFND numa ER iremos utilizar sistemas de equações de linguagens. Como veremos adiante, um AFND pode

ser representado por um sistema de equações de linguagens do tipo

$$\begin{array}{lcccccccccc}
X_1 & = & E_1 & \cup & A_{1,1}X_1 & \cup & A_{1,2} & \cup & \ldots & \cup & A_{1,n}X_n \\
X_2 & = & E_2 & \cup & A_{2,1}X_1 & \cup & A_{2,2} & \cup & \ldots & \cup & A_{2,n}X_n \\
\vdots & & \vdots & & \vdots & & \vdots & & & & \vdots \\
X_n & = & E_n & \cup & A_{n,1}X_1 & \cup & A_{n,2} & \cup & \ldots & \cup & A_{n,n}X_n
\end{array}$$

Resolvendo esse conjunto de n equações com n variáveis iremos obter uma ER que corresponderá à linguagem reconhecida pelo AFND. Deixemos para depois a explicação de como representar um AFND através de um sistema de equações e vamos antes ver como solucionar tais sistemas.

A idéia é semelhante à usada para solucionar sistemas de equações numéricas. Inicialmente iremos, através de um processo de alterações das equações, eliminar uma variável e uma equação do sistema obtendo $n-1$ equações com $n-1$ variáveis, cujas soluções coincidem com as do sistema original. Repetindo esse processo sucessivamente iremos chegar a uma única equação que tem a forma $X = E \cup AX$, para a qual conhecemos a solução. Vamos tomar como exemplo o sistema

$$\begin{aligned}
X_1 &= \epsilon \cup aX_1 \cup bX_2 \\
X_2 &= bX_1
\end{aligned}$$

Se eliminarmos a variável X_2 (e como conseqüência a segunda equação) obteremos uma única equação da forma $X_1 = E \cup AX_1$. Nesse exemplo podemos eliminar X_2 simplesmente substituindo-a na primeira equação:

$$\begin{aligned}
X_1 = &\ \epsilon \cup aX_1 \cup b\overbrace{(bX_1)}^{X_2} \\
= &\ \epsilon \cup (a \cup bb)X_1 \\
= &\ (a \cup bb)^*\epsilon \\
= &\ (a \cup bb)^*
\end{aligned}$$

Achado o valor de X_1 podemos substituí-lo no sistema original e calcular X_2:

$$\begin{aligned}
X_2 = &\ bX_1 \\
= &\ b(a \cup bb)^*
\end{aligned}$$

De uma maneira geral, dado o sistema com n equações e n variáveis

$$\begin{array}{lcccccccccc}
X_1 & = & E_1 & \cup & A_{1,1}X_1 & \cup & A_{1,2} & \cup & \ldots & \cup & A_{1,n}X_n \\
X_2 & = & E_2 & \cup & A_{2,1}X_1 & \cup & A_{2,2} & \cup & \ldots & \cup & A_{2,n}X_n \\
\vdots & & \vdots & & \vdots & & \vdots & & & & \vdots \\
X_n & = & E_n & \cup & A_{n,1}X_1 & \cup & A_{n,2} & \cup & \ldots & \cup & A_{n,n}X_n
\end{array}$$

podemos transformá-lo num sistema com $n-1$ equações e $n-1$ variáveis, eliminando X_n e obtendo:

$$\begin{array}{lcccccccccc}
X_1 & = & E'_1 & \cup & A'_{1,1}X_1 & \cup & A'_{1,2} & \cup & \ldots & \cup & A'_{1,n-1}X_{n-1} \\
X_2 & = & E'_2 & \cup & A'_{2,1}X_1 & \cup & A'_{2,2} & \cup & \ldots & \cup & A'_{2,n-1}X_{n-1} \\
\vdots & & \vdots & & \vdots & & \vdots & & & & \vdots \\
X_{n-1} & = & E'_{n-1} & \cup & A'_{n-1,1}X_1 & \cup & A'_{n-1,2} & \cup & \ldots & \cup & A'_{n-1,n-1}X_{n-1}
\end{array}$$

onde cada E' e A' são dados por:

$$\begin{array}{ll} E'_i = & E_i \cup (A_{i,n} \cdot A^*_{n,n} \cdot E_n) \\ A'_{i,j} = & A_{i,j} \cup (A_{i,n} \cdot A^*_{n,n} \cdot A_{n,j}) \end{array}$$

Tomando o exemplo visto acima,

$$\begin{array}{l} X_1 = \epsilon \cup aX_1 \cup bX_2 \\ X_2 = \emptyset \cup bX_1 \cup \emptyset X_2 \end{array}$$

Para eliminar X_2 teremos:

$$X_1 = E'_1 \cup A'_{1,1} X_1$$

onde

$$\begin{array}{rl} E'_1 = & E_1 \cup (A_{1,2} \cdot A^*_{2,2} \cdot E_2) \\ = & \epsilon \cup (b \cdot \emptyset^* \cdot \emptyset) \\ = & \epsilon \cup \emptyset \\ = & \epsilon \end{array}$$

$$\begin{array}{rl} A'_{1,1} = & A_{1,1} \cup (A_{1,2} \cdot A^*_{2,2} \cdot A_{2,1}) \\ = & a \cup (b \cdot \emptyset^* \cdot b) \\ = & a \cup bb \end{array}$$

Assim obtemos

$$\begin{array}{rl} X_1 = & \epsilon \cup (a \cup bb) X_1 \\ = & (a \cup bb)^* \end{array}$$

$$\begin{array}{rl} X_2 = & bX_1 \\ = & b(a \cup bb)^* \end{array}$$

Como outro exemplo vamos resolver o sistema

$$\begin{array}{ll} X_1 = & \epsilon \cup bX_1 \cup aX_2 \cup cX_3 \\ X_2 = & (b \cup c) X_1 \cup aX_3 \\ X_3 = & (a \cup b) X_1 \cup cX_2 \end{array}$$

$$\begin{array}{llll} E_1 = \epsilon & A_{1,1} = b & A_{1,2} = a & A_{1,3} = c \\ E_2 = \emptyset & A_{2,1} = (b \cup c) & A_{2,2} = \emptyset & A_{2,3} = a \\ E_3 = \emptyset & A_{3,1} = (a \cup b) & A_{3,2} = c & A_{3,3} = \emptyset \end{array}$$

$$\begin{array}{l} E'_1 = \epsilon \cup (c\,\emptyset^*\,\emptyset) = \epsilon \\ A'_{1,1} = b \cup (c\,\emptyset^*(a \cup b)) = b \cup (c(a \cup b)) \\ A'_{1,2} = a \cup (c\,\emptyset^* c) = a \cup cc \\ X_1 = \epsilon \cup (b \cup c(a \cup b)) X_1 \cup (a \cup cc) X_2 \end{array}$$

$$\begin{array}{l} E'_2 = \emptyset \cup (a\,\emptyset^*\,\emptyset) = \emptyset \\ A'_{2,1} = (b \cup c) \cup (a\,\emptyset^*(a \cup b)) = (b \cup c) \cup (a(a \cup b)) \\ A'_{2,2} = \emptyset \cup (a\,\emptyset^* c) = ac \\ X_2 = \emptyset \cup ((b \cup c) \cup a(a \cup b)) X_1 \cup acX_2 \end{array}$$

Obtemos então o novo sistema que iremos resolver:

$$X_1 = \epsilon \cup (b \cup c(a \cup b))X_1 \cup (a \cup cc)X_2$$

$$X_2 = \emptyset \cup ((b \cup c) \cup a(a \cup b))X_1 \cup acX_2$$

$$\begin{array}{llll} E_1 = \epsilon & A_{1,1} = b \cup c(a \cup b) & & A_{1,2} = a \cup cc \\ E_2 = \emptyset & A_{2,1} = (b \cup c) \cup a(a \cup b) & & A_{2,2} = ac \end{array}$$

$$\begin{array}{l} E'_1 = \epsilon \cup ((a \cup cc)(ac)^* \emptyset) = \epsilon \\ A'_{1,1} = (b \cup c(a \cup b)) \cup ((a \cup cc)(ac)^*((b \cup c) \cup a(a \cup b))) \\ X_1 = ((b \cup c(a \cup b)) \cup ((a \cup cc)(ac)^*((b \cup c) \cup a(a \cup b))))^* \end{array}$$

Substituindo X_1 nos sistemas anteriores podemos também obter o valor de X_2 e X_3.

Resta agora estabelecer a relação entre um Autômato Finito e um sistema de equações de linguagens para que possamos transformar o Autômato Finito numa ER através da solução do sistema de equações. Para isso, vamos observar o diagrama da Figura 5.5. Definir a linguagem reconhecida pelo AFD $A = < \Sigma, S, S_0, \delta, F >$ da figura equivale a enumerar todos os strings que, processados a partir do estado inicial S_1, levem até um estado final. Olhando a figura podemos observar que esses strings seriam:

- o string vazio
- todo string começando com a letra a seguida por um string que, processado a partir do estado S_2, leve até um estado final
- todo string começando com a letra b seguida por um string que, processado a partir do estado S_3, leve até um estado final

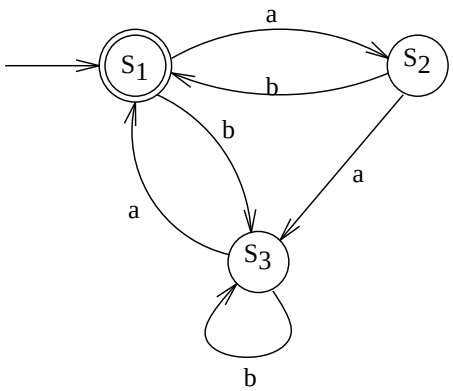


Figura 5.5: AFD a ser representado como um sistema de equações

Para sermos mais precisos, vamos definir o conjunto terminal de strings associado a um estado:

Definição 5.9 *Dado o AFD* $A = < \Sigma, S, S_0, \delta, F >$, *define-se o conjunto terminal de um estado s como:*

$$(\forall s \in S) \quad T(s) = \{x \in \Sigma^* | \overline{\delta}(s, x) \in F\}$$

Assim, a linguagem L(A) do autômato da Figura 5.5 pode ser definida como:

$$L(A) = T(S_1) = \{\lambda\} \cup \{a\} \cdot T(S_2) \cup \{b\} \cdot T(S_3)$$

e por sua vez, temos que

$$T(S_2) = \{b\} \cdot T(S_1) \cup \{a\} \cdot T(S_2)$$

$$T(S_3) = \{a\} \cdot T(S_1) \cup \{b\} \cdot T(S_3)$$

Chamando $T(S_i)$ de X_i podemos montar o sistema de equações

$$\begin{aligned} X_1 &= \epsilon \cup aX_2 \cup bX_3 \\ X_2 &= bX_1 \cup aX_3 \\ X_3 &= aX_1 \cup bX_2 \end{aligned}$$

Achando o valor de X_1 obtemos uma ER que nos dá $T(S_1)$ que é exatamente a linguagem L(A).

A definição de $T(s)$ acima foi feita para um AFD mas pode da mesma forma ser aplicada para AFND's. Assim, também um AFND pode ser representado através de um sistema de equações de linguagens de maneira igual a que acabamos de descrever. Deve-se tomar cuidado porém ao se determinar qual é a linguagem reconhecida por um AFND. No caso de um AFD, a linguagem reconhecida é dada pelo valor da variável associada ao estado inicial do AFD. Como um AFND pode possuir diversos estados iniciais é preciso fazer a união das linguagens reconhecidas a partir de cada um dos estados iniciais, ou seja, é necessário calcular o valor de todas as variáveis associadas a estados iniciais e fazer a união dessas linguagens.

Uma restrição também deve ser feita quanto aos AFND's com transições λ. Tais Autômatos Finitos dão origem a equações que possuem λ como parte de alguns conjuntos $A_{i,j}$ o que pode fazer com que o sistema de equações não possua uma única solução. Assim, devemos antes eliminar as transições λ e então montar o sistema de equações. Por outro lado, a solução calculada resolvendo o sistema como visto anteriormente é uma resposta correta e corresponde à linguagem aceita pelo AFND.

Capítulo 6

Gramáticas Regulares

Até agora vimos duas maneiras de representarem-se linguagens de modo formal: através Autômatos Finitos ou através de Expressões Regulares. Neste capítulo iremos estudar Gramáticas, que são uma terceira maneira para definirem-se linguagens.

Inicialmente definiremos o que são Gramáticas de um modo geral. Mostraremos também que existem diferentes categorias de Gramáticas que reconhecem diferentes tipos de linguagens, formando assim uma hierarquia de gramáticas.

No restante do capítulo nossa atenção estará concentrada numa categoria particular de Gramática, as Gramáticas Regulares (GR). Mostraremos que as linguagens que podem ser definidas através de uma Gramática Regular podem ser definidas também através de um Autômato Finito (e como conseqüência através de uma Expressão Regular) e vice-versa. Mostraremos portanto que existe uma equivalência entre AF's, ER's e GR's.

6.1 Definição Básica

Uma gramática é um modelo que permite definirem-se linguagens através de **regras de substituição**. Partindo-se de um símbolo inicial, substitituções podem ser feitas de acordo com essas regras até que um string pertencente à linguagem seja encontrado. A combinação de todas as seqüências de substituições irá então definir o conjunto de todas as cadeias que compõem a linguagem. Vamos supor que desejamos definir o conjunto de cadeias que representam expressões com somas e multiplicações de números com um único algarismo como

$$1 + 2 + 3 * 9$$

$$5 * 3$$

$$2 + 8$$

Para definir tal linguagem poderíamos utilizar as seguintes regras de substi-

tuição:

$$
\begin{array}{lcll}
E & \rightarrow & E+E & (1)\\
E & \rightarrow & E*E & (2)\\
E & \rightarrow & D & (3)\\
D & \rightarrow & 0 & (4)\\
D & \rightarrow & 1 & (5)\\
D & \rightarrow & 2 & (6)\\
D & \rightarrow & 3 & (7)\\
D & \rightarrow & 4 & (8)\\
D & \rightarrow & 5 & (9)\\
D & \rightarrow & 6 & (10)\\
D & \rightarrow & 7 & (11)\\
D & \rightarrow & 8 & (12)\\
D & \rightarrow & 9 & (13)
\end{array}
$$

Para determinar os strings que pertencem a essa linguagem devemos partir sempre do símbolo inicial (nesse caso o **E**) e aplicar uma seqüência de substituições permitidas pelas 13 regras dadas, até que se chegue a uma cadeia em que nenhuma substituição é possível. Tal cadeia é uma cadeia pertencente à linguagem desejada. Por exemplo, para chegar-se à cadeia $1+2+3*9$ pode-se aplicar a seguinte seqüência de substituições:

$$E \overset{(1)}{\Rightarrow} E+E \overset{(3)}{\Rightarrow} D+E \overset{(5)}{\Rightarrow} 1+E \overset{(1)}{\rightarrow} 1+E+E \overset{(3)}{\Rightarrow} 1+D+E \overset{(6)}{\Rightarrow} 1+2+E \overset{(2)}{\Rightarrow}$$

$$\overset{(2)}{\Rightarrow} 1+2+E*E \overset{(3)}{\Rightarrow} 1+2+D*E \overset{(7)}{\Rightarrow} 1+2+3+E \overset{(3)}{\Rightarrow} 1+2+3*D \overset{(13)}{\Rightarrow} 1+2+3*9$$

É fácil verificar que para um dado string existem diversas seqüências possíveis de substituições. Outra maneira de se visualizar a geração de um string a partir do símbolo inicial é através de uma **árvore de derivação** como a da Figura 6.1. As árvores de derivação serão analisadas mais cuidadosamente no Capítulo 7, quando trataremos de Gramáticas Livres de Contexto.

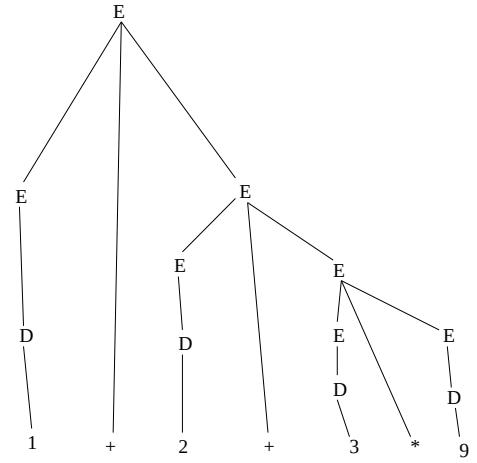


Figura 6.1: Árvore de derivação para 1 + 2 + 3 * 9

As regras iguais às vistas acima são normalmente chamadas de **produções**. Quando um determinado string α pode ser transformado em outro string β através da aplicação de uma produção diz-se que α **deriva diretamente** β, utilizando-se a notação $\alpha \Rightarrow \beta$. Quando β pode ser obtido através da aplicação de 0 ou mais produções, diz-se simplesmante que α **deriva** β notando-se $\alpha \stackrel{*}{\Rightarrow} \beta$. Por exemplo, dadas as produções do exemplo acima temos:

$$E \Rightarrow E * E$$
$$E \stackrel{*}{\Rightarrow} E * E$$
$$E \stackrel{*}{\Rightarrow} 1 + 2 + 3 * 9$$
$$E + D \Rightarrow E + 8$$

Vamos então à definição mais precisa do que é uma Gramática.

Definição 6.1 *Dado o alfabeto Σ, uma gramática sobre Σ é uma quádrupla $G = < \Omega, \Sigma, S, P >$ onde*

- *Ω é um conjunto não vazio de símbolos* **não terminais**
- *Σ é o conjunto de símbolos* **terminais** *($\Omega \cap \Sigma = \emptyset$)*
- *S é o* **símbolo inicial** *($S \in \Omega$)*
- *P é um conjunto de produções da forma $\alpha \rightarrow \beta$ tal que $\alpha \in (\Omega \cup \Sigma)^+$ e $\beta \in (\Omega \cup \Sigma)^*$*

O conjunto Ω é composto por aqueles símbolos que aparecem nas produções da gramática mas que não pertencem ao alfabeto Σ. Eles são símbolos "auxiliares" usados para definir como devem ser as regras de substituição. O strings que pertencem à linguagem definida por uma gramática são compostos exclusivamente por símbolos terminais, ou seja, do conjunto Σ. Isso implica que, para se chegar a um string da linguagem deve-se aplicar produções que substituam todos os símbolos não terminais, até que se obtenha uma cadeia que possua apenas terminais.

Um dos símbolos não terminais é especial. É o símbolo a partir do qual todas as derivações devem ser feitas. E P é o conjunto de produções da gramática. No exemplo anteriormente visto, esses quatro elementos seriam definidos como:

$< \{E, D\}, \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}, E, \{E \rightarrow E + E, E \rightarrow E * E, E \rightarrow D, D \rightarrow 0,$
$D \rightarrow 1, D \rightarrow 2, D \rightarrow 3, D \rightarrow 4, D \rightarrow 5, D \rightarrow 6, D \rightarrow 7, D \rightarrow 8, D \rightarrow 9\} >$

Outro exemplo de Gramática é $G = < \{A, B, S, T\}, \{a, b, c\}, S, \{S \rightarrow aSBc, S \rightarrow T, T \rightarrow \lambda, TB \rightarrow bT, cB \rightarrow Bc\} >$. Um exemplo de derivação para essa Gramática seria:

$$S \Rightarrow aSBc \Rightarrow aaSBcBc \Rightarrow aaTBcBc \Rightarrow aabTcBc \Rightarrow aabTBcc \Rightarrow aabbTcc \Rightarrow aabbcc$$

Dependendo do número de vezes que a produção $S \rightarrow aSBc$ é aplicada, essa gramatica gera strings da forma $\lambda, abc, aabbcc, aaabbbccc$. Podemos definir a linguagem definida por uma gramática como:

Definição 6.2 *Dada a Gramática* $G = < \Omega, \Sigma, S, P >$*, define-se a linguagem gerada por* G *como sendo:*

$$L(G) = \{x \in \Sigma^* \mid S \stackrel{*}{\Rightarrow} x\}$$

Informalmente, a linguagem $L(G)$ gerada pela gramática G é o conjunto de cadeias que podem ser derivadas a partir do símbolo inicial através da aplicação das produções de G e que só contenham símbolos terminais. Um ponto a se notar é que, contrariamente aos Autômatos Finitos, as Gramáticas possuem um carater "gerador" de strings. Por isso é comum dizer-se que um Autômato Finito "reconhece" uma linguagem e uma Gramática "gera" uma linguagem.

6.2 Hierarquia de Gramáticas

A definição 6.1 é a mais abrangente possível, no sentido que não impõe restrição no tipo de produções que podem ser utilizadas. Podem-se definir, por exemplo, produções do tipo

$$\begin{aligned} &S \to SaS \\ &aS \to b \\ &S \to c \\ &T \to aSd \end{aligned}$$

Esse tipo de Gramática é chamado de Gramática irrestrita ou do tipo 0. Uma lingugem L que pode ser definida através de tal Gramática é chamada de Linguagem do tipo 0. Se impusermos restrições ao tipo de produções que podem ser utilizadas, definiremos categorias diferentes de Gramáticas e tipos diferentes de linguagens que podem ser definidas através dessas diferentes Gramáticas. A primeira restrição que faremos é para evitar que uma produção "encolha" o tamanho do string já obtido até um determinado ponto da derivação. Supondo as produções dadas acima, vemos que ao aplicar $aS \to b$ teremos uma diminuição, ao menos momentânea, no tamanho do string gerado. Evitando tais produções iremos ter derivações mais "bem comportadas" e mais previsíveis. Assim, definimos o segundo tipo de Gramáticas.

Definição 6.3 *Dado o alfabeto* Σ*, uma* **Gramática Sensível ao Contexto Pura** *sobre* Σ *é uma quádrupla* $G = < \Omega, \Sigma, S, P >$ *onde*

- Ω *é um conjunto não vazio de símbolos* **não terminais**
- Σ *é o conjunto de símbolos* **terminais** *(*$\Omega \cap \Sigma = \emptyset$*)*
- S *é o* **símbolo inicial** *(*$S \in \Omega$*)*
- P *é um conjunto de produções da forma* $\alpha \rightarrow \beta$ *tal que* $\alpha \in (\Omega \cup \Sigma)^+$ *e* $\beta \in (\Omega \cup \Sigma)^+$ *e* $|\alpha| \leq |\beta|$

O fato de que $|\alpha| \leq |\beta|$ garante que não haverá nenhuma produção contrativa, que diminui o tamanho da cadeia gerada. Por outro lado, nunca teremos produções que possam gerar o string vazio. Isso muitas vezes não é desejado pois podemos querer definir uma linguagem L tal que $\lambda \in L$. Para contornar esse problema, faz-se então a definição das Gramáticas Sensíveis ao Contexto (não puras), permitindo que exista uma única produção do tipo $Z \rightarrow \lambda$ para gerar especificamente o string vazio.

Definição 6.4 *Dado o alfabeto* Σ*, uma* **Gramática Sensível ao Contexto** *ou do tipo 1 sobre* Σ *é uma Gramática Sensível ao Contexto pura ou uma quádrupla*

$$G' = < \Omega \cup Z, \Sigma, Z, P \cup \{Z \rightarrow \lambda, Z \rightarrow S\} >$$

onde $G = < \Omega, \Sigma, S, P >$ *é uma Gramática Sensível ao Contexto pura e* $Z \notin (\Omega \cup \Sigma)$*.*

Note que, se $\lambda \in L(G)$, deve existir um símbolo inicial Z que somente aparece do lado esquerdo de duas produções, uma para gerar o string vazio e outra para gerar os strings que podem ser gerados por uma Gramática Sensível ao Contexto pura. Uma lingugaem que pode ser gerada a partir de uma Gramática Sensível ao Contexto é chamada de Linguagem Sensível ao Contexto ou do tipo 1.

O termo "Sensível ao Contexto" vem do fato que podem ser utilizadas produções do tipo $\alpha B \beta \rightarrow \alpha \beta \gamma$, ou seja, o não terminal B é substituído somento no contexto em que ele aparece com o string α à esquerda e β à direita. Uma das catergorias de Gramáticas mais utilizadas, e que serão estudadas no Capítulo 7 é das Gramáticas Livres de Contexto. Nelas, um símbolo não terminal pode ser sempre substituído, independentemente do contexto em que ele aparece. Para isso, exige-se que o lado esquerdo de qualquer produção seja sempre formado por um único símbolo, não terminal.

Definição 6.5 *Dado o alfabeto* Σ*, uma* **Gramática Livre de Contexto Pura** *sobre* Σ *é uma quádrupla* $G = < \Omega, \Sigma, S, P >$ *onde*

- Ω *é um conjunto não vazio de símbolos* **não terminais**
- Σ *é o conjunto de símbolos* **terminais** *(*$\Omega \cap \Sigma = \emptyset$*)*
- S *é o* **símbolo inicial** *(*$S \in \Omega$*)*
- P *é um conjunto de produções da forma* $A \rightarrow \beta$ *tal que* $A \in \Omega$ *e* $\beta \in (\Omega \cup \Sigma)^+$

Note-se que por serem $|A| = 1$ e $|\beta| \geq 1$, então uma Gramática Livre de Contexto mantém a característica de que não existem produções contratoras. Mas ainda assim, não se pode, usando uma Gramática Livre de Contexto pura definir uma linguagem que inclua a cadeia vazia. Por isso define-se

Definição 6.6 *Dado o alfabeto* Σ*, uma* **Gramática Livre de Contexto** *ou do tipo 2 sobre* Σ *é uma Gramática Livre de Contexto pura ou uma quádrupla*

$$G' = < \Omega \cup Z, \Sigma, Z, P \cup \{Z \rightarrow \lambda, Z \rightarrow S\} >$$

onde $G = < \Omega, \Sigma, S, P >$ *é uma Gramática Livre de Contexto pura e* $Z \notin (\Omega \cup \Sigma)$*.*

Uma lingugaem que pode ser gerada a partir de uma Gramática Livre de Contexto é chamada de Linguagem Livre de Contexto ou do tipo 2.

Neste capítulo estaremos interessados particularmente na classe de Gramáticas chamada de Gramáticas Regulares, cuja definição é dada adiante. Antes disso devemos definir Gramáticas Lineares à Direita e Gramáticas Lineares à Esquerda.

Definição 6.7 *Dado o alfabeto* Σ*, uma* **Gramática Linear à Direita** *sobre* Σ *é uma quádrupla* $G = < \Omega, \Sigma, S, P >$ *onde*

- Ω *é um conjunto não vazio de símbolos* **não terminais**
- Σ *é o conjunto de símbolos* **terminais** *(*$\Omega \cap \Sigma = \emptyset$*)*
- S *é o* **símbolo inicial** *(*$S \in \Omega$*)*
- P *é um conjunto de produções da forma* $A \rightarrow xB$ *tal que* $A \in \Omega$ *e* $B \in (\Omega \cup \lambda)$ *e* $x \in \Sigma^*$

Nesse tipo de Gramática utilizam-se produções que têm sempre um único

símbolo não terminal do seu lado esquerdo e do seu lado direito possui um string que contém um prefixo (possivelmente λ) composto por símbolos terminais somente e um sufixo composto por um único símbolo não terminal ou λ. Ou seja, se a produção possuir um símbolo não terminal do seu lado direito ele deve vir sozinho e no final da produção. Similarmente definem-se as Gramáticas Lineares à Esquerda:

Definição 6.8 *Dado o alfabeto Σ, uma* **Gramática Linear à Esquerda** *sobre Σ é uma quádrupla $G = <\Omega, \Sigma, S, P>$ onde*

- *Ω é um conjunto não vazio de símbolos* **não terminais**
- *Σ é o conjunto de símbolos* **terminais** *($\Omega \cap \Sigma = \emptyset$)*
- *S é o* **símbolo inicial** *($S \in \Omega$)*
- *P é um conjunto de produções da forma $A \rightarrow Bx$ tal que $A \in \Omega$ e $B \in (\Omega \cup \lambda)$ e $x \in \Sigma^*$*

O conjunto de linguagens reconhecidas por Gramáticas Lineares à Direita e por Gramáticas Lineares à Esquerda é o mesmo. Esses dois tipos de Gramáticas formam as Gramáticas Regulares.

Definição 6.9 *Uma Gramática Regular ou do tipo 3 é uma Gramática Linear à Direita ou uma Gramática Linear à Esquerda*

Como foi dito, cada restrição feita ao tipo de produção utilizada nas diferentes classes de Gramáticas restringe também o tipo de linguagens que podem ser definidas por cada classe. As Gramáticas Regulares possuem exatamente o mesmo "poder" que os AFD's e as ER's, como veremos no decorrer do capítulo. Para as demais classes de Gramáticas, a relação de linguagens que elas definem é dada pela Figura 6.2.

6.3 Exemplos de Linguagens Definidas através de GR's

Vamos nesta seção ver alguns exemplos de Gramáticas Regulares para que o leitor se familiarize com esse tipo de modelo. Vamos iniciar com uma linguagem bastante simples: $L_1 = a^*$. Para L_1 podemos utilizar, por exemplo, a Gramática $G_1 = <\{S\}, \{a\}, S, \{S \rightarrow \lambda, S \rightarrow aS\}>$. É facil ver que, a cada vez que se emprega a produção $S \rightarrow aS$, inclue-se uma letra a na cadeia gerada. A geração termina sempre com $S \rightarrow \lambda$.

Aproveitando o exemplo de G_1, vamos criar $G_2 = <\{S\}, \{a\}, S, \{S \rightarrow \lambda\}>$ e $G3 = <\{S\}, \{a\}, S, \{S \rightarrow aS\}>$. Essa duas GR's produzem duas linguagens

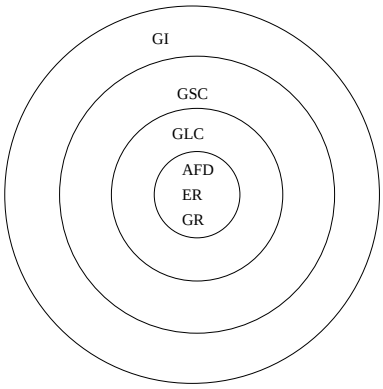


Figura 6.2: Hierarquia de Gramática

que também já conhecemos bem, que são $\{\lambda\}$ e $\emptyset$. No caso de G_2 é facil ver que o único string gerado é o string vazio. No caso de G_3, o que acontece é que a aplicação da sua única produção não leva nunca a uma cadeia que só contenha símbolos terminais. Portanto, seguindo a definição,

$$L(G_3) = \{x \in \{a\}^* \mid S \stackrel{*}{\Rightarrow} x\} = \emptyset$$

Vamos agora tentar criar uma Gramática que gere a linguagem a^*baa. Para isso, vamos aproveitar a idéia de G_1 e adicionar a cada palavra gerada pela gramática o sufixo baa. Podemos então definir $G_4 = < \{S,T\}, \{a,b\}, S, \{S \rightarrow aS, S \rightarrow bT, T \rightarrow aa\} >$. Ou ainda, $G_4 = < \{S\}, \{a,b\}, S, \{S \rightarrow aS, S \rightarrow baa\} >$. Note-se que, assim como AFD's e ER's, existem diversas formas diferentes de definir-se uma lingugem através de um GR. Assim, aproveitamos para fazer a seguinte definição:

Definição 6.10 *Duas Gramáticas G_1 e G_2 são ditas equivalentes, e denotadas $G_1 \equiv G_2$ sse $L(G_1) = L(G_2)$*

Vamos agora voltar à linguagem definida no Capítulo 2 através de um AFD que contém todas as cadeias que representam as constantes de ponto flutuante numa determinada linguagem de programação. Para essa linguagem vamos definir uma GR G_5. Uma notação comumente utilizada para representar um conjunto de produções $\{A \rightarrow \alpha_1, A \rightarrow \alpha_2, ..., A \rightarrow \alpha_n\}$ é $\{A \rightarrow \alpha_1 \mid \alpha_2, \mid ... \mid \alpha_n\}$. Vamos utilizar essa notação em $G_5 = < \{S,T,U,V,X\}, \{d,+,-,.,E\}, S, P >$, onde P é dado por:

$$\begin{array}{l} S \rightarrow +T \mid -T \mid T \\ T \rightarrow dT \mid d.U \mid d \\ U \rightarrow dU \mid V \mid \lambda \\ V \rightarrow E+X \mid E-X \mid EX \\ X \rightarrow dX \mid d \end{array}$$

Vamos ver alguns exemplos de geração de cadeias para essa Gramática.

- $S \Rightarrow +T \Rightarrow +dT \Rightarrow +dd$

- $S \Rightarrow +T \Rightarrow +dT \Rightarrow +dd.U \rightarrow +dd.$
- $S \Rightarrow +T \Rightarrow +dT \Rightarrow +dd.U \rightarrow +dd.dU \Rightarrow +dd.ddU \Rightarrow +dd.ddV \Rightarrow +dd.ddE - X \rightarrow +dd.ddE - d$

Verifique que, por outro lado, cadeias que não são constantes válidas como $+dd. + Edd$ não devem possuir seqüências de derivações que as produzam.

6.4 Equivalência entre AFD's e GR's

Tal como fizemos anteriormente para AF's e ER's, vamos mostrar nesta seção que AF's e GR's são duas formas equivalentes de se representarem linguagens, mostrando como transformar um AF numa GR correspondente e vice-versa.

6.4.1 Tranformação de um AFD numa GLD

Iniciamos mostrando que para cada AFD corresponde uma GLD de acordo com a definição:

Definição 6.11 *Dado a AFD* $A = < \Sigma, \{S_0, S_1, ..., S_n\}, S_0, \delta, F >$, *define-se a GLD* $G_A =< S, \Sigma, S_0, P_A >$ *onde* P_A *é dado por:*

$$P_A = \{S_i \rightarrow a\,\delta(S_i, a) \mid S_i \in S, a \in \Sigma\} \cup \{S_i \rightarrow \lambda \mid S_i \in F\}$$

Com essa definição torna-se simples calcular G_A. Vamos tomar como exemplo o AFD $A = < \{a, b\}, \{S_0, S_1\}, S_0, \{S_1\} >$, onde δ é dada por

$$\begin{array}{ll} \delta(S_0, a) = S_1 & \delta(S_0, b) = S_1 \\ \delta(S_1, a) = S_0 & \delta(S_1, b) = S_0 \end{array}$$

Esse AFD reconhece todas as cadeias sobre $\{a, b\}^*$ que têm tamanho ímpar. Para achar a GLD correspondente utilizaremos a definição acima e obteremos: $G_A = < \{S_0, S_1\}, \{a, b\}, S_0, P_A >$, onde P_A é dado por:

$$\begin{array}{lll} S_0 \rightarrow a\,\delta(S_0, a) & \text{ou seja} & S_0 \rightarrow aS_1 \\ S_0 \rightarrow b\,\delta(S_0, b) & \text{ou seja} & S_0 \rightarrow bS_1 \\ S_1 \rightarrow a\,\delta(S_1, a) & \text{ou seja} & S_1 \rightarrow aS_0 \\ S_1 \rightarrow a\,\delta(S_1, b) & \text{ou seja} & S_1 \rightarrow bS_0 \\ S_1 \rightarrow \lambda & & \end{array}$$

Note que a derivação

$$S_0 \Rightarrow bS_1 \Rightarrow baS_0 \Rightarrow babS_1 \Rightarrow bab$$

espelha exatamente o comportamento do AFD A. Ou seja, a cada derivação o prefixo de símbolos terminais da cadeia formada mostra qual a parte da cadeia

que já foi processada pelo AFD e o não terminal à direita mostra qual o estado corrente. Inicia-se com o AFD no estado S_0. Processando-se a letra b passa-se ao estado S_1, representado na derivação bS_1. Daí ao processar-se a letra a, o AFD volta a S_0 e a derivação correspondente é baS_0, e assim por diante. As produções de G_A só irão gerar uma cadeia de terminais quando uma das produções $S_i \to \lambda$ for aplicada. Caso contrário, sempre um novo não terminal é adicionado à cadeia gerada. Por isso as produções $S_i \to \lambda$ só existem para os estados S_i que são estados finais de A.

Teorema 6.1 *Dado o AFD A e a correspondente GLD G_A, A e G_A são equivalentes ($A \equiv G_A$, ou seja, $L(A) = L(G_A)$.*

Prova: Para demostrar esse teorema, iremos primeiro mostrar a seguinte propriedade:

$$(\forall x \in \Sigma^*)(\forall S_i \in S) \;\; \overline{\delta}(S_i, x) = S_j \Leftrightarrow S_i \stackrel{*}{\Rightarrow} xS_j \tag{6.1}$$

Vamos aplicar indução sobre o tamanho do string x.

Base: $|x| = 0$. Temos que para qualquer $S_i \in S$:

$$\begin{aligned} \overline{\delta}(S_i, \lambda) = S_j \Leftrightarrow & \quad \text{(pela definição de } \overline{\delta}(s, \lambda)) \\ S_i = S_j \Rightarrow & \quad \text{(Pela definição de } \stackrel{*}{\Rightarrow}) \\ S_i \stackrel{*}{\Rightarrow} S_j & \end{aligned}$$

e também, no sentido oposto:

$$\begin{aligned} S_i \stackrel{*}{\Rightarrow} S_j \Leftrightarrow & \quad \text{(Como não existem produções } S_i \to S_j) \\ S_i = S_j \Rightarrow & \quad \text{(Pela definição de } \overline{\delta}(s, \lambda)) \\ \overline{\delta}(S_i, \lambda) = S_j \Leftrightarrow & \end{aligned}$$

Passo da indução: supondo a hipótese válida para x, tomamos agora ax, $a \in \Sigma$. Temos para cada $S_i \in S$:

$$\begin{aligned} \overline{\delta}(S_i, ax) = S_j \Leftrightarrow & \quad \text{(Pela definição de } \overline{\delta}) \\ \overline{\delta}(\delta(S_i, a), x) = S_j \Leftrightarrow & \quad \text{(Chamando } \delta(S_i, a) \text{ de } S_k) \\ \overline{\delta}(S_k, x) = S_j \wedge \delta(S_i, a) = S_k \Leftrightarrow & \quad \text{(Pela definição de } P_A) \\ \overline{\delta}(S_k, x) = S_j \wedge S_i \to aS_k \in P_A \Leftrightarrow & \quad \text{(Pela hipótese de indução)} \\ S_k \stackrel{*}{\Rightarrow} xS_j \wedge S_i \to aS_k \in P_A \Leftrightarrow & \quad \text{(Pela definição de } \stackrel{*}{\Rightarrow}) \\ S_i \stackrel{*}{\Rightarrow} axS_j & \quad \text{C.Q.D} \end{aligned}$$

Mostraremos agora que $x \in L(A) \Leftrightarrow x \in L(G_A)$.

$$\begin{aligned} x \in L(A) \Leftrightarrow & \quad \text{(Pela definição de } L(A)) \\ \overline{\delta}(S_0, x) \in F \Leftrightarrow & \quad \text{(Pela definição de } P_A) \\ \overline{\delta}(S_0, x) \to \lambda \in P_A \Leftrightarrow & \quad \text{(Chamando } \overline{\delta}(S_0, x) \text{ de } S_k \text{ e por (6.1))} \\ S_0 \stackrel{*}{\Rightarrow} xS_k \wedge S_k \to \lambda \in P_A \Leftrightarrow & \quad \text{(Pela definição de } \stackrel{*}{\Rightarrow}) \\ S_0 \stackrel{*}{\Rightarrow} x \Leftrightarrow & \quad \text{(Pela definição de } L(G_A)) \\ x \in L(G_A) & \quad \text{C.Q.D.} \end{aligned}$$

A definição da GLD G_A foi feita com base num AFD A. Porém essa definição pode ser estendida para AFND's. Deve-se porém tomar o seguinte cuidado: num

AFD $A = < \Sigma, S, S_0, \delta, F >$, o símbolo inicial de G_A, S_0, deriva todos os strings de $L(G_A)$, assim como a partir do estado S_0 do AFD são reconhecidos todos os strings que levam a um estado final. No caso de um AFND, podemos ter diversos estados iniciais e então todas as cadeias derivadas em G_A a partir de qualquer um dos símbolos não terminais correspondentes a esses estados iniciais devem fazer parte de $L(G)$. Se S_0 é o conjunto de estados iniciais, então $L(G_A)$ deve ser dada por

$$L(G_A) = \bigcup_{t \in S_0} \{x \in \Sigma^* \mid t \stackrel{*}{\Rightarrow} x\}$$

Porém numa Gramática não podemos ter diversos símbolos iniciais. Por isso temos que adicionar as produções

$$\{Z \rightarrow t \mid t \in S_0\}$$

e fazer de Z o novo símbolo inicial da Gramática. Obs: $Z \notin S$.

Vamos tomar como exemplo o AFND $A = < \{a, b, c\}, \{S, T, U, V\}, \{S, T\}, \delta, \{V\} >$ cuja função δ é dada no diagrama da Figura 6.3.

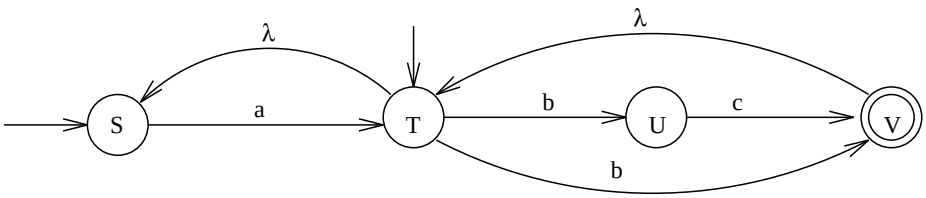


Figura 6.3: AFND para transformar para GLD

Teriamos como GLD correspondente $G_A = < \{Z, S, T, U, V\}, \{a, b, c, d\}, Z, P_A >$, onde P_A contém as seguintes produções:

$$\begin{aligned} &Z \rightarrow S \mid T \\ &S \rightarrow aT \\ &T \rightarrow bU \mid bV \mid S \\ &U \rightarrow cV \\ &V \rightarrow \lambda \mid T \end{aligned}$$

Pode-se também modificar a definição 6.11 para que G_A seja uma GLE ao invés de uma GLD. Fica por conta do leitor estabelecer como essa modificação deve ser feita.

6.4.2 Transformação de uma GLD num AFND

Vamos agora ver como achar um AFND que seja equivalente a uma GLD dada. Para isso temos a seguinte definição:

Definição 6.12 *Dada a GLD* $G = < \Omega, \Sigma, S, P >$, *define-se o AFND com transições* λ $A_G = < \Sigma, Q, q_0, \delta, F >$ *onde*

$$Q = \{<z> \in (\Sigma \cup \Omega)^* \mid z \in \Omega \vee \exists y \in (\Sigma \cup \Omega)^* \ni B \to yz \in P\}$$

$$q_0 = \{<S>\}$$

$$F = \{<\lambda>\}$$

$$(\forall <w> \in Q)(\forall a \in \Sigma) \quad \delta(<w>, a) = \{<x> \mid \exists y \in (\Omega \cup \Sigma)^* \ni w = ax \wedge B \to yw \in P\}$$

$$(\forall <B> \in Q) \quad \delta(<B>, \lambda) = \{<x> \mid B \to x \in P\}$$

O valor de δ *para todos os demais casos é* $\emptyset$.

O conjunto de estados do AFND A_G é formado por todos os sufixos das cadeias que aparecem à direita das produções da Gramática G e todos os símbolos não terminais de G. Por exemplo, tomando a GLD $G = < \{S, T\}, \{a, b\}, S, \{S \to aS, S \to bT, T \to aa\} >$, teríamos o seguinte conjunto de estados:

$$Q = \{<S>, <\lambda>, <aS>, <T>, <bT>, <a>, <aa>\}$$

O conjunto de estados iniciais q_0 é formado sempre pelo estado correspondente ao símbolo inicial da gramática, nesse caso, nesse caso $<S>$. O conjunto de estados finais também contém um único elemento que é sempre o estado $<\lambda>$.

A função δ é definida de forma que, para cada produção $B \to \alpha$, temos $\delta(< B >, \lambda) =< \alpha >$, lembrando que, quaisquer que sejam B e α, vão existir estados $<B>$ e $<\alpha>$ em A_G. Temos também, para cada estado $<a\alpha>$, uma transição do tipo $\delta(<a\alpha>, a) =<\alpha>$. Então, para a Gramática G teríamos:

$$\begin{array}{l}
\delta(<S>, \lambda) = \{<bT>, <aS>\} \\
\delta(<T>, \lambda) = \{<aa>\} \\
\delta(<aS>, a) = \{<S>\} \\
\delta(<bT>, b) = \{<T>\} \\
\delta(<a>, a) = \{<\lambda>\} \\
\delta(<aa>, a) = \{<a>\}
\end{array}$$

Para os demais pares, o valor de δ é vazio. Essa função é representada no diagrama da Figura 6.4.

Teorema 6.2 G *e* A_G *são equivalentes*

Assim como fizemos para o teorema anterior, devemos utilizar indução para mostrar que existe uma correspondência 1:1 entre o caminho percorrido em A_G para reconhecer um string e a seqüência de produções de G para gerar esse mesmo string. Por exemplo, para o string $abaa$ teríamos:

$$\text{AFND:} \quad <S>, <aS>, <S>, <bT>, <aa>, <a>, <\lambda>$$

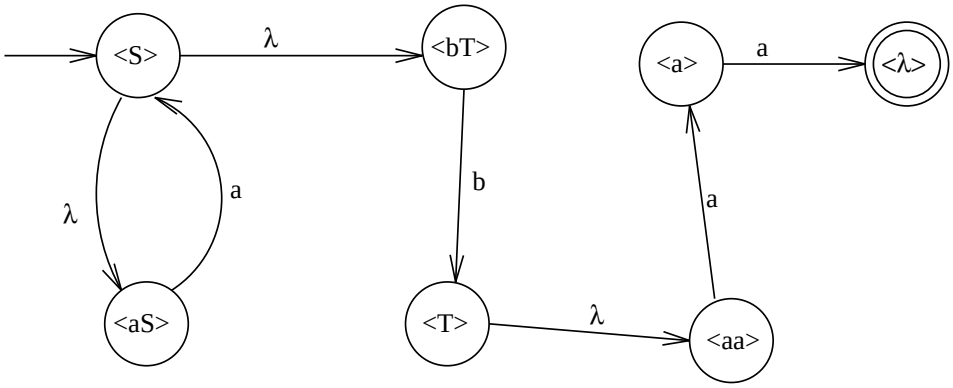


Figura 6.4: AFND A_G, correspondente à GLD G

GLD: $S \to aS \to abT \to abaa$

A demostração completa fica por conta do leitor.

6.5 Expressões Regulares X Gramáticas Regulares

Já vimos a equivalência que existe entre AFD's e ER'S e entre AF's e GR's e como transformar um AF em ER e vice-versa e AF em GR e vice-versa. Fica claro portanto a equivalência também entre ER's e GR's. Se quisermos encontrar, por exemplo, uma ER equivalente a uma GR G podemos transformar G num AF A_G e depois transformar A_G numa ER.

Por outro lado, a transformação direta de uma ER numa GR ou vice-versa é também bastante simples e e em geral mais eficiente do que essa transformação indireta. Nessa seção vamos mostrar informalmente como essas transformações podem ser feitas.

Inicialmente, para transformar uma ER numa GLD correspondente devemos proceder como faríamos para transformá-la num AFD. Iremos construir produções para cada uma das "expressões básicas" da ER e iremos combinando essas produções de acordo com os operadores da ER. Por exemplo, para a ER $(a \cup b) \cdot (c \cup d)^*$, devemos encontrar produções que gerem as linguagens $\{a\}$, $\{b\}$, $\{c\}$ e $\{d\}$. Em seguida devemos "unir" as duas últimas e as duas primeiras. Em seguida calcular o fecho de Kleene e a concatenação. Para as linguagens básicas teríamos as produções:

$\{a\}$	$\{b\}$	$\{c\}$	$\{d\}$
$A \to a$	$B \to b$	$C \to c$	$D \to d$

Para unir duas linguagens representadas através de produções cujos símbolos iniciais são S_1 e S_2, basta-nos introduzir um novo não terminal Z que será o símbolo inicial da linguagem resultante e as produções $Z \to S_1$ e $Z \to S_2$ (ou

$Z \to S_1 \mid S_2$). Teríamos então, no exemplo, os seguintes conjuntos de produções:

$\{a\} \cup \{b\}$	$\{c\} \cup \{d\}$
$T \to A \mid B$	$V \to C \mid D$
$A \to a$	$C \to c$
$B \to b$	$D \to d$

Obtivemos então T que gera $a \cup b$ e V que gera $c \cup d$. Para calcular o fecho de Kleene de uma linguagem gerada a partir do símbolo não terminal S iremos seguir os seguintes passos:

- introduzir um novo símbolo inicial Z e adicionar as produções $Z \to S$ e $Z \to \lambda$.

- alterar todas as produções que só tenham terminais do lado direito como $B \to x$ para $B \to xZ$

Com esses passos nos certificamos que as novas produções geram a cadeia vazia e "concatenamos" os strings da linguagem quantas vezes quizermos. No exemplo, teríamos:

$\{a\} \cup \{b\}$	$(\{c\} \cup \{d\})^*$
$T \to A \mid B$	$X \to V \mid \lambda$
$A \to a$	$V \to C \mid D$
$B \to b$	$C \to cX$
	$D \to dX$

E finalmente, para concatenar as linguagens cujos símbolos iniciais são S_1 e S_2 iremos alterar cada uma das produções encadeadas a S_1 que só tenha terminais à direita, do tipo $B \to x$, para $B \to xS_2$, adicionando assim os strings gerados a partir de S_2 à direita daqueles gerados a partir de S_1. O símbolo S_1 continua sendo o símbolo inicial da nova linguagem. Note-se que para tanto, os conjuntos de produções encadeadas a S_1 e a S_2 devem ser distintos. Por exemplo, se tivessemos

$$\begin{aligned} S_1 &\to aB \\ S_2 &\to cB \\ B &\to b \mid bB \end{aligned}$$

ao tentarmos concaternar S_1 e S_2 obteríamos

$$\begin{aligned} S_1 &\to aB \\ S_2 &\to cB \\ B &\to bS_2 \mid bB \end{aligned}$$

que não gera a linguagem desejada. O resultado pretendido seria ab^*bcb^*b.

Voltando ao exemplo, teríamos que concatenar as linguagens geradas a partir

de T e X, obtendo:

$$\frac{(\{a\} \cup \{b\}) \cdot (\{c\} \cup \{d\})^*)}{\begin{array}{c} T \rightarrow A \mid B \\ A \rightarrow aX \\ B \rightarrow bX \\ X \rightarrow V \mid \lambda \\ V \rightarrow C \mid D \\ C \rightarrow cX \\ D \rightarrow dX \end{array}}$$

e a gramática $G = < \{A, B, C, D, T, V, X\}, \{a, b, c, d\}, T, P >$ onde P é o conjunto de produções acima.

Para transformar uma GLD numa ER devemos proceder da mesma maneira que fizemos para transformar um AF numa ER, ou seja, montar um sistema de equações de linguagens e resolvê-lo. Neste caso iremos associar a cada símbolo não terminal S_i, uma variável X_{S_i}. Ao acharmos o valor da variável correspondente ao símbolo inicial da Gramática, obtemos a ER desejada. Para cada variável X_{S_i} constroi-se uma equação do tipo

$$X_{S_i} = E_i \cup A_{i,1}X_{S_1} \cup A_{i,2}X_{S_2} \cup ... \cup A_{i,n}X_{S_n}$$

onde E_i é a união de todas as cadeias terminais x que aparecem em produções do tipo $S_i \rightarrow x$. $A_{i,j}$ é a união de todas as cadeias terminais x que aparecem em produções do tipo $S_i \rightarrow xS_j$.

Por exemplo, vamos tomar a Gramática $G = < \{R, S, T\}, \{a, b.c\}, S, \{S \rightarrow aS \mid bT, T \rightarrow bc \mid bR, R \rightarrow T \mid \lambda\} >$. Temos então 3 equações com 3 variávaveis:

$$\begin{array}{lcccccccc} X_S & = & \emptyset & \cup & aX_S & \cup & bX_T & \cup & \emptyset X_R \\ X_T & = & bc & \cup & \emptyset X_S & \cup & \emptyset X_T & \cup & bX_R \\ X_R & = & \epsilon & \cup & \emptyset X_S & \cup & X_T & \cup & \emptyset X_R \end{array}$$

Achando o valor de X_S como visto anteriormente, obtemos $a^*bb^*(bc \cup b)$.

6.6 Formas Normais

Em alguns casos torna-se mais fácil utilizar uma Gramática se suas produções estiverem numa determinada forma particular. Para as GR's existem duas formas que podem ser úteis e que veremos a seguir. Deve-se notar que uma Gramática numa forma normal não define nenhum tipo especial de linguagem, ou seja, qualquer GR pode ser reescrita de maneira a estar numa forma normal, e continuar gerando a mesma linguagem.

Definição 6.13 *Dada a GLD* $G =< \Omega, \Sigma, S, P >$ *define-se* G^1 *tal que* $L(G) = L(G^1)$ *e todas as produções de* G^1 *têm a forma* $A \rightarrow aB$ *ou* $A \rightarrow \lambda$.

É facil ver que qualquer produção do tipo $A \rightarrow a_1a_2..a_nB$ pode ser substituída por um conjunto de produções $A \rightarrow a_1A_1$, $A_1 \rightarrow a_2A_2$, ... $A_{n-1} \rightarrow a_nB$.

Igualmente, qualquer produção do tipo $A \to a_1 a_2..a_n$ pode ser substituída por um conjunto de produções $A \to a_1 A_1$, $A_1 \to a_2 A_2$, ... $A_{n-1} \to a_n A_n$, $A_n \to \lambda$. O último tipo de produções indesejadas é do tipo $A \to B$. Esse tipo de produção pode também ser eliminada mas pode requer um esforço um pouco maior. Trataremos dessa eliminação quando tratarmos de Gramáticas Livres de Contexto.

Definição 6.14 *Dada a GLD* $G =< \Omega, \Sigma, S, P >$ *define-se* G^0 *tal que* $L(G) = L(G^0)$*, todas as produções de* G^0 *têm a forma* $A \to aB$ *ou* $A \to a$*, cuja única produção redutora permitida é* $S \to \lambda$ *e cujo símbolo inicial* S *numca aparece do lado direito de uma produção.*

Vamos tomar uma Gramática G que esteja na forma descrita na definição 6.13 (se não estiver, basta reescrevê-la para que fique naquela forma). Para criar G^0 devemos:

- manter todas as produções da forma $A \to aB$
- retirar todas as produções da forma $A \to \lambda$. para compensar essa retirada devemos incluir as produções

$$\{A \to a \mid (\exists B \in \Omega) \quad A \to aB \in P \wedge B \to \lambda \in P\}$$

- incluir um novo símbolo inicial Z e a produção $Z \to S$ e, caso $S \to \lambda \in P$, incluir $Z \to \lambda$.

Vamos tomar como exemplo a GR $G = < \{P, Q, R, S\}, \{0, 1\}, S, P >$, onde P é dado por:

$$\begin{array}{l} S \to 010S \mid 10P \mid 1 \\ P \to Q \mid 0R \\ Q \to 00S \\ R \to 10 \end{array}$$

Para calcular G^1 temos: $G^1 = < \{P, Q, Q_1, R, R_1, R_2, S, S_1, S_2, S_3, S_4\}, \{0, 1\}, S, P^1 >$, onde P^1 é dado por:

$$\begin{array}{l} S \to 0S_1 \mid 1S_2 \mid 1S_3 \\ S_1 \to 1S_4 \\ S_2 \to 0P \\ S_3 \to \lambda \\ S_4 \to 0S \\ P \to 0R \mid 0Q_1 \\ Q \to 0Q_1 \\ Q_1 \to 0S \\ R \to 1R_1 \\ R_1 \to 0R_2 \\ R_2 \to \lambda \end{array}$$

Sobre G^1 calculamos $G^0 = < \{P, Q, Q_1, R, R_1, S, S_1, S_2, S_4\}, \{0, 1\}, Z, P^0 >$, onde P^0 é dado por:

$$S \to 0S_1 \mid 1S_2 \mid 1$$
$$S_1 \to 1S_4$$
$$S_2 \to 0P$$
$$S_4 \to 0S$$
$$P \to 0R \mid 0Q_1$$
$$Q \to 0Q_1$$
$$Q_1 \to 0S$$
$$R \to 1R_1$$
$$R_1 \to 0$$

Uma vantagem imediata de se utilizar uma Gramática na forma normal de G^1 é que ela pode ser transformada diretamente num AFND, sem a necessidade de transições λ. Por exemplo, $G = < \{S, T, B, C\}, \{a, b\}, S, \{S \to aS, S \to bT, T \to aB, B \to aC, C \to \lambda\} >$ pode ser transformada diretamente no AFND cuja função de transição é dada na Figura 6.4.

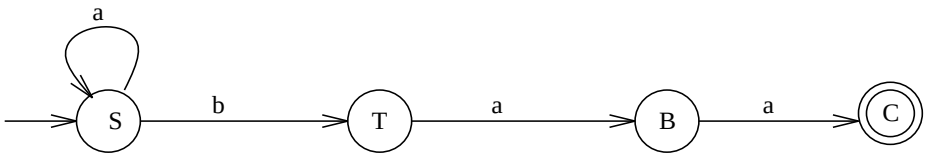


Figura 6.5: AFND correspondente a GLD na sua forma normal

Capítulo 7

Gramáticas Livres de Contexto

Vimos até agora três formas de representarem-se linguagens regulares. Comentamos também no Capítulo 6 que existem linguagens que não podem ser representadas por nenhuma dessas três formas.

Neste capítulo estaremos particularmente interessados na classe das Linguagens Livres de Contexto e em como definir tais linguagens através de Gramáticas. Iniciaremos mostrando como são as Linguagens Regulares, com o intuito de estabelecer quais são as linguagens que podem e quais não podem ser representadas através de AF's, ER's ou GR's.

Em seguida utilizaremos as Gramáticas Livres de Contexto para representar a classes das Linguagens Livres de Contexto. Finalizaremos o capítulo apresentando o Teorema Pumping, que mostra como identificar se uma linguagem é ou não Livre de Contexto.

7.1 Linguagens Autômato-definíveis

As linguagens sobre um alfabeto Σ e que podem ser definidas através de AF's são chamadas de "Linguagens Regulares", "Conjuntos Regulares" ou ainda "Liguagens Autômato-definíveis". Denotando esse conjunto por $\mathcal{D}_\Sigma$, podemos estabelecer diversas propriedades sobre $\mathcal{D}_\Sigma$, como por exemplo:

- $\mathcal{D}_\Sigma$ é fechado sob a operação de união
- $\mathcal{D}_\Sigma$ é fechado sob a operação de concatenação
- $\mathcal{D}_\Sigma$ é fechado sob o Fecho de Kleene

Essas três características em particular nos permitiram unir, concatenar e apliacar o Fecho de Kleene sobre AF's, garantindo que a linguagem resultante

também é regular. Por outro lado, existem linguagens para as quais não existem AF's que as possam representar. Tomemos, por exemplo, a linguagem

$$L = \{x \in \{a, b\}^* \mid |x|_a = |x|_b\}$$

Se o leitor tentar construir um AF que reconheça L irá notar que isso é imposível. O problema é que num AF, a única maneira de "guardar" informação sobre o que já foi processado até um determinado ponto é através dos estados. Um AF para reconhecer L deveria contabilizar, através de seus estados, qual é a diferença entre o número de a'a e o número de b's na cadeia processada. Essa diferença porém pode assumir qualquer valor, exigindo portanto um número infinito de estados, o que não é permitido num AF.

Na prática, diversas linguagens apresentam esse tipo de característica. Por exemplo, a maioria das linguagens de programação possui construções do tipo `begin/end` que precisam ser casados mas que podem estar aninhados. O número de `end`'s deve ser igual ao número de `begin`'s e em nenhum instante ao processar a cadeia x podemos ter $|x|_{end} > |x|_{begin}$

Vamos então analisar como identificar se um linguagem pode ou não ser reconhecida por um AF. Primeiro, é claro que qualquer lingugem com um número finito de elementos pode ser representada por um AF. O mesmo não acontece para linguagens com um número infinito de elementos. Nesse caso a linguagem precisa satisfazer algumas características para que seja uma Linguagem Regular. Essas características são estabelecidas no seguinte Teorema:

Teorema 7.1 *(Lema Pumping) - Seja L uma Linguagem Regular aceita pelo AFD $A = < \Sigma, S, S_0, \delta, F >$. Seja $x \in L$, tal que $|x| \geq |S|$. Então x pode ser escrito como uma cancatenação uvw, tal que $|uv| \leq |S|$, $|v| > 0$ e $uv^i w \in L$ para qualquer $i \geq 0$.*

Para entender o Lema Pumping vamos analisar o funcionamento de um AFD através de seu diagrama de transição. Seja $A = < \Sigma, S, S_0, \delta, F >$ e seja x um string pertencente a $L(A)$ tal que $|x| \geq k$ onde $k = |S|$. Ao processar o string x, existe pelo menos um estado $s \in S$ que será visitado mais do que uma vez. Assim, vamos dividir x em três partes:

- substring u, composto pelo prefixo de x até que o estado s seja alcançado pela primeira vez;
- substring v, não nulo, composto pela subpalavra de x que leva da primeira visita a s até a próxima visita a s; e
- subtring w composto pelo sufixo de x que leva de s até um estado final do AFD

Por exemplo, vamos tomar o digrama de transição de estados da Figura 7.1 e o string *ababbaaab*. Vamos dividir o string em três partes:

- $u = a$

- $v = bab$
- $w = baaab$

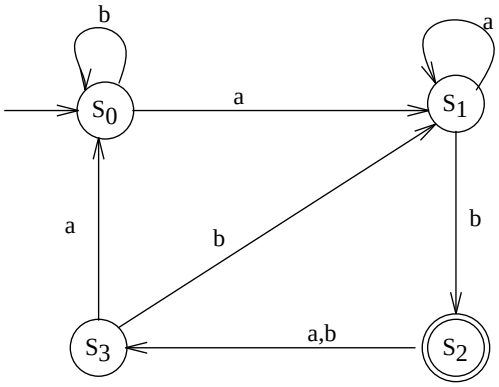


Figura 7.1: Diagrama de transição de um AFD

O string u leva o AFD do estado inicial ao estado S_1. O string v faz o string voltar a S_1, passando por S_2 e S_3. E o string w leva o AFD de S_1 ao estado final S_2. É facil verificar que se $uvw \in L(A)$, então $uvvw$ também pertence, assim como $uvvvw$ ou qualquer uv^iw, $i \geq 0$. Isso é exatamente o que diz o Lema Pumping. Qualquer string com tamanho mínimo k que é reconhecido por um AFD deve ter tal decomposição.

Na verdade, o que nos interessa são os casos em que podemos achar pelo menos um string que não satisfaça o Lema Pumping. Se formos capazes de achar tal string numa dada linguagem L, teremos mostrado que L não pode ser reconhecida por um AFD, ou seja, que L não é regular. Iremos procurar mostrar que não existe um valor de k para o qual x possa ser decomposto em uvw, tal que uv^iw sempre pertence a L.

Vamos tomar como exemplo a linguagem $L = \{x \in \{a,b\}^* \mid x = a^ib^i, i \geq 0\}$. Para provar que L não é regular vamos inicialmente supor que existe um AFD $A = < \Sigma, S, S_0, \delta, F >$ tal que $|s| = k$ e $L(A) = L$. Tomamos então o string $a^kb^k \in L$ e vamos decompô-lo nas três partes desejadas, de acordo com o Lema Pumping. Devemos ter:

$$\overbrace{a^i}^{u}\overbrace{a^j}^{v}\overbrace{a^{k-i-j}b^k}^{w}$$

onde $i + j \leq k$ e $j > 0$, ou seja, como a^k tem o mesmo número de letras que o número de estados de A, devemos dividí-lo em três partes, a^i, a^j e a^{k-i-j}, sendo que o primeiro irá constituir a parte que chamamos de substring u, a segunda, o substring v, e a terceira, junto com b^k, forma o substring w.

De acordo com o teorema, uv^iw deve também pertencer à linguagem L. Porém, iremos verificar que uv^2w não é um elemento de L pois teríamos

$$uv^2w = \overbrace{a^i}^{u}\overbrace{a^{2j}}^{v^2}\overbrace{a^{k-i-j}b^k}^{w} \notin L$$

Ou seja, uv^2w não pertence a L pois não tem o mesmo número de a's e b's. O string uv^2w tem a forma $a^{k+j}b^k$ e como $j > 0$, o número de a's difere do número de b's. Mostramos dessa maneira que a linguagem L contraria o Lema Pumping e por isso não existe um AFD que a reconheça.

Outro exemplo é linguagem $L = \{x \in \{a\}^* \mid |x| \text{ é primo}\}$, ou seja, $\{a, aaa, aaaaa, aaaaaaa, ...\}$. Da mesma forma, vamos assumir que existe um AFD com k estados que aceite essa linguagem. Tomamos então um valor n que é primo e $n > k$. O string a^n pertence a essa linguagem e deve ser decomposto em u, v e w, $|v| > 0$, $|uv| \leq k$, e $uv^iw \in L$ para satisfazer o Lema Pumping. Porém, se tomarmos $i = n + 1$ teremos:

$$\begin{aligned} |uv^{n+1}w| &= |uvw| + |v^n| \\ &= n + n|v| \\ &= n(1 + |v|) \end{aligned}$$

que não é primo pois é um valor múltiplo de n e de $1 + |v|$, mostrando assim que $uv^{n+1}w \notin L$ e que não existe um AFD para essa linguagem.

Considere agora a linguagem $L = \{x \in \{a, b\}^* \mid |x| \text{ é um quadrado perfeito}\}$. Assumindo que existe um AFD com k estados que reconheça L, tomamos um string $x \in \{a, b\}^*$ tal que $|x| = k^2$. Então $x \in L$ e, se L é uma Linguagem Regular, devemos pode decompor x de acordo com o Lema Pumping, ou seja, $x = uvw$, $|uv| \leq k$, $|v| > 0$. Sabendo que $0 < |v| \leq k$, temos portanto:

$$\begin{aligned} |uv^2w| &= |uvw| + |v| \\ &= k^2 + |v| \\ &\leq k^2 + k \\ &< k^2 + 2k + 1 \quad = (k + 1)^2 \end{aligned}$$

Mostramos assim, que $|uv^2w| > k^2$ pois $|v| > 0$ e que $|uv^2w| < (k + 1)^2$, não sendo portanto um quadrado perfeito e dessa forma contrariando o Lema Pumping, indicando que L não é uma Linguagem Regular.

7.2 Árvores de Derivação

No contexto de GLC costuma-se utilizar uma representação chamada de **Árvore de Derivação** para explicitar a forma com que uma cadeia é derivada. Dada a GLC (ou GR) $G = < \Omega, \Sigma, S, P >>$, uma Árvore de Derivação é uma árvore rotulada cuja raiz tem sempre o não terminal S como raiz. Numa derivarção

$$S \Rightarrow \alpha_1\alpha_2...\alpha_n \quad \alpha_i \in (\Omega \cup \Sigma)$$

o nó S terá como filhos n nós rotulados α_1 até α_n. Cada um desses nós que é não terminal deverá eventualmente ser substituído e será raiz de uma sub-árvore que representa outra derivação, até que todos os nós folhas sejam compostos apenas de símbolos terminais. No caso de produções vazias, do tipo $A \to \lambda$, uma derivação $Z \Rightarrow \lambda$ é representada com B como raiz e o símbolo λ como rótulo do único nó filho.

Vamos tomar como exemplo a Gramática $G_1 = < \{S, Z\}, \{a, b\}, \{Z \to \lambda \mid S, S \to aSb \mid ab\} >$. Ela gera os string λ, ab, $aabb$, $aaabbb$. Teríamos para esses string as árvores de derivação da Figura 7.2.

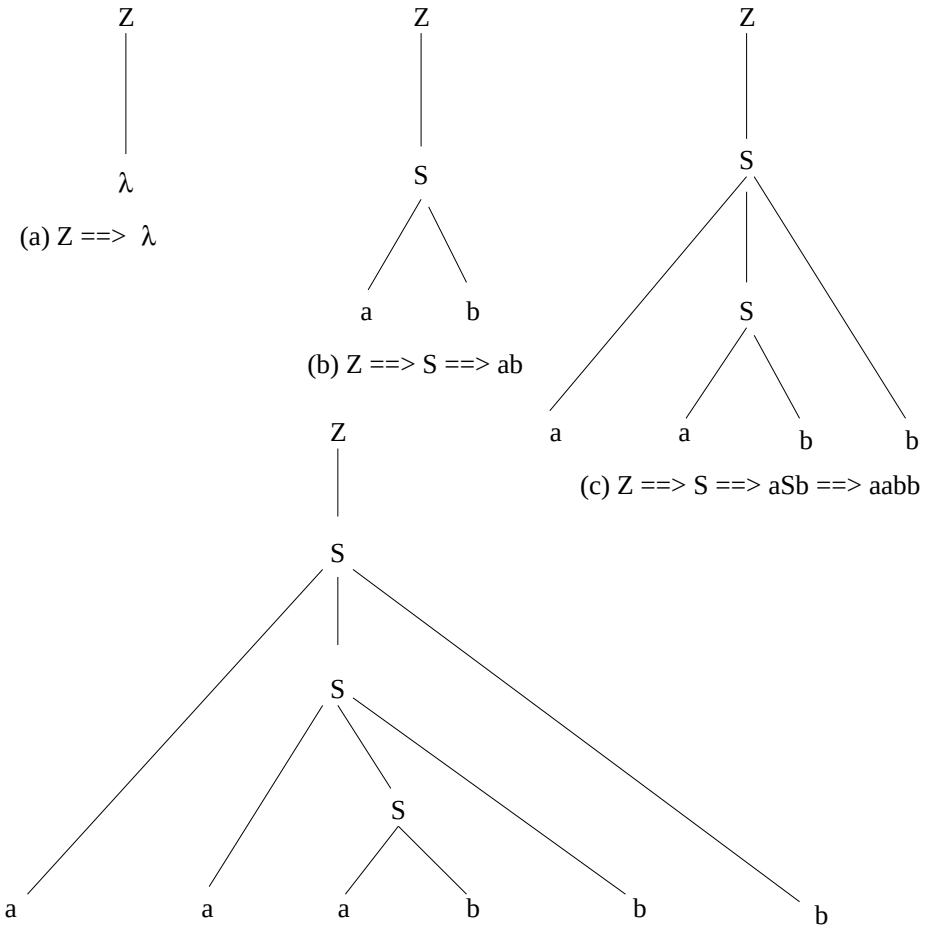


Figura 7.2: Exemplos de Árvore de Derivação

Para algumas Gramáticas podemos escolher diferentes ordens para aplicar as produções, sem contanto alterar o string gerado. Por exemplo, vamos tomar a Gramática $G_2 = < \{S, C\}, \{a, b, d, e, f\}, S, \{S \rightarrow aCSb \mid d, C \rightarrow CfC \mid e\} >$ e o string $aefedb$. Teríamos a seguinte seqüência de derivações para gerar esse string:

$$S \Rightarrow aCSb \Rightarrow aCfCSb \Rightarrow aefCSb \Rightarrow aefeSb \Rightarrow aefedb$$

$$S \Rightarrow aCSb \Rightarrow aCdb \Rightarrow aCfCdb \Rightarrow aCfedb \Rightarrow aefedb$$

$$S \Rightarrow aCSb \Rightarrow aCfCSb \Rightarrow aCfeSb \Rightarrow aCfedb \Rightarrow aefedb$$

A primeira seqüência de derivações é chamada de **derivação mais à esquerda**, pois aplica-se sempre a substituição ao não terminal que está mais à esquerda da cadeia. A segunda é a derivação **mais à direita** e a última não segue nenhuma ordem específica de substituição dos não terminais. Apesar de serem aplicadas as produções em ordens diferentes, devemos notar dois pontos

importantes: i) as produções aplicadas são sempre as mesmas; e ii) independentemente de qual seqüência escolhemos, a AD produzida será sempre a mesma (Figura 7.3).

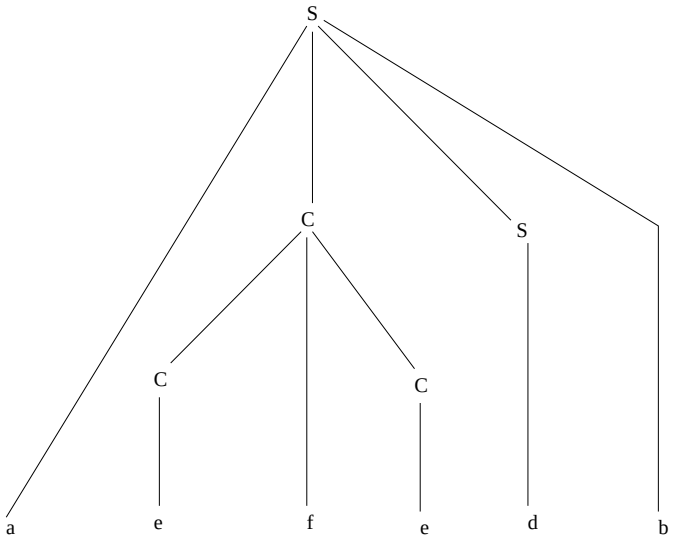


Figura 7.3: Árvore de Derivaçãoúnica para diferentes seqüências de produções

7.3 Ambigüidade

Por outro lado, existem gramáticas que permitem diferentes Árvores de Derivação(e diferentes produções) para uma mesma cadeia, conforme a definição abaixo:

Definição 7.1 *Dada a Gramática* $G = < \Omega, \Sigma, S, P >$*, diz-se que* G *é* **ambígua** *sse existe* $x \in L(G)$ *tal que* x *possui duas Árvores de Derivação distintas. Caso contrário,* G *é não ambígua.*

A mesma definição pode ser feita em termos das derivações mais à esquerda ou mais à direita. Uma gramática é ambígua se existir algum string para o qual existem mais do que uma derivação mais à esquerda (direita). Se isso ocorre é porque num determinado ponto da derivação, existem duas produções distintas para um mesmo não terminal e que é possível com qualquer uma delas chegar-se ao mesmo string.

Vamos tomar como exemplo a gramática $G = < \{E\}, \{a, +, *\}, E, \{E \rightarrow E + E \mid E * E \mid a\} >$ e o string $a * a + a$. Podemos achar duas derivações mais à esquerda para esse string que seriam:

$$E \Rightarrow E + E \Rightarrow E * E + E \Rightarrow a * E + E \Rightarrow a * a + E \Rightarrow a * a + a$$

$$E \Rightarrow E * E \Rightarrow a * E \Rightarrow a * E + E \Rightarrow a * a + E \Rightarrow a * a + a$$

e duas Árvores de Derivação das Figuras 7.4a e b.

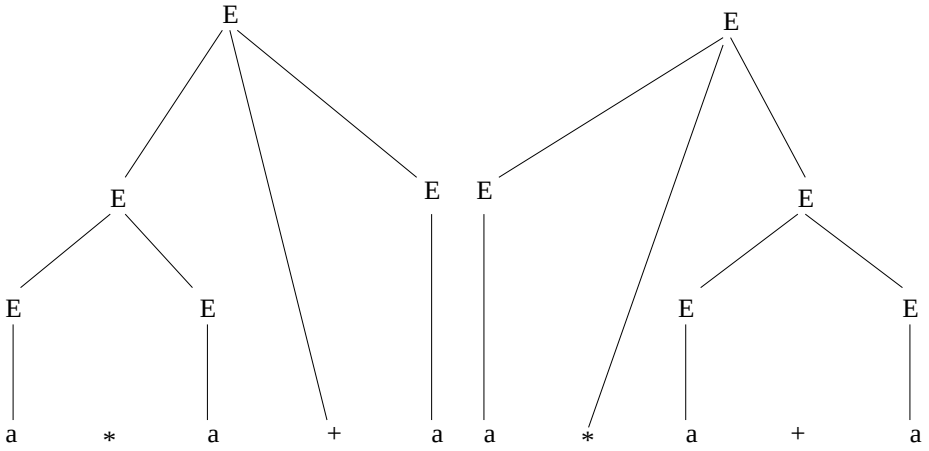


Figura 7.4: Árvores de Derivação distintas para a mesma cadeia

Dependendendo da aplicação que se deseja da Gramática, a ambigüidade não é uma característica desejada. É o caso da utilização de GLC's na análise sintática de um compilador. Além de reconhecer se um string pertence ou não à linguagem, o compilador precisa associar um significado ao string. Se um string pode ser reconhecido de maneiras distintas (possui diferentes Árvores de Derivação) então pode-se associar a ele diferentes semânticas. Por exemplo, no caso da Figura 7.4, o compilador tem que gerar código para calcular a expressão $a * a + a$. Se a primeira Árvore de Derivação for usada, assume-se que essa expressão é composta por duas subexpressões, conectdas pelo operador $+$. A expressão da direita é a e a da esquerda $a * a$. Essa última por sua vez também tem duas subexpressões conectadas pelo $*$. Já a segunda Árvore de Derivação indica que a expressão é formada por uma multiplicação de a por $a + a$. Uma Gramática que permita essas duas interpretações claramente não é adequada para utilização num compilador.

Nesse exemplo, deveríamos optar por uma gramática que interpretasse a expressão da maneira expressa pela Árvore de Derivação da Figura 7.4a. Ou seja, para avaliar $a * a + a$ deveríamos primeiro avaliar a subexpressão $a * a$ para depois podermos avaliar a expressão como um todo. Tal Gramática deve portanto estabelecer uma precedência entre os operadores $*$ e $+$. A Gramática $G_1 = < \{E, T\}, \{a, +, *\}, E, \{E \rightarrow E + E \mid T, T \rightarrow T * T \mid a\} >$ estabelece essa precedência. A Árvore de Derivação da Figura 7.5 mostra como a expressão $a * a + a$ é reconhecida.

Nessa Gramática, sempre que exista, o operador $+$ separa a expressão primeiro numa soma de subexpressões e essas então são divididas em multiplicações. G_1 porém não resolve outro tipo de ambiguidade, que surge quando temos o mesmo operador repetido diversas vezes como em $a + a + a$. Nesse caso, é preciso decidir qual é associatividade que se deseja dar a um operador. Para

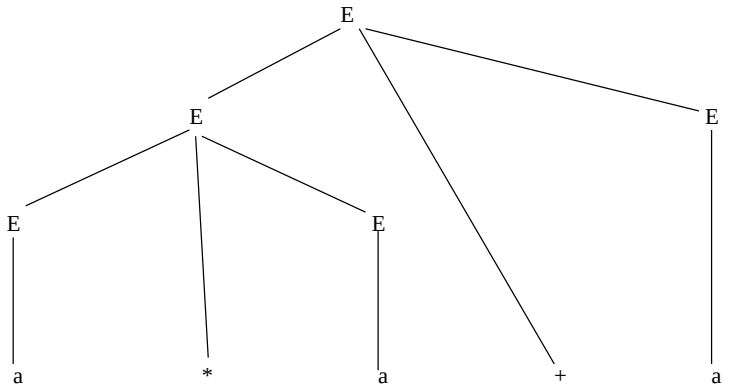


Figura 7.5: Árvore de Derivação para Gramática com precedência entre $*$ e $+$

$a + a + a$ podemos construir duas Árvores de Derivação distintas, mostradas na Figura 7.6

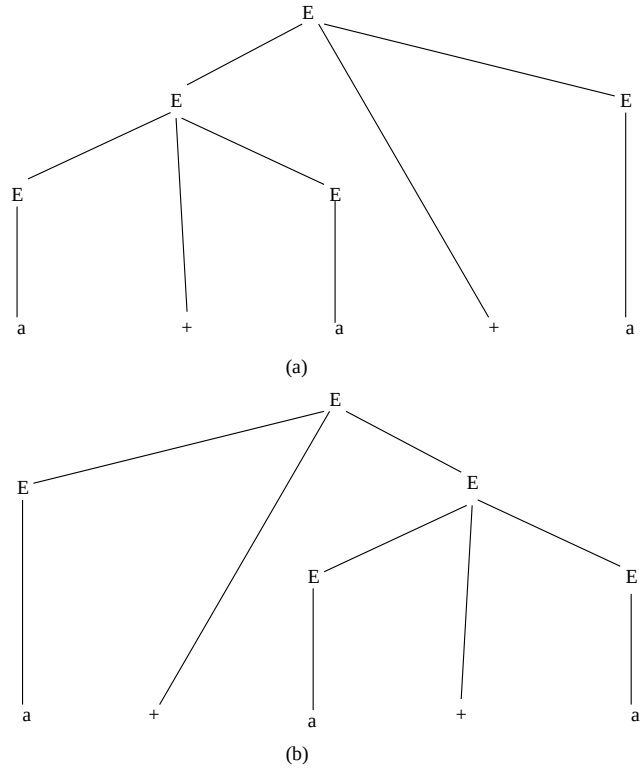


Figura 7.6: Árvores de Derivação distintas para $a + a + a$

Para resolver esse problema podemos alterar G_1, da seguinte forma, para obtermos sempre uma Árvore de Derivação como a da Figura 7.6a: $G_2 = < \{E, T, F\}, \{a, +, -\}, S, \{S \rightarrow E + T \mid T, T \rightarrow T * F \mid F, F \rightarrow a\} >$. Assim, a Árvore de Derivação de $a + a + a$ seria a mostrada na Figura 7.7. Para $a + a * a * a + a + a$, teríamos a Árvore de Derivação da Figura 7.8.

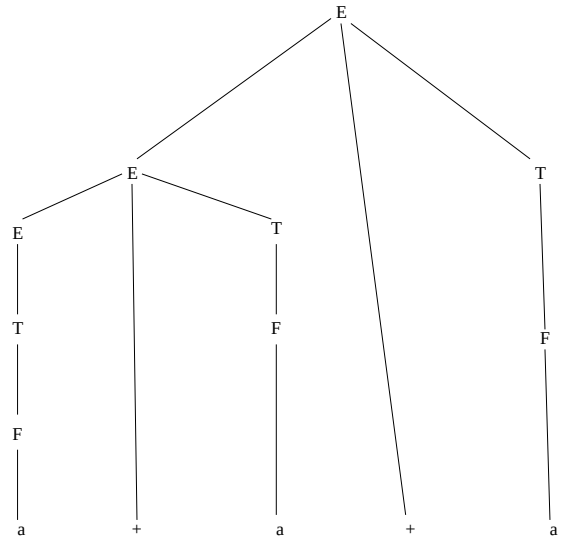


Figura 7.7: Árvore de Derivação para Gramática com associatividade para $a + a + a$

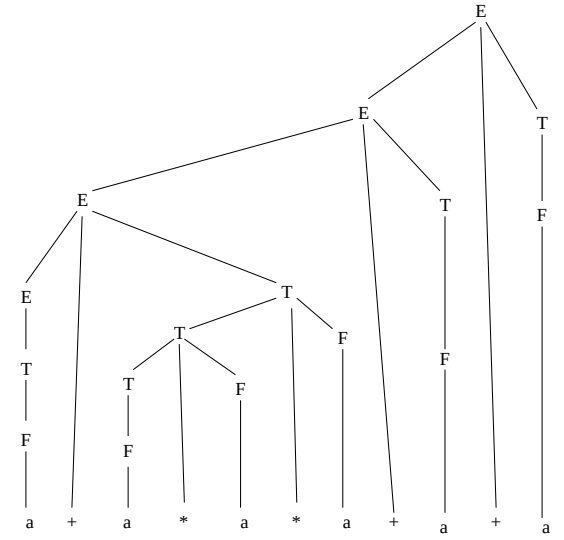


Figura 7.8: Árvore de Derivação para $a + a * a * a + a + a$

Uma Gramática completa para expressões com soma, subtração, multiplicação, divisão e subexpressãoes parentisadas é dada abaixo. Fica por conta do leitor montar algumas Árvores de Derivação para alguns strings dessa linguagem.

$$G_3 = < \{E, T, F\}, \{a, +, -, *, /, (,)\}, E, P >$$

onde P é dado por:

$E \rightarrow E + T \mid E - T \mid T$
$T \rightarrow T * F \mid T/F \mid F$

$F \rightarrow a \mid (E)$

Devemos notar que as Gramáticas G, G_1 e G_2 são equivalentes, apesar de G e G_1 serem ambíguas. A característica de ambiguidade não é desejável apenas quando é necessário associar-se um significado ao string e às partes que compõem o string gerado, como no caso da análise sintática. Precisamos também ressaltar que a ambigüidade, nesse caso, está associada à Gramática utilizada e não à linguagem em si. Existem casos porém em que a própria linguagem é ambígua, de acordo com a seguinte definição:

Definição 7.2 *Uma Linguagem Livre de Contexto L é chamada* **inerentemente ambígua** *sse toda gramática G tal que $L(G) = L$ é ambígua.*

Mais detalhes sobre linguagens inerentemante ambíguas podem ser encontrados em [HOP79].

7.4 Transformações sobre GLC's

Nesta seção iremos ver como podemos manipular as GLC's para que suas produções se adequem a determinadas restrições. Existem certas aplicações que requerem que as produções da Gramática atendam a determinadas restrições. Por exemplo, um analisador sintático descendente recursivo [?] não pode utilizar gramáticas com recursão à esquerda, portanto precisamos saber como eliminá-las para construir uma Gramática que possa ser utilizada em tal analisador. Esse e outros tipos de transformação podem fazer icom que uma determinada gramática G que não satisfaz algumas restrições passe a satisfazê-las. Obviamente, deseja-se que somente a estrutura de Gramática (produções, símbolos não terminais) seja alterada. A linguagem reconhecida deve permanecer a mesma.

7.4.1 Eliminação de Produções Vazias

Uma GLC, como vista no Capítulo 6 possui todas as suas produções $A \rightarrow \beta$ onde $\beta \in (\Omega \cup \Sigma)^+$, ou seja, não podem existir produções vazias do tipo $A \rightarrow \lambda$, para que o tamanho das cadeias não diminuam durante a derivação. Na prática, porém, esse tipo de produção costuma ser utilizada. No exemplo da Seção 7.7 onde uma GLC para Pascal é apresentada, produções λ são utilizadas e, em geral, facilitam o entendimento das construções gramaticais.

O fato de que essas gramáticas contrariam a definição que estamos utilizando, não é realmente sério, pois qualquer produção vazia pode ser eliminada da gramática. Iniciamos então essa seção mostrando como eliminar tais produções.

As produções vazias são utilizadas para retirar um símbolo não terminal da forma sentencial gerada. Por exemplo, dado o seguinte conjunto de produções

$S \rightarrow ACA$
$A \rightarrow aAa \mid B \mid C$

$B \to bB \mid b$
$C \to cC \mid \lambda$
ao derivar o string bcb temos a seguinte derivação mais à esquerda:

$$S \Rightarrow ACA \Rightarrow BCA \Rightarrow bCA \Rightarrow bcCA \Rightarrow bcA \Rightarrow bcB \Rightarrow bcb$$

Note que $C \to \lambda$ foi utilizada para retirar o C da cadeia $bcCA$, fazendo o tamanho da cadeia diminuir.

Para evitar produções vazias devemos ter uma gramática que não introduza os símbolos não terminais que mais tarde serão retirados da cadeia através de produções λ. No conjunto de produções anterior isso acontece com C mas também acontece com S e com A. Por exemplo, tomemos a derivação:

$$S \Rightarrow ACA \Rightarrow CCA \Rightarrow CA \Rightarrow A \Rightarrow C \Rightarrow \lambda$$

Nessa derivação, nenhum dos não terminais ACA deveria ter sido colocado na cadeia pois todos foram posteriormente retirados.

O primeiro passo para remover as produções vazias é achar os não terminais A tal que $A \stackrel{*}{\Rightarrow} \lambda$. Isso pode ser feito através do seguinte algoritmo:

a. Inicialize o conjunto $NULL$ com $\{A \mid A \to \lambda \in P\}$

b. Inclua em $NULL$ todo A tal que $A \to \alpha \in P, \alpha \in NULL^*$

c. repita o passo b até que nenhum novo símbolo seja acrescentado ao conjunto $NULL$

No caso das produções acima teríamos:

$NULL := \{C\}$
$NULL := NULL \cup \{A\}$
$NULL := NULL \cup \{S\}$
terminando com $NULL = \{A, C, S\}$.

Uma vez identificados os não terminais que derivam o λ (chamados de não terminais **nullable**), vamos eliminar as produções vazias. Para isso devemos alterar todas as produções que possuem do lado direito algum não terminal nullable. Por exemplo, a produção $S \to ACA$ deve ser alterada de maneira que a exclusão da produção $C \to \lambda$ não altere o conjunto de strings gerados a partir de S, uma vez que A e C não devem mais ser nullable. Por isso devemos substituir cada produção que tem esses não terminais do lado direito por um conjunto de produções da seguinte maneira:

a. se o símbolo inicial $S \in NULL$, introduza um novo não terminal Z e as produções $Z \to \lambda$ e $Z \to S$

b. seja a produção $B \to \alpha_1 A_1 ... \alpha_n An \alpha_{n+1}$ onde $A_i \in NULL$. Substitua essa produção por outras produções formadas pela eliminação de todas as combinações dos A_i (inclusive $\{\}$), uma de cada vez

c. retire todas as produções vazias, exceto talvez aquela introduzida no passo a

Para o exemplo acima teríamos $NULL = \{A, C, S\}$. Como $S \in NULL$, introduzimos as produções $Z \to \lambda \mid S$ e fazemos de Z o novo símbolo inicial. A produção $S \to ACA$ contém símbolos de $NULL$ assim devemos substitui-la por

$$S \to ACA \mid CA \mid AC \mid AA \mid A \mid C \mid \lambda$$

O mesmo ocorre com $A \to aAa$ que é substituída por

$$A \to aAa \mid aa$$

A produção $A \to C$ e $C \to cC$ viram

$$A \to C \mid \lambda$$

$$C \to cC \mid c$$

Eliminando as produções vazias obtemos o seguinte conjunto de produções:

$Z \to \lambda \mid S$
$S \to ACA \mid CA \mid AC \mid AA \mid A \mid C$
$A \to aAa \mid aa \mid C \mid B$
$B \to bB \mid b$
$C \to cC \mid c$

A derivação de bcb agora fica:

$$Z \Rightarrow S \Rightarrow ACA \Rightarrow BCA \Rightarrow bCA \Rightarrow bcA \Rightarrow bcB \Rightarrow bcb$$

No restante deste capítulo seguiremos a definição de GLC dada no Capítulo 6. Ou seja, não são permitidas transições vazias, exceto $Z \to \lambda$, quando λ faz parte da linguagem desejada.

7.4.2 Eliminação de Cadeias de Produções

Muitas vezes desejamos eliminar produções do tipo $A \to B$, chamadas de **cadeias de produções** ou **produções unitárias**, que são simplesmente uma "mudança de nome" dentro da cadeia gerada. Isso pode ser feito, substituindo essa produção por $\{A \to \alpha \mid B \to \alpha \in P\}$. Por exemplo:

$S \to aA \mid bB \mid B$
$B \to BS \mid SB$
$A \to ...$

Eliminamos $S \to B$ substituindo-a por $S \to BS$ e $S \to SB$, ou seja, S deve produzir cada um dos strings produzidos por B. Isso pode gerar problemas em casos como o das produções abaixo:

$S \to aS \mid a \mid B$
$B \to bB \mid b \mid C$
$C \to CS \mid SC \mid c$

Nesse caso, ao substituir $S \to B$, obteríamos outra produção do mesmo tipo, $S \to C$. Por isso, devemos calcular inicialmente, para cada não terminal A, o conjunto $CHAIN_A = \{B \in \Omega \mid A \stackrel{*}{\Rightarrow} B\}$. Isso é feito da seguinte maneira:

a. Inicialize o conjunto $CHAIN_A$ com $\{A\}$

b. Para cada não terminal $B \in CHAIN_A$ adicione a $CHAIN_A$ o conjunto $\{C \in \Omega \mid B \rightarrow C \in P\}$

c. Repita o passo b até que nenhum símbolo novo seja adicionado a $CHAIN_A$

Para o conjunto de produções

$Z \rightarrow \lambda \mid S$
$S \rightarrow ACA \mid CA \mid AC \mid AA \mid A \mid C$
$A \rightarrow aAa \mid aa \mid C \mid B$
$B \rightarrow bB \mid b$
$C \rightarrow cC \mid c$
teríamos

$$CHAIN_S = \{A, B, C, S\}$$

$$CHAIN_A = \{A, B, C\}$$

$$CHAIN_B = \{B\}$$

$$CHAIN_C = \{C\}$$

Uma vez feito isso, definimos o conjunto de produções da nova Gramática como:

$$P' = \{A \rightarrow \alpha \mid B \rightarrow \alpha \in P, B \in CHAIN_A, \alpha \notin \Omega\}$$

Aplicando essa definição, obteríamos as seguintes produções:

$Z \rightarrow \lambda \mid S$
$S \rightarrow ACA \mid CA \mid AC \mid AA \mid aAa \mid aa \mid bB \mid b \mid cC \mid c$
$A \rightarrow aAa \mid aa \mid bB \mid b \mid cC \mid c$
$B \rightarrow bB \mid b$
$C \rightarrow cC \mid c$

7.4.3 Eliminação de Símbolos Inúteis

Algumas vezes podemos encontrar Gramáticas que possuem símbolos não terminais que não continuem para a produção de cadeias. Isso é comum, por exemplo, ao aplicar-se o algoritmo de eliminação de produções unitárias. Vamos tomar a Gramática $< \{S, A\}, \{a, b\}, S, \{S \rightarrow a \mid A, A \rightarrow b \mid bS\} >$. Ao eliminarmos a produção unitária $S \rightarrow A$ obteremos $< \{S, A\}, \{a, b\}, S, \{S \rightarrow a \mid b \mid bS, A \rightarrow b \mid bS\} >$. O não terminal A nesse caso não desempenha nenhuma função na geração de cadeias pois não é alcançado a partir do símbolo inicial. Mais precisamente temos:

Definição 7.3 *Dada a GLC* $G = < \Omega, \Sigma, S, P >$*, diz-se que uma produção* $A \rightarrow \beta$ *é* **útil** *se existe uma derivação* $S \stackrel{*}{\Rightarrow} \alpha_1 A \alpha_2 \Rightarrow \alpha_1 \beta \alpha_2 \stackrel{*}{\Rightarrow} x$, $x \in \Sigma^*$. *Uma produção que não é útil é chamada* **inútil**. *Um não terminal que não apareça do lado esquerdo de nenhuma produção útil é também chamado inútil.*

Existem três formas pelas quais um símbolo não terminal A pode ser inútil:

a. A partir de A não se consegue chegar a nenhuma cadeia de terminais;

b. A partir do símbolo inicial não se consegue chegar a uma forma sentencial que contenha A;

c. O não terminal A é sempre produzido juntamente com outros símbolos inúteis.

Por exemplo, vamos tomar $G_1 = < \{S, A, B, C, D, U, V, W, X, Y\}, \{a, b, c, d, e, f, g, h, i, j, l, m, n, o\}, S, P >$, onde P é:

$S \rightarrow gAe \mid aYB \mid CY$
$A \rightarrow bBY \mid ooC$
$B \rightarrow dd \mid D$
$C \rightarrow jVB \mid gi$
$D \rightarrow n$
$U \rightarrow lW$
$V \rightarrow baXXX \mid oV$
$W \rightarrow c$
$X \rightarrow fV$
$Y \rightarrow Yhm$

O símbolo Y se enquadra no caso a. Sua única produção $Y \rightarrow Yhm$ sempre introduz um não terminal, fazendo com que nunca se chegue a uma cadeia de terminais. O mesmo acontece com X e V. O não terminal W está no caso b. A partir de S não existe nenhuma seqüência de derivações que faça com que W apareça numa forma sentencial. O mesmo acontece com U. Já o símbolo B produz cadeias de terminais e pode ser alcançado a partir de S, porém sempre é produzido com outros não terminal inúteis. Nas produções $S \rightarrow aYB$, $A \rightarrow bBY$, $C \rightarrow jVB$, B aparece com o Y e V, que são inúteis. Portanto, essas produções são inúteis e B não aparece em nenhuma outra produção útil.

Devemos ser cuidadosos ao tentar eliminar os símbolos inúteis de uma Gramática. No caso acima, uma vez que identifiquemos B como inútil devemos concluir também que D é inútil, visto que D só é alcançável através de B. Assim, para retirarmos os não terminais inúteis devemos proceder em dois passos. Inicialmente eliminamos os não terminais que não geram cadeias terminais. Isso pode ser feito da seguinte maneira, dada a Gramática$G = < \Omega, \Sigma, S, P >$:

a. Inicialize o conjunto $TERM$ com $\{A \mid A \rightarrow x \in P, x \in \Sigma^*\}$

b. Para cada não terminal A, verifique se existe $A \rightarrow \alpha$, $\alpha \in (TERM \cup \Sigma)^*$. Se existe então adiciona A a $TERM$

c. Repita o passo b até que nenhum símbolo novo seja adicionado a $TERM$

O conjunto $TERM$ representa o conjunto de não terminais para os quais é possível gerar alguma cadeia terminal. Para o caso da Gramática G_1 teríamos a seguinte seqüência no cálculo de $TERM$:

PASSO	$TERM$
Inicializa $TERM$	$\{W, D, C, B\}$
Adiciona A e U	$\{W, D, C.B, A, U\}$
Adiciona S	$\{W, D, C.B, A, U, S\}$
Não adicona, termina	$\{W, D, C.B, A, U, S\}$

Assim, o conjunto de não terminais fica reduzido ao conjunto $TERM$. Os demais não terminais e suas produções são descartadas. Devemos retirar também todas as produções onde aparecem não terminais de $\Omega - TERM$. Obtemos, então a Gramática $G' = < TERM, \Sigma, S, P' >$, onde P' é

$$P' = \{A \rightarrow \alpha \mid A \rightarrow \alpha \in P, A \in TERM, \alpha \in (TERM \cup \Sigma)^*\}$$

Note que essa definição presupõe que $S \in TERM$. Se isso não ocorre, então nenhuma cadeia pode ser derivada de S, e $L(G) = \emptyset$.

No exemplo de G_1, obteríamos $G_1' = < \{S, A, B, C, D, U, W\}, \{a, b, c, d, e, f, g, h, i, j, l, m, n, o\}, S, P' >$, onde P' é

$S \rightarrow gAe$
$A \rightarrow ooC$
$B \rightarrow dd \mid D$
$C \rightarrow gi$
$D \rightarrow n$
$U \rightarrow kW$
$W \rightarrow c$

Vamos agora ao segundo passo, que é retirar os não terminais que não são alcançáveis a partir do símbolo inicial. Isso é feito da seguinte forma:

a. Inicialize o conjunto $REACH$ com $\{S\}$

b. Pra cada não terminal $A \in REACH$ e cada produção $A \rightarrow \alpha$, adicione a $REACH$ o não terminal B, se B faz parte de α

c. Repita o passo b até que nenhum símbolo novo tenha sido incluído em $REACH$

Uma vez feito isso, obtemos a Gramática G'' que não contém símbolos inúteis, dada por $G'' = < REACH, \Sigma, S, P'' >$, onde P'' é

$$P'' = \{A \rightarrow \alpha \mid A \rightarrow \alpha \in P', A \in REACH\}$$

Ne exemplo teríamos a seguinte seqüência para calcular $REACH$:

PASSO	$REACH$
Inicializa $REACH$	$\{S\}$
Adiciona A	$\{S, A\}$
Adiciona C	$\{S, A, C\}$
Não adicona, termina	$\{S, A, C\}$

E a Gramática $G_1'' = < \{S, A, C\}, \{a, b, c, d, e, f, g, h, i, j, l, m, n, o\}, S, \{S \rightarrow gAe, A \rightarrow ooC, C \rightarrow gi\} >$.

7.4.4 Forma Normal de Chomsky

Como foi dito no Capítulo 6, uma forma normal é uma maneira especial de se escrever uma Gramática(no caso, uma GLC). Qualquer GLC pode ser alterada de maneira a obter-se um Gramática equivalente e que esteja numa forma normal. Uma dessas formas é a Forma Normal de Chomsky:

Definição 7.4 *Uma Gramática* $G = < \Omega, \Sigma, S, P >$ *está na Forma Normal de Chomsky Pura se* P *contém produções somente do tipo* $A \rightarrow BC$ *e* $A \rightarrow a$, $A, B, C \in \Omega$ *e* $a \in \Sigma$. *Uma Gramática* $G =< \Omega \cup Z, \Sigma, Z, P' >$ *está na Forma Normal de Chomsky se estiver na Forma Normal de Chomsky Pura ou se* $< \Omega, \Sigma, S, P >$ *estiver na Forma Normal de Chomsky Pura e* $P' = P \cup \{Z \rightarrow \Sigma, Z \rightarrow S\}$

Uma das vantagens da FNC é que, dado um string de tamanho n, podemos calcular quantas derivações são necessárias para gerar esse string. Esse fato é usado, por exemplo, na demonstração do Teorema Pumping para Linguagens Livres de Contexto, que veremos no final deste capítulo.

Vamos então ver como uma Gramática pode ser transformada para obter-se a Forma Normal de Chomsky equivalente. Partimos de uma Gramática $G = < \Omega, \Sigma, S, P >$ supondo que esta não contém símbolos inúteis nem produções unitárias. Então, toda produção de G é do tipo $A \rightarrow x$, onde $|x| > 1$ ou $x \in \Sigma$. No caso de $x \in \Sigma$, a produção não precisa ser alterada pois já se encontra na forma aceita pela FNC. Se $|x| > 1$ iremos iniciar a transformação substituindo cada elemento terminal a em x por um não terminal X_a e incluir a produção $X_a \rightarrow a$, eliminando símbolos terminais das produções com mais do que um símbolo do lado direito. Vamos tomar como exemplo a Gramática $G = < \{Z, S, A, B, C\}, \{a, b, c, d, e, f, g\}, Z, P >$ onde P é dado por

$Z \rightarrow S \mid \lambda$
$S \rightarrow SABC \mid be \mid CBh$
$A \rightarrow aaC$
$B \rightarrow Sf \mid ggg$
$C \rightarrow cA \mid d$

Nas produções de Z não precisamos fazer quaisquer alterações pois essas produções são aceitas na FNC e portanto devem continuar existindo. Já as demais produções são alteradas conforma mostrado abaixo

$Z \rightarrow S \mid \lambda$
$S \rightarrow SABC \mid X_bX_e \mid CBX_h$
$X_b \rightarrow b$
$X_e \rightarrow e$
$X_h \rightarrow h$
$A \rightarrow X_aX_aC$
$X_a \rightarrow a$
$B \rightarrow SX_f \mid X_gX_gX_g$
$X_f \rightarrow f$

$X_g \rightarrow g$
$C \rightarrow X_c A \mid d$
$X_c \rightarrow c$

Em seguida, vamos dividir uma produção cujo lado direito tenha mais do que 2 símbolos (não terminais) em subproduções. Se tivermos $A \rightarrow B_1 B_2 ... B_n$, dividimos em $A \rightarrow B_1 Y_1$ e $Y_1 \rightarrow B_2 B_3 ... B_n$. Aplicamos o mesmo procedimento para Y_1 obtendo $Y_1 \rightarrow B_2 Y 2$ e $Y_2 \rightarrow B_3 ... B_n$, até que cheguemos a $Y_{n-2} \rightarrow B_{n-1} B_n$. No exemplo acima teríamos as produções divididas da seguinte forma:

$Z \rightarrow S \mid \lambda$
$S \rightarrow SY_1 \mid X_b X_e \mid CY_3$
$Y_1 \rightarrow AY_2$
$Y_2 \rightarrow BC$
$Y_3 \rightarrow BX_h$
$X_b \rightarrow b$
$X_e \rightarrow e$
$X_h \rightarrow h$
$A \rightarrow X_a Y_4$
$Y_4 \rightarrow X_a C$
$X_a \rightarrow a$
$B \rightarrow SX_f \mid X_g Y_5$
$Y_5 \rightarrow X_g X_g$
$X_f \rightarrow f$
$X_g \rightarrow g$
$C \rightarrow X_c A \mid d$
$X_c \rightarrow c$

Obtemos assim a Gramática $G' = < \{Z, S, A, B, C, X_a, X_b, X_c, X_e, X_f, X_g, Y_1, Y_2, Y_3, Y_4, Y_5\}, \{a, b, c, d, e, f, g\}, Z, P >$, onde P é o conjunto de produções mostrado acima.

7.4.5 Eliminação de Recursão à Esquerda

Outra característica que pode ser indesejada numa GLC é a existência de recursão à esquerda. Isso ocorre quando existe uma derivação $A \stackrel{*}{\Rightarrow} A\alpha$, ou seja, quando um não terminal A pode derivar uma forma sentencial cujo símbolo mais à esquerda é o próprio A. Nesta seção veremo como eliminar a recursão à esquerda direta, ou seja, veremos como eliminar produções do tipo $A \rightarrow A\alpha$. A recursão indireta ocorre quando o A à esquerda é derivado através de outro não terminal, como por exemplo, $A \rightarrow B\alpha$ e $B \rightarrow A\beta$. A eliminação da recursão esquerda indireta pode ser sensivelmente mais complexa e pode ser feita colocando-se a Gramática na Forma Normal de Greibach, como será visto adiante neste capítulo.

A idéia dessa transformação é mudar uma recursão à esquerda para uma recursão à direita, que não é problemaática como a recursão á esquerda. Por exemplo, para as produções

$$A \rightarrow Aa \mid b$$

podemos gerar o conjunto de strings ba^*. A cada aplicação de $A \rightarrow Aa$, incluimos

um a à esquerda da forma sentencial, até que $A \rightarrow b$ termine a derivação. O mesmo pode ser feito com as produções

$$A \rightarrow b \mid bY$$

$$Y \rightarrow a \mid aY$$

que também gera ba^*. Porém a derivação é feita colocando-se letras do início para o fim da cadeia. Primeiro a letra b, através da produção $a \rightarrow bY$, depois os a's mais à esquerda até que a derivação termine com a aplicação de $Y \rightarrow a$. Esse segundo conjunto de produções substituiu a recursão à esquerda por uma recursão à direita, em $Y \rightarrow aY$.

Seja a GLC sem produções unitárias $G = < \Omega, \Sigma, S, P >$ e o não terminal $X \in \Omega$. As produções de X podem ser divididas em dois subconjuntos disjuntos: o conjunto $R_x = \{X \rightarrow X\alpha_1, X \rightarrow X\alpha_2, ..., X \rightarrow X\alpha_n\}$ que o conjunto das produções recursivas à esquerda e $N_x = \{X \rightarrow \beta_1, X \rightarrow \beta_2, ..., X \rightarrow \beta_m\}$ que o conjunto de produções não recursivas à esquerda. Para eliminar as produções recursivas à esquerda de X vamos tomar um novo não terminal $Y \notin \Omega$ e construir a Gramática $G = < \Omega \cup \{Y\}, \Sigma, S, P' >$, onde P' é dado por:

$$\begin{array}{ll} P' = & (P - R_x) \cup \{X \rightarrow \beta_1 Y \mid \beta_2 Y \mid ... \mid \beta_m Y\} \cup \\ & \{Y \rightarrow \alpha_1 \mid \alpha_2 \mid ... \mid \alpha_n\} \cup \\ & \{Y \rightarrow \alpha_1 Y \mid \alpha_2 Y \mid ... \mid \alpha_n Y\} \end{array}$$

Tomemos como exemplo a Gramática $G = < \{A, B\}, \{a, b, c\}, A, \{A \rightarrow AB \mid BA \mid a, B \rightarrow b \mid c\} >$. Para eliminar a recursão esquerda de A, teríamos $R_A = \{A \rightarrow AB\}$ e $N_A = \{A \rightarrow BA \mid a\}$. Aplicando a regra acima obteríamos a Gramática $G' = < \{A, B, Y\}, \{a, b, c\}, A, P' >$, onde P' é o seguinte conjunto de produções:

$A \rightarrow BA \mid a \mid BAY \mid aY$
$B \rightarrow b \mid c$
$Y \rightarrow BY \mid B$

7.4.6 Forma Normal de Greibach

Uma Gramática na Forma Normal de Greibach é descrita abaixo.

Definição 7.5 *Uma Gramática $G = < \Omega, \Sigma, S, P >$ está na Forma Normal de Greibach Pura se P contém produções somente do tipo $A \rightarrow a\alpha$ $A \in \Omega$, $a \in \Sigma$ e $\alpha \in (\Omega \cup \Sigma)^*$. Uma Gramática $G =< \Omega \cup Z, \Sigma, Z, P' >$ está na Forma Normal de Greibach se estiver na Forma Normal de Greibach Pura ou se $< \Omega, \Sigma, S, P >$ estiver na Forma Normal de Greibach Pura e $P' = P \cup \{Z \rightarrow \Sigma, Z \rightarrow S\}$*

Neste tipo de Gramática toda produção inicia com um símbolo terminal. Assim, uma das características que ficam garantidas por tais gramáticas é a não existência de recursão esquerda (direta ou indireta). Para transformar uma Gramática na sua Forma Normal de Greibach vamos precisar eliminar recursão

esquerda direta, como visto na seção anterior e vamos também precisar eliminar alguns não terminais inconvenientes que iniciam uma produção, como o caso de B na produção $A \rightarrow B\alpha$.

Considere a produção $A \rightarrow B\alpha$ e considere que $B \rightarrow \beta_1 \mid \beta_2 \mid ... \mid \beta_n$ são as produções do não terminal B. Para eliminar B da produção $A \rightarrow B\alpha$ devemos substituir B por cada uma das suas produções. Eliminamos assim $A \rightarrow B\alpha$ e ficamos com as produções $A \rightarrow \beta_1\alpha \mid \beta_2\alpha \mid ... \mid \beta_n\alpha$. Esse tipo de substituição será utilizado no algoritmo de transformação para a Forma Normal de Greibach, descrito a seguir.

Dada $G = < \Omega, \Sigma, S, P >$, que não possui símbolos inúteis ou produções unitárias, para criar G' na FNG, vamos proceder em duas fases. Primeiramente vamos numerar cada um dos não terminais de Ω, dando ao símbolo inicial S o número 1, e aos demais uma numeração qualquer. Para facilitar, vamos considerer que $\Omega = \{S_1, S_2, ..., S_n\}$. Nesta primeira fase da transformação vamos alterar as produções de G para que elas satisfaçam a "condição de crescimento". Para isso, toda produção deverá ter uma das seguintes formas:

- $S_i \rightarrow a\alpha$, $a \in \Sigma$ e $\alpha \in (\Omega \cup \Sigma)^*$
- $S_i \rightarrow S_j\alpha$, $S_j \in \Omega$, $i > j$ e $\alpha \in (\Omega \cup \Sigma)^*$

Isso quer dizer que para atender a condição de crescimento a produção deve iniciar com um símbolo terminal (nesse caso a produção já está na forma exigida pela FNG) ou, caso ela comece com um não terminal S_j, então j que é a numeração dada a esse não terminal, deve ser menor que a numeração dada ao não terminal do lado esquerdo da produção.

Para alcançar esse objetivo aplicaremos o seguinte procedimento para cada produção, começando com o não terminal de maior numeração, até chegarmos ao símbolo inicial, que tem a menor numeração de todos. Para cada produção $S_i \rightarrow S_j\alpha$ faremos:

a. se $i > j$, nada a fazer

b. se $i < j$, eliminamos S_j do início da produção como visto acima. Essa eliminação pode fazer surgirem outras produções do tipo $S_i \rightarrow S_k\beta$. Porém sabemos que $j > i$ e que por isso todas as produções de S_j satisfazem a condição de crescimento e portanto $k < j$. Assim podemos continuar transformando cada uma dessas produções, fazendo o número do não terminal inicial diminuir até satisfazer a condição ou até que $i = j$. Nesse último caso aplicamos c para retirar a recursão esquerda.

c. se $i = j$, vamos eliminar a recursão à esquerda do símbolo S_i conforme visto na seção anterior. O símbolo não terminal introduzido para esse fim recebe numeração superior a todos os demais não terminal existentes. Isso faz com que todas as produções desse novo não terminal satisfaçam a condição de crescimento. Além disso, as demais produções de S_i, que não são recursivas devem já ter sido alteradas para satisfazer a condição de crescimento e portanto a eliminação da recursão não deve introduzir produções que não satisfazem essa condição

Vamos tomar como exemplo a Gramática $G = < \{S, A, B\}, \{a, b, c, d, e\}, S, \{S \rightarrow SAc \mid BbB, A \rightarrow SS \mid d, B \rightarrow Ae\} >$. Para facilitar, vamos trocar os nomes dos não terminais, chamando S de S_1, A de S_2 e B de S_3, o que nos dá a numeração requerida pelo algoritmo. Temos então a Gramática $G' = < \{S_1, S_2, S_3\}, \{a, b, c, d, e\}, S_1, \{S_1 \rightarrow S_1S_2c \mid S_3bS_3, S_2 \rightarrow S_1S_1 \mid d, S_3 \rightarrow S_2e\} >$. Iniciamos então o procedimento para garantir a condição de crescimento com o não terminal S_3. Analisando a produção $S_3 \rightarrow S_2e$ vemos que esta satisfaz a condição de crescimento e portanto não precisa ser alterada. O mesmo acontece com as produções do não terminal S_2. Note-se que podemos escolher a ordem em que os não terminais são numerados para minimizar o número de transformações requeridas. Se tivessemos numerado A e B na ordem inversa teríamos que fazer transformações na produção de B que ficaria $S_1 \rightarrow S_2e$.

Chegamos ao não terminal S_1 que possui duas produções que não satisfazem a condição de crescimento. Vamos iniciar com

$$S_1 \rightarrow S_3bS_3$$

deixando a outra, que é recursiva a esquerda para depois, quando eliminaremos todas as produções recursivas à esquerda de uma vez. Para retirar S_3 do inicio da produção iremos substituí-lo pelas suas produções obtendo

$$S_1 \rightarrow S_2ebS_3$$

Essa nova produção ainda não satisfaz a condição de crescimento. Note porém que a numeração do não terminal inicial diminuiu. Como já foi dito, isso sempre acontece pois sabe-se que as produções de S_3 já satisfazem a condição. Aplicamos então a remoção de S_2 obtemos as produções

$$S_1 \rightarrow debS_3 \mid S_1S_1ebS_3$$

Obtivemos então mais duas produções de S_1, uma delas recursiva à esquerda. Temos até agora as seguintes produções:

$S_1 \rightarrow S_1S_2c \mid S_1S_1ebS_3 \mid debS_3$
$S_2 \rightarrow S_1S_1 \mid d$
$S_3 \rightarrow S_2e$

Vamos agora eliminar as produções de S_1 que ainda não satisfazem a condição de crescimento, ou seja, as recursivas à esquerda. Para elas vamos aplicar o algoritmo de eliminação de recursão visto na seção anterior. Temos:

$$R_{S_1} = \{S_1 \rightarrow S_1S_2c \mid S_1S_1ebS_3\}$$

$$N_{S_1} = \{S_1 \rightarrow debS_3\}$$

Alterando, as produções ficam

$$S_1 \rightarrow debS_3 \mid debS_3S_4$$

Note-se que as produções não recursivas de S_1 já haviam sido modificadas para satisfazerem a condição de crescimento e por isso a alteração acima não introduz novos problemas. Adiconamos também o novo não terminal S_4 com as seguintes produções:

$$S_4 \rightarrow S_2c \mid S_1ebS_3 \mid S2_cS_4 \mid S_1ebS_3S_4$$

que satisfazem a condição de crescimento pois S_4 possui a maior numeração de todos os não terminais.

Terminamos a primeira fase do algoritmo com as seguintes produções:

$S_1 \rightarrow debS_3 \mid debS_3S_4$
$S_2 \rightarrow S_1S_1 \mid d$
$S_3 \rightarrow S_2e$
$S_4 \rightarrow S_2c \mid S_1ebS_3 \mid S2_cS_4 \mid S_1ebS_3S_4$

Ainda assim temos produções que iniciam com não terminais e precisam ser modificadas. Porém nesse ponto sabemos que todas as produções de S_1 iniciam com terminais uma vez que satisfazem a condição de crescimento e que não existe um não terminal cuja numeração seja inferior à sua. As produções de S_2 por sua vez iniciam com terminais ou com S_1. As que iniciam com S_1 podem ser eliminadas substituindo S_1 por suas produções, que iniciam com terminais fazendo com que as produções de S_2 também iniciem todas com terminais. Continuando esse processo na ordem crescente de numeração iremos então fazer com que todas as produções iniciem com símbolos terminais, como requerido pela FNG.

Voltando ao exemplo, confirmamos que todas as produções de S_1 iniciam com terminais. Passamos então a S_2 que possui a produção $S_2 \rightarrow S_1S_1$ que deve ser modificada. Substituindo S_1 obtemos:

$$S_2 \rightarrow debS_3S_4S_1 \mid debS_3S_1 \mid d$$

Para S_3 temos $S_3 \rightarrow S_2e$ que deve ser substituida por

$$S_3 \rightarrow debS_3S_4S_1e \mid debS_3S_1e \mid de$$

E cada uma das produções de S_4 precisam ser substituidas. $S_4 \rightarrow S_2c$ é substituida por

$$S_4 \rightarrow debS_3S_4S_1c \mid debS_3S_1c \mid dc$$

$S_4 \rightarrow S_1ebS_3$ é substituida por

$$S_4 \rightarrow debS_3ebS_3 \mid debS_3S_4ebS_3$$

$S_4 \rightarrow S_2cS_4$ é substituido por

$$S_4 \rightarrow debS_3S_4S_1cS_4 \mid debS_3S_1cS_4 \mid dcS_4$$

$S_4 \rightarrow S_1ebS_3S_4$ é substituido por

$$S_4 \rightarrow debS_3ebS_3S_4 \mid debS_3S_4ebS_3S_4$$

Terminamos portanto com a Gramática $G'' = < \{S_1, S_2, S_3S_4\}, \{a, b, c, d, e\}, S_1, P'' >$, onde P'' é o seguinte conjunto de produções:

$S_1 \rightarrow debS_3 \mid debS_3S_4$
$S_2 \rightarrow debS_3S_4S_1 \mid debS_3S_1 \mid d$
$S_3 \rightarrow debS_3S_4S_1e \mid debS_3S_1e \mid de$
$S_4 \rightarrow debS_3S_4S_1c \mid debS_3S_1c \mid dc$

$S_4 \rightarrow debS_3ebS_3 \mid debS_3S_4ebS_3$
$S_4 \rightarrow debS_3S_4S_1cS_4 \mid debS_3S_1cS_4 \mid dcS_4$
$S_4 \rightarrow debS_3ebS_3S_4 \mid debS_3S_4ebS_3S_4$

Notem que todas essas alterações façam com que símbolo se tornem inúteis. É o caso, por exemplo de S_2 que pode ser eliminado, de acordo com o algoritmo visto anteriormente.

7.5 Teorema Pumping

Mostramos no início deste capítulo que existem certas linguagens que não podem ser definidas através de Autômatos Finitos e para as quais utilizamos Gramática Livre de Contexto. Vamos mostrar agora, que mesmo as GLC's não são capazes de representar qualquer tipo de linguagens, ou seja, existem linguagens que não são Livres de Contexto. Para isso usaremos o Teorema Pumping, que é dado abaixo:

Teorema 7.2 *(Teorema Pumping) Seja L uma Linguagem Livre de Contexto. Então, existe um inteiro k tal que todo string $z \in L$, $|z| \geq k$ pode ser escrito como uma concatenação $z = uvwxy$ onde $|vwx| \leq k$, $|vx| > 0$ e $uv^iwx^iy \in L$ para qualquer $i \geq 0$.*

Similarmente ao Lema Pumping, esse teorema afirma que qualquer string $z \in L$ que seja "suficientemente grande" deve poder ser escrito como uma concatenação de 5 substrings u, v, w, x e y de modo que uv^iwx^iy também pertença a L, para qualquer valor de i. Para demonstrar esse teorema vamos tomar uma Gramática $G = < \Omega, \Sigma, S, P >$ na FNC e o string $z \in L(G)$. Vamos escolher z tal que $|z| \geq 2^{k+1}$ onde $k = |\Omega|$. Ao construirmos a Árvore de Derivação para z, obtemos uma árvore binária que tem profundidade, no mínimo $k+1$, dado o tamanho de z. Assim, existe um caminho da raiz da árvore até uma folha que possui $k+2$ nós, sendo $k+1$ nós internos, rotulados com símbolos não terminais. Como existem apenas k simbolos não terminal distintos, então algum não terminal R irá se repetir nos últimos $k+1$ nós internos da árvore. Temos então a situação da Figura 7.9

Vemos pela Árvore de Derivação que $S \stackrel{*}{\Rightarrow} uRy$ e que $R \stackrel{*}{\Rightarrow} vRx$ e $R \stackrel{*}{\Rightarrow} w$. Com G está na FNC, não possui produção unitária ou vazia então $|vRx| > |R|$, ou seja, $|vx| > 1$. Além disso, $|vwx| \leq 2^{k+1}$ pois a segunda ocorrência de R (de baixo para cima) acorre numa altura máxima $k+1$, sendo portanto raiz de uma árvore binária que tem no máximo 2^{k+1} folhas. Finalmente, se aplicarmos i vezes a derivação $R \stackrel{*}{\Rightarrow} vRx$ e depois $R \stackrel{*}{\Rightarrow} w$, obteremos um string uv^iwx^iy que também pertence a $L(G)$, satisfazendo assim todos os requisitos do Teorema Pumping.

Da mesma maneira que para Linguagens Regulares, o teorema Pumping serve para mostrar quando uma determinada linguagem não é Livre de Contexto. Para isso devemos mostrar que não existe um valor de k para o qual todos os strings maiores do que k satisfaçam o teorema. A diferença nesse caso – que dificulta

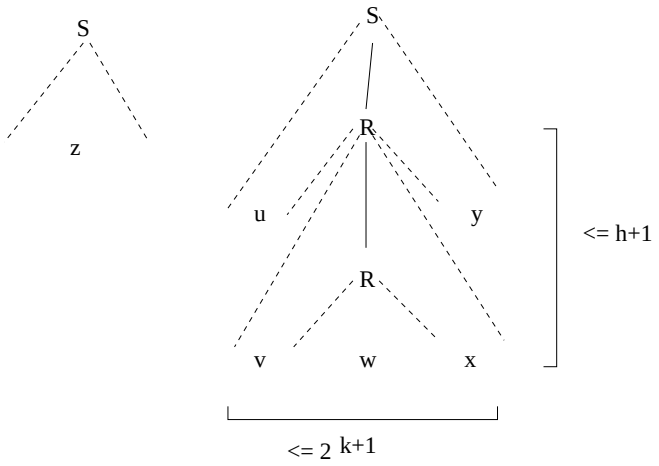


Figura 7.9: Árvore de Derivação para $z = uvwxy$

sensivelmente a utilização do teorema – é que não se sabe onde, dentro do string deve "bombear" letras para tentar mostrar que um string não satisfaz o teorema. No caso das Linguagens Regulares isso era feito sempre no início da cadeia.

Vamos tomar como exemplo a linguagem $L = \{a^i b^i c^i \mid i \geq 0\}$ e mostrar que ela não é Livre de Contexto. Vamos inicialmente supor que exista k conforme determinado pelo teorema. Tomamos então o string $z = a^k b^k c^k$ e tentaremos achar uma decomposição $uvwxy$ que satisfaça o Teorema. Para isso vamos considerar duas possibilidades:

- v ou x (pelo menos um deles) possui mais do que um tipo de letra
- v e x não possuem mais do que um tipo de letra.

No primeiro caso, o substring v (ou x) possui então um ab ou um bc, ou seja v (ou x) é da forma $\alpha ab\beta$ ou $\alpha bc\beta$. Assim, a cadeia uv^2wx^2y possui um b antes de um a ou um c antes de um b pois v^2 (ou x^2) é da forma $\alpha ab\beta\alpha ab\beta$ ou $\alpha bc\beta\alpha bc\beta$, fazendo com que o string não pertença a L.

No segundo caso o string uv^2wx^2y – uma vez que v e x não podem ser ambos vazios – irá aumentar o número de ocorrências de uma das letras, talvez duas delas, mas nunca das três. Portanto $uv^2wx^2y \notin L$. Assim, não pudemos encontrar uma decomposição para $a^k b^l c^k$, mostrando que L não é Livre de Contexto.

7.6 Propriedades de Fechamento

Assim, como para as Linguagens Regulares, podemos estabelecer algumas propriedades que permitam combinar Linguagens Livres de Contexto. Abaixo estão algumas dessas propriedades. As demonstrções para elas podem ser encontradas em [CAR89]

Sejam L_1 e L_2 duas Linguagens Livres de Contexto.

- $L_1 \cup L_2$ é Livre de Contexto
- $L_1 \cdot L_2$ é Livre de Contexto
- L_1^* é Livre de Contexto
- $L_1 \cap L_2$ não é necessariamente Livre de Contexto
- $\overline{L_1}$ não é necessariamente Livre de Contexto
- toda Linguagem Livre de Contextosobre uma alfabeto unitário é uma Linguagem Regular

7.7 Uma GLS para Pascal

Nesta seção será uma GLC para a linguagem Pascal, extraída de [SUD97]. As produções dessa Gramática utilizam uma notação conhecida como **Backus-Naur form (BNF)**. Esse tipo de notação permite que se utilize a construção

$$A \rightarrow \{\alpha\}$$

significando a repetição do string α zero ou mais vezes. Assim, a produção $A \rightarrow \{\alpha\}$ representa

$$A \stackrel{\alpha}{\rightarrow} A$$

. Essa Gramática utiliza produções vazias.

Para diferenciar terminais e não terminal utilizamos diferentes fontes: **terminais** e <*não terminais*>. O símbolo inial é <*program*>.

<*program*> → <*program heading*> ; <*program block*>
<*program block*> → <*block*>

7.8 Exercícios

7.1 *Mostre que as seguintes linguagens não são regulares*

a. *conjunto de palindromes sobre* $\{a, b\}$

b. $\{x \in \{a, b\}^* \mid x = a^n b^m, n < m\}$

c. $\{x \in \{a, b, c\}^* \mid x = a^i b^j c^{2j}, i \geq 0, j \geq 0\}$

d. $\{x \in \{a, b\}^* \mid x = ww, w \in \{a, b\}^*\}$

7.2 *Addicione o operador* $**$ *(exponenciação) na Gramática de expressões vista neste capítulo. Addicione também os operadores unários* $+$ *e* $-$.

7.3 *Dê GLC's para as seguintes linguagens:*

a. $\{x \in \{a,b\}^* \mid |x|_a < |x|_b\}$

b. $\{x \in \{a,b\}^* \mid |x|_a \geq |x|_b\}$

c. $\{x \in \{a,b\}^* \mid x = x^r\}$

d. $\{a^j b^k c^m \mid j \geq 3 \wedge k = m\}$

e. $\{x \in \{a,b\}^* \mid |x|_a = 2|x|_b\}$

f. $\{x \in \{a,b\}^* \mid |x|_a \neq |x|_b\}$

g. *conjunto de todas as expressões podfixas sobre* $\{a, b, +, -\}$

h. $\{a^n b^n c^m d^m \mid n, m \geq 0\}$

i. $\{a^i b^j c^j d^i \mid i, j \geq 0\}$

7.4 *Elimine as produções vazias das seguintes Gramáticas:*

a. $< \{S, B, C\}, \{a, b, c\}, S, \{S \to aB \mid abC, B \to bc \mid C, C \to c \mid \lambda\} >$

b. $< \{S, B, A\}, \{a, b, c\}, S, \{S \to cBA \mid B, A \to cB \mid AbbS, B \to aaa \mid \lambda\} >$

7.5 *Mostre que a GLD* $G = < \{S, A, B\}, \{a\}, S, \{S \to A \mid B, A \to aaA \mid \lambda, B \to aaaB \mid \lambda\} >$ *é ambígua. Descreva uma maneira de eliminar a ambiguidade e aplique-a a* G.

7.6 *Transforme para a Forma Normal de Chomsky:*

a. $< \{ABC\}, \{a, b, c\}, S, \{S \to aB \mid abC, B \to bc, C \to c\} >$

b. $< \{S, A, B\}, \{a, b, c\}, S, \{S \to cBA \mid B, A \to cB \mid AbbS, B \to aaa\} >$

c. $< \{R\}, \{a, b, \epsilon, \emptyset, \cup, (,), *, \cdot\}, R, \{R \to a \mid b \mid \epsilon \mid \emptyset \mid (R \cdot R) \mid (R \cup R) \mid R^*\} >$

d. $< \{T\}, \{a, +, -\}, T, \{T \to a \mid T + T \mid T - T\} >$

7.7 *Transforme para a Forma Normal de Greibach:*

a. $< \{S, A\}, \{a, b, c\}, S, \{S \to ASa \mid Ab, A \to SA \mid c\} >$

b. $< \{S, A, B\}, \{a, b, c\}, S, \{S \to BS \mid Aa, A \to ba, B \to Ac\} >$

c. $< \{S, A, B\}, \{a, b, c, d, e\}, S, \{S \to SAc \mid dB, A \to SSb|BAa, B \to Be\} >$

Capítulo 8

Autômatos de Pilha

Continuando com o estudo das Linguagens Livres de Contexto estudaremos neste capítulo as máquinas que são capazes de reconhecer este tipo de linguagens. Estudaremos os Autômatos de Pilha, que através da adição de "espaço de armazenamento" extra, incrementa o poder dos Autômatos Finitos. Veremos também que as linguagens reconhecidas por Autômatos de Pilha podem também ser definidas por Gramáticas Livres de Contexto e vice-versa, estabelecendo portanto a equivalência entre estes dois modelos.

8.1 Definições Básicas

Como vimos, um Autômato Finito não é capaz de reconhecer a linguagem $\{a^i b^i \mid i \geq 0\}$ pela incapacidade de "recordar" uma quantidade infinita de informação sobre a cadeia analisada. Os Autômatos de Pilha, ao contrário, possuem uma pilha que é utilizada para armazenar essa informação adicionando assim poder aos AF's. Vejamos a sua definição:

Definição 8.1 *Um Autômato de Pilha não determinístico (AP) é uma sextupla* $P = < \Sigma, \Gamma, S, S_0, \delta, B >$ *onde:*

- Σ *é o alfabeto de entrada do AP*
- Γ *é o alfabeto da pilha*
- S *é o conjunto finito não vazio de estados do AP*
- S_0 *é o estado inicial,* $S_0 \in S$
- δ *é a função de transição de estados,* $\delta : S \times (\Sigma \cup \{\lambda\}) \times \Gamma \rightarrow$ *conjunto de subconjuntos finitos de* $S \times \Gamma^*$
- B *é o símbolo da base da pilha,* $B \in \Gamma$

A Figura 8.1 representa uma visualização conceitual de um AP. Ele possui uma fita de entrada, de onde as letras de uma cadeia são lidas e passadas para o Controle Finito de Estados. Possui também uma pilha que também pode ser acessada pelo Controle Finito. Este possui acesso somente à última letra colocada na pilha, chamada de **topo** da pilha. Todos os elementos da pilha são letras do alfabeto Γ.

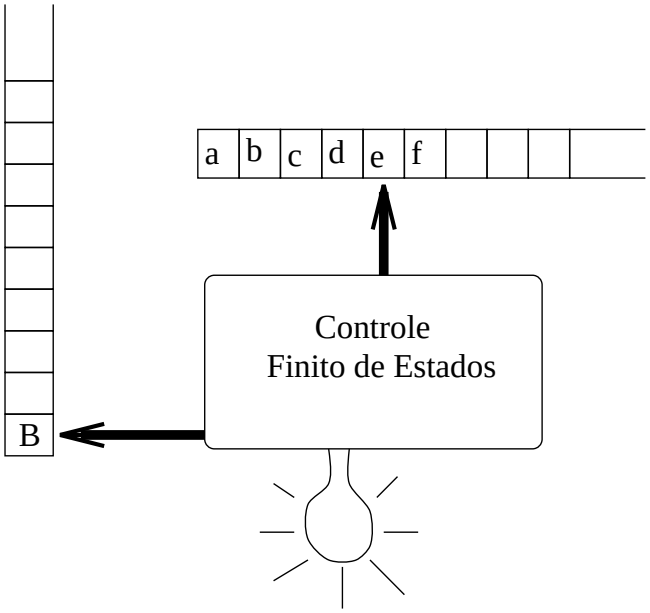


Figura 8.1: Abstração de um Autômato de Pilha como reconhecedor de cadeias

Ao contrário da fita de entrada, a pilha pode ser lida e também alterada pelo Controle Finito. Além de verificar o conteúdo do topo da pilha, o AP pode ainda retirar o símbolo do topo e colocar em seu lugar uma cadeia $\alpha \in \Gamma^*$. Se $\alpha = A, A \in \Gamma$, então o símbolo do topo é substituído por A a cabeça de leitura/escrita da pilha continua posicionada no mesmo lugar. Se $\alpha = A_1 A_2 .. A_n$, $n > 1$ então o símbolo do topo da pilha é retirado, sendo A_n colocado no seu lugar, A_{n-1} na posição seguinte, e assim por diante. A cabeça é deslocada para a posição ocupada por A_1 que é então o novo topo da pilha. Se $\alpha = \lambda$ então o símbolo do topo da pilha é retirado, fazendo a pilha decrescer e o topo ficar numa posição imediatamente inferior à anterior.

Assim como nos AF's, o funcionamento dos AP's é determinado pela função de transição. A diferença aqui é que δ é função do estado corrente, da letra corrente na fita de entrada e do símbolo no topo da pilha. Além disso, ela determina não só o próximo estado que o AP assume mas também como o topo da pilha deve ser substituído. O AP inicia sua operação num estado especial denotado por S_0 e com um único símbolo na pilha, denotado por B.

A **configuração** de um AP é dada por uma tripla $< s, x, \alpha >$ onde s é o estado corrente, x é a cadeia da fita que falta ser processada e α é o conteúdo da pilha, com o topo no inicio de α. O AP **anda** ou **move-se** de uma configuração para outra através da aplicação da função de transição. Se o AP está na

configuração $< s, ay, A\beta >$ e temos que $\delta(s, a, A) =< t, \gamma >$, então o AP move-se para a configuração $< t, y, \gamma\beta >$ e denota-se $< s, ay, A\beta > \vdash < t, y, \gamma\beta >$. Se o AP move-se de uma configuração $< s_1, x_1, \alpha_1 >$ para uma configuração $< s_2, x_2, \alpha_2 >$ por meio de um número finito (0 ou mais) de movimentos, denotamos $< s_1, x_1, \alpha_1 > \vdash^* < s_2, x_2, \alpha_2 >$. Se o valor de δ para uma determinada configuração é $\emptyset$ então o AP pára.

Vamos tomar como exemplo o AP $P = < \{a, b\}, \{A, B\}, \{S, R\}, S, \delta, B >$ onde δ é dada por:

$$\delta(S, a, B) = \{< S, A >\}$$
$$\delta(S, a, A) = \{< S, AA >\}$$
$$\delta(S, b, A) = \{< R, \lambda >\}$$
$$\delta(R, b, A) = \{< R, \lambda >\}$$

Os demais valores de δ são $\emptyset$.

Então P inicia sua execução na configuração dada na Figura 8.2a. Ao processar a letra a, aplica-se a função $\delta(S, a, B)$, colocando o AP no estado S e substituindo o topo B por A, chegando à configuração da Figura 8.2b. Processando o segundo a chega-se à Figura 8.2c e processando-se o b à Figura 8.2d.

O leitor deve ter notado que, ao contrário dos AF's, os AP's não possuem estados finais que devem ser alcaçados para que uma cadeia seja aceita. Nesse caso, um string x é aceito se, ao chegar ao final da cadeia de entrada, a pilha estiver vazia, independentemente do estado em que o AP se encontra. Formalmente temos:

Definição 8.2 *Dado o AP $P = < \Sigma, \Gamma, S, S_0, \delta, B >$ e o string x sobre Σ, diz-se que x é* **aceito** *por P sse existe $s \in S$ tal que $< S_0, x, B > \vdash^* < s, \lambda, \lambda >$. Caso contrário, x é* **rejeitado**.

Note que trabalhamos com AP's não determinísticos. Por isso, numa determinada configuração podem existir diversos movimentos possíveis. Se algum deles leva o AP a $< s, \lambda, \lambda >$ então o string é aceito. O string só é rejeitado se nenhuma seqüência de movimentos leva a $< s, \lambda, \lambda >$. A definição da linguagem aceita por um AP é:

Definição 8.3 *Dado ap AP $P = < \Sigma, \Gamma, S, S_0, \delta, B >$, a linguagem $L(P)$ definida por P é $\{x \in \Sigma^* \mid \exists s \in S \ni < S_0, x, B > \vdash^* < s, \lambda, \lambda >\}$.*

Um AP pode ser representado por um diagrama de transição de estados semelhante àquele utilizado para AF's. Cada estado é representado por um círculo e cada posível movimento por um arco direcionado entre dois estados. Cada arco é rotulado com uma letra da entrada e uma letra que deve estar no topo da pilha para que o movimento possa ocorrer e com o valor do string a ser colocado no topo da pilha, em substituição à letra lá existente. Por exemplo, se tivermos $\delta(s, a, A) = \{< r, AB >\}$, teremos o seguinte diagrama:

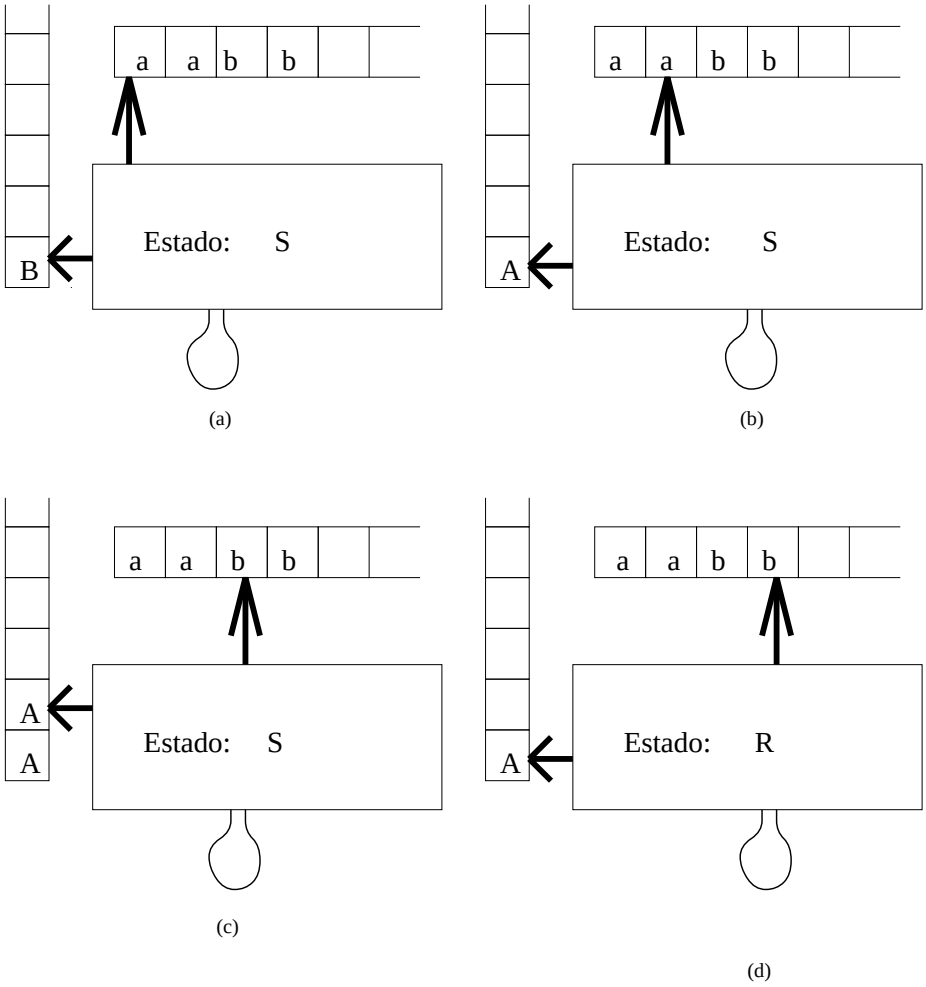
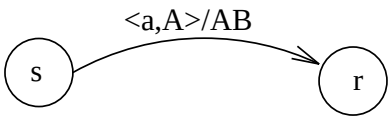
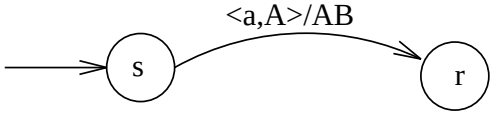


Figura 8.2: Execução de um Autômato de Pilha



Um estado inicial é marcado com um arco chegando no estado, sem nenhum estado de origem, como no diagrama abaixo.



No exemplo do AP visto anteriormente, teríamos o digrama da Figura 8.3

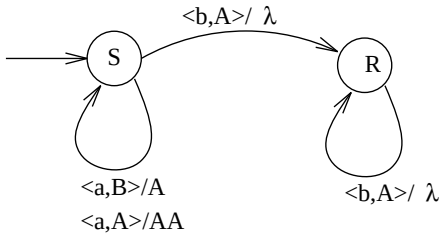


Figura 8.3: Exemplo de um diagrama de trasnsição para um AP

8.2 Exemplos de AP's

Nesta seção vamos ver exemplos de algumas linguagens representadas através de AP's. Vamos iniciar com a conhecida $\{a^i b^i \mid i \geq 0\}$. O AP $P = < \{a, b\}, \{A, B\}, \{S, R\}, S, \delta, B >$ cuja função δ é mostrada na Figura 8.3 reconhece uma linguagem semelhante à desejada. No estado S esse AP coloca na pilha um A para cada letra a lida na fita. Ao encontrar um b na entrada o AP passa para o estado R onde irá desempilhar um A para cada b lido. Se o número de b's for exatamente igual ao número de A's – e consequentemente ao número de a's lidos – então ao final do string a pilha estará vazia e portanto o string será aceito. Se o número de b's for maior que o número de A's na pilha então esta ficara vazia antes que o string da entrada tenha sido totalmente processado, fazendo o AP parar prematuramente, sendo a cadeia rejeitada. Se o número de b's for menor do que o número de A's, ao final do string ainda restarão A's na pilha e a cadeia será rejeitada. Se um b inicia a cadeia – estado S e B no topo da pilha – ou se um a aparece depois de um b – estado R – não existe movimento possível e o AP pára, rejeitando a cadeia.

Esse AP porém não reconhace a linguagem desejada pois não aceita a cadeia vazia. Para aceitá-la precisamos alterar a função de transição incluindo o movimento $\delta(s, \lambda, B) = \{< s, \lambda >\}$ representado na Figura 8.4. Esse movimento permite que, ao iniciar sua execução, o AP possa retirar o símbolo B da pilha sem consumir nenhuma letra da entrada. Fazendo isso, a pilha se esvazia e o AP pára. Se chegou-se ao fim da entrada, ou seja se a entrada é o próprio λ, tem-se as condições de aceitação satisfeitas, e portanto λ faz parte da linguagem do AP.

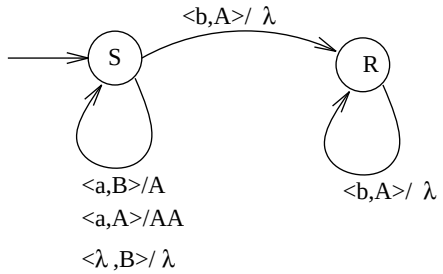


Figura 8.4: Diagrama de trasnsição para um AP que reconhece $\{a^i b^i \mid i \geq 0\}$

O segundo exemplo que veremos é $\{x \in \{a, b\}^* \mid |x|_a = |x|_b\}$. Para essa

linguagem definimos o AP $P_1 = < \{a, b\}, \{A, B, C\}, \{S\}, S, \delta, C >$, onde δ é dada na Figura 8.5.

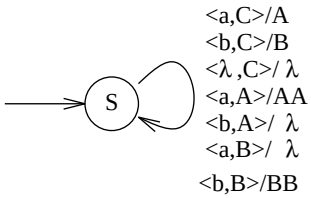


Figura 8.5: Diagrama de trasnsição para um AP tentando reconhecer $\{x \in \{a, b\}^* \mid |x|_a = |x|_b\}$

Nesse exemplo os a's e b's podem vir em qualquer ordem e por isso somente um estado deve ser utilizado. Aqui, temos dois símbolos de pilha A e B que servirão para contar o número de A's e de B's lidos. Ao ler um a da fita de entrada, o AP coloca mais um A no topo da pilha, se esta já for um A, caso contrário só retira o B que está no topo. Analogamente, um b na entrada coloca mais um B no topo se este já for um B ou retira um A caso contrário. Vamos tomar como exemplo a cadeia $aababb$. Teríamos os seguintes movimentos no AP, sendo portanto aceito:

$< S, aababb, C > \vdash < S, ababb, A > \vdash < S, babb, AA > \vdash < S, abb, A > \vdash < S, bb, AA > \vdash < S, b, A > \vdash < S, \lambda, \lambda >$

Ocorre um problema porém com strings como $abba$. Nesse caso teríamos os seguintes movimentos:

$< S, abba, C > \vdash < S, bba, A > \vdash < S, ba, \lambda >$
terminando a execução do AP e rejeitando o string que deveria ser aceito. Para solucionar esse problema vamos alterar o AP como mostrado na Figura 8.6

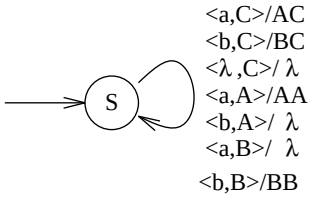


Figura 8.6: Diagrama de trasnsição para um AP que reconhece $\{x \in \{a, b\}^* \mid |x|_a = |x|_b\}$

Com essas alterações, mantém-se sempre o símbolo C no fundo da pilha. Cada vez que C aparece no topo pela remoção de um A ou um B, duas coisas podem acontecer: 1) recomeçar a contagem se tivermos mais letras na entrada; ou 2) aceitar o string através do movimento $\delta(S, \lambda, C) =< S, \lambda >$ caso a cadeia tenha terminado. Conseguimos assim o comportamento desejado do AP.

No próximo exemplo veremos como representar a linguagem $\{a^n b^i c^m \mid i = n+m\}$. Construimos o AP $P_2 =< \{a, b, c\}, \{B, N, I\}, \{S_0, S_1, S_2, S_3\}, S_0, \delta, B >$ com δ representada na Figura 8.7. No estado S_0 são lidos todos os a's e para cada um deles inclui-se um N na pilha. Ao encontrar-se um b passa-se para S_1

ou S_2. Se algum a foi lido (existem N's na pilha) passa-se para S_1 onde um N é desempilhado para cada b lido. Se nenhum a foi lido então passa-se direto de S_0 para S_2 onde são colocados I's na pilha, um para cada b lido na entrada. Em S_3 um I é retirado para cada c lido, fazendo com que o número de a's mais o número de c's seja igual ao número de b's para que o string seja aceito.

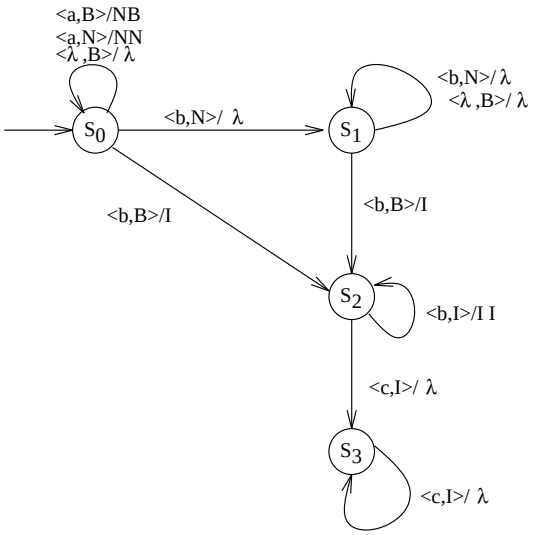


Figura 8.7: Diagrama de trasnsição para um AP que reconhece $\{a^n b^i c^m \mid i = n + m\}$

8.3 Equivalência entre AP's e GLC's

Veremos nesta seção que qualquer LLC pode ser representda através de um AP. Para isso mostraremos como transformar um AP numa GLC e vice-versa, mostrando assim que essas duas formas de representação são equivalentes.

8.3.1 Transformação de GLC para AP

Vamos mostrar aqui como uma GLC pode ser transformada num AP. Para isso partimos de uma GLG na Forma Normal de Greibach e construímos um AP com um único estado que simula as derivações mais à esquerda da Gramática.

Definição 8.4 *Dada a GLC* $G = < \Omega, \Sigma, Z, P >$ *na Forma Normal de Greibach, define-se o AP* $P_G = < \Sigma, (\Sigma \cup \Omega), \{r\}, r, \delta_G, Z >$, *onde* δ_G *é dada por:*

$$\forall a \in \Sigma, \forall \Psi \in (\Omega \cup \Sigma),$$

$$\delta_G(r, a, \Psi) = \begin{cases} \{< r, \alpha > \mid \Psi \rightarrow a\alpha \in P\} & se \Psi \in \Omega \\ \{< r, \lambda >\} & se \Psi = a \\ \{\} & c.c. \end{cases}$$

$$\delta_G(r, \lambda, \Psi) = \begin{cases} \{< r, S >, < r, \lambda >\} & se \Psi \rightarrow S \mid \lambda \in P \\ \{\} & c.c. \end{cases}$$

Essa definição estabelece que deve-se criar um AP com um único estado r e que possui como símbolos de pilha todos os terminais e todos os não terminais de G. Cada produção do tipo $A \rightarrow a\alpha$ deve dar origem a um possível movimento do AP, fazendo-se $\delta_G(r, a, A) = \{< r, alpha >\}$. Aleém disso, para cada símbolo terminal a da Gramática devemos ter o movimento $\delta_G(r, a, a) = \{< r, \lambda >\}$ que serve para retirar o terminal do topo da pilha. Finalmente, se a gramática não está na FNG Pura, então ela possui as produções $Z \rightarrow S\lambda$ para as quais devemos incluir os movimentos: $\delta_G(r, \lambda, Z) = \{< r, S >, < r, \lambda >\}$

Vamos tomar como exemplo a Gramática $G = < \{Z, S\}, \{a, b\}, Z, \{Z \rightarrow S \mid \lambda, S \rightarrow aSb \mid ab\} >$. Para essa Gramática construimos o AP $P_G = < \{a, b\}, \{Z, S, a, b\}, r, r, \delta_G, S >$ onde δ_G é definida como:

$$\delta_G(r, a, S) = \{< r, Sb >, < r, b >\}$$

$$\delta_G(r, a, a) = \{< r, \lambda >\}$$

$$\delta_G(r, b, b) = \{< r, \lambda >\}$$

$$\delta_G(r, \lambda, Z) = \{< r, S >, < r, \lambda >\}$$

Para reconhecer o string $aaabbb$ teríamos os seguintes movimentos:

$$< r, aaabbb, Z > \vdash < r, aaabbb, S > \vdash < r, aabbb, Sb >$$

Esse movimento do AP corresponde à derivação mais à esquerda $S \Rightarrow aSb$. Se S está no topo da pilha isso significa que a parte inicial da cadeia de entrada deve pode ser gerada a partir de S. Nesse caso, como S é o único símbolo na pilha, então toda a cadeia de entrada deve poder ser gerada a partir de S. Ao consumir uma letra da entrada, no caso o a, devemos substituir o topo da pilha de modo que o novo conteúdo da pilha possa casar com o restante da entrada. Temos dois possíveis movimentos. Um deles $< r, aaabbb, S > \vdash < r, aabbb, b >$ leva a uma configuração para a qual não há mais movimentos possíveis. A segunda, mostrada acima, permite que continuemos no reconhecimento da entrada:

$< r, aabbb, Sb > \vdash < r, abbb, Sbb > \vdash < r, bbb, bbb > \vdash < r, bb, bb > \vdash < r, b, b > \vdash < r, \lambda, \lambda >$

Outro exemplo de transformação é da GLC $G = < \{R\}, \{a, b, c, \epsilon, \emptyset, \cup, \cdot, *, (,)\}, R, P >$ onde P é dado por:

$$R \rightarrow a \mid b \mid c \mid \emptyset \mid \epsilon \mid (R \cup R) \mid (R \cdot R) \mid (R)^*$$

Como esta Gramática já está na FNG, não precisamos transformá-la e obtemos o AP $P = < \{a, b, c, \epsilon, \emptyset, \cup, \cdot, *, (,)\}, \{a, b, c, \epsilon, \emptyset, \cup, \cdot, *, (,), R\}, r, r, \delta, R >$ onde δ é:

$$\begin{aligned}
&\delta(r, (, R) = \{< r, R \cup R) >, < r, R \cdot R) >, < r, R)^* >\} \\
&\delta(r, a, R) = \{< r, \lambda >\} \\
&\delta(r, b, R) = \{< r, \lambda >\} \\
&\delta(r, c, R) = \{< r, \lambda >\} \\
&\delta(r, \emptyset, R) = \{< r, \lambda >\} \\
&\delta(r, \epsilon, R) = \{< r, \lambda >\} \\
&\delta(r, a, a) = \{< r, \lambda >\} \\
&\delta(r, b, b) = \{< r, \lambda >\} \\
&\delta(r, c, c) = \{< r, \lambda >\} \\
&\delta(r, \emptyset, \emptyset) = \{< r, \lambda >\} \\
&\delta(r, \epsilon, \epsilon) = \{< r, \lambda >\} \\
&\delta(r, \cdot, \cdot) = \{< r, \lambda >\} \\
&\delta(r, \cup, \cup) = \{< r, \lambda >\} \\
&\delta(r, *, *) = \{< r, \lambda >\} \\
&\delta(r, (, () = \{< r, \lambda >\} \\
&\delta(r,),)) = \{< r, \lambda >\}
\end{aligned}$$

Tomemos a cadeia $(a \cup (b \cdot c))$. A derivação segundo a Gramática seria:

$$R \Rightarrow (R \cup R) \Rightarrow (a \cup (R \cdot R)) \Rightarrow (a \cup (b \cdot R)) \Rightarrow (a \cup (b \cup c))$$

Os movimentos correspondentes do AP correspondentes a essa derivação mais à esquerda são:

$< r, (a \cup (b \cdot c)), S > \vdash < r, a \cup (b \cdot c)), R \cup R) > \vdash < r, \cup(b \cdot c)), \cup R) > \vdash$
$< r, (b \cdot c)), R > \vdash < r, b \cdot c)), R \cdot R > \vdash < r, \cdot c)), \cdot R)) > \vdash < r, c)), R)) > \vdash$
$< r,)),)) > \vdash < r, \lambda, \lambda >$

8.3.2 Transformação de AP para GLC

A definição abaixo nos mostra como um AP pode ser tranformado numa GLC. Em geral, tal transformação cria um número demasiado elevado de símbolos não terminais e de produções. Por outro lado, o número de símbolos inúteis também tende a ser grande, o que pode diminuir o número final de produções. Deve-se ressaltar porém que a aplicação manual desta definição fica, em geral, restrita a AP's bastante simples.

Definição 8.5 *Seja o AP* $P = < \Sigma, \Gamma, S, S_0, \delta, B >$. *Define-se a Gramática correspondente* $G_P = < \Omega, \Sigma, Z, P_P >$ *onde*

$$\Omega = \{Z\} \cup \{A^{st} \mid A \in \Gamma,\ s,t \in S\}$$

e P_P *é*

$\{Z \to B^{S_0 t} \mid t \in S\} \cup$
$\{A^{sr} \to a \mid r,s \in S,\ A \in \Gamma,\ a \in (\Sigma \cup \{\lambda\}),\ < r, \lambda > \in \delta(s,a,A)\} \cup$
$\{A^{sq} \to a A_1^{rt_1} A_2^{t_1 t_2} ... A_m^{t_{m-1} q} \mid A \in \Gamma,\ a \in (\Sigma \cup \{\lambda\}),\ < r, A_1 A_2 .. A_m > \in$
$\delta(s,a,A),\ s,q,r,t_1,...,t_{m-1} \in S\}$

Segundo essa definição, o conjunto de símbolos não terminais é composto por um símbolo inicial Z e por toda combinação A^{st} onde A é um símbolo da pilha de P e s,t são estados de P. Note-se que com isso pode-se obter um número muito grande de não terminais. Por exemplo, para um AP com $|\Gamma| = 2$ e $|S| = 3$ teríamos $2 * 3^2 = 18$ símbolos não terminais.

Além disso, o número de produções também pode ser bastante alto. A partir do símbolo inicial Z temos as produções $Z \to B^{S_0 t}$ onde B é o símbolo inicial da pilha de P, S_0 seu estado inicial e t representa cada um dos estados de P. O segundo tipo de produções são originadas de movimentos do tipo $\delta(s,a,A) =< r, \lambda >$. Para cada um deles é criada um aprodução $A^{sr} \to a$. Finalmente, para transições do tipo $\delta(s,a,A) =< r, A_1 A_2 ... A_n >$ são criadas produções para todos os não terminais A^{sq}, qualquer $q \in S$. Essas produções são do tipo $A^{sq} \to a A_1^{rt_1} A_2^{t_1 t_2} ... A_m^{t_{m-1} q}$, onde t_i representa cada um dos possíveis estados de S.

Vamos tomar como exemplo $P = < \{a,b\}, \{A,B\}, \{q,r\}, q, \delta, B >$ onde δ é

$$\delta(q,a,B) = \{< q, A >\}$$
$$\delta(q,a,A) = \{< q, AA >\}$$
$$\delta(q,b,A) = \{< r, \lambda >\}$$
$$\delta(r,b,A) = \{< r, \lambda >\}$$

Teríamos $G_P = < \Omega, \{a,b\}, Z, P_P >$ onde Ω é o conjunto $\{Z, B^{qq}, B^{qr}, B^{rq}, B^{rr}, A^{qq}, A^{qr}, A^{rq}, A^{rr}\}$. O primeiro tipo de produções de P_P é daquelas que têm o símbolo inicial do lado esquerdo. São:

$$Z \to B^{qq} \mid B^{qr}$$

As transições $\delta(q,b,A) =< r, \lambda >$ e $\delta(r,b,A) =< r, \lambda >$ contribuem com as produções

$$A^{qr} \to b$$
$$A^{rr} \to b$$

A transição $\delta(q,a,B) =< q, A >$ dá origem a produções para os não terminais B^{qq} e B^{qr} que são:

$$B^{qq} \to a A^{qq}$$

$$B^{qr} \rightarrow aA^{qr}$$

A transição $\delta(q, a, AA) =< q, AA >$ gera produções para os não terminais A^{qq} e A^{qr}, que são:

$$A^{qq} \rightarrow aA^{qq}A^{qq}$$
$$A^{qq} \rightarrow aA^{qr}A^{rq}$$
$$A^{qr} \rightarrow aA^{qq}A^{qr}$$
$$A^{qr} \rightarrow aA^{qr}A^{rr}$$

Eliminando os símbolos inúteis obtemos $G_P = < \{Z, B^{qr}, A^{qr}, A^{rr}\}, \{a, b\}, Z, \{Z \rightarrow B^{qr}, B^{qr} \rightarrow aA^{qr}, A^{qr} \rightarrow aA^{qr}A^{rr} \mid b, A^{rr} \rightarrow b\} >$. A derivação mais á esquerda de $aaabbb$, segundo essa Gramática é

$Z \Rightarrow B^{qr} \Rightarrow aA^{qr} \Rightarrow aaA^{qr}A^{rr} \Rightarrow aaaA^{qr}A^{rr}A^{rr} \rightarrow aaabA^{rr}A^{rr} \Rightarrow aaabbA^{rr} \Rightarrow aaabbb$

Um outro problema é que, segundo essa definição, um movimento $\delta(s, \lambda, A) = < r, \lambda >$ dá origem a um aprodução $A^{sr} \rightarrow \lambda$, o que contraria a definição de GLC. Teríamos adiconalmente que remover tal produção.

8.4 Autômato de Pilha Determinístico

A definição de AP's vista neste capítulo permite que possa-se escolher entre diferentes possibilidades de movimentos a partir de uma determinada configuração do AP. Se qualquer uma delas levar a uma condição de aceitação, então a cadeia analisada faz parte da linguagem do AP. Podemos restringir esta definição da seguinte forma:

Definição 8.6 *Um Autômato de Pilha Determinístico (APD) é um AP* $< \Sigma, \Gamma, S, S_0, \delta, B >$*, em que, para cada configuração existe na máximo um possível movimento a ser seguido pelo AP.*

O AP representado na Figura 8.3 é determinístico. O número de movimentos para cada configuração é no máximo 1, ou seja, existem configurações para as quais existem apenas um movimento possível e configurações para as quais não existem movimentos possíveis. Já o AP representado na Figura 8.4 é não determinístico. Ao tentar reconhecer a cadeia "aabb" por exemplo, tem-se inicialmente a configuração $< S, aabb, B >$ a partir da qual existem dois possíveis movimentos: $\delta(S, a, B) =< S, A >$ ou $\delta(S, \lambda, B) =< S, \lambda >$.

Ao contrário do que acontece com os AFD's, os APD's não são capazes de processar até o fim qualquer string formado sobre o alfabeto Σ. Existem cadeias em Σ^* que podem fazer um APD terminar sua execução antes de se chegar ao final da cadeia, ou por falta de movimentos poss íveis ou por ter esvaziado a pilha. Também em contraste com os AFD's, os APD's não possuem o mesmo poder de reconhecimento que os AP's não determinísticos. Existem Linguagens Livres de Contexto para as quais não existe um APD que as possam reconhecer.

Definição 8.7 *Dada a linguagem L, diz-se que que L é uma* **Linguagem Livre de Contexto Determinística** *sse existe um APD P tal que $L(P) = L$*

Embora a utilização de APD's imponha uma certa restrição nas linguagens que podem ser representadas, a utilização de AP's não determinísticos é em geral ineficiente. É o caso, por exemplo da implementação de um analisador sintático em um compilador. Embora costume-se definir um linguagem de programação através de uma gramática, seu reconhecimento é feito através de um AP. A utilização de um AP não determinístico é em geral descartada pois a possibilidade de tentar diversos movimentos diferentes a partir de uma dada configuração torna o analisador sintático ineficiente e não prático. Ao invés disso, procura-se utilizar uma gramática que, quando implementada através de um AP produza um APD.

Devemos lembrar ainda que, em muitos casos, o não determinismo não é uma qualidade intrínseca da linguagem mas sim da Gramática e do AP que se escolhe para representá-la. Por exemplo, vamos tomar a linguagem $\{a^i b^i \mid i \geq 1\}$. Podemos ter distintos AP's que reconhecem esta linguagem, como por exemplo os representados na Figura 8.3 e $< \{a, b\}, \{S, C\}, \{t\}, t, \delta, S >$ onde δ é dada na Figura 8.8. O primeiro é determinístico e o segundo não determinístico.

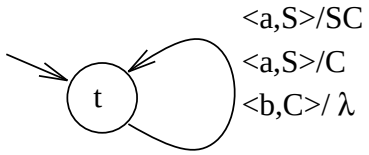


Figura 8.8: Diagrama de trasnsição para um AP não determinístico que reconhece $\{a^i b^i \mid i \geq 0\}$

8.5 Exercícios

8.1 *Dê AP's para as seguintes linguagens:*

a. $\{x \in \{a, b\}^* \mid |x|_a < |x|_b\}$

b. $\{x \in \{a, b\}^* \mid |x|_a \geq |x|_b\}$

c. $\{x \in \{a, b\}^* \mid x = x^r\}$

d. $\{a^j b^k c^m \mid j \geq 3 \wedge k = m\}$

e. $\{x \in \{a, b\}^* \mid |x|_a = 2|x|_b\}$

f. $\{x \in \{a, b\}^* \mid |x|_a \neq |x|_b\}$

g. conjunto de todas as expressões posfixas sobre $\{a, b, +, -\}$

h. $\{a^n b^n c^m d^m \mid n, m \geq 0\}$

i. $\{a^i b^j c^j d^i \mid i, j \geq 0\}$

Capítulo 9

Máquinas de Turing

Como visto anteriormente, existe uma hierarquia de linguagens para as quais podemos utilizar gramáticas como forma de representação. Até agora estudamos as Gramáticas regulares e livres de contexto. Neste capítulo estudaremos as Máquinas de Turing que são autômatos que permitem que uma classe mais ampla de linguagens sejam definidas, as linguagens dos tipos 1 e 0.

As Máquinas de Turing, além de reconhecedoras de cadeias, podem ser também utilizadas como modelo de máquina para computação de funções. Assim, uma Máquina de Turing pode ser vista como a mais simples "máquina de computação" servindo por isso modelo para determinar quais funções são computáveis e quais não são. Por isso, Máquinas de Turing podem ser também utilizadas como base para estabelecerem-se medidas de complexidade de algoritmos. Todos esses pontos de Máquinas de Turing serão abordados adiante.

9.1 Definições Básicas

Uma visualização para um Máquina de Turing é dada na Figura 9.1.

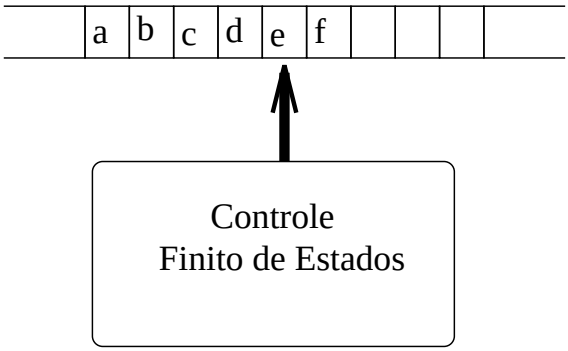


Figura 9.1: Visualização de uma Máquina de Turing

Ela possui uma fita, de tamanho ilimitado e o controle finito de estados.

Sobre a fita está uma cabeça de leitura e escrita, que pode passar o valor de uma letra que está sob ela na fita para o controle e pode trocar o conteúdo da fita, de acordo com comandos recebidos do controle de estados. Essa cabeça pode ainda ser movida, tanto para a direita quanto para a esquerda, de acordo com os comandos do controle finito de estados.

A definição de um Máquina de Turing é:

Definição 9.1 *Uma Máquina de Turing (MT) é uma quíntupla* $< \Sigma, \Gamma, S, S_0, \delta >$ *onde:*

- Σ *é o alfabeto de entrada*
- Γ *é o alfabeto auxiliar.* $\# \in \Gamma$, $\{L, R\}$, Γ *e* Σ *são disjuntos dois a dois*
- S *é o conjunto finito não vazio de estados*
- S_0 *é o estado inicial da Máquina de Turing.* $S_0 \in S$
- δ *é a função de transição de estados. delta* : $S \times (\Sigma \cup \Gamma) \rightarrow (S \cup \{h\}) \times (\Sigma \cup \Gamma \cup \{L, R\})$

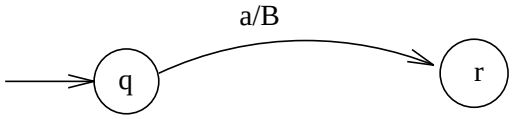
O conjunto Σ é o conhecido alfabeto de entrada. Inicialmente, a palavra colocada na fita e que se pretende analisar é composta por letras desse alfabeto. O alfabeto Γ contém um conjunto de símbolos auxiliares que podem ser escritos na fita durante o processamento da MT. Nesse alfabeto sempre está uma letra especial denotada por # chamada de "branco" ou "espaço". Dois símbolos especiais, L e R não aparecem nunca na fita, ou seja, nunca são parte de Σ nem de Γ.

O estado S_0 é o estado em que a MT começa sua operação. Inicialmente a fita contém a palavra a ser analisada com a cabeça posicionada na sua primeira letra. Todas as demais células da fita estão com a letra #. O comportamento da MT é determinado pela função δ. Ela toma como entrada o estado corrente da máquina e a letra sob a cabeça da fita. Ela determina qual deve ser o próximo estado $(S \cup \{h\})$ e uma ação a ser executada, que pode ser a escrita de uma letra na fita, no lugar da letra lida $(\Sigma \cup \Gamma)$ ou mover a cabeça para a esquerda ou direita $(\{L, R\})$. O estado h é um estado especial, chamado "halt state". Ao passar para esse estado a MT pára. Se a função δ determina que a ação a ser tomada pertence a $(\Sigma \cup \Gamma)$, por exemplo, $\delta(s, a) =< r, B >$, isso significa que o a que está na fita deve ser substituída por um B. Se a ação é um L e R, isso indica que a cabeça deve ser movida para a esquerda ou para a direita, respectivamente.

Note que a definição dada admite que para cada configuração da MT existe apenas uma possível transição, fazendo desse autômato uma máquina determinística. Mais tarde iremos tratar de MT's não determinísticas.

Uma MT pode ser representada através de um diagrama de trsnsição. As mesmas convenções de estados (círculos), transições (arcos) e determinação de

estado inicial utilizadas para AF's e AP's valem para MT's. A diferença é que cada transição é rotulada com uma letra de entrada que deve ser lida para que a transição aconteça e com a ação a ser tomada. Por exemplo, $\delta(q,a) =< r, B >$ é representada como



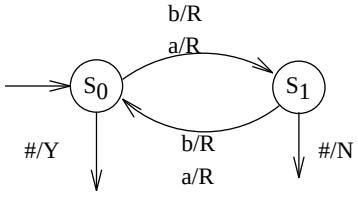
A definição da configuração de uma MT é dada por:

Definição 9.2 *Seja a MT* $M =< \Sigma, \Gamma, S, S_0, \delta >$ *cuja cabeça de leitura/escrita encontra-se sobre a letra* a *na fita que contém o string* $...\#\#\#\alpha a\beta\#\#\#...$ *tal que* $\alpha, \beta \in (\Sigma \cup \Gamma)$, $\#$ *não é prefixo de* α *e nem sufixo de* β *e cujo estado corrente é* s. *A configuração de* M *pode ser representada por* $\alpha s b \beta$

Se uma MT passa da configuração $\alpha_1 s a_1 \beta_1$ para a configuração $\alpha_2 r a_2 \beta_2$ através de uma única transição denotamos $\alpha_1 s a_1 \beta_1 \vdash \alpha_2 r a_2 \beta_2$. Se essa mudança se dá através de um número finito de transições denotamos $\alpha_1 s a_1 \beta_1 \vdash^* \alpha_2 r a_2 \beta_2$.

9.2 Exemplos de Máquinas de Turing

Vamos tomar inicialmente a linguagem $\{x \in \{a,b\}^* \mid |x| \text{ é par}\}$. Para reconhecer esta linguegem construímos a MT $M_1 = < \{a,b\}, \{\#, Y, N\}, \{S_0, S_1\}, S_0, \delta >$ onde δ é dada na Figura 9.2



$$\delta(S_0, a) =< S_1, R >$$
$$\delta(S_0, b) =< S_1, R >$$
$$\delta(S_0, \#) =< h, Y >$$
$$\delta(S_1, a) =< S_0, R >$$
$$\delta(S_1, b) =< S_0, R >$$
$$\delta(S_1, \#) =< h, N >$$

Figura 9.2: MT para $\{x \in \{a,b\}^* \mid |x| \text{ é par}\}$

Essa MT pega uma letra da entrada e sempre muda de estado, se essa letra for um a ou um b. Além disso, sempre avança a cabeça de leitura/escrita. O estado S_0 indica que um número par de letras já foi consumido e S_1 indica que um número ímpar foi consumido. Ao alcançar uma letra # (fim da cadeia), a MT vai para o estado h e coloca a letra Y na fita se o número de letras consumido é par ou põe a letra N na fita, caso esse número seja ímpar.

Esse exemplo nos mostra como pode ser determinado o critério de aceitação de uma MT. Pode-se determinar várias maneiras pelas quais uma cadeia é aceita ou rejeitada por uma MT. O critério que será adotado neste capítulo é o utilizado

pela máquina M_1. Inicialmente, para que um string x seja aceito, é preiso que a máquina atinja o estado h. Caso a MT entre num estado a partir do qual não se pode chegar a h (a MT nunca para), então o string é rejeitado. Segundo, deve-se analisar o conteúdo da fita quando a máquina para. Se, ao mover-se para o estado h, foi escrita a letra Y na fita, então a cadeia é aceita. Se a letra N foi escrita, então a cadeia é rejeitada.

O exemplo M_1 é bastante simples. Na verdade, a MT somente simula a execução de um AF, avançando sempre a cabeça de leitura/escrita que por sua vez somente lê símbolos da fita. Qualquer Linguagem Regular pode ser representada por uma MT similar. Vamos agora tomar a linguagem $\{x \in \{a,b,c\}^* \mid |x|_a = |x|_b = |x|_c\}$ que não é Regular ou Livre de Contexto, e por isso exige uma MT para se representada. Para construir tal MT vamos inicialmente estabelecer um procedimento para reconhecer essa linguagem. Esse procedimento é:

```
enquanto existe um "a"
     ache o "a" e substitua-o por um X
     volte para o inicio da cadeia
     procure um b; se nao achou termine com um "N"
     troque o b por X
     volte para o inicio da cadeia
     procure um c; se nao achou termine com um "N"
     troque o c por X
     volte para o inicio da cadeia
fim
procure b ou c; se achou termine com "N"
termine com "Y"
```

Esse procedimento pode ser realizado por $M = < \{a,b,c\}, \{X,Y,N,\#\}, \{S_0,S_1,S_2,S_3,S_4,S_5,S_6\}, S_0, \delta >$, sendo δ dada na Figura 9.3. Para melhor entender a MT vamos descrever cada um dos seus estados:

S_0 - procura por um a. Se achar, passa para S_2. Se achar um b ou um c passa para S_1. Se achar um # então todos os a's e b's foram retirados; termina com Y

S_1 - procura um a. Se chegar a um #, então faltam a's; termina com N, Se achar o a, passa para S_2

S_2, S_4, S_6 - volta ao começo da cadeia

S_3 - procura um b. Se não achar, termina com N

S_5 - procura um c. Se não achar, termina com N

9.3 Variações de MT's

O modelo de MT apresentado neste capítulo pode ser considerado um "modelo básico". Existem diversas maneiras pelas quais este modelo pode pode

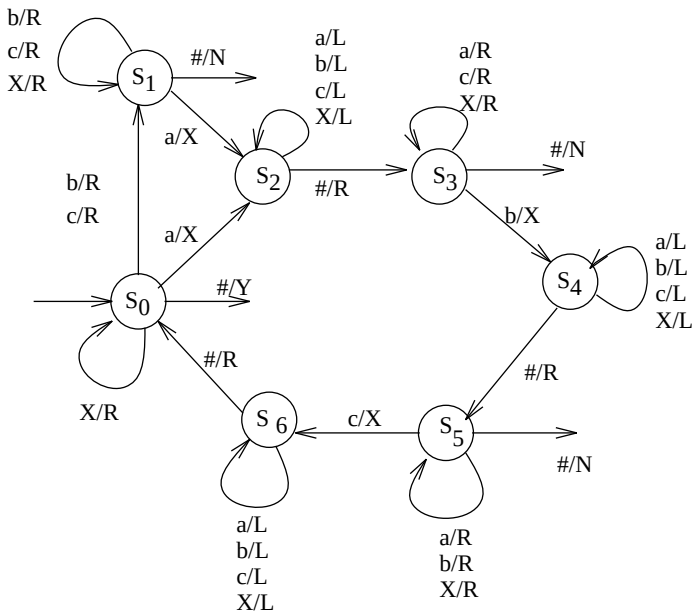


Figura 9.3: MT para $\{x \in \{a,b,c\}^* \mid |x|_a = |x|_b = |x|_c\}$

ser "aperfeiçoado". Esta seção irá mostrar algumas dessas maneiras. O ponto importante de se notar é que nenhuma das alterações propostas aqui é capaz de aumentar o poder de reconhecimento das MT's. Ou seja, cada uma dessas máquinas melhoradas possui uma MT básica capaz de realizar a mesma tarefa.

A primeira maneira de modificar uma MT seria permitir que durante uma única transição, a máquina pudesse escrever sobre a fita e mover a cabeça. Teríamos então uma transição como a mostrada na Figura 9.4a, em que numa única transição a letra b é colocada na fita e a cabeça é movida para a direita. Esse comportamento pode ser facilmente conseguido numa MT convencional através de duas transições, como a mostrada na Figura 9.4b. O símbolo b é inicialmente colocado na fita, e a máquina muda para o estado U, onde então, a transição $\delta(U,b) =< T,R >$ faz a cabeça mover-se para a direita. Portanto, essa mudança, não aumenta em nada o poder de processamento das MT.

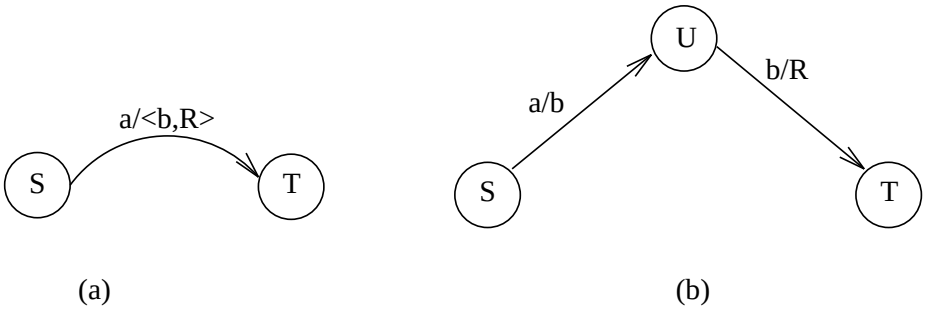


Figura 9.4: Transição que permite escrita e movimento da cabeça

Outra maneira de modificar um MT é com a adição de uma "trilha" extra na fita de entrada, como mostrada na Figura 9.5.

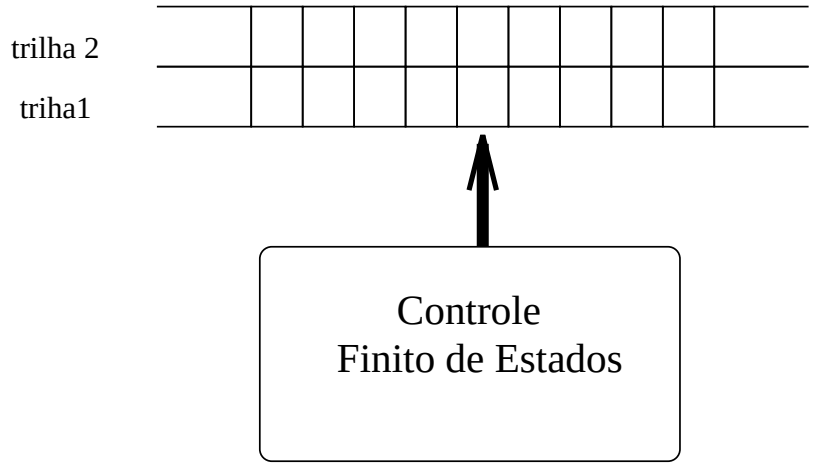


Figura 9.5: Modelo de MT com 2 trilhas

A palavra a ser reconhecida é colocada na trilha 1 e a trilha 2 inicialmente está vazia (só contém #). A trilha 1 pode ter letras de Σ e Γ e a trilha 2 possui seu próprio alfabeto, não necessariamente distinto de Σ e Γ. Cada transição é determinada pelo conteúdo das duas trilhas e deve determinar o movimento da cabeça ou como cada trilha deve ser modificada. Vamos tomar como exemplo a MT definida anteriormente para reconhecer a linguagem $\{x \in \{a,b,c\}^* \mid |x|_a = |x|_b = |x|_c\}$ (Figura 9.3). Em vez de substituirmos a letra encontrada por um X, iremos colocar uma marca na segunda trilha, indicando que aquela letra já foi usada. Ao final teremos o string original analisado e à sua direita uma letra Y indicando aceitação ou N indicando a não aceitação. Com isso, não destruímos o string original. O alfabeto da segunda trilha é portanto $\{\#, X\}$, onde X é a marca que colocaremos na trilha. O alfabeto Γ fica sendo $\{\#, Y, N\}$. As transições dessa nova méquina estão mostradas na Figura 9.6.

Não é difícil verificar que o funcionamente de qualquer máquina com duas trilhas pode ser conseguido com uma MT comum. Para isso, basta que o conjunto de símbolos da fita seja combinado com os símbolos da segunda trilha, para formar o alfabeto Γ. No exemplo da Figura 9.6, teríamos uma máquina equivalente definindo a MT $< \{a,b,c\}, \{\#, Y, N, aX, bX, cX\}, \{S_0, S_1, S_2, S_3, S_4, S_5, S_6\}, S_0, \delta >$. A função δ é definida para cada elemento de Σ e de Γ, baseado na função da MT de 2 trilhas. No exemplo anterior, obteríamos as transições mostradas na Figura 9.7. O mesmo vale também para MT com n trilhas. Todas elas são na verdade MT simples com o alfabeto Γ expandido, para representarem n-uplas ordenadas onde cada elemento corresponde a uma letra de uma das trilhas.

Embora não adicionem poder a um MT, múltiplas trilhas podem ser utilizadas para facilitar algumas demostrações ou vizualizações sobre MT's. Vamos utilizar, por exemplo, uma MT com 3 trilhas para mostrar que um MT com duas cabeças de leitura/escrita também não adiciona poder de reconhecimento ao modelo básico. Ou seja, mostrando que uma MT com duas cabeças pode ser

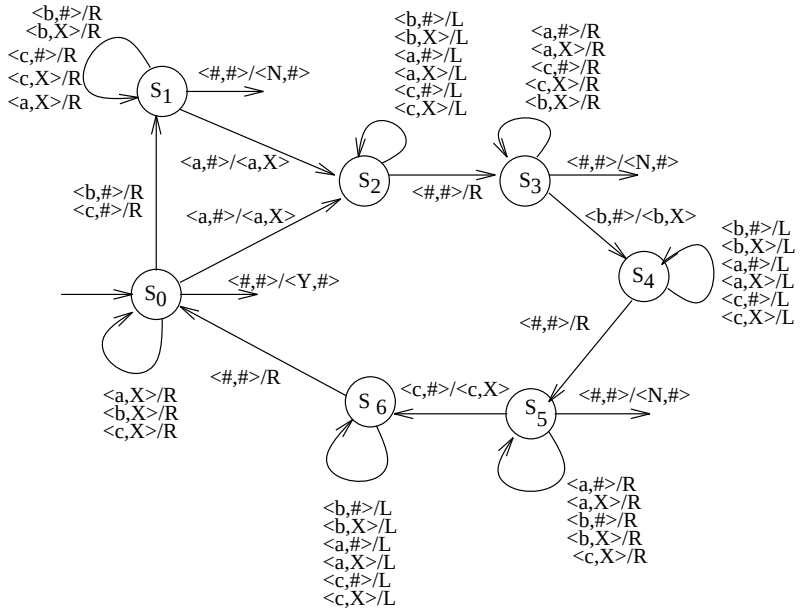


Figura 9.6: MT com 2 trilhas para $\{x \in \{a,b\}^* \mid |x|_a = |x|_b = |x|_c\}$

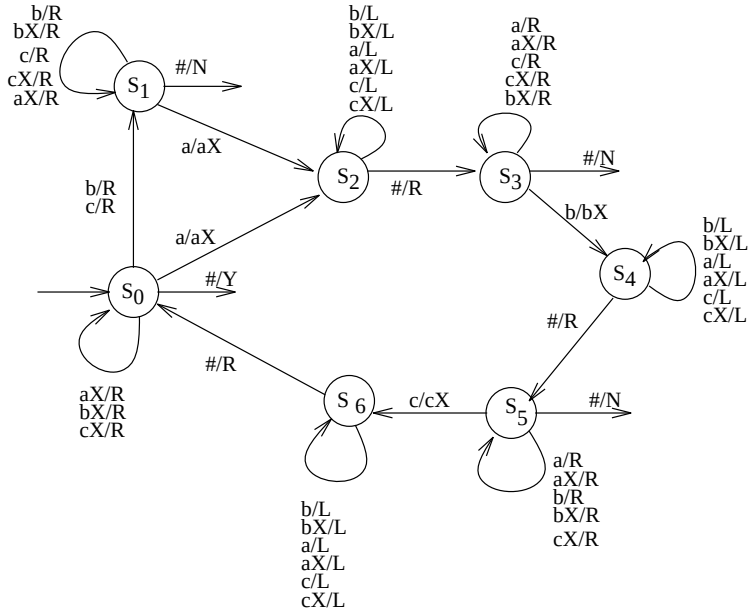


Figura 9.7: MT básica que simula MT com 2 trilhas para $\{x \in \{a,b\}^* \mid |x|_a = |x|_b = |x|_c\}$

simulada por uma outra com 3 trilhas, mostramos a equivalência daquele modelo com o modelo básico de MT. Uma MT com 2 cabeças é uma modelo que possui 2 cabeças de leitura/escrita que funcionam independentemente. A transição a ser realizada num determinado instante é determinada pelos dois símbolos, embaixo

de cada uma das cabeças e essa transição determina uma ação indepentende para cabeça. Por exemlo, para reconhecer a linguagem $\{a^n b^n \mid n \geq 0\}$, poderíamos definir a MT da Figura 9.8. Ambas as cabeças iniciam no começo da cadeia a ser analisada. Uma delas é movida até o primeiro b. A partir daí ambas as cabeças são movidas juntas para a direita até que a da direita encontre um # ou até que a da esquerda encontre um b. Se ambas as coisas acontecem simultaneamente, a cadeia é aceita, caso contrário, ela é rejeitada.

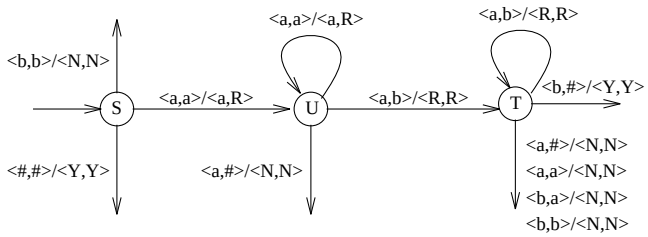


Figura 9.8: MT com 2 cabeças para $\{a^n b^n \mid n \geq 0\}$

Uma MT com 2 cabeças pode ser simulada por uma MT com 3 trilhas. A cadeia a ser analisada é colocada na primeira trilha. Na segunda e na terceira trilhas são colocadas marcas apenas para indicar as posições em que se encontram as 2 cabeças que estão sendo simuladas. Diversos movimentos da cabeça de leitura/escrita são necessários para simular uma transição da MT de 2 cabeças. Cada transição simulada começa com o movimento da cabeça até a marca da trilha 2. O conteúdo da trilha 1 é analisado e o Controle Finito de Estados deve recordar qual foi a letra lida nessa posição. Move-se então a cabeça até a segunda marca (na trilha 3) e novamente o conteúdo da trilha 1 é analisado. Nesse ponto, a máquina possui informação sobre os dois símbolos que estariam sob as duas cabeças da máquina simulada e deve atualizar o conteúdo da fita e a posição das cabeças simuladas. Então a cabeça volta novamente à primeira marca e atualiza a sua posição ou o conteúdo da trilha 1, e o mesmo é feito para a segunda marca. Isso completa a simulação de uma transição. Diversos casos especiais devem ser tratados com cuidado, como por exemplo, quando ambas as cabeças estão sobre a mesma posição.

Procedimento similar pode ser adotado para qualquer MT com n cabeças. Pode-se simular seu funcionamento através de um MT com $n + 1$ trilhas (e portanto com uma MT básica). Assim, uma MT com múltiplas cabeças não possui nenhum poder de reconhecimento extra sobre as MT normais.

Podemos ainda tentar melhorar um MT fornecendo-lhe diversas fitas independentes, cada uma com sua própria cabeça de leitura/escrita, como na Figura 9.9a. Podemos imaginar essa MT como uma máquina multi-trilha, com diversas cabeças de leitura, como mostrado na Figura 9.9b. Nessa MT multi-trilha, cada cabeça de leitura só responde aos símbolos que aparecem na trilha que corresponde à fita à qual a cabeça estava ligada na MT original. Podemos "juntar" todas as trilhas numa só, alterando o alfabeto auxiliar da MT, obtendo um MT com uma única trilha e diversas cabeças. Assim, a máquina de n fitas é também similar em poder à MT básica que definimos no início deste capítulo.

Finalmente, podemos alterar a nossa definição básica de MT's permitindo

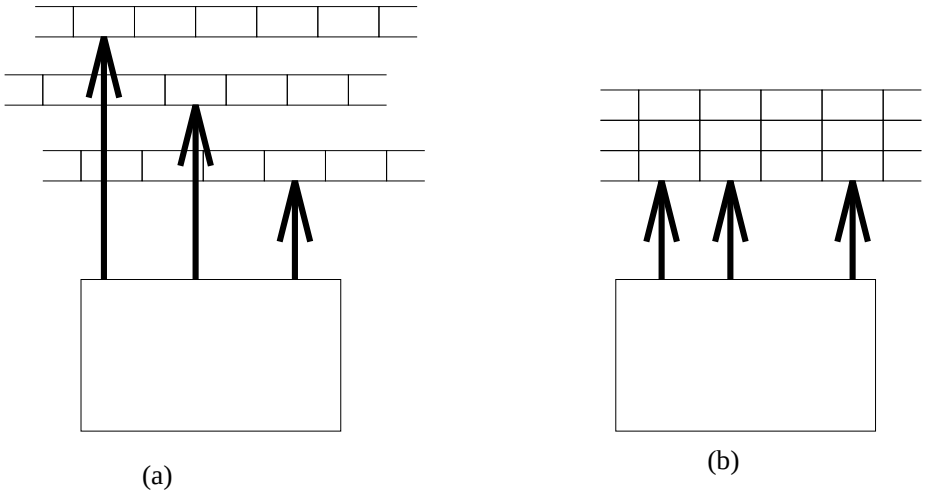


Figura 9.9: MT com 3 fitas vista como uma MT com 3 trilhas

que haja não determinismo nas transições da máquina. Isso significa que para uma dada configuração podem existir diversas possíveis transições e que se uma delas levar a uma condição de aceitação, então a cadeia faz parte da linguagem definida pela MT. Para simular uma MT não determinística através de uma MT determinística, deve-se utilizar uma MT com 3 fitas. Na primeira fita mantém-se uma cópia inalterada da cadeia de entrada. Na segunda fita copia-se essa cadeia, cada vez que uma nova tentativa de reconhecer a cadeia é feita. A terceira fita é utilizada para recordar qual é a seqeüência de transições a serem empregadas (mais detalhes de tal máquina pode ser encontrada em [CAR89]). De qualquer modo, o importante é notar-se que uma MT não determinística pode sempre ser transformada numa MT determinística e portanto, assim, como as alterações propostas acima, a introdução de não determinismo não adiciona nenhum poder às MT's.

9.4 Enumeração de Linguagens

Até agora vimos as MT's como reconhecedoras de cadeias de uma certa linguagem. Dada a cadeia, a MT responde afirmativa ou negativamente, indicando se esta foi aceita ou não. Pode-se utilizar também uma MT para enumerar os componentes de uma linguagem. Tal máquina produz (escreve na fita) uma seqüência com cada um dos elementos da linguagem. Ela não recebe nenhuma entrada (inicialmente a fita está vazia) e pára quando todos os strings tiverem sido escritos na fita.

Em geral, para tais máquinas utilizamos MT com k fitas. A primeira fita é a fita de saída, onde são escritas as cadeias da linguagem e as demais são utilizadas para controlar a geração. Vamos tomar como exemplo a linguagem $L = \{a^i b^i c^i \mid i \geq 0\}$. Definimos, para enumerar essa linguagem, uma MT com duas fitas que utiliza o algoritmo dado abaixo. Ela vai escrevendo cada elemento de L na fita de saída, separados pelo símbolo especial $*$. Os passos seguidos são:

a. escreve-se $**$ na fita 1, indicando que λ pertence á linguagem

b. um a é colocado na fita 2

c. a cabeça da fita 1 move-se para a direita, escrevendo um a para cada a existente na fita 2

d. a cabeça da fita 1 move-se para a direita, escrevendo um b para cada a existente na fita 2

e. a cabeça da fita 1 move-se para a direita, escrevendo um c para cada a existente na fita 2

f. adiciona-se um a na fita 2

g. coloca-se um $*$ na fita 1

h. volta-se ao passo c

Fica como exercício para o leitor criar a MT que implementa o algoritmo acima.

9.5 Computação de Funções

Quando uma MT é utilizada como reconhecedora de cadeias, existem somente dois resultados possíveis para sua execução: aceitação ou rejeição. Podemos porém ampliar essa interpretação e considerar o conteúdo final da fita como sendo o resultado da computação de uma MT. Temos assim, um número infinito de possíveis resultados para uma MT. Vamos utilizar essa característica para definir MT's como instrumentos para computar funções.

Uma função (possivelmente parcial) $f : X \rightarrow Y$ é um mapeamento que assinala para cada elemento do conjunto X, no máximo um elemento de Y. Sob o ponto de vista computacional, consideramos os valores do domínio X como sendo a entrada da função. O valor associado à entrada x, denotado $f(x)$, é a saída ou resultado da função. A partir daqui utilizaremos uma MT para computar o valor de $f(x)$, onde $x, f(x) \in \Sigma^*$, ou seja, $f : \Sigma^* \rightarrow \Sigma^*$ é uma função cujo domínio e contra-domínio são os strings do alfabeto de entrada. Em tal MT a cadeia x é colocada sobre a fita e a MT deve levar ao valor correspondente $f(x)$, ou seja, ao terminar seu processamento, deve-se ter sobre a fita a cadeia $f(x)$.

Como exemplo, vamos utilizar a função $g : \{a, b\}^* \rightarrow \{a, b\}^*$ dada por

$$g(x) = \begin{cases} \lambda \text{ se x contém algum } a \\ x \text{ caso contrário} \end{cases}$$

A MT $< \{a, b\}, \{\#\}, \{S, T, U, V\}, S, \delta >$ cujo diagrama de transições é dado na Figura 9.10 calcula essa função. Note que nem sempre o domínio e o contra-domínio da função precisam coincidir. No caso de g poderíamos ter definido $g : \{a, b\}^* \rightarrow \{b\}^*$ uma vez que o resultado definido pela função é sempre a cadeia vazia ou algum string que só contém b's.

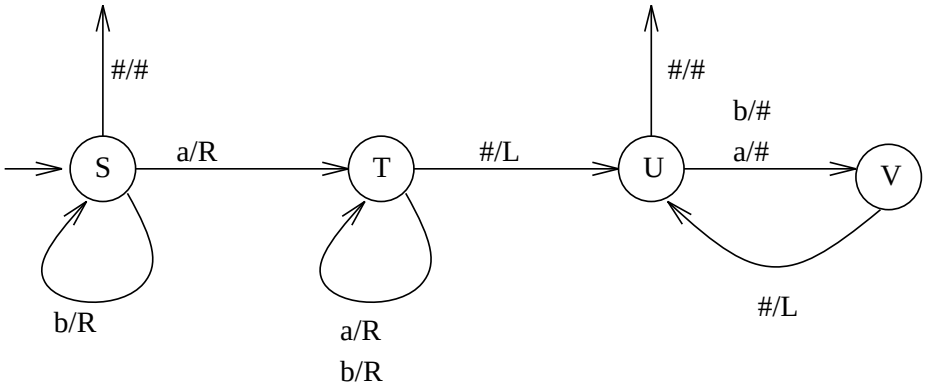


Figura 9.10: MT para computar $g(x)$

Podemos também definir funções parciais, ou seja, funções que possuem valores do domínio para qual seu valor não é definido. Para essas funções iremos projetar MT's que não param, ao receber como entrada um valor cuja saída não é definida. Por exemplo, a função

$$h(x) = \begin{cases} \lambda \text{ se x contém algum } a \\ \text{indefinido caso contrário} \end{cases}$$

Para tal função podemos definir a MT $< \{a,b\}, \{\#\}, \{S,T,U,V\}, S, \delta >$ cuja função δ é dada na Figura 9.11.

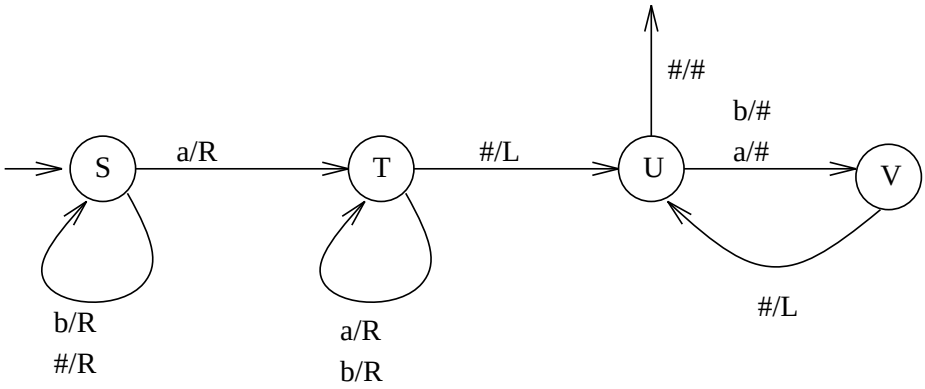


Figura 9.11: MT para computar $h(x)$

Funções com mais de um argumento podem também ser computadas. As entradas são colocadas na fita, na mesma ordem definida na função e separados por um #. Por exemplo, dada a função $f : \{a,b\}^* \times \{a,b\}^* \times \{a,b\}^* \rightarrow \{a,b\}^*$, poderíamos ter as seguintes configurações iniciais:

Vamos tomar como exemplo u função binária *conc* (concatenação) definida $conc : \{a,b\}^* \times \{a,b\}^* \rightarrow \{a,b\}^*$ e $conc(x,y) = xy$. Construímos a MT $< \{a,b\}, \{\#\}, \{S,T,U,V,W\}, S, \delta >$ e cujas transições são descritas na Figura 9.13.

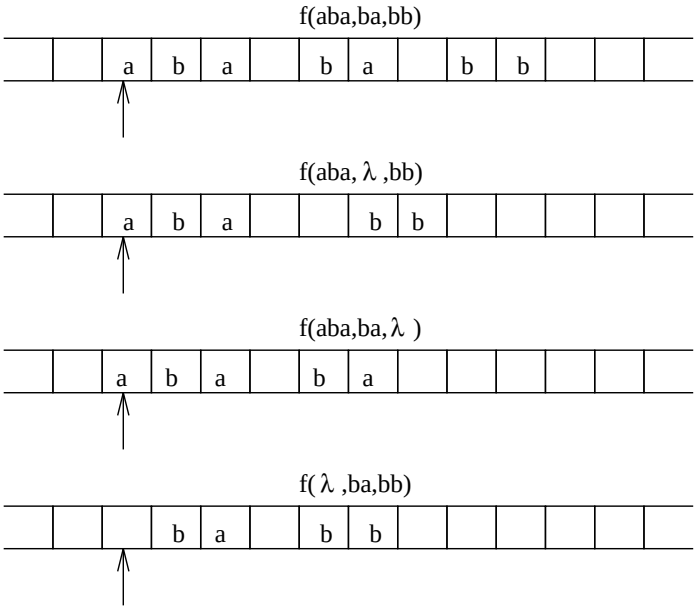


Figura 9.12: Configuração inicial para cálculo de função com 3 argumentos

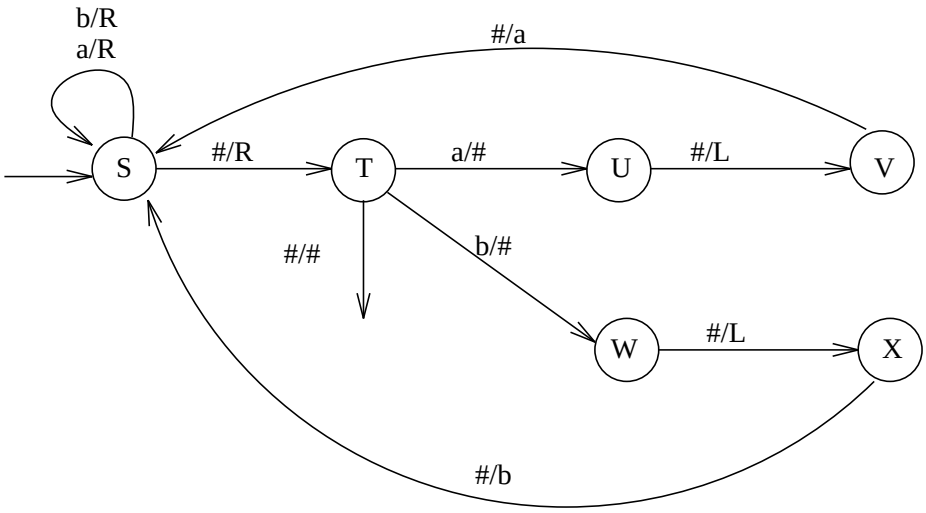


Figura 9.13: MT para cálculo de $conc(x, y)$

Uma classe particularmente interessante de funções que podem ser calculadas utilizando-se MT's são as funções **número-teóricas**, definidas $f : N \times N \times ... \times N \rightarrow N$. A entrada de tais funções é uma n-upla de números naturais e a saída é um número natural. Como exemplo de tais funções podemos citar a função unária $sqr(x) = x^2$ ou a função binária de soma $x + y$. Note-se que agora o domínio e a imagem da da função que queremos calcular não são mais cadeias mas sim, números. Essa mudança de perspectiva não representa um problema. Iremos trabalhar com cadeias de strings que representam os números

naturais e então sobre elas definir as funções desejadas. Esse processo costuma ser chamado de **codificação** pois estamos criando para cada número natural um código, que o representa.

Existem diversas maneiras de codificar um número natural. Por exemplo, podemos utilizar a representação binária, onde os naturais são codificados como cadeias sobre o alfabeto $\{0,1\}$. Nesse caso é fácil construir uma MT para calcular a função $succ(x) = x + 1$ (ver exercícios). Outras formas de codificação poderiam ser utilizadas, como a representação decimal ou octal dos números. Uma representação particularmente interessante é a **representação unária**. Nessa forma, um número natural e codificado sobre o alfabeto $\{1\}$. O número natural n é representado repetindo-se $n + 1$ vezes a letra 1, ou seja, n é representado pela cadeia 1^{n+1}. Por exemplo,

Número	Código unário
0	1
1	11
2	111
10	11111111111
32	1^{32}

Dessa forma torna-se ainda mais fácil definirem-se, por exemplo, as funções $succ(x)$ ou $x + y$ (ver exercícios).

No próximo capítulo estudaremos mais sobre as funções computadas através de MT's. Por ora vamos tentar responder à pergunta: quais funções podem ser computadas através de MT's? A resposta é dada pela **Tese de Church-Turing** que diz:

> Acredita-se que não existem funções definidas por humanos, cujo cálculo possa ser descrito por quealquer algoritmo bem definido e que as pessoas possam aprender, e que não possam ser calculadas por MT's [?]

A Tese de Church apresentada por Alonzo Church em 1936 era na verdade ligeiramente diferente pois foi proposta um pouco antes de serem criadas as MT's. Ela dizia que uma máquina que pudesse realizar um conjunto restrito de operações seria capaz de realizar qualquer algoritmo imaginável. As MT's são capazes de realizar cada uma dessas operações, sendo portanto o tipo de máquina requerido por Church.

Note-se que a tese de Church é assim chamada pois não se pode provar sua validade (caso em que seria chamada Teorema, em vez de Tese). Seu autor forneceu diversas razões sofisticadas para que se acredite nela, porém idéias como "algoritmo que pessoas possam aprender a realizar" não fazem parte do arcabouço matemático conhecido. Como não existe nenhuma definição precisa do que seja um algoritmo, se acreditarmos que a Tese de Church-Turing é verdadeira, podemos então definir um algoritmo como aquilo que uma MT pode executar. Dessa forma definimos de maneira precisa quias são os passos, ou as instruções que podem ser utilizados num algoritmo.

É possível (embora improvável) que algum dia alguém ache uma tarefa que se concorde seja um algoritmo e que não possa ser realizada por uma MT. Nesse caso deveríamos apresentar um outro modelo de máquina capaz de realizar tal tarefa. Por enquanto, todos parecem felizes com a idéia de que uma MT representa o máximo em máquina de computação.

Sudkamp [SUD97] apresenta uma abordagem interessante ao mostrar como construir uma linguagem de alto nível cujos programas possam ser executados numa MT. Ele utiliza o arqgumento intuitivo de que um programa escrito numa linguagem de programação e executado num computador pode ser simulado numa MT pois o que uma instrução de computador faz é alterar bits em certas posições de memória, e é esse exatamente o tipo de ação realizada por uma MT, escrevendo 0's e 1's na sua fita. O autor apresenta uma arquitetura de MT para computação de alto nível e um conjunto de instruções que formam uma linguagem para essa arquitetura e que podem ser utilizadas para criarem-se linguagens de alto nível.

9.6 Exercícios

9.1 *Construa MT's para as seguintes linguagens:*

a. $\{x \in \{a,b,c\}* \mid |x|_a = |x|_b = |x|_c)\}$

b. $\{x \in \{a,b,c\}^* \mid x = x^r\}$

c. $\{a^n \mid n$ *é quadrado perfeito*$\}$

d. $\{a^k b^n c^m \mid (k \neq n) \wedge (n \neq m)\}$

e. $\{x \in \{a,b,c\}^* \mid (|x|_a \neq |x|_b) \wedge (|x|_b \neq |x|_c)\}$

f. $\{xx \mid x \in \{a,b,c\}^*\}$

g. $\{a^n \mid n$ *é potência de 2*$\}$

h. $\{a^i b^j c^k \mid i + j = k\}$

i. $\{x \in \{a,b,c\}^* \mid |x|_a + |x|_b = |x|_c\}$

j. $\{a^n b^n c^m d^m \mid n, m \geq 0\}$

k. $\{a^n b^m c^m d^n \mid , n, m \geq 0\}$

9.2 *Construa MT's que enumerem as seguintes linguagens:*

a. $\{a^i b^i c^i \mid i \geq 0\}$

b. $\{a^n \mid n$ *é quadrado perfeito*$\}$

c. $\{x \in \{a,b\}^* \mid |x|_a = |x|_b\}$

9.3 *Caso as MT's definidas no exercício anterior tenham mais do que uma trilha, transforme-as em MT convencionais.*

9.4 *Construa MT's sobre* $\{a, b\}$ *para realizar as seguintes operações:*

a. Mover a entrada um espaço à direita

b. Concatenar uma cópia do reverso da cadeia, na cadeia original

c. Inserir um # entre cada letra da entrada

d. Excluir todos os b's da cadeia de entrada. Por exemplo, a cadeia abbaaba deve virar aaaa

9.5 *Construa MT's sobre* $\{a, b\}$ *que compute*

a. $f(x) = aaa$

b. $f(x) = \begin{cases} a \text{ se } |x| \text{ é par} \\ b \text{ caso contrário} \end{cases}$

c. $f(x) = x^r$

d. $f(x, y) = \begin{cases} x \text{ se } |x| > |y| \text{ é par} \\ y \text{ caso contrário} \end{cases}$

9.6 *Construa MT's para calcular as seguintes funções numero-teóricas:*

a. $succ(x) = x + 1$

b. $f(n) = 2n + 3$

c. $half(n) = \lfloor \frac{n}{2} \rfloor$

d. $f(a, b, c) = a + b + c$

e. $even(n) = \begin{cases} 1 \text{ se } n \text{ é ímpar} \\ 0 \text{ caso contrário} \end{cases}$

f. $eq(n, m) = \begin{cases} 1 \text{ se } n = m \\ 0 \text{ caso contrário} \end{cases}$

g. $lt(n, m) = \begin{cases} 1 \text{ se } n < m \\ 0 \text{ caso contrário} \end{cases}$

h. $n \dot{-} m = \begin{cases} n - m \text{ se } n \geq m \\ 0 \text{ caso contrário} \end{cases}$

Capítulo 10

Decidibilidade

Neste capítulo utilizaremos as Máquinas de Turing para estudar os Problemas de Decisão, que constituem uma série de questões cujas respostas podem ser afirmativas ou negativas. Uma solução para um Problema de Decisão é um procedimento que deve determinar a resposta para cada uma das perguntas. A tese de Church-Turing nos diz que pode-se então projetar uma MT para resolver um Problema de Decisão, ou, se mostrarmos que não não existe uma MT que solucione este problema, estaremos mostrando que ele não possui solução.

10.1 Problemas de Decisão

Um **Problema de Decisão** é um conjunto de questões, cada uma com uma resposta afirmativa ou negativa. Por exemplo, a questão "8 é um quadrado perfeito ?" é o tipo de questão sob consideração nos Problemas de Decisão. Em geral, um Problema de Decisão é composto por um número infinito de questões relacionadas. Por exemplo, o problema P_{SQ}, que consiste em determinar se um número natural é um quadrado perfeito é um Problema de Decisão composto pelas perguntas:

P_0 - 0 é um quadrado perfeito?

P_1 - 1 é um quadrado perfeito?

P_2 - 2 é um quadrado perfeito?

...

Uma solução para esse problema é um algoritmo (uma MT) que determine as respostas corretas para cada uma dessas perguntas. A suloção para P_{SQ} seria uma uma MT que aceitasse todas as cadeias da forma $\{a^k \mid \text{k é um quadrado perfeito}\}$ e rejeitasse as demais.

Definição 10.1 *Um Problema de Decisão é solucionável sse existe uma MT cuja execução termina para qualquer entrada e que produz as respostas corretas. Um Problema de Decisão é parcialmente solucionável sse existe uma MT que aceita precisamente os elementos cuja resposta é afirmativa*

Se existe uma MT que soluciona um Problema de Decisão dizemos que tal problema é **decidível**. Iremos ver nas próximas seções alguns problemas indcidíveis famosos, e como mostrar sua indecidibilidade. Para fazer isso vamos restringir nossa análise a MT's definidas sobre o alfabeto $\{0,1\}$ e com alfabeto auxiliar $\{\#\}$. A restrição dos alfabetos não impõe nenhuma restrição na capacidade das MT's. Qualquer MT $M = < \Sigma, \Gamma, S, S_0, \delta >$ pode ser simulada por uma outra MT $< \{0,1\}, \{\#\}, S, S_0, \delta >$ através da codificação dos símbolos de M com strings sobre o alfabeto $\{0,1\}$. Essa é exatamente a abordagem utilizada nos computadores digitais que utilizam strings de 0's e 1's para representar números, letras, etc.

Vamos utilizar como exemplo a MT da Figura **??** que reconhece a linguagem $L = \{x \in \{a,b\}^* \mid |x| \text{é par}\}$. Para codificar essa máquina podemos utilizar o seguinte código:

Símbolo	Código
a	0
b	1
Y	1
N	0

e definir a MT $< \{0,1\}, \{\#\}, \{S,T\}, S, \delta >$ cuja função δ é dada na Figura 10.1a. Uma entrada como *abbab* seria representada na fita por 01101 e a saída seria 011010, indicando que a cadeia não foi aceita.

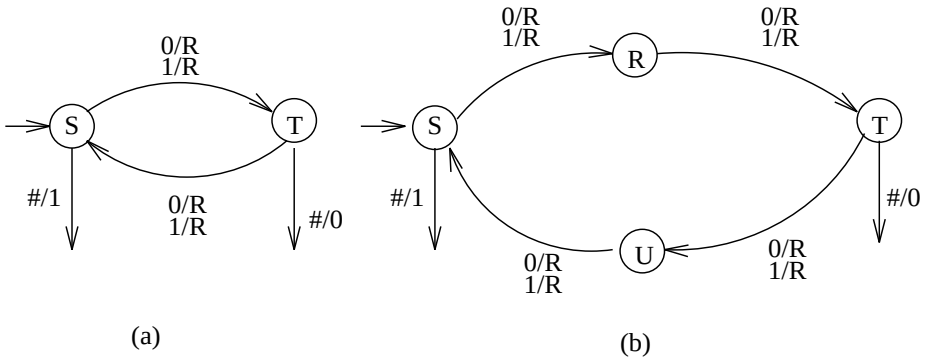


Figura 10.1: Máquinas de Turing sobre o alfabeto $\{0,1\}$

Se tivessemos, na máquina original, o alfabeto de entrada $\{a,b,c\}$ teríamos que utilizar uma outra codificação como

Símbolo	Código
a	00
b	01
c	10
Y	1
N	0

e construir uma máquina como $< \{0,1\}, \{\#\}, \{S,T,U,V\}, S, \delta >$ cuja função δ é dada na Figura 10.1b. A entrada *caababc* apareceria na fita como 10000001000110 e a saída correspondente seria 100000010001100. Os exercícios sugerem outras máquinas a serem codificadas.

10.2 Problema da Parada de Máquinas de Turing

O mais famoso problema indecidível diz respeito ao comportamento das próprias Máquinas de Turing. O **Problema da Parada de Máquinas de Turing** (P_{MT}) pode ser definido como segue:

> Dada uma Máquina de Turing $M = < \Sigma, \Gamma, S, S_0, \delta >$ e um string $x \in \Sigma^*$, a execução de M com a entrada x pára?

É importante entender que para solucionar tal problema devemos apresentar uma MT PMT que aceite como entrada M e x e que sempre apresente a resposta correta, ou seja, se M pára quando executada com x, então PMT deve terminar sua execução com uma resposta afirmativa e se M nunca pára quando executada com x, então PMT deve terminar sua execução com uma resposta negativa.

A primeira idéia para tentar solucionar esse problema seria apresentar um algoritmo do tipo "execute M com x e verifique se sua execução termina ou não". Essa idéia não é boa o suficiente pois para os casos em que M não pára não existe meio de decidir se a execução realmente não termina nunca ou se simplesmente trata-se de uma execução demorada e que eventualmente terminará.

Para executar PMT devemos ter na entrada a descrição de M e o string x. Como mostrado anteriormente, podemos nos restringir a máquinas sobre o alfabeto de entrada $\{0,1\}$. Sem perda de generalidade, iremos chamar os estados de M de $q_0, q_1, ..., q_n$, com q_0 sendo o estado inicial. Iremos representar a máquina M também com uma codificação sobre o alfabeto $\{0,1\}$, utilizando por exemplo:

§ímbolo	Código
0	1
1	11
#	111
L	1111
R	11111
q_0	1
q_1	11
...	...
q_n	1^{n+1}
h	1^{n+2}

Colocaremos na fita, como entrada de PMT a função δ de M, que define completamente M. Para representar δ iremos colocar cada transição de M separada por dois 0's consecutivos. A transição $\delta(q_i, a) =< q_j, b >$ aparece na fita como

$$cod(q_i)0cod((a)0cod(q_j)0cod(b)$$

onde $cod(k)$ representa a codificação do símbolo k, de acordo com a tabela acima. O início e o final da representação de M são delimitados por uma seqüência 000.

Vamos tomar como exemplo a MT da Figura 10.1a. Teríamos a seguinte codificação para as transições:

Transição	Código
$\delta(q_0, 0) =< q_1, R >$	101011011111
$\delta(q_0, 1) =< q_1, R >$	1011011011111
$\delta(q_1, 0) =< q_0, R >$	110101011111
$\delta(q_1, 1) =< q_0, R >$	1101101011111
$\delta(q_0, \#) =< h, R >$	101110111011
$\delta(q_1, \#) =< h, R >$	110111011101

A codificação $R(M)$ da máquina M aparece na fita como

$$000\overbrace{10101101111}^{\delta(q_0,0)}00\underbrace{1011011011111}_{\delta(q_0,1)}00\overbrace{1101010111111}^{\delta(q_1,0)}$$

$$00\overbrace{1101101011111}^{\delta(q_1,1)}00\underbrace{1011011101101}_{\delta(q_0,\#)}00\overbrace{1101110111101}^{\delta(q_1,\#)}000$$

Utilizando esse tipo de codificação, ou seja, codificando M sobre o mesmo alfabeto de entrada de M poderemos demostrar o seguinte teorema:

Teorema 10.1 *O problema da parada de Máquinas de Turing (P_{MT}) é indecidível*

Prova: A demostração desse teorema é feita por contradição. Inicialmente vamos assumir que existe uma MT PMT que soluciona o problema P_{MT}, de modo que um string é aceito por PMT se

- a entrada consiste de um acadeia $R(M)x$ onde $R(M)$ é a codificação de uma MT sobre $\{0,1\}$ e x uma cadeia sobre esse mesmo alfabeto; e
- a computação de M com x termina.

Se alguma dessas condições não é satisfeita, então PMT rejeita a entrada, como mostra a Figura 10.2. Para a primeira condição podemos construir um MT que verifique se a entrada é composta pela codificação de um MT seguida por uma cadeia x (ver exercícios). Vamos então nos deter na segunda parte ou seja, verificar se a execução de M pára ou não com x.

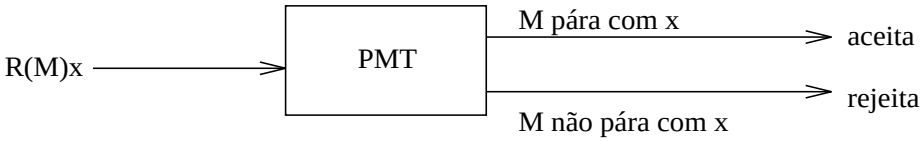


Figura 10.2: MT PMT que decide sobre a parada de M

Vamos agora modificar PMT e construir uma máquina PMT' que não pára nos casos casos em que PMT aceita sua entrada. Isso é facilmente feito trocando as transições do tipo $\delta(s,a) =< h, Y >$ por uma transição que leve a máquina a um estado em que a cabeça é indefinidamente movida para um dos lados. Teríamos então o comportamento para PMT' exibido na Figura 10.3

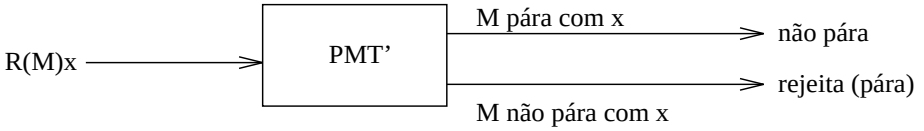


Figura 10.3: Máquina PMT', uma modificação de PMT

Vamos agora combinar PMT' com uma MT C que simplesmente duplica a sua entrada, ou seja, C computa $f(x) = xx$ para $x \in \{0,1\}^*$. Obtemos assim a máquina D mostrada na Figura 10.4. Analisando essa MT vemos que

- D não pára se a execução de M com $R(M)$ (sua própri descrição) termina
- D pára se a execução de M com $R(M)$ não termina

Consideremos agora o que acontece quando executamos D com a entrada $R(D)$, ou seja, quando fornecemos a própria descrição de D como entrada. Teremos a situação mostrada na Figura 10.5 que nos diz que D ao ser executada com entrada $R(D)$:

- não pára se a execução de D com $R(D)$ termina
- pára se a execução de D com $R(D)$ não termina

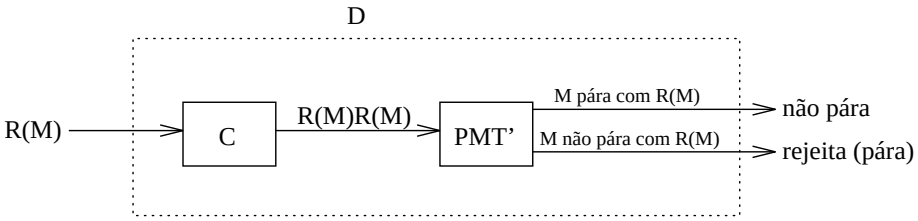


Figura 10.4: MT D que utiliza PMT'

o que obviamente é uma contradição. A máquina D foi construída a partir de PMT que soluciona P_{MT}, o que indica que erramos ao assumir que PMT pode ser construída e que portanto P_{MT} é indedcidível.

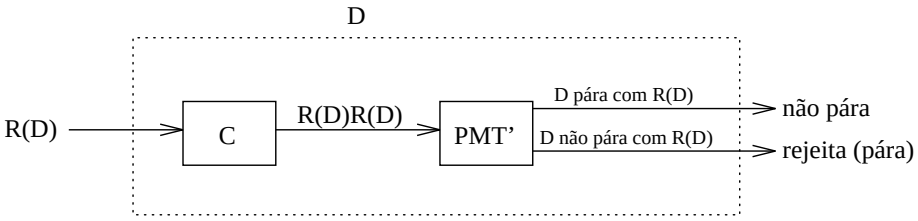


Figura 10.5: Comportamento de D ao ser executado com $R(D)$

10.3 Problema da Fita Vazia

Esse problema é um caso particular de P_{MT}. Nesse caso queremos saber se um MT M pára ou não quando a sua execução é feita com a fita de entrada inicialmente vazia. Antes de analisarmos esse problema vamos ver uma técnica chamada de **redução**.

Definição 10.2 *Um Problema de Decisão P é redutível a outro Problema de Decisão P' se existe um algoritmo que tome cada $p_i \in P$ como entrada e produza como saída uma questão $p_i' \in P'$. O mapeamento de P para P' não precisa ser 1 para 1; múltiplas questões em P podem ser mapeadas para uma única questão em P'*

Se um Problema de Decisão P' é decidível e P é redutível a P', então é fácil ver que P' também é decidível. Dada uma questão $p_i \in P$, basta reduzi-la a $p_i' \in P'$ e então resolver p_i'. O mesmo vale para problemas indecidíveis, ou seja, se um Problema de Decisão é redutível a P' e P é indecidível, então P' também é indecidível. Se P' fosse decidível, então P também poderia ser solucionado.

Voltando ao problema P_{FV}da fita vazia, vamos inicialmente assumir que existe uma máquina FV que soluciona esse problema. Tal máquina recebe como entrada a descrição de uma MT e deve aceitar aquelas cadeias que representam

máquinas que param quando executadas com λ e rejeitar as que não param. Esse comportamento é representado na Figura 10.6.

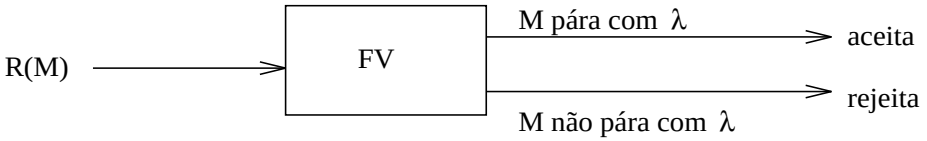


Figura 10.6: MT FV que soluciona P_{FV}

Vamos mostrar que o problema P_{MT} pode ser reduzido ao problema P_{FV} através de uma MT N que recebe como entrada a cadeia $R(M)x$ e produz como saída a descrição $R(M')$ onde M' é uma MT que:

- inicia sua execução com a fita vazia
- escreve x na fita
- retirna a cabeçara o início de x
- executa as mesmas operações de M

$R(M')$ é obtida adicionando-se algumas transições à descrição $R(M)$ e mudando o estado inicial, de modo que essas transições, que irão realizar os passos iniciais decritos acima, sejam executadas antes das transições de M. Temos então que M', por construção pára quando executado com λ sse M pára com x.

Utilizando N e e FV podemos construir a MT da Figura 10.7. Ela soluciona P_{MT} reduzindo inicialmente esse problema a P_{FV} e depois utilizando a MT FV. Isso mostra que FV não pode existir e portanto P_{FV} é indecidível.

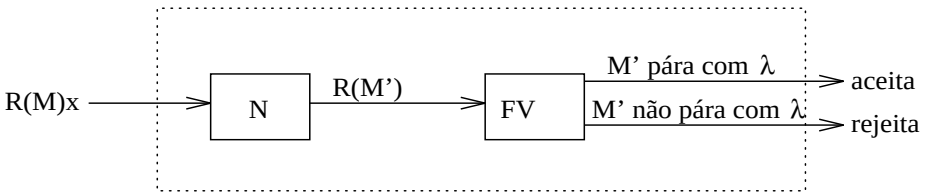


Figura 10.7: Redução de P_{MT} a P_{FV}

10.4 Problema da Parada para Σ^*

Outro problema para o qual podemos utilizar a técnica de redução é P_{Σ^*} descrito como:

> Dada a MT M, M pára com qualquer cadeia?

Queremos construir uma MT A como a da Figura 10.8.

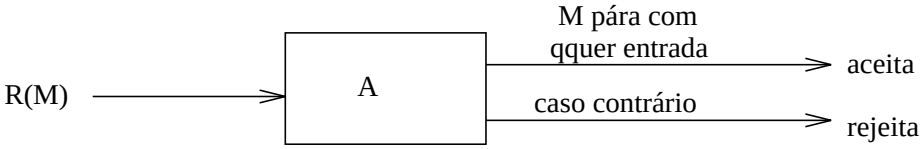


Figura 10.8: MT A que soluciona P_{Σ^*}

Podemos reduzir o problema P_{MT} para P_{Σ^*} através de uma MT que tome como entrada $R(M)$ e produza como saída a descrição de uma MT M' com as seguintes características:

- M' aceita uma cadeia y como entrada;
- apaga y da fita
- coloca x na fita
- volta a cabeça para o início de x
- executa M

Essa máquina M' pode ser construída adicionando-se algumas transições à descrição de M para realizar os passos iniciais descritos acima. Por construção, M' irá parar com a entrada y sse M parar com a entrada x. Poderíamos então ter a MT da Figura 10.9 que resolveria P_{MT}, mostrando que P_{Σ^*} também é indecidível.

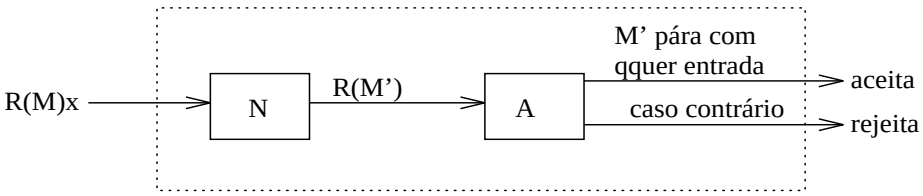


Figura 10.9: Redução de P_{MT} a P_{Σ^*}

10.5 Outros Problemas

Muitos outros problemas podem ser mostrados decidíveis ou indecidíveis através das técnicas vistas neste capítulo. Abaixo mostramos alguns dos problemas relacionados com os modelos vistos neste texto

Linguagem	Modelo	Decidível
Regular	Autômato Finito	$\equiv$
	Expressão Regular	$L = \emptyset$
	Gramática Regular	L finito
		$x \in L$
Livre de	Autômato de Pilha	$L = \emptyset$
Contexto	Linguagem Livre de Contexto	L finito
		$x \in L$
Recursiva	Máquina de Turing	
Enumerável		

Para as linguagens regulares e seus modelos de representação vários problemas são decidíveis entre eles:

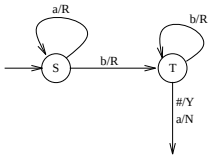
- dados os AF's A e B, $L(A) = L(B)$?
- dado o AF A, $L(A) = \emptyset$?
- dado o AF A, $L(A)$ é finita?
- dado o AF A e a cadeia x, $x \in L(A)$?

Para as linguagens livres de contexto somente a equivalência não pode ser decidida. Já para as linguagens do tipo 0, ou Recursivas Enumeráveis, que são as linguegens definidas através de MT's, nenhum desses problemas tem uma solução computável.

10.6 Exercícios

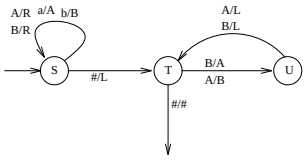
10.1 *Codifique a seguinte MT M sobre o alfabeto $\{0,1\}$. Dê também a representação $R(M)$.*

$< \{a,b\}, \{\#, Y, N\}, \{S,T\}, S, \delta >$



10.2 *Codifique a seguinte MT H sobre o alfabeto $\{0,1\}$. Dê também a representação $R(H)$.*

$< \{a,b\}, \{\#, A, B\}, \{S,T,U\}, S, \delta >$



10.3 *Construa uma MT que decida se um string sobre* $\{0,1\}$ *é uma representação válida de uma MT.*

10.4 *Descreva uma MT que solucione o seguinte Problema de Decisão: "Dada uma MT* M *e um string* x*, a execução de* M *com* x *termina em menos de 100 transições?"*

10.5 *Use redução para mostrar que os seguintes problemas são indecidíveis:*

a. *dada a MT* M*, ela pára quando executada com a cadeia 101?*

b. *dada a MT* M*, ela pára quando executada com alguma cadeia ?*

c. *dada a MT* M*, um estado* q_i *de* M *e ums string* x*, a execução de* M *com* x *entra no estado* q_i*?*

d. *dada a MT* M *e a cadeia* x*,* M *volta a* q_0 *quando executada com* x*?*

Bibliografia

[HOP79] J. E. Hopcroft e J. D. Ullman, Introduction to Automata Theory, Languages and Computation, Addison-Wesley, 1979.

[CAR89] J. Carrol e D. Long, Theory of Finite Automata With an Introduction to Formal Languages, Prentice-Hall, 1989.

[SUD97] T. A. Sudkamp, Languages and Machines: An Introduction to the Theory of Computer Science, Second Edition, Addison Wesley, 1997