Conceitos Elementares da Teoria da Computação
(Módulo 2 - 2.1)

Sonia Limoeiro Monteiro

slmo@lncc.br

Coordenação de Matemática Aplicada e Computacional

Laboratório Nacional de Computação Científica – MCT
http://www.lncc.br

Resumo
Este trabalho aborda conceitos fundamentais relacionados com: os conjuntos recursivamente enumeráveis, as máquinas de Turing, sistemas formais, sistemas de produção e linguagens formais. Neste sentido, ele é uma continuação natural do trabalho anterior publicado no Relatório n^{0} 43/2002 LNCC-MCT.

Abstract
In this work some fundamental concepts related to Recursively Enumerable Sets, Turing Machine, Formal Systems, Production Systems and Formal Language are presented. The contents included in this volume constitute the second part of a previous work published in Relatório n^{0} 43/2002 LNCC-MCT.

Índice

Temos o objetivo de apresentar, de modo resumido e acessível, os conceitos elementares da Teoria da Computação.
O presente texto faz parte de um projeto composto por três módulos que poderão ser utilizados como apoio a cursos de graduação ou pós graduação em áreas diversas.
O primeiro módulo [59] é pré-requisito para os conceitos aqui apresentados e um resumo é encontrado no apêndice deste trabalho.
O módulo 2 é uma continuação natural do módulo 1 e será dividido em duas partes, a saber, módulo 2-2.1 e módulo 2-2.2.
O módulo 2-2.1 que é o presente texto, contém 4 seções numeradas e descritas como abaixo:

- Seção 6 -definimos e apresentamos algumas propriedades dos conjuntos recursivamente enumeráveis e alguns problemas indecidíveis.
- Seção 7 -baseado em [3] e [13], introduzimos três modelos de máquinas com características de memória distintas, sendo um deles a máquina de Turing. Mostramos que eles possuem o mesmo *poder* computacional.
- Seção 8 -baseado em [1], [12], [14] e [53], fazemos um breve histórico relacionando teoria da computação com os fundamentos da matemática.
- Seção 9 -definimos os sistemas de Post (SPP), mostramos que as linguagens geradas por SPP são recursivamente enumeráveis e então, apresentamos a hierarquia de Chomsky e de um modo informativo, as gramáticas estocásticas e as gramáticas fuzzy.

Com isso, fica estabelecido o aparato teórico para o desenvolvimento do módulo2-2.2.

Seção 6

6.1- **Conjuntos Recursivamente Enumeráveis**

Existem conjuntos que podem ser representados por um algoritmo(seção2-2.1) que responde **sim** para os elementos que pertencem ao conjunto e **não** para os elementos que não estão no conjunto. E também, existem conjuntos que não podem ser representados por um algoritmo, mas podem ser representados por um procedimento que responde *sim* apenas para os elementos do conjunto. Tais métodos são denominados *reconhecedores*. Veremos que existem conjuntos que não podem ser reconhecidos por procedimentos ou algoritmos.
Um outro caminho para representar (alguns) conjuntos é do ponto de vista *gerativo*, isto é, podemos construir um procedimento efetivo que gera sucessivamente elementos do conjunto em alguma ordem.
Os conjuntos cujos elementos podem ser efetivamente gerados ou reconhecidos por um procedimento efetivo (programa) são denominados *recursivamente enumeráveis*.
Nesta seção, apresentaremos algumas propriedades dos conjuntos recursivamente enumeráveis. É interessante observar que subconjuntos A de $\mathcal{N}^n$ podem ser codificados como subconjuntos recursivos de $\mathcal{N}$, bastando mostrar que : '*A é recursivo se e somente se* $\{2^{x_1}.3^{x_2}....p_n^{x_n} : (x_1, x_2, ..., x_n) \in A\}$ *é recursivo*'. Assim, a idéia de conjuntos recursivos pode ser estendida a subconjuntos de $\mathcal{N}^n$.

6.1.1-**Definição**

Um subconjunto S dos números naturais é *recursivamente enumerável* (r.e.) se ele é vazio ou se ele é a imagem de alguma função recursiva.

Por exemplo, a função de emparelhamento J *(*seção1-1.8-29*)* define um algoritmo que enumera pares de inteiros positivos.

	0	1	2	3	4	⋯
0	0	2	5	9		⋯
1	1	4	8			⋯
2	3	7				⋯
3	6					⋯
⋮	⋯					

fig.6.1 J(x,y)

Dado qualquer par de inteiros (i, j) ele aparecerá na lista. De fato será o $i+(1/2\ (i+j).(i+j+1))$*-ésimo* par enumerado. Vimos na seção 1-1.8-29 que as funções J e J^{-1} são primitivamente recursivas, portanto, recursivas e totais. Conseqüentemente, o conjunto de pares em $\mathcal{N}$ é um conjunto recursivamente enumerável.

O mesmo raciocínio pode ser aplicado para as linguagens (seção 3-3.4). Podemos pensar em um procedimento que enumera cadeias de símbolos ou sentenças de uma linguagem $\mathcal{L}$. Se as sentenças de $\mathcal{L}$ podem ser geradas por um procedimento efetivo então $\mathcal{L}$ é dita *recursivamente enumerável.* Alternativamente, uma linguagem $\mathcal{L}$ é dita *recursivamente enumerável* se existe um procedimento que reconhece as sentenças da linguagem $\mathcal{L}$. Desta forma, para determinar se uma sentença x está em $\mathcal{L}$, simplesmente enumeramos as sentenças de $\mathcal{L}$ e comparamos x com cada sentença da enumeração. Se x é gerada o procedimento pára reconhecendo x como elemento de $\mathcal{L}$. Naturalmente se x não pertencer a $\mathcal{L}$, o procedimento nunca irá parar.
Existem muitas caracterizações equivalentes da classe dos conjuntos recursivamente enumeráveis, desenvolveremos algumas delas.

6.1.2- **Lema**

Se S é um conjunto r.e. então S é o domínio de uma função parcial recursiva, isto é, existe um programa P tal que para todo $n \geq 0$, P pára com entrada n se e somente se $n \in S$.

Prova

Se S é vazio podemos construir um programa P que entra em *loop* infinito para qualquer entrada. Tal programa tem a seguinte propriedade: para qualquer $n \geq 0$, P pára com entrada n se e somente se $n \in$ S.
Se S é não vazio, então S é a imagem de alguma função recursiva, digamos f. Seja g a função definida por $g(y) = \mu x(f(x)=y)$ e seja h a função definida por $h(y) = f(g(y))$. Desde que f é recursiva, ambas g e h são parciais recursivas. Vemos claramente que S é o domínio de h. Alternativamente, seja P_f um programa que computa f. Seja P o seguinte programa

início: desvie para 2
2: x:= 0 desvie para 3
3: z:= f(x) desvie para 4
4: se $\mathbf{m}(z, y) = 0$ *então desvie para 5 senão desvie para 6*
5: pare
6: x:= S(*x*) *desvie para 3*

Com a variável de entrada y, o programa P pára se e somente se existir algum x tal que $f(x)=y$. Assim, P pára com entrada x se e somente se $y \in S$.

A recíproca do lema 6.1.2 é verdadeira, como se segue.

6.1.3-**Lema**

Para qualquer programa P, o conjunto $\{n \ / \ n \geq 0$ e $P(n)\downarrow\}$ é recursivamente enumerável, isto é, o domínio de uma função parcial recursiva é r.e.

Prova

Se P não pára para qualquer que seja a entrada então o domínio de P é vazio, logo r.e. Por outro lado, seja a o menor n tal que P pára com entrada n. Seja e o número de Gödel de P. Defina o predicado $Pare_e(x,y)$ como

$$Pare_e(x,y) = \begin{cases} 1 & \text{se } P \text{ pára com entrada } x \text{ em } y \text{ passos} \\ 0 & \text{em cc} \end{cases}$$

Seja f a função definida por

$$f(0) = a$$

$$f(S(z)) = \begin{cases} (\mu y \leq z)(\ Pare_e(x,y) \ \text{ e } y \in \{\ f(0),...,f(z)\ \} \\ f(z) \qquad \text{se tal } y \text{ não existir} \end{cases}$$

É fácil ver que f é uma função total computável daí é recursiva (teorema 4.12.3) e que f enumera o domínio de P. Logo, o domínio de P é r.e.

6.1.4 -**Corolário**

Um conjunto é r.e. se e somente se ele é o domínio de alguma função parcial recursiva.

Prova-

Segue imediatamente dos lemas 6.1.2 e 6.1.3

Seja $\varphi_1, \varphi_2, \varphi_3, \ldots$ uma enumeração das funções parciais recursivas (de uma variável), então podemos enumerar os conjuntos r.e. por definir para todo número natural n, $W_n = dom(\varphi_n)$, isto é, $W_n = \{x \mid \varphi_n(x)\downarrow\}$.

Pelo fato de existirem funções parciais recursivas que não são recursivas (e que não podem ser estendidas a funções recursivas totais em um caminho efetivo(lema5.5.5)), nós vemos que existem conjuntos r.e. que não são recursivos. Em particular, defina $K=\{x \mid \varphi_x(x)\downarrow\}$ e $K_0 = \{J(x,\ y) \mid \varphi_x(y)\downarrow\ \}$(onde J é a função de emparelhamento), então ambos K e K_0 são r.e. mas não são recursivos (lemas 5.5.2 , 5.5.3,5.5.4 e corolário 6.1.4). Em nossa notação estabelecida anteriormente, $K=\{x \mid x \in W_x\}$ e $K_0 = \{J(x,\ y) \mid\ y \in W_x\}$. Podemos observar que K e K_0 são definidos considerando os números de Gödel de programas.

6.1.5 –**Teorema**

Um conjunto B é recursivo se e somente se B e $\overline{B}$ são r.e.

Prova (Via tese de Church)

($\Rightarrow$)
Seja B um conjunto recursivo então existe uma função f recursiva tal que:

$$f(n) = \begin{cases} 1 & \text{se} \quad n \in B \\ 0 & \text{se} \quad n \notin B \end{cases}$$

B pode ser gerado através da seguinte função g : $g(0)$ é o menor i tal que $f(i)$ =1, $g(1)$ é o próximo menor , pela tese de Church isto é suficiente. No entanto, suponha B infinito então g pode ser definida:

$$g(0) = \mu\, y\ (f(y) = 1)$$
$$g(n+1) = \mu\ \ y(\, sg(\boldsymbol{m}(y, g(n)).f(y) = 1)$$

Analogamente, mostramos que $\overline{B}$ é r.e.
$\Leftarrow$
Suponhamos que ambos B e $\overline{B}$ são r.e. e que g gera B e h gera $\overline{B}$. Para mostrar que B é recursivo, precisamos construir uma função recursiva que reconhece os elementos de B. Assim, suponhamos que desejamos saber se um x qualquer está ou não em B. Se utilizarmos as funções g e h acima definidas, então x será computado pois $g(\mathcal{N}) \cup h(\mathcal{N}) = \mathcal{N}$, se x é computada por g então $x \in B$, se por h então $x \in \overline{B}$.

Pelo teorema 5.4.1 os conjuntos recursivos formam uma álgebra Booleana de conjuntos enquanto que os conjuntos r.e. são fechados para a união e interseção porém não são fechados para o complemento.

6.1.6 –**Teorema**
Todo conjunto recursivo é r.e.

6.1.7 – **Teorema**
Um conjunto A é recursivo se e somente se A é finito ou A é a imagem de uma função recursiva estritamente crescente.

Prova
$\Rightarrow$
Seja A recursivo e f tal que

$$f(0) = (\mu y)\ (\chi_A(y))$$
$$f(x+1) = (\mu y)\ (\, y > f(x) \wedge \chi_A(y))$$

Os valores *f(0),f(1),...* são elementos de *A* em ordem crescente.

$\Leftarrow$
Seja *A* infinito e imagem de uma função recursiva estritamente crescente *f*. Seja $h(x) = \mu y\ (f(y) \geq x)$. Então

$$\chi_A(x) = \begin{cases} 1 & \text{se } f(h(x)) = x \\ 0 & \text{se } f(h(x)) > x \end{cases}$$

6.1.8 – **Corolário**

O complemento $\overline{K}$ de *K* não é r.e., e o complemento $\overline{K_0}$ de K_0 não é r.e.

Prova

Se $\overline{K}$ e $\overline{K_0}$ fossem r.e. então K e K_0 seriam recursivos, o que não acontece.

6.1.9 –**Teorema**

Os seguintes conjuntos de números de Gödel não são r.e.

a) **Total**= $\{n \mid \varphi_n$ é total$\}$

b) **Vazio =** $\{n \mid dom\ \varphi_n$ é vazio$\}$

c) **Finito** = $\{n \mid dom\ \varphi_n$ é finito$\}$

Seguindo Cutland[3], apresentaremos a prova para o conjunto Total

Prova

Seja *f* uma função total de um argumento computável que enumera Total; isto é, $\varphi_{f(0)}, \phi_{f(1)}, \ldots$ é uma lista de todas as funções de um argumento que são computáveis. Pelo método da diagonal de Cantor, podemos fazer uma construção de uma função computável e total *g* que não ocorre na lista acima, assim

$$g(x) = \phi_{f(x)} + 1$$

Então *g* é computável e total, mas $g \neq \phi_{f(m)}$ para todo *m*, o que é uma contradição.

6.1.10– **Teorema**

Todo conjunto r.e. infinito tem um subconjunto recursivo infinito.

Prova

Seja A a imagem de uma função recursiva. Podemos efetivamente enumerar um subconjunto de A em ordem crescente por uma função g como se segue:

$$g(0) = f(0)$$
$$g(n+1) = f(x) \qquad \text{onde } x=\mu y\ (f(y)>g(n))$$

Desde que A é a imagem de f, g é totalmente definida. Por construção $Imag\ g \subseteq Imag\ f$ e g é crescente. Como g é construída por minimização e recursão ela é computável. Daí, pelo teorema 6.1.6 $Imag\ g$ é um suconjunto recursivo infinito de A.

6.2 - **Indecidibilidade de Problemas**

Vimos que o problema de decisão para um conjunto A é o problema de decidir efetivamente (através de um algoritmo) se um determinado elemento está ou não em A. Pela tese de Church, um conjunto é decidível se ele tem uma função característica recursiva, isto é, a função

$$\chi_A = \begin{cases} 1 & \text{se } x \in A \\ 0 & \text{se } x \notin A \end{cases}$$

é computável.

Vimos que o conjunto K não é recursivo, daí é indecidível pela tese de Church. Isto significa que não existe um algoritmo para decidir se $x \in dom\,\varphi_x$, ou de forma equivalente, decidir se $U(x,x)\downarrow$. Tanto o problema de decisão para K quanto para K_0 são denominados: **o problema da parada** (lema 5.5.3). Podemos usar a indecidibilidade do conjunto K para provar que outros problemas também são indecidíveis. Este método de reduzir o problema da parada para outro problema é um dos mais importantes para provar indecidibilidade.

6.2.1 – **Alguns Problemas Indecidíveis**

Neste item, enunciaremos alguns problemas indecidíveis.

a) Detecção de vírus

Seria interessante se pudéssemos detectar automaticamente quais programas podem espalhar vírus e quais não podem, submetendo-os a um programa que funcionasse como um filtro. Dowling[30] mostrou que isto não é possível, ou seja, não existe nenhum programa que faça isso corretamente. Para isso, ele considerou que os programas são executados em uma máquina tal como um computador moderno e

apresentou uma definição para vírus como sendo um programa que quando executado altera o código do sistema operacional. Seja a seguinte definição:

'Um programa P, com uma entrada x, espalha um vírus se: quando P é executado no sistema operacional SO com entrada x, P(x) altera SO. Caso contrário, P é ***safo com entrada x****. Um programa é* ***safo*** *se ele é safo para qualquer entrada'.*

Para reduzir este problema ao problema da parada , seja o seguinte programa

$$safo(P,x) = \begin{cases} 1 & \text{se } P \text{ é safo com entrada } x \\ 0 & \text{caso contrário} \end{cases}$$

A partir de *safo*, podemos construir um programa que tem o seguinte comportamento

$$teste(\mathrm{P}) = \begin{cases} \text{imprime 'OK'} & \text{se } safo(P,P)=0 \\ \text{altera SO} & \text{caso contrário} \end{cases}$$

Usando o método da diagonal, ou melhor, pressupondo que : a) *teste* com entrada *teste* **é** safo, ou, b) *teste* com entrada *teste* **não é** safo e com argumentos semelhantes aos expostos na seção 4 - 4.13, podemos concluir que não pode existir um programa que decide se um **programa arbitrário** está ou não com vírus.

b) O problema '$\varphi_x = \varphi_y$' é indecidível

O que este resultado indica é que não é possível verificar automaticamente se dois programas arbitrários são equivalentes.

c) Teorias de primeira ordem

É indecidível se uma expressão do cálculo dos predicados de primeira ordem é um teorema da aritmética.

d) O problema da correspondência de Post (PCP) [Hopcroft26]

Seja $\sum$ um alfabeto finito e sejam α e β palavras em $\sum^+$ tal que α e β formam um conjunto finito de pares de palavras sobre $\sum$,isto é, $\{(\alpha_1, \beta_1), (\alpha_2, \beta_2),..., (\alpha_n, \beta_n)\}$ onde

$$\alpha = \alpha_1\alpha_2...\alpha_n \text{ e } \beta = \beta_1\beta_2...\beta_n$$

Dizemos que uma instância de um PCP tem solução se existe uma seqüência finita de índices $i_1, i_2, ...i_m$, com $m \geq 1$, tal que:

$$\alpha_{i_1}\alpha_{i_2}...\alpha_{i_m} = \beta_{i_1}\beta_{i_2}...\beta_{i_m}.$$

Neste sentido $i_1, i_2, ...i_m$ é uma solução para PCP.

É indecidível se um PCP arbitrário tem solução. No entanto, um PCP particular pode ter solução, como é mostrado no exemplo abaixo.

6.2.1.1- **Exemplo**

Seja $\sum = \{0,1\}$. Sejam α e β, como na tabela abaixo, palavras que contém três seqüências de símbolos

	α	β
i	α_i	β_i
1	1	111
2	10111	10
3	10	0

Neste caso, PCP tem uma solução. Seja $m = 4$, $i_1 = 2$, $i_2 = 1$, $i_3 = 1$ e $i_4 = 3$. Então $\alpha_2\alpha_1\alpha_1\alpha_3 = \beta_2\beta_1\beta_1\beta_3 = 101111110$.

e) O décimo problema de Hilbert[60] [58]

Um polinômio é uma soma de termos, onde cada termo é um produto de certas variáveis multiplicada por uma constante, chamada um coeficiente. Por exemplo, $2x.x.x.y.z = 2x^3yz$ é um termo com coeficiente 2. O polinômio $6x^3yz^2 + 3xy^2 - x^3 - 10$ tem 4 termos sobre as variáveis x,y e z. Uma raiz de um polinômio é uma atribuição de valores para suas variáveis de modo que o valor do polinômio é 0. O polinômio acima tem raiz em $x = 5$, $y = 3$, e $z = 0$. Quando, para todas as variáveis são atribuídos valores inteiros, a raiz é dita integral. O décimo problema de Hilbert seria a construção de um algoritmo que testasse se um polinômio **qualquer** possui ou não raiz integral. Assim, o décimo problema de Hilbert é equivalente a perguntar se o conjunto

$$D = \{p \mid p \text{ é um polinômio com uma raiz integral}\}$$

é decidível. Davis,Putnam&Robson[22] demonstraram que D é indecidivel.

Na próxima seção, introduziremos algumas máquinas que têm aplicações importantes em Ciência da Computação. Veremos que as máquinas com apenas um único registro são capazes de computar todas as funções computáveis em máquinas com um número ilimitado de registros em um alfabeto com no mínimo 2 símbolos.

Seção **7**

Nesta seção, apresentaremos uma definição geral para máquina, e então caracterizaremos três tipos de máquinas, a saber: máquinas com um número ilimitado de registros(MIR, seção 2), máquinas com um único registro (MUR) e máquinas de Turing (MT). Tais modelos possuem características de memória fixa. Comandos para as linguagens desenhadas para estas máquinas devem especificar, de modo explícito, como palavras armazenadas na memória devem ser manipuladas. Para isso definiremos algumas funções que manipulam palavras e mostraremos que são primitivas recursivas. A necessidade de introduzir novas funções se dá pelo fato de que as funções computáveis foram definidas para domínios e contradomínios em $\mathcal{N}$. Na verdade, funções $f:\mathcal{N}^n \to \mathcal{N}$ definidas como parciais recursivas(recursivas e primitivas recursivas) são conjugados de funções parciais recursivas (recursivas e primitivas recursivas) em um alfabeto Σ. O conjugado da multiplicação em Σ_n é o conjugado da multiplicação em Σ_m. Por outro lado, concatenação em Σ_m não é o conjugado da concatenação em Σ_n, porque

$$x \cap y = m^{comprimento(y)} \times x + y \text{ em } \Sigma_m \quad \text{e} \quad x \cap y = n^{comprimento(y)} \times x + y \text{ em } \Sigma_n .$$

O principal objetivo desta seção é apresentar as máquinas de Turing e mostrar que os diferentes modelos de máquinas que apresentaremos possuem o mesmo poder computacional, isto é, se uma função é efetivamente computável então ela é efetivamente computável em qualquer um dos modelos mencionados acima.

7.1 – **Recursão sobre palavras**

Neste item introduziremos algumas funções para manipulação de palavras.

7.1.1- **Teorema**

Seja Σ um alfabeto. Então, as seguintes funções são primitivas recursivas

a) $hip(x) = \begin{cases} x' & \text{se } x = x'\sigma \ , \ \sigma \in \Sigma \\ 0 & \text{se } x = 0 \end{cases}$

b) $concl(x) = \begin{cases} \sigma & \text{se } x = x'\sigma \\ 0 & \text{se } x = 0 \end{cases}$

É conveniente notar que o símbolo '0' representa a palavra nula.

Prova -

As funções anteriores são obtidas por operações aritméticas. Observando que, se $\sigma_k\sigma_{k-1}...\sigma_0$ então *hip(x)*= $\sigma_k\sigma_{k-1}...\sigma_1$. Como

$$\nu(x) = (\sum_{j=0}^{k} v(\sigma_j).n^j$$

e

$$\nu(x') = (\sum_{j=1}^{j=k} v(\sigma_j).n^{j-1}$$

$$v(hip(x)) = \frac{v(x) - v(\sigma_0)}{n}$$

assim, como a função *hip* tem a propriedade que para várias entradas diferentes existe a mesma saída, (por exemplo em $\sum_2$, *hip(*1221*)=hip(*1222*)*=122), então o último símbolo pode ser eliminado subtraindo-se 1 e dividindo-se o resultado por *n*, no caso 2. Daí

$$hip(x) = div\,\mathrm{int}(p(x), n)$$

e

$$concl(x)=\mathbf{m}(x,.(hip(x),n)$$

7.1.2 – **Exemplo**

Suponhamos $\sum_3$ ={1,2,3}, dado *w*= 123 *concl*(*w*) =3, pois

concl(123) =*m(123, (.(12,3)*$_3$*)*= $\mathbf{m}(123,113) = 3.3^0 + 2.3^1 + 1.3^2 - (3.3^0 + 1.3^1 + 1.3^2) = 3$

7.1.3 - **Teorema**

Se $g : \omega^r \to \omega^s$, $h_\sigma : \omega^{r+s+1} \to \omega^s, \sigma \in \sum$ são funções parciais (primitiva/recursiva) recursiva, então *f* é definida recursivamente de *g* e *h* por

$$f(x,0) = g(x)$$
$$f(x,y\sigma) = h_\sigma[x,y,f(x,y)]$$

7.1.4 –**Teorema**

As seguintes funções são primitivamente recursivas:

$a) D_\sigma = x.\sigma, \quad \sigma \in \sum$

$b) E_\sigma = \sigma.x, \quad \sigma \in \sum$

c) $x \frown y$ concatenação

d) ρ a função reversa

e) $cdr(x) = \begin{cases} x' & \text{se } x=\sigma x' \\ 0 & \text{se } x=0 \end{cases}$

f) comp(x) o número de letras de *x*

g) ocorr(w,x) o número de vezes que *w* ocorre em *x*

$$h)\ x \leq w = \begin{cases} 1 & \text{se x é subpalavra de } w \\ 0 & \text{cc} \end{cases}$$

$$i)\ menori(w,x) = \begin{cases} u & \text{se } u \text{ é a menor palavra tal que } w=u.x.v \text{ para algum v} \\ 0 & \text{se } \neg(x \leq w) \end{cases}$$

$$j)\ menorfi(w,x) = \begin{cases} v & \text{se } u \text{ é a menor palavra tal que } w=u.x.v \\ 0 & \text{se } \neg(x \leq w) \end{cases}$$

$$k)\ subst(w,x,y) = \begin{cases} u.y.v & \text{se } u \text{ é a menor palavra tal que } w = u.x.v \\ w & \text{se } \neg(x \leq w) \end{cases}$$

l) part(w,x,k) = a subpalavra de w que está entre a k-ésima e a k+1-ésima ocorrência disjunta de x em w quando w é varrido da esquerda para a direita. Se o número de ocorrências disjuntas de x em w for menor do que k+1 então $part(w,x,k) = 0$.

Prova

Apenas para a,b e c.

a) $D_\sigma(x) = +(.(n,x),\sigma)$

b) usando c

$E_\sigma(x) = \sigma \cap x$

c)

$\cap(x,0) = x$

$\cap(x,y\sigma) = D_\sigma(\cap(x,y))$

7.2- **Máquinas e Programas**

Em termos muito simples um computador consiste de uma coleção de registros e algumas instruções que podem modificar e testar dados armazenados nestes registros. Um computador real possui facilidades de entrada e saída que transferem informações entre o mundo externo e os registros da máquina. A seguir apresentaremos uma definição para máquinas que captura as propriedades básicas de um computador real e que nos permite estudar suas propriedades formais.

7.2.1 - **Definição**

Uma **máquina** em um alfabeto Σ é uma 6-tupla $\mathcal{M} = <\Sigma, \mathcal{R}, \mathcal{C}, \mathcal{I}, e, s>$ consistindo de :

a) Um alfabeto $\Sigma = \Sigma_n$, para algum $n \geq 1$, denominado alfabeto de entrada.
b) Um conjunto enumerável $\mathcal{R}$ de registros, onde cada qual contém qualquer elemento de $\mathcal{W} = \Sigma^*$.
c) Um conjunto $\mathcal{C}$ de configurações de memória.
d) Um conjunto $\mathcal{I}$ de instruções, onde cada elemento de $\mathcal{I}$ é uma função f: $\mathcal{C} \rightarrow \mathcal{C}$.
e) Uma função de entrada $e : \mathcal{W}^r \rightarrow \mathcal{C}$ e uma função de saída $s : \mathcal{C} \rightarrow \mathcal{W}^r$.

Cada elemento de $\mathcal{C}$ é uma possível coleção de informações que podem ser armazenadas nos registros. Por exemplo, se existem 2 registros e se $\mathcal{C} = \mathcal{W} \times \mathcal{W}$, então $(x_1, x_2) \in \mathcal{C}$ significa que $x_1 \in \mathcal{C}$ está armazenado no registro 1 e $x_2 \in \mathcal{C}$ esta armazenado no registro 2. As instruções de uma máquina são funções não necessariamente totais cujos argumentos e valores são configurações de memória. A função e mapeia uma *n-tupla* de palavras de entrada na configuração de memória enquanto s mapeia uma configuração de memória em uma *s-tupla* de palavras de saída. As funções e e s não são necessariamente totais.

Associada a cada máquina existe uma linguagem *algoritimica*. Para as máquinas que trabalharemos estas linguagens de máquina são equivalentes a LP (seção 4). A diferença é que as linguagens de máquina são projetadas para as características de uma particular máquina, o que não acontece com as linguagens denominadas de alto nível.

Um programa para uma máquina é um sistema $\mathcal{P} = < \mathcal{L}, \mathcal{I}n, \mathcal{S}, \mathcal{T}e >$ onde $\mathcal{L}$ é um conjunto finito de rótulos, $\mathcal{I}n \subseteq \mathcal{L}$ é um conjunto de rótulos iniciais, $\mathcal{S} \subseteq \mathcal{L} \times \mathcal{I} \times \mathcal{L}$ é um conjunto finito de comandos e $\mathcal{T}e \subseteq \mathcal{L}$ é um conjunto de rótulos terminais.

Vamos considerar as seguintes funções como funções primitivas necessárias ao funcionamento de uma máquina.

a) ${}_rE_\sigma : W \rightarrow W; E(x) = \sigma \cap x = \sigma x$, concatena um símbolo à esquerda da palavra armazenada no registro r .

b) ${}_rD_\sigma : W \rightarrow W; D_\sigma(x) = x \cap \sigma = x\sigma$, concatena um símbolo à direita da palavra armazenada no registro r.

c) ${}_rE_\sigma^{-1} : W \rightarrow W; E_\sigma^{-1}(x) = \begin{cases} x' & se\ \ x = \sigma x' \\ \uparrow & caso\ contrario \end{cases}$, retira um símbolo à esquerda da palavra armazenada no registro r.

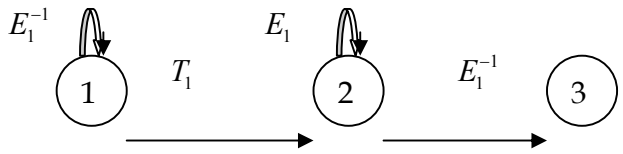
d) ${}_rD_\sigma^{-1} : W \rightarrow W; D_\sigma^{-1}(x) = \begin{cases} x' & se\ x = x'\sigma \\ \uparrow & caso\ contrario \end{cases}$, retira um símbolo à direita da palavra armazenada no registro r.

e) ${}_rT : W \rightarrow \{0, \uparrow\}; T(x) = \begin{cases} 0 & se\ \ x = 0 \\ \uparrow & caso\ contrario \end{cases}$, testa se o registro r está vazio(contém a palavra nula).

Podemos observar que as funções (a) e (b) são totais, ao passo que (c), (d) e (e) são parciais.

7.2.2- **Exemplo**

Consideremos uma máquina com um único registro onde $\mathcal{C} = W_2 = \Sigma^*$ e $\mathcal{I} = \{ E_1^{-1}, E_1, T \}$ e seja $\mathcal{P}$ o programa $< \mathcal{L}, \mathcal{I}n, \mathcal{S}, \mathcal{T}e >$ onde $\mathcal{L} = \{1,2,3\}$, $\mathcal{I}n = \{1\}$, $\mathcal{S} = \{(1, E_1^{-1},1), (1, T, 2),(2, E_1, 2),(2, E_1^{-1},3)\}$ e $\mathcal{T}e = \{3\}$. Podemos visualizar este programa através do seguinte grafo dirigido



7.3 - **Programa Determinístico e Completo**

Um caminho de comprimento k é uma seqüência de comandos da forma $p = (l_0, f_1, l_1) (l_1, f_2, l_2) ... (l_{k-1}, f_k, l_k)$. Cada rótulo é considerado um caminho de comprimento 0. A computação de um caminho p é $|p| = f_k \, o \, ... o \, f_1$. A computação de um caminho de comprimento 0 é a função identidade em $\mathcal{C}$. A computação de um caminho aplicado a uma configuração de memória produz uma saída somente se a configuração inicial está no domínio da computação. Um **caminho é bem sucedido** (*cbs*) se $l_0 \in \mathcal{I}n$ e $l_k \in \mathcal{T}e$.

Cada programa $\mathcal{P}$ define uma relação $|\mathcal{P}| = \{(x,y) | x, y \in \mathcal{C}$ e existe um *cbs* $|p| = f_k \, o \, ... o \, f_1\}$. Assim $(x, y) \in |\mathcal{P}|$ significa que para algum *cbs* p, se o conteúdo inicial dos registros da máquina for $x \in \mathcal{C}$, o conteúdo dos registros poderia ser y quando o programa alcança o rótulo terminal no fim do comando.

Note que $|\mathcal{P}|$ não é necessariamente uma função, desde que para dois diferentes valores de y, (x, y) poderia estar na relação $|\mathcal{P}|$. Por exemplo, para o programa anterior, ambos (111,1) e (111,11) estão em $|\mathcal{P}|$. E ainda, para nenhum y, (22,y) está em $|\mathcal{P}|$, pois nenhum comando com rótulo 1 contém uma instrução tendo 22 em seu domínio.

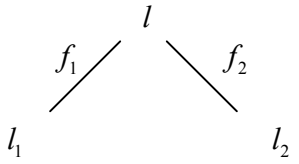
Os programas não determinísticos, isto é, programas que computam dois diferentes valores de y para um dado x, não serão tratados nesta seção. Entretanto um resultado da Teoria da Computação, decorrente da tese de Church, assegura que os programas não determinísticos não incrementam o poder computacional de qualquer máquina que já computa as funções parciais recursivas (Brainerd[4]).

7.3.1 –**Definição**

Um programa $\mathcal{P}$ é determinístico se

(a) ele possui um rótulo inicial
(b) se (l, f_1, l_1)≠ (l, f_2, l_2) então $dom\ f_1 \cap dom\ f_2 = \varnothing$.

Em (a) temos que toda computação começa no mesmo rótulo e em (b) temos que para qualquer configuração de memória e qualquer rótulo, pelo menos uma instrução tendo aquele rótulo é aplicável a configuração de memória.



Assim, um programa é determinístico se na hipótese de existirem dois comandos com o mesmo rótulo, com funções ou instruções diferentes, a interseção do domínio destas funções é vazia. Ou seja, apenas uma das instruções poderá ser executada a cada vez.
Um *programa saída* é um programa que para todo rótulo não contém comandos da forma (l_1, f, l_2) com l_1 terminal.

7.3.2 **- Lema**

Se p_1 , p_2 são *cbs* diferentes de um programa saída e determinístico $\mathcal{P}$, então $dom|\ p_1| \cap dom\ |p_1| = \varnothing$.

Prova

Como $\mathcal{P}$ é um programa saída então p_1 *e* p_2 se ramificam em algum ponto. Seja $f = |p|$, $f'_1 = |p'_1|$ e $f'_2 = |p'_2|$. Então $f_1 = f'_1\ o\ f = |p_1|$ e $f_2 = f'_2\ o\ f = |p_2|$. As seguintes asserções são sempre verdadeiras para quaisquer funções *g,f,k,h,s* :

a) $dom(gof) \subseteq domf$
b) Se $domh \cap domk = \varnothing$ então $dom(soh) \cap dom(gok) = \varnothing$
c) Se $domh \cap domk = \varnothing$ então $dom(hos) \cap dom(kog) = \varnothing$

Como $\mathcal{P}$ é determinístico, por (b) temos que $dom\ f'_1 \cap dom\ f'_2 = \varnothing$. Logo $dom|p_1| \cap dom|p_2| = dom\ f_1 \cap dom\ f_2 = dom(f'_1\ o\ f) \cap dom(f'_2\ o\ f) = \varnothing$ por (c).

7.3.3 -**Definição**

Um *programa completo* é um programa saída e determinístico que tem a seguinte propriedade adicional: para cada rótulo não terminal *l*, $\cup domf = \mathcal{C}$, onde a união é tomada sobre toda *f* tal que (*l,f,l´)* é um comando para algum rótulo *l´*.

7.3.4- **Exemplo**

Seja a máquina em $\sum_2 = \{1, 2\}$ com 2 registros , C ={(1,1),(1,2),(2,1),(2,2)} e o programa

1 $i \times E_1^{-1}$ 2, $T \times i$ 4, $i \times E_2^{-1}$ 3

2 $D_1^{-1} \times i$ 1

3 $D_2^{-1} \times i$ 1

4 $i \times T$ 5

Nosso objetivo é verificar se este programa é completo, para isto, testaremos se em todos rótulos se aplicam os pares de instruções de cada comando. Sabemos que a função identidade *i* não altera o conteúdo do registro. Do rótulo 1 saem três instruções que não podem ocorrer simultaneamente, e portanto, os domínios destas funções têm interseção vazia. Nos rótulos 2,3,4 só há uma instrução a cumprir em cada um deles, resultando uma só saída possível de cada uma. Como a união dos domínios das funções que compõem o programa é definida pelo conjunto {(1,1),(1,2),(2,1),(2,2)}, podemos afirmar que este programa não é completo, de vez que, se tivéssemos o registro da direita vazio não poderíamos executar nenhuma das instruções do rótulo 1.

7.3.5 - **Teorema**

Todo programa, saída, completo e determinístico computa uma função.

Prova

Como a composição de funções é uma função, a computação de um *cbs* também é uma função. Pelo lema anterior, o domínio de todos os *cbs´*s deve ser disjunto, daí $|\mathcal{P}|$ é uma função , e portanto $\omega \, o \, |\mathcal{P}| \, o \, \alpha$ será também uma função.

7.4 **- Máquina com um número ilimitado de registros** (MIR)

Reformularemos a definição da máquina MIR (seção2) adicionando funções de entrada e saída que não eram explícitas em MIR. Introduziremos uma nova linguagem $\mathcal{L}_{mir}$, de nível mais baixo do que a linguagem apresentada na seção 2, que foi construída através dos comandos *Z(n), S(n), T(m,n) e P(m,n,q).* Esboçaremos um compilador para a linguagem LP em MIR com a finalidade de demonstrar que toda função computada por um programa LP é também computada por um programa na linguagem $\mathcal{L}_{mir}$.

As máquinas MIR consistem de um conjunto enumerável de registros com configurações de memória $\mathcal{C} = W^{\infty}$ e o seguinte conjunto de instruções

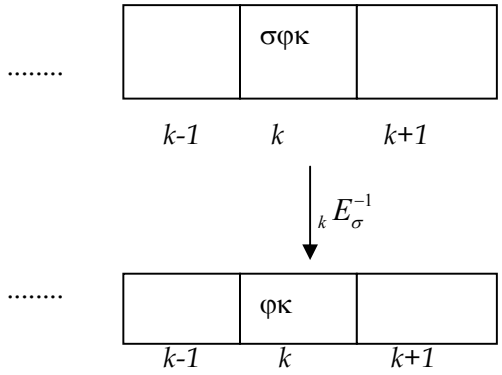
$$\mathcal{I} = \{ {}_kE^{-1}{}_\sigma, {}_kD_\sigma, {}_kT \mid k \geq 0 \text{ e } \sigma \in \Sigma \}$$

onde

$${}_kE^{-1}{}_\sigma = i_k \times E^{-1}{}_\sigma \times i_\infty$$
$${}_kD_\sigma = i_k \times D_\sigma \times i_\infty$$
$${}_kT = i_k \times T \times i_\infty$$

e i_∞ significa $i \times i \times i \times ...$, k indica o *k-ésimo* registro

A instrução ${}_kE_\sigma^{-1}$ testa se o símbolo à esquerda do *k-ésimo* registro é σ; e se for, ele é retirado



A instrução ${}_kD_\sigma$ escreve σ à direita do registro k e a instrução ${}_kT$ checa se o registro k está vazio(se contém a palavra nula). As funções de entrada/saída serão definidas por $e_r(x_1, x_2, ..., x_r) = (0, x_1, x_2, ..., x_r, 0, 0, ...)$ e $s_s(y_0, y_1, y_2, ..., y_s, y_{s+1}, ...) = (y_1, y_2, ..., y_s)$

7.4.1 –**Macros**

O nosso objetivo é construir um compilador para a linguagem LP, utilizando as instruções que esta máquina é capaz de executar. Para tanto é conveniente definir previamente, MACRO instruções. Uma MACRO instrução é uma instrução que contém um conjunto de outras instruções mais primitivas, e cujo nome, quando chamado, permite a execução do conjunto. A sua utilização facilita o cumprimento de seqüências de instruções que serão usadas em várias partes do programa.
Para simplificar a notação escreveremos:

$$l \quad {}_kE_\sigma^{-1} \quad l.\sigma \quad \sigma \in \Sigma \quad \text{para} \quad l \quad {}_kE_{\sigma_1}^{-1} \quad l.\sigma_1, ..., {}_kE_{\sigma_m}^{-1} \quad l.\sigma_m \quad \text{onde } \Sigma = \{\sigma_1, ..., \sigma_n\}$$

nome da macro	tabela 7.4.1	definição ($\sigma \in \sum$)
1. *l limpe(k) l´*		$l \;\; {}_kT \;\; l', {}_k E_\sigma^{-1} \;\; l \quad \sigma \in \sum$
2. *l conc(k,j) l´* $j \neq k$		$l \;\; {}_jT \;\; l', {}_j E_\sigma^{-1} \;\; l.\sigma$ $l.\sigma \;\; {}_kD_\sigma \;\; l$
3. *l transfere (k,j)* $j \neq k$		*l limpe(k)* *l.1 conc(k,j) l´*
4. *l copie (k,j) l´* $j \neq 0, k \neq 0, j \neq k$		*l limpe(k)* *l.1 limpe(0)* $l.2 \;\; {}_jT \;\; l.2.1, {}_j E_\sigma^{-1} \;\; l.3.\sigma$ $l.3.\sigma \;\; {}_0D_\sigma \;\; l.4.\sigma$ $l.4.\sigma \;\; {}_kD_\sigma \;\; l.2$
5. $l \;\; L_\sigma \;\; l' \;\; k \neq 0$		*l limpe(0)* $l.1 \;\; {}_0D_\sigma$ *l.2 conc(0,k)* *1.3 transfere(k,0) l´*

Explicação

- A macro *limpe(k)* faz com que o conteúdo do registro *k* seja zero.

- Se $j \neq k$, *conc(k,j)* concatena o conteúdo do registro *k* ao fim da direita do registro *k*. O conteúdo do registro *j* , após a execução de *conc* ,passa a ser 0.

- Se $j \neq k$, *transfere(k,j)* faz com que o conteúdo do registro *k* seja igual ao conteúdo do registro *j* . Após a execução de *transfere,* o conteúdo do registro *j* passa a ser 0.
- *Copie(k,j)* é a mesma que *transfere* sem alterar o conteúdo do registro *j*. Uma vez que o conteúdo do registro 0 é 0 , *transfere(k,j)* esta correto somente se $j \neq 0$ e $k \neq 0$. O uso de *transfere* requer $j \neq k$.
- Para $k \neq 0$, ${}_kE_\sigma$ concatena σ no final da esquerda do conteúdo do registro *k.*

7.4.2- **Exemplo**

Seja ζ a função sucessor, definida indutivamente através da concatenação , como se segue

Para $\sigma \in \sum_n$, $\sigma \neq n$, seja σ' o próximo dígito maior que σ.

$$\zeta(0) = 1$$

$$\zeta(x\sigma) = \begin{cases} x \cap \sigma' & \text{se } \sigma \neq n \\ \zeta(x) \cap 1 & \text{se } \sigma = n \end{cases}$$

Considerando o registro *k+1* vazio um programa em $\mathcal{L}_{mir}$ (não otimizado), que computa a função acima em $\sum_2$ ={1,2}, pode ser dado da seguinte forma:

1 ${}_kT\ 4,\ {}_kE_1^{-1}\ 2,\ {}_kE_2^{-1}\ 3$
2 ${}_{k+1}E_1\ 1$
3 ${}_{k+1}E_2\ 1$
4 ${}_{k+1}T\ 6,\ {}_{k+1}E_1^{-1}\ 5,\ {}_{k+1}E_2^{-1}\ 8$
5 ${}_kE_2\ 6$
6 ${}_{k+1}T\ 7,\ {}_{k+1}E_1^{-1}\ 6.1,\ {}_{k+1}E_2^{-1}\ 6.2$
6.1 ${}_kE_1\ 6$
6.2 ${}_kE_2\ 6$
7 ${}_kT\ 9,\ {}_kE_1^{-1}\ 7.1,\ {}_kE_2^{-1}\ 7.2$
7.1 ${}_kE_1\ 10$
7.2 ${}_kE_2\ 10$
8 ${}_kE_1\ 8.1$
8.1 ${}_{k+1}T\ 8.2,\ {}_{k+1}E_1^{-1}\ 5,\ {}_{k+1}E_2^{-1}\ 8$
8.2 ${}_{k+1}E_1\ 6$
9 ${}_kD_1\ 10$
10 *Pare*

Veremos que os programas em MIR podem computar qualquer função LP computável. Isso implica pela tese de Church que MIR pode computar qualquer função efetivamente computável.

7.4.3 - **Um tradutor para Programas LP**

Vimos que toda função parcial recursiva é LP computável. Um caminho para mostrar que cada função computada por programas LP é também computada por um programa em MIR é dar um procedimento que traduz programas LP em programas equivalentes MIR. Tal procedimento pode ser chamado *compilador*.

De forma bem simplificada, esboçaremos um compilador para a linguagem LP, considerando o alfabeto $\sum_1$. A descrição do compilador é bastante informal. A tabela 7.4.2 mostra a tradução do comando LP em $\mathcal{L}_{mir}$ em $\sum_1$.

$\mathcal{LP}$	tabela 7.4.2	$\mathcal{L}_{mir}$
l *inicio* *desvie para p*		l *limpe(0) p*
$l\ x_k := 0$ *desvie para p*		l *limpe (k) p*
$l\ x_i := x_j$ *desvie para p*		l *copie*(i,j) *se* $i \neq j$ l *limpe(0)* *se* $i = j$
$l\ x_k = S(x_k)$ *desvie para p*		$l\ {}_kD_1\ p$
$l\ x_k := P(x_k)$ *desvie para p*		$l\ {}_kT\ l.1,\ {}_kE_1^{-1}\ p$ $l.1\ \lim pe(0)\ p$
l *se* $x_k := 0$ *entao desvie para p senao desvie para q*		$l\ {}_kT\ p,\ \lim pe(0)\ q$
l *pare*		l *limpe(0)* l' *(onde* l' *é um rótulo não existente)*

Seja o programa LP que calcula a soma entre 2 números em $\sum_1$

1 *inicio* 4
4 *se* $x_1 := 0$ *então desvie para 5*
senão desvie para 3
3 $y_3 := y_3 + 1$ *desvie para 6*
6 $x_1 := P(x_1)$ *desvie para 4*
5 *pare*

Vamos compilar o programa acima em MIR. Por questão de simplificação, vamos assumir o alfabeto $\sum_1$, o que não tem o menor problema, pois, podemos mostrar que toda função LP computável em $\sum_n$ é MIR computável em $\sum_1$. Para isso, é suficiente utilizarmos as funções de entrada e saída $e(x_1,...,x_r) = (0, k(x_1),...,k(x_r),0...)$ e $s(y_0, y_1,...y_s,...) = (k^{-1}(y_1),...,k^{-1}(y_s))$ onde $k = v_1^{-1} \circ v_n$ e $v_1 : W_1 \to N, v_n : W_n \to N$ são as bijeções definidas na seção 3 do módulo 1.

Para compilar o programa LP , vamos seguir os seguintes passos:

- Passo1- Substitua todas as variáveis no programa LP pelos nomes $x_1,...,x_j,x_{j+1},...,x_{j+k}$. A variável x_k será armazenada no registro *k* da MIR.

- Passo 2- Transforme o programa LP em sua forma padrão (módulo1-2.5). Desta forma, o primeiro comando tem rótulo 1, o segundo tem rótulo 2 e etc.

Aplicando os passos 1 e 2 no programa LP temos:

1 *início* 2
2 *se* $x_1 := 0$ *então desvie para 5*
senão desvie para 3
3 $x_2 := x_2 + 1$ *desvie para 4*
4 $x_1 := P(x_1)$ *desvie para 2*
5 *pare*

- Passo 3 – Faça $m = \max\{k \mid x_k \text{ ocorre no programa}\}$. No exemplo *m* = *2*. Em qualquer ponto do processo de tradução *m+1* será o menor número do registro permitido para uso. Substitua cada comando do programa por instruções MIR seguindo a tabela 7.4.2.
- Passo 4 - Como a função de saída da máquina MIR é $s_s(y_0,...,y_s,y_{s+1},...) = (y_1,...,y_s)$, devemos adicionar instruções para transferir os valores das variáveis $y_1,...,y_s$ que aparecem no programa, para os registros 1,2,... , respectivamente e então zerar todos os outros registros.

O programa acima fica em L_{mir}

1 *limpe(0)* 2
2 $_1T$ 5, lim *pe*(0) 3
3 $_2D$ 4
4 $_1T$ 4.1, $_1E^{-1}$ 2
4.1 limpe(0) 2
5 *limpe(0)* 6
6 *copie(1,2)* 7
7 *limpe(2)* 8
8 *Pare*

7.5 -**Máquina com um único registro (MUR)**

A máquina com um único registro (MUR) em ∑ é uma MIR com um único registro. Para que possamos operar em MUR é necessário adicionar ao alfabeto de MUR um símbolo extra que funcionará como um separador de palavras. Desta forma, funções em ∑ serão computadas por MUR em $\sum \cup \{\$\}$.

Assuma agora que $\$ \notin \Sigma$. Seja $\Sigma' = \Sigma \cup \{\$\}$ e $\mathcal{W}' = (\Sigma')^*$. O símbolo \$ atuará como um separador entre palavras em $\mathcal{W} = \Sigma^*$. A função de entrada para MUR em Σ' é $e_r : W^r \to W'$ dada por

$$e_r(x_1,...,x_r) = x_1\$x_2\$...x_r\$$$

A função saída $s_s : W' \to W^s$ é dada por

$$s_s(y_1\$y_2\$...\$y_t\$u) = \begin{cases} (y_1, y_2, ..., y_s) & se \quad s \le t \\ (y_1, y_2, ..., y_t, 0, 0, ...) & se \quad s > t \end{cases}$$

onde $y_1,...,y_t,u \in W$.

Note que $s_s(y) = (part(y,\$,0)...(part(y,\$,s-1))$

Como ilustração, a função $f(x_1,x_2) = (x_2,x_1)$ em Σ_2 é computada por

1 $E_\$^{-1}$ 3, E_1^{-1} 2.1, E_2^{-1} 2.2

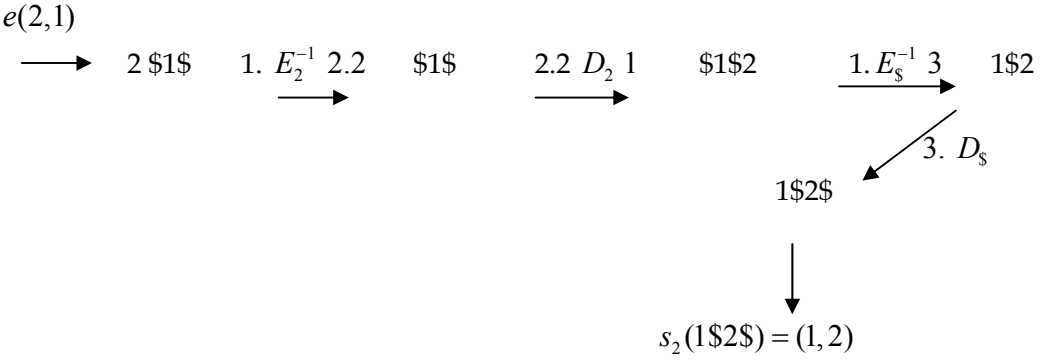
2.1 D_1 1

2.2 D_2 1

3 $D_\$$

4 *Pare*

Vamos executar o programa acima para a entrada (2,1). Então



Se o separador (\$) é adicionado ao alfabeto, uma MUR pode computar qualquer função que pode ser computada por MIR. Entretanto, o símbolo adicional \$ não é necessário desde que o alfabeto original contenha pelo menos 2 caracteres. Isto pode ser feito mudando as funções de entrada e saída e codificando palavras de $\mathcal{W}$ em palavras de $\mathcal{W}_1$, usando a segunda letra do alfabeto como símbolo separador. A máquina MUR de

um símbolo não pode computar todas as funções parciais recursivas com uma entrada e uma saída. Por exemplo, não é possível construir um programa $\mathcal{P}$ em MUR com alfabeto $\sum_1$ tal que $|\mathcal{P}|(1^k) = (1^{2k})$ o que contrasta com a MIR.
Os teoremas que se seguem, mostram que as máquinas com um número ilimitado de registros possuem o mesmo poder computacional das máquinas com um único registro, exceto para alfabetos com um único símbolo. Introduziremos algumas Macros que facilitarão as provas dos teoremas dados a seguir.

a) ***deslocat***- A macro *deslocat* transforma a palavra $x_0\$...x_m\$$ em $x_1\$...x_m\$x_0$

l *deslocat* l' $, \sigma \in \sum$

l L_s^{-1} l', L_σ^{-1} $l.\sigma$

$l.\sigma$ R_σ l

b) ***deslocat$*** -** A macro $deslocat_\$$ transforma $x_0\$x_1\$...x_m$ em $x_1\$...x_m\$x_0\$$

l $deslocat_\$$ l'

l *deslocat* l.1

l.1 $D_\$$ l'

c) ***deslocat$(k)*** –** Transforma $x_0\$...x_k\$...x_m\$$ em $x_k\$...x_m\$x_0\$...x_{k-1}\$$, permitindo a modificação de x_k apropriadamente.

l $deslocat_\$(k)$ l'

$1 \le k$

l $deslocat_\$$

l.1 $deslocat_\$$

$\vdots$

$l.k$ $deslocat_\$$ l'

7.5.1- **Teorema**

Se $\mathcal{P}$ é um programa MIR em $\sum$ e m é maior que o maior registro referenciado em $\mathcal{P}$, então: existe um programa $\mathcal{P}'$ em MIR com alfabeto $\sum'$, tal que $|\mathcal{P}|(0, x_1, ..., x_m, 0, ...) = (y_0, y_1, ..., y_m, 0, 0...)$ se e somente se $|\mathcal{P}'|(\$x_1\$...x_m\$) = (y_0\$y_1\$...y_m\$)$. Além do mais, se $\mathcal{P}$ é completo então $\mathcal{P}'$ também o será.

Prova

A representação da configuração MIR dada por $(x_0, x_1, \ldots x_n, 0, 0, \ldots)$ é $(x_0 \$ x_1 \$ \ldots x_n \$)$ em MUR. Podemos observar que existem (sempre) *m+1* \$´s no registro da MUR, acarretando que $deslocat_{\$}(m+1)$ não tem efeito no conteúdo do registro. Assim, $deslocat_{\$}(m+1+k)$ será usado ao invés de $deslocat_{\$}(k)$ para abranger o caso em que *k=0.*

Sejam os rótulos de $\mathcal{P}'$ os rótulos de $\mathcal{P}$ mais os novos rótulos necessários para construir macros adicionais. Sejam os rótulo inicial e terminal de $\mathcal{P}'$ os mesmos rótulos inicial e terminal de $\mathcal{P}$. Para cada linha de $\mathcal{P}$ inclua em $\mathcal{P}'$ a linha apropriada descrita a seguir

MIR	tabela 7.5.1	MUR
$l \quad {}_k D_\sigma \quad l'$		$l \quad deslocat_{\$}(m+1+k)$ $l.1 \; deslocat$ $l.2 \; D_\sigma$ $l.3 \; D_{\$}$ $l.4 \; deslocat_{\$}(m-k) \; l'$
$l \quad {}_k E_\sigma^{-1} \quad l'$		$l \quad deslocat_{\$}(m+1-k)$ $l.1 \quad E_\sigma^{-1} \; l.2.\sigma$ $l.2.\sigma \; deslocat_{\$}(m+1-k) \quad l'$
$l \quad {}_k T \quad l'$		$l \quad deslocat_{\$}(m+1+k)$ $l.1 \; E_{\$}^{-1} \; l.2, \quad T \quad l$ $l.2 \; D_{\$}$ $l.3 \; deslocat_{\$}(m-k) \quad l'$

O programa resultante $\mathcal{P}'$ simula $\mathcal{P}$, como requer o teorema. Não é difícil mostrar que $\mathcal{P}'$ é completo se $\mathcal{P}$ o for.

7.5.2- **Teorema**

Toda função $f: W^r \to W^s$ computada por um programa completo L_{MIR} em Σ é computada por um programa completo L_{MUR} em $\Sigma' = \{\Sigma \cup \$\}$.

Prova

Seja f uma função computada por um programa completo $\mathcal{P}$. Seja $m = max\{p,r,s\}+1$, onde p é o maior registro referenciado em $\mathcal{P}$. Seja $\mathcal{P}'$ a MUR construída usando o teorema anterior e seja $\mathcal{P}''$ o programa consistindo de $\mathcal{P}'$ ao qual são adicionados, no inicio, os seguintes comandos:

$$1 \quad D_{\$}$$

$$2 \quad D_{\$}$$

$$\vdots$$

$$m-r+1 \quad R_{\$}$$

$$m-r+2 \quad deslocat_{\$}(m)$$

Além disso, todos os comandos de *Pare* são substituídos por $deslocat_{\$}(m+1)\ l$, onde l é um novo rótulo, e os comandos

$$l \quad E_{\$}^{-1}\ l', \quad T\ \ l, \quad E_{\sigma}^{-1}\ \ l$$

$$l' \quad pare$$

são adicionados no final do programa.
Então $f(x_1,x_2,...,x_r) = (y_1,y_2,...,y_s)$ se e somente se $|\mathcal{P}|(0,x_1,...x_r,0,0,...) = (y_0,y_1,...,y_s,y_{s+1},...)$ para algum $y_0, y_{s+1},...$ se e somente se $|\mathcal{P}'|(\$x_1\$...x_r\$\$...\$) = (y_0\$y_1\$...y_s\$y_{s+1},\$...y_m\$)$ se e somente se $|\mathcal{P}''| = (x_1\$...x_r\$) = (y_1,\$...y_s\$y_{s+1}\$...y_m\$)$ se e somente se $f = s_s \circ |\mathcal{P}''| \circ e_r$.

Desde que o programa MIR é completo, o programa MUR também o será. O procedimento dado para construir $\mathcal{P}''$ pode especificar que certos comandos MUR devem ser incluídos mais de uma vez. Por exemplo, se $\mathcal{P}$ contém os comandos

$$7 \quad {}_5E_1^{-1}\ \ 9, \quad {}_5E_2^{-1}\ \ 6$$

então $\mathcal{P}'$ conterá o conjunto de comandos:

$$7 \quad deslocat_{\$}(m+1+5)$$

$$7.1 \quad E_1^{-1}\ \ 7.2.1$$

$$7.2.1 \quad deslocat_{\$}(m+1-5)\ \ 9$$

$$7 \quad deslocat_{\$}(m+1+5)$$

$$7.1 \quad E_2^{-1}\ \ 7.2.2$$

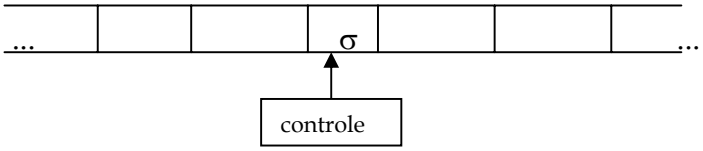
$$7.2.2 \quad deslocat_{\$}(m+1-5)\ \ 6$$

que é idêntico ao conjunto de comandos:

$$\begin{cases} 7 \quad deslocat(m+1+5) \\ 7.1\ E_1^{-1}\ 7.2.1, \quad E_2^{-1}\ 7.2.2 \\ 7.2.1\ deslocat(m+1-5) \\ 7.2.2\ deslocat(m+1-5) \end{cases}$$

7.6 - **Máquinas de Turing**

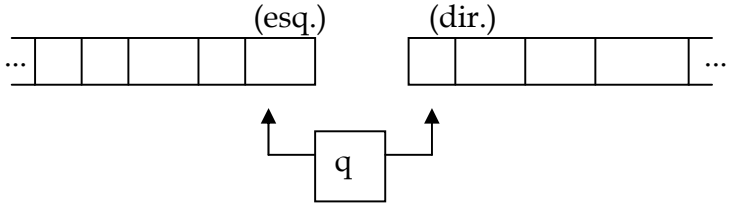
As máquinas MIR e MUR permitem o acesso aos símbolos de dados somente à esquerda e ao final da direita de palavras armazenadas nos registros. É natural imaginar máquinas que permitam o acesso local a qualquer símbolo armazenado. Um possível modelo é considerar máquinas que utilizem fitas ao invés de registros. Uma tal máquina poderia mover os símbolos, para frente e para trás, examinando-os e possivelmente modificando-os a cada instante sob a cabeça de leitura e gravação, como na figura abaixo



Este tipo de máquina foi proposta por Alan Turing em 1930. Pela sua simplicidade ela tem sido uma importante formalização da noção de computabilidade efetiva e tem jogado um papel central no desenvolvimento da teoria dos algoritmos.

Tradicionalmente, uma MT é descrita como um conjunto de quíntuplas da forma *(q,σ,σ´, M,q´)* onde *q* e *q´*são chamados *estados* da máquina e que substitui os rótulos em nossa definição para MIR e MUR. O símbolo $\mathcal{M}$ representa um movimento que pode ser *D* ou *E* ou algumas vezes () isto é, direita ou esquerda ou parada. O símbolo σ representa o símbolo que está armazenado e está sendo lido pela máquina e σ´ é para ser escrito em seu lugar. Como um exemplo, a quíntupla *(q,a,b,D ,q´)* indica que se a máquina está no estado *q* e *a* é o símbolo que está sendo lido, apontado em um instante *t*, na fita então substitua *a* pelo símbolo *b*, mova para a direita e vá para o estado q´.

Nesta seção, vamos enquadrar a máquina de Turing dentro da definição formal de máquinas dada anteriormente e em seções posteriores, voltaremos com a definição tradicional. Assim sendo, a fita pode ser considerada como logicamente equivalente a dois registros onde o símbolo na extremidade de cada registro é acessível como mostrado na figura abaixo



O conteúdo da fita é, portanto, a concatenação das palavras armazenadas nos dois registros (só podemos acessar a extremidade esquerda do registrador a direita e somente a extremidade direita do registrador esquerdo).

7.6.1 - **Definição**

Uma máquina de Turing (MT) é uma máquina com dois registros ($C = W^2$ **)** com o conjunto de instruções

$$\mathcal{I} = \{i \times (E_\tau \circ E_\sigma^{-1}), (D_\tau \circ D_\sigma^{-1}) \times i, D_\sigma^{-1} \times E_\sigma,$$

$$D_\sigma \times E_\sigma^{-1}, T \times E_\sigma, T \times E_\sigma^{-1}, D_\sigma \times T, D_\sigma^{-1} \times T, T \times i, i \times T\}$$

onde os índices $\sigma, \tau \in \sum$.

Observando as instruções do conjunto $\mathcal{I}$, que modificam o conteúdo dos registros, vemos que o termo à esquerda de cada instrução opera sobre a extremidade direita do registrador esquerdo e vice-versa.

As funções de entrada e saída para a MT são

$$e_r(x_1, ..., x_r) = (0, x_1\$x_2\$...x_r\$)$$

e

$$s_s(x, y_1\$y_2\$...\$y_t\$z) = \begin{cases} (y_1, ..., y_s) & \text{se } s \leq t \\ (y_1, ..., y_t, 0...0) & \text{se } s > t \end{cases}$$

onde $z \in \sum^*$ e $y_i \in \sum^*$ para *i* = 1,...,t. A entrada é codificada com $´s entre argumentos e colocada no registro 1 (registro da direita), do qual a saída é também tomada. Assim um programa $\mathcal{P}$ em L_{MT} no alfabeto $\sum \cup \{\$\}$ computa $f : W^r \to W^s$ se $|\mathcal{P}|(0, x_1\$...x_r\$) = (w, y)$ para algum w e y tal que $s_s(w, y) = (y_1, y_2, ..., y_s) = f(x_1, ..., x_r)$.

Vários tipos de máquinas de Turing já foram estudadas. No entanto, com respeito à classe de funções que elas podem computar, cada uma delas tem sido mostrada equivalente ao modelo apresentado acima. Nosso modelo tem uma única fita ilimitada nas extremidades, isto é, uma fita que pode ser estendida arbitrariamente tanto para esquerda quanto para direita. Entretanto, podemos mostrar que as máquinas de Turing com *n* fitas *n>1* são equivalentes as máquinas com uma única fita (Hopcroft[26]).

As macros abaixo serão convenientes

l *deslocaesq* l' $\quad$ l $D_\sigma \times E_\sigma^{-1}$ l' $\quad \sigma \in \sum$

l *deslocadir* l' $\quad$ l $D_\sigma \times E_\sigma^{-1}$ l' $\quad \sigma \in \sum$

A macro *deslocaesq* desloca um símbolo do final do registro 1 para à direita do registro 0. Isto é equivalente a tradicional máquina de Turing, deslocando em sua fita um quadrado (registro) para a direita.

7.6.2 –**Exemplo**

a) A seguinte máquina de Turing em Σ_3' computa a função sucessor em Σ_3.

1 *deslocaesq* 1, $i \times T$ 2

2 *deslocadir* (*desloca* \$)

3 $(D_2 \circ D_1^{-1}) \times i$ 5, $(D_3 \circ D_2^{-1}) \times i$ 5, $(D_1 \circ D_3^{-1}) \times i$ 4, $T \times E_1$ 6

4 *deslocadir* 3

5 *deslocadir* 5, $T \times i$ 6

6 *Pare*

b) A seguinte máquina de Turing em Σ_2' computa a concatenação em Σ_2 deletando o \$ entre os argumentos.

1 $D_1 \times E_1^{-1}$ *l*, $D_2 \times E_2^{-1}$ *l*, $D_\$ \times E_\$^{-1}$ 2

2 *deslocadir*

3 $(D_1 \circ D_1^{-1}) \times i$ 4.1, $(D_2 \circ D_2^{-1}) \times i$ 4.2, $T \times E_\$^{-1}$ 6

4.1 $i \times (E_1 \circ E_\$^{-1})$ 5

4.2 $i \times (E_2 \circ E_\$^{-1})$ 5

5 $(D_\$ \circ D_1^{-1}) \times i$ 2, $(D_\$ \circ D_2^{-1}) \times i$ 2

6 *Pare*

7.6.3 –**Teorema**

Para cada programa P_{MUR} no alfabeto Σ, existe um programa P_{MT}' no alfabeto Σ tal que $|P_{MUR}|(x) = (y)$ se e somente se $|P_{MT}'|(0,x) = (0,y)$.

Prova

Sejam os rótulos de P_{MT}' aqueles de P_{MUR} mais os que são necessários, pois, muitas vezes, um único comando de P_{MUR} será traduzido em vários comandos de P_{MT}'. Os rótulos inicial e final de P_{MT}' são os mesmos rótulos inicial e final de P_{MUR}. Assim, para cada comando de P_{MUR}, sejam os comandos abaixo:

MUR	tabela 7.6.1	MT
$l \;\; L_\sigma^{-1} \;\; l'$		$l \;\; \xi \times L_\sigma^{-1} \;\; l'$
$l \;\; R_\sigma \;\; l'$		$l \;\; deslocaesq \;\; l, \;\; R_\sigma \times \xi \;\; l.1$
		$l.1 \;\; deslocadir \;\; l.1, \;\; \xi \times i \;\; l'$
$l \;\; \xi \;\; l'$		$l \;\; i \times \xi \;\; l'$

A prova estará completa notando que o registro 0 é vazio antes e depois que cada instrução MUR é simulada.

7.6.4–**Teorema**

Cada função $f : W^r \to W^s$ computada em $\sum$ por um programa completo MUR em $\sum' = \sum \cup \{\$\}$ é também computada por um programa completo MT em $\sum'$.

Os exemplos acima mostram a complexidade dos programas MT mesmo para aquelas funções muito simples. Entretanto as MT desempenham um papel central no desenvolvimento da Teoria da Computação e isto é ilustrado no texto de Davis (1958) e também por uma grande quantidade de trabalhos dedicados ao estudo deste modelo. As máquinas MUR e MIR foram introduzidas por Sheperdson e Sturgis [13]. Eles provaram a equivalência das MUR e MIR-funções computáveis com as funções parciais recursivas. Das máquinas apresentadas nesta seção, as MIR são as que mais se aproximam das máquinas reais conhecidas como máquinas RAM(Random Access Machine).
No próximo item, apresentaremos máquinas com poder computacional menor que as MT mas que desempenham um importante papel em ciência da computação e na teoria das linguagens formais.

7.7- Conjuntos aceitos por máquinas

Vimos que um programa pode representar um conjunto, computando a sua função característica. Existe um caminho mais geral pelo qual um programa pode determinar um conjunto. Um programa de máquina $\mathcal{P}$ pára para uma entrada x se $x \in dom|\mathcal{P}|$, isto é, quando o programa começa em um estado inicial com x no registro da máquina , existe alguma computação que chega ao estado terminal. Para as funções de entrada e

saída $e_r : W^r \to X$ e $s : X \to W^0$, o programa $\mathcal{P}$ é dito aceitar ou reconhecer (ou definir) o conjunto ou a linguagem $\mathcal{L}(\mathcal{P}) = \{x \in W^r \mid x \in dom(s \circ |\mathcal{P}| \circ e_r)\}$.

Uma possível função de saída é a função total $s : X \to W^0$ que tem o valor constante (), isto é, o único elemento de W^0. Neste caso $\mathcal{L}(\mathcal{P}) = \{x \in W^r \mid x \in dom(|\mathcal{P}| \circ e_r)\}$. Esta função saída será usada nas discussões que se seguem.

O programa determinístico mas não completo MUR em $\sum_1'$

1 $E_{\$}^{-1}$ 3, E_1^{-1} 2

2 E_1^{-1} 1

3 *Pare*

pára sobre todas as palavras em $(\sum_1')^* = \{\$,1\}^*$ da forma $(11)^k \$x, 0 \le k$. Para r =1, $\mathcal{L}(\mathcal{P})$ = (11)*, para r = 2, $\mathcal{L}(\mathcal{P})$ = (11)*× W_1.

O programa MIR em $\sum_2$`

1 ${}_1E_1^{-1}$ 2.1, ${}_1E_2^{-1}$ 2.2, ${}_1T$ 3

2.1 ${}_2E_1^{-1}$ 1

2.2 ${}_2E_2^{-1}$ 1

3 ${}_2T$ 4

4 *Pare*

pára para todas as entradas $(x_1, x_2, ...)$ tal que $x_1 = x_2$. Logo, para r = 2, $\mathcal{L}(\mathcal{P})$ = $\{(x, y) \mid x = y\}$ e para r = 1, $\mathcal{L}(\mathcal{P})$ = {0}.

Os conjuntos que são aceitos por MIR são denominados *conjuntos recursivamente enumeráveis* (vistos na seção 6).

Existem vários tipos de máquinas que têm poder computacional menor que MIR,MUR e MT e que têm sido estudadas extensivamente. As três mais comuns são: autômato limitado linearmente, autômato de pilha e autômato finito.

Um autômato limitado linearmente é uma máquina com 2 registros com o conjunto de instruções

$$\mathcal{R} = \{i \times E_\sigma^{-1}, D_\sigma \times E_\tau^{-1}, D_\sigma^{-1} \times E_\tau, i \times T, T \times i\}$$

Tais máquinas reconhecem as linguagens denominadas **sensível ao contexto.**

O autômato limitado linearmente é tradicionalmente descrito como uma máquina de uma única fita . A máquina pode deslocar a fita para trás e para frente mudando e apagando símbolos. A função de entrada é a mesma que a da MT.

Um autômato de pilha é uma máquina com dois registros e com o seguinte conjunto de instruções

$$\mathcal{R} = \{i \times E_\sigma^{-1}, D_\sigma \times i, D_\sigma^{-1} \times i, i \times T, T \times i\}$$

As linguagens reconhecidas pelos autômatos de pilha são denominadas **livres de contexto.**
O registro esquerdo é denominado *push-down stack*. A máquina pode escrever ou apagar símbolos na pilha , mas pode somente ler da fita de entrada (o registro direito). A função de entrada é a mesma das MT. A noção de um *push- down stack* tem jsido exaustivamente utilizada na descrição de tradutores para as linguagens de programação. O seguinte autômato de pilha aceita o conjunto de seqüências balanceadas de parênteses.

1 $i \times E_{\$}^{-1}$ 4, $i \times E_{(}^{-1}$ 2, $i \times E_{)}^{-1}$ 3

2 $D_{(} \times i$ 1

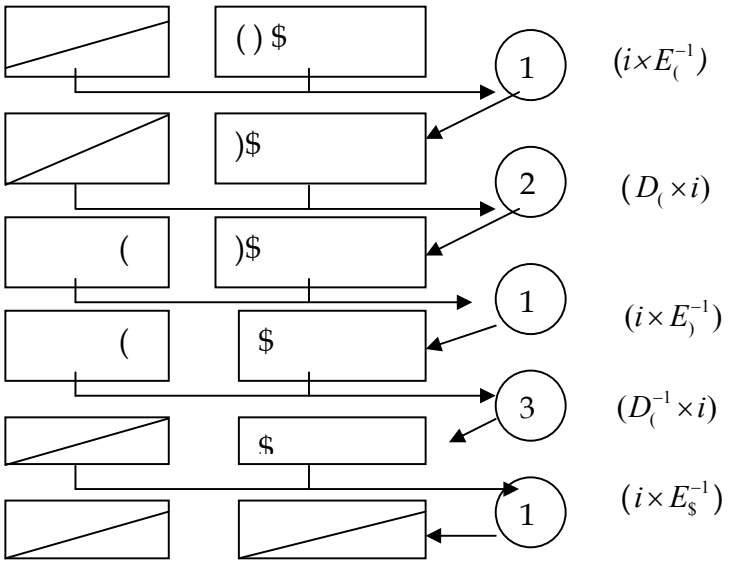
3 $D_{(}^{-1} \times i$ 1

4 $T \times i$ 5

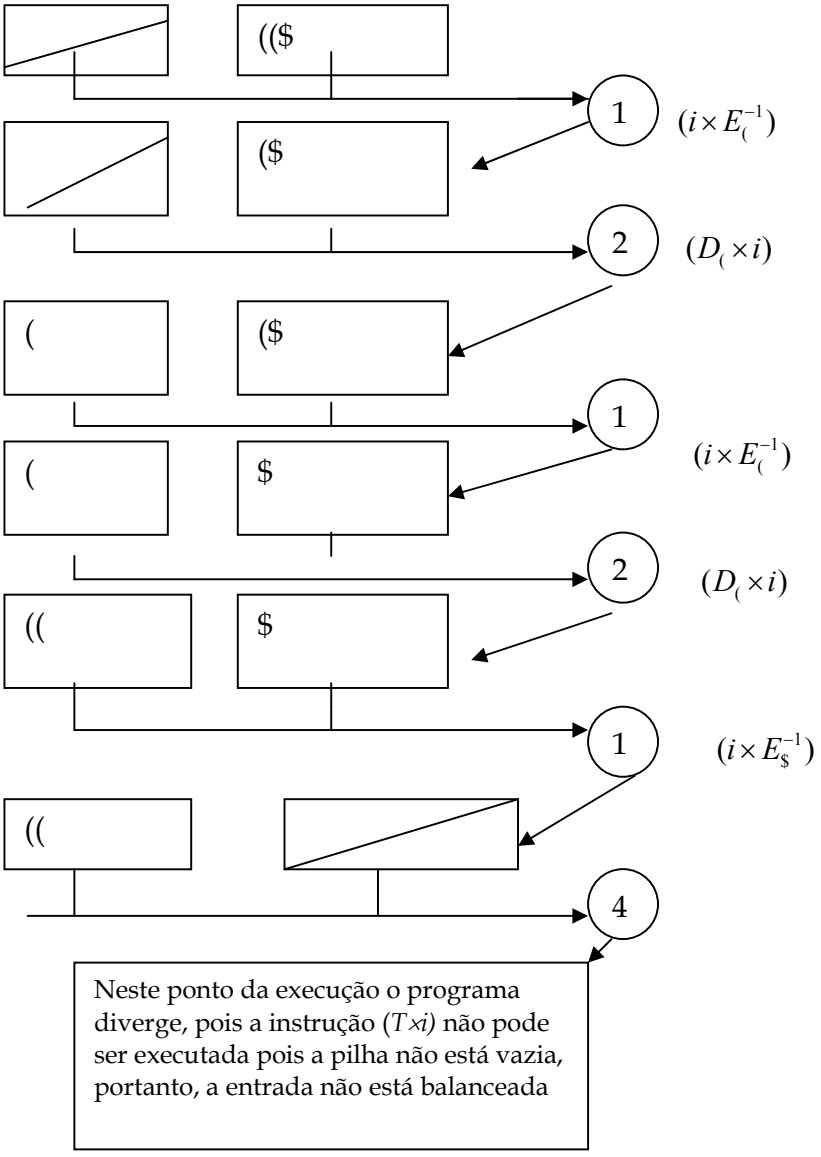
5 *Pare*

O programa opera da seguinte forma: se um abrir parêntese '(' é encontrado no registro de entrada de dados, coloque-o na pilha (registro da esquerda): se um fechar parêntese ')' está mais à esquerda no registro de entrada , então ele é retirado do registro da direita e também será desempilhado um abrir parêntese do registro da esquerda . Assim o par de parênteses é aceito e o processo continua até que a pilha fique vazia, ou a seqüência de parênteses não é aceita. Vamos simular dois exemplos:

a) Para a entrada balanceada: ()



b) Para a entrada desbalanceada: ((



Um autômato finito é uma máquina de um registro com instruções $\{E_{\sigma}^{-1}\}$. A função de entrada é $e_r(x_1,...x_r) = x_1\$...x_r\$$. Os autômatos finitos reconhecem as ***linguagens regulares.***

Um autômato finito que aceita o conjunto $\varnothing$ é o programa com nenhum comando , ou qualquer programa com nenhum rótulo inicial ou terminal. Um programa não determinístico aceitando o conjunto $0 \cup (1^* \cup (12)^*)2$ é

1.1 E_1^{-1} 1.1, E_2^{-1} 3

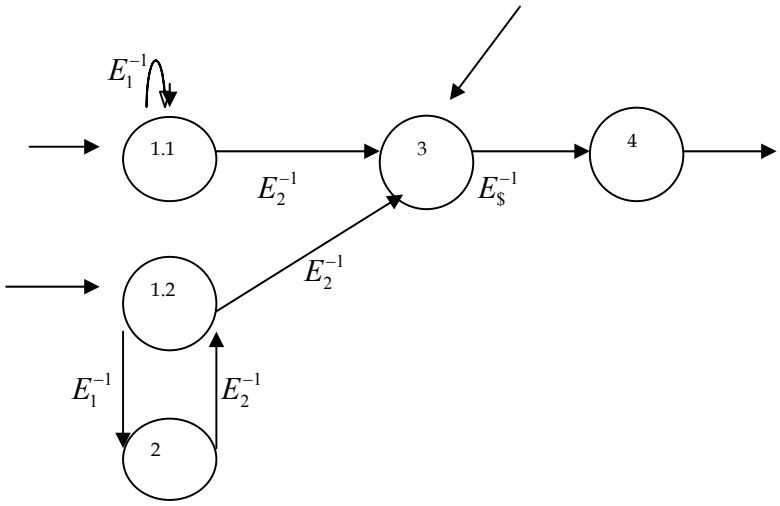
1.2 E_1^{-1} 2, E_2^{-1} 3

2 E_2^{-1} 1.2

3 $E_\$^{-1}$ 4

4 *Pare*

Onde 1.1,1.2 e 3 são rótulos iniciais. A representação gráfica deste programa é dada pela figura abaixo.



- *Autômato que reconhece a linguagem* $0 \cup (1^* \cup (12)^*)2$

7.7.1 - **Teorema**

S é um conjunto recursivamente enumerável se e somente se *S* é aceito por um programa LP, um programa MIR, um programa MUR e por um programa MT.

Prova

Usando qualquer função saída total, o conjunto aceito por um programa é o $dom\ \varphi_m$, onde φ_m é a função computada pelo programa. O teorema segue do corolário 6.1.4 e a equivalência dos modelos LP,MIR,MUR e MT.

Nas próximas seções, estudaremos alguns sistemas formais que são relevantes em ciência da computação

Seção 8

8.1 **Um breve histórico sobre sistemas formais**

Leibnitz, em torno de 1700, contemplava a idéia de desenvolver uma linguagem para a matemática que fosse única e universal. No entanto, as investidas nesta direção resultaram infrutíferas até meados do século XIX. Realmente, Frege em 1879, foi quem primeiro desenvolveu uma linguagem formal e lógica para a matemática no sentido moderno. A noção básica de sistemas formais foi introduzida por Frege em uma breve monografia de 80 páginas denominada Begriffsschrift e que é reconhecida como a maior obra de lógica escrita depois de Aristóteles(Putnam[1.a]).

Antes dos trabalhos de Frege não existia uma formulação rigorosa para os sistemas axiomáticos, nem uma definição para *prova* e tampouco um modo consistente, tanto quanto sabemos, de desenvolver a teoria dos conjuntos e, portanto, de formalizar toda a matemática, incluindo a análise, topologia e etc...

O programa de Frege, assim como o de Peano (1888), (que realizou importantes contribuições ao desenvolvimento lógico da teoria dos números), era o de construir uma notação simbólica e um conjunto de regras que fossem adequadas para todas as demonstrações dedutivas- e, em particular, para a análise das demonstrações matemáticas.

A Begriffsschrift continha essencialmente o sistema lógico. A análise completa da matemática teria de esperar pela publicação do primeiro volume dos Grundgesetze Arithmetik [1893-1903] e, antes que o segundo volume fosse publicado, dez anos mais tarde, Russell já havia descoberto uma contradição (conhecida como o 'paradoxo de Russell') na construção de Frege.

Russell e Whitehead[1905], publicaram os Principia Mathemática, tornando a lógica de Frege mais acessível e introduzindo uma nova teoria dos conjuntos denominada 'teoria dos tipos' (TP), e que tinha o objetivo de evitar a ocorrência de certas definições impredicativas em teorias matemáticas. Neste trabalho Russell e Whitehead apresentam um sistema axiomático para a matemática, o qual foi posteriormente simplificado por Ramsey em 1926.

Na mesma época em que Frege desenvolveu seu trabalho, Cantor criou a 'teoria dos conjuntos'. Russell publicou sua teoria dos tipos seguindo a linha de Frege e paralelamente, Zermelo formalizou a teoria dos conjuntos de Cantor. A teoria dos conjuntos apresentada por Zermelo foi aperfeiçoada juntamente com Fraenkel e resultou na tão conhecida teoria dos conjuntos de Zermelo-Fraenkel (ZF) que é bastante usada na matemática atual.

Foi Hilbert, em torno de 1900, que era um seguidor de Cantor, quem originariamente formulou a disciplina da metamatemática, realizando seus estudos de sistemas formais através de métodos da teoria dos números, e com o objetivo de apresentar uma absoluta prova de consistência de um sistema no qual toda a matemática poderia ser deduzida. Isto significa que, dado um sistema tal como ZF ou TP, não pode ser possível derivar destes sistemas enunciados do tipo ' p **e** ¬p'. Para provar a inconsistência de um

sistema, basta exibir uma dedução de uma contradição. No entanto, para provar a consistência de um sistema, temos que provar que é impossível que qualquer contradição seja derivada do sistema em questão. Assim, **consistência** é uma noção da teoria da prova que era desenvolvida por Hilbert. Ele acreditava que a matemática não só poderia ser totalmente formalizada, mas também que a noção matemática e informal de dedução, (isto é, a noção implícita quando se fala em deduzir um teorema do conjunto de axiomas de um certo ramo da matemática), seria corretamente formalizada pela lógica de primeira ordem (a lógica de primeira ordem é a teoria que resulta da álgebra Booleana juntando-lhe os quantificadores universal e existencial). Dizer que a noção de dedução é formalizada na lógica de primeira ordem é dizer que qualquer demonstração em matemática informal pode, após adequada reconstrução, ser representada na lógica de primeira ordem. Isto significa que cada passo da dedução pode ser justificado por uma das regras da lógica. Tais regras são tão simples que podem ser programadas em uma máquina de Turing.(Os computadores modernos já aceitam linguagens cujo paradigma é a lógica de primeira ordem, como é o caso da linguagem Prolog [Kowalski,1983] que trabalha essencialmente por dedução formal. Na verdade a linguagem Prolog é um provador mecânico de teoremas). O projeto de Hilbert ficou conhecido como ´Formalismo´ e ganhou ímpeto a partir de 1920 com a contribuição de Ackermann, von Neumann e outros. Em 1900 Hilbert havia provado a consistência interna da geometria elementar e a prova da consistência da teoria elementar dos números (dada pelos axiomas de Peano), para a análise real, e para ZF era o objetivo da linha formalista.
Nesta época, Löwenhein, um matemático que trabalhava com teoria de modelos, e que não seguia a linha formalista, coloca a seguinte questão:

Existirão fórmulas da lógica de primeira ordem que possam ser satisfeitas num universo não-enumerável, mas não num universo infinito enumerável?
Ele provou que a resposta para esta questão é negativa. Para Löwenhein , que era um seguidor de Peirce, o universo de discurso da lógica de primeira ordem era *relativo*, isto é, dependente do discurso que é formalizado. Este resultado de Löwenhein é um resultado da teoria pura dos modelos. Ele pressupõe o conceito de interpretação, mas não o conceito de dedução, ou melhor, regra de inferência.
Utilizando a teoria da demonstração de Hilbert e a teoria dos modelos de Löwenhein, Gödel formulou e respondeu afirmativamente à seguinte questão:

´Será que todo conjunto consistente de fórmulas possui um modelo?
Com esta prova, Gödel estabelecia uma conexão entre a teoria dos modelos e a teoria da demonstração. Em outras palavras, o que este resultado nos diz é: a noção sintática de 'consistência' coincide com a noção semântica de *satisfação* da teoria dos modelos. Uma outra noção importante introduzida por Gödel é a de *sistema completo* que é a seguinte: uma fórmula é valida, isto é, verdadeira em todas as interpretações em todos os universos não vazios, se e somente se, puder ser demonstrada no sistema. Gödel

demonstrou que a lógica de primeira ordem é completa (mas não é decidivel!). Depois de obter estas respostas viria a questão principal:

Serão os Principia Mathematica completos?

Esta pergunta pode ser traduzida da seguinte forma: será possível *demonstrar*, no âmbito dos Principia, todo enunciado que seja *verdadeiro* no quadro da interpretação dos Principia que se tem em mente?

A resposta foi negativa e ela é conhecida como o **teorema de incompletude de Gödel**. E para além dos Principia, Gödel demonstrou que *nenhum* sistema consistente e completo, suficientemente forte para exprimir a teoria elementar dos números, pode possuir um conjunto recursivo de axiomas. Como vimos anteriormente, os conjuntos recursivos são os únicos conjuntos que permitem um procedimento de decisão, daí um conjunto não recursivo não seria uma formalização aceitável. No decurso dessas investigações Gödel foi levado a elaborar a noção de função recursiva geral e em 1936, Church percebeu serem equivalentes todas as propostas para definir o conceito de calculabilidade ou computabilidade e enunciou aquilo que viria a ser conhecido como a **tese de Church**: a tese de que recursividade coincide com computabilidade. Embora sejam equivalentes, cada abordagem tem sua utilidade prática na ciência da computação. As máquinas de Turing (seção 7) juntamente com o teorema da forma normal de Kleene (seção 4), por exemplo, permitiu a von Neumann elaborar um modelo para construção de máquinas digitais, e que são utilizadas até hoje como padrão para projetos de computadores. Os sistemas de Post juntamente com a abordagem linguística de Chomsky permitem, por exemplo, a elaboração de técnicas para tradução automática das linguagens de programação de alto nível para as linguagens de máquinas, tão utilizadas em compilação.

Vimos que os sistemas formais foram idealizados com o objetivo de ´mecanizar´ a prova de teoremas em matemática, e o agente considerado, para executar as deduções, era imaginado que fosse um ser humano com bons conhecimentos em matemática. Atualmente, os agentes para os quais os sistemas formais são construídos são também , ou talvez principalmente, computadores.

Tudo aquilo que os computadores são capazes de realizar consiste na execução de algoritmos. É importante notar que cada programa de computador destinado a resolver um problema, inicialmente formulado em linguagem corrente, é uma formalização do problema e de certos processos de raciocínio utilizados para o resolver. A abordagem lógica, e as linguagens formalizadas para a lógica, foram e têm sido indispensáveis para o desenvolvimento de *software* para computadores. A linguagem usada para escrever programas é ela própria *literalmente* formalizada. Mas para além desta linguagem, existem programas, em áreas tão diversas como reconhecimento de padrão, tradução automática, processamento de linguagem natural e etc... que exigem que a máquina - e não apenas o programador- *raciocine* numa linguagem formalizada.

Com este breve histórico, podemos observar que a noção bem antiga de formalização tem sido modificada, ampliada e tem se tornado bastante importante e útil nos tempos atuais, com o advento dos computadores. Por outro lado, esperamos também que tenha

ficado claro que a ciência da computação tem seus fundamentos oriundos dos fundamentos da matemática, ou seja, torna-se impossível desvincular ciência da computação da matemática. A tese de Church, embora possa ser refutada no futuro, é uma conseqüência de pesquisas sérias de grandes matemáticos e pensadores. E finalizamos este breve histórico, sob um ponto de vista filosófico, citando Rezende [53]:
"Em suas várias interfaces com outras ciências, a Computação tem aberto passagem, principalmente através da lógica matemática, para que certas questões filosóficas perenes ressurjam em novas formas e possam ser abordadas por novos ângulos, como por exemplo, a relação empírica entre o problema da representação do mundo levantado por Kant e o conceito de inteligência. Numa digressão sobre o possível conceito finitário de *inteligência* , Turing nos convida, já em 1936, à reflexão sobre esta relação. O problema filosófico mais dramaticamente ligado à teoria da computação talvez seja ainda aquele em sua gênesis, pois a abordagem à questão do infinito oculta ou revela detalhes de outros problemas que em torno dela gravitam..."
Na próxima seção, desenvolveremos alguns sistemas formais denominados gerativos e que têm aplicações relevantes em ciência da computação. Inicialmente consideraremos os sistemas de produção de Post, que são métodos para construir linguagens recursivamente enumeráveis, obtidos a partir de um conjunto finito de regras que geram conjuntos infinitos de sentenças, tais sistemas não computam funções, eles geram teoremas (sentenças de uma linguagem) por dedução.
Os elementos básicos destes sistemas são: um *alfabeto* finitoΣ, uma *linguagem* recursiva L (cujos elementos são formados a partir de Σ^*), um conjunto recursivo de *axiomas* e um conjunto recursivo de *regras* (regras de inferência). Os axiomas são asserções, (elementos de L), que aceitamos ou simplesmente acreditamos serem verdadeiras, as regras de inferências permite realizar *deduções*, isto é, nos diz precisamente como podemos obter novas asserções, chamadas teoremas, dos axiomas ou de teoremas *deduzidos* previamente.
Terminaremos esta seção com uma definição bem geral para os sistemas formais ilustrando-a com alguns exemplos.

8.1.2 –**Definição**

Um sistema formal é uma quádrupla $F = < \Sigma, L, R, P >$ onde
Σ - é o alfabeto de F
L - a linguagem de F construída a partir de Σ
R - é um conjunto de relações onde cada elemento $r \in R$ é denominado regra de de inferência de F.
P - *é o conjunto de axiomas de* F

A definição acima é mais geral do que realmente necessitamos. Consideraremos que o alfabeto é um conjunto enumerável de símbolos e em geral será finito. Os conjuntos Σ, L, e R são conjuntos recursivos. Desta forma, é sempre possível decidir para qualquer *n-pla*

$< x_1, x_2, ..., x_n >$ de elementos de L determinar se $< x_1, x_2, ..., x_n >$ é uma relação de F, e o conjunto R de regras será finito.
Associado com qualquer sistema formal existe uma estrutura dedutiva como descrita abaixo.

8.1.3 – **Definição**

Dado um sistema formal F e um conjunto $X \subset L$, dizemos que $y \in L$ é *diretamente derivável* (deduzida,inferida) do conjunto X se existe $r \in R$ e uma seqüência $b_1, b_2, ..., b_{n-1}$ de elementos de X tal que $< b_1, b_2, ..., b_{n-1}, y > \in r$.

8.1.3.1 – **Definição**

Dado um conjunto $X \subset L$ dizemos que $y \in X$ é *derivável* do conjunto X de hipóteses se existe uma seqüência finita $b_1, b_2, ..., b_{n-1}, b_n$ onde

a) cada b_i, $1 \leq i \leq n$, ou é um axioma ou é diretamente derivável do conjunto cujos elementos precedem b_i na seqüência
b) b_n é y

A seqüência finita $b_1, b_2, ..., b_{n-1}, b_n$ é denominada uma prova formal (ou dedução formal) do conjunto de hipóteses X. Se X é vazio (assim, somente os axiomas e as regras de inferência são usados na dedução) então a seqüência $b_1, b_2, ..., b_{n-1} b_n$ é apenas denominada prova em F. Neste caso y é dito ser um teorema de F.

8.1.4 – **Exemplos de sistemas formais**:

a) Seja F tal que $\sum$ = {a,b,c}, $L = \sum^*$, P = {a,b,c,aa,bb,cc} e
$R = \{ r_1 =< x, axa >, r_2 =< x, bxb >, r_3 =< x, cxc > \}$
então cabac é um teorema de F pois

$$(*) \quad \begin{cases} 1.\ b \in P \quad (axioma) \\ 2.\ aba \quad (< b, aba > \quad r_1) \\ 3.\ cabac (< aba, cabac > \quad r_3) \end{cases}$$

ou

(**) $b \Rightarrow^{r_1} aba \Rightarrow^{r_3} cabac$

Podemos observar que este sistema deduz todas as palavras palindromos construídas a partir de $\sum$. Usaremos os formatos (*) ou (**) para representar uma dedução.

b) Cálculo proposicional

$\sum$ - é o alfabeto $\{p_i \mid i \geq 1\} \cup \{\Rightarrow, \neg, (,)\}$

L_p - é uma linguagem consistindo de fórmulas definidas recursivamente como abaixo:

1) para todo i, p_i é uma fórmula
2) se α é uma fórmula então ¬α é uma fórmula
3) se α e β são fórmulas então (α⇒β) é uma fórmula
4) as únicas fórmulas são as expressões satisfazendo os critérios 1,2 e 3 acima.

A - é o seguinte conjunto dos axiomas

$$Ax1 - (\alpha \Rightarrow (\beta \Rightarrow \alpha))$$
$$Ax2 - ((\alpha \Rightarrow (\beta \Rightarrow \delta)) \Rightarrow ((\alpha \Rightarrow \beta) \Rightarrow (\alpha \Rightarrow \delta)))$$
$$Ax3 - ((\neg\alpha \Rightarrow \neg\beta) \Rightarrow (\beta \Rightarrow \alpha))$$

R – é o seguinte conjunto de regras de inferências

$$\{((\alpha, \alpha \Rightarrow \beta), \beta)\}$$

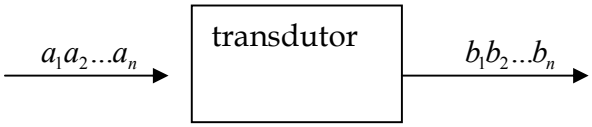
É conveniente observar que as letras α, β e δ não fazem parte do sistema formal, são variáveis denominadas sintáticas, utilizadas como auxiliares para a descrição do sistema formal.

A regra deste sistema é conhecida como *modus ponens*. Este sistema quando interpretado como uma lógica bivalente ele é uma forma da álgebra Booleana, isto significa que, suas fórmulas podem assumir valores no conjunto {0,1} (significando verdadeiro ou falso). Este sistema é completo no sentido de Gödel e também decidível. Isto significa, como vimos, que existe um procedimento de decisão para avaliar a veracidade ou falsidade de uma fórmula do sistema. O método para tal avaliação é o tão conhecido método da tabela verdade introduzido por Wittgenstein e (independentemente também por Post) em torno de 1920. Putnam[Putnam,1.a] comenta que quando Boole apresentou a álgebra, forneceu um procedimento de decisão . Já Peirce, contudo, quando deu à lógica de primeira ordem o nome pelo qual hoje é conhecida, "omitiu" a apresentação deste procedimento de decisão (embora pareça que estava convencido que não era difícil conseguir um). Analogamente, Frege, Russell e Whitehead, apresentaram axiomas e regras de inferências, mas não um procedimento de decisão para a lógica de primeira ordem. Gödel mostrou porque isto aconteceu.

Seção 9

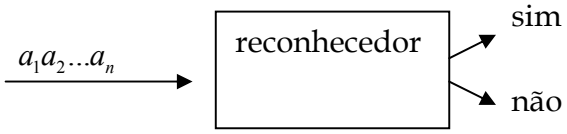
9.0 – **Sistemas de Produção e Linguagens Formais**

Nesta seção, focalizaremos os sistemas de manipulação simbólica, que são semelhantes a regras de um jogo. Podemos classificá-los em três tipos: transdutores, reconhecedores (ou aceitadores) e geradores.
Um *transdutor* é um sistema que recebe seqüências de símbolos na entrada e as transforma em seqüências de símbolos na saída



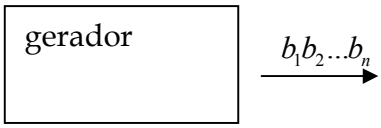
Por exemplo, um somador seqüencial recebe uma seqüência de pares de bits e a transforma em uma seqüência de bits, representando sua soma.

Um *reconhecedor* (ou aceitador) é um sistema que recebe uma seqüência na entrada e a classifica como possuindo ou não uma determinada propriedade.



Na seção 7-7.7 foram apresentados três tipos de máquinas que são classificadas como reconhecedores.
Os estudos dos reconhecedores estão ligados a Teoria dos Autômatos que sofreu forte influência tanto da construção de equipamentos eletrônicos digitais, como computadores, quanto pela tentativa de modelar certos aspectos do sistema nervoso, como transmissão de informação entre neurônios (redes neurais).

Um *gerador* é um sistema que produz uma seqüência de símbolos na saída, isto é, produz cadeias somente a partir dos axiomas.



Como exemplo temos os sistemas dados em 8.1.4 .a.

Estes três tipos de modelos são maneiras diferentes de se usar uma mesma idéia básica: definições recursivas para dar uma descrição finita para objetos possivelmente infinitos.

9.1 –**Sistemas de produção de Post(SPP)**

Os sistemas de produção foram primeiro desenvolvidos por Thue[50] para transformar cadeias de símbolos. Eles foram usados como base para o desenvolvimento computacional dos sistemas de Post[51] e Markov[52]. Foram adaptados por Chomsky[34.1] para descrição formal de gramáticas e paralelamente por Backus para desenvolver a linguagem FORTRAN. Derivações destes sistemas são amplamente usadas em várias áreas, tais como: lingüística, inteligência artificial, construção de compiladores, ´datamining´, bioinformática e outras.
Embora a aplicação destes sistemas seja bastante vasta, nos limitaremos ao processamento de palavras. Nossa apresentação será baseada principalmente em Brainerd&Landweber[4].

Na nossa terminologia, uma linguagem nada mais é do que um subconjunto de $W=\Sigma^*$, o conjunto de palavras sobre qualquer alfabeto finito Σ. Temos um interesse particular nas linguagens recursivamente enumeráveis, pois tais linguagens têm a propriedade de serem geradas ou reconhecidas por procedimentos que são efetivamente computáveis. As linguagens de programação pertencem ao subconjunto das linguagens recursivas denominadas linguagens sensíveis ao contexto.

9.1.1 – **Definição**

Um sistema de produção de Post (SPP) no alfabeto V_t é uma quádrupla

$S = < V_t , V_n , R, P>$, onde:

1- V_n é um alfabeto finito contendo V_t; os elementos de V_t são denominados símbolos terminais e os elementos de $V_n - V_t$ são chamados símbolos não terminais ou auxiliares.
2- $P \subseteq V_n^*$ é um conjunto finito de axiomas ou palavras de partida; quando P contém um único elemento não terminal, este símbolo também será denotado por P.
3- R é um conjunto finito de regras de reescrita ou produção da forma

$$x_0 | \sigma_1 | x_1 | \sigma_2 | \ldots | \sigma_k | x_k \rightarrow y_0 | \alpha_1 | \ldots | \alpha_{k'} | y_{k'}$$

onde cada x e cada y é elemento de V_n^*. Os σ_i, α_j, com $1 \leq i,j \leq n$ são números naturais tais que cada número α_j do lado direito da regra é um σ_i do lado esquerdo da regra.

Para um SPP S e palavras x e y em Φ* , x⇒y, ou y é ***diretamente derivável*** de x, significa que existe um caminho para preencher as caixas de alguma regra R com palavras de V_n*. Então as caixas com os mesmos números recebem palavras idênticas e o resultado é x do lado esquerdo da regra e y do lado direito. Por exemplo, para:
V_n={a,b}, abaabaaa⇒abaab (abaab é derivável de abaabaaa) usando a regra ab|1|b|1|a|2|⇒|2|ab|1|b, onde aa é colocado na caixa 1 e 0 é colocado na caixa |2|. Várias palavras podem ser derivadas de uma única palavra x, mesmo se apenas uma regra é usada.
Se $x = w_0 \Rightarrow w_1 \Rightarrow ... \Rightarrow w_m = y$, então y é ***derivável*** de x e escrevemos $x \Rightarrow^* y$ e $w_0 \Rightarrow w_1 \Rightarrow ... \Rightarrow w_m$ é uma derivação de comprimento m de y a partir de x. Note que $x \Rightarrow^* x$ para cada x por uma derivação de comprimento 0.

9.1.2- **Exemplo**

Sejam V_t={1}, $V_n = V_t \cup${a,b}, P = {ab1} e
R ={<a|1|b|2|, a|2|b|1| |2|>, <a|1|b|2|, |1|>}
Este sistema gera os números de Fibonacci em $\sum_1$. Vamos mostrar que ab1⇒*111 (111 é derivável do axioma ab1)

1.<ab1,a1b1> (o axioma ab1 dispara a regra1, com|1|=0 e|2|=1, produzindo a1b1)
2.<a1b1,a1b11> (regra1, com|1|=1 e |2|= 1)
3.<a1b11,a11b111> (regra1, com |1|=1 e|2|= 11)
4. <a11b111,a111b1111> (regra1, com |1|=11 e |2|=111)
5. <a111b1111,111> (regra2, com |1|=111 e |2|=0)

ou

$$ab1 \Rightarrow^{r1} a1b1 \Rightarrow^{r1} a1b11 \Rightarrow^{r1} a11b111 \Rightarrow^{r1} a111b1111 \Rightarrow^{r2} 111$$

9.1.2- **Linguagem Gerada por SPP**

Seja S um SPP e $|S|=\{x \in V_n^* | s \Rightarrow^* x$, para algum $s \in P\}$ o conjunto de todas as seqüências derivadas de axiomas de S. A linguagem gerada por S $\mathcal{L}(S)=|S| \cap V_t$ *. No exemplo anterior a linguagem gerada é o conjunto dos números de Fibonacci em $\sum_1$. A linguagem gerada por um sistema de Post contém todas as palavras consistindo apenas de símbolos de V_t.

9.1.3 – **Teorema**

Para cada SPP S , $\mathcal{L}(S)$ é recursivamente enumerável.

Prova

Podemos observar que o conjunto de derivações de S é primitivo recursivo. Seja $d_r(x,y)$ o predicado que estabelece que y é diretamente derivável de x usando a regra r, suponhamos a regra

$$x_0 \mid 1 \mid x_1 \mid 2 \mid x_2 \mid 2 \mid x_3 \Rightarrow y_0 \mid 2 \mid y_1 \mid 2 \mid y_2$$

então

$$d_r(x,y) = \vee_{u_1=0}^{x} \vee_{u_2=0}^{x} [x = x_0 u_1 x_1 u_2 x_2 u_2 x_3 \wedge y = y_0 u_2 y_1 u_2 y_2]$$

As funções $\vee_{u_1=0}^{x} \vee_{u_2=0}^{x}$ e $\wedge$ foram definidas no módulo1 seção1-1.19 e 1.22. Assim o predicado $d_r(x,y)$ assegura que y é diretamente derivável de x. Seja $V=V_n \cup \{\Rightarrow\}$. Se uma palavra z é da forma $w_0 \Rightarrow w_1 \Rightarrow ... \Rightarrow w_m \in V^*$, onde cada w é um elemento de V^*, então o número de ocorrências de $\Rightarrow$ em z é m e $part(z \cap \Rightarrow, \Rightarrow, k) = w_k$, $0 \leq k \leq m$. Assim,

$$deriv(z) = \left(\wedge_{k=0}^{ocorr(z,\Rightarrow)\ -1} (part(z \cap \Rightarrow, \Rightarrow, k) \Rightarrow part(z \cap \Rightarrow, \Rightarrow, k+1) \right) \vee [ocorr(z, \Rightarrow) = 0]$$

assegura que z é uma derivação em S e pela construção acima ela é primitiva recursiva.

Um método de enumerar elementos de $\mathcal{L}(S)$ é examinar sucessivamente cada palavra de V^* para certificar-se de que ela é uma derivação. Se z é uma derivação, w_0 é um axioma, e w_m, a ultima palavra da derivação depois de $\Rightarrow$, contém apenas símbolos terminal, então é só imprimir w_m.

Suponhamos que $A = \mathcal{L}(S)$, então

$$\chi_A = \vee_z [deriv(z) \wedge part(z \cap \Rightarrow, \Rightarrow, 0) \in P \wedge part(z \cap \Rightarrow, \Rightarrow, ocorr(z, \Rightarrow)) = x]$$

daí $\mathcal{L}(S)$ é r.e. A função $part(z \cap \Rightarrow, \Rightarrow, 0) \in P$, verifica se a cadeia obtida através de *part* é um axioma de S, pois, vimos que o conjunto de axiomas de S é recursivo, logo decidível.

9.2- **Gramáticas**

Um SPP S é denominado *gramática* se e somente se toda regra de S é da forma

$$|1|x|2| \Rightarrow |1|y|2|$$

9.2.1-**Exemplo**

Seja S tal que $V_n = V_t = \{(,)\}$ $P=\{0\}$ e $R=\{<|1||2|,|1|()|2|\}$. Então S é uma gramática e a linguagem gerada por S $\mathcal{L}(S)=\{(()),(),((()))),...\}$

As gramáticas refletem o caminho através do qual sentenças complexas de uma linguagem são construídas a partir de certas unidades básicas e de acordo com as regras

gramaticais. Uma gramática pode ser pensada não somente como uma definição intencional de uma linguagem, mas também como a descrição de uma máquina abstrata que é capaz de aceitar (gerar) aquelas sentenças que pertencem à linguagem. A complexidade do potencial destas maquinas é diretamente determinada pela complexidade interna das regras das gramáticas que a elas correspondem.

9.3 –<u>Gramáticas e Conjuntos Recursivamente Enumeráveis.</u>

Vimos que os conjuntos r.e. são aceitos por máquinas de Turing. Neste item mostraremos que:

a) Os conjuntos r.e. podem ser gerados por algum SPP.
b) Todo conjunto que é aceito por uma MT não determinística é aceito por uma MT determinística.

9.3.1 – <u>Teorema</u>

Para todo programa P na linguagem MT (seção 7) não determinística, uma gramática G pode ser construída tal que a linguagem gerada $L(G)$ seja um subconjunto Ψ aceito por P. (O que devemos mostrar é: $\forall x(x \in L(G)$ sss $P(x)\downarrow)$).

Para isso construiremos uma gramática que simula os passos da computação de P via MT em ordem inversa, isto é, as regras de G simulam movimentos para trás de MT. Assim , a gramática gerará uma palavra x se e somente se o programa na MT aceitar x.

Prova
Seja P um programa em MT e I o conjunto de rótulos de P, então P é construído pelo seguinte conjunto de instruções:

$$\{l\ i\times(E_\tau\circ E_\sigma^{-1})\ l',l\ (D_\tau\circ D_\sigma^{-1})\times i\ l',l\ D_\sigma^{-1}\times E_\sigma l',l\ D_\sigma\times E_\sigma^{-1}\ l',l\ T\times E_\sigma\ l',$$
$$l\ T\times E_\sigma^{-1}\ l',l\ D_\sigma\times T\ l',l\ D_\sigma^{-1}\times T\ l',l\ T\times i\ l',l\ i\times T\ l'\}$$

Seja G a seguinte gramática:
$V_n=\sum\cup\{\$\}\cup I\cup\{J,T,[,]\}$, $\sum'=\sum\cup\{\$\}$ o alfabeto de MT.
Axioma: [T]
Regras:

$$T\Rightarrow\sigma T,\sigma\in\sum\cup\{\$\}$$
$$T\Rightarrow T\sigma,\sigma\in\sum\cup\{\$\}$$
$$T\Rightarrow l$$

onde l é um rótulo terminal.
São exemplos de cadeias geradas por estas regras, supondo $\sum$ ={a,b,$}

1.$[T] \Rightarrow [\$T]$

2.$[T] \Rightarrow^{*} [\$\$\$\$T]$

3.$[T] \Rightarrow^{*} [a\$b\$l\$b\$b]$

4.$[T] \Rightarrow [aT] \Rightarrow [aTb] \Rightarrow [alb]$

Uma derivação em *G* deve começar com [*T*] e alcançar alguma cadeia no formato [*u l´v]* (como nos exemplos .3 e.4 descritos acima), onde $u,v \in (\Sigma')^*$, observando que a palavra nula não está excluída. Neste ponto, a derivação começa a aplicar as regras (tabela 9.1) que simulam os passos da computação de *P* na ordem inversa. Se a MT tem *u* e *v* em seus registros e executa o comando com rótulo *l*, então no instante seguinte executará o próximo comando com configuração *u´* e *v´* nos registros. Desta forma [*ulv*] é derivável em um passo de [*u´l´v´*](ordem inversa). Para a derivação terminar com $x \in \Sigma^*$, o ultimo passo deve estar no formato $[lx\$ \Rightarrow Jx\$] \Rightarrow ... \Rightarrow xJ\$ \Rightarrow x$ onde *l* é um rótulo inicial. Assim, $x \in L(G)$ se e somente se para algum *u* e $v \in (\Sigma')^*$, algum rótulo inicial *l*, e algum rótulo terminal *l´*, $[ul'v] \Rightarrow [lx\$]$. Isto acontecerá se e somente $|P|(0,x\$) = (u,v)$ isto é, o programa *P* pára quando a computação começa com 0 em um registro e $\alpha_1(x) = x\$$ no outro registro.Para completar a prova, segue a tabela que traduz comandos de *P* em regras de *G*.

P *tabela 9.1*	*G*
$l\ i \times (E_\tau \circ E_\sigma^{-1})\ l'$	$l'\tau \Rightarrow l\sigma$
$l\ (D_\tau \circ D_\sigma^{-1}) \times i\ l'$	$\tau l' \Rightarrow \sigma l$
$l\ D_\sigma^{-1} \times E_\sigma\ l'$	$l'\sigma \Rightarrow \sigma l$
$l\ D_\sigma \times E_\sigma^{-1}\ l'$	$\sigma l' \Rightarrow l\sigma$
$l\ T \times E_\sigma\ l'$	$[l'\sigma \Rightarrow [l$
$l\ T \times E_\sigma^{-1}\ l'$	$[l' \Rightarrow [l\sigma$
$l\ D_\sigma \times T\ l'$	$\sigma l'] \Rightarrow l]$
$l\ D_\sigma^{-1} \times T\ l'$	$l'] \Rightarrow \sigma l]$
$l\ T \times i\ l'$	$[l' \Rightarrow [l$
$l\ i \times T\ l'$	$l'] \Rightarrow l]$

E ainda :

$[l \Rightarrow J$

$J\sigma \Rightarrow \sigma J$

$J\$ \Rightarrow 0$

9.3.2 –**Teorema**

Um conjunto $A \subseteq \mathcal{W}$ é r.e. se e somente se *A* = *L(S)* para algum SPP.

Prova

Segue dos teoremas 9.1.3 e 9.3.1

9.3.3 –**Teorema**

Todo conjunto que é aceito por um programa em uma MT(MIR,MUR) não determinística é também aceito por uma máquina determinística MT (MIR,MUR).

Prova

Este resultado segue dos teoremas 9.3.1 e 9.3.2. Note que as simulações de MIR e MUR em MT são válidas para as máquinas não determinísticas.

9.3.4- **Teorema**

Se $A = L(S)$ para algum SPP, então uma gramática $G´$ pode ser construída tal que $A = L(G´)$.

Prova

Segue dos teoremas 9.1.3, 9.3.1, 9.3.2 e 9.3.3.

9.4 –Linguagens Formais

A teoria das linguagens formais é uma disciplina da ciência da computação que surgiu dos estudos das linguagens naturais, mais especificamente, dos trabalhos do lingüista Noam Chomsky.

Na publicação pioneira de Chomsky, uma linguagem é vista como consistindo de um alfabeto Σ e um subconjunto $\mathcal{L} \subseteq \Sigma^*$. Em 1956, Chomsky[34.1] apresentou o conceito de gramáticas gerativas e em 1959[34.2] classificou as linguagens em 4 tipos: sem restrição, sensível ao contexto, livre de contexto e lineares (regulares). Com isso, indicou um caminho para definir famílias de linguagens, isto é, por gramáticas.

Paralelamente, Kleene [35] se dedicava ao estudo de redes neurais no sentido Macculloch-Pitts, e obteve como resultado a caracterização de conjuntos regulares em termos da união, concatenação e fecho (fecho de Kleene).

Myhill em 1957 [36] associou os conjuntos regulares com um tipo de máquina muito simples, hoje conhecido como autômatos finitos .

Chomsky em 1959[34.2] mencionou a equivalência das linguagens lineares (regulares) com os conjuntos regulares de Kleene, e então, conectando este resultado com os trabalhos de Myhill, ficou estabelecido uma nova forma de definir as linguagens formais através de máquinas abstratas denominadas aceitadores (reconhecedores).

Em 1960, uma comissão internacional de cientistas construiu uma linguagem de programação denominada ALGOL-60 [33]. Para os teóricos de linguagens, a importância da ALGOL-60 era a sua descrição através do formalismo denominado BNF[33], pois, pela primeira vez, a sintaxe de uma linguagem de programação era totalmente formalizada.

Pouco tempo depois que a ALGOL-60 foi publicada, Ginsburg e Rice[37] apresentaram um trabalho e mostraram que o formalismo BNF era equivalente às gramáticas livres de contexto. Com este resultado, a teoria das linguagens formais tornou-se objeto de estudo

de vários grupos interessados em desenvolver linguagens de programação. A conexão entre as linguagens livres de contexto e BNF é a razão mais importante desta disciplina pertencer à ciência da computação.
Na verdade, como é colocado por Galernter[38]: 'ALGOL-60 foi o pontapé inicial de uma verdadeira febre de projetos de novas linguagens que gerou uma serie de linguagens como C, C++, ADA e PASCAL'.
Além da especificação em BNF, duas grandes idéias estão presentes na ALGOL-60: a estrutura recursiva (que é o desenvolvimento isolado mais importante na história do software) e a idéia de sub-rotina, que é a base para a técnica de programação denominada "programação orientada à objetos", descrita inicialmente em 1967 pelos projetistas da linguagem Simula 67[39]. A ALGOL-60 já havia encontrado a idéia de ambiente independente e fornecido uma operação criadora de ambiente que é a chamada à procedimentos. A linguagem Simula precisou apenas de um segundo tipo de chamada: aquela que cria um ambiente que se prolonga indefinidamente [38].
Voltando ao nosso tema, as **linguagens regulares** e **livres de contexto** são muito úteis e importantes em ciência da computação e tem aplicação direta em analisadores sintáticos (*parser*) em construção de compiladores. Em bioinformática, são aplicadas através das expressões regulares e linguagens estocásticas, em classificações de seqüências de nucleotídeos. Por outro lado, uma boa parte das linguagens naturais pode ser especificada formalmente via estes tipos de linguagens, portanto tem aplicação direta em construção de sistemas para interfaces amigáveis (em linguagem natural), em tradução automática, em reconhecimento de voz e etc.

9.4.1- **Tipos de Gramáticas**

Vimos que um SPP cujas regras obedecem ao formato

$$|1|x|2| \Rightarrow |1|y|2|$$

são definidos como gramáticas. Chomsky fazendo certas restrições sobre a natureza de x e y hierarquizou as gramáticas em 4 tipos. E cada tipo está associado um tipo de máquina ou reconhecedor. De modo a tornar mais simples os estudos de algumas das propriedades destes sistemas, vamos introduzir a hierarquia de Chomsky reformulando a definição de gramática.

9.4.2- **Definição**

Uma gramática é uma quádrupla $G = < V_n, V_t, A, P >$ onde:

V_n - símbolos de variáveis

V_t - símbolos terminais

A - axiomas, símbolos de partida

P- regras de re-escrita ou produções

Assumimos que $V_n \cap V_t = \varnothing$, $V_n \cup V_t = V$

O conjunto de produções ***P*** consiste de regras no formato Post, isto é,

$$x \rightarrow y$$

onde $x \in V^*$ e $y \in V*$.
Vamos definir o *vocabulário* de G, como sendo *V*, o alfabeto composto pelos símbolos não terminais e terminais. O conjunto ***P*** é uma relação binária no conjunto *V** de cadeias de símbolos quaisquer, isto é, ***P*** $\subseteq$ *V** x *V**, correspondendo cada regra individual a um par de cadeias (*x, y*). Reunimos regras com o mesmo lado esquerdo *x*, tais como $x \rightarrow y_1, x \rightarrow y_2, ..., x \rightarrow y_n$, na abreviação $x \rightarrow y_1 \mid y_2 \mid ... \mid y_n$.
Reservaremos as letras maiúsculas para representarem as variáveis e as minúsculas para representarem os símbolos terminais.

9.4.3-**<u>Hierarquia de Chomsky</u>**

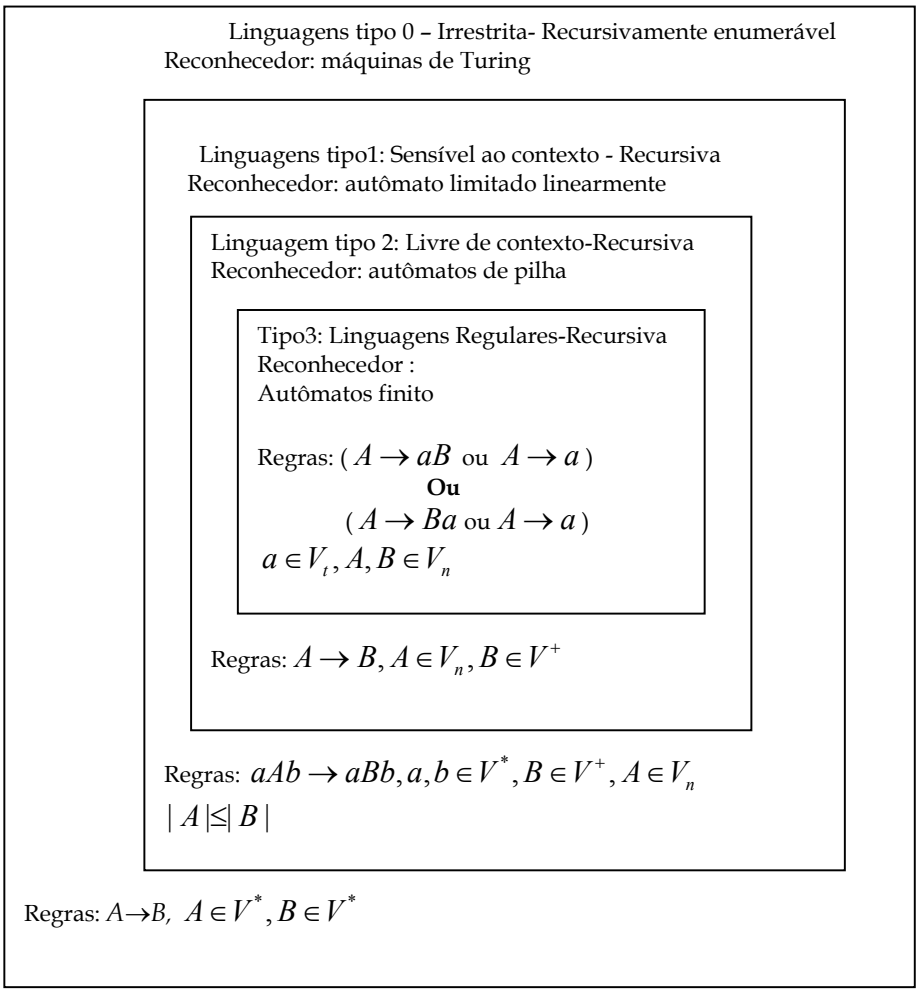
A hierarquia é dada da seguinte forma:

- *Tipo 0* – é a gramática cujas produções são definidas no formato da definição 9.4.1.
- *Tipo 1* – é a gramática cujas produções são da forma
  $$x \rightarrow y$$
  com $|y| \geq |x|$ ($|\alpha|$ significa o número de símbolos que ocorrem em α)
  ou também, $aAb \rightarrow aBb$ com $a,b,B \in V*$, $B \neq 0$ e A em V_n.
- *Tipo 2*- é a gramática cujas produções são da forma
  $$A \rightarrow B$$
  onde *A* é formada por uma única variável, isto é, $A \in V_n$, $B \in V^+$.
- *Tipo 3* – é a gramática cujas produções obedecem a apenas um dos formatos:
  $(A \rightarrow aB$ ou $A \rightarrow a)$ linear a direita
  ou $(A \rightarrow Ba$ ou $A \rightarrow a)$ linear a esquerda
  onde $a \in V_t$, $A, B \in V_n$.

Se analisarmos cuidadosamente as regras de cada tipo de gramática podemos mostrar que se L_i é uma linguagem gerada por uma gramática *tipo i*, $3 \leq i \leq 0$ então

$$L_3 \subset L_2 \subset L_1 \subset L_0$$

A hierarquia acima pode ser visualizada na figura que se segue:



Observações:
Estes são os tipos mais básicos de gramáticas, linguagens e reconhecedores apresentados na literatura. Esta hierarquia foi obtida pelo estabelecimento de restrições sobre regras gramaticais pré-definidas. Vimos que uma máquina aceitadora (reconhecedora) é aquela que dados uma linguagem L e uma sentença s, ela define se $s \in L$. O autômato limitado linearmente é uma máquina em que o comprimento da palavra não cresce, ou seja, o registrador tem sempre o tamanho da palavra de entrada. Recentemente surgiram linguagens que se enquadram entre os quatro tipos acima, principalmente entre os tipos 1 e 2 e entre os tipos 3 e 4.
O estudo das linguagens formais é basicamente o estudo da correspondência entre as linguagens e as máquinas.

9.4.3 –<u>Exemplos de linguagens</u>

1) **Livre de Contexto**
   a) $V_n = \{S, A, B\}, V_t = \{a, b\}$ e P o seguinte conjunto de regras

$1) S \rightarrow aB \quad 5) A \rightarrow bAA$

$2) S \rightarrow bA \quad 6) B \rightarrow b$

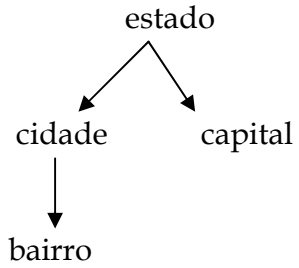
$3) A \rightarrow a \quad 7) B \rightarrow bS$

$4) A \rightarrow aS \quad 8) B \rightarrow aBB$

A linguagem gerada por G consiste de todas as palavras que tem a propriedade de possuir o mesmo número de a´s e b´s. Assim, abba, abab, aaaabbbbb, aabbaabb e etc...

b) Vamos considerar um exemplo onde a linguagem gerada por G é um subconjunto do Português cujas sentenças são estruturadas obedecendo a seguinte hierarquia:

(*)



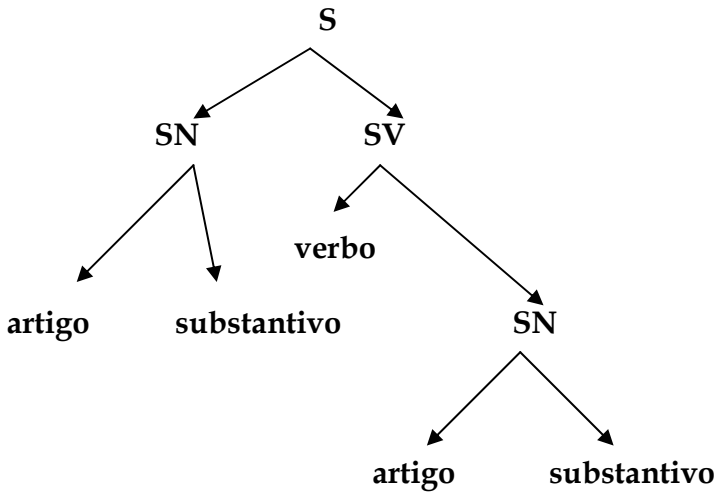
Por exemplo, as seguintes sentenças pertencerão a linguagem gerada por G :

a) Porto Alegre é a capital do Rio Grande do Sul
b) Niterói é uma cidade do Rio de Janeiro
c) Barra da Tijuca é um bairro do Rio de Janeiro

Para isso, vamos voltar ao Português e definir os sintagmas nominais e verbais. Representaremos por SN os sintagmas nominais e SV os sintagmas verbais. Observemos que na nossa gramática, estas variáveis estarão em V_n. Precisaremos de outras variáveis para representar os artigos definidos, indefinidos, verbos substantivos próprios e comuns e preposições. Sejam as variáveis em negrito que poderão ser substituídas por quaisquer elementos dos conjuntos a elas associados.

art >:{ o,a,um,uma} **sub>:**{capital, cidade, bairro}
np>: {Porto Alegre, Rio Grande do Sul, Ceará, Fortaleza .....}
prep>:{ do, de da} **ver>:** {é}

Estas variáveis em negrito funcionarão como representantes dos elementos de V_t, ou seja são símbolos terminais. Os elementos de V_n serão variáveis que representam as estruturas dos sintagmas verbais ou nominais, por exemplo, uma sentença em Português pode ser estruturada da seguinte forma:



Seja G = $< V_n, V_t, P, S >$ onde:

$V_n = \{S, SN_1, SV, SN\}$

$V_t = np \cup sub \cup art \cup prep \cup ver$

e P as seguintes produções

1) $S \rightarrow SN1 \ SV$

2) $SN1 \rightarrow np$

3) $SN \rightarrow art \ sub \ prep \ np$

4) $SN \rightarrow SN1$

5) $SV \rightarrow ver \ SN$

A gramática G gera as sentenças estruturadas como em (*), mas gera também sentenças que não são desejadas como:

(**) *Porto Alegre é um bairro de Salvador*

Podemos observar que este é um problema semântico, pois do ponto de vista sintático, a sentença dada em (**) está correta. Vamos tentar manter um nível de congruência entre as estruturas sintáticas e a estrutura semântica que se pretende descrever. Para isso, precisamos refinar as regras da gramática acima. Então vamos reestruturar nosso conjunto de dados da seguinte forma:

Capitais: **np1>:**{Porto Alegre, Fortaleza, Rio de Janeiro, Salvador...}
Bairros : **np2>:**{Icaraí, Itapuã, Barra da Tijuca, Ipanema...}
Estados: **np3>:**{Rio de Janeiro, Rio Grade do Sul, Bahia,...}
sub1>:{capital}, **sub2>:**{bairro}, **sub3>:**{estado} e **sub4>:**{cidade}
e as novas regras

$1) S' \rightarrow F$

$2) SN1 \rightarrow np1$

$3) SN2 \rightarrow np2$

$4) SN3 \rightarrow np3$

$5) Sn4 \rightarrow np4$

(***) $6) SN5 \rightarrow art\ sub1\ prep\ SN3$

$7) SN6 \rightarrow art\ sub2\ prep\ SN4$

$8) SN7 \rightarrow art\ sub4\ prep\ SN3$

$9) F \rightarrow SN1\ verSN5$

$10) F \rightarrow SN2\ ver\ SN6$

$11) F \rightarrow SN4\ ver\ SN7$

A nova gramática será G´= $< V_{n'}, V_{t'}, P', S' >$ onde:

$V_{n'} = \{S', F, SN1, SN2, SN3, SN4, SN5, SN6, SN7\}$

$V_{t'} = np1 \cup np2 \cup np3 \cup sub1 \cup sub2 \cup sub3 \cup sb4 \cup prep \cup ver$

e P´ são as regras dadas em (***)

A sentença (**) não é mais gerada e a hierarquia (*) é preservada. No entanto, esta gramática não resolve problemas do tipo:

Salvador é a capital do Maranhão

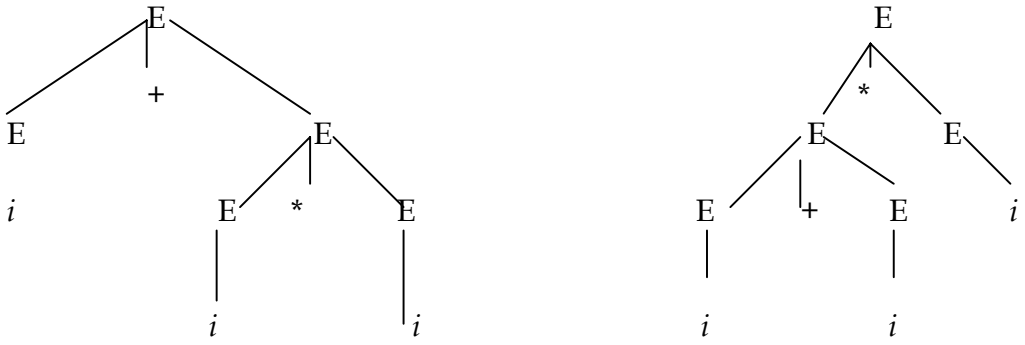
Este exemplo mostra como os aspectos semânticos de uma linguagem natural ou artificial podem ser difíceis de se capturar através de especificações por gramáticas. E este fato é uma das maiores críticas sofridas por Chomsky em suas teorias lingüísticas.

c) Vamos apresentar um outro tipo de problema muito comum em especificações de linguagens e que é denominado *ambigüidade sintática*, como abaixo:

$V_n = \{E\}, V_t = \{+, i, *\}$

$P = \{E \rightarrow E + E, E \rightarrow E * E, E \rightarrow i\}$

Podemos visualizar a geração de *i*+*i***i* pelas árvores



e neste caso dizemos que a gramática é ambígua, pois uma mesma sentença pode ser gerada por diferentes caminhos.
A árvore é a estrutura de dados associada a gramáticas livres de contexto, como assegura o seguinte teorema:

´Se G é uma gramática livre de contexto então para todo $\alpha \neq 0$, $S \rightarrow^* \alpha$ *se e somente se existe uma árvore de derivação na gramática G com resultado* α.

2) **Sensível ao Contexto**

a) $V_n = \{S, B, C\}, V_t = \{a, b, c\}$ e P as seguintes produções:

$1) S \rightarrow aSBC$
$2) S \rightarrow aBC$
$3) CB \rightarrow BC$
$4) aB \rightarrow ab$
$5) bB \rightarrow bb$
$6) bC \rightarrow bc$
$7) cC \rightarrow cc$

A linguagem gerada por G é $L(G) = \{a^n b^n c^n \mid n \geq 1\}$. Podemos observar que G é sensível ao contexto. É possível demonstrar que tal linguagem não pode ser especificada por nenhuma gramática livre de contexto.

3) **Regular**

a) Vamos especificar um subconjunto regular da língua Portuguesa.
De modo a facilitar nossa descrição, as categorias gramaticais (variáveis) serão baseadas em algumas categorias gramaticais tradicionais do Português, como no exemplo 9.4.3 – 1.b, conforme abaixo:

Artigos definidos, artigos indefinidos, substantivos, nomes próprios, adjetivos, verbos, preposições.
que serão representados respectivamente pelas variáveis:

art, sub, np, adj, v, prep

onde
art -representa os artigos podendo também aparecer sob a forma **arti** (indefinidos) ou **artd** (definidos)
sub- representa a classe dos substantivos
np representa a classe dos nomes próprios.
Adj -representa a classe dos adjetivos.

V- representa a classe dos verbos intransitivos.
SN0 – sintagmas nominais.
S – representando o símbolo inicial (axioma)
FV- representa os sintagmas verbais
AUX- verbo de ligação (auxiliar)
As regras de nossa gramática são:

$$S \rightarrow SN0\ V$$
$$S \rightarrow SN0\ aux\ sub$$
$$S \rightarrow SN0\ aux\ adj$$
$$SN0 \rightarrow np$$
$$SN0 \rightarrow sub$$
$$SN0 \rightarrow art\ np$$
$$SN0 \rightarrow art\ sub$$
$$SN0 \rightarrow art\ sub\ adj$$

A linguagem gerada por esta gramática é regular. É interessante observar que em *G* não ocorrem regras recursivas, isto é, regras do tipo $X \rightarrow Xa$ e também como os elementos associados às categorias gramaticais do Português é um conjunto finito então a linguagem gerada por ela é finita. Baseado nisto, o número de regras pôde ser otimizado de tal forma que os formatos da primeira e ultima regra não estão sob o formato exigido pela definição 9.4.2. Na próxima seção analisaremos algumas propriedades importantes das linguagens regulares.
As seguintes sentenças são exemplos daquelas que podem ser geradas por esta gramática:

1. *Lula é presidente.*
2 .O menino caiu.
3.O tempo está ruim

Monteiro[48] demonstrou que os subconjuntos do Português definidos através destas categorias gramaticais são regulares, muito embora na literatura eles sejam especificados, neste caso, via gramáticas livres de contexto. A grande vantagem em se especificar linguagens via gramáticas regulares está associada à característica da máquina de reconhecimento, que são, neste caso, mais simples e mais velozes, pois os autômatos finitos reconhecem as sentenças de uma linguagem em tempo real e não possuem memória fora de seu processador central fixo. O mesmo não ocorre com os reconhecedores para as linguagens de tipo 0, 1 ou 2 que necessitam de uma memória auxiliar para realizar a análise sintática de sentenças, conforme foi visto na seção 7. Nas linguagens de programação esta característica das linguagens regulares é muito importante, pois seus analisadores devem ser máquinas bastante velozes. Mas,

infelizmente, as gramáticas regulares não são suficientes para especificar totalmente tais linguagens. De fato, elas especificam apenas uma parte delas, como exemplo: encontrar a ocorrência de uma cadeia de símbolos dentro de outra e etc...

b) Sejam $V_n = \{S\}, V_t = \{0,1,2,3,4,5,6,7,8,9\}$ e P as produções

$S \to 0|1|2|3|4|5|6|7|8|9$

$S \to S0|S1|S2|.....|S9$, onde " | " significa "ou", ou seja, o número total de regras é dez ao todo.

Esta gramática gera os números naturais. Note que a regra **S→0** permite a geração de números com ocorrências de zeros à esquerda. Assim, $S \to^{*} 0109$ é gerado como se segue

$$S \to^{rg2} S9 \to^{rg2} S09 \to^{rg2} S109 \to^{rg1} 0190$$

Neste exemplo, temos a ocorrência de regras recursivas tal como $S \to S1$.

9.4.4 – **A palavra nula (λ)**

Na definição dada para as gramáticas, não foi permitido que a palavra nula λ pudesse pertencer às linguagens geradas pelas gramáticas tipo1,2 e 3. No entanto, as definições em questão podem ser estendidas com produções $S \to \lambda$, onde S é o símbolo de partida, desde que S não ocorra do lado direito de qualquer produção. Se isto ocorrer, é claro que a condição $|A| \leq |B|$ não é satisfeita e o sistema gerará sentenças menores que o comprimento da entrada. Esta cláusula é garantida pelos seguintes resultados (cuja prova pode ser encontrada em [26]):

9.4.4.1- **Teorema**

Para toda gramática sensível ao contexto G_1 existe uma gramática G_2 sensível ao contexto, gerando a mesma linguagem que G_1, para o qual o símbolo inicial de G_2 não ocorre do lado direito de qualquer produção de G_2. E ainda, se G_1 é tipo 2 ou tipo 3 então G_2 pode ser encontrada.

9.4.2 – **Teorema**

Se L é uma linguagem sensível ao contexto, livre de contexto ou regular então $L \cup \{\lambda\}$ e $L - \{\lambda\}$ é uma linguagem sensível ao contexto, livre de contexto ou regular, respectivamente.

9.5 – **Recursividade das gramáticas sensíveis ao contexto**

Vimos que um conjunto recursivo é aquele que pode ser definido por um algoritmo. Dizer que uma gramática é recursiva é afirmar a existência de um algoritmo que garante a decidibilidade da linguagem gerada pela gramática. Esta condição é crucial em analisadores sintáticos para as linguagens de programação.

Uma das principais motivações da definição de gramáticas sensíveis ao contexto em restringir o comprimento das cadeias geradas, é garantir a recursividade das linguagens geradas por estas gramáticas, isto é, a existência de algoritmos que indicam se uma certa cadeia pertence ou não a linguagem gerada. É o que garante o teorema que apresentaremos a seguir, cuja prova pode ser facilmente encontrada na literatura relacionada a teoria da computação. Seguiremos Hopcroft & Ullman[26].

9.5.1 –**Teorema**

Toda linguagem sensível ao contexto é um conjunto recursivo.

Prova -

Seja *G* uma gramática tipo2 e *L =L(G).* A sentença λ está em *L(G)* se e somente se a regra $S \to \lambda$ está em *P.* Desta forma temos um teste para verificar se λ ∈ *L(G)*. Vamos supor que *P* não contém $S \to \lambda$ e seja $\sigma \in V_t^+$ tal que $|\sigma|$=n. Seja T_m o conjunto de cadeias $\alpha \in V^+$ tal que $|\alpha| \le n$ e $S \to^* \alpha$ por uma derivação de *m* passos. Então

$$T_0 = \{S\}$$
$$T_m = T_{m-1} \cup \{\alpha \mid \exists \beta \in T_{m-1}, \beta \to \alpha, |\alpha| \le n\}$$

Desta forma, para qualquer α se $S \to^* \alpha, |\alpha| \le n$ tem-se $\alpha \in T_m$ para algum *m*. Como T_m depende apenas de T_{m-1}, se $T_m = T_{m-1}$ então $T_m = T_{m+1} = T_{m+2} = \ldots$ O algoritmo que propomos irá calcular $T_1, T_2, \ldots$ até que para algum *m*, $T_m = T_{m+1}$. É claro que se σ não estiver em T_m então não estará em *L(G)* e também se $\sigma \in T_m$ então $S \to^* \sigma$.
Devemos mostrar que para algum *m* $T_m = T_{m-1}$. Suponhamos que $T_i \neq T_{i-1}$, logo o número de elementos em T_i é maior do que em T_{i-1}. Seja *V* com *k* elementos. Então o número de cadeias em V^+ de comprimento menor ou igual a n é $k + k^2 + \ldots + k^n \le (k+1)^{n+1}$. Assim, para algum $m \le (k+1)^{n+1}$, $T_m = T_{m-1}$ o que determina a condição de parada para o algoritmo que calcula $T_i, i \ge 1$ até encontrar dois conjuntos iguais.

Este algoritmo é aplicável também para as gramáticas livres de contexto e regulares.

9.5.2 – **Exemplo**

Seja a gramática livre de contexto com alfabeto ∑={a,b} e as seguintes produções:

$1) S \to aB$

$2) S \to bA$

$3) B \to b$

$4) A \to a\ S$

Vamos verificar se aaa está em *L(G)*
Então podemos visualizar as seguintes derivações:

$S \to aB \to ab$

$S \to bA \to baS$ → baaB → baab

$S \to bA \to baS$ → babA → babaS

Assim

$T_0 = \{S\}$

$T_1 = \{S,aB,bA\}$

$T_2 = \{S,aB,bA,ab,baS\}$

$T_3 = \{S,aB,bA,ab,baS\}$

Podemos observar que baaB e babA não estão em $T_3 = T_2$ porque possuem comprimentos maiores do que 3. Como aaa não esta em T_2 então $aaa \notin L(G)$.

É importante observar que a recíproca do teorema 9.5.1 não acontece. Portanto, existem linguagens recursivas que não podem ser geradas por gramáticas sensíveis ao contexto.

9.6 – **Gramáticas Estocásticas**

No exemplo 9.4.3.b, vimos que uma mesma cadeia pode ser gerada por vários caminhos distintos o que caracteriza ambigüidade estrutural. Uma forma de amenizar problemas desta natureza é enfocá-lo sob o ponto de vista estocástico, isto é, fazendo com que cada produção tenha uma probabilidade associada a ela. Assim, define-se uma gramática estocástica como uma gramática onde associamos uma probabilidade a cada uma das produções. Os alfabetos V_n e V_t tem os mesmos significados que em gramáticas não estocásticas. O que muda, é o conjunto de produções que passa a ser um conjunto estocástico contento produções da forma:

$$\alpha_i \to^{p_{ij}} \beta_{ij},\ \ j = 1,...,n_i,\ \ j = 1,...,k$$

onde $\alpha_i \in V^+$ e $\beta_{ij} \in V^*$ e p_{ij} é a probabilidade associada com a aplicação dessa produção estocástica onde $0 < p_{ij} \leq 1$ e $\sum p_{ij} = 1$.

Se $\alpha_i \rightarrow^{p_{ij}} \beta_{ij}$ então a cadeia $\sigma = \beta_1 \alpha_i \beta_2$ pode ser substituída por $\tau = \beta_1 \beta_{ij} \beta_2$ com probabilidade p_{ij}. Estas derivações são denotadas por $\sigma \rightarrow^{p_{ij}} \tau$ e dizemos que σ gera diretamente τ com probabilidade p_{ij}. Se existe uma seqüência de símbolos $\tau_1, \tau_2, ..., \tau_{n+1}$ tal que $\sigma = \tau_1$, $\tau = \tau_{n+1}$ e $\tau_1 \rightarrow^{p_i} \tau_{n+1}$, i =1,...,n dizemos que σ gera τ com probabilidade $p = \prod_{i=1}^{n} p_i$ e denotamos esta derivação por $\sigma \rightarrow_*^p \tau$, ou seja, a probabilidade destas derivações é calculada multiplicando- se os valores das probabilidades das produções utilizadas na derivação. O conjunto das linguagens geradas por uma gramática estocástica é o mesmo das linguagens geradas por gramáticas não estocásticas. As linguagens estocásticas são aceitas por autômatos probabilísticos [42][45].
As gramáticas estocásticas podem ser aplicadas em reconhecimento de padrões , como por exemplo:

- Redes Neurais – Na investigação de novos paradigmas e arquiteturas cujos modelos (artificiais) das estruturas neurais podem ser sintetizados por gramáticas estocásticas.[54]
- Textura – quando a textura é descrita através de uma abordagem estrutural, a idéia básica é a de que uma primitiva de textura simples pode ser usada na formação de padrões complexos de textura através de regras que limitam o número de arranjos possíveis.[55]
- Bioinformática - Como o DNA é definido por 4 cadeias de nucleotídeos (adenina, citosina, guanina, e tinina) podemos descrever seqüências genéticas como cadeias de 4 letras C,G,A,T. Desta forma, podemos caracterizar um conjunto qualquer de cadeias de DNA como uma linguagem sobre o alfabeto {C,G,A,T }. As gramáticas estocásticas podem ser utilizadas para gerar classificadores de seqüências de nucleotídeos, que por sua vez, serão utilizados para investigar banco de dados biológicos em busca de estabelecer novas anotações de seqüências por homologia.[56]
- Podem ser utilizadas em "parser" de sentenças de subconjuntos de linguagens naturais.[61]

9.7 - **Gramáticas Fuzzy**

Historicamente, a teoria da probabilidade tem sido a ferramenta utilizada para a representação de *incertezas* em modelagem matemática. Zadeh [64] afirma que nem toda incerteza pode ser tratada randomicamente e propõe a teoria de conjuntos nebulosos, mais conhecida na literatura como conjuntos 'Fuzzy', para tratar com certos tipos de

incertezas onde o alto grau de imprecisão torna impossível a aplicação dos processos randômicos.

Atualmente, a teoria dos conjuntos *fuzzy*, mais especificamente as lógicas *fuzzy*, têm sido aplicadas em várias áreas do conhecimento e utilizadas com grande sucesso em projetos e produtos tecnológicos produzidos pelos japoneses, Mendel[65].

Neste item, introduziremos as gramáticas *fuzzy*, que são ferramentas para geração de conjuntos *fuzzy*. Entretanto, como nosso trabalho é desenvolvido sob o ponto de vista da teoria da computação, ou seja, ele é baseado na teoria clássica de conjuntos, estes sistemas serão abordados apenas de modo informativo, pois do contrário, necessitaríamos desenvolver com mais profundidade a lógica e a teoria dos conjuntos *fuzzy.*

No exemplo 9.4.3.a, vimos que existem situações em que é difícil manter um nível de congruência entre a estrutura sintática e a estrutura semântica que se pretende descrever. Quando estamos trabalhando com informações estruturais mal definidas, a utilização das linguagens denominadas *fuzzy* pode ser uma boa ferramenta. Neste caso, o poder de uma gramática é aumentado associando, na definição de seu vocabulário ou de suas produções, conjuntos f*uzzy*. A linguagem gerada por uma gramática *fuzzy* é um conjunto *fuzzy*, onde cada sentença gerada tem um valor denotando o grau de pertinência da sentença na linguagem. Este valor pode ser obtido usando-se regras denominadas regras de composição *max-min* [61], [62],[65].

9.7.1 - **Conjuntos Fuzzy**

Sem entrar em detalhes teóricos da teoria de conjuntos clássica, suponhamos que temos um universo de discurso X, que pode ser visto como um universo de dados ou informações composto por elementos individuais x. Combinando estes elementos de várias maneiras, podemos construir conjuntos, digamos A, sobre X. Do ponto de vista clássico um elemento deste universo pertence ou não a A . Esta relação de pertinência pode ser representada matematicamente como uma função (função característica), conforme vimos em alguns exemplos em seções anteriores. Assim, podemos representar tais funções como:

$$\chi_A = \begin{cases} 1 & \text{se} \quad x \in A \\ 0 & \text{se} \quad x \notin A \end{cases}$$

onde $\chi_A(x)$ indica de forma não ambígua a pertinência de x em A.

Como foi visto na seção 8 o sistema lógico pressuposto para a formalização da teoria clássica de conjuntos é um sistema binário. Na teoria de conjuntos *fuzzy*, o sistema lógico é construído no intervalo continuo [0,1]. Portanto, pode-se falar num grau de pertinência de um determinado elemento de um universo num conjunto construído a partir deste universo. Conjuntos *fuzzy* são funções que mapeiam um universo de objetos, digamos X, no intervalo [0,1], se F é um conjunto *fuzzy* então a função que define F é denotada por

μ_F e denominada função de pertinência de *F*. Seja *X* um universo de discurso , *x* um elemento de *X* e *A* um conjunto *fuzzy* em *X*. Então a expressão

$$\mu_A(x) \in [0,1]$$

significa o grau de pertinência do elemento *x* no conjunto *fuzzy*. Equivalentemente, $\mu_A(x)$ = grau para o qual $x \in A$, ou seja, $\mu_A(x)$ é um valor em [0,1].
Seja X o universo de discurso então a notação para conjuntos *fuzzy* ,discretos e finitos, é dada pela seguinte forma:

$$A = \sum_i \frac{\mu_A(x_i)}{x_i}$$

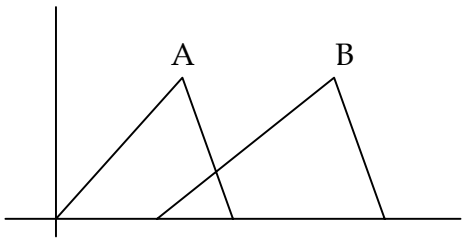
quando *A* é contínuo e infinito temos:

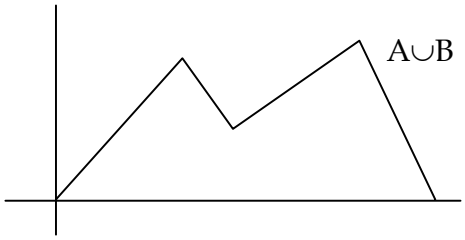
$$A = \int \frac{\mu_A(x)}{x}$$

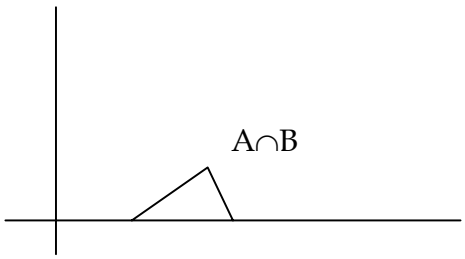
Convém observar que os símbolos acima não significam adição, quociente e nem integração algébrica e sim, união, delimitador e a notação para união de funções continuas para variáveis continuas, respectivamente.
Sejam *A* e *B* conjuntos *fuzzy* sobre X, então, as operações, união, interseção e complemento são definidas para *A* e *B* sobre X, como:

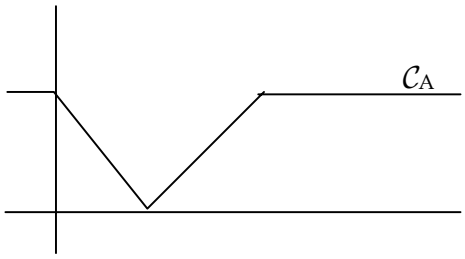
- União $\mu_{A\cup B}(x) = \mu_A(x) \vee \mu_B(x) = \max(\mu_A(x), \mu_B(x))$
- Interseção $\mu_{A\cap B}(x) = \mu_A(x) \wedge \mu_B(x) = \min(\mu_A(x), \mu_B(x))$
- Complemento $\mu_{\bar{A}}(x) = 1 - \mu_A(x)$

Exemplo gráfico [62]:









E ainda,

- $A \subseteq X \Rightarrow \mu_A(x) \leq \mu_X(x)$
- Para todo $x \in X,\ \mu_\varnothing(x) = 0$
- Para todo $x \in X,\ \mu_X(x) = 1$

As leis de De Morgan para conjuntos Fuzzy são dadas:

- $\overline{A \cap B} = \overline{A} \cup \overline{B}$ e $\overline{A \cup B} = \overline{A} \cap \overline{B}$

Como exposto, as operações para conjuntos clássicos também são válidas para conjuntos *fuzzy*, exceto para as leis do terceiro excluído e da contradição, assim, se A é um conjunto *fuzzy*, então:

- $A \cup \overline{A} \neq X$ e $A \cap \overline{A} \neq \varnothing$

As propriedades (associativa, distributiva e etc.) são as mesmas tanto para os conjuntos clássicos quanto para os *fuzzy*. Por isso, e também pelo fato dos valores de pertinência dos conjuntos clássicos pertencerem ao intervalo [0,1] os conjuntos clássicos podem ser tratados como conjuntos *fuzzy*.

As definições das operações de união e interseção apresentadas utilizaram as funções **Max** e **Min**. No entanto, existe uma variedade ilimitada de modos para se chegar ao mesmo fim, e cada um deles tem sua utilidade dependendo da área de aplicação, pois podem alterar significativamente os resultados das inferências nebulosas[62]. Para selecionar as expressões numéricas mais apropriadas para operações difusas, alguns requisitos são necessários: respeitar os resultados usuais quando aplicados a conjuntos clássicos e a conservação de características algébricas e lógicas. Para isso define-se dois operadores denominados t-normas e t-conormas que generalizam a idéia de interseção e união, respectivamente, para a lógica *fuzzy*[62].

9.7.2 – **Gramáticas**

Seja V_t^* o conjunto das cadeias fintas sobre o alfabeto V_t, incluindo a palavra nula λ. Então a linguagem *fuzzy* ($\mathcal{LF}$) sobre V_t é definida como um subconjunto *fuzzy* de V_t^*, como abaixo:

$$\mathcal{LF} = \sum_{x \in V_t^*} \frac{\mu_{LF}(x)}{x}$$

onde $\mu_{\mathcal{LF}}(x)$ é o grau de pertinência de *x* em $\mathcal{LF}$.

Para duas linguagens *fuzzy* quaisquer as operações de união, interseção e etc. seguem das definições dadas anteriormente.

9.7.2.1-**Definição**

Uma gramática *fuzzy* *GF* é uma 6-tupla dada por

$$GF = < V_n, V_t, P, S, J, \mu >$$

onde,

V_n - conjunto de símbolos não terminais.

V_t - conjunto de símbolos terminais.

P – conjunto de regras produções do tipo $\alpha \rightarrow \beta$.

$S \in V_n$ é o símbolo de partida (ou uma sentença a ser reconhecida)

J – é um conjunto de rótulos das regras de produção em P, isto é,
$J = \{ r_i / i = 1,2,...,n$ e n é a cardinalidade de $P \}$.

μ - é um mapeamento $\mu : J \rightarrow [0,1]$, tal que $\mu(r_i)$ denota o grau de pertinência em P da regra rotulada r_i.

Exemplo : (Pal &Majumder[63])

Seja $GF = <\{A,B,S\},\{a,b\},P,S,\{1,2,3,4\},\mu>$ onde *J,P* e μ são:

1. $S \rightarrow AB.....\mu(1) = 0.8$
2. $S \rightarrow aSb.....\mu(2) = 0.2$
3. $A \rightarrow a.....\mu(3) = 1$
4. $B \rightarrow b.....\mu(4) = 1$

9.7.2.2- **Linguagem Gerada**

Uma cadeia $x \in V_t^*$ é dita estar em $\mathcal{L}(GF)$ se e somente se x é derivável de *S* e seu grau de pertinência $\mu_{L(GF)}(x)$ em $\mathcal{L}(GF)$ é maior do que 0, onde

$$\mu_{L(GF)}(x) = \max_{1\le k\le m}\left[\min_{1\le k\le l_k} \mu(r_i^K)\right]$$

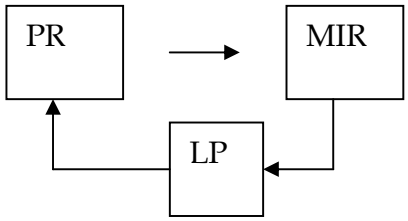
onde *m* é o número de derivações que *x* tem em *GF*; l_k é o comprimento da *k-ésima* cadeia derivada, *k*= 1,2,...,*m* ; e r_i^k é o rótulo da *i-ésima* produção usada na *k-ésima* cadeia derivada, $i = 1,2,...,l_k$.

No módulo 2-2.2, abordaremos com mais detalhes as linguagens dadas pela hierarquia de Chomsky, suas propriedades e seus respectivos reconhecedores.

Módulo 1 – Resumo

No módulo 1, consideramos inicialmente as máquinas MIR como modelo de computação e mostramos que as funções parciais recursivas PR são funções MIR-computáveis. Introduzimos uma linguagem de programação denominada LP e mostramos que todo programa MIR é LP computável, conseqüentemente, LP computa a classe das funções MIR-computáveis, o que pode ser visualizado pela figura abaixo:

Mostramos (seção 4) que o conjunto de programas LP pode ser efetivamente enumerado. Apresentamos um programa na linguagem LP, denominado **universal** e representado por $\mathcal{U}$, que pode executar qualquer programa LP. Através de **universal,** apresentamos o teorema da forma normal de Kleene o qual assegura que qualquer função parcial computável pode ser construída a partir de funções primitivas recursivas, usando uma única aplicação de repetição através da minimização.
Com o teorema de Kleene, mostramos que se uma função é computável via LP então ela é parcial recursiva. E assim, fechamos o ciclo, que pode ser visualizado pela figura:



Através da enumeração efetiva de programas LP, pode-se listar

$$LP_o,\ LP_1,...,LP_n,...$$

onde LP_n é o programa com número de Gödel *n*, assim *n* é o índice do programa LP_n. Com isso, é possível falar de uma lista de funções LP-computáveis

$$\varphi_0,\varphi_1,...,\varphi_n,...$$

onde φ_n é uma função de uma variável computada pelo programa LP_n. Assim,

$$\varphi_n(x)\downarrow \text{ se e somente se } \mathcal{U}(n,x)\downarrow$$

Com isso, mostramos que existem funções parciais recursivas que não são recursivas e finalmente apresentamos o problema da parada.

Referencias:

[1] – Putnam,H. - 1988
[1.a] – Lógica
[1.b] - Formalização
[1.c] - Recursividade
Lógica Combinatória- Enciclopédia Einaudi
Imprensa Nacional – Casa da Moeda
Edição Portuguesa

[2] - Davis,M. – 1958
Computability & Unsolvability
McGraw-Hill, New York

[3] - Cutland,N.J. – 1980
Computability: An Introduction To recursive Funtion Theory
Cambridge University Press

[4] - Brainerd,W.S. & Landweber – 1974
Theory of Computation
Wiley, New York

[5] - Book, R.V. – 1975
Lectures on the theory of Computation
Department of Computer Science
Yale University

[6] - Divério,T.A. & Menezes, P.B. – 1999
Teoria da Computação
Universidade Federal do Rio Grande do Sul
Serie livros Didáticos – Sagra Luzzato

[7] - Carvalho, D.B. – 2001
Criptografia: Métodos e Algoritmos
Editora Book Express

[8] - Minsky, M.L. -1967
Computation:Finite and Infinite Machines
Prantice- Hall

[9] - Luchesi C., Simon&Simon, Kowltowski –1979
Aspectos Teóricos da Computação
IMPA

[10] - Carvalho, R.L. – 1998
Modelos de Computação e Sistemas Formais
11ª. Escola de Computação

[11]- Nagel,E. – 1973
Prova de Gödel
Editora Perspectiva

[12] – Gödel,Kurt - 1979
O teorema de Gödel e a Hipótese do Continuo
Fundação Calouste Gulbenkian

[13] – Shepherdson, J.C.& Sturgis,H.E. – 1963
Computability of recursive functions
J. Ass. Computing Machinery

[14] – Bell,J.&Machover,M. – 1977
A course in Mathematical Logic
North- Holland, Amsterdam

[15] - Kleene, S.C. – 1952
Introduction in Metamathematics
Van Nostrand, Princepton andNorth Holland, Amsterdam

[16] - Smullyan, R.M. – 1961
Theory of Formal Systems
Annals of Mathematics Studies no.47, Princeton

[17] - Papadimitriou, H. & Lewis, R. H. – 1998
Elementos de Teoria da Computação
Bookman

[18] - Singh, Simon – 1997
O Ultimo Teorema de Fermat
Editora Record

[19] - Suppes, P. –
Axiomatic Set Theory
D. Van Nostrand Company, Inc., USA

[20] - Whitehead,A.N.& Russel,B. –1913
Principia Mathematica
Cambridge University, Londres

[21] - Grzegorczyk, A. –1961
Fonctions Recursives
Gauthier-Villars ,Paris

[22] - Davis,M.,Putnam,H.,e Robinson,J. –1961
The decision problem for exponential Diophantine equations,
in Annals of Mathematics, LXXIV, pp 425-36

[23] - Matijasévič,Y. – 1970
Diofontovost pere çislimyh množestv,
in <<Doklady Akademi Nauk SSSR, CXCI,2,pp.279-82

[24] – McCarthy,J. – 1960
Recursive Functions and Their Computation by Machine
Comunications of the ACM

[25] – Yasuhara,A. –1971
Recursive Function Theory & Logic
Academic Press – London

[26] – Hopcroft,J.E & Ullman,J.E. – 1969
Formal Languages and Their Relation to Automata
Addison-Wesley Publishing Company

[27] - Carvalho,R.L. –1989
O escopo da Inteligência Artificial
Relatório Técnico no.015/89
Laboratório Nacional de Computação Científica

[28] – Monteiro,S.L.- 1999
Uma Análise Computacional das Listas de JEFB através das Funções Primitivas Recursivas
Relatório Técnico no. 17/99
Laboratório Nacional de Computação Científica

[29] – Medeiros,J.& Lula,B.- 1982
Teoria da Computação
Relatório Técnico, Departamento de Informática,PUC/RJ.

[30] – Dowling,W.F. – 1989
There Are No Safe Virus Tests
Department of Mathematics and Computer Science
Drexel University, Philadelphia

[31] – Frege,G. 1879,1903
[31.1]- Begriffschrift.,1879
Halle (Repreinted in vanHeijenoort [1],pp.1-82.)
[312]- Grundgesetze der Arithmetik
Jena, Vol.II,1903

[32] – Kowalski ,1983
Logic Programming
Proc.IFIP´83 WorldCongress
North –Holland Publishing Company, Amsterdam,133-145

[33] – Naur,P. – 1963
Revised on the Algorithimic Language Algol 60
Communications of the ACM 6(1) 1-17

[34] - Chomsky,N. – 1956,1959
[34.1] Three models for the description of language-1958
[34.2] On certain formal proprieties of grammars
Inf. And control 2:2 137-167 1959

[35] – Kleene,S.C. – 1956
Representations of events in nerve nets and finite automata
Automata studies, Princeton
University Press

[36] – Myhill, J. – 1957
Finite Automata and the Representation of Events
WADC Tech Report , 57-624, Wright-Patterson AFB, Ohio.113,116

[37] – GinsburgS.,Rice,H.G. –1962
JACM, 9: 350

[38] – Galernter, D. ,2000
A beleza das máquinas – editora Rocco

[39] – Simula 67 –1970
Dahl, O ., Myrhang,B.,Mygaard,K.
Simula 67 Common Base language
Norwegian Computing Center
Oslo –S-22

[40] – Veloso, P.A.S, 1979
Uma Introdução à Teoria de Autômatos
Escola de Computação- São Paulo

[41] - Manna, Z.- 1974
Mathematical Theory of Computation
McGraw-Hill

[42] – Salomaa, 1969
Theory of automata
Pergamon Press, Oxford,1969

[44] - Herman-Rozemberg

[45] - Rabin,M.O. 1967
Mathematical theory of automata;in J.T. Schartz(ed.)
Mathematical aspects of computer science (Proc. Symp. Appl. Math 19);
Amer Math. Soc., RI,1967,pp.153-175.

[46] - Menezes, P.B.,1997
Linguagens Formais e Autômatos
Volume 3- Instituto de Informática UFRGS.

[47] – Booth,T.L.,1967
Sequencial machine and automata theory
Wiley, NY

[48] – Monteiro,S.L. [1987]
Um subconjunto regular do Português
Relatório Técnico – Laboratório Nacional de Computação Científica

[49] – Hilbert, D. 1922
Neubegrundung der Mathematik.
Abhandl. Mathematischen Sem.
Hamburg Univ.,1:151-165

[50] – Thue,A., 1914
Probleme über Veränderugen von Zeichenreihen nach gegebenen Regeln
Skrifter utgit av Videnskapsselskapet i Kristiania 1:10

[51] - Post, E. L. , 1943
Formal reductions of the general combinatorial decision problem
American J. of Mathematics 65,197-268.

[52] – Markov, A.A., 1954
Theory of Algorithms
Academy of Sciences of the USSR, Moscow

[53] – Rezende, P.A.A.de –1999
A crise nos fundamentos da Matemática e a teoria da Computação
Departamento de Ciência da Computação
http://www.cic.unb.br/docentes/pedro/trabs/acrise.htm

[54] - http://www.icmsc.sc.usp.br/~norberto/geral/node 16.html

[55]- http://www.geoc.ufpr.br/~gfoto/pdi/textura/textura.html

[56] – http://www.vision.ime.usp.br/~dvieira/fapesp/nodel.html

[57] – Rangel,J.L.-1997
Linguagens Formais
http://www-di.inf.puc-rio.br/~rangel/lf.html

[58] – Sipser,M.-1997
Introduction to Theory of Computation
PWS Publishing Company

[59] –Monteiro,S.L.-2002
Conceitos elementares da Teoria da Computação (Módulo 1)
Relatório Técnico n 0 43/2002
Laboratório Nacional de Computação Científica –MCT
http://www.lncc.br

[60] –Matiyasevivh,Y.V. –1993
Hilbert's Tenth Problem
MIT Press

[61] – Ross,T.J.,1997
Fuzzy logic with Engineering Applications
McGraw –Hill,Inc.

[62] – Aguiar,H.-1999
Lógica Difusa
Editora Interciência

[63]- Pal, S.&D. Manjuber – 1986
Fuzzy Mathematical Approach to Pattern Recognition
John Wiley & Sons, New York

[64] - Zadeh, L.A.- 1965
Fuzzy Sets
Information and Control, vol.8,pp.338-353

[65] - Mendal,J.M. – 1995
Fuzzy Logic Systems for Engineering: A Tutorial
Proceedings of the IEEE, vol.83, no.3, march