SOLUÇÃO DOS EXERCICIOS SELECIONADOS

Exercícios do Capítulo 1

1. Ex. 1.1)

- **Fluxograma em instruções rotuladas**

  1: **if** T_1 **then goto** 2 **else goto** 3 {o rótulo inicial é o 1}
  2: **do** F **then goto** 3
  3: **if** T_2 **then goto** 4 **else goto** 7
  4: **do** G **then goto** 5
  5: **if** T_1 **then** 7 **else goto** 6 {o rótulo terminal é o 7}
  6: **if do** F **then goto** 1.

- **Função computada**

  (1,v) (2,v) (3, f(v)) (4, f(v)) (5, g(f(v))) (7, g(f(v)))
  (3,v) (4,v) (7, f(v)) (6, g(f(v))) (7, v)

- **O que o programa faz**

  T_1 e T_2 - falso $\rightarrow$ não faz nada
  T_1 e T_2 - verdadeiro $\rightarrow$ g(f(x))
  T_1 falso e T_2 verdadeiro $\rightarrow$ f(g(x))
  T_1 verdadeiro e T_2 falso $\rightarrow$ f(x)

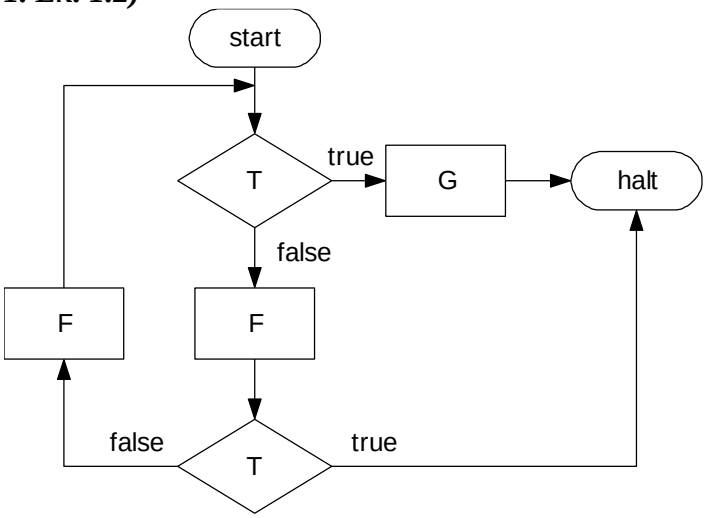
- **Procedure - transição**

  R_1 **where**
  R_1 *is* (**if** T_1 **then** R_2 **else** R_3)
  R_2 *is* F; R_3
  R_3 *is* (**if** T_2 **then** R_4 **else** R_7)
  R_4 *is* G; R5
  R_5 *is* (**if** T1 **then** R_7 **else** R_6)
  R_6 *is* F; R_1
  R_7 *is* I.

- **Procedure - simplificada**

  R_1 **where**
  R_1 *is* (**if** T_1 **then** F; R_3 **else** R_3)
  R_3 *is* (**if** T_2 **then** G; R_5 **else** I)
  R_5 *is* (**if** T_1 **then** I **else** F; R_1)

1. Ex. 1.2)



Primeiramente, traduzimos o fluxograma dado em instruções rotuladas. Assumimos que a computação inicia no rótulo 1 e termina com sucesso quando uma instrução desvia para um rótulo inexistente.

Fluxograma em instruções rotuladas:

1: **se** T **então vá para** 2 **senão vá para** 3
2: **faça** G **então vá para** 6
3: **faça** F **então vá para** 4
4: **se** T **então vá para** 6 **senão vá para** 5
5: **faça** F **então vá para** 1

Agora, convertemos esse fluxograma de instruções rotuladas em um procedimento conforme as seguintes regras:

a) E_i = F; R_j se P contém a instrução
l_i: **faça** F **então vá para** l_j,

b) E_i = (**se** T **então vá para** R_j **senão vá para** R_k) se P contém a instrução
l_i: **if** T **então vá para** l_j **senão vá para** l_k.

Procedimento:

R_1 onde

R_1 *é* (**se** T **então** R_2 **senão** R_3),
R_2 *é* G; R_6,
R_3 *é* F; R_4,
R_4 *é* (**se** T **então** R_6 **senão** R_5),
R_5 *é* F; R_1,
R_6 *é* I.

A seguir, simplificamos o procedimento substituindo os identificadores de procedimento que ocorrem em uma expressão pela expressão que define o identificador de procedimento.

Procedimento simplificado:

R_1 onde

R_1 *é* (**se** T **então** G; I **senão** F; R_4),

R_4 *é* (**se** T **então** I **senão** F; R_1),

1. Ex. 1.3)

- **Fluxograma em instruções rotuladas:**

  1: **do** F **then goto** 2
  2: **if** T_1 **then goto** 1 **else goto** 3
  3: **do** G **then goto** 4
  4: **if** T_2 **then goto** 5 **else goto** 1

- **Procedure:**

  R_1 **where**
  R_1 *is* F; R_2,
  R_2 *is* (**if** T_1 **then** R_1 **else** R_3),
  R_3 *is* G; R_4,
  R_4 *is* (**if** T_2 **then** R_5 **else** R_1),
  R_5 *is* I.

- **Procedure simplificada:**

  R_1 **where**
  R_1 *is* F; (**if** T_1 **then** R_1 **else** G; R_4),
  R_4 *is* (**if** T_2 **then** I **else** R_1).

1. Ex. 1.4)

Primeiro vamos escrever o fluxograma em instruções rotuladas, observando as seguintes convenções:

- a computação sempre inicia no rótulo 1 ;
- a computação sempre termina com sucesso quando uma cláusula faz saltar para um rótulo inexistente de instrução.

1: IF T then goto 2 else goto 3

2: Do F then goto 1

Escrevendo-se agora estas instruções rotuladas em forma de procedure temos:

R1 Where

R1 is (IF T then R2 else R3)

R2 is F;R1

R3 is I

Onde I = expressão nula, computação termina quando se chega a I.

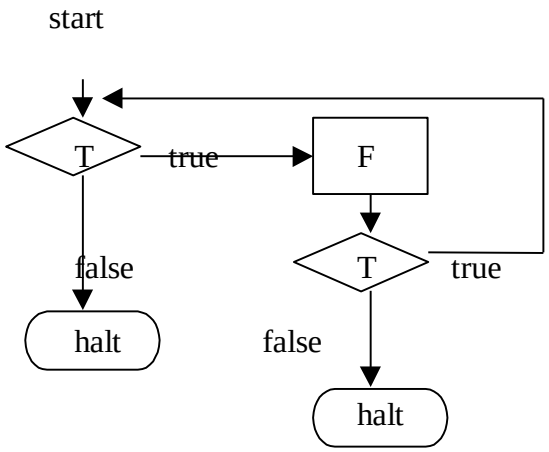
Pelo teorema 1.2 (pg 20) pode-se substituir os identificadores de procedure por suas expressões, com isso temos a procedure simplificada:

R1 Where

R1 is (IF T then F;R1 else I).

1. Ex. 1.5)

Primeiro traduzimos o fluxograma para instruções rotuladas:



Substituindo o identificador de teste T por l_i: **if** T **then goto** l_j **else goto** l_k; e substituindo o indicador de operação O por l_j: **do** O **then goto** l_m temos então

- **Fluxograma em instruções rotuladas:**

  1: **if** T **then goto** 2 **else goto** 4
  2: **do** F **then goto** 3
  3: **If** T **then goto** 1 **else goto** 4

  Onde a instrução 4 indica o fim do programa.

  Para transformarmos as instruções rotuladas em Procedure é necessário trocarmos os rótulos l por expressões do tipo **Ri** is **Ei** .

  Onde E_i é **O; R_j**, se a instrução l for do tipo do O then goto l_i,

  ou E_i é **R_1 is (if** T **then** R_j **else** R_k**)**, se a instrução l_i for do tipo **If** T **then goto** l_j **else goto** l_k**.**

  Incluirmos o rótulo terminal em uma expressão do tipo **R_i** is **I**.

- **Procedure:**

  R_1 **where**
  R_1 is (**if** T **then** R2 **else** R4),
  R_2 *is* F; R_3,
  R_3 *is* (**if** T **then** R_1 **else** R_4),
  R_4 *is* I.

  Para simplificarmos as procedures temos apenas de substituir as expressões do tipo R_i is O; R_j, nas expressões R_l is (if T then R_i else R_j), onde toda vez que aparece trocamos por O; R_j.

Procedure:	**Procedure simplificada:**
R_1 **where** R_1 *is* (**if** T **then** ~~R_2~~ **else** ~~R_4~~), R_2 *is* F; R_3, R_3 *is* (**if** T **then** R_1 **else** ~~R_4~~), R_4 *is* I.	R_1 **where** R_1 *is* (**if** T **then** F; R_3 **else** I), R_3 *is* (**if** T **then** R_1 **else** I).

1. Ex. 2)

{programa tipo **while**}

until T **do** (I);
while T **do** F; G; (**if** T **then** F; (**until** T **do** (I)) **else** I);

ou

until T **do** I;
while T **do** (F;G;(**if** T **then** F));

1. Ex. 3)

Suponha o programa tipo enquanto W. Uma computação P que termina em uma máquina M é uma seqüência finita (x_1,v_1) ,..., (x_n, V_n) de pares programa enquanto x_j e elemento v_j em V tal que:

a) $x_1 = E_1$ e $x_j = I$.
b) para cada j: $1 \leq j \leq n$, x_j é da forma
b.1) F;X' para algum identificador de operação F e nesse caso
$X_{j+1} = X'$ e $V_{j+1} = F_M(vj)$, ou então

b.2) (se T então P senão Q); X' e nesse caso
$X_{j+1} = P; X'$; caso $T_M(vj) = VERDADE$
Q; X'; caso contrário
e $V_{j+1} = Vj$, ou então

b.3) enquanto T faça P; X' e
$X_{j+1} = X_j$ e $V_{j+1} = P_M(Vj)$ se $T_M(vj) = VERDADE$
$X_{j+1} = X'$ e $V_{j+1} = V$ se $T_M(vj) = FALSO$,

ou então,
b.4) Enquanto não T faça (V); W
$X_{j+1} = X_j$ e $V_{j+1} = P_M(Vj)$ se $T_M(vj) = FALSO$
$X_{j+1} = X'$ e $V_{j+1} = V$ se $T_M(vj) = VERDADE$.

1. Ex. 4)

Sim, pois ela permite a verificação de equivalência para programas do mesmo tipo, e ainda compara programas do tipo enquanto com fluxogramas e estes com Procedimentos, dizendo se são equivalentes para qualquer máquina. É aqui que surgem as dificuldades, pois existem máquinas bem simples onde não conseguimos fluxogramas para representar Procedimentos, e nem programas tipo enquanto para representar fluxogramas. Assim esta definição, apesar de ser bem genérica, fica com pouca utilidade e temos que partir para o estudo de equivalência de programas em máquinas especificamente.

Do ponto de vista prático, pode-se passar programas de uma máquina a outra para que executem as mesmas funções corretamente, se verificado que eles são fortemente equivalentes.

1. Ex. 5)

Enquanto T faça **G ; I**
Enquanto não T faça **F** ; se **T** então **I** senão **F.**

Para visualizar melhor, o fluxograma 1.2) foi escrito em instruções rotuladas e depois transformado numa procedure simplificada **equivalente**, como segue:

Instruções Rotuladas

1. If T then goto 2 else goto 3
2. Do G then goto 6
3. Do F then goto 4
4. If T then goto 6 else goto 5
5. Do F then goto 1

Procedure

R1 where
R1 is (if T then R2 else R3)
R2 is G ; R6
R3 is F ; R4
R4 is (if T then R6 else R5)
R5 is F ; R1
R6 is I

Procedure simplificada

R1 where
R1 is (if T then G ; I else F ; R4)
R4 is (if T then I else F ; R1)

1. Ex. 6)

Seja T um identificador de teste e F um identificador de operação:

1) F é um programa do tipo **enquanto**;
2) Sejam V, U dois programas do tipo **enquanto**:
   - I) $V;U$ é um programa do tipo **enquanto**;
   - II) Se T então V senão U é um programa do tipo **enquanto**;
   - III) Enquanto T faça (V) é um programa do tipo **enquanto**;
   - IV) Enquanto não T faça (V) é um programa do tipo **enquanto**;

Definição de computação:

A função computada será definida para uma computação que termina de um programa **enquanto W** em um máquina M.

Seja uma seqüência finita $(W_1, V_1), \ldots, (W_n, V_n)$ onde (W_1, V_1) é um par ordenado com $W_i \in$ **Enquanto** e $V_i \in$ **V** (conjunto de memória de M).

Sendo que I é a condição de parada.

$W_1 = P; I$	,onde W_1 é um programa do tipo **enquanto**;
$V_1 = \mathrm{Im}(E)$	,onde V_1 é o conjunto de memória inicial e recebe a entrada E;
$W_n = I$	,onde W_n indica o fim da computação;
$O_M(V_n) = S$	,onde S é a saída e recebe os dados contidos na memória final (V_n);
$mP : E \to S$	,onde mP é a função computada do programa P na máquina M ;
$mP(x) = O_m(V_N)$	,onde mP para a entrada x é o conteúdo do conjunto de saída quando termina o programa;

A computação da função é executada da seguinte forma:

a) Seqüência:

$Wi = F; W'$ $\qquad (W_i, V_i) \Rightarrow (W_{i+1}, V_{i+1})$

$W_{i+1} = W'$

$V_{i+1} = F_n(V_i)$

Então, (Wi,Vi) ➔ (W',F(Vi))

b) Comando Condicional:

$W_i = (\text{Se } T \text{ então } W' \text{ senão } W''); W'''$

$T_M(V_i) = V \Rightarrow W_{i+1} = W'; W'''$ $\quad V_{i+1} = V_i$,então (Wi,Vi) ➔(W';W''' , Vi)

$T_M(V_i) = F \Rightarrow W_{i+1} = W''; W'''$ $\quad V_{i+1} = V_i$,então (Wi,Vi) ➔(W'';W''', Vi)

c) Comando *Enquanto*:

$W_i =$ Enquanto T faça $(W'); W''$ é um programa do tipo **enquanto**;

$T_M(V_i) = V \Rightarrow W_{i+1} = W'; While\ T\ do\ (W'); W''$ $V_{i+1} = V_i$,então (Wi,Vi) ➔(W';Wi , Vi)
$T_M(V_i) = F \Rightarrow W_{i+1} = W''$ $V_{i+1} = V_i$,então (Wi,Vi) ➔(W'' , Vi)

d) Comando *Enquanto não:*

$W_i =$ Enquanto não T faça $(W'); W''$ é um programa do tipo **enquanto**;
$T_M(V_i) = V \Rightarrow W_{i+1} = W''$; $V_{i+1} = V_i$,então (Wi,Vi) ➔(W'',Vi)
$T_M(V_i) = F \Rightarrow W_{i+1} = W', W_i$ $V_{i+1} = V_i$,então (Wi,Vi) ➔(W';Wi , Vi)

1. Ex. 7)

Seja A: Z := 0
F: Z := Z + 1
T: Z = 0

Um programa while que não termina é:

A; F; While not T do (F);

Este programa inicializa a variável Z com 0, soma um a Z e entra em loop no comando While, já que não é possível variável Z ter o valor 0, uma vez que estamos sempre incrementando um número positivo.

Outro exemplo de programa While que não termina seria:

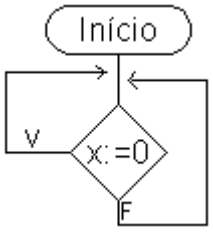
Seja T um teste qualquer:
While T do (I);
Until T do (I);

Neste programa, caso T retorne Verdadeiro, o programa entra em loop no comando while, já que I é o programa vazio e não alterara o valor de T. Caso contrário (T retorna Falso) o programa entra em loop.

1. Ex. 8)

O fluxograma sem intruções, é exatamente um fluxograma vazio, ou seja, um desenho em branco. Um fluxograma sem rótulo teminal pode ser feito de infinitas formas.

Apresentando graficamente o fluxograma sem instruções e um exemplo de fluxograma sem rótulo terminal, temos:



Fluxograma sem instrução

Fluxograma sem rótulo terminal

1. ***Ex. 9)***

Se a máquina M2 é compatível com a máquina M1, podendo ter mais testes e operações, sabe-se que qualquer programa P que tenha uma computação finita em M1 também o terá em M2, logo suas funções computadas em ambas as máquinas serão iguais . Pode ocorrer, entretanto, que um programa Q, tenha uma computação finita em M2 mas não em M1, neste caso a computação de Q em M1 pode não ser definida ou estar em Loop .
O * **poder Computacional** de M1, não necessariamente é menor do que M2. Os novos testes e operações de M2 podem simplesmente facilitar a programação, mas não possibilitar a resolução de um novo problema . Um exemplo disto está no relacionamento das máquinas **NORMA e SAM definidas no próximo capítulo.

* O grau de Poder computacional de uma determinada máquina é medido através do número de algoritmos possíveis de serem representados em cada modelo.O poder computacional das máquinas, pode ser visto como a classe de linguagens que eles são capazes de reconhecer.

** Norma é uma máquina de registradores e o conjunto de saída são os inteiros não nulos . As operações e testes que ela define são as mais naturais, como adicionar e subtrair uma unidade ao registrador e verificar se um regitrador está vazio.SAM é definida como uma máquina NORMA que possui um array de registradores A(1), A(2),... além dos registradores existentes em Norma. Norma pode simular Sam (pag 44 da apostila). Apesar da nova maquina ter novas operações Norma e Sam possuem o mesmo poder computacional já que Norma é uma máquina Uuniversal .

1. Ex. 10)

Se, em determinada máquina, as funções que especificam as operações e os testes não são totais, então existem valores do conjunto de memória para os quais tais operações e testes não são definidos.

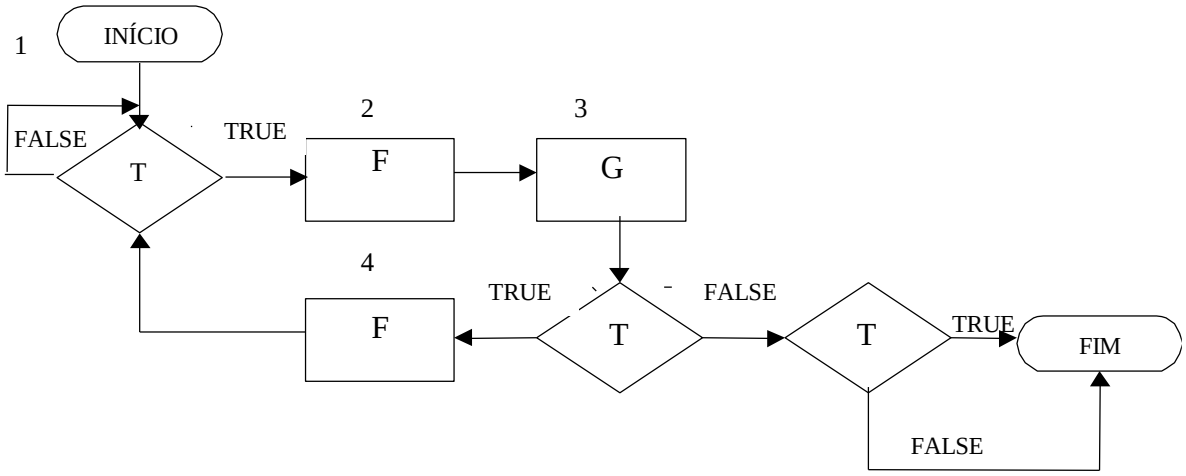
Para as operações e os testes indefinidos, a computação termina com erro devido a esta indefinição. Neste caso, define-se a função computada como indefinida, utilizando um símbolo especial, como, por exemplo, $\uparrow$. A identificação da função computada indefinida através de um símbolo especial faz-se necessária para evitar que os valores contidos na memória, nos casos em que as operações e os testes estão indefinidos, sejam assumidos como função de saída quando a computação termina.

Para os valores do conjunto de memória para os quais as operações e os testes estão definidos, se a computação termina, define-se a função computada como o valor contido na memória no instante em que a computação termina (função de saída). E, quando a computação não termina, define-se que a função computada fica em loop, representando-a pelo símbolo ω.

1. Ex. 11)

1 Etapa:

O fluxograma deve ser mostrado:



Devem ser acrescidos os comentários:

São rotulados os nós do fluxograma que estão associados a identificadores (F ou G) ou às caixas (Início e Fim)

Transcrevendo o fluxograma num conjunto de instruções compostas:

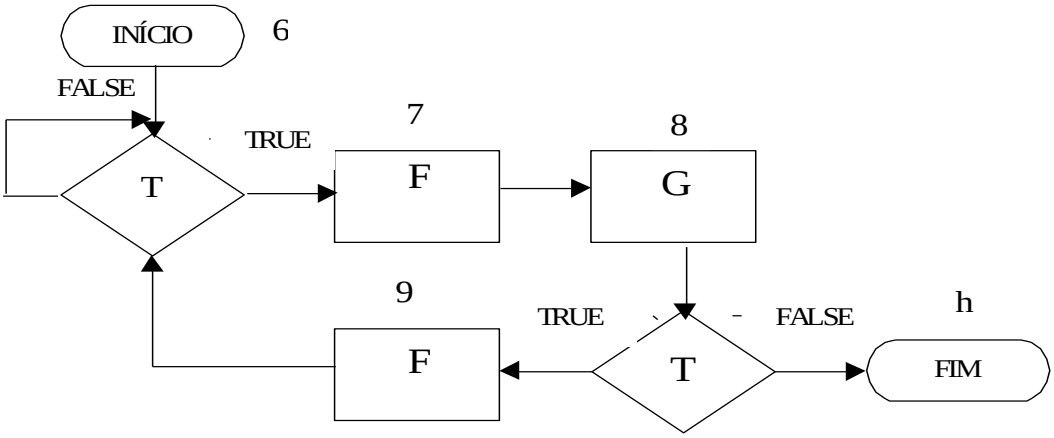
1: (F, 2) or (L,w)
2: (G,3) or (G,3)
3: (F,4) or (H,h)
4: (F,2) or (L,w)
w: (L,w) or (L,w)

Construir a Cadeia Finita do fluxograma P1 a fim de verificar se o fluxograma é simplificado ou não.

$H_0 = \{ h \}$
$H_1 = \{ h, 3\}$
$H_2 = \{ h, 3, 2\}$
$H_3 = \{ h, 3, 2, 1, 4\} \leftarrow lim\ H_k$

2 Etapa,

Procede-se com Q da mesma forma que procedeu-se com P.



6: (F, 7) or (L,w)
7: (G,8) or (G,8)
8: (F,9) or (H,h)
9: (F,7) or (L,w)
w: (L,w) or (L,w)

Verificar se o fluxograma é simplificado ou não.

$H_0 = \{ h \}$
$H_1 = \{ h, 8\}$
$H_2 = \{ h, 8, 7\}$
$H_3 = \{ h, 8, 7, 6, 9\} \leftarrow lim\, H_k$

3Etapa

Compactar os 2 programas P_1 e Q_1 como nos mostra os lemas 1.3 e 1.4 e o algotitmo que segue:

1) Determinar $U_0 = \{(i,j)\}$
   i: primeira instrução de P
   j: primeira instrução de Q
2) Verificar se, no conjunto U_0, os identificadores de operação são os mesmos e se os rótulos sucessores são correspondentes.
3) Formar U_{R+1} e determinar a consistência, ou seja, proceder como no item acima. Se não for consistente, termina.

1: (F, 2) or (L,w)
2: (G,3) or (G,3) — $W_0 = \{ (1,6)\}$ é consistente.
3: (F,4) or (H,h) — $W_1 = \{ (1,6), (2,7), (w,w)\}$ é consistente.
4: (F,2) or (L,w) — $W_2 = \{ (1,6), (2,7), (w,w), (3,8)\}$ é consistente.
w: (L,w) or (L,w) — $W_3 = \{ (1,6), (2,7), (w,w), (3,8), (4,9), (h,h)\}$ é consistente.
6: (F, 7) or (L,w) — $W_4 = W_3$
7: (G,8) or (G,8) — $P_1 \equiv Q_1$ são fortemente equivalentes.
8: (F,9) or (H,h) — $P \equiv Q$ são fortemente equivalentes.

Ex.12)
Verifique se os fluxogramas são fortemente equivalente:

1: do F then goto 2
2: if T then goto 3 else goto 5
3: do G then goto 4
4: if T then goto 1 else goto h
5: do F then goto 6
6: if T then goto 7 else goto 2
7: do G then goto 8
8: if T then goto 6 else goto h

1: do F then goto 2
2: if T then goto 3 else goto 1
3: do G then goto 4
4: if T then goto 1 else goto h

Resposta:

Passo 1: Determinar o conjunto $U_0 = \{(i.j)\}$ onde i é a primeira instrução rotulada composta do programa P e J é a primeira instrução rotulada composta do programa Q.

Passo 2: Verificar se o conjunto U_0 é consistente, ou seja, se os identificadores de operação são os mesmos e se os rótulos sucessores são correspondentes. Caso não seja consistentes, encerrar o algoritmo como não equivalente.

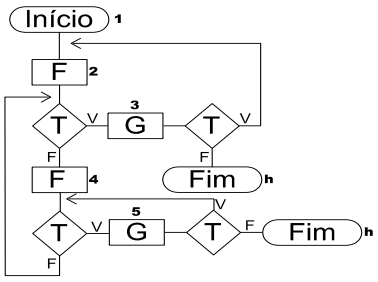
Passo 3: Determinar o conjunto U_{R+1} , a partir dos rótulos sucessores, e verificar a consistência de U_{R+1} , de forma a generalizar cada $(s,t) \in U_R$, onde $s \neq t$, e $s,t \neq h$, os sucessores (s_k , t_k) de (s,t). U_{R+1} consiste de todos os pares generalizados que já não estão em $U_1 , U_2 , ..., U_R$.

Passo 4: Se U_{R+1} é não vazio,

Então conjunto R = R + 1 e retorna ao passo 3,

Senão encerra o algoritmo com equivalente, sendo $W_R = U_0 \cup U_1 \cup ... \cup U_R$, o algoritmo termina e está correto.

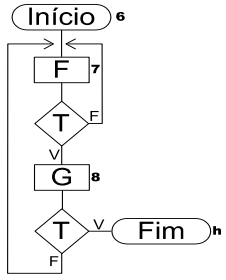
Fluxograma P1



Instruções rotuladas compostas P sobre fluxograma P1
1: (F,2) ou (F,2)
2: (G,3) ou (F,4)
3: (F,2) ou (H,h)
4: (G,5) ou (F,4)
5: (G,5) ou (H,h)

Simplificando: (Lema 1.2)
H0 = {h}
H1 = {h, 5, 3}
H2 = {h, 5, 4, 3, 2}
H3 = {h, 5, 4, 3, 2, 1}

Fluxograma Q1



Instruções rotuladas compostas Q sobre fluxograma Q1
6: (F,7) ou (F,7)
7: (G,8) ou (F,7)
8: (H,h) ou (F,7)

Simplificando: (Lema 1.2)
H0 = {h}
H1 = {h, 8}
H2 = {h, 8, 7}
H3 = {h, 8, 7, 6}

RESPOSTAS DO CAPITULO 3

2. Ex. 1)

O modelo da Máquina de Turing é importante à Ciência da Computação pois através dele é possível determinar quais as funções são computáveis e quais não são. Isto é, através de um conceito bastante simples (no qual é possível provar os teoremas de forma facilitada) define-se a classe das funções calculáveis. Assim, se uma função pode ser calculada, há um modelo de Turing ou equivalente para tal função.

2. Ex. 2)

	Máquina de Turing	**Máquina de Post**	**Aut. F. 2 pilhas**
Memória de Trabalho	▪ fita subdividida em células, que contém um símbolo da palavra ▪ Infinita a direita e finita a esquerda	▪ variável X com comprimento igual ao da palavra de entrada ▪ X não tem tamanho fixo	▪ variável Yi do tipo pilha, sem tamanho fixo ▪ o tamanho é igual ao da palavra de entrada
Entrada	▪ a própria fita	▪ variável X	▪ variável X como a da MP
Saída	▪ a própria fita	▪ variável X	▪ variável Yi
Programa	f(estado(corrente), símbolo(lido)) ⇓ (símbolo, estado, movimento)	seqüência finita de comandos: a) início e fim (aceita e rejeita); b) teste: lê ai e deleta após decisão de fluxo; c) atribuição: X←Xa.	seqüência finita de comandos: a) início e fim; b) teste sobre X ou Yi análogos aos de MP; c) atribuição: Y←ajYi, onde aj ∈ Σ.

2. Ex. 3)

a) O não-determinismo é uma generalização do modelo das máquinas determinísticas. Ele permite que após a leitura de um símbolo, exista mais de uma possibilidade de resolução, de tal forma que as computações de cada "ramo" ocorram em paralelo, cada uma com "memória própria" (fita) e sendo avaliadas até que alcancem o estado final (aceita, rejeita ou loop).

b) Além do programa seguir computações distintas, analisando diversas possibilidades "simultaneamente", o não-determinismo sugere alguma forma natural de paralelismo. No momento em que é encontrado mais de um caminho a ser seguido, uma cópia do programa e suas memórias é remetida a um outro processador. Uma máquina não-determinística *aceitará* uma determinada palavra W, se algum dos caminhos percorridos aceitar W; *rejeitará*, se todos os caminhos percorridos rejeitam W e, entrará em *loop*, se um dos caminhos fica em loop e os demais rejeitam W.

c) Apesar de ser aparentemente mais poderoso, o não-determinismo não acrescenta poder computacional a um modelo. Na melhor das hipóteses o que pode acontecer é chegar a um estado de *aceita* com um número menor de passos, mas isto não implica que dada uma função programa para máquina não determinística, esta não possa ser calculada por uma máquina determinística.

2. Ex. 4)

a) A Hipótese de Church diz que a capacidade de computação representada pela Máquina de Turing é o limite máximo que pode ser atingido por qualquer dispositivo de computação, ou seja, a Máquina de Turing é uma máquina Universal. A partir disso, tem-se a definição das classes de funções computáveis (o que pode ser feito em Turing ou em outro formalismo equivalente) e de funções não computáveis (não existe máquina de Turing que a resolva). Essa Hipótese não é demonstrável pois a noção de algoritmo e função computada é intuitiva.

O fato da noção de algoritmo ser intuitiva é importante pois não pode-se provar matematicamente algo baseado nessa noção intuitiva, é preciso de uma base mais sólida, como um teorema, por exemplo.

b) A Máquina de Turing é um modelo matemático de computador que expressa a solução de problemas. Na Teoria da Computação é importante pois define a classe de funções computáveis e não computáveis. Assim, todos os algoritmos, para os quais existe uma máquina de Turing, são efetivamente implementáveis (solucionáveis) , uma vez que, segundo Church, essa máquina é o dispositivo de computação mais geral existente, não havendo, portanto, qualquer problema solucionável que não possa ser nela implementado.

Essa generalidade da máquina de Turing permite aos estudiosos do assunto desenvolver várias teorias baseados na máquina de Turing e na certeza de ser ela o limiar entre o que pode e o que não pode ser solucionado.

2. Ex. 5)

a) $L_1 = \{ \}$

Seja uma Máquina de Turing $M = (\sum, V, \beta, Q, F, q_0, \pi)$, onde:

$\sum = \{a, b\}$
$V = \{\neg, \beta\}$
β = símbolo "branco"
$Q = \{q_0, q_f\}$
$F = \{q_f\}$
q_0 = estado inicial
π = função programa definida a seguir:

	¬	a	b	ß
q0	(¬,q0,D)			(ß,qf,D)
qf				

A máquina deve aceitar somente uma palavra vazia. Então, no estado q_0, se for encontrado o símbolo de início de fita a máquina passa para o próximo símbolo. Caso esse for o símbolo de *Branco* termina e aceita a palavra. Caso contrário, qualquer que seja o símbolo a máquina não está definida para ele, o que significa que a palavra foi rejeitada.
Desenvolva Máquina de Turing, determinísticas ou não, que aceitem as seguintes linguagens sobre $\Sigma = \{a,b\}$:

b) $L2 = \{\uparrow\}$

Seja uma Máquina de Turing $M = (\sum, V, \beta, Q, F, q_0, \pi)$, onde:

$\sum = \{a, b\}$
$V = \{\neg, \beta\}$
β = símbolo "branco"
$Q = \{q_0, q_r\}$
$F = \{q_r\}$
q_0 = estado inicial
π = função programa definida a seguir:

	¬	a	b	ß
q0	(¬,q0,D)	(a,qr,D)	(b,qr,D)	(ß,qr,D)
qr				

A máquina deve aceitar qualquer palavra. No estado q_0, qualquer que seja o símbolo encontrado (excetuando-se o símbolo de início de fita) a máquina passa para o estado final, aceitando a palavra. Se o símbolo de início for encontrado a máquina permanece no seu estado atual (q_0) e avança para o próximo símbolo a direita.
Desenvolva Máquina de Turing, determinísticas ou não, que aceitem as seguintes linguagens sobre $\Sigma = \{a,b\}$:

c) L3 = {w | w tem o mesmo número de símbolos "a" e "b"}

Seja uma Máquina de Turing $M = (\sum, V, \beta, Q, F, q_0, \pi)$, onde:
$\sum = \{a, b\}$
$V = \{\neg, \beta, A, B\}$.
β = símbolo "branco"
$Q = \{q_0, q_1, q_2, q_3, q_f\}$
$F = \{q_f\}$
q_0 = estado inicial
π = função programa definida a seguir:

	¬	a	b	A	B	ß
q0	(¬,q0,D)	(A,q1,D)	(B,q3,D)	(A,q0,D)	(B,q0,D)	(ß,qf,D)
q1		(a,q1,D)	(B,q2,E)	(A,q1,D)	(B,q1,D)	
q2	(¬,q0,D)	(a,q2,E)	(b,q2,E)	(A,q2,E)	(B,q2,E)	
q3		(A,q2,E)	(b,q3,D)	(A,q3,D)	(B,q3,D)	
qf						

A máquina deve aceitar qualquer palavra que contenha o mesmo número de símbolos "a" e "b". No estado inicial (q_0) a máquina procura pelos símbolos "a" e "b". Caso um desses seja encontrado é substituído por "A" ou "B", respectivamente e avança para o estado q_1 ou q_3 respectivamente. Se no estado q_0 a máquina encontrar "A" ou "B" ela ignora o símbolo e continua sua varredura a direita. No caso de um "β" ser encontrado nesse estado a máquina aceita a palavra.

No estado q_1 a máquina acabou de encontrar um "a" em q_0 e está a procura de um "b". Se este for encontrado é substituído por "B" e a máquina vai para o estado q_2. Se "a", "A" ou "B" forem encontrados a máquina continua sua procura à direita. No entanto, se um "β" for encontrado neste estado a máquina rejeita a palavra pois a entrada contém um número desbalanceado de "a"s e "b"s.

O estado q_2 é utilizado apenas para que a máquina retorne até o início da fita a fim de recomeçar a busca por um par “a” “b”.

O estado q_3 é semelhante ao estado q_1, com a diferença de que ao invés de a máquina ter encontrado um “a” em q_0 ela encontrou um “b” em q_0 e está procurando um “a” para formar o par “a” “b”.

d) L4 = {w | o 10º símbolo da direita para a esquerda é “a”}

Uma Máquina de Turing M = ($\sum$, V, β, Q, F, q_0, π), onde:
$\sum$ = {a, b}
V = {¬, β}
β = símbolo “branco”
Q = {q_0, q_1, q_2, q_3, q_4, q_5, q_6, q_7, q_8, q_9, q_{10}, q_f}
F = {q_f}
q_0 = estado inicial
π = função programa definida a seguir:

	¬	a	b	ß
q0	(¬,q0,D)	(a,q0,D)	(b,q0,D)	(ß,q1,E)
q1		(a,q2,E)	(b,q2,E)	
q2		(a,q3,E)	(b,q3,E)	
q3		(a,q4,E)	(b,q4,E)	
q4		(a,q5,E)	(b,q5,E)	
q5		(a,q6,E)	(b,q6,E)	
q6		(a,q7,E)	(b,q7,E)	
q7		(a,q8,E)	(b,q8,E)	
q8		(a,q9,E)	(b,q9,E)	
q9		(a,q10,E)	(b,q10,E)	
q10		(a,qf,D)		
qf				

e) L5 = {waw | w é palavra de Σ^*}

Uma Máquina de Turing M = ($\sum$, V, β, Q, F, q_0, π), onde:
$\sum$ = {a, b}
V = {¬, A, B, C, β}
β = símbolo "branco"
Q = {q_0, q_1, q_2, q_3, q_4, q_5, q_6, q_7, q_8, q_9, q_{10}, q_{11}, q_{12}, q_{13}, q_f}
F = {q_f}
q_0 = estado inicial
π = função programa definida a seguir:

	¬	a	b	A	B	C	β
q0	(¬,q0,D)	(C,q1,D)	(b,q0,D)				
q1		(a,q1,D)	(b,q1,D)	(A,q2,E)			(ß,q2,E)
q2		(A,q3,E)	(b,q2,E)			(C,q4,D)	
q3		(a,q3,E)	(b,q3,E)	(A,q0,D)		(a,q0,D)	
q4			(b,q4,D)	(a,q4,D)			(ß,q5,E)
q5	(¬,q6,D)	(a,q5,E)	(b,q5,E)			(C,q5,E)	
q6		(A,q7,D)	(B,q11,D)			(C,q13,D)	
q7		(a,q7,D)	(b,q7,D)			(C,q8,D)	
q8		(A,q9,E)		(A,q8,D)	(B,q8,D)		
q9				(A,q9,E)	(B,q9,E)	(C,q10,E)	
q10		(a,q10,E)	(b,q10,E)	(A,q6,D)	(B,q6,D)		
q11		(a,q11,D)	(b,q11,D)			(C,q12,D)	
q12			(B,q9,E)	(A,q12,D)	(B,q12,D)		
q13				(A,q13,D)	(B,q13,D)		(ß,qf,D)
qf							

f) L6 = {ww | w é palavra de Σ^*}
Uma Máquina de Turing M = ($\sum$, V, β, Q, F, q_0, π), onde:
$\sum$ = {a, b}
V = {¬, A, B, β}
β = símbolo "branco"
Q = {q_0, q_1, q_2, q_3, q_4, q_5, q_6, q_7, q_8, q_9, q_f}
F = {q_f}
q_0 = estado inicial
π = função programa definida a seguir:

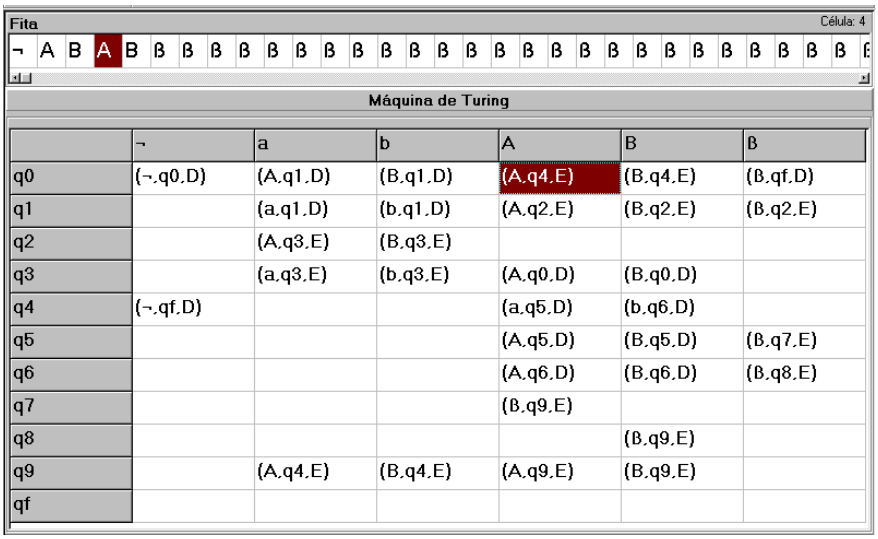
	¬	a	b	A	B	ß
q0	(¬,q0,D)	(A,q1,D)	(B,q1,D)	(A,q4,E)	(B,q4,E)	(ß,qf,D)
q1		(a,q1,D)	(b,q1,D)	(A,q2,E)	(B,q2,E)	(ß,q2,E)
q2		(A,q3,E)	(B,q3,E)			
q3		(a,q3,E)	(b,q3,E)	(A,q0,D)	(B,q0,D)	
q4	(¬,qf,D)			(a,q5,D)	(b,q6,D)	
q5				(A,q5,D)	(B,q5,D)	(ß,q7,E)
q6				(A,q6,D)	(B,q6,D)	(ß,q8,E)
q7				(ß,q9,E)		
q8					(ß,q9,E)	
q9		(A,q4,E)	(B,q4,E)	(A,q9,E)	(B,q9,E)	
qf						

Comentários:

Exemplo de palavra de entrada: ww : abab

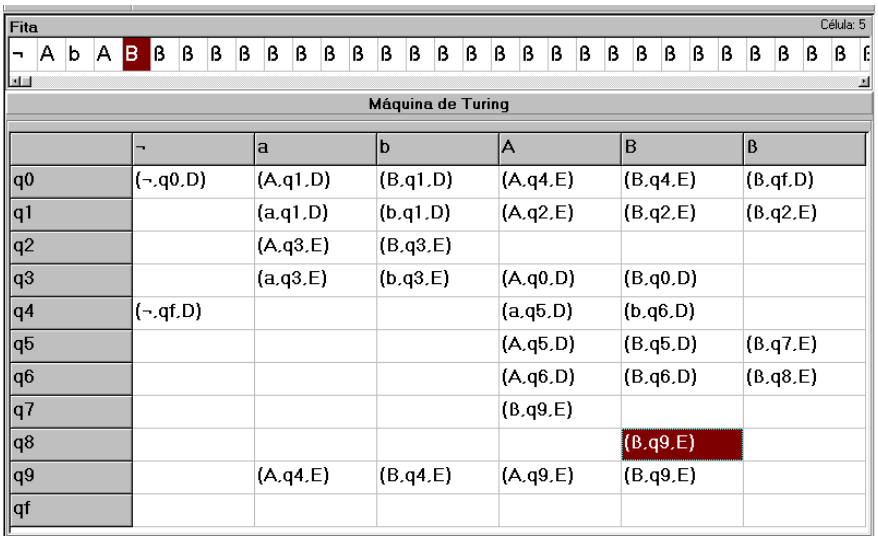
Para melhor entendimento, ww = w^1w^2, onde $w^1 = w^2 = w$.

A máquina posiciona a cabeça da fita no inicio da palavra w^2 . Esse processamento funciona da seguinte maneira: pega a primeira letra e escreve-a em maiúscula. Logo depois percorre a fita até o fim e escreve a última letra em maiúscula. Percorre a fita à esquerda até encontrar um símbolo em maiúsculo. Escreve a letra `a direita desse símbolo em maiúscula... E assim sucessivamente até encontrar o meio da palavra ww.



	¬	a	b	A	B	ß
q0	(¬,q0,D)	(A,q1,D)	(B,q1,D)	(A,q4,E)	(B,q4,E)	(ß,qf,D)
q1		(a,q1,D)	(b,q1,D)	(A,q2,E)	(B,q2,E)	(ß,q2,E)
q2		(A,q3,E)	(B,q3,E)			
q3		(a,q3,E)	(b,q3,E)	(A,q0,D)	(B,q0,D)	
q4	(¬,qf,D)			(a,q5,D)	(b,q6,D)	
q5				(A,q5,D)	(B,q5,D)	(ß,q7,E)
q6				(A,q6,D)	(B,q6,D)	(ß,q8,E)
q7				(ß,q9,E)		
q8					(ß,q9,E)	
q9		(A,q4,E)	(B,q4,E)	(A,q9,E)	(B,q9,E)	
qf						

Pega a última letra da palavra w^1, escreve-a em minúscula . Compara com a última letra da w^2. Caso seja igual, é escrito o símbolo branco ß.



	¬	a	b	A	B	ß
q0	(¬,q0,D)	(A,q1,D)	(B,q1,D)	(A,q4,E)	(B,q4,E)	(ß,qf,D)
q1		(a,q1,D)	(b,q1,D)	(A,q2,E)	(B,q2,E)	(ß,q2,E)
q2		(A,q3,E)	(B,q3,E)			
q3		(a,q3,E)	(b,q3,E)	(A,q0,D)	(B,q0,D)	
q4	(¬,qf,D)			(a,q5,D)	(b,q6,D)	
q5				(A,q5,D)	(B,q5,D)	(ß,q7,E)
q6				(A,q6,D)	(B,q6,D)	(ß,q8,E)
q7				(ß,q9,E)		
q8					(ß,q9,E)	
q9		(A,q4,E)	(B,q4,E)	(A,q9,E)	(B,q9,E)	
qf						

A cabeça da fita vai até a próxima célula a ser comparada. Neste caso a letra à esquerda da minúscula(que será rescrito novamente em maiúscula). Caso ww seja uma palavra de Σ*, a palavra é aceita e a computação finalizada.

g) L7 = {ww^r | w é palavra de Σ^*}

Uma Máquina de Turing M = ($\sum$, V, β, Q, F, q_0, π), onde:

$\sum$ = {a, b}

V = {¬, A, B, β}

β = símbolo “branco”
$Q = \{q_0, q_1, q_2, q_3, q_4, q_5, q_6, q_7, q_8, q_9, q_{10}, q_{11}, q_{12}, q_{13}, q_f\}$
$F = \{q_f\}$
q_0 = estado inicial
π = função programa definida a seguir:

	¬	a	b	A	B	ß
q0	(¬,q0,D)	(A,q1,D)	(B,q1,D)	(a,q4,D)	(b,q4,D)	(ß,qf,E)
q1		(a,q1,D)	(b,q1,D)	(A,q2,E)	(B,q2,E)	(ß,q2,E)
q2		(A,q3,E)	(B,q3,E)			
q3		(a,q3,E)	(b,q3,E)	(A,q0,D)	(B,q0,D)	
q4				(a,q4,D)	(b,q4,D)	(ß,q5,E)
q5		(ß,q6,E)	(ß,q11,E)			
q6		(a,q6,E)	(b,q6,E)	(A,q7,E)	(B,q7,E)	
q7	(¬,q8,D)	(a,q8,D)	(b,q8,D)	(A,q7,E)	(B,q7,E)	
q8				(a,q9,D)		
q9		(a,q10,D)	(b,q10,D)	(A,q9,D)	(B,q9,D)	(ß,qf,E)
q10		(a,q10,D)	(b,q10,D)			(ß,q5,E)
q11		(a,q11,E)	(b,q11,E)	(A,q12,E)	(B,q12,E)	
q12	(¬,q13,D)	(a,q13,D)	(b,q13,D)	(A,q12,E)	(B,q12,E)	
q13					(b,q9,D)	
qf						

Comentários:

Obs: w^r é a palavra w escrita ao contrário.

Exemplo de palavra de entrada: ww^r : abba

A máquina posiciona a cabeça da fita no início da palavra w^r . Esse processamento funciona da seguinte maneira: pega a primeira letra e rescreve-a em maiúscula. Logo depois percorre a fita até o fim e rescreve a última letra em maiúscula. Percorre a fita à esquerda até encontrar um símbolo em maiúsculo. Rescreve a letra `a direita desse símbolo em maiúscula... E assim sucessivamente até encontrar o meio da palavra ww^r.

Escreve a palavra w^r em minúscula. Pega o último símbolo da palavra w^r e escreve branco. Compara com a primeira letra da palavra w e escreve em minúscula(se for a mesma letra). Pega o próximo e faz o mesmo processo.

Fita — Célula: 4

¬	A	B	B	A	ß	ß	ß	ß	ß	ß	ß	ß	ß	ß	ß	ß	ß	ß	ß	ß	ß	ß	ß	ß	ß	ß	ß	ß	ß	ß

Máquina de Turing

	¬	a	b	A	B	ß
q0	(¬,q0,D)	(A,q1,D)	(B,q1,D)	(a,q4,D)	(b,q4,D)	(ß,qf,E)
q1		(a,q1,D)	(b,q1,D)	(A,q2,E)	(B,q2,E)	(ß,q2,E)
q2		(A,q3,E)	(B,q3,E)			
q3		(a,q3,E)	(b,q3,E)	(A,q0,D)	(B,q0,D)	
q4				(a,q4,D)	(b,q4,D)	(ß,q5,E)
q5		(ß,q6,E)	(ß,q11,E)			
q6		(a,q6,E)	(b,q6,E)	(A,q7,E)	(B,q7,E)	
q7	(¬,q8,D)	(a,q8,D)	(b,q8,D)	(A,q7,E)	(B,q7,E)	
q8				(a,q9,D)		
q9		(a,q10,D)	(b,q10,D)	(A,q9,D)	(B,q9,D)	(ß,qf,E)
q10		(a,q10,D)	(b,q10,D)			(ß,q5,E)
q11		(a,q11,E)	(b,q11,E)	(A,q12,E)	(B,q12,E)	
q12	(¬,q13,D)	(a,q13,D)	(b,q13,D)	(A,q12,E)	(B,q12,E)	
q13					(b,q9,D)	
qf						

Fita — Célula: 5

¬	A	B	b	a	ß	ß	ß	ß	ß	ß	ß	ß	ß	ß	ß	ß	ß	ß	ß	ß	ß	ß	ß	ß	ß	ß	ß	ß	ß	ß

Máquina de Turing

	¬	a	b	A	B	ß
q0	(¬,q0,D)	(A,q1,D)	(B,q1,D)	(a,q4,D)	(b,q4,D)	(ß,qf,E)
q1		(a,q1,D)	(b,q1,D)	(A,q2,E)	(B,q2,E)	(ß,q2,E)
q2		(A,q3,E)	(B,q3,E)			
q3		(a,q3,E)	(b,q3,E)	(A,q0,D)	(B,q0,D)	
q4				(a,q4,D)	(b,q4,D)	(ß,q5,E)
q5		(ß,q6,E)	(ß,q11,E)			
q6		(a,q6,E)	(b,q6,E)	(A,q7,E)	(B,q7,E)	
q7	(¬,q8,D)	(a,q8,D)	(b,q8,D)	(A,q7,E)	(B,q7,E)	
q8				(a,q9,D)		
q9		(a,q10,D)	(b,q10,D)	(A,q9,D)	(B,q9,D)	(ß,qf,E)
q10		(a,q10,D)	(b,q10,D)			(ß,q5,E)
q11		(a,q11,E)	(b,q11,E)	(A,q12,E)	(B,q12,E)	
q12	(¬,q13,D)	(a,q13,D)	(b,q13,D)	(A,q12,E)	(B,q12,E)	
q13					(b,q9,D)	
qf						

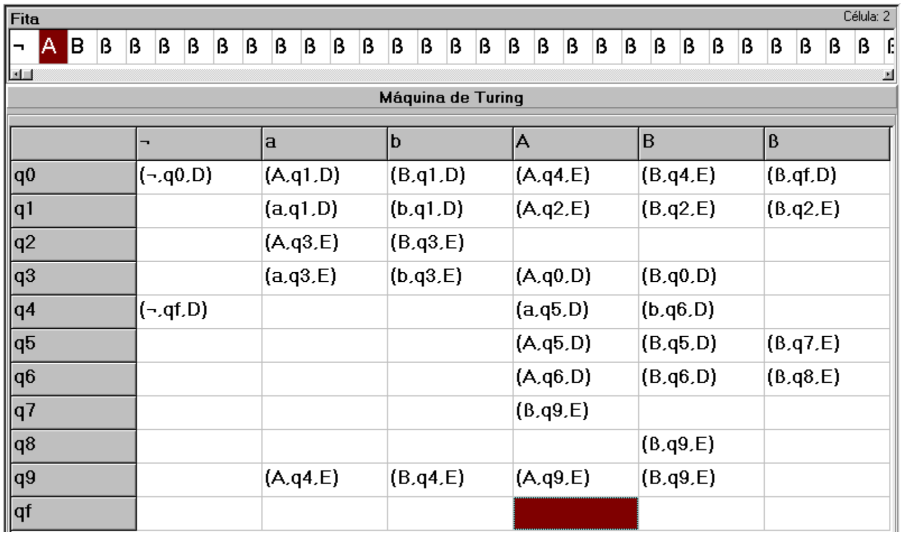
Se estiver tudo certo, a palavra é aceita e a computação finalizada.

Fita — Célula: 3

¬	a	b	ß	ß	ß	ß	ß	ß	ß	ß	ß	ß	ß	ß	ß	ß	ß	ß	ß	ß	ß	ß	ß	ß	ß	ß	ß	ß	ß	ß

Máquina de Turing

	¬	a	b	A	B	ß
q0	(¬,q0,D)	(A,q1,D)	(B,q1,D)	(a,q4,D)	(b,q4,D)	(ß,qf,E)
q1		(a,q1,D)	(b,q1,D)	(A,q2,E)	(B,q2,E)	(ß,q2,E)
q2		(A,q3,E)	(B,q3,E)			
q3		(a,q3,E)	(b,q3,E)	(A,q0,D)	(B,q0,D)	
q4				(a,q4,D)	(b,q4,D)	(ß,q5,E)
q5		(ß,q6,E)	(ß,q11,E)			
q6		(a,q6,E)	(b,q6,E)	(A,q7,E)	(B,q7,E)	
q7	(¬,q8,D)	(a,q8,D)	(b,q8,D)	(A,q7,E)	(B,q7,E)	
q8				(a,q9,D)		
q9		(a,q10,D)	(b,q10,D)	(A,q9,D)	(B,q9,D)	(ß,qf,E)
q10		(a,q10,D)	(b,q10,D)			(ß,q5,E)
q11		(a,q11,E)	(b,q11,E)	(A,q12,E)	(B,q12,E)	
q12	(¬,q13,D)	(a,q13,D)	(b,q13,D)	(A,q12,E)	(B,q12,E)	
q13					(b,q9,D)	
qf						



	¬	a	b	A	B	β
q0	(¬,q0,D)	(A,q1,D)	(B,q1,D)	(A,q4,E)	(B,q4,E)	(β,qf,D)
q1		(a,q1,D)	(b,q1,D)	(A,q2,E)	(B,q2,E)	(β,q2,E)
q2		(A,q3,E)	(B,q3,E)			
q3		(a,q3,E)	(b,q3,E)	(A,q0,D)	(B,q0,D)	
q4	(¬,qf,D)			(a,q5,D)	(b,q6,D)	
q5				(A,q5,D)	(B,q5,D)	(β,q7,E)
q6				(A,q6,D)	(B,q6,D)	(β,q8,E)
q7				(β,q9,E)		
q8					(β,q9,E)	
q9		(A,q4,E)	(B,q4,E)	(A,q9,E)	(B,q9,E)	
qf						

2. Ex. 13)

Prova: Para provar isso devemos, pela definição de equivalência, mostrar que uma máquina de Turing não-determinística M_1 pode simular uma máquina de Turing determinística M_2 e, vice-versa. Assim sendo, vamos mostrar que M_2 *simula* M_1 e M_1 *simula* M_2.

a)M_2 simula M_1: Ou seja, é possível, a partir de uma máquina de Turing não-determinística M_1 construir uma máquina de Turing determinística M_2 que realiza o mesmo processamento

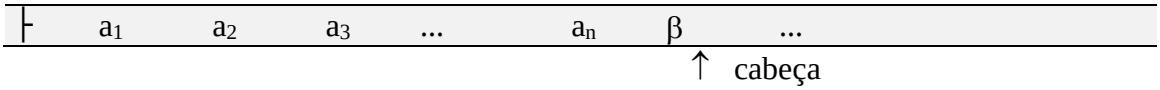
Na máquina de Turing, o alfabeto de símbolos e o número de estados são finitos, então o número de pares estado/símbolo da fita da máquina não-determinística M_1, é também finito, ou seja, existe um número finito de caminhos (movimentos) a serem escolhidos a cada movimento.

Sendo finito o número de movimentos, podemos numerá-los de 1 a n, onde n é o número máximo de possibilidades de escolhas para o próximo movimento. Então qualquer seqüência de escolhas pode ser representado por uma seqüência de dígitos de 1 a n (algumas seqüências podem ser menor que n).

Seja M2 uma máquina de Turing determinísitica com três fitas (equivalente a maq. de Turing com 1 fita).

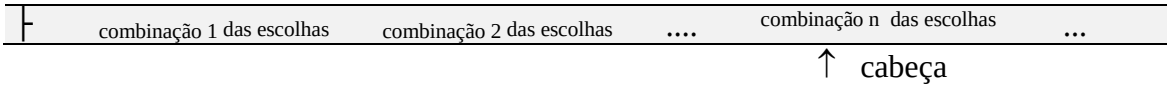
Seja então uma linguagem arbitrária L={w | w é composta por algum alfabeto}. A primeira fita representará a entrada para M_1.

Fita 1



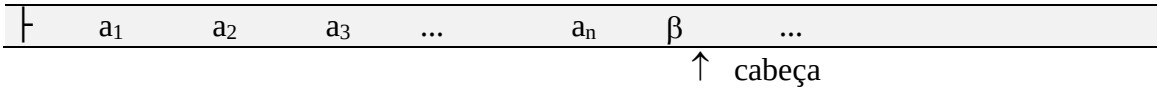
A segunda fita contém todas as combinações de "instruções" (relações estado-símbolo) em ordem crescente.

Fita 2



Cada combinação da segunda fita é copiada para a entrada da terceira fita, onde é simulada a M_1 de acordo com cada combinação da fita 2.

Fita 3



Então se M_2 encontra o estado final de ACEITA, M_1 também encontra o mesmo estado final. E se M_2 não aceitar a entrada, M_1também não aceitará, e ambas irão para o estado de REJEITA.

b)M_1 simula M_2: Ou seja, é possível, a partir de uma máquina de Turing determinística M_2 construir uma máquina de Turing não-determinística M_1 que realiza o mesmo processamento

Tendo uma máquina de Turing determinísitica M_2, é fácil transformá-la numa máquina de Turing não-determinísitica M_1. Basta acrescermos um caminho alternativo em qualquer ponto do programa que não altere o caminho já existente, e que leve para o estado final de REJEITA. A criação deste caminho transformará uma máquina determinísitica em uma máquina não-determinísitica. Os caminhos que acrescidos em M_1 levam ao estado final de ACEITA deve aceitar a mesma linguagem aceita por que em M_2.

2. Ex.19)

A linguagem aceita por uma máquina de Turing MT é denotada por L(M), que é o conjunto de todas as palavras $w \in \Sigma^*$, que ao serem processadas na máquina resulta que ela entre no estado final de ACEITA. Se a máquina para no estado final ACEITA, diz-se que a máquina aceita a entrada $w \in \Sigma^*$, portanto $w \in L(M)$. Caso contrário (pára em rejeita ou permanece em loop indefinidamente) ela rejeita a entrada $w \in \Sigma^*$ e $w \notin L(M)$.

A)

Isso tem implicações sobre a classe das palavras reconhecidas e, para verificar as diferenças observemos que:

i) Pode-se definir que uma linguagem aceita por uma máquina de Turing MT é *enumerável recursivamente* (ER), se todas as linguagens podem ser enumeradas ou listadas. Daí se $L \in$ classe ER, então para toda máquina M que aceita a linguagem L, em pelo menos uma máquina destas ,onde existe pelo menos uma palavra $w \notin L$ que ao ser processada por ela resulta que a máquina M entre em Loop infinito, ou seja:

a) Se $w \in L$, a máquina M pára e aceita w;

b) Se $w \notin L$, a máquina M pode parar rejeitando w ou permanecer em Loop infinito.

ii) Uma classe de ER, denominada *Recursiva* é a classe das linguagens para as quais existe pelo menos uma máquina de Turing que pára, para qualquer entrada, aceitando ou rejeitando a palavra de entrada.

B)

Para verificar a importância da classe *enumerável recursivamente* (ER) e da *Recursiva* observemos que das suas definições, a classe *Recursiva* pára necessariamente, rejeitando ou aceitando a palavra, já a classe ER pode entrar em Loop.

De uma forma análoga às classificações anteriores, as linguagens ER podem ser comparadas como *funções parciais*, enquanto as *recursivas* podem ser comparadas como *funções*

2. Ex.23)

Inicialmente é necessário codificar o fluxograma para instruções rotuladas, lembrando que o rótulo terminal (se houver) é o rótulo 0. O conjunto de instruções rotuladas é dado por:

1: Faça F_1 Então Vá Para 2

2: Se T_1 Então Vá Para 1 Senão Vá Para 0

Deve-se fazer agora a codificação através da representação por uma 4-tupla, observando que para o Faça tem-se a 4-tupla (0,k,l,l) e para o Se tem-se (1,k,l,m). Assim temos:

para o rótulo 1: $2^0.3^1.5^2.7^2$;

para o rótulo 2: $2^1.3^1.5^1.7^0$;

Ou seja $n = \sigma^2(1,2) = 2^{3675} .3^{30}$

Comentário: Primeiro passo, como dito na própria resposta, foi converter para instruções rotuladas o fluxograma. Para cada uma das duas instruções, montamos a 4-tupla através de 4 números primos (2,3,5,7) com os expoentes apropriados. No último passo, chegamos ao número natural pedido a que a questão refer-se-ía, sendo que cada um dos fatores representava uma das duas instruções.

2. Ex.24)

Sendo k representado por *true* e m representado por *false*.

B:=0;
B:=B+1;
B:=B+1;
B:=B+1;
Until A=0 do (A:=A-1, B:=B-1);
If B=0 then *true* else *false.*

Comentário

Inicializa-se o registrador B com o valor 3. Decrementa-se tanto A como B, até A chegar a zero. Se B for zero, ou seja, foi decrementado no mínimo 3 vezes, conclui-se que A>2, caso contrário A ≤ 2.

Lembra-se que em NORMA se o registrador esta zerado e é decrementado o seu valor permanece em zero.

2. Ex.25)

As macros pré-definidas são: A:=0 e A:=A+B usando C, assim para responder a questão é necessário construir apenas macros A:=B+D usando F e G e a macro A:=B*D, assim:

Macro A:=B+D

A:=0;
A:=A+B usando C
A:=A+D usando C;

Macro A:=B*D

A:=0;
Até D=0 faça (A:=A+B usando C; D:=D-1);
Portanto pode-se determinar a macro X:=(Y*Z+T) fazendo:
X:=0, B:=0, H:=0
Até que Z=0 faça
X:=X+Y; Z:=Z-1
X:=X+T
B:=B+X
Até que X=0 faça
H:=H+1, X:=X-1
Até que H=0 faça
X:=X+B, H:=H-1

Exemplo: para Y=2, Z=3 e T=1
X=0+2; Z=3-1
X=2+2; Z=2-1
X=4+2 =6;Z:=1-1=0

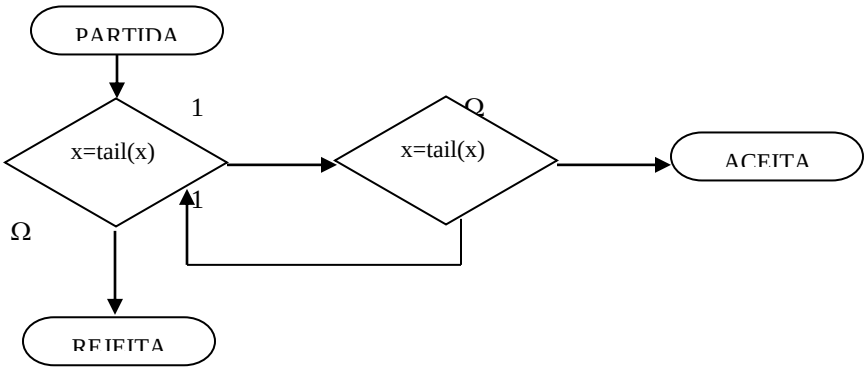
X=6+1=7
B=0+7=7

H=0+1; X=7-1
H=1+1; X=6-1
H=2+1; X=5-1
H=3+1; X=4-1
H=4+1; X=3-1
H=5+1; X=2-1
H=6+1=7; X=1-1=0

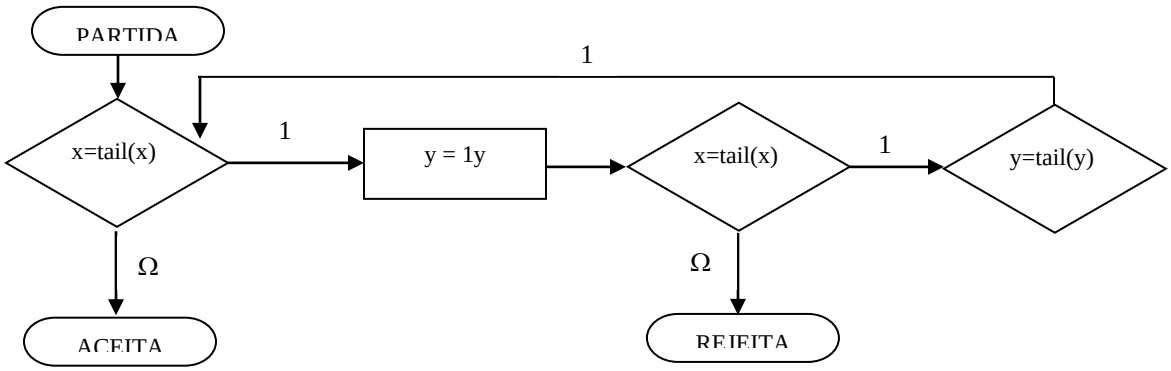
X=X+0; H=6
X=14;H=5
X=21;H=4
X=28;H=3
X=35;H=2
X=42;H=1
X=49;H=0

2. Ex. 26

A versão dada é autômato com uma pilha que aceita números pares e rejeita números ímpares.
Para cada número 1 tirado da pilha ele tira outro número 1 (retira pares de 1's). Assim, se, ao tentar retirar o segundo elemento do par, a pilha estiver vazia, o número de entrada era ímpar e será rejeitado.
Uma variação simples seria a inversão. O autômato aceita números ímpares e rejeita os pares.



Uma outra variação seria um autômato com duas pilhas que aceita números pares e rejeita os ímpares.

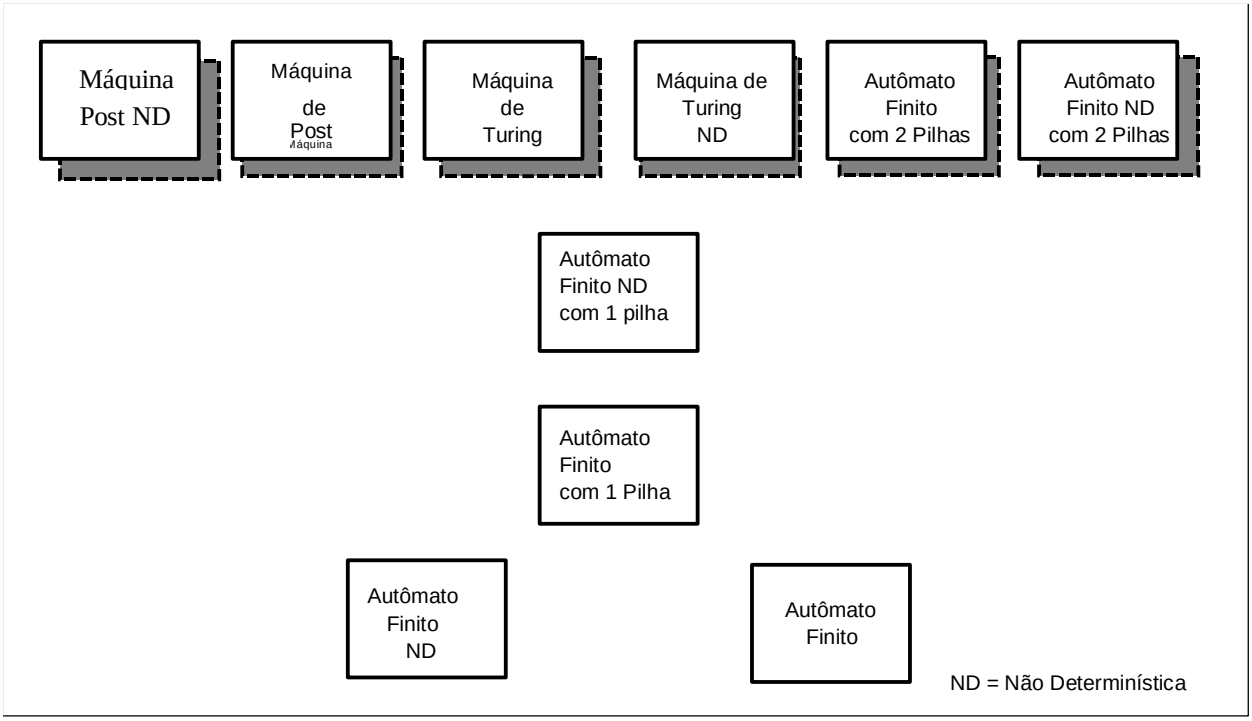


2. Ex.27)

a)

	Máquina de Turing	Máquina de Post	Autômato Finito com n pilhas	Autômato Finito com 2 pilhas	Autômato Finito com 1 pilha	Autômato Finito sem pilhas
Memória de Trabalho	Fita subdividida em células, infinita à direita e finita à esquerda, cada célula contém um símbolo da palavra	Variável X, sem tamanho nem limites, comprimento igual ao da palavra corrente	Variáveis Yi /i(1)n do tipo pilha, não possui tamanho fixo, o tamanho é igual ao da palavra corrente.	Variáveis Y1 e Y2 do tipo pilha, não possui tamanho fixo, o tamanho é igual ao da palavra corrente.	Variável Y do tipo pilha, não possui tamanho fixo, o tamanho é igual ao da palavra corrente.	Não possui memória
Entrada	A própria fita	Variável X	Variável X, parecida com a MP	Variável X, parecida com a MP	Variável X, parecida com a MP	Variável X, parecida com a MP
Saída	A própria fita	Variável X	Variável Yi /i(1)n	Variável Y1,Y2	Variável Y	Não tem
Programa	Definido do produto cartesiano entre o conjunto de símbolos lidos e de estados, o conjunto de chegada é o produto cartesiano entre símbolo, estados e movimentos $(\Sigma\cup V)xQx\{E,D\}$ π(estado(corrente),símbolo(lido)) ➔(símbolo, estado, movimento)	Seqüência finita de comandos : a) Início e Fim (Aceita ou rejeita); b) Teste : Lê a1 e deleta após decisão de fluxo; c) Atribuição: X← Xa	Seqüência finita de comandos : a)Início e Fim b)De teste sobre X ou Yi análogos a MP; c) atribuição: Yi ← aiYi onde ai $\in\Sigma$	Seqüência finita de comandos : a)Início e Fim b)De teste sobre X ou Yi análogos a MP; c) atribuição: Yi ← aiYi onde ai $\in\Sigma$	Seqüência finita de comandos : a)Início e Fim b)De teste sobre X ou Y análogos a MP; c) atribuição: Y ← aiY onde ai $\in\Sigma$	Seqüência finita de comandos : a)Início e Fim b)De teste sobre X
Símbolos	Σ : Alfabeto de entrada V : Alfabeto Auxiliar que contém: Branco(β), Sinal de início(¬), Mais símbolos adicionais a serem definidos durante a construção da máquina	Σ : Alfabeto de entrada # : símbolo auxiliar Ω : indica variável vazia	Σ : Alfabeto de entrada Ω ou £: indica variável vazia	Σ : Alfabeto de entrada Ω ou £: indica variável vazia	Σ : Alfabeto de entrada Ω ou £: indica variável vazia	Σ : Alfabeto de entrada Ω ou £: indica variável vazia

b)



A figura ilustra a relação das classes de máquinas estudadas até agora em relação aos seus poderes computacionais. As máquinas representadas com molduras sombreadas são consideradas universais (por exemplo : Máquina de Turing , Máquina de Post) e são equivalentes. As que estão em uma mesma linha (por exemplo, Autômato Finito e Autômato Finito Não Determinístico) pertencem a uma mesma classe em termos de poder computacional, o qual é menor para máquinas com posições mais baixas (por exemplo, Autômato Finito com uma Pilha tem menor grau de poder computacional do que Autômato Finito Não Determinístico com uma Pilha).

A justificativa é que cada máquina possui um grau de poder computacional (as máquinas com mesmo grau computacional pertencem à mesma classe) e modificações aplicadas a tais máquinas podem torná-las mais poderosas ou apenas facilitar a manipulação das mesmas, porém considerando-se máquinas universais sabe-se que, por definição, nenhuma modificação pode aumentar seu poder computacional.

c)

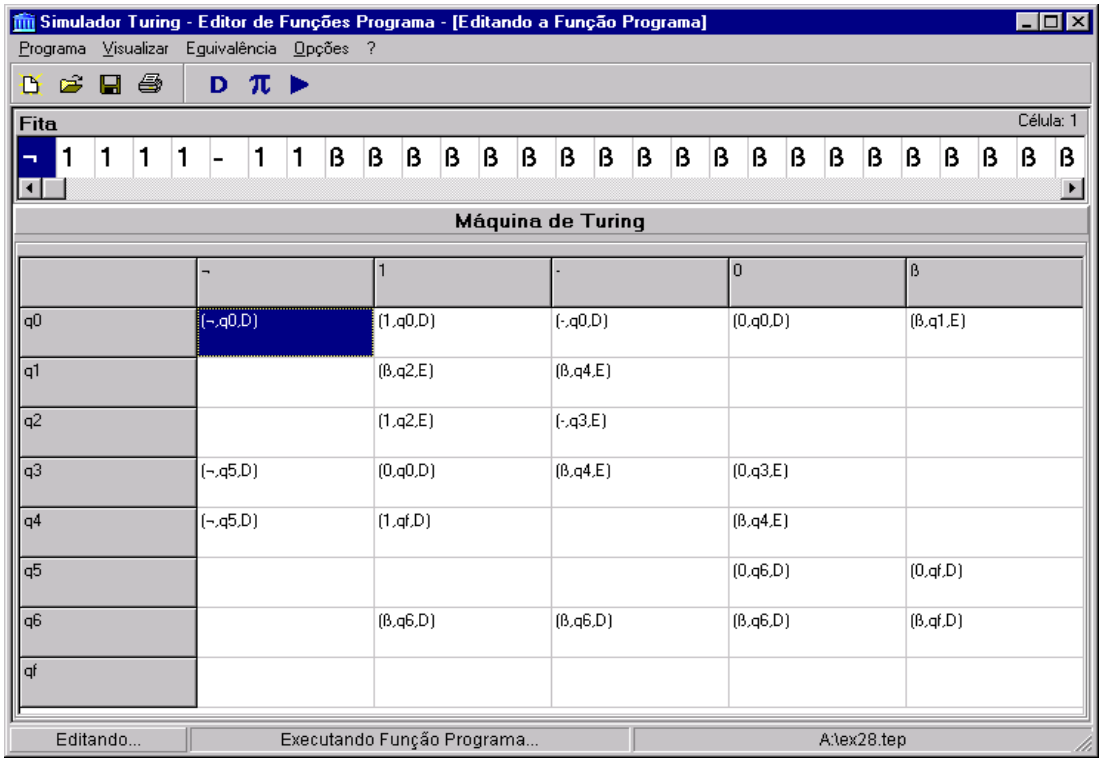
O programa das classes de máquinas, apresentadas acima, é parcial porque para todas máquinas, o conjunto de todas palavras sobre o alfabeto de entrada Σ*, as máquinas vão apresentar conjunto de estados finais REJEITA(M) e LOOP(M) diferentes de ∅.

2. Ex. 28)

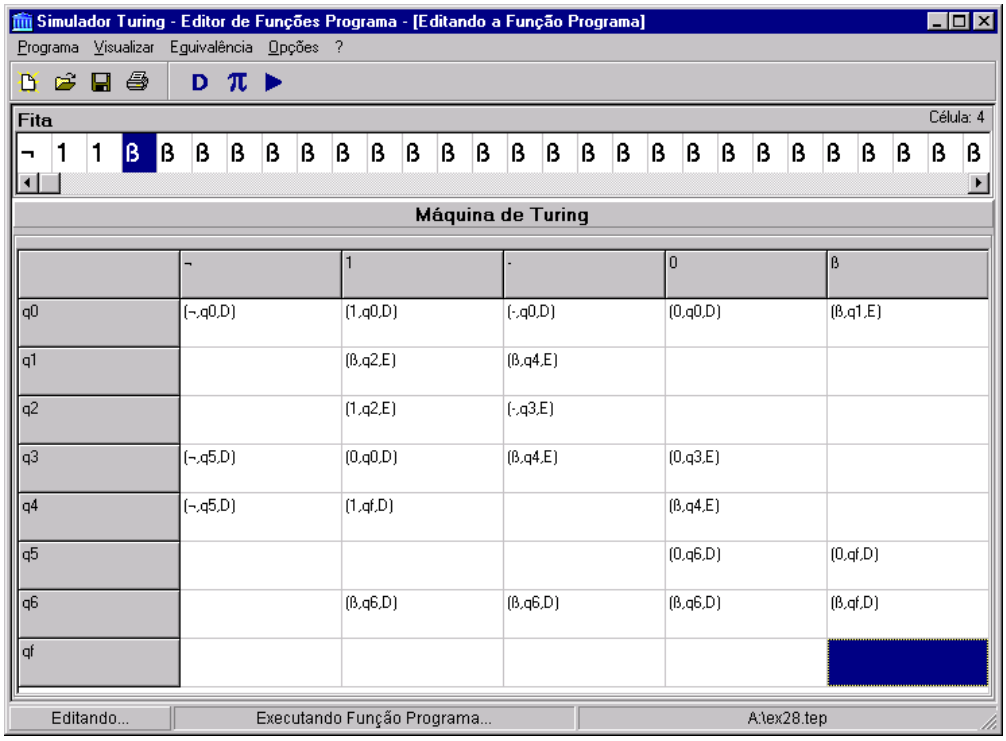
(uma das possíveis máquinas determinísticas)

	¬	1	-	0	ß
q0	(¬,q0,D)	(1,q0,D)	(-,q0,D)	(0,q0,D)	(ß,q1,E)
q1		(ß,q2,E)	(ß,q4,E)		
q2		(1,q2,E)	(-,q3,E)		
q3	(¬,q5,D)	(0,q0,D)	(ß,q4,E)	(0,q3,E)	
q4	(¬,q5,D)	(1,qf,D)		(ß,q4,E)	
q5				(0,q6,D)	(0,qf,D)
q6		(ß,q6,D)	(ß,q6,D)	(ß,q6,D)	(ß,qf,D)
qf					

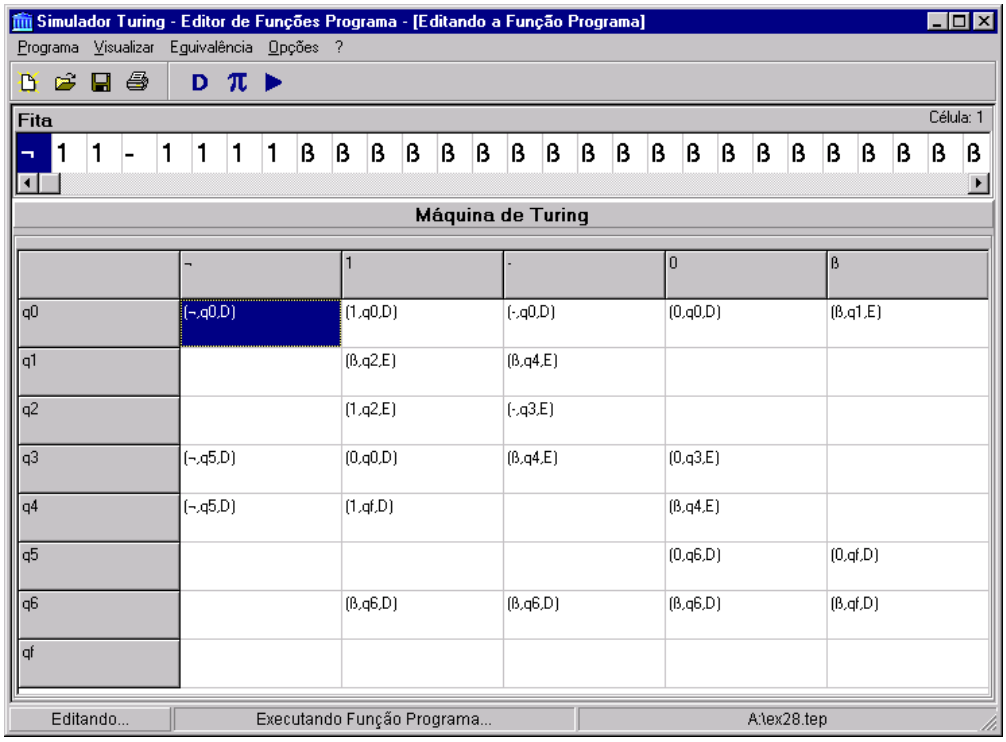
Este programam funciona da seguinte maneira: para cada 1 encontrado depois do – ele substitui por um 0 antes do - , o 1 encontrado é substituido por β. Após este passo, o programa substitui o 0 e o – por β ficando como resposta o número de uns correspondentes a resposta certa. Segue dois exemplos testados no simulador Turing:

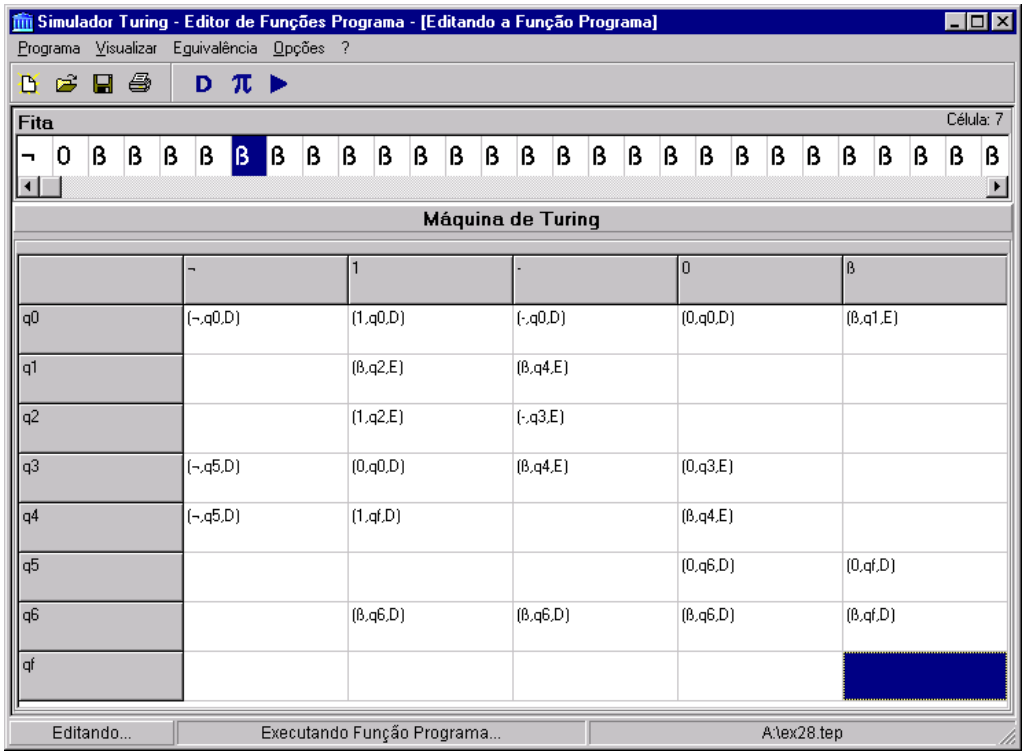


	¬	1	-	0	ß
q0	(¬,q0,D)	(1,q0,D)	(-,q0,D)	(0,q0,D)	(ß,q1,E)
q1		(ß,q2,E)	(ß,q4,E)		
q2		(1,q2,E)	(-,q3,E)		
q3	(¬,q5,D)	(0,q0,D)	(ß,q4,E)	(0,q3,E)	
q4	(¬,q5,D)	(1,qf,D)		(ß,q4,E)	
q5				(0,q6,D)	(0,qf,D)
q6		(ß,q6,D)	(ß,q6,D)	(ß,q6,D)	(ß,qf,D)
qf					



	¬	1	-	0	ß
q0	(¬,q0,D)	(1,q0,D)	(-,q0,D)	(0,q0,D)	(ß,q1,E)
q1		(ß,q2,E)	(ß,q4,E)		
q2		(1,q2,E)	(-,q3,E)		
q3	(¬,q5,D)	(0,q0,D)	(ß,q4,E)	(0,q3,E)	
q4	(¬,q5,D)	(1,qf,D)		(ß,q4,E)	
q5				(0,q6,D)	(0,qf,D)
q6		(ß,q6,D)	(ß,q6,D)	(ß,q6,D)	(ß,qf,D)
qf					


Simulador Turing - Editor de Funções Programa - [Editando a Função Programa]
Programa Visualizar Eguivalência Opções ?
Fita
Célula: 1
¬ 1 1 - 1 1 1 1 ß
Máquina de Turing
¬ 1 - 0 ß
q0 (¬,q0,D) (1,q0,D) (-,q0,D) (0,q0,D) (ß,q1,E)
q1 (ß,q2,E) (ß,q4,E)
q2 (1,q2,E) (-,q3,E)
q3 (¬,q5,D) (0,q0,D) (ß,q4,E) (0,q3,E)
q4 (¬,q5,D) (1,qf,D) (ß,q4,E)
q5 (0,q6,D) (0,qf,D)
q6 (ß,q6,D) (ß,q6,D) (ß,q6,D) (ß,qf,D)
qf
Editando...
Executando Função Programa...
A:\ex28.tep


Simulador Turing - Editor de Funções Programa - [Editando a Função Programa]
Programa Visualizar Eguivalência Opções ?
Fita
Célula: 7
¬ 0 ß
Máquina de Turing
¬ 1 - 0 ß
q0 (¬,q0,D) (1,q0,D) (-,q0,D) (0,q0,D) (ß,q1,E)
q1 (ß,q2,E) (ß,q4,E)
q2 (1,q2,E) (-,q3,E)
q3 (¬,q5,D) (0,q0,D) (ß,q4,E) (0,q3,E)
q4 (¬,q5,D) (1,qf,D) (ß,q4,E)
q5 (0,q6,D) (0,qf,D)
q6 (ß,q6,D) (ß,q6,D) (ß,q6,D) (ß,qf,D)
qf
Editando...
Executando Função Programa...
A:\ex28.tep

2. Ex.29)

A hipótese de Church diz que qualquer função que seja computável é executável em uma máquina de Turing, ou seja, a máquina de Turing é o dispositivo de computação mais geral que existe e qualquer outro dispositivo tem no máximo o mesmo poder do que ela.

A importância dessa hipótese na Teoria da Comutação é o fato de que a máquina de Turing é uma máquina genérica, podendo executar qualquer função computável e, ao mesmo tempo é uma máquina suficientemente simples para permitir estudos e demonstrações .

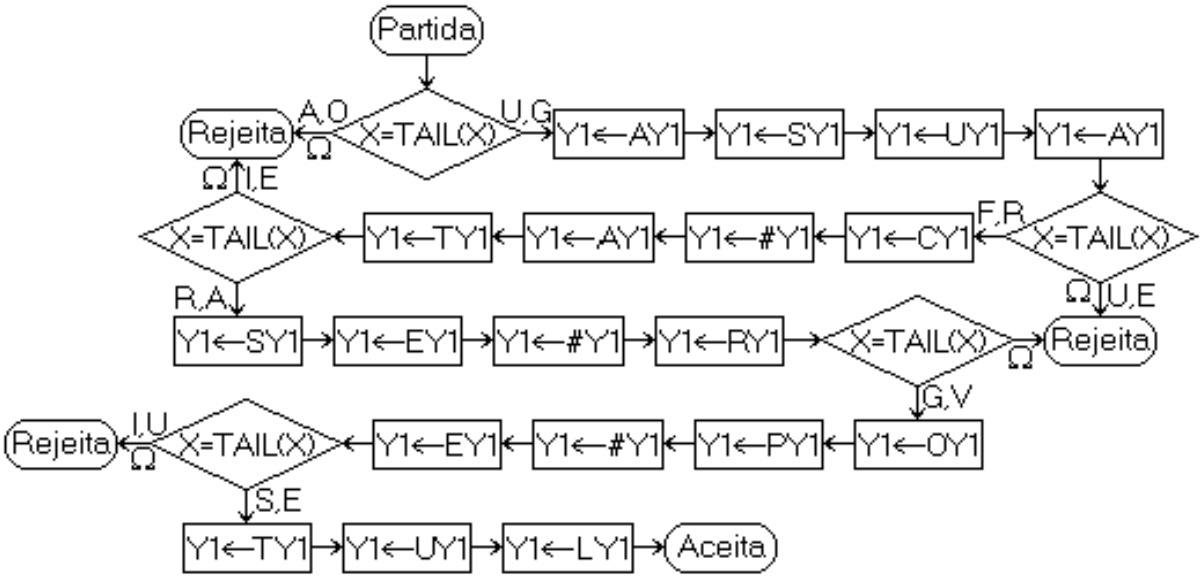
É uma hipótese e não uma teoria ou teorema de Church, pois não é demonstrável, já que, tendo apenas uma definição intuitiva para algoritmos, não se tem como saber exatamente quais são as funções computáveis.

A máquina de Turing é pensada como uma máquina que realiza tarefas muito simples. Na verdades, Turing idealizou a máquina pensando em uma situação na qual uma pessoa X qualquer, equipada com um instrumento de escrita e um apagador, realiza cálculos numa folha de papel, realizando tarefas extremamente simples. Então, a hipótese de Church mostra que uma máquina poderosa não precisa ser complexa. Uma máquina extremamente simples tem este poder.

Por ser simples, demonstrações na máquina de Turing são simples também. Provar que algo é computável pode ser resumido em realizar uma máquina de Turing que faça tal tarefa. Segundo a hipótese, se não puder se criar uma máquina de Turing para computar algo, pode-se dizer que este algo não é computável.

A hipótese de Church não é teorema, porque não foi demonstrada ainda. Isto porque não existe uma definição formal de um algorítmo. A noção de classe computável de algorítmos é intuitiva, e por isso, não há como construir uma prova formal para a hipótese de Chuch. Mas as evidências mostram que Church está certo, pois todas as máquinas construidas até hoje tem, no máximo, o mesmo poder computacional que a Máquina de Turing.

2. Ex. 30)



Computação:

X ⇒ UFRGS → FRGS → RGS → GS
S → Ω

Y ⇒ AUSA → TA#CAUSA → R#ESTA#CAUSA → E#POR#ESTA#CAUSA
→
LUTE#POR#ESTA#CAUSA

Função computada:

LUTE#POR#ESTA#CAUSA

2. Ex.31)

	¬	a	b	A	B	ß
q0	(¬,q0,D)	(A,q1,D)	(b,q0,D)	(A,q0,D)	(B,q0,D)	(ß,q4,E)
q1		(a,q1,D)	(b,q1,D)		(B,q2,E)	(ß,q2,E)
q2	(¬,q5,D)	(a,q2,E)	(B,q3,E)	(A,q2,E)	(B,q2,E)	
q3	(¬,q0,D)	(a,q3,E)	(b,q3,E)	(A,q0,D)		
q4	(¬,q7,D)		(b,q6,D)	(A,q4,E)	(B,q4,E)	
q5		(a,q6,D)		(A,q5,D)	(B,q5,D)	
q6	(¬,q6,D)	(a,q6,D)	(b,q6,D)	(A,q6,D)	(B,q6,D)	(ß,q6,E)
q7						

Seqüência de passos de execução dessa máquina para a palavra “ab”:

q0 | => - a b β

q0 | =>

q1 | => - a b β

q1 | =>

q2 | => - A b β

q1 | => - a b β

q3 | => - A B β

q0 | => - A B β

q0 | => - A B β

q4 | => - A B β

q4 | => - A B β

q7 | - **A** **B** **β**

A máquina de Turing aceitou a palavra “ab”, pois parou em q7, que é o estado de aceitação.

Seqüência de passos de execução dessa máquina para a palavra “aba”:

q0 | =>

-	a	b	a	β

q0 | =>

-	a	b	a	β

q1 | =>

-	A	b	a	β

q1 | =>

-	A	b	a	β

q1 |=>

-	A	b	a	β

q2 | =>

-	A	b	a	β

q2 | =>

-	A	b	a	β

q3 | =>

-	A	B	a	β

q0 | =>

-	A	B	a	β

q1 |=>

-	A	B	A	β

q2 | =>

-	A	B	A	β

q2 | =>

-	A	B	A	β

q2 | =>

-	A	B	A	β

q2 | =>

-	A	B	A	β

q5 | =>

-	A	B	A	β

q5 | =>

-	A	B	A	β

q5 | =>

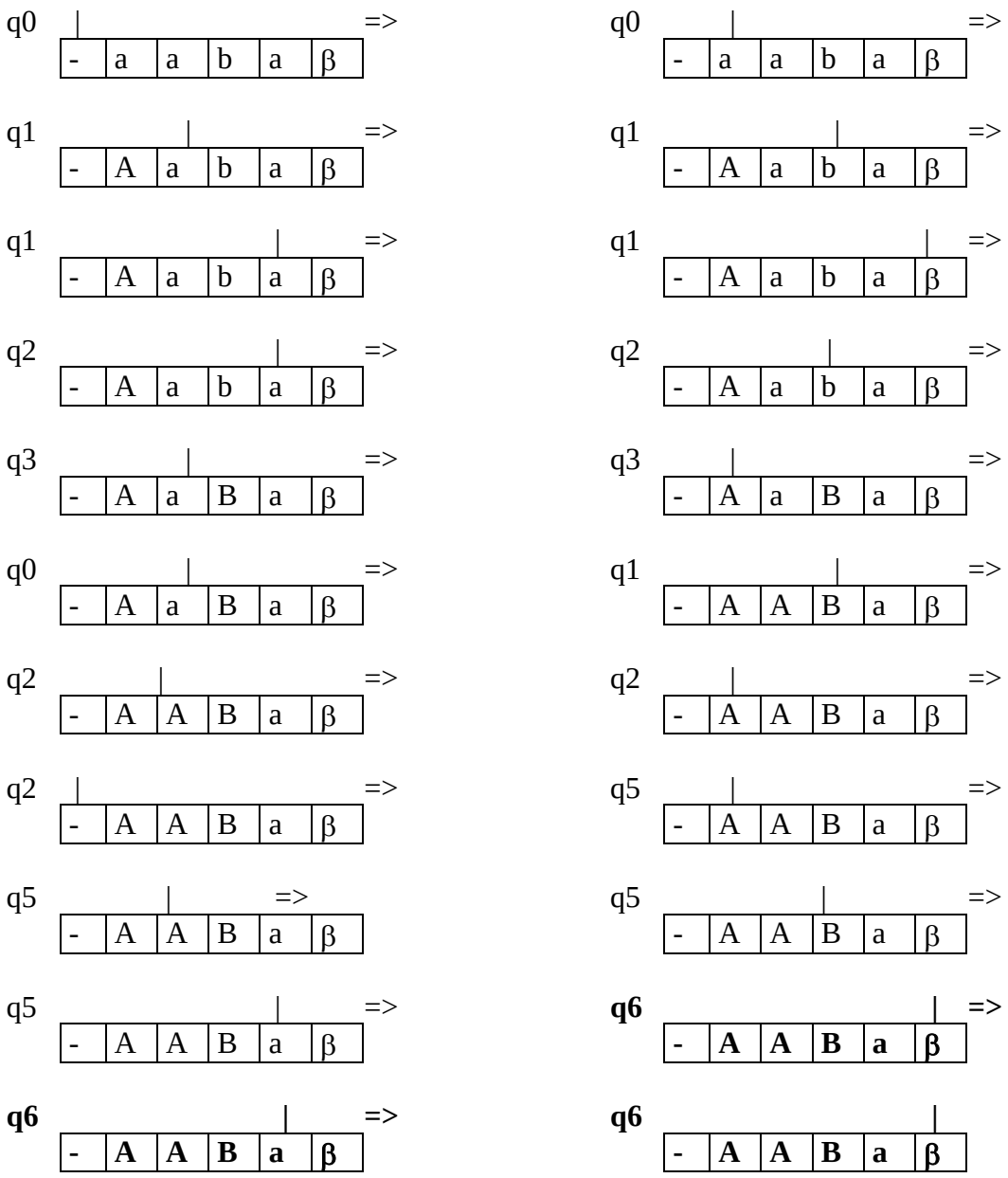
-	A	B	A	β

q5 |

-	**A**	**B**	**A**	**β**

A máquina de Turing rejeitou a palavra “aba”, pois parou em q5, que não é estado de aceitação.

Seqüência de passos de execução dessa máquina para a palavra “aaba”:

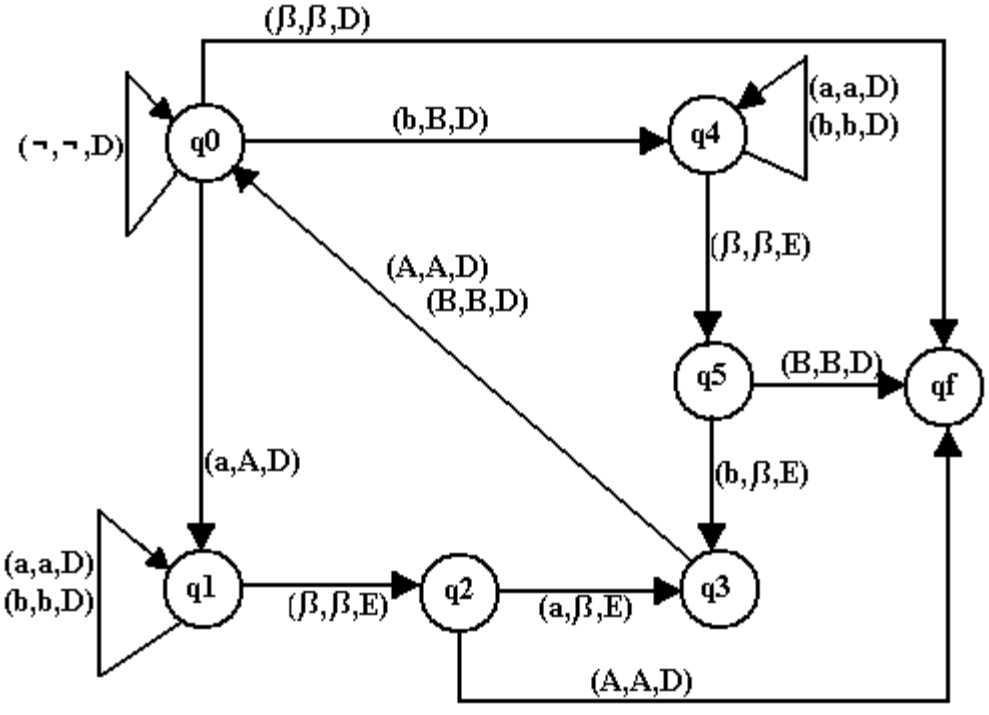


A máquina de Turing entrou em loop com a palavra “aaba”. Isso pode ser notado porque a fita chegou no estado q6 da mesma forma que já havia chegado nele dois passos antes, e assim o fará infinitamente.

2. Ex. 32)

	¬	a	b	A	B	ß
q0	(¬,q0,D)	(A,q1,D)	(B,q4,D)			(ß,qf,D)
q1		(a,q1,D)	(b,q1,D)			(ß,q2,E)
q2		(ß,q3,E)		(A,qf,D)		
q3		(a,q3,E)	(b,q3,E)	(A,q0,D)	(B,q0,D)	
q4		(a,q4,D)	(b,q4,D)			(ß,q5,E)
q5			(ß,q3,E)		(B,qf,D)	
qf						

Grafo de estados :



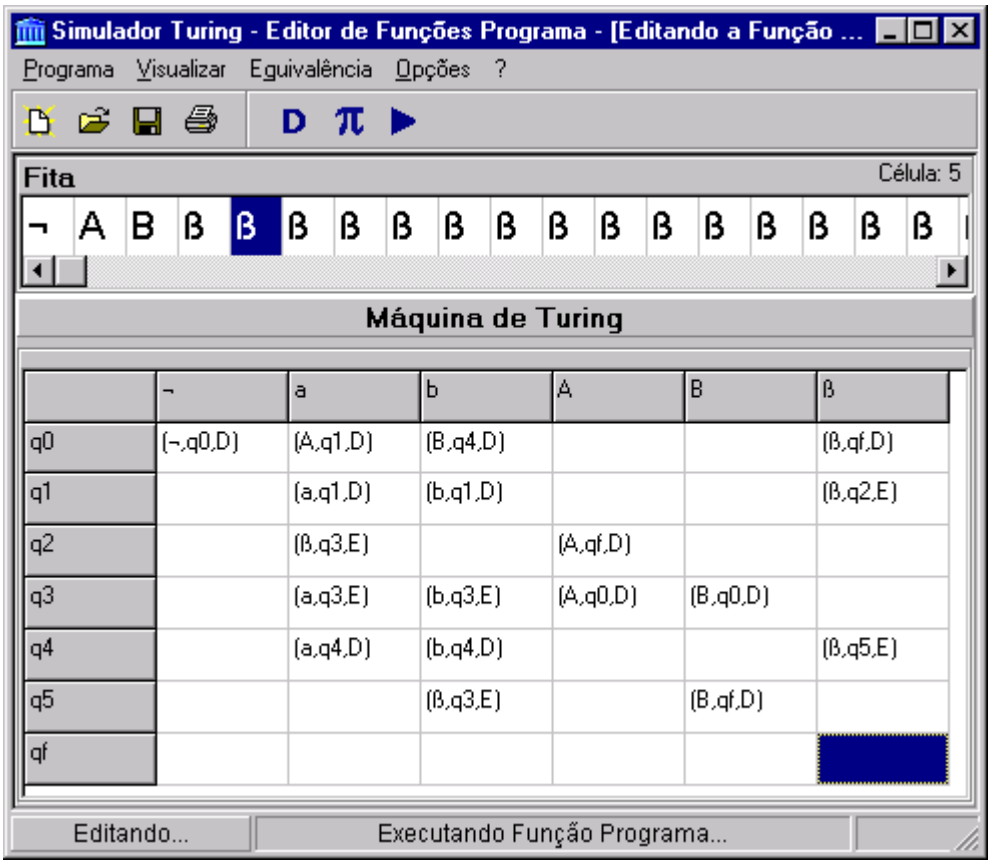
Comentário :

No estado q0 a máquina procura na fita por um a ou b. Ao achar um “a” o transforma em “A” e passa para o estado q1. No estado q1 percorre a fita até achar o

símbolo de branco, ao achar β volta uma posição para a esquerda (para analisar o último símbolo e a máquina) e a máquina vai para o estado q2. No estado q2 é testado o último símbolo. Caso seja um símbolo A vai-se direto para o estado qf (a palavra é aceita), se o símbolo for "a" a máquina de Turing vai para o estado q3, se não for nenhum desses dois símbolos a palavra é rejeitada. O estado q3 é especial porque percorre a fita em direção ao início tanto a procura tanto de um A como de um B, ao achar um desses símbolos volta para o estado q0. Em q0 a fita é percorrida no sentido direito, caso o símbolo encontrado seja β a máquina assume o estado qf (palavra aceita) e a computação termina, se for achado um "a" é feito novamente o caminho até aqui explicado, por outro lado se o símbolo for um b a máquina passa por q4 e q5, estados análogos a q1 e q2 e recai novamente no estado q3.

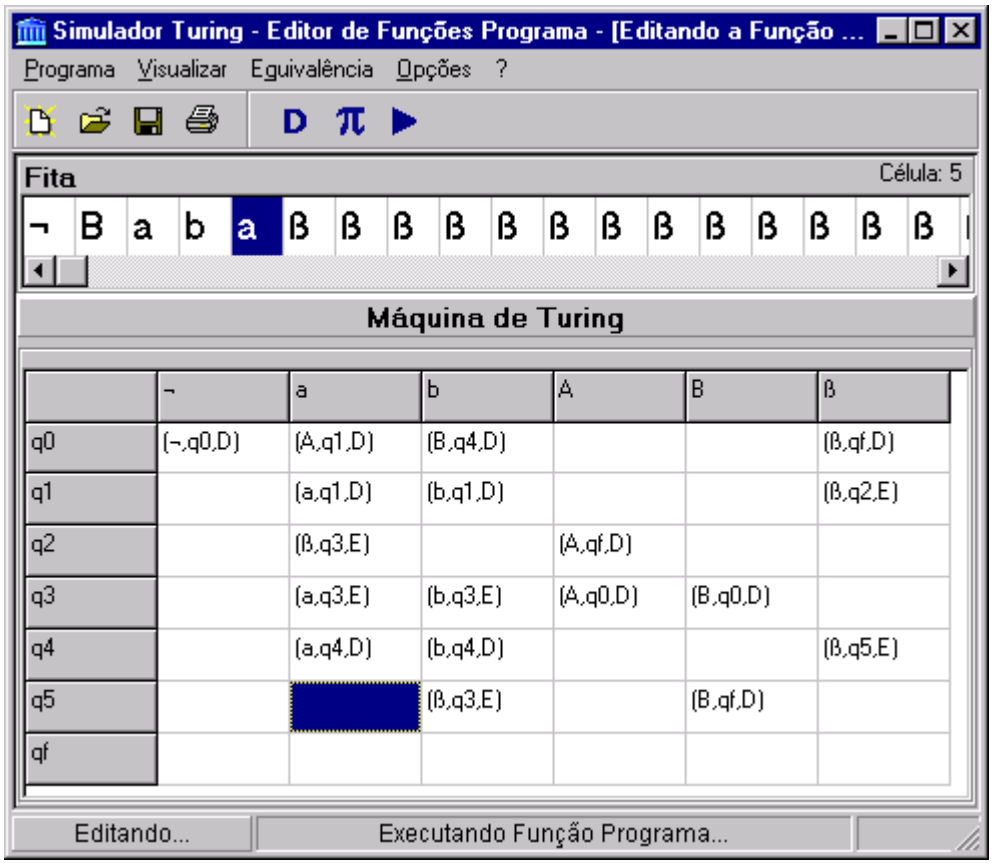
Obs1 : A computação para uma palavra que inicia com b ocorre da mesma forma, mas do estado q0 a máquina passa para o estado q4.
Obs2: A máquina aceita palavra vazia.

Exemplos :



	¬	a	b	A	B	ß
q0	(¬,q0,D)	(A,q1,D)	(B,q4,D)			(ß,qf,D)
q1		(a,q1,D)	(b,q1,D)			(ß,q2,E)
q2		(ß,q3,E)		(A,qf,D)		
q3		(a,q3,E)	(b,q3,E)	(A,q0,D)	(B,q0,D)	
q4		(a,q4,D)	(b,q4,D)			(ß,q5,E)
q5			(ß,q3,E)		(B,qf,D)	
qf						

Palavra de entrada : abba



Palavra de entrada : baba

Conclusão:

A máquina é do tipo ***enumerável recursivamente*** pois aceita as palavras com o mesmo número de ***a*** e ***b***, rejeita nos casos em que o número de ***a*** é maior que o de ***b*** por uma unidade e entra em loop em qualquer outra situação. Se fosse uma máquina do tipo ***recursiva*** não haveria a situação de ***loop***.

2. EX.33)

Máquina M:

M = {Σ, V, β, Q, F, q0, π};

Σ = {a, b};

V = {A, B, ¬, β};

Q = {q0, q1, q2, q3, q4, q5};

q0 = estado inicial;

F = {q5};

ACEITA = {qualquer palavra que inicie com **a** e que após cada **a** exista pelo menos um **b**}, por exemplo ab, abbab;

LOOP = {qualquer palavra onde exista um **b** entre dois **a**}, por exemplo aba, abbaba;

REJEITA = {qualquer outra situação};

Abaixo segue o programa π:

	¬	a	b	A	B	ß
q0	(¬,q0,D)	(A,q1,D)				
q1			(B,q2,D)			
q2		(a,q4,E)	(B,q3,D)			(ß,q5,E)
q3		(A,q1,D)	(B,q3,D)			(ß,q5,E)
q4					(B,q2,D)	
q5						

Exemplos de computação:

ab

q0¬abβ → ¬q0abβ → ¬Aq1bβ → ¬ABq2β → ¬Aq5Bβ → ACEITA;

aba

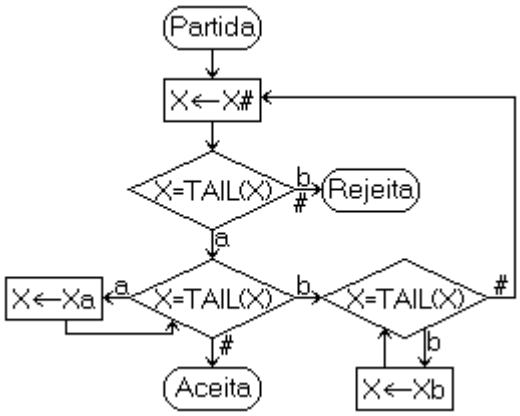
q0¬abaβ → ¬q0abaβ → ¬Aq1baβ → ¬ABq2aβ → ¬Aq4Baβ → ¬ABq2aβ → ¬Aq4Baβ → ... LOOP;

ba

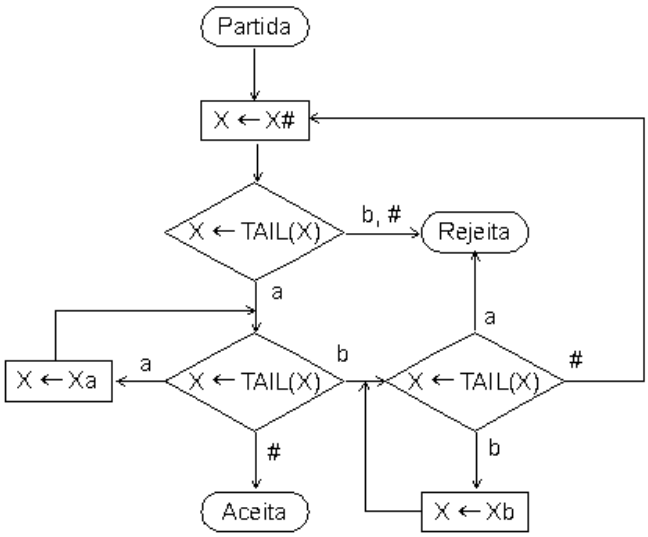
q0¬baβ → ¬q0baβ → REJEITA.

2. EX.34)

Construir uma Máquina de Post M sobre Σ={a, b} que reconheça a linguagem ACEITA(M)={$a^m b^n$ | m > n} e REJEITA(M) = Σ^* - ACEITA(M).

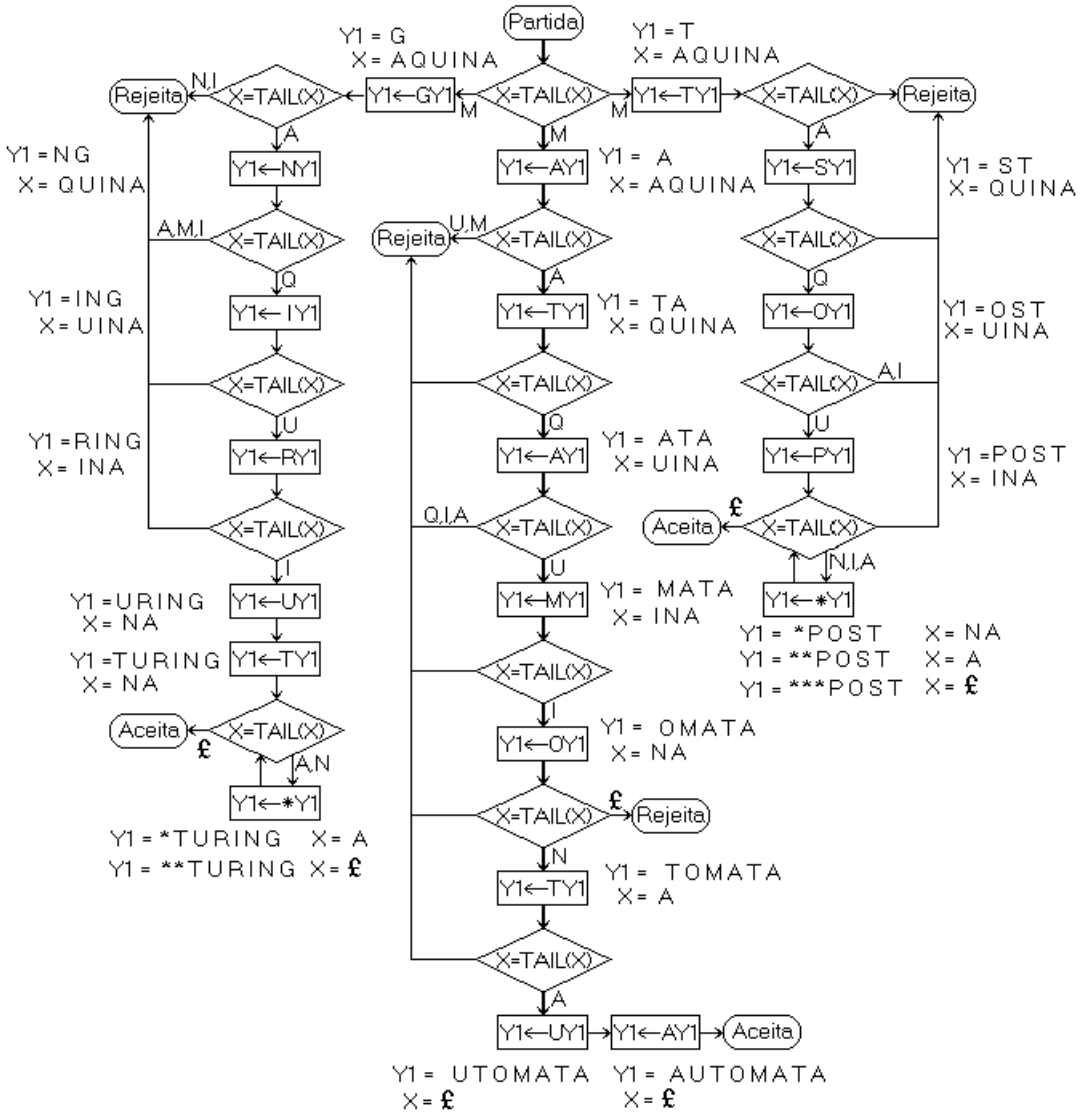


A solução dada não funciona corretamente para todos os valores de entrada. Cada comando de teste deve apresentar uma aresta para cada símbolo do alfabeto além de uma para "#" e outra para "Ω". O teste por "Ω" é desnecessário pois há a concatenação da palavra de entrada com o símbolo "#". A solução mais correta seria:



2. Ex. 35)

Como o automata é não determinístico, fazendo a computação passo a passo temos três caminhos possíveis e, em conseqüência, três funções computadas:



(1) Y1 = **TURING e Y2 = £
(2) Y1 = AUTOMATA e Y2 = £
(3) Y1 = ***POST e Y2 = £

RESPOSTAS DO CAPÍTULO 3

3. Ex. 1)

a) Uma função recursiva (FR) sobre um alfabeto é definida como um número finito de *composições, recursões primitivas* ou *minimizações* sobre 3 funções básicas descritas a seguir sobre $\sum=\{a, b\}$:

Funções básicas

(i) NIL(x) = £ : função constante
(ii) CONSa(x) = ax : concatena "a" à esquerda de x
(iii) CONSb(x) = bx : concatena "b" à esquerda de x

Composição de Funções: Se h, g_1, g_2,..., g_m são funções recursivas, então a função f, que é composição, também o é.
$f(x_1, x_2,...,x_m) = h(g_1(x_1, x_2,...,x_m), g_2(x_1, x_2,...,x_m))$
Isso é valido para as Funções Recursivas Parciais, Totais e Primitivas.

Recursão Primitiva: Se g, h_1, h_2 são funções recursivas, então a função f também o é.
Para n=1) $f(£) = w$
$f(ax) = h_1(x(f(x))$
$f(bx) = h_2(x(f(x))$
Para n≥2) $f(£,x_2,...,x_n) = f(x_2,...,x_n)$
$f(ax_1,x_2,...,x_n) = h_1(x_1,x_2,...,x_n, f(x_1,x_2,...,x_n))$
$f(bx_1,x_2,...,x_n) = h_2(x_1,x_2,...,x_n, f(x_1,x_2,...,x_n))$
Isso é válido para todas as Funções Recursivas, porém nas Parciais f(£) ou $f(£,x_2,...,x_n)$ pode ser indefinida.

Minimização: Se h é função recursiva então a função f que define a menor palavra também o é.
$f(x_1,x_2,...,x_n) = \min(h(x_1,x_2,...,x_n,z))$
Na minimização, excetuam-se as Funções Recursivas Primitivas e é justamente isso que as diferenciam das Funções Recursivas Totais.

b) As funções recursivas primitivas contém todas as operações aritméticas comuns sobre os inteiros e racionais e a maioria das funções de interesse prático. A definição de função recursiva primitiva é idêntica à da função recursiva total, excetuando-se a minimização. As funções recursivas totais definidas sobre as funções básicas, composição e recursão primitiva são funções recursivas primitivas. Assim, as funções recursivas primitivas estão contidas na classe das funções recursivas totais (identifica as funções computáveis que sempre terminam),

que por sua vez estão contidas nas funções recursivas parciais (que podem ter resultados indefinidos na recursão).

c)

Funções recursivas definem a classe das funções que podem ser avaliadas algoritmicamente, ou seja, a classe das funções computáveis. Diz-se que:

- se uma função é recursiva parcial então ela é computável, permitindo os estados aceita, rejeita e loop (há assim uma analogia com as linguagens ER);
- se uma função é recursiva total então ela é computável por uma máquina de Turing e sempre pára, aceitando ou rejeitando (há assim uma analogia com as linguagens recursivas)

A classe das funções primitivas é um subconjunto das recursivas totais. Ela contém todas as operações aritméticas comuns sobre os inteiros e racionais bem como a maioria das funções de interesse prático.
As funções recursivas primitivas estão contidas nas funções recursivas totais, que por sua vez estão contidas nas funções recursivas parciais.
Portanto, em ordem decrescente de complexidade estão : funções recursivas parciais, funções recursivas totais e funções recursivas primitivas.

3. Ex. 2)

a) Adição

Uma Máquina de Turing M = ($\sum$, V, β, Q, F, q_0, π), onde:
$\sum$ = {a, *}
V = {¬, β}
β = símbolo “branco”
Q = {q_0, q_1, q_2, q_3,q_4}
F = {q_4}
q_0 = estado inicial
π = função programa definida a seguir:

	¬	a	*	ß
q0	(¬,q0,D)	(a,q0,D)	(*,q0,D)	(ß,q1,E)
q1		(a,q1,E)	(a,q2,D)	
q2		(a,q2,D)		(ß,q3,E)
q3		(ß,q4,E)		
q4				

Entrada de dados :

Os operandos são representados por a’s , cada “a” valendo uma unidade. Entre os operandos, colocamos o separador “ * ” , a fim de que a máquina os reconheça adequadamente.

Função de cada estado no programa :
q0 : estado inicial, faz com que a cabeça de leitura vá se movendo para a direita até encontrar o símbolo "branco" (β), que indica o fim dos dados de entrada. A cabeça se desloca para a equerda.
q1 : após encontrado o símbolo "branco"em q0, no estado q1 a cabeça de leitura se move sempre para a esquerda da fita, até encontrar o símbolo " * ", e o elimina, escrevendo um "a" em seu lugar. Após, a máquina passa para o estado q2, deslocando a cabeça uma posição para a direita na fita.
q2 : em q2 a cabeça se desloca para a direita da fita, procurando o símbolo "branco". Quando o encontra, passamos para o estado q3, e voltamos uma posição à esquerda na fita.
q3 : agora, no lugar do " a ", escrevemos "β" e a máquina passa ao estado q4, que finaliza a computação da função programa adição.

b) multiplicação
Uma Máquina de Turing M = (∑, V, β, Q, F, q_0, π), onde:
∑ = {a, *}
V = {¬, β, x, y, z}
β = símbolo "branco"
Q = {q_0, q_1, q_2, q_3, q_4, q_5, q_6, q_7, q_8, q_9, q_{10}}
F = {q_{10}}
q_0 = estado inicial
π = função programa definida a seguir:

	¬	a	*	x	y	z	ß
q0	(¬,q0,D)	(z,q1,D)	(z,q6,D)				
q1		(a,q1,D)	(*,q2,D)				
q2		(y,q3,D)		(x,q5,E)	(y,q2,D)		
q3		(a,q3,D)		(x,q3,D)			(x,q4,E)
q4		(a,q4,E)		(x,q4,E)	(y,q2,D)		
q5		(a,q5,E)	(*,q5,E)		(a,q5,E)	(z,q0,D)	
q6		(z,q6,D)		(z,q7,E)		(z,q6,D)	(ß,q9,E)
q7	(¬,q8,D)	(a,q8,D)				(z,q7,E)	
q8						(a,q6,D)	
q9		(a,q10,D)				(ß,q9,E)	
q10							

c) quadrado
Uma Máquina de Turing M = (Σ , V, β , Q, F, q0, π), onde:

Σ = {a}

V = {¬, β , * , x , y , z }

β = símbolo "branco"

Q = {q0, q1, q2, q3,q4, q5, q6, q7, q8, q9, q10, q11, q12, q13, q14, q15}

F = {q15}

q0 = estado inicial

π = função programa definida a seguir:

	¬	a	*	x	y	z	ß
q0	(¬,q0,D)	(y,q1,D)	(*,q4,E)				(ß,q15,E)
q1		(a,q1,D)	(*,q2,D)				(*,q2,D)
q2		(a,q2,D)					(a,q3,E)
q3		(a,q3,E)	(*,q3,E)		(y,q0,D)		
q4	(¬,q5,D)				(a,q4,E)		
q5		(z,q6,D)	(z,q11,D)				
q6		(a,q6,D)	(*,q7,D)				
q7		(y,q8,D)		(x,q10,E)	(y,q7,D)		
q8		(a,q8,D)		(x,q8,D)			(x,q9,E)
q9		(a,q9,E)		(x,q9,E)	(y,q7,D)		
q10		(a,q10,E)	(*,q10,E)		(a,q10,E)	(z,q5,D)	
q11		(z,q11,D)		(z,q12,E)		(z,q11,D)	(ß,q14,E)
q12	(¬,q13,D)	(a,q13,D)				(z,q12,E)	
q13						(a,q11,D)	
q14		(a,q15,D)				(ß,q14,E)	
q15							

Computação com as entradas ‘a’ e ‘aa’, respectivamente, para provar o funcionamento do programa:

(q0)¬aß ⇒ ¬(q0)aß ⇒ ¬y(q1)ß ⇒ ¬y(q2)ß ⇒ ¬y(q3)*aß ⇒ ¬(q3)y*aß ⇒ ¬y(q0)*aß ⇒ ¬(q4)y*aß ⇒ (q4)¬a*aß ⇒ ¬(q5)a*aß ⇒ ¬z(q6)*aß ⇒ ¬z*(q7)aß ⇒ ¬z*y(q8)ß ⇒ ¬z*(q9)yxß ⇒ ¬z*y(q7)xß ⇒ ¬z*(q10)yxß ⇒ ¬z(q10)*axß ⇒ ¬(q10)z*axß ⇒ ¬z(q5)*axß ⇒ ¬zz(q11)axß ⇒ ¬zzz(q11)xß ⇒ ¬zz(q12)zzß ⇒ ¬z(q12)zzzß ⇒ ¬(q12)zzzzß ⇒ (q12)¬zzzzß ⇒ ¬(q13)zzzzß ⇒ ¬a(q11)zzzß ⇒ ¬az(q11)zzß ⇒ ¬azz(q11)zß ⇒ ¬azzz(q11)ß ⇒ ¬azz(q14)zß ⇒ ¬az(q14)zß ⇒ ¬a(q14)zß ⇒ ¬(q14)aß ⇒ ¬a(q15)ß*
Computação Finalizada!

(q0)¬aaß ⇒ ¬(q0)aaß ⇒ ¬y(q1)aß ⇒ ¬ya(q1)ß ⇒ ¬ya(q2)ß ⇒ ¬ya(q3)*aß ⇒ ¬y(q3)a*aß ⇒ ¬(q3)ya*aß ⇒ ¬y(q0)a*aß ⇒ ¬yy(q1)*aß ⇒ ¬yy*(q2)aß ⇒ ¬yy*a(q2)ß ⇒ ¬yy*(q3)aaß ⇒ ¬yy(q3)*aaß ⇒ ¬y(q3)y*aaß ⇒ ¬yy(q0)*aaß ⇒ ¬y(q4)y*aaß ⇒ ¬(q4)ya*aaß ⇒ (q4)¬aa*aaß ⇒ ¬(q5)aa*aaß ⇒ ¬z(q6)a*aaß ⇒ ¬za(q6)*aaß ⇒ ¬za*(q7)aaß ⇒ ¬za*y(q8)aß ⇒ ¬za*ya(q8)ß ⇒ ¬za*y(q9)axß ⇒ ¬za*(q9)yaxß ⇒ ¬za*y(q7)axß ⇒ ¬za*yy(q8)xß ⇒ ¬za*yyx(q8)ß ⇒ ¬za*yy(q9)xxß ⇒ ¬za*y(q9)yxxß ⇒ ¬za*yy(q7)xxß ⇒ ¬za*y(q10)yxxß ⇒ ¬za*(q10)yaxxß ⇒ ¬za(q10)*aaxxß ⇒ ¬z(q10)a*aaxxß ⇒ ¬(q10)za*aaxxß ⇒ ¬z(q5)a*aaxxß ⇒ ¬zz(q6)*aaxxß ⇒ ¬zz*(q7)aaxxß ⇒ ¬zz*y(q8)axxß ⇒ ¬zz*ya(q8)xxß ⇒ ¬zz*yax(q8)xß ⇒ ¬zz*yaxx(q8)ß ⇒ ¬zz*yax(q9)xxß ⇒ ¬zz*ya(q9)xxxß ⇒ ¬zz*y(q9)axxxß ⇒ ¬zz*(q9)yaxxxß ⇒ ¬zz*y(q7)axxxß ⇒ ¬zz*yy(q8)xxxß ⇒ ¬zz*yyx(q8)xxß ⇒ ¬zz*yyxx(q8)xß ⇒ ¬zz*yyxxx(q8)ß ⇒ ¬zz*yyxx(q9)xxß ⇒ ¬zz*yyx(q9)xxxß ⇒ ¬zz*yy(q9)xxxxß ⇒ ¬zz*y(q9)yxxxxß ⇒ ¬zz*yy(q7)xxxxß ⇒ ¬zz*y(q10)yxxxxß ⇒ ¬zz*(q10)yaxxxxß ⇒ ¬zz(q10)*aaxxxxß ⇒ ¬z(q10)z*aaxxxxß ⇒ ¬zz(q5)*aaxxxxß ⇒ ¬zzz(q11)aaxxxxß ⇒ ¬zzzz(q11)axxxxß ⇒ zzzzz(q11)xxxxß ⇒ ¬zzzz(q12)zzxxxß ⇒ ¬zzz(q12)zzzxxxß ⇒ ¬zz(q12)zzzzxxxß ⇒ ¬z(q12)zzzzzxxxß ⇒ ¬(q12)zzzzzzxxxß ⇒ (q12)¬zzzzzzxxxß ⇒ ¬(q13)zzzzzzxxxß ⇒ ¬a(q11)zzzzzxxxß ⇒ ¬az(q11)zzzzxxxß ⇒ ¬azz(q11)zzzxxxß ⇒ ¬azzz(q11)zzxxxß ⇒ ¬azzzz(q11)zxxxß ⇒ ¬azzzzz(q11)xxxß ⇒ ¬azzzz(q12)zzxxß ⇒ ¬azzz(q12)zzzxxß ⇒ ¬azz(q12)zzzzxxß ⇒ ¬az(q12)zzzzzxxß ⇒ ¬a(q12)zzzzzzxxß ⇒ ¬(q12)azzzzzzxxß ⇒ ¬a(q13)zzzzzzxxß ⇒ ¬aa(q11)zzzzzxxß ⇒ ¬aaz(q11)zzzzxxß ⇒ ¬aazz(q11)zzzxxß ⇒ ¬aazzz(q11)zzxxß ⇒ ¬aazzzz(q11)zxxß ⇒ ¬aazzzzz(q11)xxß ⇒ ¬aazzzz(q12)zzxß ⇒ ¬aazzz(q12)zzzxß ⇒ ¬aazz(q12)zzzzxß ⇒ ¬aaz(q12)zzzzzxß ⇒ ¬aa(q12)zzzzzzxß ⇒ ¬a(q12)azzzzzzxß ⇒ ¬aa(q13)zzzzzzxß ⇒ ¬aaa(q11)zzzzzxß ⇒ ¬aaaz(q11)zzzzxß ⇒ ¬aaazz(q11)zzzxß ⇒ ¬aaazzz(q11)zzxß ⇒ ¬aaazzzz(q11)zxß ⇒ ¬aaazzzzz(q11)xß ⇒ ¬aaazzzz(q12)zzß ⇒ ¬aaazzz(q12)zzzß ⇒ ¬aaazz(q12)zzzzß ⇒ ¬aaaz(q12)zzzzzß ⇒ ¬aaa(q12)zzzzzzß ⇒ ¬aa(q12)azzzzzzß ⇒ ¬aaa(q13)zzzzzzß ⇒ ¬aaaa(q11)zzzzzß ⇒ ¬aaaaz(q11)zzzzß ⇒ ¬aaaazz(q11)zzzß ⇒ aaaazzz(q11)zzß ⇒ ¬aaaazzzz(q11)zß ⇒ ¬aaaazzzzz(q11)ß ⇒ ¬aaaazzzz(q14)zß ⇒ ¬aaaazzz(q14)zß ⇒ ¬aaaazz(q14)zß ⇒ ¬aaaaz(q14)zß ⇒ ¬aaaa(q14)zß ⇒ ¬aaa(q14)aß ⇒ ¬aaaa(q15)ß*
Computação Finalizada!

c) fatorial (n!)

Uma Máquina de Turing M = (∑, V, β, Q, F, q_0, π), onde:
∑ = {a}
V = {¬, β, x, y, z,1}
β = símbolo "branco"
Q = {q_0, q_1, q_2, q_3, q_4, q_5, q_6, q_7, q_8, q_9, q_{10}, q_{11}, q_{12}, q_{13}, q_{14}, q_{15}, q_{16}, q_{17}, q_{18}, q_{19}, q_{20}, q_{21}, q_{22}}
F = {q_{22}}
q_0 = estado inicial
π = função programa definida a seguir:

	¬	a	x	y	z	1	ß
q0	(¬,q0,D)	(a,q1,D)					(a,q22,E)
q1		(a,q1,D)					(x,q2,E)
q2	(¬,q3,D)	(a,q2,E)	(x,q2,E)			(1,q2,E)	
q3		(1,q4,D)	(x,q5,E)			(1,q3,D)	
q4		(a,q4,D)	(x,q4,D)			(1,q4,D)	(1,q2,E)
q5						(y,q6,E)	
q6	(¬,q18,D)					(y,q7,E)	
q7	(¬,q18,D)					(z,q8,D)	
q8		(a,q11,E)	(x,q8,D)	(y,q8,D)	(z,q8,D)	(z,q9,D)	
q9		(a,q9,D)				(1,q9,D)	(a,q10,E)
q10		(a,q10,E)			(z,q8,D)	(1,q10,E)	
q11			(x,q12,E)		(1,q11,E)		
q12	(¬,q13,D)			(y,q12,E)	(z,q12,E)	(z,q8,D)	
q13			(x,q14,D)	(y,q13,D)	(z,q13,D)		
q14		(1,q14,D)				(1,q14,D)	(ß,q15,E)
q15			(x,q15,E)	(y,q15,E)	(y,q16,E)	(1,q15,E)	
q16	(¬,q16,D)			(y,q17,E)	(1,q16,E)	(1,q16,D)	
q17	(¬,q18,D)				(z,q8,D)	(z,q8,D)	
q18			(x,q18,D)	(y,q18,D)		(x,q19,E)	(ß,q21,E)
q19	(¬,q20,D)	(a,q20,D)	(x,q19,E)	(y,q19,E)			
q20		(a,q20,D)	(a,q18,D)	(a,q18,D)			
q21		(a,q22,E)	(ß,q21,E)				
q22							

3. Ex. 3)

a) ESQUERDA (*x*): retorna o primeiro símbolo de *x*;

A função

ESQUERDA (£) = £;
ESQUERDA (*ax*) = *a;*
ESQUERDA (*bx*) = *b*;

pode ser dita recursiva, por obedecer a definição de uma FR, na forma de uma função básica.

A defínção diz o seguinte:

Esquerda retorna o símbolo mais a esquerda de uma dada String, ou seqüência de símbolos. Então, ax, onde x é uma string que contém somente “a”s e “b”s, tem como símbolo a esquerda o a. Para bx, usando o mesmo raciocínio, o símbolo mais a esquerda de bx é “b”.

b) DIREITA (*x*): retorna o último símbolo de *x*;
DIREITA (*x*) = COND(TAIL (*x*), x, ESQUERDA(REVERSO (*x*)))

c) CONCREVERSO (*x*): retorna a concatenação de *x* com o reverso de *x*;

A função

CONCREVERSO (£) = £;
CONCREVERSO (*x*) = CONC(*x*, REVERSO (*x*));

pode ser dita recursiva, por obedecer a definição de uma FR, na forma de uma função básica.

A defingão diz o seguinte:

O concreverso de uma palavra vazia é uma palavra vazia. No caso de x não ser uma palavra vazia, podemos observar que foi criada uma função Conc, que retorna a concatenação do primeiro argumento com o segundo argumento, e foi também criada a função reverso, que retorna a palavra reversa de seu argumento, ou seja, a última letra passa a ser a primeira, e assim por diante.

d) CONCPREFIXO (*x*) = retorna a concatenação de todos os prefixos possíveis de *x*:

CONCPREFIXO (£) = £;
CONCPREFIXO (*x*) = CONC(*x*, CONCPREFIXO(REVERSO(TAIL(REVERSO(*x*))))

Uma Função Recursiva (FR) sobre um alfabeto é definida como um número finito de composições, recursões e minimizações sobre três funções básicas.

A função vai empilhando todos os possíveis prefixos de x e quando chega a condição de parada da recursão, volta desempilhando e juntando os prefixos, tenho como resultado a concatenação de todos os prefixos possíveis de x.

e) MÓDULO (x): retorna o número de símbolos de x; o resultado deve ser em unário (sobre {*a*}

{ Define que para uma entrada vazia o resultado será vazio }
MÓDULO (£) = £;

{ A função abaixo concatena ´as´ até que a palavra x seja vazia, que é o ponto de parada da recursividade. O número de ´as´ concatenados será igual ao número de símbolos que havia em *x }*
MÓDULO (*x*) = CONC(*a*, MÓDULO (TAIL(*x*)))

3. Ex. 4)

a) ADIÇÃO (a^i,a^j) = a^{i+j} ;
ADIÇÃO(□,□)=□;
ADIÇÃO (a^i,a^j)= CONC (a^i,a^j);

EXPLICAÇÕES:

1)Uma função recursiva sobre um alfabeto é definido como um número finito de composições, recursões primitivas e minimizações sobre três funções básicas:
NIL(x)= □;
CONSa(x)=ax (concatena a à esquerda de x);
CONSb(x)=bx (concatena b à esquerda de x);

2)Composição :
Se h, g1, g2 ,..., gm são funções recursivas, então a função f, também é uma função recursiva. As funções g podem ter argumentos xi ausentes.
f(x1,x2,...,xn)= h(g1(x1,x2,...,xn), g2(x1,x2,xn))

3)Recursão primitiva

Se g, h1 e h2 são funções recursivas, então a função f, também é uma função recursiva:

f(□)=w;
f(ax)= h1(x,f(x));
f(bx)= h2(x,f(x));

A definição formal de uma função recursiva primitiva é idêntica à da função recursiva total, excetuando-se a minimização . Portanto ***qualquer função total definida sobre as funções básicas, composição e recursão primitiva é recursiva primitiva.***
As seguintes funções são ***primitivas recursivas: constante,identidade, TAIL,reverso, condicional e antecessor.***

NESTE EXERCÍCIO FOI UTILIZADO O CONCEITO DE RECURSÃO PRIMITIVA PARA PROVAR QUE TAL FUNÇÃO É RECURSIVA PRIMITIVA.

b)

SUBTRAÇÃO (£, £) = £;
SUBTRAÇÃO (a^i, £) = a^i;
SUBTRAÇÃO (£, a^j) = £;
SUBTRAÇÃO (a^i, a^j) = SUBTRAÇÃO(TAIL(a^i), TAIL(a^j));

Usando-se recursão primitiva, com dois parâmetros (n=2), cria-se a função subtração. Em sua própria chamada, SUBTRAÇÃO, recebe como parâmetro TAIL(x). Sendo TAIL(£) = £, TAIL(ax) = x e TAIL(bx) = x.

c)

MULTIPLICAÇÃO (a^i,a^j) = a^{i*j} ;
MULTIPLICAÇÃO (a^i,a^j) = a^{i*j} ;
MULTIPLICAÇÃO(a^i,□)=□;
MULTIPLICAÇÃO (a^i,a^j)= CONC (a^i,multiplicação(a^i , TAIL(a^i,a^j)));

Explicação:

Neste exercício foi utilizada a recursão primitiva e a composicão para se provar que a multiplicação é uma função recursiva primitiva. Maiores detalhes sobre os

conceitos de recursão primitiva e recursão já foram abordados no Ex. 04, letra ª a (arquivo E304c02).

d) FATORIAL (a^i) = $a^{i*(i-1)*....*1}$
FATORIAL (£) = *a*;
FATORIAL (a^i) = MULTIPLICAÇÃO(a^i, FATORIAL (TAIL (a^i)))

3. Ex. 5)

a) ORDEM_LEXICOGRÁFICA (x): ordem lexicográfica de x sobre $\{a,b\}$; o resultado deve ser unário (sobre $\{a\}$);
ORDEM_LEXICOGRÁFICA (£) = £;
ORDEM_LEXICOGRÁFICA (x) = CONC(a, ORDEM_LEXICOGRÁFICA(ANT(x)));
O resultado ser unário sobre {a} significa que se a ordem da palavra for *i* o resultado será a a^i. Se a entrada for £ a função retorna £, caso contrário, retornará um "a" concatenado com o resultado da chamada da função com a palavra de ordem anterior. OGRÁFICA (*x*) = CONC(a, ORDEM_LEXICOGRÁFICA(ANT(*x*)));

b) VALOR (a^i) = retorna a *i-ésima* palavra na ordem lexicográfica;

Uma Função Recursiva (FR) sobre um alfabeto é definida como um número finito de composições, recursões e minimizações sobre três funções básicas.

A diferença entre funções recursivas parciais e totais é que para funções recursivas parciais, f($\Im$) ou f($\Im$,x_2,...,x_n) na recursão pode ser indefinida.

VALOR (£) = £;
VALOR (*x*) = SUCC(VALOR(TAIL(*x*)));

A definição formal de uma função recursiva primitiva é idêntica à da função recursiva total, excetuando-se a minimização. Portanto, qualquer função total definida sobre as funções básicas, composição e recursão primitiva é recursiva primitiva. A função TAIL e a sucessor são recursivas primitivas, conforme demonstrado na apostila.

A classe das funções primitivas está contida propriamente na classe das funções recursivas totais.

c) PALÍNDROME (*x*) = verifica se *x*, sobre $\{a,b\}$, é palíndrome. O resultado dever ser "*a*" para o caso negativo e £ para o caso afirmativo.
PALÍNDROME (*x*) = PAL(*x*, *x*)
PAL (£, *x*) = a;
PAL (*x*, £) = a;
PAL (£, £) = £;
PAL (*x*, *x*) = PAL(ANT(*x*), REVERSO(ANT(*x*));

Como a função está definida nos casos PAL(£, ...) na recursão, então, é recursiva total.

d) ORDEM (x_1, x_2): verifica se x_1 antecede x_2 na ordem lexicográfica sobre $\{a,b\}$; o resultado dever ser "£" para $x_1 = x_2$, "*a*" para $x_1 > x_2$ e "*b*" para $x_1 < x_2$;

ORDEM(£,£) = £;
ORDEM(x1,£) = a;
ORDEM(£,x2) = b;
ORDEM(x1,x2) = ORDEM(ANT(x1),ANT(x2));

Sugiro, em primeiro lugar, uma explicação mais detalhada de como funciona a função. Por exemplo numa linguagem "para leigos", como segue:

Executa-se a função com seus argumentos iniciais;
Se nenhum for = £, então
Executa-se a função, agora com os antecessores de seus argumentos iniciais;
E assim sucessivamente e recursivamente
até que um, ou os dois, seja(m) = £;

3. Ex. 6)

a) Definição Recursiva da Adição:

$+(x, y) = x + y;$

$+(x,1) = S(U_1^2(x,1));$

$+(x, y+1) = +(S(U_1^2(x, y), y);$

Depois de definida a função, poderia ser pedido ao leitor que fizesse uma soma (fornecendo a ele os valores, é claro) e verificasse o tempo todo os valores em U e a recursão.

b) Multiplicação:

$\times(x, y) = x \times y;$

$\times(x,1) = U_1^2(x,1);$

$\times(x, y+1) = +(U_1^2(x, y), \times(x, y));$

No primeiro item a multiplicação é representada na notação posfix. No segundo item é definida a identidade, ou seja qualquer que seja a entrada a saída será o x. Já no terceiro item é definida recursivamente a multiplicação.
Ex :

3 X 5 :

X (3,4+1) = +(3, X(3,4));

+(3, X(3,3));

+(3, X(3,2));

+(3, X(3,2));

+(3, X(3,1));

Como X(3,1) está definido vai se desempilhando até chegar-se na resposta.

Ao meu ver seria mais fácil a visualização se a recursividade fosse definida da seguinte forma :

X (x,y) = + (U(x,y), X(x,y-1);

Aplicando-se ao exemplo anterior :

3 X 5 :

X (3,5) = +(3, X(3,4));

+(3, X(3,3));

+(3, X(3,2));

+(3, X(3,2));

+(3, X(3,1));

Ou seja, o resultado final seria o mesmo e a recursividade ficaria mais evidente.

c) Exponenciação:

$e(x, y) = x^y$;

$e(x,1) = U_1^2(x,1)$;

$e(x, y+1) = \times(U_1^2(x, y), e(x, y))$

para esta terceira sentença, temos que

$e(x,y+1) = x^{y+1} = x.x^y = \times(x,x^y) = \times(U_1^2(x, y), e(x, y))$

d) Fatorial:

Temos fatorial de x definido como:

Para x = 0 → Fat(0) = 1;

Para x ≥ 1 → Fat(x) = x · (x-1) ... ·1

mas $Fat(Z(x)) = Fat(0) = 1 = S(Z(x)) = U_1^1(S(Z(x)))$

e também $Fat(x+1) = Fat(S(x)) = S(x) \cdot Fat(x) = \times(S(x), Fat(x)) = \times(x+1, Fat(x))$

3. Ex.7)

Ack (2, 1)	: X,Y ≠ 0 ,então usa-se a equação Ack(m,n) =Ack(m-1,Ack(m,n-1));
⇒Ack (1, Ack (2, 0))	: Y = 0 ,então utiliza-se a equação Ack(m,0) = Ack(m-1,1);
⇒ Ack(1, 1)	: X,Y ≠ 0 ,então usa-se a equação Ack(m,n) =Ack(m-1,Ack(m,n-1));
⇒ Ack (0, Ack (1, 0))	: Y = 0 ,então utiliza-se a equação Ack(m,0) = Ack(m-1,1);
⇒ Ack (0, 1)	: X = 0, então é utilizada a equação Ack(0,n) = n+1;
⇒ 2	: volta ao estágio anterior da recursão;
⇒ Ack (0, 2)	: X = 0, então é utilizada a equação Ack(0,n) = n+1;
⇒ 3	: volta ao estágio anterior da recursão;
⇒ Ack (1, 3)	: X,Y ≠ 0 ,então usa-se a equação Ack(m,n) =Ack(m-1,Ack(m,n-1));
⇒ Ack (0, Ack (1, 2))	: X,Y ≠ 0 ,então usa-se a equação Ack(m,n) =Ack(m-1,Ack(m,n-1));
⇒ (0, Ack (1, 1))	: X,Y ≠ 0 ,então usa-se a equação Ack(m,n) =Ack(m-1,Ack(m,n-1));
⇒ Ack (0, Ack (1, 0))	: Y = 0 ,então utiliza-se a equação Ack(m,0) = Ack(m-1,1);
⇒ Ack (0, 1)	: X = 0, então é utilizada a equação Ack(0,n) = n+1;
⇒ 2	: volta ao estágio anterior da recursão;
⇒ Ack (0, 2)	: X = 0, então é utilizada a equação Ack(0,n) = n+1;
⇒ 3	: volta ao estágio anterior da recursão;
⇒ Ack (0, 3)	: X = 0, então é utilizada a equação Ack(0,n) = n+1;
⇒ 4	: volta ao estágio anterior da recursão;
⇒ Ack (0, 4)	: X = 0, então é utilizada a equação Ack(0,n) = n+1;
⇒ 5	: resultado final.

Percebe-se que a função, a medida que há um crescimento em seus parâmetros de entrada, sofre crescimento em sua pilha de recursão. Isso ocorre devido à função não parar de chamar a si própria com novos valores até que x seja 0. Quanto maior for o x, maior será o número de recursões já que a operação realizada sobre ele é de decremento, até que ele seja 0.

3. Ex. 8)

i.£ $\in \sum_{i=1}^{2}{}^{*}$ pois sendo £ a palavra vazia ela pertence ao conjunto $\sum_{i=1}^{2}{}^{*}$.

De fato, pois se assim não fosse, existiria pelo menos um símbolo x tal que $x \equiv$ £ e tal que $x \notin \sum_{i=1}^{2}{}^{*}$. Porém, como por definição, $\nexists x$ tal que $x \equiv$ £. Já que £ é a palavra vazia tem-se uma só escolha:

£ $\in \sum_{i=1}^{2}{}^{*}$

ii.Vamos provar agora que £ $< x, \forall x \in \sum_{i=1}^{2}{}^{*}$ construindo a seqüência $\{$£, $< a < b < aa < ab < ba < bb\}$.

De fato:

1.Se SUCCe(£) = a, então SUCC(£) = REVERSO(SUCCe(REVERSO(£)))
= REVERSO(SUCCe(£))
= REVERSO(a)
= a

Logo, £ < a;

2.Se SUCCe(a) = SUCCe(a£) = b£ = CONSb(£) = CONSb£ = b
Então: SUCC(a) = REVERSO(SUCCe(REVERSO(a)))
= REVERSO(SUCCe(a))
= REVERSO(b)
= b

Logo, £ < a < b;

3.Se SUCCe(b) = SUCCe(b£) = CONSa(SUCC(£)) = CONSa(a) = aa
Então: SUCC(a) = REVERSO(SUCCe(REVERSO(b)))
= REVERSO(SUCCe(b))
= REVERSO(aa)
= aa

Logo, £ < a < b < aa;

4.Se SUCCe(aa) = ba = CONSb(a)
Então: SUCC(aa) = REVERSO(SUCCe(REVERSO(aa)))
= REVERSO(SUCCe(aa))
= REVERSO(ba)
= ab

Logo, £ < a < b < aa < ab;

5.Se SUCCe(ba) = CONSa(SUCC(a)) = CONSa(b) = ab
Então: SUCC(ab) = REVERSO(SUCCe(REVERSO(ab)))
= REVERSO(SUCCe(ba))
= REVERSO(ab)
= ba

Logo, £ < a < b < aa < ab < ba;

6.Se SUCCe(ab) = bb = CONSb(b)
Então: SUCC(ba) = REVERSO(SUCCe(REVERSO(ba)))
= REVERSO(SUCCe(ab))
= REVERSO(bb)
= bb

Portanto: £ < a < b < aa < ab < ba < bb;

3. Ex. 9)

i) Prova de que £ pertence ao alfabeto {a,b}.

$£ \in \sum_{i=1}^{2}{}^{*}$. De fato, pois em caso contrário, $\exists x\,;\ x \equiv £$ e tal que $x \notin \sum_{i=1}^{2}{}^{*}$. Porém, como $\nexists x$ tal que $x \equiv £$, pois $£ \overset{def}{\equiv}$ palavra vazia, tem-se que $£ \in \sum_{i=1}^{2}{}^{*}$.

ii) Vamos agora provar que $x < bb, \forall x \in \sum_{i=1}^{2}{}^{*}$ construindo a seqüência $\{bb > ba > ab > aa > b > a > £\}$.

Para descobrirmos todos os antecessores de bb usaremos a função ANTL (antecessor lexográfico) que é definida sobre ∑ = {a,b} onde

ANTL(x) = REVERSO(ANTL(REVERSO(x))),
onde ANTL(£) = £;
ANTL(ax) = COND(x, NIL(x), CONSb(ANTL(x));
ANTL(bx) = CONSa(x);
e CONSb(x) = bx = função concatena e esquerda de x;
NIL(x) = £ = função nula;
COND(x, £, bx) = função condicional = bx se x ≠ £ ,e =£ se x = £;
REVERSO(x) = w^r;

Então:
1)ANTL(bb) = REVERSO(ANTL(REVERSO(bb)))
= REVERSO(ANTL(bb))
ANTL(bb) = CONSa(b) = ab
= REVERSO(ab)
= ba
Logo: bb > ba;

2)ANTL(ba) = REVERSO(ANTL(REVERSO(ba)))
= REVERSO(ANTL(ab))
ANTL(ab) = COND(b, NIL(b), CONSb(ANTL(b))
= COND(b, NIL(b), CONSb(a))
= COND(b, b, ba)

= ba
= REVERSO(ba)
= ab
Logo bb > ba > ab;

3) ANTL(ab) = REVERSO(ANTL(REVERSO(ab)))
= REVERSO(ANTL(ba))
ANTL(ba) = CONSa(a) = aa
= REVERSO(a)
= aa
Logo bb > ba > ab > aa;

4) ANTL(aa) = REVERSO(ANTL(REVERSO(aa)))
= REVERSO(ANTL(aa))
ANTL(aa) = COND(a, NIL(a), CONSb(ANTL(a))
= COND(a, a, CONSb(£))
= COND(b, b, b£)
= b
= REVERSO(b)
= b
Logo bb > ba > ab > aa > b;

5) ANTL(b) = REVERSO(ANTL(REVERSO(b)))
= REVERSO(ANTL(b))
ANTL(b) = ANTL(b£) = CONSa(£) = a£ = a;
= REVERSO(a)
= a
Logo bb > ba > ab > aa > b > a;

6) E, *ANTL(a) = ANTL (a£) = COND(£, NIL(£), CONSb(ANTL(£))*
= COND(£, £, CONSb(£))
= COND(£,£, b£)
= £
Portanto: bb > ba > ab > aa > b > a > £;

3. Ex. 10

a) a.f(g) (x,y) = (g (g (x, y), y), y)

Tipo: f: $(N^2 \rightarrow N) \rightarrow (N^2 \rightarrow N^2)$

A notação λ de Church fornece um método para a construção de prefixos com os nomes das funções, dessa forma, podemos escrever f da seguinte maneira:

λ : λ g. λ x, y. (g (g (x, y), y), y)

A resposta está correta, pois:

g (x, y) ∈ N, porque g : $N^2 \rightarrow N$
y ∈ N
g (g (x, y), y) ∈ N, porque g : $N^2 \rightarrow N$
(g (g (x, y), y), y) ∈ N^2, porque {g (g (x, y), y) ∈ N} e {y ∈ N}
(x,y) ∈ N^2
Se f(x,y) = (g (g (x, y), y), y), então f é do tipo f: $N^2 \rightarrow N^2$
Logo, conclui-se que f: $(N^2 \rightarrow N) \rightarrow (N^2 \rightarrow N^2)$ se f(g) (x,y) = (g (g (x, y), y), y).

Imagem da função: (g (g (x, y), y), y)
Domínio da função: (x,y)
Portanto, λ : λ x, y. (g (g (x, y), y), y).
Entretanto, como é um funcional e g determina as funções geradas, fica:
λ : λ g. λ x, y. (g (g (x, y), y), y)

b) f(g, x, y) = g(x, y)
Tipo: f: $((N^2 \rightarrow N) \times N^2) \rightarrow N$
Notação λ: λg, x, y. g (x, y)

c) f(g, x) = g (g (x, x), x)
Tipo: f: $((N^2 \rightarrow N) \times N) \rightarrow N$

O tipo de f é composto pela função g, que juntamente com x é um argumento da função. Em semelhança à função do exercício b, podemos chamar esta função de FUNCIONAL.

Notação □□□□g, x . g (g (x, x), x)

Podemos assim escrever a notação □: "a função f para os argumentos g e x produz g (g (x, x), x)". Se quisermos há também a possibilidade de introduzir o conjunto domínio dos argumentos na notação, resultando em: □g ∈ N^2, x ∈ N . g (g (x, x), x)

d) f(x)(y) = g(x, y)

A função f é um funcional onde para cada inteiro não negativo x (x ∈ N) é definida a função f(x) como seu valor. E, a função f(x) produz um valor em N (f:N→N) pois, de acordo com a definição, a função g(x, y) produz um valor em N (g:N^2→N) e f(x)(y) = g(x, y). Logo, o <u>tipo da função f</u> é:

f: N → (N → N)

A função **f** escrita na notação **λ** de Church é:

λx.λy.g(x, y)

ou seja, a função apresentada, para argumentos arbitrários x e y, produz g(x, y). Esta mesma função, com descrição do domínio, é representada assim:

λx ∈ N. λy ∈ N.g(x, y)

3. EX 11)

Para verificar se as regras *eal* e *eaf* definem a mesma função, avaliaremos como exemplo a execução de f(1,0).

a) Avaliação de f(1,0) pela regra *eaf*:

f(1,0)	
⇨ (1=0→1, 0=0→f(1,1), f(0-1, f(1, 0-1)))	**Pela regra a)** Na definição, substituímos todos os **x** por **1** e todos os **y** por **0**.
⇨ f(1,1)	**Pela regra b)** Inicialmente, avaliamos a expressão da forma (1=0→**1**, $\mathbf{e_2}$). Como 1=0 é falso, tomamos como valor da expressão o valor de $\mathbf{e_2}$. Para isso, avaliamos (0=0→**f(1,1)**, $\mathbf{e_4}$). Como 0=0 é verdadeiro, o valor da expressão é dado por f(1, 1).
⇨ (1=0→1, 1=0→f(1,1), f(1-1, f(1,1-1)))	**pela regra a)** Para avaliar f(1, 1), substituímos todos os **x** por **1** e todos os **y** por **1**, na definição.
⇨ f(0, f(1,0))	**pela regra b)** Avaliamos a expressão da forma (1=0→**1**, $\mathbf{e_2}$). Como 1=0 é falso, o valor da expressão é dado por $\mathbf{e_2}$. Avaliamos, então, $\mathbf{e_2}$ da forma (1=0→**f(1,1)**, $\mathbf{e_4}$). Como 1=0 é falso, devemos avaliar $\mathbf{e_4}$ da forma **f(1-1, f(1,1-1))**. **pelas regras c) e e)** Avaliamos as expressões (**1-1**), que resultam nas expressões constantes **0** em suas duas ocorrências ("**c)**"). A constante **1**, antes da última ocorrência de (1-1), é tomada como ela própria ("**e)**").
⇨ f(1,0)	**pela regra d)** Avaliamos 0 e f(1,0) na seqüência. O valor de **0** é tomado como ele próprio. Mas, a avaliação de f(1,0) implica em reinicializar o processo de avaliação novamente...

Como a avaliação de f(1, 0) implica em reiniciar todo processo novamente, isto implicará em *loop*. O processo continuará sem fim. Assim, pela regra *eaf*, f(1, 0) é indefinida.

a) Avaliação de f(1,0) pela regra *eal*:

f(1,0)	
⇨ (1=0→1, 0=0→f(1,1), f(0-1, f(1, 0-1)))	**pela regra a)** Na definição, substituímos todos os **x** por **1** e todos os **y** por **0**.
⇨ f(1,1)	**pela regra b)** Inicialmente, avaliamos a expressão da forma (1=0→**1**, $\mathbf{e_2}$). Como 1=0 é falso, tomamos como valor da expressão o valor de $\mathbf{e_2}$. Para isso, avaliamos (0=0→**f(1,1)**, $\mathbf{e_4}$). Como 0=0 é verdadeiro, o valor da expressão é dado por f(1, 1).
⇨ (1=0→1, 1=0→f(1,1), f(1-1, f(1,1-1)))	**pela regra a)** Para avaliar f(1, 1), substituímos todos os **x** por **1** e todos os **y** por **1**, na definição.
⇨ f(1-1, f(1,1-1)	**pela regra b)** Avaliamos a expressão da forma (1=0→**1**, $\mathbf{e_2}$). Como 1=0 é falso, o valor da expressão é dado por $\mathbf{e_2}$. Avaliamos, então, $\mathbf{e_2}$ da forma (1=0→**f(1,1)**, $\mathbf{e_4}$). Como 1=0 é falso, devemos avaliar, agora, $\mathbf{e_4}$ da forma **f(1-1, f(1,1-1))**.

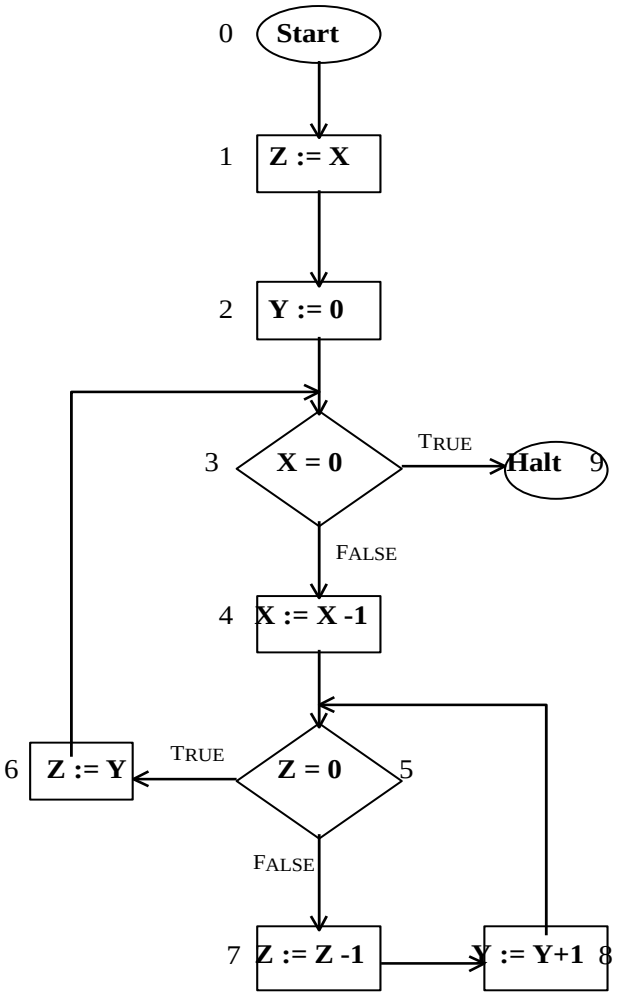
⇨ f(1-1=0→1, f(1,1-1)=0→f(1-1,1),
f(f(1,1-1)-1,f(1-1,f(1,1-1)-1)))

pela regra d') Aplicamos a regra "**a)**" substituindo, na definição, todos os **x** por (**1-1**) e todos os **y** por (**f(1, 1-1)**).
pela regra c) Avaliamos a primeira ocorrência de (**1-1**) obtendo **0**.
pela regra b) Avaliamos (0=0→**1**, $\mathbf{e_2}$). Como 0=0 é verdadeiro, o resultado da expressão é **1**.

⇨ 1

Assim, a avaliação de f(1, 0), pela regra *eal* produz como resultado o valor 1.
As duas avaliações (eal e eaf) produziram resultados diferentes, logo, elas não definem a mesma função para a definição apresentada.

3. Ex 12)



Esta função pode ser definida usando-se apenas três registradores (**X**, **Y** e **Z**) pertencentes ao conjunto memória **V**, lembrando que **X** servirá como entrada e **Y** como saída.

Inicialmente faremos a transformação do programa **While** fornecido para o programa **Fluxograma** esquematizado acima, já com a numeração apropriada para convertê-lo numa seqüência de **Instruções Rotuladas**:

Portanto, a seqüência de instruções rotuladas é:
1: do Z:=X then goto 2
2: do Y:=0 then goto 3
3: if X=0 then goto 9 else goto 4
4: do X:=X-1 then goto 5
5: if Z=0 then goto 6 else goto 7
6: do Z:=Y then goto 3
7: do Z:=Z-1 then goto 8
8: do Y:=Y+1 then goto 5

Agora vamos traduzir as instruções rotuladas no conjunto de definições recursivas simultâneas que segue:
$f_0(X)$ is $f_1(X,0,0)$
$f_1(X,Y,Z)$ is $f_2(X,Y,X)$
$f_2(X,Y,Z)$ is $f_3(X,0,Z)$
$f_3(X,Y,Z)$ is ($X=0 \rightarrow f_9(X,Y,Z)$, $f_4(X,Y,Z)$)
$f_4(X,Y,Z)$ is $f_5(X-1,Y,Z)$
$f_5(X,Y,Z)$ is ($Z=0 \rightarrow f_6(X,Y,Z)$, $f_7(X,Y,Z)$)
$f_6(X,Y,Z)$ is $f_3(X,Y,Y)$
$f_7(X,Y,Z)$ is $f_8(X,Y,Z-1)$
$f_8(X,Y,Z)$ is $f_5(X,Y+1,Z)$
$f_9(X,Y,Z)$ is Y

Vale lembrar que pela forma das definições, está assegurado que a avaliação passo-a-passo de $f_0(X)$ usando-se a *regra-eaf* dará exatamente o mesmo resultado (mas em ordem diferente) do que quando for usada a *regra-eal.*

Podemos então, se desejarmos, efetuar a simplificação correspondente, quando eliminaremos f_1, f_2, f_4, f_6, f_7, f_8 e f_9. Basta fazer a substituição de suas expressões de definição nas definições restantes, resultando:
$f_0(X)$ is $f_3(X,0,X)$
$f_3(X,Y,Z)$ is ($X=0 \rightarrow Y$, $f_5(X-1,Y,Z)$)
$f_5(X,Y,Z)$ is ($Z=0 \rightarrow f_3(X,Y,Y)$, $f_5(X,Y+1,Z-1)$)

Mais uma vez, a simplificação só foi possível devido ao fato de que ambas as regras de avaliação proporcionam o mesmo resultado.

3. Ex. 13)

a) Definição recursiva da f:

$$f(x) = \prod_{y=0}^{x} g(y) = g(0) \cdot g(1) \cdot ... \cdot g(x-1) \cdot g(x) = f(x-1) \cdot g(x)$$

$f(0) = \prod_{y=0}^{0} g(y) = g(0)$, logo podemos escrever f recursivamente como:

$$f(x) = (x = 0 \rightarrow g(x),\ f(x-1) \cdot g(x))$$

b) Notação λ:

f(x) na realidade é função de g e de x, mas g por sua vez é função de y, então temos na realidade f (y) (x).

Portanto na notação λ:

$$\lambda f.\lambda y.\lambda x(x = 0 \rightarrow g(x),\ f(x-1) \cdot g(x))$$

3. Ex. 14)

a) **Função exponencial:** $h(x) = e^x \Rightarrow \lambda x.e^x$ **;**

b) **Composição de funções simples:** $h(x) = g(f(x)) \Rightarrow \lambda g.\lambda f.\lambda x.g(f(x))$ **;**

Descreva a função exponencial e a composição simples de funções através da notação λ.

Função exponencial: $f(x)=e^x \Rightarrow$ **$\lambda x.e^x$** onde:

λx. indica que a função é aplicada sobre o argumento **x;**

.e^x indica o que a operação realizada pela função é a exponencial **e^x.**

Composição de funções simples: $h(x)=g(f(x)) \Rightarrow$ **$\lambda g.\lambda f.\lambda x.g(f(x))$** onde:

λg.λf.λx. indica que a operação será realizada sobre as funções **g**, **f** e o argumento **x.**

.g(f(x)) indica que a operação realizada é a composição das funções **g** e **f** em função do argumento **x.**

3. Ex. 15)

perfect **(x) é (x = 0 → next (x+1),** next **(**perfect **(x-1)+1))**

next **(x) é (**isperfect **(x) → x,** next **(x+1))**
isperfect **(x)** **é** equal **(x,** sumdiv **(x, x-1))**
equal **(x, y)** **é (x = 0 → y = 0, (y = 0 → false,** equal **(x-1, y-1)))**
sumdiv **(x, y) é (y = 0 → 0,** add **(**sumdiv **(x, y-1),** isdiv **(x, y)))**
add **(x, y)** **é (y = 0 → x,** add **(x+1, y-1))**
isdiv **(x,y)** **é (**equal **(x,** mult **(y,** div **(x,y))) = true → y, 0)**
div **(x, y)** **é (**sub **(x, y) = 0 → 0,** div **(**sub **(x, y), y)+1)**
sub **(x, y)** **é (x = 0 → 0 (y = 0 → x,** sub **(x-1, y-1)))**
mult **(x, y)** **é (y = 0 → 0,** add **(**mult **(x, y-1), x))**

<u>O que faz cada função:</u>

perfect(x): Se x=0, retorna o primeiro número perfeito. Senão, calcula o primeiro número perfeito que vem depois do resultado de uma chamada recursiva de perfect incrementado de uma unidade.

next(x): Se *x* for perfeito, retorna o seu valor, senão chama-se recursivamente até que seja encontrado um número perfeito.

isperfect(x): Testa se *x* é igual a soma de seus divisores, retornando *true* ou *false.*

equal(x,y): Se x=0, retorna *true*;
Se x≠0 e y=0, retorna *false;*
Se x≠0 e y≠0, chama-se recursivamente, decrementando *x* e *y* até recair em um dos casos anteriores.

sumdiv(x,y): *y* representa um possível divisor de *x* e é igual a x-1 na primeira chamada (xperfeito = soma dos divisores de x excluindo o próprio).São feitas chamadas recursivas, decrementando y a cada chamada, até y=0. Ao "desempilhar" as chamadas, os resultados de isdiv (0 ou o próprio divisor) vão sendo somados.

add(x,y): Chama-se recursivamente, incrementando x e decrementando y, até y=0, ou seja, até que a x tenha sido adicionado o valor de y.

isdiv(x,y): Testa se *x* é igual a *y* multiplicado pela divisão inteira de *x* por *y*, ou seja, se *y* é divisor de *x.* Retorna 0 se y não for divisor e o valor do próprio y, caso contrário.

div(x,y): Chama-se recursivamente, contando, através de sub(x,y) quantas vezes y "cabe" dentro de x. Essa contagem é feita através somando-se ao desempilhar cada chamada recursiva. O resultado dessas somas é a divisão inteira de x por y.

sub(x,y): Se x=0, o resultado é zero;

Se x≠0 e y=0, o resultado é x;

Se x≠0 e y≠0, chama-se recursivamente até recair em um dos casos anteriores.

Obs: se y>x, o resultado é zero (não considera-se números negativos).

mult(x,y): Se y=0, o resultado é zero, senão chama-se recursivamente, decrementando y a cada chamada até y=0. Então começa a "desempilhar" as chamadas, somando *x* a cada desempilhada. O resultado dessas somas é a multiplicação de x por y.

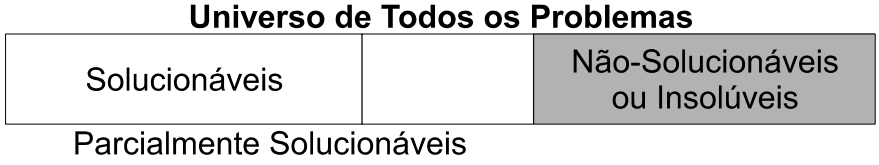
obs: Note que o número de chamadas recursivas é igual a y, que por sua vez é igual ao número de vezes que x será somado a si próprio.

RESPOSTAS DO CAPÍTULO 4

4. Ex. 1)

A classe de problemas solucionáveis (aqueles que em uma máquina de Turing sempre finalizam em *aceita* **ou** *rejeita*) está contida na classe dos problemas parcialmente solucionáveis (problemas que possuem um algoritmo que **pelo menos** com uma palavra de entrada resulta em *loop*) que contém alguns problemas não-solucionáveis (que para alguma palavra de entrada possuem solução parcial). Assim temos o diagrama a seguir:

Pode-se afirmar que todo problema solucionável é parcialmente solucionável, embora nem todos problemas parcialmente solucionáveis sejam totalmente solucionáveis. A intercessão entre as classes de problemas totalmente solucionáveis e a classe dos não-solucionáveis é nula, já que nenhum problema pode estar contido nas duas simultaneamente.



4. Ex. 2)

Uma forma precisa de expressar e solucionar problemas é através do conceito de Linguagem, pois permite a investigação da existência de um algoritmo que determine se uma palavra pertence ou não à linguagem que traduz tais problemas. Traduzindo-se um problema em uma linguagem, obtem-se um problema de linguagem que é descrito e solucionado da seguinte forma:

a) O primeiro passo consiste em reescrever o problema a ser estudado sob forma de um problema de decisão, que produz respostas do tipo *afirmativo/ negativo (sim/ não)*. Esta tradução é possível em grande parte dos casos.
b) Os argumentos do problema *sim/ não* são codificados como palavras de um alfabeto, gerando uma linguagem.

Com este tipo de tradução, o problema envolvendo a solucionabilidade é traduzido para um problema de linguagem, no qual deve-se definir se a linguagem gerada é Recursiva (correspondendo a um problema solucionável) ou Enumeravelmente Recursiva (problema parcialmente solucionável).

4. Ex. 3)

O Problema da Parada da parada da Máquina de Turing é um dos mais importantes problemas insolúveis conhecidos.
Sua definição: dada uma máquina de Turing M, existe um algoritmo que verifique se M pára ou não, aceitando ou rejeitando a palavra de entrada.
Tal problema não tem solução, ou seja, não existe tal algoritmo.
A importância disso reside no fato de que, através desse problema, pode-se definir a classe de solucionabilidade de outros problemas, ou seja, se um problema A pode ser reduzido a um problema B insolúvel, o problema A também é insolúvel.
Logo, é possível analisar a solucionabilidade de diversas classes de problemas a partir do P.P.M.T., e assim sucessivamente. Isso evita que se perca tempo buscando solução para problemas insolúveis.

4. Ex. 4)

O Princípio da Técnica de Redução de Turing consiste basicamente na investigação da solucionabilidade de um problema a partir de outro, cuja classe de solucionabilidade é conhecida:

A idéia é construir um algoritmo de mapeamento entre as linguagens que representam os problemas. Assim, se a classe de uma dessas linguagens é conhecida, então podem-se estabelecer algumas conclusões sobre a linguagem que se deseja investigar e, portanto, do grau de solucionabilidade do problema representado.

A função de mapeamento deve ser Turing-Computável total e é denominada Máquina de Turing de Redução. A Redução R entre duas linguagens L_1 e L_2 sobre Σ deve satisfazer as seguintes condições (suponha w uma palavra de Σ^*):

i) Se $w \in L1$ então $R(w) \in L2$
ii) Se $w \notin L1$ então $R(w) \notin L2$

4. Ex. 5)

Pode-se reduzir o *Problema-SA* (que é insolúvel) no *Problema da Parada da Palavra Constante* da seguinte forma:

Para cada fluxograma P em NORMA, seja P' o programa X:=p; P, onde p é a codificação de P. Assim, considerando a computação em NORMA, normaP'(x) = normaP(p), para qualquer entrada x. Seja ξ a função $\xi(p) = p'$, onde p' é a codificação de P'.

Suponha que Q com entrada p seja um programa para decidir o *Problema da Parada da Palavra Constante*, ou seja, a computação de normaQ(p) decide se normaP(w) é definido ou não, onde w é uma palavra constante. Seja R o programa X := ξ(X); Q. Então, a computação de normaR(p) resultará em normaQ(p'), e R decidirá se normaP'(w) é definido. Como normaP'(w) = normaP(p), para qualquer entrada w, então R decidirá se normaP(p) é definido, resolvendo o *Problema-SA*, o que é um absurdo. Logo, o programa Q não pode existir e o *Problema da Parada da Palavra Constante* não tem solução.

4. Ex. 6)

a)
Se o Problema de Aceitação da palavra é definido por: dada uma Máquina de Turing M qualquer e uma palavra w ∈ ∑*, M aceita w ? então a linguagem correspondente a este problema é conhecida como a Linguagem Universal.
Seja esta Linguagem Universal denotada por L_U , então pela definição do Problema de Aceitação, pode-se definir L_U como:

$$L_U = \{ \ (M, w);\ w \in ACEITA(M);\ \ w \in \sum^* \}$$

Comentários "para leigos": A linguagem universal, como definida acima, é dada pelo conjunto de todas as palavras para as quais exista pelo menos uma Máquina de Turing que as reconheça.

b)
Observa-se inicialmente que da definição de L_U tem-se que ∑* – L_U = LOOP(M) REJEITA(M). Então prova-se que L_U é enumerável recursivamente consequentemente L_U é parcialmente solucionável.
E para provar que L_U é ER, pela definição de ER, basta provar que existe uma Máquina de Turing T, tal que satisfaça às seguintes condições:

i) ACEITA(T)= L_U
ii) REJEITA(T) U LOOP(T) = ∑* – L_U

Sejam então w ∈ ∑* e M uma Máquina de Turing qualquer codificada como uma palavra em ∑* (isto é M ∈ ∑*) e seja T um simulador genérico da Máquina de Turing M que é, também, uma Máquina de Turing.
Então como T recebe como entrada o par (M,w) e simula a Máquina M para a palavra tem-se que:
1) (M,w) ∈ ACEITA(T) ⇔ (M,w) ∈ ACEITA(M)
2) (M,w) ∈ REJEITAT) ⇔ (M,w) ∈ REJEITA(M)
3) (M,w) ∈ LOOP(T) ⇔ (M,w) ∈ LOOP(M)

Portanto tem-se que ACEITA (T) = L_U e REJEITA(T) U LOOP(T) = ∑* – L_U

Comentários "para leigos": Para provarmos que o problema é parcialmente solucionável basta provarmos que a linguagem é enumerável recursivamente e para isso basta provarmos que existe uma Máquina de Turing T para a qual o conjunto ACEITA(T) é igual à linguagem universal e o que não pertence a L_U está em REJEITA(T) ou LOOP(T).

Definimos então uma máquina T que seja um simulador genérico de uma Máquina de Turing. A entrada dessa máquina T é uma Máquina de Turing M codificada em uma palavra. Então, temos que:

a) Para qualquer par (M, w) que T aceita, M também aceita.
b) Para qualquer par (M, w) que T rejeita, M também rejeita.
c) Para qualquer par (M, w) que T entra em loop, M também entre em loop.

Logo, o conjunto ACEITA(T) = L_U e REJEITA(T) $\cup$ LOOP(T) = $\sum^* - L_U$.

C)
Prova-se que L_U é insolúvel provando que L_U não é recursiva. E para provar que L_U não é recursiva, pela definição de Recursiva, basta provar que não existe uma Máquina de Turing *T*, tal que satisfaça às seguintes condições:
i) ACEITA(T)= L_U
ii) REJEITA(T) = $\sum^* - L_U$
iii) LOOP(T) = { }

Para provar que L_U não é recursiva supõe-se que isso não seja verdade e ver-se-á que tem-se um absurdo. De fato, se L_U é recursiva sejam $w \in \sum^*$ e *M* uma Máquina de Turing qualquer codificada como uma palavra em $\sum^*$ (isto é $M \in \sum^*$), então:

1) Sendo L_U recursiva e *T* um simulador genérico da Máquina de Turing *M* (que é também uma Máquina de Turing) tem-se que $ACEITA(T_1) = L_U$, $REJEITA(T) = \sum^* - L_U$ e $LOOP(T) = \{ \}$. Com isso o mapeamento T_1 pode ser definido como:

$$T_1(M, w) = \begin{cases} ACEITA, & se \quad w \in ACEITA(M) \quad \text{ou} \quad w \in REJEITA(M) \\ REJEITA, & se \quad w \in LOOP(M) \end{cases}$$

Ou seja, como L_U é recursiva T_1 sempre pára e assim T_1 identifica quando *M* fica em LOOP para a entrada *w* e rejeita o par *(M,w)*.

2) Seja T_2 a Máquina de Turing T_1 modificada como segue: Se *M* aceita ou rejeita *w* T_2 fica em LOOP e, se *M* fica em LOOP, T_2 rejeita. Ou seja, o mapeamento T_2 pode ser definido como:

$$T_2(M, w) = \begin{cases} LOOP, & se \quad w \in LOOP(M) \\ ACEITA, & se \quad w \in LOOP(M) \text{ ou } w \in REJEITA(M) \end{cases}$$

3) Seja agora T_3 um simulador genérico da Máquina de Turing *M* (que é também uma Máquina de Turing) que recebe como entrada o código *M* (que também é um simulador da Máquina de Turing *M)* que realiza o seguinte mapeamento:
i) Duplica a entrada M , gerando na fita o par (M,M)
ii) O para (M,M) é processado por T_3 (observe que $M \in \sum^*$)
Então tem-se que T_3 pode ser definido por:

$$T_3 = \begin{cases} T_3(M) = (M, M) \\ T_3(T_2(M, M)) = \begin{cases} LOOP, & \text{se} \quad M \in REJEITA(M) \quad \text{ou} \quad M \in ACEITA(M) \\ ACEITA, & \text{se} \quad M \in LOOP(M) \end{cases} \end{cases}$$

4) Agora como *M* é o código de uma Máquina de Turing qualquer então pode-se fazer em particular que $M = T_3$. Ou seja pode-se aplicar a entrada T_3 (observe que $T_3 \in \Sigma^*$) para a Máquina de Turing T_3 obtendo o seguinte mapeamento:

$$T_3 = \begin{cases} T_3(T_3) = (T_3, T_3) \\ T_3(T_2(T_3, T_3)) = \begin{cases} LOOP, & \text{se } T_3 \in REJEITA(T_3) \text{ ou } T_3 \in ACEITA(T_3) \\ ACEITA, & \text{se } T_3 \in LOOP(T_3) \end{cases} \end{cases}$$

Assim, desta construção obtemos as seguintes conseqüências:
a) T_3 fica em LOOP quando T_3 pára
b) T_3 pára quando T_3 entra em LOOP

Daí como a) e b) são contraditórios não existe Máquina T_1 que o defina e daí como T_1 não existe, tem-se que não existe uma Máquina de Turing *T* tal que sejam verdadeiras as condições : $ACEITA(T) = L_U$, $REJEITA(T) = \Sigma^* - L_U$ e $LOOP(T) = \{\ \}$. Portanto L_U não é recursiva.

Comentário: Este é, sem dúvida, um dos exercícios mais complexos desta disciplina. Nos limitamos a concluir que o procedimento tomado é bastante razoável. Parte-se do pressuposto de que L_u é uma função recursiva. Cria-se então um mapeamento de forma que a função é rejeitada apenas quando a mesma entra em loop, bem como uma máquina de Turing modificada de acordo com esse mapeamento e um outro simulador genérico. Enfim, chega-se a uma contradição, de forma que a suposição inicial de que L_u é recursiva não é verdadeira.

4. Ex.7)

Para a demonstração iremos utilizar as hipóteses de que o problema da entrada vazia para Turing é insolúvel e que o problema da equivalência para norma é equivalente ao problema da equivalência para Turing. Posteriormente tentar-se-á demonstrar que se o problema da equivalência fosse solúvel o problema da parada vazia também deveria ser.

Sabe-se que o problema de Entrada Vazia é insolúvel:
Demonstração:
A demonstração é realizada usando o item c) do Teorema 4.5 que afirma que: *Se L_1 não é recursiva, então L_2 não é recursiva*

Especificamente, a linguagem L_P que especifica o Problema da Parada da Máquina de Turing (que não é recursiva) é reduzida a linguagem
$L£ = \{ M; £ \text{ pertence a } ACEITA(M) \cup REJEITA(M) \}$

Seja *T* uma Máquina de Turing qualquer e *w* uma palavra qualquer sobre Σ. E sejam ainda, *W* uma Máquina de Turing que recebe como entrada a palavra vazia e gera na fita a palavra *w* e seja *M* uma Máquina de Turing definida em termos de *T* e *W*. As conclusões que podem ser estabelecidas são as seguintes:

- Se *T* aceita a palavra *w*, então *M* aceita a palavra vazia
- Se *T* não aceita a palavra *w* (rejeita ou fica em loop), então *M* não aceita a palavra vazia (rejeita ou fica em loop).

Portanto, o Problema da Parada da Máquina de Turing foi reduzido a algum problema da palavra vazia (que consequentemente é insolúvel). Como *L£* é equivalente a L_p, que é insolúvel, conclui-se que *L£* também o é.

Seja LOOP um programa em Norma descrito como:
While X=0 do (I); While X<> 0 do (I).

Pela Hipótese de Church sabe-se que Máquina Norma é equivalente a Máquina de Turing e vice-versa. Logo, o programa LOOP é facilmente traduzido para a Máquina de Turing.

Observa-se que normaLOOP(x) é indefinido para qualquer x. Ainda, pode se dizer que para qualquer programa P em Norma (facilmente traduzível para Máquina de Turing), P é equivalente a LOOP, se e somente se normaP é indefinido para qualquer x. Isto equivale dizer que P é equivalente a LOOP se o programa P executado na Máquina de Turing é indefinido para qualquer entrada.

O problema de Equivalência de duas Máquinas de Turing é :

M é uma Máquina de Turing que executa o programa P e W é uma Máquina de Turing que executa o programa LOOP, conclui-se que para P ser equivalente a M deve valer a seguinte afirmação :

- A computação de W é indefinida para qualquer entrada, então a computação de M também deve ser indefinida para qualquer entrada.

Portanto dado um programa que decide o problema da equivalência pode se construir um programa que decide o problema da Entrada Vazia (que é insolúvel), uma vez que se fosse possível determinar para qualquer entrada que as computações de 2 maquinas diferentes não entram nem em aceita ,nem em rejeita, também seria possível determinar para 2 maquinas diferentes se elas teriam estados finais idênticos para entradas diferentes.

Portanto o problema de equivalência também é insolúvel.

4. Ex. 8)

a)

Comparam-se diferentes problemas pertencentes a mesma classe, ou seja, tanto P_1 quanto P_2 possuem o mesmo índice de solucionabilidade. Ao realizar tais operações está-se determinando, na verdade, a possibilidade de uma agregação dos problemas que são representados através de P_1P_2.

Destaca-se que este estudo é muito importante para Ciência da Computação de uma forma geral pois pode-se determinar se, quando operados, dois problemas ***gerarão alguma solução*** (resultando em um problema Parcialmente Solucionável ou Solucionável, como será visto abaixo) ou quando o problema gerado não oferece nenhuma solução (caso em que é chamado de insolúvel).

Agora, supondo que tem-se dois problemas, a saber, P_1 e P_2, pode-se inferir o seguinte comportamento quando os conetivos abaixo são utilizados:

b)

Supondo que tem-se dois problemas, a saber, P_1 e P_2, pode-se inferir o seguinte comportamento quando os conetivos abaixo são utilizados:

- Conjunção: $P_1 \wedge P_2 \equiv P_{\text{resultante}}$ é solucionável;
- Disjunção: $P_1 \vee P_2 \equiv P_{\text{resultante}}$ é solucionável;

- Negação: $\neg P_1 \equiv P_{\text{resultante}}$ é parcialmente solucionável ou não solucionável; *

Formalmente, pode-se abordar o problema da seguinte forma: Por definição, se um problema é solucionável, a linguagem gerada em sua redução é dita *Recursiva*.

Pela definição de linguagem Recursiva tem-se que uma linguagem λ é Recursiva se existe pelo menos uma Máquina de Turing tal que aceite a linguagem λ e sempre pare para qualquer que seja a entrada, isto é, significa dizer que a linguagem nunca causa um *loop*.

Dados 2 problemas solucionáveis P1 e P2. Pelas definições acima:

$$\begin{cases} \text{Loop}(P_1) = \{\} \\ \text{Aceita}(P_1) \cup \text{Rejeita}(P_1) = Lp_1 \end{cases} \qquad \begin{cases} \text{Loop}(P_2) = \{\} \\ \text{Aceita}(P_2) \cup \text{Rejeita}(P_2) = Lp_2 \end{cases}$$

Desta forma, os operadores lógicos são assim definidos:

Conjunção
$(\nexists P_1\ Loop(P_1) \wedge \nexists P_2\ Loop(P_2)) \Rightarrow \nexists (P_1, P_2)\ Loop(P_1 \wedge P_2)$, logo a conjunção de dois problemas sabidamente solúveis gera um problema também solucionável.

Disjunção
$(\nexists P_1\ Loop(P_1) \vee \nexists P_2\ Loop(P_2)) \Rightarrow \nexists (P_1, P_2)\ Loop(P_1 \vee P_2)$, desta forma a disjunção de dois problemas solucionáveis também gera um problema solucionável.

Negação
$\neg(\forall P_1\ Loop(P_1)) \Rightarrow \exists P_1\ \neg Loop(P_1)$, significa que existe pelo menos uma situação em que é gerado uma definição (estado de aceita ou rejeita) ou pode ser possível que a linguagem Lp_1 entre em *loop*.

Portanto, se a máquina pode entrar em Loop ou parar (aceitando ou rejeitando) ela é chamada de *Enumeravelmente Recursiva*, linguagem esta que define os problemas parcialmente solucionáveis. Pode acontecer que a máquina nunca pára (quando a união entre Aceita(P_1) e Rejeita(P_1) for vazia), que caracteriza a insolubilidade de problema proposto. Portando não é possível determinar exatamente para este caso qual das duas situações que ocorrerão.

c)

Na resolução deste problema, foram observadas as respostas das letras a), b), d) e a figura 4.2 da pg 105 , podemos pensar em termos de $\cup e \cap$ de conjuntos.

P1 e P2 são sempre da mesma classe, e por b) e d) temos:

Conjunção → $P_1 \wedge P_2$, o P_{Res} é da mesma classe de P1 e P2;

Disjunção → $P_1 \vee P_2$, o P_{Res} é da mesma classe de P1 e P2;

Negação → $\sim P_1$, o P_{Res} não é da mesma classe de P1 e P2.

Portanto no caso da letra c) em que P1 e P2 são não-solucionável, teríamos:

Conjunção → $P1 \wedge P2 \equiv P_{Res}$ é **não-solucionável**;

Disjunção → $P_1 \vee P_2 \equiv P_{Res}$ é **não-solucionável**;

Negação → $\sim P_1 \equiv P_{Res}$ é **parcialmente solucionável ou solucionável**.

d)

- Conjunção: $P_1 \wedge P_2 \equiv P_{\text{resultante}}$ é parcialmente solucionável;
- Disjunção: $P_1 \vee P_2 \equiv P_{\text{resultante}}$ é parcialmente solucionável;
- Negação: $\neg P_1 \equiv P_{\text{resultante}}$ é não – solucionável;

Sejam P_1 e P_2 problemas parcialmente solucionáveis.
Conforme as definições do item (b) acima, as linguagens geradas pelos respectivos problemas apresentam as seguintes características:

$$\begin{cases} \text{Loop}(P_1) = \{\Sigma^* - Lp_1\} \text{ tal que } \Sigma^* - Lp_1 \neq \{\} \\ \text{Aceita}(P_1) \cup \text{Rejeita}(P_1) = Lp_1 \end{cases} \quad \begin{cases} \text{Loop}(P_2) = \{\Sigma^* - Lp_2\} \text{ tal que } \Sigma^* - Lp_2 \neq \{\} \\ \text{Aceita}(P_2) \cup \text{Rejeita}(P_2) = Lp_2 \end{cases}$$

Daí constrói-se o seguinte raciocínio com os operadores lógicos apresentados:

Conjunção
$(\exists P_1\ Loop(P_1) \wedge \exists P_2\ Loop(P_2)) \Rightarrow \exists (P_1, P_2)\ Loop(P_1 \wedge P_2)$. Assim existirá no mínimo uma situação em que o conjunto gerado pela conjunção de P_1, P_2 entra em *Loop*, ou seja, é dito um problema parcialmente solucionável.

Disjunção
$(\exists P_1\ Loop(P_1) \vee \exists P_2\ Loop(P_2)) \Rightarrow \exists (P_1, P_2)\ Loop(P_1 \vee P_2)$. O resultado é análogo aquele encontrado para a conjunção de problemas, entretanto há um detalhe que deve receber especial importância: neste caso se P_1 fosse parcialmente solucionável, não importaria a

natureza do outro problema pois a disjunção seria parcialmente solucionável (sendo o contrário também verdadeiro).

Negação
$\neg(\exists P_1\ Loop(P_1)) \Rightarrow \forall P_1\ \neg Loop(P_1)$. Ao negar-se a existência da condição de *Loop*, encontra-se a expressão "*não é verdade que exista um Loop em* P_1. Logo, a linguagem resultante só pode ser Recursiva, o que evidencia os problemas solucionáveis.

Comentários

As respostas estão corretas. Porém, mudaríamos algumas definições feitas nas respostas. Em vez de

- Conjunção: $P_1 \wedge P_2 \equiv P_{resultante}$ é parcialmente solucionável;
- Disjunção: $P_1 \vee P_2 \equiv P_{resultante}$ é parcialmente solucionável;

Seria

- Conjunção: $P_1 \vee P_2 \equiv P_{resultante}$ é parcialmente solucionável;
- Disjunção: $P_1 \wedge P_2 \equiv P_{resultante}$ é parcialmente solucionável;

4. Ex. 9)

Se S={(x_1,y_2), ...,(x_n,y_n); x_i e y_i são strings sobre Σ } é um Sistema Normal de Post com solução, então uma solução para S é, por definição, quando existe uma seqüência não vazia de inteiros $i_1,...,i_k$ tal que $x_{i1}...x_{ik}=y_{i1}...y_{ik}$.

Observe inicialmente da definição de solução (existem índices $i_1,...,i_k$ tal que $x_{i1}...x_{ik}=y_{i1}...y_{ik}$) que os índices $i_1,...,i_k$ podem variar em m, isto é $1\leq i \leq n$ e $1\leq k \leq m$ (podendo ou não ser m=n) o que significa que os índices podem se repetir ou não assumir todos os valores de n, desde que conserve a propriedade de que $x_{i1}...x_{ik}=y_{i1}...y_{ik}$.

Sejam então CONC(x_{ik}; $1\leq i, k \leq n$) = CONC(y_{ik}; $1\leq i, k \leq n$), onde CONC é a função Concatenação definida anteriormente. Então o conjunto definido por

Π = {(x_i,y_i), $1\leq i \leq n$); | CONC(x_{ik}; $1\leq i, k \leq n$) | = | CONC(y_{ik}; $1\leq i, k \leq n$), | } (1)

dá uma solução para S, Assim são várias as soluções para S; basta que se satisfaça a condição (1).

Então, se $\Pi_1,...,\Pi_k$ são soluções para S basta tomar o min{ Π_j ; $1\leq j \leq k$ }, isto é, min é o menor tamanho das palavras $x_{i1}...x_{ik} = y_{i1}...y_{ik}$. Isso fornece uma menor solução para S.

4. Ex. 10)

a)

Solução de S_1

	1	**2**	**3**
X	b	Babbb	ba
Y	bbb	a	a

O par (x_1, y_1) é o único que inicia com símbolos iguais, portanto ele é necessariamente o primeiro elemento na nossa seqüência. Em seguida, tentamos achar o próximo elemento da seqüência. Tentamos o par (x_2, y_2), que não é possível, já que quebra a igualdade no 3º e 4º elementos (1, 2 $\Rightarrow$ x concatenados = bbabbb e y concatenados = bbba). Daí só nos resta o par (x_3, y_3), que também não é possível, pois quebra a igualdade no 3º elemento (1, 3 $\Rightarrow$ x concatenados = bba e y concatenados = bbba). Portanto, o sistema não tem solução.

Resposta: O sistema não tem solução.

Solução de S_2

	1	**2**	**3**	**4**	**5**	**6**	**7**
X	£	a	b	aa	Ab	ba	bb
Y	a	b	aa	ab	Ba	bb	£

As seguintes seqüências de produções resolvem o sistema proposto

- 7, 6, 2, 1, 1 $\Rightarrow$ (*x concatenados*) bb ba a = bb b a a (*y concatenados*)
- 7, 2, 2, 1, 1 $\Rightarrow$ (*x concatenados*) bb a a = b b a a (*y concatenados*)
- 1, 2, 7, 2, 1 $\Rightarrow$ (*x concatenados*) a bb a = a b b a (*y concatenados*)
- 1, 4, 7, 2, 1 $\Rightarrow$ (*x concatenados*) aa bb a = a ab b a (*y concatenados*)

b)

$S_3 = \{(aab, a), (ab, abb), (ab, bab), (ba, aab)\}$

Construa uma tabela para associar a cada produção um número de identificação.

	1	**2**	**3**	**4**
X	aab	ab	ab	ba
Y	a	abb	bab	aab

Observe que esse sistema não é monogênico, ou seja, existe mais de uma possibilidade para a correspondência.
Devemos encontrar uma combinação de Xi e Yi (i = 1,2,3,4) em que a palavra formada por essa seqüência seja igual.

Começamos com

x2	x4
ab	ba
y2	y4
abb	aab

vemos que para a próxima opção poderemos escolher x2 ou y2;

a) Escolhendo x2 teremos a seguinte seqüência : 2 4 2

Ab	ba	ab
Abb		aab
abb		

Podemos escolher novamente entre x2 e x3

a 1) se x2, teremos a seqüência : 2 4 2 2

Ab	ba	ab	ab
Abb	aab	abb	abb

Não poderemos continuar, pois não há nenhuma palavra em x onde que inicie com a letra b, necessária para continuar a seqüência para Y.

a .2) se x3, teremos a seqüência : 2 4 2 3

Ab	ba	ab	ab
Abb	aab	abb	bab

Também não poderemos continuar, pois não existe nenhuma palavra em x que inicie com a seqüência bb.
Logo excluímos a possibilidade da seqüência 2 4 2, e tentamos então a seqüência
2 4 3

b)

Ab	ba	ab	ba	ba
Abb	aab	bab	aab	aab

Chegamos a seqüência 2 4 3 4 4 e teremos novamente outra bifurcação, devemos continuar assim, até encontrarmos uma seqüência em que a palavra formada por uma combinação de X seja a mesma que a palavra formada pela mesma combinação de Y.
Obs.: a menor seqüência tem 66 índices.

c) Para mostrar que sistema de Post dado não tem solução precisamos provar que não existe uma seqüência não vazia de inteiros i_1, i_2, i_3, i_4, ..., i_k, tal que
$x_{i1}\ x_{i2}\ x_{i3} \ldots x_{1k} = y_{i1}\ y_{i2}\ y_{i3} \ldots y_{1k,}$ o que pode ser feito da seguinte maneira:

$S_4 = \{(ba, bab), (abb, bb), (bab, abb)\}$

	1	**2**	**3**
X	ba	abb	bab
Y	bab	bb	abb

Por ser o único par do sistema que possui símbolos iniciais iguais o par 1 deve ser o inicial. O próximo par é, obrigatoriamente, o par 3, pois é o único que mantém a igualdade. Após isso único par a manter a igualdade continua sendo o par 3, sendo isto repetido infinitamente, sem nunca alcançar o fim do sistema. Por isso, o sistema de Post proposto não é solucionável, mesmo com $|x_{ij}| = |y_{ij}|$, para $1 \leq j \leq k$ e tendo (x_1, y_1) começando com símbolos iguais.

A seqüência gerada pelas produções seria a seguinte:

Regras: 1, 3, 3, 3, 3,...
Seqüência X: ba bab bab bab bab...
Seqüência Y: bab abb abb abb abb...

4. Ex. 11)

Imaginando um ambiente computacional (paralelo ou seqüencial), e analisando a composição de problemas, poderemos determinar se em algum momento o sistema pára gerando uma resposta ou fica rodando infinitamente.

Um exemplo que sempre deve ser lembrado é o algoritmo de Cramer para resolução de sistemas de Equações Lineares: a tabela abaixo compara o algoritmo com outros que realizam a mesma tarefa:

$$\begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \vdots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix} \times \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}$$

Lembrando que o algoritmo de Cramer calcula a determinante da matriz dos coeficientes e mais os n determinantes Δx_i da coluna i-ésima coluna da matriz dos coeficientes pelo vetor dos termos independentes. Desta forma, pelo teorema de Cramer a solução $(x_1, x_2, x_3, \dots x_n)$ é dada por:

$$x_i = \frac{\Delta x_i}{\Delta}$$

Tal algoritmo acarreta em, no mínimo, $(n+1)!(n-1)$ operações aritméticas. Levando-se em consideração o tempo gasto para cada operação (adições e subtrações: 40μs e Multiplicações: 400μs), tem-se o seguinte quadro:

	$n = 5$	$n = 10$	$n = 20$
Cramer (det. Pela definição)	2.5s	3.4 dias	20 Bilhões de Anos
Cramer (det. por Laplace)	0.4s	6 min	5 meses
Gauss – Método de Eliminação	36ms	0.22s	1.5s

Agora por exemplo, tomando como entrada um número n para entrada, com n superior a 20. Podemos construir um algoritmo correto (que efetivamente computa a função determinada) entretanto não poderemos estimar, em tempo de execução, se o algoritmo dado é ou não certo em relação a parada do mesmo. Daí o estudo da solucionabilidade de problemas em termos de linguagens ganha tônus e importância.

Comentário :

O exemplo dado acima mostra que, além de sabermos se um problema é solucionável teoricamente ou não por algum algoritmo, devemos também avaliar se este algoritmo é eficiente o bastante em termos da realidade em que o tratamos, ou seja, estudar a solucionabilidade do problema em levando em consideração as linguagens e o tempo de parada do algoritmo, sendo que este tempo deve estar dentro de um certo intervalo estimado .