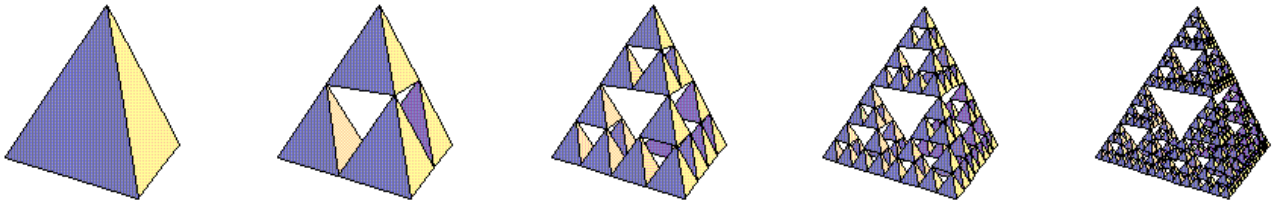
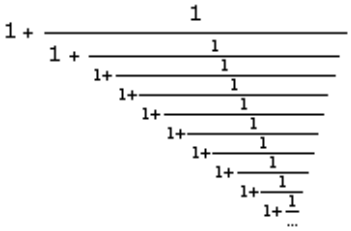
Capítulo VII : A Recorrência

Quando algo é definido em termos de si próprio.

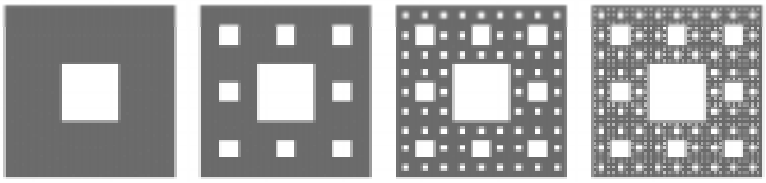
Ex1: O Tetraedro de Sierpinski



Ex2: Fracções Contínuas



Ex3: A Carpete de Sierpinski



Ex4: A Instrução Composta em Pascal

Ex5: A Função Factorial

$$n! = \begin{cases} n \cdot (n-1)! & \text{se } n > 0 \\ 1 & \text{se } n = 0 \end{cases}$$

Algoritmo Recorrente:

$\forall n \in \mathbb{N}_0$

se n = 0
então n! ← 1
senão n! ← n * (n-1)!

Função Pascal Recorrente:

```
function factorial (n : intnneg) : intnneg;

    begin if n=0
          then factorial:= 1
          else factorial:= n * factorial(n-1)
    end;
```

Atribuição do Valor ao Identificador da Função (sem o Parâmetro)

Chamada da Função numa Expressão (com o Parâmetro)

- **O Bloco da Função pertence ao Domínio do seu Identificador.**
- **Por isso, é possível utilizar esse Identificador (Chamar a própria Função) dentro do Bloco da sua Declaração.**

Simulação:

```
factorial(4)
n=4
factorial ← 4 * factorial(3)
                n=3
                factorial ← 3 * factorial(2)
                                n=2
                                factorial ← 2 * factorial(1)
                                                n=1
                                                factorial ← 1 * factorial(0)
                                                                n=0
                                                                factorial ← 1
                                                factorial ← 1 * 1 = 1
                                factorial ← 2 * 1 = 2
                factorial ← 3 * 2 = 6
factorial ← 4 * 6 = 24
```

Comentários:

Número de Chamadas da Função: n+1

Número de Multiplicações: n

Espaço de Memória utilizado:

- **Cada Chamada gera uma cópia do Parâmetro** n **(da Classe Valor);**
- **O Valor de** factorial, **calculado por cada Chamada da Função, ocupa um Registo de Memória;**
- **Total:** 2(n+1)

A utilização da Recorrência no cálculo da Função Factorial:

- **Conduz a uma solução simples e elegante;**
- **É pouco eficiente, em particular, em termos da Memória utilizada.**

(Ver Cap. VI, pág. 27)

Ainda menos eficiente ...

Ex6: A Sucessão de Fibonacci

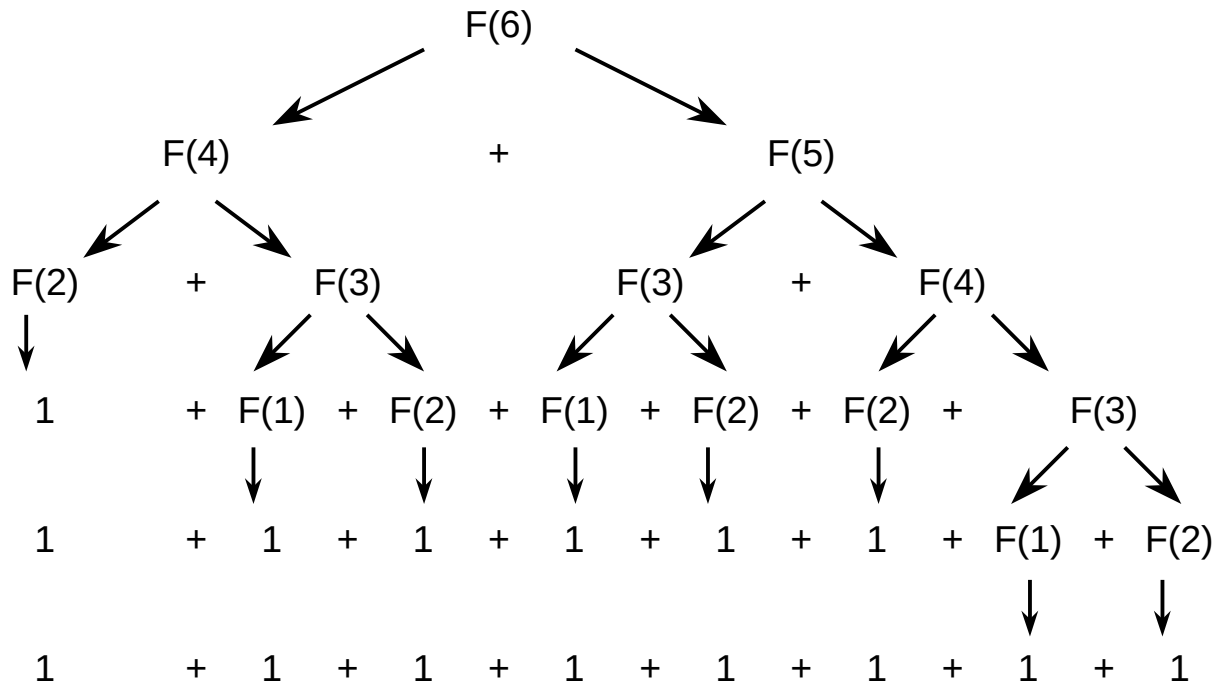
1, 1, 2, 3, 5, 8, 13, 21, 34, ...

$$F_n = \begin{cases} F_{n-2} + F_{n-1} & \text{se } n > 2 \\ F_1 = F_2 = 1 \end{cases}$$

Função Pascal Recorrente:

```
function  fibonacci (n : intpos) : intpos;
          begin  if n<=2
                 then  fibonacci:= 1
                 else  fibonacci:= fibonacci(n-2) + fibonacci(n-1)
          end;
```

Simulação: F(6) = 8



Comentários:

- **Solução simples, clara e elegante;**
- **Mesmo nada eficiente, nem em termos das Operações realizadas, nem do número de Chamadas, nem da utilização de Memória;**
- **A definição recorrente pode contudo facilitar a elaboração de uma solução iterativa eficiente.**
  Exercício: Versão iterativa só com 3 variáveis.

A versão recorrente de um Algoritmo é, por vezes, mais clara e concisa do que a correspondente versão iterativa.
Como, por exemplo:

Ex7: Algoritmo de Euclides

O Algoritmo de Euclides, para o cálculo do Máximo Divisor Comum de dois números inteiros não-negativos, baseia-se nas seguintes Propriedades:

$$mdc(a,b) = mdc(b, a \bmod b) \qquad \text{se } b \neq 0$$
$$mdc(a,0) = a$$

Função Pascal Recorrente:

```
function  mdc (a, b : intnneg) : intnneg;
          begin  if b = 0
                    then  mdc:= a
                    else  mdc:= mdc(b, a mod b)
          end;
```

- **A solução Recorrente é mais clara do que a versão Iterativa e mais fácil de elaborar, directamente a partir da teoria;**
- **A perda de eficiência é, neste caso, compensada pela Clareza.**

O modo de utilização de Memória das soluções Recorrentes pode tornar-se bastante útil...

Ex8: Inverter uma palavra

Ler uma palavra, terminada por um espaço, e escrever as suas letras por ordem inversa.

Algoritmo Iterativo:

ler cada letra, até espaço, registando num vector;
escrever letras por ordem inversa.

'S'	'T'	'A'	'R'	' '								

(Desperdício de espaço de Memória, provocado pela declaração do vector)

Algoritmo Recorrente:

ler uma letra;
se letra ≠ espaço
então Inverter o resto da palavra;
escrever letra.

Procedimento Pascal Recorrente:

```
procedure inverter;
          var letra : char;
          begin read(letra);
                if letra <> ' '
                then inverter;
                write(letra:1)
          end;
```

Simulação:

```
inverter:  ler('S');
           inverter: ler('T');
                     inverter: ler('A');
                               inverter: ler('R');
                                         inverter: ler(' ');
                                                   escrever(' ');
                                         escrever('R');
                               escrever('A');
                     escrever('T');
           escrever('S');
```

Espaço de Memória utilizado: # letras + 1

Cada chamada gera uma cópia da variável local letra, onde são registadas as sucessivas letras da palavra. Neste caso, a versão recorrente é mais eficiente, em termos de utilização de memória.

Ex9: Inverter um número

Escrever, por ordem inversa, os dígitos de um dado inteiro positivo.

Algoritmo Recorrente:

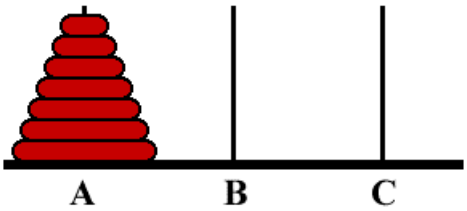
escrever o último dígito (unidades);
se restarem dígitos
então Inverter o resto do número.

Procedimento Pascal Recorrente:

```
procedure inverter(numero : intpos);
          begin write(numero mod 10 : 1);
                if (numero div 10) <> 0
                then inverter(numero div 10)
          end;
```

A solução recorrente de um problema é, por vezes, a mais fácil ...

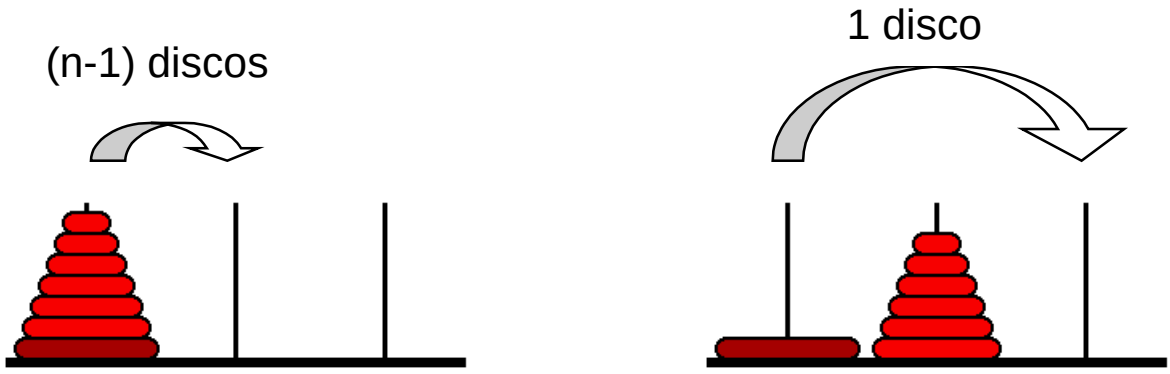
Ex10: Torres de Hanoi (Edouard Lucas, 1883)



Deslocar os (7) discos da Torre A para a Torre C, um de cada vez. Utilizar a Torre B como auxiliar, mas nunca colocar um disco, sobre outro menor.

Algoritmo Recorrente:

- **n = 1 : sabemos deslocar 1 disco;**
- **Como deslocaríamos n discos, se soubessemos deslocar n-1 discos?**



Portanto:

Mover n discos de A para C, usando B como auxiliar:
Mover n-1 discos de A para B, usando C como auxiliar;
Mover 1 disco de A para C;
Mover n-1 discos de B para C, usando A como auxiliar.

Ou, melhor:

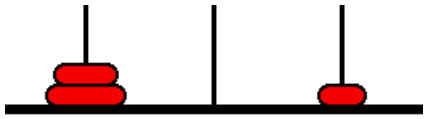
MoverTorre(n, origem, auxiliar, destino):
MoverTorre(n-1, origem, destino, auxiliar);
MoverDisco(origem, destino);
MoverTorre(n-1, auxiliar, origem, destino).

Programa Pascal:

```
program TorresdeHanoi(input, output);
type  torre = 'A'..'C';
      intnneg = 0..maxint;
var n : intnneg;

procedure  MoverDisco(desde, para : torre);
           begin writeln(desde:1, '->', para:1)
           end;

procedure  MoverTorre(n : intnneg; origem, auxiliar, destino : torre);
           begin if n>1
                 then begin  MoverTorre(n-1, origem, destino, auxiliar);
                             MoverDisco(origem, destino);
                             MoverTorre(n-1, auxiliar, origem, destino)
                      end
                 else MoverDisco(origem, destino)
           end;

begin writeln('Quantos discos tem a Torre de Hanoi?');
      readln(n);
      MoverTorre(n, 'A', 'B', 'C')
end.
```

Simulação:

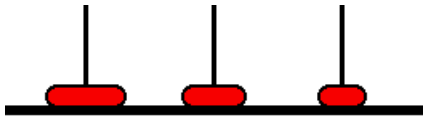
MoverTorre(3, A, B, C):



MoverTorre(2, A, C, B):

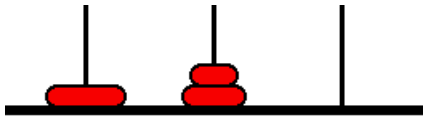
MoverTorre(1, A, B, C):

MoverDisco(A, C);



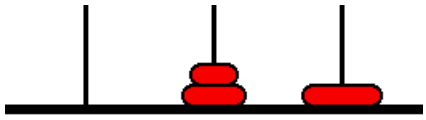
MoverDisco(A, B);

MoverTorre(1, C, A, B):



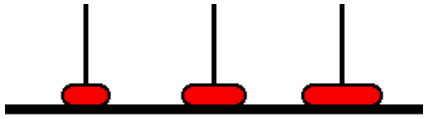
MoverDisco(C, B);

MoverDisco(A, C);



MoverTorre(2, B, A, C):

MoverTorre(1, B, C, A):

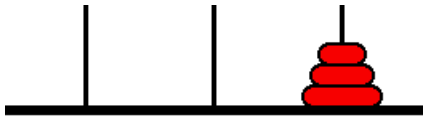


MoverDisco(B, A);

MoverDisco(B, C);

MoverTorre(1, A, B, C):

MoverDisco(A, C);



Teorema: **O número de Movimentos simples necessários para mover uma Torre de Hanoi com n discos:**

$$M(n) = 2^n - 1$$

Demonstrar: **(Por Indução ...)**

Exercícios:

- Elabore um algoritmo recorrente para calcular M(n).
- Elabore um algoritmo iterativo para calcular M(n).

Questões associadas à "lenda" das Torres de Hanoi:

- Se um monge budista demorar um segundo para deslocar um disco, quanto tempo será necessário para mover a Torre completa (64 discos)?
- Qual o valor estimado para a esperança de vida do Sol?
- Construir a solução iterativa do problema das Torres de Hanoi.

A Construção de Soluções Recorrentes:

1. **Como definir o problema em termos de um problema menor e do mesmo tipo?**
2. **Como diminui o tamanho do problema, em cada chamada recorrente?**
3. **Qual o caso particular do problema, que pode servir para a paragem? Verificar se o modo como diminui o tamanho do problema, em cada chamada, assegura que o caso base (paragem) é sempre atingido.**

Exercícios: **Elaborar soluções puramente recorrentes para contruir cada uma das figuras seguintes (não utilizar nenhum ciclo).**

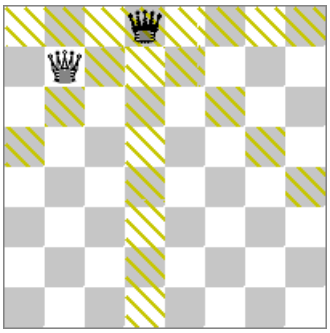
```
        1
      1 2 1
    1 2 3 2 1
  1 2 3 4 3 2 1
1 2 3 4 5 4 3 2 1
```

```
        1
      1 2 1
    1 2 3 2 1
  1 2 3 4 3 2 1
1 2 3 4 5 4 3 2 1
  1 2 3 4 3 2 1
    1 2 3 2 1
      1 2 1
        1
```

Em certos casos, quando não é conhecido um algoritmo iterativo, é necessário gerar um processo recorrente de Pesquisa Exaustiva (Backtracking).

Ex11: O Problema da 8 Rainhas

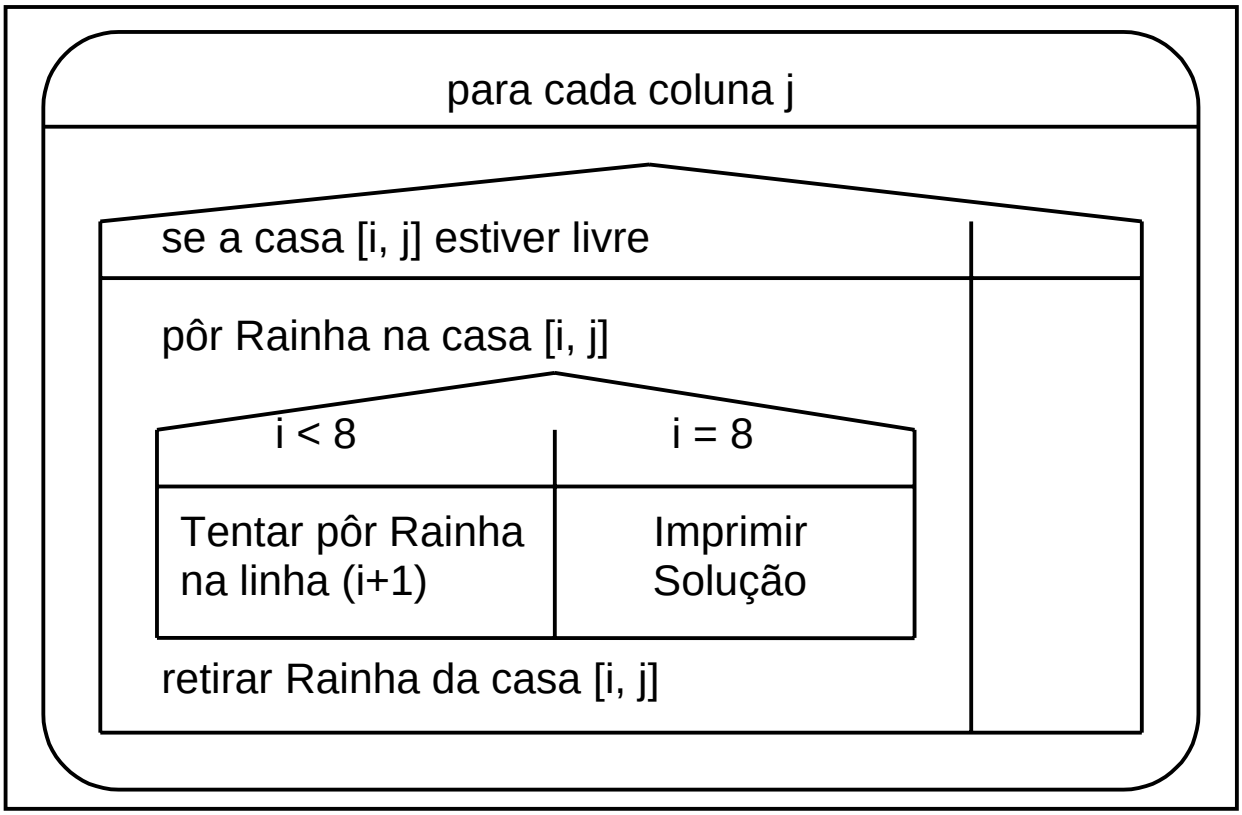
Num tabuleiro de Xadrez, a Rainha domina toda a linha, a coluna e as duas diagonais da casa onde estiver colocada.



Como colocar 8 Rainhas (não dominadas) num tabuleiro? (Existem 92 soluções possíveis)

Processo de Backtracking:

Tentar pôr Rainha na linha (i):



Como assinalar as casas que estão, ou não, livres?

Consideremos um conjunto de vectores do tipo boolean, onde:

true ⇒ casa livre
false ⇒ casa dominada

Para as colunas:

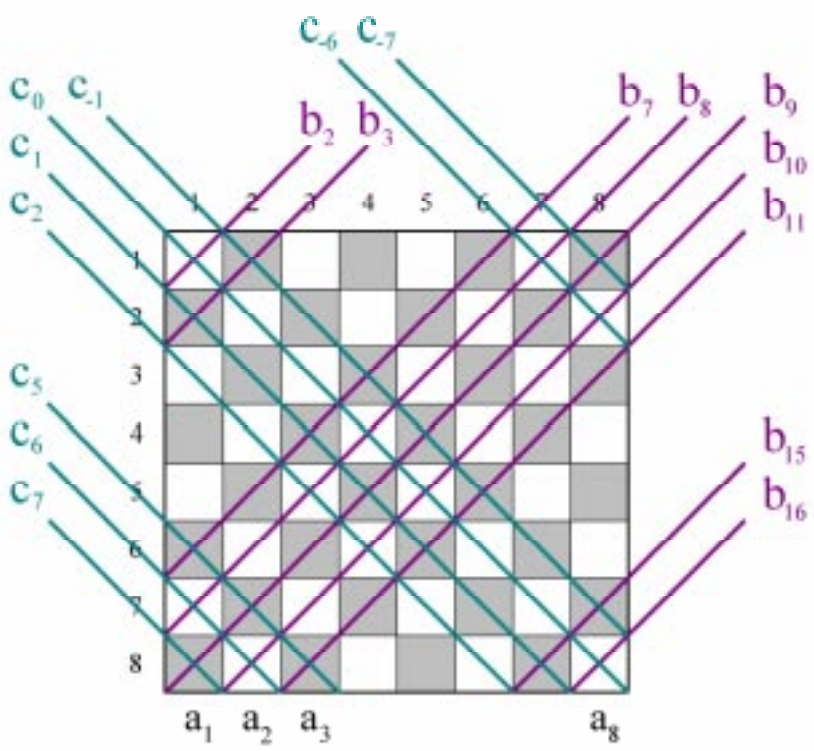
var a : **array** [1..8] of boolean;

Para as diagonais ascendentes (i+j):

var b : **array** [2..16] of boolean;

Para as diagonais descendentes (i-j):

var c : **array** [-7..7] of boolean;



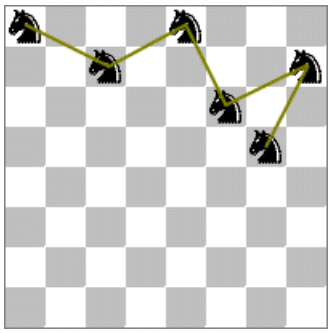
O processo tenta, sucessivamente, colocar uma Rainha em cada linha.
Para registar a posição (coluna) ocupada por cada Rainha basta:

var x : **array** [1..8] of 1..8;

```
program OitoRainhas(input, output);
type linha, coluna = 1..8;
var a : array [coluna] of boolean;
var b : array [2..16] of boolean;
var c : array [-7..7] of boolean;
var x : array [linha] of coluna;
var  i : integer;

procedure   EscreverSolucao;
            begin  ... (* Como? *) ...
            end;

procedure  Tentar(i : linha);
           var j : coluna;
           begin  for  j:= 1 to 8 do
                       if a[j] and b[i+j] and c[i-j]
                       then  begin  (* Pôr Rainha em [i, j] *)
                                    x[i]:= j;
                                    a[j]:= false;
                                    b[i+j]:= false;
                                    c[i-j]:= false;
                                    if i<8
                                    then Tentar(i+1)
                                    else EscreverSolucao;
                                    (* Retirar Rainha de [i, j] *)
                                    x[i]:= 0;
                                    a[j]:= true;
                                    b[i+j]:= true;
                                    c[i-j]:= true
                             end
           end (* Tentar *);

begin   for i:= 1   to 8   do  a[i]:= true;
        for i:= 2   to 16  do  b[i]:= true;
        for i:= -7  to 7   do  c[i]:= true;
        for i:= 1   to 8   do  x[i]:= 0;
        Tentar(1)
end.
```

Exercício: Circuito de Cavalo

Outro problema clássico associado ao Xadrez consiste em, partindo de uma dada casa e sempre em "Salto de Cavalo", percorrer cada uma das casas do tabuleiro uma só vez, acabando na casa inicial.



Construa uma Solução Recorrente para este problema.

Problemas que permitem uma definição recorrente, conduzem naturalmente a soluções recorrentes ...

Ex12: Notação Polaca (Lukasiewicz, 1878-1956)

Na notação habitual (*infix*) das Expressões Aritméticas, cada operador **é colocado entre os dois** operandos **correspondentes.**

a + b * c

Para saber a ordem por que são efectuados os cálculos, é necessário estabelecer um conjunto de Regras de Prioridade entre os operadores **e (para alterar essa ordem) é também necessário o uso de Parênteses.**

Na Notação Polaca (*postfix*), cada operador **é colocado depois dos dois** operandos **correspondentes.**

A ordem dos cálculos a efectuar fica univocamente definida, sem Regras de Prioridade nem Parênteses.

Exemplos:

infix	*postfix*
a + b	a b +
(a + b) * c	a b + c *
(a + b) * (c - d)	a b + c d - *
a + b * c - d	a b c * + d -
a + b * (c - d)	a b c d - * +
(a + b) * c - d	a b + c * d -

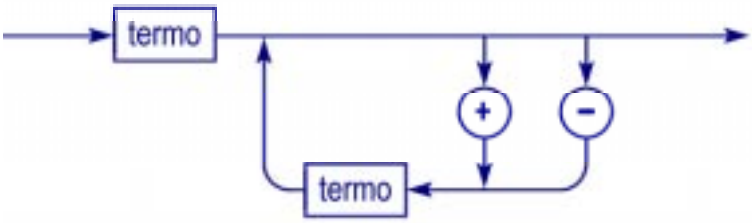
A Notação Polaca é utilizada na Compilação das Expressões Aritméticas escritas nas Linguagens de Alto Nível.

Problema: Conversão *infix* →*postfix*

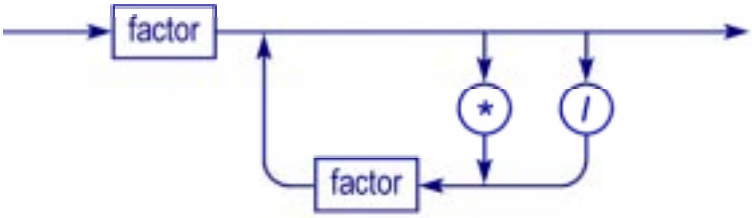
Vamos considerar "uma" Linguagem de Alto Nível, que consiste numa simplificação da Linguagem Pascal.

Definições:

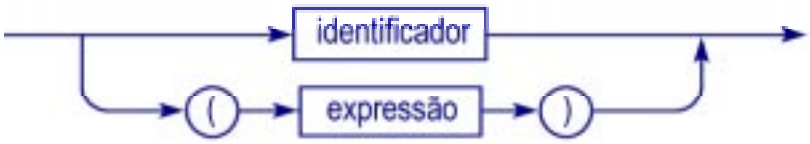
Expressão:



Termo:



Factor:



Identificador: