INTRODUÇÃO À LINGUAGEM PRISMA

Manual básico

Por Adalberto Amorim Felipe

maio/2015

Introdução

Este material é para iniciantes, ou para quem tem dificuldades em adentrar nos conceitos técnicos da linguagem Prisma. Portanto, é para ser lido com paciência percorrendo em detalhes a linguagem, ou para ser consultado na parte que mais interessa. Ele começa bem do básico mesmo e a tendência é ir ao avançado, espero que seja útil.

Em primeiro lugar você deve conhecer alguns aspectos sobre a linguagem Prisma:

O que é Prisma?

É um conjunto que inclui:

1 – linguagem de programação e suas regras de comandos;

2 – interpretador;

3 – bibliotecas e arquivos externos.

Vejamos cada item anterior:

linguagem de programação e suas regras de comandos:

Digamos que qualquer linguagem possa ser inventada, basta você determinar as regras, as palavras reservadas funções da biblioteca padrão, enfim, o modo como escrevê-la, os símbolos permitidos e para que servem, o que pode e o que não pode.

É exatamente as regras e mecanismos de funcionamento da linguagem que você deve dominar para fazer os programas, os aspectos da linguagem. Esses aspectos são a essência da linguagem e Prisma tem isso, podemos citar uma característica importante dela que possui comandos em português ao invés do inglês que é de costume.

Definidos teoricamente os aspectos da linguagem, basta criar um meio de fazê-la funcionar, afinal, não queremos algo na teoria somente mas na prática também.

Interpretador:

Diversas linguagens possuem como mecanismo de execução a compilação do código-fonte para código nativo do processador.

Prisma é diferente, ela possui um mecanismo de interpretação, isto é, há o programa feito em C que interpreta os comandos do código-fonte ou código-byte. O interpretador pode ser chamado da seguinte forma por linha de comando:

prisma.exe programa_prisma.prisma, na sequência temos o carregamento do arquivo fonte chamado 'programa_prisma.prisma' que é convertido internamente para byte-codes (códigos em bytes) e após isso é executado. Pode-se, ainda, compilar o código-fonte do programa prisma para byte-codes usando o prismac, seu modo de chamar é:

prismac -o programa_compilado.pbrexe programa_fonte.prisma, será gerado um arquivo binário chamado

programa_compilado.pbrexe.

Mas o que é programa fonte, ou código-fonte? Simples, todo programa em linguagem de alto-nível é escrito, isto é, o

programa é feito em frases, geralmente frases no imperativo, por exemplo, escreva 'ola', x = leia ( ); ig.feche ( );

estas frases devem seguir as regras da linguagem. Portanto o código fonte é o arquivo texto que contém as frases dos comandos do programa que se quer fazer. O prismac.exe converte esse texto para código byte, isto é, as instruções (frases) são traduzidas para zeros e uns (bytes) cuja organização pode ser compreendida e interpretada pelo prisma.exe (o interpretador);

Enfim, o interpretador é o que torna a execução de um programa prisma possível, sem ele não há como executar os programas feitos nesta linguagem. Várias outras linguagens são assim, podemos citar Python, C#, Java, Ruby, Lua, Lisp, Php entre outras. O fato de uma linguagem ser interpretada garante algumas vantagens e perde outras, perde principalmente no uso da memória e velocidade, mas ganha em portabilidade e simplicidade na criação dos softwares.

Prisma é, na verdade, uma modificação dos fontes de Lua, uma linguagem brasileira de fama internacional muito usadas em jogos, pois é das linguagens de scripts a mais veloz e leve. Como Prisma tem nela sua origem herda esses traços, sendo mais leve que Java, por exemplo, mais simples de se aprender que Python. Prisma tem um enorme potencial que se encontra no fato de quase tudo feito em C pode ser estendido a Prisma. Por exemplo, é simples implementar funções do gtk em Prisma, assim como em Lua.

Bibliotecas e arquivos externos

Nem todo recurso necessário será encontrado no interpretador Prisma diretamente. Por exemplo, se você quiser fazer um programa com janelas e botões, não conseguirá fazer isso somente em prisma, deverá usar funções gráficas que estão em outro programa chamado **igbr,** que é uma biblioteca em prisma.

Bibliotecas servem para isso, criar funções que ainda não existem em prisma estendendo novas funcionalidades. Inclusive, você pode criar novas bibliotecas, seja na própria linguagem Prisma ou em C. As bibliotecas chamadas também de módulos podem ter as seguintes extensões: .pris para módulo em prisma (ou _pris.dll) , e .dll para módulos nativos em C. (para módulos em C é necessário saber programar em C e compilar dlls);

(Em linux essas extensões podem alterar para _pris.so e .so);

Adiante veremos com detalhe como fazer suas próprias bibliotecas.

Por fim, você precisa saber que para programar em Prisma deve estudar a linguagem – a parte do manual que trata da linguagem em si. Já para programar com um tipo específico de biblioteca precisa estudar sobre a biblioteca a ser usada, seja ela um módulo de funções gráficas, base de dados, ou network, etc.

Aqui mais especificamente veremos sobre a linguagem em si!

Unidade I – Introdução à linguagem

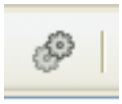
Cap. 1 – Primeiro programa: Função imprima() e leia() , strings, números e comentários

Caso não tenha instalado prisma vá para a página de downloads e primeiros passos.

Vamos programar então:

Com o editor aberto, crie um arquivo, salve-o com o nome de ola.prisma em uma pasta de sua escolha e digite o seguinte comando dentro dele :

```
1 imprima  "Ola mundo em prisma!";
```

 Execute-o pressionando o botão executar que se encontra na barra de ferramentas do prismacod ou no menu ferramentas->executar.

Sim, uma única linha aqui já é um programa, que exibe a mensagem Ola mundo em prisma! na janela de comandos.

Vejamos, você aprendeu aqui uma função *imprima* (que imprime “frases na tela”), esse texto entre aspas é chamado

em programação de string, que em inglês significa corda ou cadeia, ou seja, no sentido de sequência de caracteres.

O interpretador não interpreta nada do que está dentro de uma string, ela é tratada como um objeto que se pode

manipular e ser exibido na tela.

A função imprima dispensa o uso do parênteses, assim como qualquer outra função, na condição de que tenha um único parâmetro e seja uma string (Tabelas como parâmetro único também dispensam o uso de parênteses).

reescreva o comando anterior da seguinte forma:

```
imprima ("Ola mundo em Prisma!"); // execute e veja que o resultado será o mesmo.
```

tente agora:

```
imprima ( "Ola mundo \n em \n Prisma" ); //execute
```

Deve ter notado uma grande diferença, cada trecho saiu em uma linha diferente, isso foi devido ao **\n** que é chamado de caractere de escape nova linha, quando há um desses dentro de uma string ele não é impresso, ao invés disso o texto vai para linha seguinte. Há diversos caracteres de escape, veremos adiante.

Alguns detalhes dos últimos dois comandos:

note que na frente do comando escrevemos duas barras oblíquas **// e em seguida uma frase.** Essas barras indicam que o que estiver depois delas até o fim da linha não deve ser interpretado, logo, o interpretador ignora esta parte do programa, isto é chamado de comentário, eles são muito úteis em programas de grande porte onde você deve lembrar para que serve cada comando criado, descrevendo explicações nos comentários.

Outra forma de fazer comentários é usando a sequência **/**(para abrir) e ** (para fechar),** com isso pode-se criar comentários de várias linhas sem precisar colocar as **//** em cada uma delas, ex.:

Prisma

```
/**

este é um comentário

multi-linha! Esta parte do código não será executada

**
```

A função imprima pode mostrar não só textos entre aspas como também números diretamente, veja:

```
imprima ( 1000000 ) ; //imprime o número um milhão na tela de comandos
```

Tente este outro comando:

```
imprima ( 444 - 100 ); //imprime o número 344, que é o resultado de 444 menos 100
```

*obs. tente mudar os números e a operação, os sinais básicos de matemática são: / divisão, * mutilplicação, + soma, – subtração.*

Note que prisma primeiro executa a operação entre parênteses até obter um único resultado para depois imprimir na tela.

 *Dica: no final de cada comando coloque um ponte-e-vírgula para separar uns dos outros, veja:*

```
imprima( 5 ) ; imprima ( 6 ) ; imprima( "Numeros 5 e 6");
```

Sem os pontos-e-vírgulas, a execução quase sempre acontecerá sem erros, mas é bem mais organizado com o uso do ponto-e-vírgula.

Até agora aprendemos a mostrar textos e números na janela de comandos (tela preta);

Vejamos uma função diferente do **imprima ()**, a função **leia(),** veja:

```
//exemplo do uso da função leia(), salve este exemplo como: leia.prisma

  x = leia( );
// aguarda o usuário digitar algo e armazena o resultado em x
// depois da tecla enter ser pressionada ;

imprima ( "Voce digitou : " , x );

//fim do programa
```

Note que ao executar, o cursor ficou piscando dentro da tela de comandos do windows ou terminal do linux, esperando você digitar algo. Ao digitar e apertar a tecla Enter o resultado ficou gravado na palavra **x**, e depois é impresso com a função imprima.

Esse efeito de gravar um resultado em um nome é chamado de atribuir uma variável (no caso o ***x,*** *mas poderia ser qualquer outro nome), veremos sobre variáveis mais adiante.*

Talvez você tenha notado que usamos uma vírgula dentro da função imprima desta vez, isso é porque temos dois argumentos colocados dentro da função, primeiro o texto : **"Voce digitou : "** e segundo a variável **x,** então a função mostra o texto e na sequência o que está armazenado em **x;** Veja que não é impresso o x mas o valor que foi digitado, este é o conceito de variável.

Obs. note que variável é um nome criado sem aspas, tudo o que tiver entre aspas geralmente é uma string.

Obs. Strings em prisma podem ser feitas de outras duas maneiras além do uso de aspas duplas:

1 – usando aspas simples : 'isto é uma string\n';

2 – usando duplos colchetes, permitindo uma string multi-linha, não aceita caracteres de escape, se tiver algum ele será impresso

na tela como uma string:

Prisma

```
texto = [[

isto é uma string

de várias linhas

o texto será impresso exatamente como está aqui!

]]

imprima ( texto );
```

Há outra forma de fazer strings multilinhas:

Prisma

```
texto = "isto é uma string\
de várias linhas\
o texto será impresso\
exatamente como está aqui!"
```

```
5
6 imprima( texto );
```

Para esse estilo dar certo é necessário, após a barra, o fim da linha. Para isso pressione enter após ela, caso tenha um único espaço, não dará certo. Note que o mesmo sinal que abriu a string (“) no início, é o que fechou no final (“).

Cap. 2 – Palavras reservadas, operadores relacionais e lógicos

Anteriormente, nós vimos que além de outras coisas, uma linguagem é composta por palavras reservadas, mas o que será isso.

Palavras reservadas: são palavras-comando da própria linguagem, não podendo ser usadas com outro propósito:

Português // inglês:

```
1      "e", //and
2      "quebre", //break
3      "inicio", //do
4      "senao", //else
5      "senaose",//elseif",
6      "fim", // "end",
7      "falso", // "false",
8      "para" , // "for",
9      "funcao" , // "function",
10     "vapara", // "goto",
11     "se", //   "if",
12     "em", //  "in",
13     "local",
14     "nulo", // "nil",
15     "nao", // "not",
16     "ou", // "or",
17     "repita", //  "repeat",
18     "retorne", //  "return",
19     "entao", // "then",
20     "verdadeiro", // "true",
21     "ate", // "until",
22     "enquanto" , // "while",
23     "este" // this / self
```

Não se preocupe em decorá-las, você aprenderá a usá-las no decorrer dos estudos, só estão listadas aqui para ilustração.

Note que devemos usar a palavra-chave tal como ela se apresenta, ex:

use o **se** e não **SE ou Se ou sE,** Prisma é case sensitive, isto é, diferencia maiúsculas de minúsculas.

Palavras reservadas e funções são coisas diferentes, as palavras são fixas e imutáveis, a não ser,

que se modifique a própria linguagem, já as funções podem ser criadas modificadas e atribuídas. logo:
'enquanto' é de categoria diferente de 'imprima' (uma é palavra-chave e outra função);

————————————————————————————

Além das palavras reservadas, temos os sinais e suas funções específicas dentro da linguagem.

Sinais especiais e operadores relacionais:

```
1    ..    ...    ==    >=    <=    <>    <    >
```

obs.: quando usar os sinais compostos jamais deixe espaço entre eles, ex.: (**< =** //errado) (**<=** //certo)

veja o uso deles respectivamente:

concatenar strings / argumentos indefinidos para funções / testa igualdade / maior ou igual a /

menor ou igual a / diferente de / menor que / maior que

Exemplos de como usar:

'..'

Prisma

```
1   x = "Ola" .. " " .. "Mundo" .. " em Prisma!!!";
2  //une as strings e armazena o resultado na variável x ficando assim:
3
4  // x = ( "Ola Mundo em Prisma!!!" );
```

————————————————————————————————————-

‘...’

Prisma

```
funcao escreva (...) //aceita número variável de parâmetros

      imprima( ... ); //imprime os parâmetros passados

fim

escreva ( "ola" , "mundo" , "em" , "Prisma");
```

Se não conseguiu entender completamente os ... não tem problema, o veremos mais adiante, e além disso, não são muitas as situações que você precisará usar no momento.

Prisma

```
// poderia converter a sequência de parâmetros em tabela (matriz):
//   args = { ... }; ex.:

funcao escreva( ... )

           t = { ... };

           para i = 1 , #t  inicio
                    imprima( t[i] );
           fim

fim

escreva( "ola" , "mundo" , "em" , "prisma");
```

————————————————————————————

```
==       >      <       <=       >=      <> ;
```

(são muito úteis para testes de condições, retornam verdadeiro ou falso)

veja:

Prisma

```
a = 1; b=2; // atribuindo valores às váriaveis

//não confunda o operador de atribuição '=' com o de teste de igualdade '=='

se a == b entao

                  imprima("iguais");

senaose a <> b entao

               imprima ("diferentes");

senaose a < b entao

              imprima( "a menor que b");

senaose a > b entao
```

```
              imprima ("a maior que b");

senaose a <= b entao

            imprima ( "a menor ou igual a b");

senaose a >= b entao

           imprima( "a maior ou igual a b");

senao

           imprima ( "nenhuma das alternativas");

fim
```

Operadores lógicos : *ou , e , nao*

(em especial, dê uma atenção ao operadores lógicos, verá o quanto será recompensador, pois problemas complexos serão facilmente resolvidos com o uso de um operador lógico, ok!)

ou :

O **ou** retorna o primeiro valor válido dentre vários, ou então nulo caso nenhum seja válido, muito útil para testar se uma variável já existe e atribuir ou não um valor a ela. Veja:

Prisma

```
 x = x ou 1;
imprima(x);
```

*Interprete assim: **x** é igual ao próprio **x** se ele já tiver valor ou a 1 se ele ainda não tem valor, isto é, se é igual a nulo.*

Você pode colocar vários operadores '**ou**' em sequência, veja:

Prisma

```
x = T  ou B ou "Eu sou valido";

imprima ( x );
//caso T tivesse um valor atribuído x seria T, mas T no caso é nulo, então

//o teste passa para B, que também é nulo, logo o teste 'ou'
// passa para "Eu sou valido" que é uma string, isto é,

//válida, portanto x pega entre os três o terceiro valor "Eu sou valido";
```

Se você testar vários valores e todos forem nulo, o retorno é nulo.

Se você testar vários valores e um for falso e o restante for nulo, então falso é atribuído, veja:

Prisma

```
z = falso; b = nulo; t = nulo; k = nulo;

x = b ou t ou k ou z;

imprima( x ) ; //saída será falso;
```

———

*O **ou** pode ser usado em testes de condições com o comando **'se'** que veremos mais adiante, veja:*

Prisma

```
a = 1 ; b = 3;

se a > 0 ou 3 > 0 entao

        imprima(" a ou b e maior que zero \n");

senao

        imprima("Nem a nem b e mairo que zero \n");

fim
```

e :

O operador 'e' retorna o primeiro valor inválido (falso ou nulo), caso contrario retorna o próximo até encontrar um nulo ou falso.

Prisma

```
x = "valor valido" e TT ;
//TT é nulo já que não atribuímos ela ainda

imprima (x); //saída nulo

a = falso; b = 2; // somente b é valido

x = b e a;

imprima( x); //saída falso

x = falso e nulo;

imprima (x); //saída falso
```

Caso não tenha nenhum valor inválido o último valor é retornado, veja:

```Prisma
x = 2 e 4 e 5;

imprima (x) // saída 5
```

Podemos usar o operador ‘e’ em um teste de condição, neste caso ambos os testes devem ser válidos para a condição ser satisfeita:

```Prisma
a = 1;  b = 10;

se a < 100 e b < 100 entao
     imprima"os dois numeros sao menores que 100\n";
senao
     imprima "Um dos números nao e menor que 100\n";
fim
```

nao :

O operador lógico ‘nao’ é unário isto é não exige dois valores para a operação, apenas um, antecedendo-o, veja:

x = **nao** y;

imprima(x);

Este operador retorna verdadeiro ou falso:

Caso o valor seja válido, ele retorna o oposto, ou seja, falso.

Caso o valor seja inválido, isto é, se o valor for nulo ou falso, ele retorna verdadeiro;

ex.:

```Prisma
imprima ( nao nulo ); //saída - verdadeiro;

imprima( nao falso ); //saída - verdadeiro;

imprima ( nao verdadeiro ); //saída - falso;

a = 1;

imprima ( nao a ); //saída - falso;
```

É muito útil em condições:

```
a = es.abra( 'arquivo.txt' , 'leitura' );

se nao a entao imprima 'erro ao tentar abrir arquivo' fim
```

Caso haja um erro ao tentar abrir o arquivo o retorno é nulo, na condição o nao retorna o contrario do valor inválido que é

'verdadeiro', logo a condição é satisfeita e o erro é impresso.

Cap. 3 – Variáveis, tipos e a função tipo()

Ao se trabalhar com uma linguagem de programação temos que saber quais são os tipos suportados, isto é, que dados podem ser usados nesta linguagem. Já vimos dois tipos de dados nos primeiros programas, o tipo string "texto entre aspas" e o tipo número 1000000 por exemplo.

Dados podem ser 'apelidados', para que ao invés de passarmos o dado em si, passemos o apelido, que é chamado de variável em programação, por exemplo se quisermos chamar o valor 1200.00 de salario, salario será a variável que pode ser atribuída da seguinte forma: salario = 1200.00, o '=' é o operador de atribuição, podemos até mudar o valor, basta reatribuir a variável:

salario = 2000.00, desta forma onde usarmos a palavra salario, será passado o valor atribuído nela, ex.:

```
imprima( salario );
// mostrará na tela o número armazenado na variável salario.
```

O que são variáveis afinal? *São nomes criados pelo programador para simbolizar um dado dentro da linguagem, geralmente a variável representa o endereço na memória do*

valor atribuído.

Dica:

Uma variável pode ser criada com qualquer nome que não seja uma palavra reservada, mas temos que seguir alguns critérios:

***1-** nunca inicie o nome de uma variável por número ou dígito, gerará erro. Pode iniciar por Underline (sublinhado) ou letra.*

***2-** Escolha nomes que têm algo a ver com o dado armazenado, ex.: gasto = 10.00 ; lucro = 100.00 ao invés de n1 = 10.00; n2 = 100.00, visto que no primeiro caso ficou muito mais fácil entender sobre o que se trata o dado.*

***3 -** Se for um nome composto use iniciais maiúsculas, ex.: NomeDoCliente = "Marcos"; ou separados por sublinhado, ex.: nome_do_cliente = "Marcos", fica muito mais legível do que nomedocliente ;*

***4** – Cuidado para não sobre-escrever um valor de uma variável já existente, ex.: imprima = "nome do comando"; se depois disso você tentar usar a função imprima( ), verá que ela não funciona mais, pois você reatribuiu o valor da variável imprima. Sim é claro, funções são tipos de variáveis em Prisma.*

Variáveis em Prisma não possuem tipos fixos, são tipadas dinamicamente, ou seja, não é necessário declarar qual tipo uma variável irá armazenar (como é feito em C por exemplo: **int x = 12;**), basta inicializá-la, ex:

Prisma

```
1  x = 12; x = 3.333333 ; x = "uma frase qualquer" ; x = {};
2
3  //veja que a variável x trocou de dados várias vezes.
```

Note que ao atribuirmos um novo valor a uma variável já existente, o valor antigo se perde e o novo fica em seu lugar!

**Em Prisma variáveis não têm um tipo fixo, mas o dado sim carrega seu próprio tipo, Prisma sabe que um número é um número e não uma string ou vice-versa, por exemplo.*

Se você é iniciante em programação, deve ter um pouco de dificuldade em assimilar o conceito de variável de início, não se preocupe, você vai dominar o conceito conforme o tempo de estudo, o que em Prisma não é tão demorado, visto a sua simplicidade.

Apenas saiba que usamos o sinal de '=' para atribuir um valor que está à direita do sinal a um nome à esquerda do sinal, este nome passa a carregar este valor atribuído.

Em Prisma podemos atribuir diversos valores para várias variáveis ao mesmo tempo, usando a vírgula, veja:

Prisma

```
a , b , c = 1 , 10 , 100;
//a recebe 1, b recebe 10 e c recebe 100;
```

caso sobre uma variável, o valor daquela que sobrou será nulo, veja:

Prisma

```
a , b , c = 1 , 2;
//a = 1 ; b = 2 e c = nulo pois não tem um terceiro valor para c;
```

caso o número de variáveis seja menor que o número de valores atribuídos, os valores extras são perdidos, veja:

Prisma

```
a , b = 1 , 2 , 3;
// a = 1 ; b = 2 O valor 3 se perde pois não há
// uma terceira variável para guardar seu valor.
```

Diferentemente de outras linguagens que têm muitos tipos de dados, em Prisma o número deles é bem reduzido, o que favorece a rapidez do aprendizado. Bom, vejamos os dados em

Prisma:

número, string , tabela, funcao , userdata, boolean, nulo;

obs.: para sabermos o tipo de um dado existe a função tipo ( ); ex.: imprima (tipo(5)) ; //saída é 'numero'

Agora analisaremos cada um deles com exemplos, se possível tente assimilar ao máximo o conceito e utilidade deles:

numero

É qualquer número, não diferencia entre decimal e inteiros.

ex:

Prisma

```
 x = 12; y = 3.14;

imprima( tipo(x) , tipo(y) ); //saída será :  numero       numero

//a funcao tipo() retorna o tipo de variável.
```

*Obs. não use virgula para separar as casas decimais, em programação é usado **o ponto** para isso, ok!*

string

Qualquer sequência de caracteres entre aspas simples ou duplas, ou entre duplo-colchetes(multilinha);

ex:

Prisma

```
x = "\n\n Isto e uma string\n" ;
```

```
//aceita caracteres de escape \n = nova linha;

y = '\n\nIsto e uma string entre aspas simples'  ;
//sempre feche a string como o mesmo sinal que iniciou

z = [[esta

é uma

string multilinha

não aceita caracteres de escape ela e visualizada

tal como se encontra aqui!

]]

imprima ( tipo(x), tipo(y) , tipo (z) )
// -->saida = string        string        string
```

tabela

Poderoso recurso para manipulação de dados em massa, nada mais é do que uma variável que armazena mais de um valor ao mesmo tempo e esses valores ficam guardados em subdivisões dentro da tabela através de índices ou campos. Não é necessário declarar o tamanho da tabela como na maioria das linguagens.

ex.:

Prisma

```
//inicializando uma tabela vazia use chaves vazias:
tab = {};

//inicializando uma tabela com alguns índices:
tab2 = { "um elemento string" , 12 , 14 , "Quinta" , "nome" };

// por padrão tabelas em prisma iniciam pelo índice 1
// e não zero como na maioria das linguagens

imprima( tab2 [1] , tab2 [2] , tab2 [3] , tab2 [4] , tab2 [5]  );

//ou use o comando desempacote:

imprima( desempacote( tab2) );

// atribuindo novos elementos a matriz

tab[1] = "primeiro elemento";

tab[2] = "segundo elemento";

//fazendo um array associativo, isto é,
//índices serão 'strings' e não números:

tab3 = { ["dia"] = 15 , ["mes"] = 12 , ["ano"] = 2014 };

imprima( tab3[ 'dia' ] .. '-' .. tab3['mes'] .. '-' tab3['ano'] )
```

```
28  //--saída = 15-12-2014;
29
30  // logicamente existe um jeito muito
31  // mais prático de fazer isso acima, veja:
32
33  tab3 = {};
34
35  tab3.dia = 15; tab3.mes = 12 ; tab3.ano = 2014;
36
37  imprima( tab3.dia .. '-' .. tab3.mes .. '-' .. tab3.ano )
38  // -->saída = 15-12-2014
39
40  //poderia ser também = tab3 = { dia = 15 , mes = 12 , ano = 2014 };
```

é possível fazer uma matriz de componentes facilmente em prisma:

Prisma

```
1  para i = 1 , 4 inicio
2
3      rotulo = "botao" .. i
4
5      botao[ i ] = ig.botao( rotulo );
6
7  fim   //fim para
```

cria quatro botões; (botao[1], botao[2] ... etc.)

Tabela é um recurso tão poderoso que podemos armazenar qualquer valor válido, inclusive funções:

Prisma

```
1  console = {};
2  console.version = "Console-1.0";
3
4  funcao console.puts ( ... )
5      //lembre-se que usando os três pontos
6      //passamos número variável de argumentos.
7
8     imprima( ... );
9
10     // recebe e imprime todos os argumentos passados
11 fim  //fim funcao
```

Um array associativo de uma função funciona como um método de console. Ex.:

Prisma

```
console.puts( 'Ola Mundo ' .. console.version ); // --> saída = Ola mundo Console-1.0

console.puts( tipo(console) ); // --> saída = tabela;

console.puts( tipo( console.puts ) ) ; // -- > saída = funcao
```

Obs.: lembre-se que o ponto separa o nome da tabela de seu campo.

Poderíamos fazer uma função que passasse a própria tabela como parâmetro padrão “este”:

Prisma

```
//ex:
console = {};
console.versao = "Console-1.0";

funcao console:mostreVersao ( )

      imprima( este.versao );


fim //fim funcao

console:mostreVersao() ;
// saída = Console-1.0
```

Ao usar dois pontos **‘:’** passamos a própria tabela na variável reservada **‘este’**. Portanto, **este.versao** é o mesmo que **console.versao**.

*Obs.: **‘este’** é uma variável reservada, não podendo ser usada de outra forma.*

funcao

São trechos automatizados de rotinas de execução, funções podem ou não receber parâmetros e retornar valores, são muito úteis para organizar o programa em sub blocos executáveis.

ex.:

Prisma

```
//declarando uma função soma use a palavra reservada funcao:

funcao soma ( a , b)

  retorne a + b ; //retorna o valor de a somado com b

fim

funcao quadrado ( a )

   retorne a * a; // retorna a multiplicado por a

fim

funcao raizquadrada ( a )

    retorne a^ ( 1/2 );
//retorna a elevado a 0.5 , o que
//equivale a raiz quadrada de "a";

fim

//chamando as funcoes criadas

imprima( "soma de 2 e 5 = " , soma( 2 , 5 )  );
//-- > saída = soma de 2 e 5 = 7

imprima ( "5 ao quadrado = " , quadrado( 5 );
// --> saída = 5 ao quadrado = 25

imprima( "raiz quadrada de 25 = " , raizquadrada( 25 );
//--> saída = raiz quadrada de 25 = 5
```

Funções em Prisma podem retornar mais de um valor:

Prisma

```
funcao duploquadrado ( a , b )

     retorne a*a , b*b;
//cada retorno deve estar separado por vírgula.

fim

 //recebendo os retorno separados por vírgulas

n1 , n2 = duploquadrado ( 2 , 4 );
// retorna o numero 2 e o numero 4 elevados ao quadrado;

//repare que para cada retorno deve ter
//uma variável à esquerda separada por vírgula.
```

Se uma função retornar mais de um valor e você se esquecer de colocar mais de uma variável à esquerda do **'='** , só o primeiro valor será atribuído os demais se perderão.

Userdata

É um tipo de dado que armazena um ponteiro para um programa C; Só é utilizado para acessar

dados e funções em Prisma do executável C. ex: ao usar a função jan = ig.janela( ) o retorno é um endereço de um ponteiro GtkWindow em C.

Prisma

```
inclua 'igbr'

jan = ig.janela();
imprima(  tipo ( jan )  );
// -- > saída = userdata
```

Boolean

Armazena apenas dois valores: **falso e verdadeiro** (em inglês: false , true);

ex:

Prisma

```
imprima ( tipo(falso) );
// -- > saída = boolean
a = falso;
b = verdadeiro;
se a == falso entao imprima'falso' fim
se b == verdadeiro entao imprima 'verdadeiro' fim
```

Nulo

É qualquer valor inexistente, vazio. ex:

Prisma

```
imprima (   tipo( x )    );
// se x não foi declarada ainda então não existe,
// logo a saída será nulo.
```

Obs.: nulo é considerado falso em testes de condição em Prisma, ex.:

Prisma

```
x = nulo;

se x entao
   imprima 'existe'
senao
   imprima 'nao existe'
fim //fim  se
```

Para matar uma variável basta atribuir a ela o valor **nulo:**

```Prisma
x = 20; //x existe, possui um valor

x = nulo; // x não existe mais, seu valor é nulo;
```

Cap. 4 – Escopo das variáveis

O escopo pode ser: global ou local.

Em Prisma, qualquer variável declarada é **global**, isto é, seu valor pode ser usado em qualquer parte do programa ou dos módulos do programa principal.

Já a variável de escopo local ao ser declarada deve ser antecedida pela palavra reservada **local** ex.: local x = 1; deste modo este tipo de variável não é visível fora do bloco ao qual pertence. O bloco pode ser uma função, um laço condicional ou de repetição, um bloco simples ou um programa Prisma.

A vantagem em se declarar uma variável local é a segurança para que ela não seja modificada por uma função externa acidentalmente, ou em caso de dúvidas ao atribuir variáveis com mesmo nome.

Para entender melhor vejamos os exemplos:

```Prisma
//exemplo de variáveis locais e globais:

funcao mostre_var ( )
       imprima ( var1 , var2 );
fim //fim funcao mostre_var

funcao inicie_var ( )
     var1 = "Ola!" ;
     local var2 = "Mundo!";
fim //fim funcao inicie_var


funcao inicie_programa ( )
      inicie_var( ); //chamando inicie_var
      mostre_var ( );//chamando mostre_var
fim //fim funcao inicie_programa

inicie_programa ( ) ;
```

O resultado será: **Ola! nulo**

A segunda variável é local por isso não é visível fora da função **inicie_var ( )** e seu valor fora dela é inexistente: nulo.


Para controlar o escopo das variáveis sem precisar, para isso, fazer uma função toda vez, podemos criar um bloco de comandos com as palavras reservadas **inicio fim**, veja:

Prisma

```
inicio  //bloco
   local var1 = 1;
   local var2 = 2;

  imprima ( var1 , var2 );
  //saída será 1   2
fim //fim bloco

imprima ( var1 , var2 );
//saída será nulo   nulo pois a variável
// não é visível fora do bloco
```

Cap. 5 – Controle de fluxo e laços de repetição (se, enquanto, repita, para)

Ao escrever um programa podemos querer que ele tome decisões, por exemplo, fechar o programa se o usuário digitar **‘S’** ou executar repetidas vezes uma função até chegar ao fim de um arquivo.

Enfim, para isso existem os laços de controle e repetição. Esses comandos são tão necessários que qualquer linguagem os possui, e com Prisma não é diferente.

Para exemplificar tomemos o primeiro exemplo acima, fechar o programa ao digitar **‘S’.**

Usando o comando ‘se’ :

A estrutura básica é: **se** condicao **entao** ...código... **fim**

Prisma

```
imprima ("Digite S para Sair do programa, ou aperte Enter para continuar");

    opcao = leia ( );
//lê o que for digitado pelo usuário e armazena o valor em opcao

  se opcao == 'S' ou opcao == 's' entao
```

```
            sis.saia( )
   fim //fim se opcao

imprima ("Você escolheu continuar...");
leia ( );
```

Não esqueça de fechar o comando **se** com um **fim** respectivo Não se esqueça de usar a palavra **entao** depois da condição.

Se o usuário digitar o S ou s o programa é fechado imediatamente.

Note que mesmo o usuário não digitando nada o programa não tem muito o que fazer depois, e logo fecha.

Colocando uma exceção para o comando se, usando o ‘senao':

Estrutura básica é: **se** condicao **entao** ...código... **senao** ...código da excecao... **fim**

Prisma

```
imprima ("digite um número:");

numero = leia ( );
// lê o que é digitado pelo usuário e guarda o valor em número

numero = convnumero (numero);
//converte string para número

se numero < 0 entao

       imprima ("Numero menor que zero\n");

senao

      imprima( "Numero maior que zero\n");

fim
```

Neste caso se a verificação **numero < 0** retornar verdadeiro a primeira função imprima será executada, caso ela retorne falso a função depois do senao será executada, é fácil entender, interprete desta maneira: *se numero digitado pelo usuário for menor que zero realiza a primeira opção senão realize a exceção que é o segundo imprima.*

usando o ‘se’ o ‘senaose’ e ‘senao':

Estrutura básica:

se condicao **entao** ...código... **senaose** condicao **entao** ...codigo2 ... **senao** ...exceção... **fim**

Pode-se omitir o **senao** e usar somente o fim se não quiser nenhuma exceção, ficando assim:

se condicao **entao** ...código... **senaose** condicao **entao** ...codigo2 ... **fim**

O **senaose** é justamente para testar uma série de condições subsequentes, caso uma delas seja verdadeira, o código do seu corpo é executado e as demais ignoradas. Veja um exemplo:

Prisma

```
imprima( "Digite A B  ou C");

letra = leia ( );
// lê os dados digitados pelo usuário e armazena em letra

se letra == 'a' entao

     imprima ("digitou A");

senaose letra == 'b' entao

     imprima ("digitou B");

senaose letra == 'c' entao

     imprima("digitou C");

senao

     imprima("nao digitou A nem B nem C");

fim

leia ();
```

Comando enquanto

Repete um bloco enquanto o retorno da condição for verdadeiro, sua estrutura básica é:

enquanto condição **inicio**

....código .....

fim

Exemplo:

Prisma

```
 teste = ""

cont =  1;

enquanto teste <> 's' inicio
```

```
//enquanto teste for diferente de 's'

 imprima ("Digite s para sair, ou tecle enter para continuar " , cont  );

 cont = cont + 1;
//acrescenta 1 no valor atual de cont

 teste = leia ( );
//lê os dados digitados pelo usuário e armazena em teste

fim
```

usando o comando quebre:

Se preferir você pode fazer um laço **enquanto** infinito e controlar a repetição por dentro do laço usando o **quebre**:

Prisma

```
 teste = ""

 cont =  1;

 enquanto verdadeiro inicio
//o laço é infinito já que verdadeiro é sempre verdadeiro

   imprima ("Digite s para sair, ou tecle enter para continuar " , cont  );
   cont = cont + 1; //acrescenta 1 no valor atual de cont
   teste = leia ( );
   //lê os dados digitados pelo usuário e armazena em teste

   se teste == 's' entao quebre fim;
fim
```

Ao usar **quebre** o laço é interrompido imediatamente.

Comando repita

Seu funcionamento é semelhante ao enquanto, repete um bloco até que a condição seja verdadeira.

Sua estrutura básica é:

repita .... comando ... **ate** condicao

Prisma

```
teste = "";

```

```
x = 1;

repita

    imprima( "Digite s para sair ou aperte enter para continuar" , x );

    teste = leia ( );

    x = x + 1;

ate teste == 's';
// quando teste for igual a 's' o laço é interrompido
```

Outro exemplo imprimindo de 1 a 1000:

Prisma

```
n = 1;

repita

            imprima ( n );

           n = n + 1;

ate n == 1000; //imprime ate 1000 e para a repetição
```

Comando 'para' (numérico)

Obs.: este modelo é chamado de numérico porque consiste na repetição de um número inicial até um número final.

Este comando também serve para repetição. Ao contrário dos demais, nele é preestabelecido o número de repetições no próprio cabeçalho do comando. Para entender melhor veja:

Estrutura básica – **para** var_inicial, var_final **inicio** ....comandos .... **fim**

ex.:

exemplo do comando para — Prisma

```
para i = 1 , 1000 inicio
   imprima ( i );
fim
```

Veja que o comando em si é bem simples, leia o exemplo acima da seguinte forma:

para o valor inicial de i que é 1 até o valor 1000 repita;

Note que i é uma variável local ao bloco do comando para, poderia ser um nome de sua escolha (desde que válido, lembre-se das regras ao declarar uma variável em Prisma) – ex.: **para** cont = 1 , 1000 **inicio** ...código... **fim**;

Este primeiro exemplo foi o modo simples, existem outros modos do comando **para**, não se preocupe, veremos todos eles com calma, você dominará cada um aos poucos.

Adicionando o terceiro parâmetro no comando para, o incremento:

As vezes você pode querer que a repetição seja de 2 em 2 ao invés de 1 em 1, neste caso devemos usar um terceiro parâmetro no comando para, o incremento, que foi omitido no primeiro exemplo. Veja:

Estrutura básica – **para** var_inicial , var_final , incremento **inicio** ...comandos... **fim**

Prisma

```
1  para i = 1 , 100 , 2 inicio
2
3          imprima ( i );
4
5  fim
```

Obs.: o número 2 é o tipo de incremento que significa o acréscimo de 2 ao valor de i a cada repetição, teremos então na execução: 1 3 5 ... e assim por diante sempre somando 2 . Poderia ser outro número de sua preferência, por exemplo se quisesse contar de 10 em 10 deveria usar 10 no lugar de 2. N*ote que ao omitir o terceiro parâmetro o valor padrão é 1.*


Nunca use o sinal de mais para o incremento, ou gerará erro! Veja que os exemplo ***para*** *aqui são simples, apenas imprimem o valor do incremento a cada repetição, mas você pode adaptar e colocar qualquer comando válido no bloco de repetição, visto que o comando para é muito poderoso em determinadas tarefas, veremos mais adiante.*

Determinando um decremento

Em algumas situações você vai preferir que o comando para inicie em um valor alto, 100 por exemplo, e diminua até chegar ao valor mínino, 1 ou 0 por exemplo.

Veja:

Estrutura básica – **para** valor_maximo, valor_minimo , decremento **inicio** ... código ... **fim**

Prisma

```
para i = 100 , 1 , -1 inicio

      imprima ( i );

fim
```

Note que usamos o sinal de menos para o decremento. O valor do decremento foi -1, isto é, diminui 1 de i a cada repetição, mas poderia ser outro valor como -2, -3 ou -10, para diminuir 2, 3 ou 10 de i a cada repetição, isso depende do propósito do programa e do programador.

Comando 'para' no modo especial (genérico)

Trata-se de um modo bem específico de uso do comando para em que usamos iteradores especiais (incrementos que percorrem algum tipo de dado em sequência).

Primeiro modo, lendo os índices de uma matriz:

Estrutura básica – **para** var_indice , var_valor **em ipares(matriz) inicio** ... código ... **fim**

para genérico Prisma

```
//declarando uma matriz, lembre-se que em prisma
//o primeiro elemento é o índice 1 e não zero como nas outras linguagens

matriz = { "amarelo", "azul", "branco",
            "cinza" , "marrom" , "verde",
           "vermelho" };
//a matriz pode ser feita em várias linhas!

para indice, valor em ipares( matriz ) inicio

      imprima (  indice ,  '=' , valor );
     //imprime o índice e o valor correspondente

fim
```

Note que **indice** e **valor** são variáveis locais que receberão valores da tabela a cada repetição. A função ipares() percorre os índices e valores de uma tabela em Prisma. Note também que não usamos a vírgula no cabeçalho do comando para ao invés disso usamos a palavra reservada **em**.

(ipares vem de indice-pares.)

Este exemplo do para genérico funciona muito bem em matrizes que possuem apenas índices numéricos, mas não funcionam quando o índice é um campo, por exemplo, matriz ['nome'] = ‘Amanda'; no exemplo acima não imprimiria Amanda, somente as chaves numéricas.

Não se preocupe para isso temos algo muito parecido com o **ipares,** o **pares**, simples, não é?

veja:

estrutura básica - **para** chave , valor **em pares (**tabela**) inicio** ...comandos ... **fim**

Prisma

```
//exemplo do uso da função pares() no comando para genérico:

dados = { 12, 123, 1234, nome = "Amanda",
          idade = 23 , telefone = '65 xxxx-xxxx',
          12345 };
dados.novocampo = "novocampo";

para chave, valor em pares ( dados ) inicio
    imprima ( chave , valor );
//imprimirá cada campo e valor da tabela criada acima
fim //fim para
```

Note que a tabela criada é mista, ou seja, possui valores indexados de forma numérica e pares de campo e valor, ainda incluímos um novo campo depois da criação da tabela dados.

Veja que a atribuição de chave/valor na função pares( ) vai na ordem de índices numéricos (na sequência 1-2-3 ...) para depois os campos. Estes são impressos na sequência de criação.

Cap.6 Funções

Em Prisma podemos dizer que função é uma variável especial que pode executar um bloco de código ao ser chamada, além disso, ela pode receber, manipular e devolver valores. Em outras linguagens as vezes são chamadas de procedures ou subrotinas.

Enfim, na prática temos um programa bem organizado se utilizar funções da forma correta do que um arquivo com um enorme emaranhado de comandos.

Para fazer uma função usamos a palavra reservada **funcao,** veja:

Estrutura básica – **funcao** nome (parâmetros **)** ...códigos... **fim**

exemplo de função Prisma

```
//exemplo de função em Prisma

funcao mostre_ola ( )
      imprima( "Ola Mundo\n");
fim //fim funcao

//chamando a função:
mostre_ola ( );

leia( );
//no windows é bom usar o comando leia para o programa
// não fechar de repente. No prismacod não é necessário!
```

Note que a função é bem simples, não recebe parâmetro nem retorna dados, apenas imprima a frase Ola Mundo.

Como funções em Prisma são variáveis, existe um outra forma de fazer a função acima, veja:

Prisma

```
mostre_ola = funcao ( )
    imprima ("Ola Mundo\n");
fim //fim funcao

mostre_ola( );
```

Definindo parâmetros

Veja, o mesmo exemplo acima, mas modificado para receber um valor e imprimir esse valor:

Prisma

```
//exemplo funcao recebendo valores
funcao escreva ( msg )
    imprima ( msg );
fim //fim funcao

escreva ( "Ola Mundo em Prisma!\n");
```

No exemplo acima nós definimos um parâmetro para a função escreva, criando uma variável entre os parênteses, sendo assim essa variável é local, isto é, só pode ser acessada no bloco da função.

Dentro do corpo da função usamos a função imprima para imprimir o argumento passado para a função escreva.

Obs. Nunca coloque ponto-e-vírgula após fechar o parênteses da função. (funcao (msg); = errado)

Devolvendo valores com o comando retorne

Além de receber valores uma função pode retornar valores, veja:

Prisma

```
funcao soma (  n1 , n2 )
  local resultado = n1 + n2 ;
  retorne resultado;
 //retornando o valor da variável resultado
fim //fim funcao

a = soma ( 9 , 3 );
// a soma de 9 e 3 é retornada para a variável a;

imprima ( a );
//imprime o valor retornado;
```

Obs. que uma função pode receber muito mais do que dois parâmetros

Dica – sempre nomeie as funções de acordo com a tarefa que elas realizam, isso ajuda a ler e entender um programa futuramente.

Ex.: raiz_quadrada ( ) ; le_texto ( ) ; dobro ( ) ; etc.

Uma vantagem em Prisma é que uma função pode retornar mais de um resultado ao mesmo tempo, isso é feito por meio do uso de vírgulas. Veja:

Prisma

```
funcao maior_para_menor ( n1 , n2 )

             se n1 > n2 entao
                  primeiro = n1 ;
                  segundo = n2;
            senao
                  primeiro = n2;
                  segundo = n1;
            fim //fim se

    retorne primeiro , segundo;
    //retorna dois valores;
```

```
13  fim //fim funcao
14
15
16  valor1 , valor2 = maior_para_menor ( 100 , 80 );
17  //recebendo os dois valores em ordem do maior para o menor
18
19  imprima( valor1, valor2 );
```

Veja que para cada retorno deve haver uma variável à esquerda do operador de atribuição '=', e cada variável deve estar separada por vírgula.

Caso um parâmetro fosse omitido ao chamar a função, o segundo parâmetro ficaria com valor nulo, veja:

valor1 , valor2 = maior_para_menor (9) ; //n2 na função não foi passado então seu valor é nulo. E isso ocasionaria um erro interno na função, pois ao tentar concatenar, executar uma operação, ou comparar um valor com **nulo** on programa falha e uma mensagem de erro é exibida.

E se o número de variável não for igual ao número de retorno?

valor = maior_para_menor (1, 2);

Neste caso somente o primeiro retorno foi passado para a variável valor, o segundo retorno foi descartado!

Se o número de argumentos passados for maior do que os parâmetros exigidos pela função, os argumentos extras são descartados.

Recursividade em funções

Este efeito é obtido quando uma função chama a si mesma. Não é muito recomendado usar este recurso pelo perigo de erro difícil de ser detectado, e pelo simples fato de o mesmo poder ser feito de outra forma mais segura.

Veja, a função chama a si mesma e soma à variável um valor que é o retorno da própria função com o parâmetro igual a ela mesma menos 1 várias vezes até chegar a zero:

Prisma

```
1  //exemplo de recursividade em Prisma,
2  // quando uma função chama a si mesma de dentro si mesma
3
4  funcao soma ( valor )
5         se valor == 0  entao
```

```
6                   imprima ( valor ) ;
7                   retorne valor;
8           senao
9                   valor = valor + soma( valor - 1 ) ;
10                  imprima ( valor );
11
12                  retorne valor;
13           fim //fim se
14
15  fim //fim funcao
16
17  soma ( 3 ) ;
18  // o resultado é a soma de 3 + 2 + 1 + 0 que é 6
```

Note que a sequência de impressão na tela é: 0 1 3 e 6; em cada linha. Isso ocorre porque a função busca um resultado em seu retorno até chegar a zero, então de trás para frente ela vai somando os resultados.

Para entender, imagine ao passar o número 3 todo o processo que a função faz:

1 – função soma é chamada com o parâmetro numero 3:

compara 3 com 0 não é igual, entao: valor é igual a valor + soma (3 – 1);

2 – função soma chamada novamente mas com o parâmetro número 2 (que é o resultado de 3 – 1);

compara 2 com 0 não é igual então: valor é igual valor + a soma (2 – 1);

3 – soma é chamada novamente de dentro dela mesma com o valor 1 (resultado de 2 -1)

compara 1 com 0 não é igual então: valor é igual a valor + soma(1 – 1);

4 – soma é chamada novamente com valor 0 como parâmetro (resultado de 1 – 1)

compara 0 com 0, é igual então retorna valor que é 0;

ao chegar ao resultado zero, todas as chamadas da função soma vão retrocedendo o valor somando o último

com o anterior, 0 + 1 + 2 + 3 chegando no resultado final 6, por isso na tela apareceu os números em ordem inversa:

```
0

1

2
```

```
3

6
```

Como dito antes não é recomendado o uso da recursividade pelo fato de que as possibilidades de gerar um erro oculto são maiores, e porque o mesmo resultado pode ser obtido de uma outra forma mais segura. É totalmente dispensável o uso da recursividade, até mesmo pela sua complexidade.

Veja a função anterior de outra forma mais segura:

Prisma

```
funcao soma ( valor )

  temp = 0;
   para i = 0 , valor  inicio
   //de i que é igual a 0 até valor
           temp = temp + i;
           imprima (temp);
   fim //fim para

  retorne temp;
fim //fim funcao

soma ( 3 );
```

Veja que o comando **para** torna a recursividade obsoleta, sem necessidade.

Uma coisa a se falar de funções é que quando o único argumento for uma string ou tabela, podemos dispensar o uso de parênteses, veja:

Prisma

```
 imprima 'ola mundo';

 funcao imp_tabela( tab )

     para i = 1 , #tab inicio
            imprima( tab[i] );
     fim //fim para

 fim //fim funcao



imp_tabela {"Maria", "Joao",
            "Marcos", "Paulo",
            "Gabriela"};

//note que funciona se a string ou tabela
//for passada diretamente como único argumento.
//com uma variável não daria certo.
```

Além de tudo que vimos acima sobre funções, um aspecto interessante é que elas podem ser passadas como argumentos para outras funções ou retornar funções, veja:

Prisma

```
funcao recebe_funcao (   func , param )
      retorne      func( param );
fim


funcao quadrado ( num )
       retorne num*num;
   //retorna o número vezes ele mesmo,
  // ou seja, elevado ao quadrado;
fim

funcao cubo ( num )
      retorne num*num*num;
fim

a = recebe_funcao ( quadrado , 2 );

imprima ( a ) ; //saida será 4;

b = recebe_funcao ( cubo , 2 );

imprima( b ) ; //saída será 8 ( 2 * 2 * 2 );
```

Número indefinido de parâmetros em funções

Lembre-se da função imprima() que pode receber um, dois, três, ou indefinidamente mais (claro que não infinito). Para fazermos isso em Prisma usamos o sinal de reticências '...' dentro dos parênteses no cabeçalho da função. Veja:

Prisma

```
funcao some_todos (  ...  )
     local tabela_num = { ... };
     local tmp = 0;
     para i = 1 , #tabela_num inicio
          tmp = tmp + tabela_num[i];
     fim //fim para
 retorne tmp;
fim //fim funcao

imprima(   some_todos( 2 , 3 , 5 )  );
imprima( some_todos( 2 , 3 , 6, 54, 32 , 556 , 7)  );

//não importa se é um ou cem argumentos passados,
//a funcao retornará a soma de todos.
```

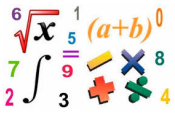
Veja que os **'...'** representam o número variado de argumentos que podemos colocar entre chaves e com isso criar uma tabela com todos os argumentos passados, não importa se um ou cem ou mais.

Veja a saída no modo prompt do interpretador:

Prisma 1.0.9 Interpretador Prisma por <adalbermirom@gmail.com>
Seja Bem-vindo! Tente digitar algum comando como: imprima('Ola, Mundo!')

```
>>> funcao some_todos ( ... )
... local tabela_num = { ... };
... local tmp = 0;
... para i = 1 , #tabela_num inicio
... tmp = tmp + tabela_num[i];
... fim //fim para
... retorne tmp;
... fim //fim funcao
>>>
>>> imprima( some_todos( 2 , 3 , 5 ) );
10
>>> imprima( some_todos( 2 , 3 , 6, 54, 32 , 556 , 7) );
660
>>>
>>> //não importa se é um ou cem argumentos passados,
>>> //a funcao retornará a soma de todos.
```

Cap. 7 Operador de tamanho e operadores aritméticos



São eles:

———————————————————————————————————-

(tamanho)

* (vezes), **/** (divisão) **+** (soma) , **-** (subtração) , **%** (resto) , **^** (elevado a);

———————————————————————————————————-

Veja como usar cada um deles:

#

(determina o tamanho de strings e tabelas) :

Prisma

```
frase = "o tamanho desta frase e = "

imprima(  frase , #frase );
// saída - o tamanho desta frase e =    26

tab = { "um elemento" , "outro elemento" , 1 , 2 , 4 , 5 , 6 , 7 , 9 };

imprima( "A tabela tab tem" , #tab , "elementos\n" );
//saída - A tabela tab tem      9     elementos
```

Operadores matemáticos (servem para fazer operações aritméticas):

Prisma

```
imprima ( "\na soma de 10 e 2 = " .. 10 + 2 );
// o \n Não aparece na saída, caractere invisível de nova linha

imprima ( "\na subtracao de 10 por 2 = " .. 10 - 2 );
//lembre-se que os '..' serve para unir strings
//ao unir um número a uma string,
//o número é convertido automaticamente para string!

imprima( "\n10 vezes 2 = " .. 10 * 2 );

imprima( "\n10 dividido por 2 = " .. 10 / 2 );

imprima ( "\no resto de 10 dividido por 3 = " .. 10 % 3 );

imprima( string.formate (
 "numero com 4 casas decimais = %.4f , formatado para 2 casas = %.2f " ,
 2.1234 ,
 2.1234
 )
);

imprima ( "\n10 elevado a 2 = " .. 10^2 );

imprima ( "\nRaiz quadrada de 100 = " .. 100^ (1/2)   );

imprima ( "\nRaiz cubica de 27 = " .. 27^(1/3)    );

imprima ( "\nRaiz Quadrupla de 16 = "  .. 16^(1/4)  );

imprima( "\n\n15 por cento de 39 = " .. 15 * (39/100)  );
```

Mais adiante veremos a biblioteca ‘mat’ com funções especiais para matemática e trigonometria;

Unidade II – Biblioteca padrão

Introdução

Prisma, assim como outras linguagens, possui funções predefinidas, e que ficam organizadas em conjuntos chamados de biblioteca. Bibliotecas, também chamadas libs, podem ser nativas ou não nativas.

As nativas são as embutidas no interpretador Prisma, e por isso não precisam ser importadas

com o comando ‘inclua’.

Todas as outras bibliotecas externas que precisam ser incluídas não são nativas, não são predefinidas, portanto é necessário carregá-las na execução do programa prisma.

Bibliotecas são importantes pois potencializam o poder da linguagem, é com suas funções que é possível escrever programas mais complexos.

Aqui serão tratadas o conjunto de biblioteca padrão, isto é, somente as nativas, são elas:

base, string, mat, sis, es, tabela, bit32 e debug;

A biblioteca win só existe na versão Prisma Windows.

Ela é tratada separadamente, veja na página de documentação: Documentação

Cap. 1 – Funções da biblioteca base

São as funções simples tais como a imprima( ) ; tipo ( ) entre outras. Elas não seguem o modelo das demais que usam o nome da biblioteca e o ponto separando o método.

Veja as funções e seus exemplos de uso:

imprima (...)

- exibe na tela preta do cmd texto ou números. Podemos passar vários argumentos de uma só vez separados por vírgulas;

ex.:

Prisma

```
imprima("Ola Mundo!");
```

leia ( )

Uma versão simples do **es.leia( )** (que trataremos adiante), a função leia aguarda o usuário digitar algo e retorna o que foi digitado, podendo ser armazenado em uma variável do seguinte modo:

Prisma

```
poe "Digite algo: ";

algo = leia ( );

poe ( algo );
//saída será aquilo que o usuário digitar;
```

tipo (valor)

Esta função retorna uma string descrevendo o tipo de valor. Muito útil para sabermos qual tipo de variável é um dado. **V*alor*** pode ser uma variável ou diretamente um valor. Exempo de uso:

Prisma

```
x = tipo ( y );

    se x == "numero" entao
            imprima("variavel do tipo numero") ;
    senaose x == "string" entao
            imprima("variavel do tipo string");
    senao
            imprima ( "a variavel nao e do tipo numero nem string");
            imprima ("o tipo e = " , x );
    fim
```

Aproveitando aqui veja algumas variáveis

predefinidas importantes:

* **_versao -** variável que guarda a versão de prisma, tente executar: **imprima(_versao);**

* **prismadir** – variável que guarda o diretório do executável prisma: **imprima (prismadir);**

* **args -** é uma ***tabela*** dos argumentos externos passados como parâmetros em linha de comando:

Prisma

```
imprima (  args [ 0 ] );
// imprime o nome do programa prisma sendo executado.

imprima ( args [-1] );
//imprime o nome do interpretador,
//que pode ser o prisma.exe ou o igprisma.exe;
// no linux prisma
```

Voltando para as funções da biblioteca base

poe (string)

Vêm do verbo pôr (poe, do inglês = put), coloca na tela preta o resultado, é o mesmo que a função imprima, mas com limitações, não aceita por exemplo mais de um argumento, é bom para strings apenas. Ex.:

Prisma

```
poe("Ola mundo em Prisma!");
```

executestring(string)

Executa um trecho de comando em uma string em tempo de execução, retorna zero caso tenha sucesso ou 1 caso falhe.

Muito útil quando você quer que seu programa leia e interprete comandos. Exemplo:

Prisma

```
 x = executestring ( "imprima ( 'Ola Mundo' ) ; " );
//imprime na tela Ola Mundo

   se x == 0 entao
        imprima 'string executada com sucesso'
```

```
6     senaose x == 1 entao
7          imprima 'a string nao foi executada por algum erro';
8     fim
```

Dica: cuidado com as aspas, se a string a ser executada possui apas internamente você tem algumas opções:

1 – Usar aspas simples internamente, e duplas externamente ou o inverso, como no exemplo acima;

Prisma

```
1  x = executestring ( "imprima ( 'Ola Mundo' ) ; " );
```

2 – Usar o mesmo tipo de aspas. Mas as internas devem ser antecedidas por barra oblíqua: **\" *ou* \' *ex.:***

Prisma

```
1  executestring ( 'imprima(\'ola mundo\') ' );
```

Ou

3 – Usar duplos colchetes:

Prisma

```
1   executestring( [[ imprima ("ola mundo" ) ]] );
```

Como o argumento é uma única string podemos omitir o parênteses da função como fazemos desde o início com a função ***inclua***. Ficando assim: **executestring "seu_comando_aqui" .**

executearquivo ("caminho_do_arquivo.prisma")

Semelhante ao comando executestring, mas ao invés de executar uma string, executa um programa prisma, que pode estar compilado ou não.

Muito útil quando seu programa tem vários subprogramas separados, sendo chamados pelo programa principal. Esta função, *executearquivo*, em caso de haver erro no programa executado, simplesmente trava a execução do programa principal, fechando-o e imprimindo mensagem de erro na tela cmd (no Línux terminal).

Se o programa prisma a ser executado estiver na mesma pasta que o programa principal, não é necessário usar o caminho completo, caso contrário sim.

ex.:

Prisma

```
executearquivo( 'ola.prisma');
//--> executa um programa prisma chamado ola.prisma;
```

Obs. Lembre-se que o nome do arquivo a ser executado deve ser escrito com a extensão.

Lembre-se de que no windows as barras devem ser duplas, ex:

Prisma

```
"C:\\Users\\Admin\\ola.prisma";
```

Ou use duplos colchetes:

Prisma

```
 [[C:\Users\Admin\ola.prisma]];
```

Já no Línux não é necessário, pois as barras de endereço são assim: **/**

Prisma

```

'/home/user/ola.prisma'
```

carregue (string)

Semelhante a função ***executestring*** com a diferença de que esta carrega a string e retorna uma função dela para a variáve. **Ex:**

Prisma

```
imprima_ola = carregue ( "imprima 'ola mundo' " );
//primeiro carrega a string em uma função

 imprima_ola() ;
//executa a string carregada nesta funcao;
```

Se houver erros, simplesmente nenhum valor é retornado e o valor da função será nulo;

carreguearquivo (“programa.prisma”)

Similar à função carregue, no entanto, esta carrega um arquivo fonte prisma ou programa compilado prisma em uma função que ao ser executada executa o programa carregado. Exemplo:

Prisma

```
execute_arquivo = carreguearquivo ( 'ola.prisma' );

execute_arquivo ( ) ; //executa o programa carregado.
```

Esta função, ao contrário da executearquivo, dá a possibilidade de fazer um tratamento de erro:

```Prisma
execute_arquivo = carreguearquivo ( 'ola.prisma' );

se execute_arquivo entao
        execute_arquivo ( ) ;
senao
       imprima ( "erro no arquivo");
fim //executa o programa carregado.
```

convnumero(string)

Converte uma string para numero. Exemplo:

```Prisma
  x = "12";
// x = x + x -- errado!

  x = convnumero( x ); //correto!

  x = x + x;
imprima(x);
//depois de convertermos para número
```

convstring (numero)

Faz o contrário da função anterior, converte um número para string.

```Prisma
x = "1 + 1 = " .. 2 ;
//ao concatenar a conversão é automática,
//mas é recomendável que se converta antes:

x = "1 + 1 = " .. convstring ( 2 );
imprima(x);
```

tente (comando , ‘mensagem erro’);

Esta função verifica um valor e retorna um mensagem de erro caso esse valor seja falso ou nulo, ou o próprio valor caso ele seja válido.

Sua sintaxe é : **x = tente (valor , String_erro);**

Esta função é ótima para testar abertura de arquivos. Veja seu uso:

```Prisma
x = tente ( falso , "erro - valor invalido" );

/**
retorna o valor se for válido ou a mensagem
de erro se valor for inválido, isto é, se for
falso ou nulo
**
```

Outro exemplo de uso com a função que abre arquivo em Prisma:

```Prisma
x = tente(
    es.abra ( "teste.txt" , "escrita" ),
   "erro - arquivo nao pode ser aberto"
); //tente

/**
se a função conseguir abrir o arquivo retorna o
arquivo para x senão fecha o programa e retorna
a mensagem de erro, que é a string definida no
segundo parâmetro.
**

se x entao
// se x tiver algum valor válido entao:
   x:escreva ("uma linha escrita no arquivo");
//escreve no arquivo aberto

   x:feche ( );
// depois é necessário fechar o arquivo aberto.

fim //fim se
```

cod (‘Car’)

Esta função converte um caractere para sua representação em bytes (número), há uma certa correspondência ao número de cada tecla do teclado.

Exemplo:

```Prisma
imprima ( cod ('A')  ) ; // saída --> 65
```

No exemplo, podemos observar que esta função converte a letra ‘A’ para o número 65.

Obs.: O nome da função ‘cod’ faz referência a código (‘codifique’);

car (cod)

Esta função converte de codigo para caractere, ou seja, o parâmetro é um número que será convertido para caractere.

Ex.:

Prisma

```
imprima ( car ( 65 )   ); // saída -- > A
```

compile(função)

Esta função converte uma função prisma em byte-code em tempo de execução retornando em forma de string codificada, é possível gravar em arquivo o resultado e chamá-lo posteriormente com os comandos inclua, carreguearquivo ou executearquivo.

ex.:

Prisma

```
//ex. compile
funcao soma ( a , b )
//criando uma função para ser compilada

  imprima ( a + b);
fim

 xsoma = compile( soma );
//compilando a funçao, o retorno é uma string compilada

   x = carregue( xsoma );
//para usar basta carregar a string compilad
//e depois chamar como uma funcao qualquer:

   x( 3 , 4 ) ; //saida -->      7
```

Não precisa necessariamente declarar a função antes, pode passar a função sem nomeá-la, isto é chamado de função anônima:

ex.2:

Prisma

```
//ex. compile()
x = compile ( funcao( ) imprima( 'Ola Mundo') fim   );
//a função é passada como argumento, mas sem nome;

g = carregue ( x ); //carregando a string compilada
```

```prisma

g ( );
//executando como uma função, saída -- > Ola Mundo
```

No próximo exemplo vou demonstrar como gravar em um arquivo e depois executar o arquivo:

ex.3:

```prisma
//ex. compile()
x = compile(
//o argumento pode ocupar normalmente mais que uma linha:

funcao (  )

         para i = 1 , 100       inicio  imprima ( i )       fim

fim


);

arquivo = es.abra ( 'teste.prisma' , 'escritabin' );
//abre um arquivo em modo escrita binária

     arquivo:escreva ( x ) ;
// escrevendo no arquivo a string compilada 'x'

arquivo:feche( ); //pronto, arquivo fechado

//executando o arquivo, pode ser de várias
//maneiras, mas aqui vou usar o 'executearquivo'

executearquivo "teste.prisma" ;
//deve ser o mesmo nome criado acima, ok!

//saída -- >  de 1 a 100 na tela do cmd.
```

Você pode usar em combinação com a função carreguearquivo e compilar um arquivo, veja:

```prisma
//ex. carreguearquivo

x = carreguearquivo( 'teste.prisma') ;
 //carrega um arquivo prisma em uma função

g = compile ( x ) ;
//podemos compilar x já que é uma função
// retornando a string compilada para g;

a = es.abra( 'out.prisma' , 'escritabin');
//abrindo um arquivo em modo escrita binária

a:escreva( g ) ;
//escrevendo a string compilada para o arquivo

a:feche( );

//pronto o arquivo foi compilado para byte-code
//a partir de um programa feito em prisma!
```

pchame (funcao, argumentos)

Chame (executa) uma função em modo protegido, isto é, se a função tiver erros a mesma não interfere na execução do programa principal. Ela retorna dois valores, o primeiro é um boolean (falso ou verdadeiro) e o segundo é a mensagem de erro caso tenha.

Caso haja um erro a função não é executada e o primeiro retorno é um falso seguido pela mensagem de erro se não tiver erro a função é executada, o primeiro retorno é verdadeiro e o segundo é o valor de retorno da função se houver, veja:

O primeiro parâmetro é a função a ser chamada e do segundo em diante são seus argumentos se tiver.

Prisma

```
funcao soma ( a , b )
   imprima( a + b );
fim

x , msg = pchame ( soma , 'a' , 3 );
//isto gerará um devido a soma de uma letra com um número

se x == falso entao

          imprima( msg )
//imprimindo a mensagem de erro

senao

         imprima 'Função executada com êxito' ;

fim
```

xpchame(funcao , funcao_erro , argumentos ...)

Muito semelhante a de cima, mas com uma função de erro customizada, veja:

Prisma

```
funcao soma( a , b )

     imprima( a  + b );

fim

funcao imprima_erro ( msg_erro )

    imprima ( "ocorreu um erro, veja: " , msg_erro );

fim

a , b = xpchame ( soma , imprima_erro , 2 , 'a' ) ;
```

```
// o 'a' gerará um erro, pois é uma operação aritmética com string

imprima ( a , b );

leia( );
```

selecione(['#' ou inteiro] , args ...)

Essa função realiza uma seleção entre vários argumentos passados como parâmetro, ou se usar o '#' retorna o número de argumentos.

sintaxe: selecione (tipo , args); //onde tipo pode ser '#' ou um número inteiro.

ex.:

Prisma

```
  funcao    tam ( ... );
      imprima(  'voce passou ' , selecione('#' , ... ) , 'argumentos');
  fim
tam( 1 , 2 , 3, 4 , 5 ); //saida = voce passou    5   argumentos

```

No exemplo acima retorna o número de argumentos variados, pois usamos o tipo '#' como primeiro parâmetro para selecione;

Podemos usar um número inteiro e definir o retorno a partir dessa posição para frente dos argumentos passados:

Prisma

```
funcao selec ( num , ... )
  local args = selecione ( num  , ... );
  imprima('voce escolheu a partir do argumento' , num , selecione(tipo , ... )  )
fim
selec( 3 , 12 , 34 , 56 , 78 , 90 );
```

Saída será: voce escolher a partir do argumento 3 34 56 78 90

Perceba que ao usar o 3 como primeiro parâmetro da função selecione, ela retorna os elementos a partir do terceiro, se usássemos o 4 seria a partir do quarto, se 5 a partir do quinto, se 2 a partir do segundo e assim por diante.

Cap. 2 – Funções da biblioteca string

Muito importante para o processamento de strings como procurar, recortar, substituir trecho de string por outra, etc.

string.cod / string.byte

A mesma função com nomes diferentes, use a forma que preferir.

Relembre a funçao **cod,** é exatamente a mesma função empacotada na biblioteca string. Ela converte um caractere para sua representação em bytes, veja:

Aceita dois parâmetros, o primeiro é o caractere ou string, o segundo é o número que indica na sequência qual caractere da string a ser convertido. Quando o segundo parâmetro é omitido, o primeiro caractere da string é o padrão.

Prisma

```
imprima (string.byte ("ABC")) // saída --> 65 *(A é 65 in ASCII)
//converte o primeiro car, o 'A' pois o segundo parâmetro foi omitido
imprima (string.cod ("ABC", 1 ,  3 ) )
// ( 'string' , ponto_inicial , ponto_final )
//retorna 3 valores a primeira, segunda e terceira letra.
//saída -- > 65   66    67
// se quiséssemos salvar os valores em variáveis, fica assim:
a , b , c = string.byte( "ABC" , 1  , 3 ) ;
// ( 'string' ,  ponto_inicial , ponto_final )
imprima( a , b , c ) ; //saída -->  65    66    67

//ou em uma tabela:
tab = tabela.empacote(   string.cod ( "ABCDE" , 1 , 5 )    );
//empacota os resultados em uma tabela.
imprima( tabela.desempacote( tab )   );
```

string.car

exatamente a mesma função car vista na biblioteca base, use qual preferir

Converte o código numérico (ASCII) em caractere;

Aceita número variado de parâmetros separados por vírgula e retorna a string formada por eles, ou caractere se for único.

sintaxe: s = string.car (n1, n2, n3, ...)

Descrição:

Recebe 0 ou mais números e converte todos em caracteres correspondentes. É o oposto da função string.byte / string.cod

Prisma

```
imprima ( string.car (65, 66 , 67 ) ) //saída --> ABC
```

string.compile

Exatamente a mesma função que vimos na base: compile, use o modo que preferir.

Converte uma função em binário (byte-codes);

Sintaxe: s = string.compile (f)

Descrição:

Converte uma função f em sua representação binária, que pode ser posteriormente processada com a função carregue( ).

Como a função compile foi bem tratada na biblioteca base, aqui só vou demonstrar um simples exemplo.

Ex.:

```
Prisma
1  funcao f () imprima "Alo, Mundo" fim
2      s = string.compile (f)
3  tente (carregue (s) ) () //--> Alo, Mundo
4
5  //não estranhe ao colocar os () na frente da função tente,
6  // o retorno, que é uma função é executado logo após tente ( carregue(s) );
7  //é o mesmo que g = tente ( carregue (s) ) ;    g ( ); mas sem o g.
```

string.procure

Procura uma ‘string’ chave em uma string maior retornado sua posição:

sintaxe: string.procure(“string maior” , “chave” , número_inicio);

string maior é a string onde queremos procurar a string chave;

chave é a string a ser encontrada na maior;

número_inicio é a posição inicial para começar a procurar na string maior, se omitido, seu valor padrão é 1;

Ex.:

```
Prisma
1   s = "esta é uma string onde iremos procurar por uma string menor";
2
3           pos_inic , pos_fim = string.procure( s , 'menor' );
4  //localiza a string 'menor' dentro de s;
5
6          imprima ( pos_inic , pos_fim );
7  //saída --> 56   60
```

Há dois retornos, o primeiro é a posição inicial da string localizada, o segundo é a posição final, portanto precisamos usar duas variáveis para receber os valores.

Como dito antes o terceiro parâmetro, índice de posição inicial de busca, é opcional, e por padrão é 1, isto é a busca inicia pelo primeiro caractere da string.

Mas, as vezes você pode querer procurar por mais resultados da mesma chave de busca, e para isso o terceiro parâmetro é indispensável. A estratégia é guardar a posição do primeiro resultado de busca e por meio de um laço de repetição reiniciar a busca do ponto posterior ao último resultado, veja:

Prisma

```
s = " alo mundo, alo mundo , alo , mundo, alo";

pi = {};
//vamos criar duas tabelas (matrizes)
// uma posição inicial (pi) e outra posição final (pf)

pf = {};

x = 1;

pos_busca = 1; //ponto inicial da busca

enquanto 1 inicio //cria um laço infinito


    pi[ x ] , pf [ x ] = string.procure ( s , 'alo' , pos_busca );
//como pf[1] = 0 entao 0+1 = 1 , inicia do primeiro caractere a busca



    se pi[x] == nulo entao
//se o retorno de pi for nulo entao interrompe o laço
                quebre
    senao                                //senao salva uma nova posicao de busca
                 pos_busca = pf [ x ] + 1;
    fim;


x = x + 1;
//fazendo o incremento de x

fim

//imprimindo todos os resultados:

para i = 1 , #pi  inicio

      imprima( 'Busca numero ' .. i .. ' : ' .. pi[ i ]  .. ' - ' .. pf [ i ] );

//imprime todos os resultados para a busca 'alo'
// a posição inicial e final dentro da string s

fim
```

É possível usar patterns de busca (caracteres especiais de busca).

Localizando uma string entre aspas, parenteses e chaves:

Prisma

```
 s = " x = 'entre_aspas' ; imprima(entre_parenteses) ; tab = {entre_chaves} {asdf} ";
 imprima( string.procure( s , "'.*'" , 1 ); //--> 6  18
 imprima( string.procure( s , "'(.*)'" , 1 ) ) ;
//--> 6 18 entre_aspas

 imprima( string.procure( s , '%((.*)%)' , 1 ) ) ;
//como ( já é um sinal interno, usamos %( e %)
  //--> 29 46 entre_parenteses

```

```
10   imprima( string.procure( s , '{(.+)}' , 1 ) );
11  //-> 56 76 entre_chaves} {asdf
12   imprima( string.procure( s , '{(.-)}' , 1 ) );
13  //o menos obtém somente a 1ª ocorrêcia
14  //-> 56 69 entre_chaves
```

string.formate

Esta função serve para formatar strings por meio de caracteres de formatação, muito semelhante ao sprintf do C.

Sintaxe: retorno = string.formate(“String_formatacao” , args , ...);

Exemplo básico:

Prisma

```
1  s = string.formate( "Ola %s!" , "Mundo");
2  imprims(s); // -> Ola Mundo!
```

Perceba que a saída é a primeira string mas no lugar do %s é posto o segundo argumento “Mundo”.

O %s é um exemplo de caractere de formatação para strings.

Número variado de argumentos

```
1  s = string.formate ( 'Ola %s em %s!' , 'Mundo' , 'Prisma');
2
3  imprima(s); // -> Ola Mundo em Prisma!
```

Note que poderá ser muito mais do que dois argumentos de formatação.

Obs.: para cada argumento de formatação deve haver um caractere de formatação respectivo(na mesma sequência).

Outros caracteres de formatação e exemplos:

Prisma

```
1  %c, %d, %E, %e, %f, %g, %G, %i, %o, %u, %X, e %x são usados para números.
2
3  %q e %s para string.
```

Veja os exemplos de uso a seguir:

Prisma

```
s = string.format("%s %q", "Ola", "Programador Prisma!")
// string normal e string com aspas
imprima(s); //-> Ola "Programador Prisma!"

s = string.formate("%c%c%c", 79,108,97) // caracteres
imprima(s); //-> Ola

s = string.formate("%e, %E", mat.pi,mat.pi)
// exponenciação(elevado)
imprima(s); //-> 3.141593e+000, 3.141593E+000

s = string.formate("%f, %g", mat.pi,mat.pi)
// float(real) e compacto float
imprima(s); //-> 3.141593, 3.14159

s = string.formate("%d, %i, %u", -100,-100,-100)
// signed, signed, unsigned (todos inteiros)
imprima(s); //-> -100, -100, 4294967196

s = string.formate("%o, %x, %X", -100,-100,-100)
// números: octal, hexadecimal, hexadecimal
imprima(s); //-> 37777777634, ffffff9c, FFFFFF9C
```

string.troque

Sintaxe: retorno1 , retorno2 = string.troque(“stringOriginal” , “string_pesquisada” , “troca” [, numero_trocas])

Um recurso muito útil quando você quer trocar uma string por outra dentro de uma maior. Um quarto parâmetro opcional limita a quantidade de substituições. Veja que retorno 1 é a string original já trocada, e retorno2 é o numero de trocas feitas.

Ex.:

```
imprima( string.troque( "Ola Mundo!" , "Mundo" , "Prisma" ) ); //-> Ola Prisma!
```

Observe que a função procura a string ‘Mundo’ dentro da primeira string, caso a localize, a substituição é feita por “Prisma”.

Outros exemplos:

Podemos usar caracteres especias para troca:

```
 s , n = string.troque("banana", "(an)", "%1-")
// captura toda ocorrência an e substitui por an-
imprima(s,n); // ban-an-a        2

s , n = string.troque("banana", "a(n)", "a(%1)")
// parenteses em torno dos 'n' depois de 'a'
imprima(s , n ) ; // ba(n)a(n)a      2

s , n = string.troque("banana", "(a)(n)", "%2%1")
```

Prisma

```
// em toda ocorrência 'an' é trocado por 'na'
imrpima( s , n ); //   bnanaa       2
```

Entendendo Melhor

Você pode estar um pouco confuso sobre os ‘patterns’ (caracteres especiais para troca),

lembre-se de que o segundo argumento é para pesquisa e o terceiro para troca:

“(an)” = onde encontrar ‘an’ na string original ocorrerá a troca;

‘%1-‘ = %1 simboliza o argumento de pesquisa, o primeiro entre parênteses seguido pelo – ; logo a troca será an-

‘a(n)’ = todo ‘an’ para pesquisa sendo o ‘n’ capturado para usar como %1 na troca;

‘a(%1)’ = a troca será por a + ‘n’ simbolizado por %1

‘(a)(n)’ = an para pesquisa, na troca o a será = %1 e o n = %2

Trocando toda palavra válida por ela mesma entre sinais: (ex.: ola por <ola> ou “ola”)

Prisma

```
s = "ola mundo em Prisma";
imprima( string.troque (  s , '(%w+)' , '<%1>' )   );
// <ola> <mundo> <em> <Prisma> 4

imprima( string.troque( s , '(%w+)' , '"%1"' )  );
// "ola" "mundo" "em" "Prisma" 4
imprima( string.troque ( "dia 15 de abril" , '(%w+)' , '/%1/' ) );
//= /dia/ /15/ /de/ /abril/
```

string.capte

sintaxe: iterador = string.capte(string , ‘chave’)

Esta função retorna um iterador de busca por chave. O iterador procurará através da string passada buscas por resultados da chave passada. Observe que é passado um iterador que pode ser usado com a função ‘para’, ex. :

Prisma

```
para palavra em string.capte( 'ola mundo em <prisma>; 2015 ', "%a+") inicio
    imprima( palavra )
fim
/**saída:
ola
mundo
em
```

```
prisma
**
```

o pattern (a chave) %a+ representa qualquer sequência de letra válida exceto números e sinais;

Tente modificar o programa acima substituindo “%a+” por “%d+” ou “%w+” e veja os resultados.

string.tamanho

Retorna o tamanho de uma string, sintaxe: retorno(número) = string.tamanho(‘string’).

Ex.:

Prisma

```
imprima(   string.tamanho (  'Ola Mundo em Prisma'  )   );
// 19
```

string.maiuscula

Retorna uma string que está em minúscula no formato maiúscula.

Sintaxe: retorno(STRING) = string.maiuscula(‘string’)

Ex.:

Prisma

```
 s = string.maiuscula( 'ola mundo');
    imprima(s); //-> OLA MUNDO
```

Obs.: caso já tenha alguma letra em maiúscula, nada é feito.

string.minuscula

Retorna uma string que está em maiúscula para minúscula.

Ex.:

Prisma

```
s = string.minuscula('OLA MUNDO');
imprima(s); //--> ola mundo
```

Obs.: caso já tenha alguma letra em minúscula, nada é feito.

string.nconcat

Retorna uma string concatenada n vezes. Sintaxe: retorno = string.nconcat(“string” , numero_concat).

Ex.:

```
Prisma
s = string.nconcat( 'Ola' , 3 );
imprima( s ); //-> OlaOlaOla
```

string.inverta

Retorna uma string com ordem das letras invertidas. Sintaxe: retorno = string.inverta(“string”);

Ex.:

```
Prisma
s = string.inverta('ABC');
imprima(s); //-> CBA
```

string.sub
(ou se preferir: string.corte)

Retorna uma substring de uma maior. Sintaxe: retorno = string.sub(“string_origial” , inicio , fim);

Onde inicio e fim são números, sendo 1 o início da string.

Números negativos representam do fim para o início, -1 é o último caractere.

Ex.:

```
Prisma
s = 'ola mundo';
imprima( string.sub( s , 1 , 2)  ); //->; ol
imprima( string.sub( s , 1 , 3 ) ); //-> ola
imprima( string.sub( s , 1 , -1 )  ); //-> ola mundo
imprima( string.sub( s , -2 , -1)  ); //-> do
```

Caso o segundo argumento seja omitido, então o valor padrão é o último caractere da string.

```
Prisma
s = 'ola mundo';
imprima( string.sub( s , 2 ); //-> la mundo   //é contado do 2 caractere até o final.
```

Cap. 3 funções do sistema operacional

Esta biblioteca permite a comunicação com algumas funções do sistema operacional, tais como data, horário, execute, etc.

Todos os métodos ficam guardados na tabela ‘sis':

sis.relogio()

Retorna o tempo da CPU desde o início da execução de Prisma, em segundos.

Prisma

```
t = sis.relogio();
imprima(t) // saida: 11056.989
```

sis.data()

Retorna informações sobre data e horários. Sintaxe: sis.data(formatacao/opções);

ex.: imprimindo a data sem formatação, o resultado depende do sistema operacional:

Prisma

```
imprima( sis.data() ); //saida: Wed Apr 22 20:46:23 2015
```

exemplo formatado: (%d = dia, %m = mês, %Y = ano)

Prisma

```
imprima( sis.data("%d.%m.%Y")  ); //   22.04.2015
```

Retornando uma tabela (‘*t’)

Prisma

```
T = sis.data("*t");
imprima( T.hora , T.minuto , T.segundo );
```

No exemplo logo acima, o argumento “*t” força o retorno de uma tabela.

Qualquer nome que escolher como retorno conterá os campos:

Prisma

```
.diaano // dia corrido de 1 a 365

.dia // dia do mês 1 - 31 ou 1-30;

.mes // o mês em forma numérica. (1-12)

.hora // hora(1-23/ 24=00 dependendo do idioma do sistema)

.minuto // minuto (1-59)

.segundo // segundo( 1-59)

.horariodeverao // se é horário de verão (falso ou verdadeiro)

.diasemana // dia da semana em número(1=domingo, 2 = segunda ...);

.ano // ano em forma numérica ( 2015, por exemplo)
```

sis.diftempo()

Retorna a diferença de tempo entre t2 e t1

ex.:

Prisma

```
t1 = sis.tempo();
 imprima( sis.diftempo(  sis.tempo( ) , t1 ) );  // 43
```

sis.execute()

Execute um comando de terminal do sistema operacional, semelhante ao system() do C:

Prisma

```
sis.execute('clear'); //linux limpa tela.
```

Prisma

```
sis.execute('cls') ; //Windows limpa tela.
```

É possível chamar programas externos:

Prisma

```
sis.execute('firefox www.google.com.br');
```

retorna verdadeiro exit 0 em caso de sucesso ou nulo exit: cod_erro em caso de falha

sis.saia()

Fecha a execução do programa e devolve um número de retorno para o sistema operacional:

```Prisma
sis.saia( 0 ); //fecha o programa e retorna zero ao sistema operacional
```

sis.obtvarambiente()

Retorna o valor de uma variável de ambiente se ela estiver definida:

```Prisma
imprima( sis.obtvarambiente( "USER")  );
```

sis.remova()

Remove um arquivo, retorna verdadeiro caso tenha sucesso ou nulo msg_erro em caso de falha:

```Prisma
imprima (  sis.remova( "teste.txt" )  );
```

sis.renomeie()

Renomeia um arquivo, retorna verdadeiro ou falso + msg_erro;

```Prisma
imprima( sis.renomeie ( "teste.txt" , "novo_nome.txt" )  );
```

sis.nometmp( );

Sintaxe: nome = sis.nometmp();

Retorna um nome aleatório para ser usado como arquivo temporário. O retorno depende do sistema:

```
nome_arq = sis.nometmp(); imprima (nome_arq)
```

Saída no linux Ubuntu 14.04 = /tmp/prisma_yShId3

Windows Xp = \s1fo.

Você pode acrescentar extensões se quiser usando o operador de concatenar strings o “..”:

```
imprima( nome_arq .. ".ext" )
```

saída provável: linux = /tmp/prisma_yShId3.ext

no Windows = \s1fo..ext //no windows basta usar ‘ext’ ao invés de ‘.ext’ para não duplicar o ponto

OBS.: Note que esse comando apenas gera uma string não um arquivo, você teria que usar a função es.abra e passar o nome temporário gerado para ela.

sis.deflocal(local [, tipo])

Define o idioma (local) atual do programa.

sintaxe: sis.deflocal(local [, tipo])

Em que local é o idioma a ser aplicado, a string idioma depende do sistema operacional;

Se o primeiro e segundo parâmetros forem omitidos, então o retorno é o idioma já definido:

```
imprima ( sis.deflocal() ) //-- saída em Ubuntu 14.04 = pt_BR.UTF-8
```

//saída no Windows XP = Portuguese_Brazil.1252

Se quiséssemos definir o idioma seria :

```
sis.deflocal ( "pt_BR.UTF-8" );
```

no Win :

```
sis.deflocal('Portuguese_Brazil.1252' );
```

Note que se o segundo parâmetro for evitado a definição do idioma é geral, mas podemos escolher o que mudar, por exemplo, se quiséssemos mudar somente os caracteres:

Prisma

```
sis.deflocal( "pt_BR.UTF-8" , "ctype" );
//só mudaria o teclado e os caracteres de saída
```

Eis abaixo as opções disponíveis.

Prisma

```
"collate" // controla a ordem alfabética das strings;

"ctype" // codificação dos caracteres

"monetary" // não influencia nada em prisma;

"numeric"// formatação de números,
//ex. no inglês separa-se casas decimais por ponto, no português por vírgula

"time" // formatação de datas e horários

"all" //ativa todas as anteriores
```

OBS.: eu deixei em inglês mesmo para não haver confusão, visto que é fácil. No entanto, caso não se acostume copie o código abaixo em seu programa e use as variáveis personalizadas a seu gosto:

Prisma

```
string_formato = "collate";

caracteres = "ctype"

moeda = "monetary"

numerico = "numeric"

tempo_formato = 'time' ;

tudo = 'all'
```

use assim:

```
sis.deflocal("pt_BR.UTF-8" , caracteres ) ; //ou  string_formato, numerico, moeda, tempo_formato,
```

Cap. 4 funções de entrada e saída padrão ‘es’

Essa biblioteca providencia funções para abrir e manipular arquivos, escrever, ler;

es.escreva(‘strings’ , ...);

Esta função escreve na tela preta do terminal no Linux ou cmd no Windows. Semelhante ao comando imprima() com a diferença de que este não pula linhas. Ex:

```
es.escreva( 'Ola Mundo');
es.escreva(" Em Prisma");
```

Saída = Ola Mundo E Prisma

Se tivéssemos usado o imprima, sairia assim:

Ola Mundo

Em Prisma

Obs.: es.escreva aceita vários argumentos, ex.:

```
es.escreva ( 'ola' , ' Mundo ' , ' em  Prisma')
```

es.leia ( );

é o mesmo que leia()

Esta função é usada para ler dados digitados pelo usuário, ou os dados de um arquivo;

O segundo uso veremos adiante, agora veja como ler do teclado:

```
es.escreva('Digite seu nome: ');
nome = es.leia();//o que o usuário digitar será armazenado na variável nome
imprima('Seu nome é' , nome );
```

———

PRIMEIRO MODO DE LER E ESCREVER EM ARQUIVOS

es.saida( )

Abre um arquivo de saida padrão para escrita, ex.:

```
    es.saida( "C:\nomearquivo.txt" ); //abre o arquivo
    es.escreva("primeira linha do arquivo\n" ; //escreve uma linha
    es.escreva("\npulou uma linha"); //o escape \n pula linhas na escrita
    es.feche();
```

//vá até a pasta disco C:\ e abra o arquivo criado ‘nomearquivo.txt’ em um editor de sua

preferência e veja as linhas gravadas.

es.entrada( )

Esta função abre um arquivo de entrada padrão para leitura:

```
es.entrada("C:\nomearquivo.txt");
x = es.leia("*t"); //colocando o "*t" o arquivo é lido inteiro
es.feche();
imprima(x); //imprime o conteúdo do arquivo.
```

es.linhas( )

Essa função retorna um iterador para usar com o comando ‘para’, percorrendo todas as linhas de um arquivo:

ex.:

```
para lin em es.linhas('C:\nomearquivo.txt')  inicio
   imprima( lin );
fim
```

No exemplo acima, será impressa cada linha do arquivo. Se preferir pode armazenar em uma variável string ou tabela.:

```
tab = {};
i = 1;
para lin em es.linhas('C:\nomearquivo.txt')  inicio
   tab[i] = lin;
i = i + 1;
fim
```

es.feche( )

Fecha um arquivo padrão aberto:

```
es.entrada('C:\nomearquivo.txt');//abre o arquivo para leitura
 x = es.leia"*t"; //le todo o arquivo
es.feche(); //fecha o arquivo para disponibilizar o conteudo
imprima(x); //imprime o conteudo de x (que é o arquivo lido);
```

SEGUNDO MODO DE LER E ESCREVER EM ARQUIVOS

es.abra( )

Sintaxe: objeto_arquivo = es.abra(‘nomearquivo’ , ‘modo’) ;

em que nome arquivo é o caminho absoluto ou o relativo do arquivo, e modo pode ser:

‘leitura’ — para ler um arquivo

‘escrita’ — para escrever no arquivo, apagando o conteúdo original

‘leiturabin’ — leitura binária

‘escritabin’ — escrita binária, apaga o conteúdo original

‘adicao’ — escreve no final do arquivo, não apaga o conteúdo original

ex.:

Escrevendo

```
arq  =  es.abra('C:\nomearquivo' , 'escrita');
     arq:escreva("ola mundo" , " em Prisma\n"); // o escape \n pula para outra linha
arq:feche();
```

Note que neste caso criamos uma variável que herda os métodos escreva e feche;

OBS.: não esqueça de usar os dois pontos para acessar os métodos do objeto aberto.

Lendo

‘

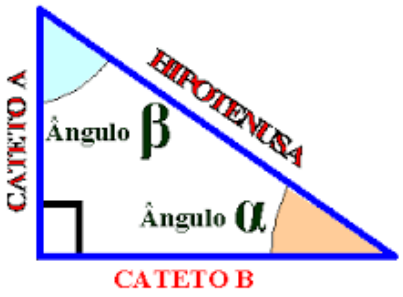
```
arq  =  es.abra('C:\nomearquivo' , 'leitura');
     var_conteudo = arq:leia("*t"); //com o "*t" todo o arquivo é lido de uma vez
arq:feche();

imprima( var_conteudo );  //imprime o conteúdo lido do arquivo.
```

Adicionando

```
arq  =  es.abra('C:\nomearquivo' , 'adicao');
     arq:escreva("ola mundo" , " em Prisma\n"); // o escape \n pula para outra linha
arq:feche();
```

Semelhante ao modo ‘escrita’ com a diferença que esse adiciona no fim do arquivo a gravação, e não apaga o conteúdo original.

Cap. 5 funções da biblioteca mat – trigonometria e aritmética



Suas funções ficam na tabela ‘mat’

Muito importante para realizar operações trigonométricas e aritméticas, como obter o arco tangente, o co-seno, exponenciação, absoluto, módulo(resto), raiz quadrada etc.

Vamos começar então!

mat.absoluto(n);

Retorna o valor absoluto de n. Veja:

Prisma

```
1  imprima( mat.absoluto(5) , mat.absoluto(-5) );
2  //saída =   5
```

Não quero aprofundar em matemática, portanto somente farei a descrição da sintaxe e retorno das funções, ok!

mat.arcocosseno(n)

Retorna o arco cosseno de n (em radianos);

ex.:

Prisma

```
 imprima( mat.arcocosseno(0.3) )
// saida: 1,2661036727795
 imprima( mat.arcocosseno(1) )
//saída: 0
 imprima( mat.arcocosseno(2) )
//saída: nan
 imprima( mat.arcocosseno(-1) )
//saída: 3,1415926535898
```

mat.arcoseno(n)

Retorna o arco seno de n (em radianos);

Ex.:

Prisma

```
imprima( mat.arcoseno(1) )
//saída: 1,5707963267949
imprima( mat.arcoseno(2) )
//saída: nan
imprima( mat.arcoseno(0) )
//saída: 0
imprima( mat.arcoseno(-1) )
//saída: -1,5707963267949
imprima( mat.arcoseno(0.3) )
//saída: 0,3046926540154
```

mat.arcotangente(n)

Retorna o arco tangente de n (em radianos);

Prisma

```
imprima( mat.arcotangente(9) )
//saída: 1,460139105621
imprima( mat.arcotangente(1) )
//saída: 0,78539816339745
imprima( mat.arcotangente(3) )
//saída: 1,2490457723983
```

mat.arcotangente2(n1 , n2)

Retorna o arco tangente de n1/n2 (em radianos), no entanto, utiliza o sinal dos dois parâmetros para achar o quadrante do resultado. (Também trata corretamente o caso de x ser zero.)

Prisma

```
imprima( mat.arcotangente2( 2 , 4) )
```

```
//saída: 0,46364760900081
imprima( mat.arcotangente2( 1 , 5) )
//saída: 0,19739555984988
imprima( mat.arcotangente2( 4 , 8) )
//saída: 0,46364760900081
```

mat.arredonde (n)

Retorna o n caso seja inteiro, ou arredonda para mais caso seja fracionário. Veja:

Prisma

```
imprima( mat.arredonde( 9.2 )  )
//10
imprima( mat.arredonde( 1.4 ) )
//2
 imprima( mat.arredonde( 4 ) )
//4
```

mat.cosseno (n)

Retorna o cosseno de n (n deve estar em radianos).

Prisma

```
imprima( mat.cosseno(0.5) )
//0,87758256189037
imprima( mat.cosseno(0.9) )
//0,62160996827066
imprima( mat.cosseno(1) )
//0,54030230586814
imprima( mat.cosseno(2) )
// -0,41614683654714
```

mat.cossenoh (n)

Retorna o cosseno hiperbólico de n.

Prisma

```
 imprima( mat.cossenoh(9) )
//4051,5420254926
imprima( mat.cossenoh(1) )
//1,5430806348152
imprima( mat.cossenoh(-2) )
//3,7621956910836
```

mat.emGrau (r)

Converte r que está em radianos para graus.

Prisma

```
 x = mat.arcocosseno(0.9)
imprima( x , mat.emGrau(x) )
//0,45102681179626    25,841932763167
```

mat.exp (n)

Retorna o número de euler elevado a n (e^n). Euler é um número irracional, aproximadamente é 2,718281828459045, ou, às vezes apenas 2,718281828459).
O número de Euler é comum ser chamado de **e.**

Prisma

```
imprima(
  "o numero 2,718281828459045 elevado a 3 = " ,
  mat.exp(3)

)
//o numero 2,718281828459045 elevado a 3 =     20,085536923188
```

mat.corte (n)

Arredonta por baixo, se 2.5, por exemplo, será 2, ou quando o número for inteiro retorna o próprio número.

Prisma

```
imprima(  mat.corte(1.3) , mat.corte( 2.9 ) , mat.corte(5)   );
//1        2         5
```

mat.cmodulo (n1 , n2)

Retorna o resto da divisão de n1 por n2 que arredonda o quociente em direção a zero.

Prisma

```
imprima( mat.cmodulo ( 5 , 3) , 5%3)
 //2    2
imprima( mat.cmodulo ( 10 , 3) , 10%3)
//1    1
```

mat.frexp (n)

Retorna **m** e **e** tais que **n = m2^e**, **e** é um inteiro e o valor absoluto de **m** está no intervalo **[0.5, 1)** (**ou zero quando x é zero**).

Prisma

```
_M , _E = mat.frexp(9)
 imprima(  _M , _E , _M*2^_E )
//  0,5625      4      9
```

mat.infinito

Na verdade não é uma função, é um dado que representa o valor de HUGE_VAL, um valor maior ou igual a qualquer outro valor numérico.

Prisma

```
imprima( mat.infinito)
//  inf
imprima( 10 < mat.infinito )
// verdadeiro
imprima( mat.infinito > 1000000000 );
//verdadeiro
imprima( mat.infinito > mat.infinito )
//falso
imprima( mat.infinito <  mat.infinito )
//falso
```

mat.ldexp(m , e)

Retorna m2^e (**m** vezes 2 evevado a **e.** **e** deve ser um inteiro).

O inverso de **mat.frexp(n).**

Prisma

```
x = 8
imprima( mat.frexp( 8 )  )
// 0,5    4
imprima( mat.ldexp( 0.5 , 4 ) )
//8
```

mat.log(n)

Retorna o logaritmo natural de n. Isto é, retorna a potência em que o número de Euler (**2,718281828459045**) precisa estar elevado para que resulte no seu logaritmo.

Prisma

```
 x = 2.718281828459045
 r = x^3 //x elevado a 3
 imprima( r )
// 20,085536923188
imprima( mat.log(r) )
// 3
//retornou 3 pois é preciso o número
//de Euler ( x ) se elevar a 3 para dar
// r.
```

mat.log10(n)

Retorna o logaritmo base-10 de n. Ou seja, retorna o valor que dez precisa ser elevado para dar **n.**

Prisma

```
imprima( mat.log10(100 ) )
// 2     dez elevado a 2 é 100

imprima( mat.log10( 900 ) )
// 2.9542425094393
//dez elevado a 2.954245094393 é 900

imprima( mat.log10(1000 ) )
// 3
// 10 elevado a 3 é 1000
```

mat.maximo(...)

Retorna o valor máximo entre os seus argumentos.

Prisma

```
imprima ( mat.maximo( 1 , 3 , 5 , 6 , 100 , -1 , -1000 , 500 )  )

// 500
```

mat.minimo(...)

Retorna o valor mínimo entre os seus argumentos.

Prisma

```
imprima ( mat.minimo( 1 , 3 , 5 , 6 , 100 , -1 , -1000 , 500 )  )

// -1000
```

mat.separe(n)

O retorno é duplo, considerando **n** um número fracionário (1.5 por exemplo), serão retornadas a parte integral de n (1) e a parte fracionária de n (0.5).

Prisma

```
int , frac = mat.separe( 2.9999 )
imprima( int , frac )
// 2    0,9999

//quando o número é inteiro
imprima( mat.separe( 2 ) )
2    0
// a parte fracionária é zero
```

mat.pi

Não é função, representa o valor de pi:

Prisma

```
 imprima ( mat.pi );
 // 3,1415926535898

imprima( mat.separe( mat.pi ) )
// 3    0,14159265358979
//acontece a aproximação ao
//usar a função mat.separe
```

mat.elevado(n1 , n2)

Retorna n1^n2 (n1 elevado a n2). Esta operação é possível com o sinal de ^ assim como se pode somar n1+n2 direto em prisma, pode-se usar n1^n2 diretamente.

Prisma

```
 imprima( mat.elevado(10 , 2 ) , 10^2   );
// 100    100
// 10^² = 100
```

rad = mat.emRadianos(graus)

Converte de graus para radianos:

Prisma

```
 rad = mat.arcocosseno(0.9)
 imprima(rad)//radianos
// 0,45102681179626
grau = mat.emGrau( rad );//graus
imprima( grau )
// 25,841932763167
rad2 = mat.emRadianos( grau )
imprima( rad2 ) //radianos novamente
// 0,45102681179626
```

mat.randonico(inicial , final)

Retorna um inteiro pseudo-aleatório no intervalo inicial e final.

Prisma

```
 imprima( mat.randonico( 1 , 5 ) )
//5
 imprima( mat.randonico( 1 , 5 ) )
//2
imprima( mat.randonico( 1 , 5 ) )
//4
imprima( mat.randonico( 1 , 5 ) )
//4
imprima( mat.randonico( 1 , 5 ) )
//5
imprima( mat.randonico( 1 , 5 ) )
//1
imprima( mat.randonico( 1 , 5 ) )
//2
imprima( mat.randonico( 1 , 5 ) )
//4
```

Se omitido os parâmetros inicial e final, o retorno é um aleatório entre 0 e 1;

Prisma

```
//imprima( mat.randonico() )
//0,84018771715471
imprima( mat.randonico() )
// 0,39438292681909
imprima( mat.randonico() )
// 0,78309922375861
imprima( mat.randonico() )
//0,79844003347607
imprima( mat.randonico() )
// 0,91164735793678
imprima( mat.randonico() )
// 0,19755136929338
```

Se passarmos apenas um argumento, o retorno será um aleatório entre 1 até o argumento passado.

Prisma

```
imprima( mat.randonico(9) )
// 8
imprima( mat.randonico(9) )
//4
imprima( mat.randonico(9) )
//8
imprima( mat.randonico(9) )
// 8
imprima( mat.randonico(9) )
// 9
imprima( mat.randonico(9) )
// 2
```

mat.xrandonico(n)

Define um número gerador pseudo-randônico. Gerador igual produz sequências iguais:

Prisma

```
mat.xrandonico(1234)
imprima(  mat.randonico(), mat.randonico(), mat.randonico()   )
// 0.12414929654836        0.0065004425183874      0.3894466994232
mat.xrandonico(1234)
imprima( mat.randonico(), mat.randonico(), mat.randonico()
// 0.12414929654836        0.0065004425183874      0.3894466994232
```

Um ótimo recurso para criar um gerador é a função sis.tempo():

Prisma

```
mat.xrandonico( sis.tempo() )
```

mat.seno(n)

Retorna o seno de n *(n deve estar em radianos).*

Prisma

```
rad = mat.arcotangente( 9 )
imprima( rad )
// 1,460139105621
seno = mat.seno( rad )
imprima( seno )
// 0,99388373467362
```

mat.senoh(n)

Retorna o seno hiperbólico de n.

Prisma

```
imprima( mat.senoh( 1 ) )
// 1,1752011936438
```

mat.raizquad (n)

Retorna a raiz quadrada de n.

Prisma

```
imprima( mat.raizquad( 100 )  );
// 10
```

A mesma operação pode ser feita com o ^(1/2)

Prisma

```
imprima( 100^(1/2) );
// 10
```

Raiz cúbica:

Prisma

```
imprima( 1000^(1/3) );
// 10
```

Raiz quártica, e assim por diante:

Prisma

```
imprima( 16^(1/4) );
// 2
imprima( 32^(1/5)  );
//2

 x = 3 ^ 9 //3 elevado a 9
 imprima(x)
 //  19683
 imprima( x^(1/9) )
 // 3
 x = 2^1000 //2 vezes 2 1000 vezes)
 imprima(x)
// 1,0715086071863e+301
 imprima( x^(1/1000) ) //raiz milésima de x:
// 2
```

mat.raiz (n , z)

Retorna a raiz de índice 'z' de n:

Prisma

```
imprima(  "raiz cúbica de 8 = " , mat.raiz( 8 , 3 ) )
// 2
imprima( "raiz de indice 10 de 1024 = " , mat.raiz(1024,10) );
//2
// 2 vezes 2, 10 vezes é 1024
```

mat.tangente (n)

Retorna a tangente de n (n deve estar em radianos).

Prisma

```
rad = mat.arcocosseno(0.6)
imprima(rad)
// 0,92729521800161
tan = mat.tangente(rad)
imprima(tan)
//1,3333333333333
```

mat.tangenteh (n)

Retorna a tangente hiperbólica de n.

```Prisma
imprima( mat.tangenteh(1) )
// 0,76159415595576
```

Cap. 6 Funções da biblioteca tabela

Tabela é um poderoso recurso de manipulação e descrição de dados em Prisma. Com ela é possível fazer uma simples matriz como em C, ou até mesmo algo mais complexo e de forma simples como um registro.

Tabelas em Prisma não precisam ser declaradas com tamanho fixo, são dinâmicas e flexíveis sempre aceitando mais um elemento. Ex:

```Prisma
tab = {}; //iniciando uma tabela vazia;

//adicionando elementos na tabela:

tab [1] = 'segunda';
tab[2] = 'terça';
tab.ano = 2015;
tab.mes = 'maio';

//iniciando uma tabela já com elementos definidos:
tab2 = { 'segunda' , 'terca' , ano = 2015 , mes = 'maio' };
```

A biblioteca tabela oferece facilidades para manipular esse tipo de dado. Seus métodos estão contidos no nome “tabela.”

tabela.concat(tabela [,sep [, inicial [, final]]])

Concatena (une) os elementos da tabela.

Omitindo os demais parâmetros:

```Prisma
tab = { "Ola " , "Mundo " , 2015 };

imprima( tabela.concat( tab )  );

//Ola Mundo 2015
```

Usando os demais parâmetros:

```Prisma
tabela.concat({ 1, 2, "tres", 4, "cinco" }, ", ")
//separador (vírgula)
// 1, 2, tres, 4, cinco
tabela.concat({ 1, 2, "tres", 4, "cinco" }, ", ", 2)
//separador mais ponto inicial a partir do segundo índice da tabela.
// 2, tres, 4, cinco
tabela.concat({ 1, 2, "tres", 4, "cinco" }, ", ", 2, 4);
//separador, delimitador inicial e final(do 2 ao 4 índice);
//2, tres, 4
```

tabela.ordene(tabela);

Coloca os índices em ordem crescente:

```Prisma
 tab = {9,7,8,1,6,2,5,4,3,0}
 imprima( tabela.concat( tab , ', ' ) )
// 9, 7, 8, 1, 6, 2, 5, 4, 3, 0
tabela.ordene( tab );
imprima( tabela.concat( tab , ', ' ) );
// 0, 1, 2, 3, 4, 5, 6, 7, 8, 9


 t = {'x' , 'a' , 'd' , 'c' , 'b'}
tabela.ordene( t )
imprima( tabela.concat( t , ', ') )
// a, b, c, d, x
```

A ordenação da tabela pode ser customizada por uma função como segundo parâmetro, esta função deve retornar um valor boolean (verdadeiro ou falso). O retorno padrão é **a<b**, veja:

```Prisma
 t = { 3,2,5,1,4 }
tabela.ordene(t, funcao(a,b) retorne a < b fim)
imprima(  tabela.concat(t, ", ")  );
// 1, 2, 3, 4, 5
```

ou defina **a>b** para ordenar em decrescente:

Prisma

```
t = { 3,2,5,1,4 };
tabela.ordene(t, funcao(a,b) retorne a > b fim );
imprima( table.concat(t, ", ")  );
// 5, 4, 3, 2, 1
```

tabela.insira (tabela , [pos,] valor);

Insere um valor em uma tabela. Se uma posição for dada, o valor é inserido no lugar do valor atual que sobe uma posição no índice:

Prisma

```
t = { 1,3,"quatro" }
tabela.insira(t, 2, "dois");
//insere "dois" na posicao antes do segundo elemento
imprima( tabela.concat(t, ", ") );
//1, dois, 3, quatro
```

Se a posição não for especificada, o novo valor é inserido no fim da tabela

Prisma

```
t = { 1,"dois" , 3 , "quatro" };
tabela.insira(t, 5);
imprima(  table.concat(t, ", ")  );
1, dois, 3, quatro, 5
```

Quando um novo valor é inserido em uma tabela, ela atualiza o tamanho e a sequência dos elementos:

Prisma

```
t = { 1,"dois",3 }; //criando a tabela
imprima( #t ); //imprimido o tamanho da tabela
// 3
//percorrendo em imprimindo os indices:
para indice , valor em ipares( t ) inicio
   imprima( indice , valor );
fim
/**
1       1
2       dois
3       3
**//

tabela.insira(t, 1, "novo_valor");
//novo_valor inserido no indice 1;

imprima( tabela.concat(t, ", ") );
//inserted, 1, two, 3
imprima( # t  );//tamanho atual da tabela
//4
para indice, valor em ipares( t ) inicio
```

```
   imprima( indice , valor );
fim
/**
1       novo_valor
2       1
3       dois
4       3

**//
//valores foram deslocados para um índice acima
//para dar espaço ao novo_valor
```

tabela.remova(tabela [, pos])

Remove um elemento da tabela na posição definida e retorna o elemento removido. Se a posição não for definida o padrão é o último elemento:

Prisma

```
t = { 1,"dois",3,"quatro" };//cria a tabela
imprima( #t ); //tamanho da tabela
// 4
para i , v em ipares( t ) inicio
   imprima( i , v );
fim
/**
1       1
2       dois
3       3
4       quatro
**//
removido = tabela.remova(t,2);
//remove o índice 2 e o retorna:
// dois

para i , v em ipares ( t )  inicio
  imprima( i , v );
fim
/**
1       1
2       3
3       quatro
**//
imprima( #t ); //tamanho atualizado
// 3
```

tabela.maxn(tabela);

Retorna o número de elementos da tabela:

Prisma

```
t = {123,23,3,2,3,3,4,5,5,5,6,4,4,2,43,0}
imprima( tabela.maxn(t) )
// 16
```

ou use #t

tabela.empacote(...);

Transforma o número variável de argumentos '...' em tabela, passando o retorno a uma variável.

Prisma

```
tab = tabela.empacote( 12, 234, 2345, 465, 213, 2, 44);

imprima( tabela.concat( tab , ', ' )  );

//  12, 234, 2345, 465, 213, 2, 44
```

tabela.desempacote(tabela);

O inverso da função anterior, transforma uma tabela em argumentos variados

Prisma

```
tab = { 12, 234, 2345, 465, 213, 2, 44 };

imprima( tabela.desempacote( tab )    );

//  12     234     2345     465    213     2     44
```

Obrigado por baixar o manual e usar Prisma, atualizações em breve!

Fique ligado, acesse o site:

http://linguagemprisma.net