==========
==========

Técnicas Alternativas de Programação:
Programação Orientada a Objetos
Prof. Alexandre Direne – Dep. de Informática – UFPR

==========
==========

Recomendação de Software e Bibliografia:

- Pacote de software para Programação Orientada a Objetos com linguagem Flavours - ambiente Poplog - obtido no seguinte endereço:
  http://www.cs.bham.ac.uk/research/poplog/freepoplog.html
- Programação Orientada a Objetos. Isaias Camilo Boratti. Editora Visual Books.
- Análise Baseada em Objetos. Peter Coad e Edward Yourdon. Editora Campus.
- Object-Oriented Languages, Systems and Applications. Gordon Balir et ali. Pitman Press.
- Fundamentals of Object-Oriented Design in UML. Larry Meilir Page-Jones. Addison-Wesley Object Technology Series.
- Design Patterns: Elements of Reusable Object-Oriented Software. Erich Gamma, et ali. Addison-Wesley Professional Computing.
- Breve introdução e exemplos:

  `http://media.wiley.com/assets/264/19/0764557599_bonus_AppA.pdf`

- Boa explicação e aprofundamento de princípios:

  `http://www.cs.vu.nl/~eliens/poosd/`
  `http://www.cs.vu.nl/~eliens/online/oo/contents.html`

- Um artigo clássico:

  `http://www.inf.ufpr.br/andrey/soft/docs/beckCunningham89.pdf`

1 INTRODUÇÃO

Em linhas gerais, o paradigma de "orientação a objetos" auxilia desenvolvedores a descrever disciplinadamente a solução de problemas complexos de especificação e de programação (e não a complexidade dos pontos de vista de domínio específico e esforço computacional). De forma mais detalhada, o termo complexo se refere à integração das seguintes tarefas de programação:

- gerência de tamanho (divisão de trabalho em responsabilidades diferentes);
- gerência de responsabilidades (evitar a interpenetração de atividades de programação);
- manutenção da consistência (garantir veracidade das relações lógicas entre os dados do programa, ao longo do tempo);
- garantia de completude do conteúdo de um programa para que tal conteúdo seja, na medida do possível, uma descrição autocontida tanto em "ponto grande" (especificação) quanto em "ponto pequeno" (o programa em si).

1.1 PRINCÍPIOS DE PROGRAMAÇÃO

1.1.1 O DADO COMO FOCO DE MODELAGEM

A visão tradicional do paradigma de programação imperativista e procedimental enfoca prioritariamente as operações (ou funções) e, em segundo plano, os dados que são manipulados (consultados e alterados) pelas operações. Nestas condições, o "ataque" indisciplinado e não padronizado aos dados pode provocar graves problemas de desenvolvimento.

Na visão da programação orientada a objetos, a idéia geral é a de deslocar o foco primário de atenção para o dado, de maneira que, em segundo plano, um conjunto padronizado de operações possa ser definido e subordinado ao dado. Dessa forma, objetos de dados podem "se comunicar" uns com os outros por meio de canais "estreitos" e muito precisamente definidos.

Diante de um princípio tão básico, podemos dizer que tanto a programação orientada por eventos quanto a programação orientada por agentes são variações da programação orientada a objetos.

1.1.2 ABSTRAÇÃO

Para que seja possível a descrição amplamente abstrata de programas, o paradigma de programação orientada a objetos oferece duas disciplinas principais: (a) modularização e (b) ocultação parcial de informação.

A modularização é uma disciplina que se preocupa com a fragmentação de um programa em partes (denominados módulos) com descrição auto-contida. Em termos computacionais, isto significa que um módulo pussui todas as estruturas de dados e algoritmos responsáveis por implementar, "independentemente", sua parte diante do sistema completo.

A ocultação parcial de informação eleva o nível de abstração um passo acima por se preocupar em esconder (parcial ou totalmente), dos usuários de um módulo, os detalhes de impelemtação do próprio módulo. Com a ocultação (parcial ou total) da informação, os usuários têm acesso a um objeto somente por meio de uma interface protegida (translúcida) de operações de comunicação. Sendo assim, ao usuário de um módulo, não é permitido o acesso a alguns detalhes internos de implementação de estruturas de dados e algoritmos.

1.1.3 POLIMORFISMO

O Polimorfismo, na verdade, pode ser enquadrado de duas grandes categorias diferentes:

- Como um meio para a "coerção" de tipos/classes aos objetos, o que promove a possibilidade de um objeto (por definição estática ou dinâmica) pertencer a mais de uma classe durante a sua existência. Nos casos especiais de linguagens que permitem a definição dinâmica da classe de um objeto, dizemos que há "amarração tardia" (dynamic binding ou late binding) de valor;

- Como um meio para o "compartilhamento de comportamento" de diferentes objetos, permitindo que as semelhanças de comportamento (comportamento comum) entre os referidos objetos sejam definidas segundo um mesmo nome de método através da "sobre-carga" de variações das definições de elementos de natureza operativas (exemplos: funcionalistas, eventualistas, serialistas).

1.1.4 DEFINIÇÃO EVOLUTIVA

Um sistema ou programa nunca se encontra em estado "final". Mesmo depois de uma primeira versão em execução, um programa sofre constantes alterações em sua especificação de requisitos. Adicionalmente, quando a velocidade de mudança é muito alta ou desconhecida, novas funcionalidades podem ter que ser inseridas no programa sem que este seja retirado de execução.

Dois aspectos de evolução devem ser considerados:

- Evolução de especificação de requisitos;
- Evolução da programação da solução.

1.1.5 CORREÇÃO POR CONSTRUÇÃO

Apesar de não ser um princípio original da programação orientada a objetos, a elaboração de código "correto por construção" tem assumido algumas faces exclusivas deste paradigma. Dentre as várias conotações ligadas ao conceito de "correção" que podem ser assumidas, aquelas mais especificamente ligadas à programação orientada a objetos são:

- Conformidade com teste sistemático: o programa tem sempre seu código sistematicamente (formalmente) testado durante a construção;
- Tolerância a falhas: o programa nunca é retirado da execução pela decorrência de qualquer operação realizada sobre seus objetos;
- Degradação progressiva: na ocorrência de imprevistos, o interpretador sempre garante que o resultado de algum processamento alternativo para qualquer operação realizada sobre seus objetos seja informado ao objeto requisitante (no mundo imperativo de processamento, os imprevistos provocam a ruptura brusca da linha de execução do código com a imediata devolu ção do controle ao Sistema Operacional).

1.1.6 REUTILIZAÇÃO DE CÓDIGO

Uma das conclusções mais alarmentes de estudos feitos por pesquisadores dos campos de Linguagens de Programação e Engenharia de Software aponta para um desperdício de mais de 70% do código constrído no mundo. Em outras palavras, 70% de todo o código programado é descartável. Sendo assim, um importante princípio é o da reutilização de código, por onde se pretende permitir a construção de programas reutilizáveis por sua própria natureza.

1.2 TÉCNICAS DE PROGRAMAÇÃO

1.2.1 ENCAPSULAMENTO DE DADOS E MÉTODOS

É a técnica que permite a aglomeração da definição dos dois elementos mais básicos da construção modular de uma classe ou objeto em si: dado e operação. Por meio desta técnica, é ainda possível "esconder", do mundo externo a um objeto, os detalhes de implementação tanto das estruturas de dodos quanto das operações definidas no referido objeto. As operação definidas em um objeto são também comumente chamadas de métodos.

1.2.2 ENVIO DE MENSAGENS

Objetos de dados podem "se comunicar" uns com os outros por meio de canais "estreitos" e muito precisamente definidos, conhecidos como o envio de mensagens. Esta forma de comunicação entre objetos permite uma série de interpretações dinâmicas, as quais podem ser resumidamente apresentadas como:

- O objeto-1 envia mensagem para o objeto-2;
- O objeto-2 recebe e interpreta a mensagem do objeto-1;
- O objeto-2 pode alterar seu estado interno de estruturas de dados;
- O objeto-2 pode devolver uma mensagem para o objeto-1 e também para outros;
- O objeto-1 recebe e interpreta a mensagem de reconhecimento do objeto-2;

1.2.3 HERANÇA SIMPLES E MÚLTIPLA

A técnica de programação por herança de código é uma das mais conhecidas e contempla os conceitos de classes, subclasses, e instâncias como objetos. Em seu estágio mais elementar, a herança simples permite que todas as subclasses ou instâncias de uma classe tenham acesso automático a todos os dados e métodos da referida classe.

Em uma definição hierárquica contendo um número indefinido de níveis de classes, subclasses e instâncias, esta técnica permite que um objeto qualquer da hierarquia tenha acesso automático a todos os dados e métodos de todas as classes de objetos aos quais ele está subordinado.

1.2.4 ORGANIZAÇÃO TAXONÔMICA E ESPECIALIZAÇÃO

As técnicas de organização taxonômica e especialização de código são talvez as maiores motivadoras de conceitos na programação orientada a objetos. A técnica de organização taxonômica do código permite o grupamento de objetos que compartilham comportamentos semelhantes. Este grupamento é chamado de classificação da informação, o qual também admite que uma classificação inclua outra. Por exemplo, uma "pilha" com comportamento "empilha" e "desempilha" pode ser incluída em uma "nova pilha" a qual tem comportamentos "empilha", "desempilha" e "imprime".

De forma análoga, a especialização permite a extensão de comportamentos genericamente semelhantes por meio de adaptações ou refinamentos sucessivos que ficam gravados explicitamente como parte integrante do código. Por exemplo, uma "fila" pode ser classificada como uma entidade que possui os comportamentos "insere", "elimina" e "imprime". Sendo assim, todas as sub-classes derivadas de "fila" terão, consequentemente, os mesmos nomes de comportamentos ("insere", "elimina" e "imprime"), definidos, porém, de formas mais e mais refinadas, estendendo ou substituindo integralmente o código dos comportamentos equivalentes nas super-classes.

1.2.5 PREENCHIMENTO DE VALOR AUSENTE

Mais comumente chamado de valor "default", o preenchimento de valor ausente é uma forma originalmente criada no âmbito dos primeiros passos da programação orientada a objetos. A idéia básica desta técnica é a de obtenção dinâmica de um valor, assumido na falta de outro mais especificamente dedicado a um objeto.

Em vários casos, a combinação da técnica de valor ausente com a técnica de herança oferece um poderoso mecanismo de reutilização de código. Por meio de tal mecanismo, é possível realizar a criação de grandes quantidades de objetos contendo, na descrição de cada um, apenas a definição explícita dos elementos de diferenciadores. A definição explícita dos elementos de semelhança fica localizada na programação da classe dos objetos.

2 A LINGUAGEM FLAVOURS

2.1 DEFINIÇÃO DE CLASSES

Na linguagem flavours, uma classe (ou flavour) tem a seguinte forma genérica:

```
uses flavours;
flavour <NOME> <COMPLEMENTOS>;
    <CORPO>
endflavour;
```

O corpo de uma classe é constituído de variáveis de instância e de métodos. As variáveis de instância são atributos que compõem o estado dos dados de um objeto. Os métodos são descrições de como receber e manipular mensagens enviadas ao objeto. A sessão de variáveis de instância é definida com a decclaração "ivars" ao passo que cada método é deninido com a declaração "defmethod". O exemplo abaixo mostra a definição de uma classe "pessoa" contendo 3 variáveis de instância e 2 métodos.

```
uses flavours;
flavour pessoa;
  ivars nome idade sexo;
    defmethod aniversario;
        idade + 1 -> idade;
        [ feliz aniversario para ^nome] =>
    enddefmethod;
    defmethod identifique_se;
        pr('<pessoa ');
        pr(nome);
        pr('>\n');
    enddefmethod;
endflavour;
```

2.2 CRIAÇÃO DE INSTÂNCIAS

A criação de instâncias de classes é feita por meio da ativação do procedimento "make_instance", o qual deve receber como parâmetro único, uma lista contendo seqências das seguintes informações combinadas: *<variável_de_instância> e seu <valor_inicial>*. O exemplo abaixo mostra o criação da instância "joao".

```
: vars joao;
: make_instance([pessoa nome joao idade 30 sexo masculino]) -> joao;
```

Observação: neste caso, o nome da instância ("joao") coincidiu com o valor inicial da variável de instância "nome".

Se a instância "joao" da classe "pessoa" for empilhada e, em seguida, impressa, obteremos o seguinte:

```
: joao =>
** <instance of pessoa>
```

2.3 ENVIO DE MENSAGENS

O envio de mensagens é feito da seguinte forma geral:

```
<INSTANCIA>  <-  <SELETOR>(<ARG_1>, <ARG_2>, ... <ARG_n>);
```

A mensagem é estruturada com base no conceito de seleção do foco de comunicação. O "seletor" de foco é sempre associado a uma veriável de instância a ser simplesmente consultada (de forma padrão) ou alterada e/ou consultada (por meio de métodos definidos na classe).

Em vários casos, não há argumentos a serem associados ao seletor.

No exemplo abaixo, várias mensagens são enviadas à instância "joao", nas quais os seletores coincidem propositalmente com os nomes das variáveis de instância "idade", "sexo" e "nome" (consulta padrão).

```
: joao <- idade =>
** 30
: joao <- sexo =>
** masculino
: joao <- nome =>
** joao
```

Todavia, também podemos utilizar métodos como seletores de foco para consulta (especificada) e alteração do estado de dados de uma instância (veja o exemplo abaixo).

```
: joao <- aniversario;
** [feliz aniversario para joao]
: joao <- idade =>
** 31
```

Observação: note que a exploão de elementos globais de memória é bem mais sentida na programação orientada a objetos do que em outros paradigmas.

2.4 VALORAÇÃO DEFAULT E AUTO-REFERÊNCIA

Em um ambiente incremental, novas variáveis de instância e novos métodos podem ser adicionados (ou re-definidos) durante a execução do programa. Sendo assim, qualquer instância já criada para a classes, adquire automaticamente as novas variáveis e os novos métodos (ou suas re-definições).

No exemplo abaixo, uma nova veriável de instância com nome "conjuge" é definida, a qual tem valor inicial "false". Isto fará com que todas as instâncias da classe "pessoa" já criadas adquiram a nova variável, já acompanhada de seu valor inicial. Adicionalmente, um novo método "casa" é definido, o qual inclui um argumento.

Obsercação: note a presença da variável especial "self", a qual, no momento do envio de uma mensagem, ganha como valor a instância que recebe a mensagem.

```
flavour pessoa; ;;; Incremente *dinamicamente* a definicao da classe pessoa.
   ivars conjuge = false; ;;; Nova variavel com valor *inicial* "false".
   defmethod casa(pessoa);
     lvars pessoa; ;;; note a privacidade do mnemonico "pessoa"
       if pessoa<-sexo == sexo then  ;;; Somos do mesmo sexo ?
           'Casamento modernissimo!' =>
       endif;
       if conjuge then ;;; Ja estou casado ... ?
           unless conjuge == pessoa do ;;; ... com outra pessoa ?
              mishap('BIGAMIA !', [% self, conjuge, pessoa %]);
           endunless
       else
           ;;; Alterar o valor da variavel de instancia "conjuge".
           pessoa -> conjuge;
           ;;; Expressar votos de fidelidade.
           if pessoa<-sexo = "masculino" then
              [Eu , ^nome , aceito ^(conjuge<-nome) como legtimo esposo] =>
           else
              [Eu , ^nome , aceito ^(conjuge<-nome) como legtima esposa] =>
           endif;
           ;;; Forcar a relacao biunivoca nas variaveis da outra instancia.
           conjuge <- casa(self);
       endif;
   enddefmethod;
endflavour;
```

Agora então podemos executar os seguintes passos:

```
: joao <- conjuge=>
** <false>
: vars maria;
: make_instance([pessoa nome maria idade 34 sexo feminino]) -> maria;
: maria <- conjuge=>
** <false>
: joao <- casa(maria);
** [Eu , joao , aceito maria como legtima esposa]
** [Eu , maria , aceito joao como legtimo esposo]
: joao <- conjuge <- identifique_se;
<pessoa maria>
: maria <- conjuge <- identifique_se;
<pessoa joao>
: joao <- conjuge <- conjuge  ==  joao  =>
** <true>
: joao <- casa(maria);
: joao <- casa(maria);
: joao <- casa(maria);
: joao <- casa(maria);
: vars joana;
: make_instance([pessoa nome joana idade 37 sexo feminino]) -> joana;
: joana <- casa(joao);
** [Eu , joana , aceito joao como legtimo esposo]

;;; MISHAP - BIGAMIA
;;; INVOLVING: <instance of pessoa> <instance of pessoa> <instance of pessoa>
;;; DOING :  pessoa<-casa pessoa<-casa pop_setpop_compiler runproc runproc
```

Cuidado! A repetição do envio de uma mesma mensagem pode não provocar o mesmo efeito. Você pode imaginar por que ? Por exemplo, no caso abaixo, pode ser que mais um método ("suspende_casamento") tenha que ser definido para evitar a situação.

```
: joana <- casa(joao);
:
```

2.5 HERANÇA SIMPLES

A herança na linguagem flavours é definida por meio da declaração "isa". Ela permite que uma sub-classe ou super-classe de outra classe seja especificada, permitindo assim a visibilidade automática, por parte de uma sub-classe, das variáveis de instância e dos métodos definidos na classe.

No exemplo abaixo, "professor" é definida como sub-classe de "pessoa". Sendo assim, instâncias da classe "professor" possuirão automaticamente também as variáveis "nome", "idade", "sexo" e "conjuge". Além destas, foram adicionadas 2 variáveis específicas (exclusivas) da classe professor: "matricula" e "area_de_interesse". Sendo assim, uma instância de professor, além de responder a todas as mensagens que uma instância de pessoa é capaz de responder, ainda responderá às mensagens "leciona" e "identifique_se".

```
flavour professor isa pessoa;
  ivars matricula area_de_interesse;
    defmethod leciona(assunto);
       [^nome da aula de ^assunto] =>
    enddefmethod;
    defmethod identifique_se;
```

```
        pr('<professor '); pr(nome); pr('>\n');
    enddefmethod;
endflavour;
```

Observação: note que, aparentemente, uma instância de professor tem agora 2 (dois) métodos para manipular a mensagem "identifique_se". Todavia, se a mensagem "identifique_se" for enviada para uma instância de professor, o método mais específico será utilizado para manipular a mensagem (aquele definido dentro da classe "professor"). Alguns exemplos de uso de herança simples são:

```
: vars alex;
: make_instance([professor nome alex sexo masculino idade 26
                 area_de_interesse programacao_oo ]) -> alex;
: alex =>
** <instance of professor>
: alex <- identifique_se;
<professor alex>
: alex <- area_de_interesse =>
** programacao_oo
: alex <- aniversario;
** [feliz aniversario para alex]
: alex <- idade =>
** 27
: alex <- leciona("tecnicas_alternativas");
** [alex da aula de tecnicas_alternativas]
```

2.6 HERANÇA MÚLTIPLA

Vários sistemas de programação orientada a objetos permitem que uma classe herde veriáveis e métodos de mais de uma classe. Esta técnica é chamada de herança múltipla e também é especificada com a declaração "isa".

No exemplo abaixo, uma nova classe, a de "professor_ingles", é definida como sub-classe de "professor", a qual, por sua vez, é também uma sub-classe de "pessoa".

```
flavour professor_ingles isa professor;
   ...
   ...
endflavour;
```

Uma forma alternativa de definir "professor_ingles" por meio de herança múltipla seria por meio da criação de uma classe "pessoa_inglesa" e, em seguida, definindo "professor_ingles" como sub-classe tanto de "pessoa_inglesa" quanto de "professor". Isto poderia ser atingido da seguinte forma:

```
flavour professor_ingles isa pessoa_inglesa professor;
   ...
   ...
endflavour;
```

3 LINGUAGEM DE PROGRAMAÇÃO DE MÉTODOS

3.1 PRINCIPAIS TIPOS BÁSICOS DA LINGUAGEM

3.1.1 NÚMEROS (numbers):

- Inteiros: 66 99876789 -66
- Decimais: 88.532 1.2345e3 (=1234.5) 123.4e-3 (=0.1234)

- Variações destes

```
: 345 ** 297 ==>
53984767681040038584873836603429526974878733278426757450775175466632528372331796
63606465753622036752342902247211010308865420323355605646516428285989835098494306
06052099586370643520125361828970878814013195546530880886143930005782691336152906
29930060194680082540031107496419360065715705865149146741430987702672250554620706
91120263368405458835821786280549767076054676672663368283581696023346061619596706
60851748132959079548351283517355013955315905717802137052423402870619063562319206
03871685258815112855331390621958532419534258019169181653039722976609989202513306
72451875126668862820684750039296020217067324941287964548685317264347576968283506
99034509726022954865123391113282788337622622672889996251107513410746819945729506
73290949757785606166748948453459888696670532226562 5
: isnumber( 'quatro' )  =>
** <false>
: isnumber(32.85)  =>
** <true>
```

REFERÊNCIAS:
teach teach, teach vedpop, teach lmr, teach xved, teach moreved teach numbers arith - help math

3.1.2 CADEIAS DE CARACTERES (strings):

```
Exemplos:   'gato'   'noventa'   '66'   'um string com espacos ++**@@@@'

: isstring(1234)  =>
** <false>
: isstring('abc')  =>
** <true>
: 'linguagem' >< ' ' >< 'de programacao' =>
** linguagem de programacao
: consstring(97, 98, 99, 3)  =>
** abc
: explode('abc')  =>
** 97 98 99
```

3.1.3 SÍMBOLOS OU NOMES (words):

```
Exemplos:    "gato"   "noventa"   "roupa_velha"   "+"   "+**+"

: isword(91)  =>
** false
: isword("begin")  =>
** true
: consword('noventa')  =>
** noventa
: "ana" <> "maria" =>
** anamaria
```

REFERÊNCIAS:
help numbers, help strings, help words

3.1.4 VETORES UNIDIMENSIONAIS (vectors):

Um vetor é uma estrutura de dados compostas de um número arbitrário de elementos, alocados em áreas contiguas de memória. O exemplo abaixo mostra como criar um vetor de 4 elementos:

```
    {3  gato  'curso de Pop-11'   99 }
```

Para se ter acesso a um determinado de um vetor tanto para consulta quanto para alteração, um índice numérico deve ser utilizado, de acordo com os exemplos abaixo:

```
: {um gato} -> vetor;
: vetor(2) =>
** gato
: subscrv(2,vetor) =>
** gato
: "rato" -> subscrv(1, vetor);
: vetor =>
** {rato gato}
```

Vetores, assim como words, striings e outros objetos, podem ser concatenados da seguinte forma:

```
: {o curso}  <>  {de pop esta emocionante} =>
** {o curso de pop esta emocionante}
```

REFERÊNCIAS:
help vectors, help consvector, help subscrv, help vectorclass

3.1.5 VETORES MULTIDIMENSIONAIS OU MATRIZES (arrays):

Um array é uma estrutura multidimensional composta de:

- Um vetor contendo os dados;
- Uma lista de limites inferior e superior para indexar cada dimensao;
- Um procedimento de acesso tanto para consulta como para alteração, orientados por índices numéricos.

Para se criar um array, deve ser utilizado o procedimento "newarray" da seguinte forma:

```
newarray(<LISTA-DE-LIMITES>, <VALOR-INICIAL>) -> <NOME-MATRIZ> ;
```

Por exemplo:

```
: newarray([1 10 1 10], 0) -> matriz;
: matriz =>
** <array [1 10 1 10]>
```

Em modo de consulta, temos por exemplo:

```
: matriz(3,9) =>
** 0
```

Em modo de alteração, temos por exemplo:

```
: 1 -> matriz(3,9);
```

Se o acesso tentar utilizar um índice inválido, teremos:

```
: matriz(3,29) =>
;;; MISHAP - INVALID ARRAY SUBSCRIPT
;;; INVOLVING:  29
;;; DOING    :  compile nextitem popval compile
```

Caso seja necessário criar um array com valores iniciais diferentes para cada elemento, a seguinte forma do "newarray" pode ser usada:

```
newarray(<LISTA-DE-LIMITES>, <PROCEDIMENTO>) -> <NOME-MATRIZ> ;
```

Neste caso, $< LISTA-DE-LIMITES >$ tem 2N elementos e $< PROCEDIMENTO >$, quando é ativado, recebe N inteiros como parametros, os quais são resultantes de cada elemento do produto cartesiano dos índices. A quantidade de vezes que $< PROCEDIMENTO >$ é ativado é igual ao número total de elementos do array. Para cada ativação, o valor inicial do referido elemento é calculado com base nos valores dos índices do elemento. Sendo assim, o valor de $< NOME-MATRIZ > (i1, i2, ..., iN)$ será calculado por uma ativação automática da forma: $< PROCEDIMENTO > (i1, i2, ..., iN)$ onde $i1$ e' um valor dentro dos limites numéricos da dimensão 1, valendo o mesmo para $i2, ..., iN$.

REFERÊNCIAS:
teach arrays, help newarray

3.1.6 LISTAS DINÂMICAS (lists):

Listas dinamicas são estruturas muito flexíveis e que contam com um grande número de procedimentos de manipulação, constituindo uma poderosa biblioteca para a implementação de algoritmos diversos. O exemplo abaixo mostra como criar uma lista de 4 elementos:

```
[ 333  'curso maravilhoso'  [a b c ]   99 ]
```

Listas podem ser facilmente manipuladas, por exemplo, para a inversão de seus elementos:

```
: rev([1 2 3 4]) =>
** [4 3 2 1]
```

O último elemento de uma lista é obtido com o procedimento "last":

```
: last([gato cachorro cobra]) =>
** cobra
: last([ [marcus vinicius] [luiz alberto] [ana maria] ]) =>
** [ana maria]
```

Ou o primeiro elemento:

```
: hd([1 2 3 4]) =>
** 1
: hd(rev([gato cachorro cobra])) =>
** cobra
: hd(last([ [marcus vinicius] [luiz alberto] [ana maria] ])) =>
** ana
```

Porém, se uma lista vazia for processada, o seguinte ocorre:

```
: hd([]) =>

;;; MISHAP - NON-EMPTY LIST NEEDED
;;; INVOLVING:  []
;;; DOING    :  hd pop_setpop_compiler runproc runproc
```

O procedimento "tl" recebe uma lista e produz uma nova lista contendo todos os elementos da lista de entrada, com excessão do primeiro.

```
: tl([a b c d e]) =>
** [b c d e]
: tl([a]) =>
** []
: hd(tl([a b])) =>
** b
```

Da mesma forma que ocorre com “hd”, temos:

```
: tl([]) =>

;;; MISHAP - NON-EMPTY LIST NEEDED
;;; INVOLVING:  []
;;; DOING    :  tl pop_setpop_compiler runproc runproc
```

Pertinência de elementos a uma lista:

```
member(2, [1 5 4 6 2 3 7 9]) =>
** <true>
member([3 4 5], [1 2 [3 4 5] 6 7]) =>
** <true>
member(4, [1 2 [3 4 5] 6 7]) =>
** <false>
```

Escolha aleatória de elementos em uma lista:

```
 repeat 5 times
   oneof([
           [o curso de pop esta maravilhoso]
           [o dia esta lindo]
           [alguns alunos gostam de programar]
           [outros nao]
           [espero ansioso pela aula pratica]
       ]) =>
 endrepeat;

** [alguns alunos gostam de programar]
** [o curso de pop esta maravilhoso]
** [alguns alunos gostam de programar]
** [alguns alunos gostam de programar]
** [outros nao]
```

Eliminação total ou parcial de um dado elemento de uma lista:

```
: delete(3, [1 2 3 4]) =>
** [1 2 4]
```

Para a eliminação de um símbolo (“word”) de uma lista, as aspas duplas devem ser aplicadas no primeiro argumento do procedimento “delete”.

```
: delete("aula", [A aula esta animada]) =>
** [A esta animada]
: delete(3, [3 3 3]) =>
** []
```

Para eliminacao parcial de um elemento repetido:

```
: delete(3, [3 3 3], 1) =>
** [3 3]
: delete(3, [3 3 3], 2) =>
** [3]
```

Considere o seguinte comportamento:

```
: vars l, l1;
: [ a b c ] -> l;
: l -> l1;
: 93 -> l1(2);
: l1 =>
** [a 93 c]
: l =>
** [a 93 c]
```

Listas são compostas de pares. Um par é uma estrutura com os componentes <FRONT> e <BACK>, conforme a ilustração abaixo:

```
----------------
| FRONT | BACK |
----------------
```

Uma lista de 3 elementos é constituida de 3 pares encadeados:

```
 _____ variavel l
 |  _____ variavel l1
 v  |
____v___________        _________________        _________________
|   a   |   ----------> |  93   |    ---------->|   c   |    ------>/
----------------        -----------------        -----------------
```

Logo, ambas as variáveis "l" e "l1" apontam para o mesmo local de memória.
Cuidado!!! A concatenação de listas pode produzir efeitos indesejáveis. Por exemplo:

```
: [ a b c ] -> l1;
: [ d e f ] -> l2;
: l1 <> l2  -> l3;
```

Aém das operações acima, ainda temos:

```
: 3 :: [4 5 6] =>
** [3 4 5 6]
: "CAT" :: []  =>
** [CAT]
: "CAT" :: "DOG" =>

;;; MISHAP - LIST NEEDED
;;; INVOLVING:  DOG
;;; DOING    :  :: pop_setpop_compiler runproc runproc
```

Quando da intercalação ou composição de listas, cuidados devem ser tomados com a diferença entre inserção e cópia dos elementos um a um. Veja o exemplo abaixo:

```
vars lista;
[ b c d ] -> lista;
[a ^lista e ] =>
** [a [b c d] e]
[a ^^lista e ] =>
** [a b c d e]
```

Além disso, os seguintes comandos são equivalentes:

```
: [^x] <> l -> l;
: x :: l -> l;
: [^x ^^l] -> l;
```

.

Também são equivalentes os seguintes comandos:

```
: [ ^X + ^Y is ^(X + Y) ] -> l;
: [% X, "+", Y, "is", X + Y %] ->l;
: [ %X% + %Y% is %X + Y% ] -> l;
```

Finalmente, é importante entender alguns detalhes das expressões abaixo:

```
[ % 3, aaa,  hd(l) % ]
[ 3, aaa, hd(l) ]
[ %  aaa, hd(l)% ]
[% [% a %], [% b, c%], [% d %] %]
```

REFERÊNCIAS:
teach lists, teach listsummary, help morelists help destlist, dest, dl, explode, applist, maplist, sort, ::, <>

3.1.7 REGISTROS (defclass):

Classes de registros são definidas para que seja possível alocar estruturas heterogêneas na memória de forma orientada por atributos mnemônicos. Listas e vetores também admitem valoração heterogênea porém, o acesso indexado é puramente numérico.

No exemplo abaixo, a classe de registros "reg_aluno" é definida com atributos "matric_aluno", "nome_aluno", "ira_aluno". Em seguida, o objeto registro "alexd" é definido.

```
: defclass reg_aluno {matric_aluno, nome_aluno, ira_aluno};
: consreg_aluno(981334, 'Alexandre Direne', 7.46) -> alexd;
```

3.2 PRINCIPAIS COMANDOS BÁSICOS DA LINGUAGEM

3.2.1 ATRIBUIÇÃO DESTRUTIVA E NÃO DESTRUTIVA

A atribuição, expressa por "− >" e uma operação que remove um item da pilha do programador, e o coloca em outra posição de memória. Por exemplo, coloca-se quatro itens na pilha da seguinte forma:

```
: 9, 'gato', 99, "azul";
```

A seta de atribuição pode ser usada para remover o topo da pilha e armazena-lo como o valor da variável "x" da seguinte forma:

```
:  -> x;
```

Logo, 3 itens ainda permanecem na pilha.

```
: stacklength()  ==>
** 3
```

Agora, se exibirmos o que restou na pilha teremos:

```
: =>
** 9 gato 99
```

Além disso, o quarto item colocado na pilha agora é o valor de "x".

```
: x =>
 ** azul
```

O comando de atribuição utilizando o simbolo "− >>" não provoca a remoção de nenhum item da pilha.

```
: 2  ->> x  ->> y  ->> z  -> w;
```

REFERÊNCIAS:
teach arrow

3.2.2 DECISÃO SIMPLES, DUPLA E MÚLTIPLA

A forma geral da decisão simples é:

```
if  <CONDICAO>  then
  <COMANDO 1> ;
    . . .
  <COMANDO n> ;
endif;
```

A forma geral da decisão dupla é:

```
if  <CONDICAO>  then
  <COMANDO 1> ;
    . . .
  <COMANDO n> ;
else
  <COMANDO 1> ;
    . . .
  <COMANDO m> ;
endif;
```

A forma geral da decisão múltipla é:

```
if  <CONDICAO 1>  then
  <COMANDO 1> ;
    . . .
  <COMANDO n> ;
elseif  <CONDICAO 2> then
  <COMANDO 1> ;
    . . .
  <COMANDO m> ;
elseif  <CONDICAO 3> then  . . .
elseif  . . .
elseif  . . .
elseunless <CONDICAO k> then
  <COMANDO 1> ;
    . . .
  <COMANDO i> ;
else
  <COMANDO 1> ;
    . . .
  <COMANDO j> ;
endif;
```

Ou ainda, a forma geral da decisão por negação é:

```
unless  <CONDICAO>  then
  <COMANDO 1> ;
    . . .
  <COMANDO n> ;
endunless;
```

Esta é equivalente a:

```
if not(<CONDICAO>) then ... endif
```

Seja o exemplo abaixo:

```
if    x > y
then  x =>
else  y =>
endif;
```

Compare com:

```
if       x > y
then     x =>
elseif   y > x
then     y =>
else     "iguais" =>
endif;
```

Considere agora o exemplo abaixo:

```
if    2 < x and x < 6
then  true ;
else  false ;
endif  =>
```

Note que o exemplo acima é equivalente a:

```
2 < x and x < 6  =>
```

REFERÊNCIAS:
help if, help switchon

3.2.3 REPETIÇÃO COM FORMAS VARIADAS

Forma geral:

```
until <CONDICAO> do <COMANDOS>  enduntil;
```

Exemplo:

```
3 -> num;
until num > 99
do    num =>
  num + 1 -> num;
enduntil;
```

Forma geral:

```
repeat <NUMERO> times <COMANDOS>  endrepeat;
```

Exemplo:

```
repeat 10 times
    pr(newline)
endrepeat;
```

Forma geral:

```
repeat <COMANDOS>  endrepeat;
```

Observação: repetição infinita (a para deve utilizar quitif).
Forma geral:

```
while <CONDICAO> do <COMANDOS>  endwhile
```

Exemplo:

```
1 -> num;
while num * num < 1000 do
    num + 1 -> num
endwhile;
```

Forma geral:

```
for <ATRIBUICAO> step <INCREMENTO> till <CONDICAO> do
     <COMANDOS>
endfor;
```

Exemplo:

```
for 1->x
step x+2 -> x
till  x > 42
do    spr(x);
endfor;
```

```
: 1 3 5 7 9 11 13 15 17 19 21 23 25 27 29 31 33 35 37 39 41
```

Forma geral:

```
for <ITEM>  in <LISTA>  do  <COMANDOS>  endfor;
```

Exemplo:

```
[1 2 3 4 5] -> numbers;
for num in numbers do
    pr('\n O quadrado de: '); pr(num); pr(' e: '); pr(num*num);
endfor;
```

```
: O quadrado de: 1 e: 1
O quadrado de: 2 e: 4
O quadrado de: 3 e: 9
O quadrado de: 4 e: 16
O quadrado de: 5 e: 25
```

Forma geral:

```
for <SUBLISTA> on <LISTA> do <COMANDOS>  endfor;
```

Exemplo:

```
for L on [a b c] do L => endfor;
    ** [a b c]
    ** [b c]
    ** [c]
```

Forma geral:

```
  for <VARIAVEL> from <INICIAL> by <INCREMENTO> to <FINAL> do
    <COMANDOS>
  endfor;
```

Exemplo:

```
: for x from 2 by 7 to 20 do  x => endfor;
** 2
** 9
** 16
```

Ou ainda, utilizado em ordem decrescente. Exemplo:

```
: for x from 50 by -12.5 to -20 do x => endfor;
** 50
** 37.5
** 25.0
** 12.5
** 0.0
** -12.5
```

Utilizando valores default, temos o seguinte exemplo:

```
: vars x;
: for x to 6 do pr(x); pr(space) endfor;
1 2 3 4 5 6
```

REFERÊNCIAS:
help loops

3.3 INTERPRETAÇÃO DE MENSAGENS DE ERRO

O compilador Pop-11 gera código para enviar mensagens de erro baseado no procedimento "MISHAP". Por exemplo, se o sinal de + for esquecido no comando abaixo teremos um erro sintático:

```
: 6   6  =>
;;; MISHAP - MSEP: MISSING SEPARATOR (eg semicolon)
;;; INVOLVING: 6 6
;;; DOING    : compile
```

O código listado antes do ":" na linha superior da mensagem é a chave para se obter mais ajuda ("MSEP").

```
: help msep

     ... OU MESMO  ...

: explain msep  ;;; Cuidado: comando do pop11 (resultado no standard I/O)
```

Quando o erro é semântico, este receberá tratamento semelhante ao sintático mas com a exibição da mensagem somente em tempo de execução do comando (e não de compilação). Por exemplo, considere a compilação correta do procedimento abaixo:

```
define teste;
  "um" + "dois" =>
enddefine;
```

A atição do mesmo prodizirá:

```
: teste();
;;; MISHAP - NUMBER(S) NEEDED
;;; INVOLVING:  um  dois
;;; DOING    :  + teste pop_setpop_compiler runproc runproc
```

REFERÊNCIAS:
help mishap

3.4 MODULARIZAÇÃO DE MÉTODOS E VISIBILIDADE DE NOMES

3.4.1 DECLARAÇÃO E ESCOPO DE VARIÁVEIS E CONSTANTES

Variáveis e constantes são identificadores os quais podem assumir escopo DINÂMICO ou LÉXICO.

Todo identificador em Pop-11 é local ao seu escopo, a não ser que este seja explicitamente declarado como GLOBAL.

Observações Importantes:

- os identificadores de escopo dinâmico são permanentes durante toda a execução do programa e podem ser consultados ou alterados em qualquer local deste programa (a não ser que tenham sido destruidos explicitamente);
- os de escopo léxico são temporários, só podendo ser consultados ou alterados dentro da unidade de programa (ex. um procedimento) onde foram declarados.

Forma geral para declaração:

```
vars      <VARIAVEL PERMANENTE>
lvars     <VARIAVEL TEMPORARIA>
constant  <CONSTANTE PERMANENTE>
lconstant <CONSTANTE TEMPORARIA>
```

REFERÊNCIAS:
help vars, help constant, help global

Exemplos:

```
define teste11;
 vars a11;
   1 -> a11;
enddefine;

define teste12;
 lvars a12;
   2 -> a12;
enddefine;

: isdeclared("a11") =>
** <ident <undef a11>>
```

```
: isdeclared("a12") =>
** <false>
: a11 =>
** <undef a11>
: a12 =>
;;; DECLARING VARIABLE a12
** <undef a12>
```

Observações adicionais:

- é importante notar que todas as variáveis, tanto as passadas como parâmetros com as produzidas como resultado, declaradas dentro do procedimento utilizando VARS e LVARS, serão locais ao procedimento. Portanto, em um procedimento recursivo, para cada ativação do mesmo, teremos posições de memória distintas que "nao interferem" uma na outra;
- Existem ainda variáveis chamadas ativas que são na verdade "funções automaticamente executadas toda vez que o valor da variável for consultado ou alterado.

3.4.2 DEFINIÇÃO DE PROCEDIMENTOS E FUNÇÕES

A declaração de procedimentos e funções como unidades de um programa é feita da seguinte forma:

```
define <CABECALHO>;
  <DECLARACOES>
  <COMANDOS>
enddefine;
```

Quando for o caso, o "ATUALIZADOR" do procedimento ou função também pode ser declarado de forma semelhante:

```
define updaterof  <CABECALHO>;
  <DECLARACOES >
  <COMANDOS>
enddefine;
```

De forma mais específica, o procedimento abaixo define um identificador "proc" de 3 argumentos e 2 resultados. O nome "proc" é global e o corpo do procedimento e' associado a "proc" como sendo seu valor.

```
define proc(a,b,c) ->d ->e;
  < ... >
  < ... >
enddfine;
```

Adicionalmente, cinco variáveis LOCAIS, 3 de entrada e 2 de saída, são declaradas implicitamente e serão, por default, de carater PERMANENTE (não são de escopo léxico). Caso seja requisito de programação que as variáveis (algumas ou todas) sejam de carater TEMPORÁRIO, a declaração explícita das mesmas se faz necessário. A forma genérica para isso é a seguinte:

```
define proc(a,b,c) ->d ->e;
  lvars a,b,c,d,e;
  <sequencia de comandos>
enddefine;
```

Em seguida estão alguns exemplos básicos de modularização de comandos explicitando as questões de VISIBILIDADE (escopo) e VALIDADE (tempo) de variáveis.

```
define aaa;
 vars x1;
   5 -> x1;
enddefine;

define bbb;
 lvars x2;
   6 -> x2;
   x1 =>
enddefine;

define ccc;
  x2 =>
enddefine;

;;; DECLARING VARIABLE x2
: aaa();
:
: bbb();
** <undef x1>
:
: ccc();
** <undef x2>
```

Mudando-se a forma de ativação de “bbb”, verifa-se também uma alteração na consulta ao valor da variável “x1”.

```
define aaa;
 vars x1;
   5 -> x1;
   ;;; Ativando-se bbb dentro de aaa
   bbb();
enddefine;

define bbb;
 lvars x2;
   6 -> x2;
   x1 =>
enddefine;

define ccc;
   x2 =>
enddefine;

;;; DECLARING VARIABLE bbb
;;; DECLARING VARIABLE x2

: aaa();
** 5
: bbb();
** <undef x1>
: ccc();
** <undef x2>
```

Porém, quando um procedimento é definido dentro do escopo de eoutro, o próprio acesso ao procedimento embutido se torna restrito. Por exemplo:

```
define aaa;
vars x1;

  define bbb;
   lvars x2;
     6 -> x2;
     x1 =>
  enddefine;

  5 -> x1;
  bbb();
enddefine;

define ccc;
   x2 =>
enddefine;

;;; DECLARING VARIABLE x2

: aaa();
** 5
: bbb();

;;; MISHAP - enp: EXECUTING NON-PROCEDURE
;;; INVOLVING:  <undef bbb>
;;; DOING    :  compile pop_setpop_compiler runproc runproc compile

: ccc();
** <undef x2>
```

REFERÊNCIAS:
help define, help updater, help procedure

Porém, CUIDADO com listas dinâmicas !!!

```
define aaa;
 vars l1;
   [a b c ] -> l1;
   ccc(l1);
   l1 =>
enddefine;

define ccc(l2);
 vars l2 ;
   93 -> l2(2);
enddefine;

: aaa();
** [a 93 c]
```

Ou mesmo declarando-se as variaveis com "lvars"

```
define aaa;
 lvars l1;
   [a b c ] -> l1;
   ccc(l1);
```

```
   l1 =>
enddefine;

define ccc(l2);
 lvars l2 ;
   93 -> l2(2)
enddefine;

: aaa();
** [a 93 c]
```

ATENÇÃO: procedimentos também podem ser "concatenados"!!! Veja o exemplo abaixo:

```
: sqrt(4);
: pr();
2.0
: vars procedure prsqrt;
: sqrt <> pr -> prsqrt;
: prsqrt(4);
2.0
```

3.4.3 CONSTRUÇÃO DE PROCEDIMENTOS RECURSIVOS

Em sua forma recursiva tradicional, a versão recursiva do cálculo do fatorial de N em Pop-11 pode ser dado pelo seguinte:

```
define fatorial(n) -> resposta;
  if n = 0 then
     1 -> resposta
  else
     n * fatorial(n-1) -> resposta
  endif;
enddefine;
```

Quando não só o procedimento é recursivo mas também a natureza do dado a ser manipulado pelo procedimento, podemos identificar aspectos de semelhança entre diversas soluções. Os exemplos que seguem são aplicados a listas dinâmicas.

Exemplo: construir um procedimento recursivo em Pop-11 que seja capaz de contar o número de elementos ("conta_elem") de uma lista dada. O comportamento do procedimento é expresso abaixo.

```
: conta_elem([]) =>
** 0
: conta_elem([1 3 5 7 9 ]) =>
** 5
: conta_elem([ 195  'hoje e quarta'  paulo [tribunal da historia] ]) =>
** 4
```

O seu conteúdo do procedimento pode ser definido sa seguinte forma:

```
define conta_elem(lista);
   if lista == [] then
      0;
   else
      1 +  conta_elem(tl(lista));
   endif;
enddefine;
```

Como o procedimento tem característica funcional, ele também poderia ser definido segundo a forma abaixo:

```
define conta_elem(lista) -> num;
   if lista == [] then
      0 -> num;
   else
      1 +  conta_elem(tl(lista)) -> num;
   endif;
enddefine;
```

Exemplo: construir um procedimento recursivo em Pop-11 denominado "conta_ocorrencias", cujo comportamento é o expresso abaixo:

```
: conta_ocorrencias("a", []) =>
** 0
: conta_ocorrencias("a", [a]) =>
** 1
: conta_ocorrencias("a", [a menina]) =>
** 1
: conta_ocorrencias("a", [a menina viu a uva]) =>
** 2
: conta_ocorrencias("menina", [o menino espantou o gato]) =>
** 0
```

O seu conteúdo do procedimento pode ser definido sa seguinte forma:

```
define conta_ocorrencias(elem, lista) -> num;
   if lista == [] then
      0 -> num;
   elseif elem = hd(lista) then
      1 + conta_ocorrencias(elem, tl(lista)) -> num;
   else
      conta_ocorrencias(elem, tl(lista)) -> num;
   endif;
enddefine;
```

Exemplo: construir um procedimento em Pop-11, denominado "listifica", cujo comportamento é o expresso abaixo:

```
: listifica([]) =>
** []
: listifica([a]) =>
** [[a]]
: listifica([a b c d]) =>
** [[a] [b] [c] [d]]
```

O seu conteúdo do procedimento pode ser definido sa seguinte forma:

```
define listifica(lista) -> resultado;
  if  lista = [] then
     [] -> resultado
  else
     [ [^(hd(lista))]  ^^(listifica(tl(lista))) ] -> resultado
  endif
enddefine;
```

Exemplo: construir um procedimento em Pop-11, denominado "lineariza", o qual recebe como entrada uma lista de objetos (palavras e inclusive outras listas de qualquer profundidade) e devolve uma lista de profundidade 1, contendo apenas as palavras inseridas nos diversos pontos da lista de entrada. Seu comportamento é o expresso abaixo:

```
: lineariza( [a b c ] ) =>
** [a b c]
: lineariza( [[a] [b] [c] ] ) =>
** [a b c]
: lineariza( [[a] [b  [w q [u ] ] ] [c [f [t o]  g]] ] ) =>
** [a b w q u c f t o g]
```

O seu conteúdo do procedimento pode ser definido sa seguinte forma:

```
define lineariza(lista) -> resultado;
  if  lista = []
    then [] -> resultado
  elseif islist(hd(lista)) then
     [ ^^(lineariza(hd(lista))) ^^(lineariza(tl(lista))) ] -> resultado
  else
     [ ^(hd(lista)) ^^(lineariza(tl(lista))) ] -> resultado
  endif
enddefine;
```

Exemplo: construir um procedimento em Pop-11, denominado "todas_antes", o qual relaciona uma lista de palavras com uma sub-lista de todas as palavras da lista que precedem uma dada palavra. Seu comportamento é o expresso abaixo:

```
: todas_antes("casaco", []) =>
** []
: todas_antes("casaco", [casaco]) =>
** []
: todas_antes("casaco", [casaco de pele]) =>
** []
: todas_antes("abriu", [o homem abriu a porta]) =>
** [o homem]
: todas_antes("casaco", [o homem abriu a porta]) =>
** []
```

O seu conteúdo do procedimento pode ser definido sa seguinte forma:

```
define todas_antes(palavra,lista) -> resultado;
  if  lista = []
    then [] -> resultado
  elseif  not(member(palavra, lista)) then
     [] -> resultado
  elseif  palavra = hd(lista) then
     [] -> resultado
  else
     [ ^(hd(lista)) ^^(todas_antes(palavra, tl(lista)))] -> resultado
  endif
enddefine;
```

É dado o procedimento "surpresa" com o conteúdo abaixo:

```
define surpresa(item,lista) -> resultado;
  if  lista = []
  then    [] -> resultado
  elseif  item = hd(lista)
  then    tl(lista) -> resultado
  else    [ ^(hd(lista)) ^^(surpresa(item,tl(lista)))]
          -> resultado
  endif
enddefine;
```

Qual o seu comportamento para cada uma das entradas abaixo ?

```
surpresa("b",[b])=>
surpresa("b",[a b c])=>
surpresa("b",[[a] [b] [c]])=>
surpresa("b",[a b c d e])=>
surpresa("b",[a b c b a])=>
surpresa([b],[[a] [b] [c]])=>
```

3.4.4 LEITURA E GRAVAÇÃO DE DADOS

Tanto a leitura quanto a gravação de dados em memória secundária pode ser feita com o procedimento "datafile". A leitura tem o seguinte formato genérico:

```
datafile('<arquivo>') -> <estrutura>;
```

A gravação tem o seguinte formato genérico:

```
<estrutura> -> datafile('<arquivo>');
```

Exemplo:

```
: [1 2 3 'uma string' uma_palavra [e uma lista]] -> datafile('arq_dados');
: vars meus_dados;
: datafile('arq_dados') -> meus_dados;
: meus_dados =>
** [1 2 3 uma string uma_palavra [e uma lista]]
```

Para que seja efetuada a leitura de dados do teclado (*standard input*), pode-se utilizar, dentre outros, o procedimento "readline". Por exemplo:

```
: vars Linha_Lida;
: readline() -> Linha_Lida;
? Como vao todos voces ?
: Linha_Lida =>
** [Como vao todos voces ?]
```

Para a impressão de dados (*standard output*), pode-se utilizar, dentre outros, os procedimentos "=>" , "==>" , "pr" ou ainda o "ppr". Por exemplo:

```
: obj_tree =>
** [thorax [heart [left_ventricle] [right_ventricle] [left_atrium] [right_at
    rium]] [aorta [ascending_aorta] [descending_aorta] [aortic_arch]]
     [vena_cava [superior_vena_cava] [inferior_vena_cava]] [left_lung]
     [right_lung]]

: obj_tree ==>
```

```
** [thorax [heart [left_ventricle]
                 [right_ventricle]
                 [left_atrium]
                 [right_atrium]]
          [aorta [ascending_aorta]
                 [descending_aorta]
                 [aortic_arch]]
          [vena_cava [superior_vena_cava] [inferior_vena_cava]]
          [left_lung]
          [right_lung]]

 : pr(obj_tree);
 [thorax [heart [left_ventricle] [right_ventricle] [left_atrium] [right_atriu
     m]] [aorta [ascending_aorta] [descending_aorta] [aortic_arch]] [vena_cav
     a [superior_vena_cava] [inferior_vena_cava]] [left_lung] [right_lung]]:
```

REFERÊNCIAS:
help datafile, teach readline, help getline, requestline, pr, ppr, =¿, ==¿

3.4.5 DEPURAÇÃO DE PROGRAMAS EM POP-11

São dados e compilados os seguintes procedimentos funcionais:

```
define cubo(x) -> z;
  x ** 3 -> z;
enddefine;

define somacubos(x, y) -> z;
  cubo(x) + cubo(y) -> z;
enddefine;
```

O segundo procedimento ativa a primeiro da seguinte forma:

```
: somacubos(2, 3) =>
** 35
```

Para acompanhamento detalhado da execução, do ponto de vista de ATIVAÇÃO de procedimentos, parâmetros de ENTRADA, e parâmetros de SAÍDA, o procedimento "trace" é o mais indicado. Por exemplo:

```
: trace cubo somacubos;
: somacubos(2, 3) =>
> somacubos 2 3
!> cubo 2
!< cubo 8
!> cubo 3
!< cubo 27
< somacubos 35
** 35
```

Se compilarmos agora o procedimento recursivo "fatorial", podemos acompanhar melhor sua execução com o comando "trace":

```
: trace fatorial;
: fatorial(5)  =>
> fatorial 5
```

```
!> fatorial 4
!!> fatorial 3
!!!> fatorial 2
!!!!> fatorial 1
!!!!!> fatorial 0
!!!!!< fatorial 1
!!!!< fatorial 1
!!!< fatorial 2
!!< fatorial 6
!< fatorial 24
< fatorial 120
** 120
```

REFERÊNCIAS:
teach trace, help trace, help inspect

Um exemplo de *ERRO* típico pode ser encontrado abaixo:

```
define conta_ocorrencias(elem, lista) -> num;
   if lista == [] then
      0 -> num;
   elseif elem = hd(lista) then
      1 + conta_ocorrencias(tl(lista)) -> num;
   else
      conta_ocorrencias(tl(lista)) -> num;
   endif;
enddefine;

: trace conta_ocorrencias;
: conta_ocorrencias(1, [ 3 4 1 ]);
> conta_ocorrencias 1 [3 4 1]

;;; MISHAP - TOO FEW ARGUMENTS FOR conta_ocorrencias
;;; INVOLVING:  [4 1]
;;; DOING    :  conta_ocorrencias pop_setpop_compiler runproc runproc
!X conta_ocorrencias
X conta_ocorrencias
```

Ou ainda, o esquecimento do teste da lista VAZIA (final da recursão) !!!

```
define conta_ocorrencias(elem, lista) -> num;
   if elem = hd(lista) then
      1 + conta_ocorrencias(elem, tl(lista)) -> num;
   else
      conta_ocorrencias(elem, tl(lista)) -> num;
   endif;
enddefine;

: conta_ocorrencias(1, [ 4 3 2 1 ]);
> conta_ocorrencias 1 [4 3 2 1]
!> conta_ocorrencias 1 [3 2 1]
!!> conta_ocorrencias 1 [2 1]
!!!> conta_ocorrencias 1 [1]
!!!!> conta_ocorrencias 1 []

;;; MISHAP - NON-EMPTY LIST NEEDED
```

```
;;; INVOLVING:  []
;;; DOING    :  hd conta_ocorrencias conta_ocorrencias conta_ocorrencias
    conta_ocorrencias num_ocorrenci as pop_setpop_compiler runproc runproc
!!!!X conta_ocorrencias
!!!X conta_ocorrencias
!!X conta_ocorrencias
!X conta_ocorrencias
X conta_ocorrencias
```

O trace pode ser desetivado temporariamente da seguinte forma:

```
: false -> tracing;    ;;; ou assim
: untrace;
```

Para restaurar a aplicação do trace anteriormente suspendida, executa-se:

```
: true -> tracing;     ;;;  ou mesmo
: trace;
```

Para cancelar permanentemente todos os pontos de chamada do trace, basta:

```
: untraceall;
```

Adicionalmente, estruturas de dados podem ser reveladas em detalhes com o procedimento "inspect". Por exemplo, em uma árvore de nodos compostos por valores do tipo "string" temos:

```
: obj_tree =>
** [thorax [heart [left_ventricle] [right_ventricle] [left_atrium] [right_at
        rium]] [aorta [ascending_aorta] [descending_aorta] [aortic_arch]]
         [vena_cava [superior_vena_cava] [inferior_vena_cava]] [left_lung]
         [right_lung]]

: inspect(hd(obj_tree));
A string of length 6
  [0]  116
  [1]  104
  [2]  111
  [3]  114
  [4]  97
  [5]  120
inspect>

: inspect(obj_tree);
A list of length 6
  [0]  'thorax'
  [1]  ['heart' ['left_ventricle'] ['right_ventricle'] ['left_atrium']
         ['right_atrium']]
  [2]  ['aorta' ['ascending_aorta'] ['descending_aorta'] ['aortic_arch']]
  [3]  ['vena_cava' ['superior_vena_cava'] ['inferior_vena_cava']]
  [4]  ['left_lung']
  [5]  ['right_lung']
  [6]  End:  []
inspect>
```

Ou mesmo quando verificamos o conteúdo de um vetor de bits:

```
: arrayvector(sunrasterfile('DPROC.rs')) -> imagem;
: inspect(imagem);
A v1 of length 469248
  [0]  0
  [1]  0
  [2]  0
  [3]  0
  [4]  0
  . . . .

  [48]  0
  [49]  0
  [50]  0
  [51]  0
*** MORE ***
inspect>m
  . . . .

  [2543]  1
  [2544]  1
  [2545]  1
  [2546]  1
  [2547]  1
*** MORE ***
inspect>
```

3.4.6 CASAMENTO DE PADRÕES

Um "comparador automático de padroes", semelhantemente a um ser humano, deve ser capaz de:

- RECONHECER um dado padrção (true);
- REJEITAR um dado padrao (false);
- INSTANCIAR valores a objetos (por exemplo variáveis).

Isto deve ocorrer levando-se em conta os seguintes parâmetros de comparação de objetos:

- QUANTIDADE;
- TIPO;
- VALOR;
- ORDEM;
- TOLERÂNCIA (faixa de objetos permissíveis em uma "cena" imcompleta).

Mais detalhadamente, em algumas linguagens de programação, um comparador serve para checar a correspondência entre uma "lista FONTE" de objetos e uma "lista de PADROES" (e de objetos).

Isto torna transparente várias tarefas de "baixo-nivel" de abstração para análise combinatória, as quais estão disponíveis em forma sintática simplificada de mais alto nivel.

Em Pop-11, o casamento de padões é efetuado com a ajuda de listas dinâmicas. A lista FONTE (argumento de comparação) pode ser formada por qualquer quantidade de objetos criados em um programa Pop-11 (ex. um procedimento, uma lista, um inteiro).

A lista de PADRÕES pode ser formada por qualquer quantidade tanto dos objetos descritos acima quanto de meta-símbolos como:

- = (casa com um e somente um objeto da lista FONTE)
- == (casa com zero ou mais objetos da lista FONTE)
- ? < *variavel* > (casa com um e somente um objeto da lista FONTE e atribui seu valor ao conteúdo da < *variavel* >)
- ?? < *variavel* > (casa com zero ou mais objetos da lista FONTE e atribui ao conteúdo da < *variavel* > o valor de uma lista contendo os zero ou mais objetos casados).

Exemplos:

```
: [ 1 2 3 ]  matches  [ 1 2 3 ] =>
** <true>
: [ 1 2 3 ]  matches  [ 3 2 1 ] =>
** <false>
: [ 1 2 3 ]  matches  [ = 2 = ] =>
** <true>
: [ 1 2 3 ]  matches  [ == 2 = ] =>
** <true>
: [ 1 2 3 ]  matches  [ == 1 2 3 ] =>
** <true>
: [ 1 2 3 ]  matches  [ = 1 2 3 ]  =>
** <false>
: [ uma estacao e melhor que um micro ]  matches  [ ==  e um  == ]  =>
** <false>
: [ joao e muito feliz ]  matches  [ ?primeiro  e  ??segundo ]  =>
;;; DECLARING VARIABLE primeiro
;;; DECLARING VARIABLE segundo
** <true>
: primeiro =>
** joao
: segundo =>
** [muito feliz]
```

REFERÊNCIAS:
teach matches, help matches, help fmatches

Mais exemplos:

```
: [ joao e muito feliz ]  matches  [ ??primeiro  joao  ??segundo ]  =>
** <true>
: primeiro =>
** []
: segundo =>
** [e muito feliz]
: [ joao e muito feliz ]  matches  [ ?primeiro  joao  ??segundo ]  =>
** <false>
>>>>>>>  C u i d a d o   ! ! !  <<<<<<<
: primeiro =>
** joao
: segundo =>
** [e muito feliz]
: [ joao e muito feliz ]  matches  [ joao  ?verbo ??complemento ]  =>
;;; DECLARING VARIABLE verbo
;;; DECLARING VARIABLE complemento
** <true>
```

```
: verbo =>
** e
: complemento =>
** [muito feliz]
```

Cuidado também com tipos incompatíveis. Ex:

```
: [ joao e muito feliz ]  matches  [ 'joao'  ?verbo ??complemento ]  =>
** <false>
: verbo =>
** e
: complemento =>
** [muito feliz]
```

Utilizando-se de recursos de checagem por restrição para reduzir a tolerância do casamento, as seguintes formas de uso do "matches" podem ser aplicadas:

```
: [a b c d e] -> lista
: lista  matches  [ ?x:isinteger  ??y ]  =>
** <false>
: lista  matches  [ ?x:isword ??y ]  =>
** <true>
: x =>
** a
: y =>
** [b c d e]
:
: 5 -> last(lista);
: lista  matches  [ == ?x:isinteger == ]  =>
** <true>
: x =>
** 5
: 2 -> lista(2);
: lista  matches  [ == ?x:isinteger == ]  =>
** <true>
: x =>
** 2
```

Como base de exemplos de uso do "matches", digite os comandos "showlib eliza" e 'showlib elizaprog", na linha de comando do editor VED, e veja como as citadas bibliotecas foram construídas.