PROLOG

Versão 2.1

- 1999 -

Sumário

1. INTRODUÇÃO

Alguns fatos sobre Prolog:

- O termo Prolog é derivado da expressão "Programming in Logic", uma vez que é baseado em Lógica de Predicados de 1ª ordem.
- Foi criado em 1973, na Universidade de Marseille, França.
- Propósito da criação: criar programas para tradução de linguagem natural (= linguagens faladas, como português, inglês).
- Não é uma linguagem padronizada: padrão ANSI esta sendo formalizado. Enquanto isso, o livro de Clocksin e Mellish, "Programming in Prolog" (1984) é um padrão não oficial.
- Geralmente é interpretado, mas pode ser compilado.
- Escolhida para o projeto japonês da linguagem de 5a. Geração.

1.1 APLICAÇÕES

Principais aplicações se dão na área de computação simbólica:

- Lógica matemática, prova automática de teoremas e semântica;
- Solução de equações simbólicas;
- Bancos de dados relacionais;
- Linguagem Natural;
- Sistemas Especialistas;
- Planejamento Automático de Atividades;
- Aplicações de "General Problem Solving", como jogos (Xadrez, Damas, Jogo da Velha, etc.);
- Compiladores;
- Análise Bioquímica e projetos de novas drogas.

1.2 PROLOG E OUTRAS LINGUAGENS

C, Pascal, Basic: são linguagens Procedimentais (ou imperativas) - especificam **como** deve ser feita alguma coisa. Codificam algoritmos.

C++, SmallTalk, Eiffel: são linguagens Orientadas a Objetos - especificam objetos e seus métodos.

Prolog: é uma linguagem declarativa - especifica **o quê** se sabe e **o quê** deve ser feito.

Prolog é mais direcionada ao **conhecimento**, menos direcionada aos **algoritmos**.

Prolog não possui estruturas de controle como *do-while*, *repeat-until*, *if-then-else*, *for*, *case* ou *switch* como os encontrados em outras linguagens: em Prolog utiliza-se métodos lógicos para declarar como o programa atinge seu objetivo.

A força do Prolog reside em sua capacidade de **Busca** e **Casamento de Padrões**.

1.3 TURBO PROLOG E VISUAL PROLOG

Turbo Prolog é a linguagem que será utilizada no curso. Foi criada pela Borland e hoje é desenvolvida e comercializada pelo Prolog Development Center, na Dinamarca.

Turbo Prolog é diferente do Prolog padrão assim com Turbo Pascal é diferente do Pascal e Turbo C é diferente do C ANSI. O Visual Prolog é uma implementação para Windows do Turbo Prolog, também da PDC.

A maioria das diferenças está relacionada com programação avançada e visa facilitar a compilação de programas. Ainda, o Turbo Prolog possui diversas bibliotecas prontas, como as matemáticas e gráficas (BGI).

Existem outras implementações de Prolog, sendo as mais conhecidas o ARITY Prolog e o SWI Prolog.

2. CONCEITOS BÁSICOS

Definição: Prolog é uma linguagem de programação que é utilizada para resolver problemas que envolvam objetos e relações entre objetos.

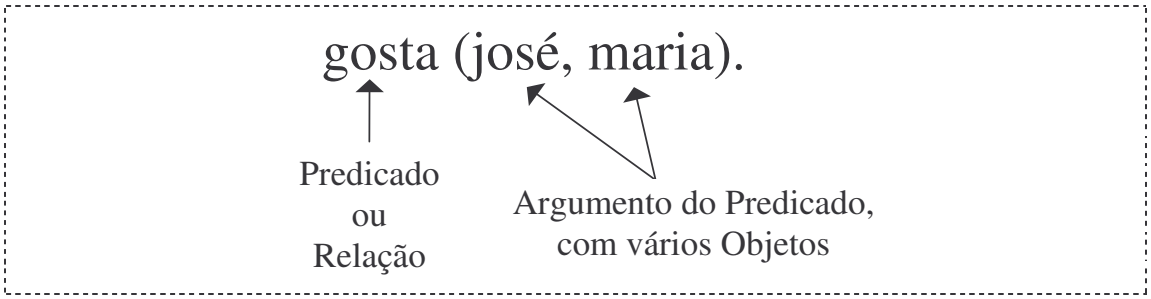
Um programa em Prolog consiste basicamente de:

- declaração de **fatos** (*facts*) sobre objetos e suas relações;
- definições de **regras** (*rules*)sobre os objetos e suas relações;
- **questões** (*goals*) que são feitas sobre os objetos e suas relações.

2.1 FATOS

Os fatos são os elementos fundamentais da programação em Prolog, pois determinam as relações que existem entre os objetos conhecidos.

Exemplo 1:



Características dos fatos:

- Os nomes dos predicados e dos objetos devem começar com letra minúscula. Por exemplo: joão, casa, gosta.
- Os predicados são escritos primeiro e os objetos são escritos depois, separados por vírgulas.
- Os objetos são escritos dentro de parênteses.
- Todo fato é terminado com um ponto final.
- A ordem dos objetos é importante: gosta (maria, josé). ≠ gosta (josé, maria).
- Uma coleção de fatos é chamada de "banco de conhecimento" ou "banco de dados".
- Os fatos podem ter um número arbitrário de objetos como argumento.

NOTA: Para padronizar a criação de predicados (tanto os fatos quanto as regras), devemos sempre pensar que o predicado é uma propriedade do primeiro objeto do argumento. Assim, "gosta (joão, maria)." deve ser lido como "João gosta de Maria".

2.2 QUESTÕES

Uma questão é escrita da mesma maneira que um fato, sendo diferenciada pelo interpretador ou por sua colocação em um local previamente definido e separado dos fatos, como no Turbo Prolog, ou por possuir um ponto de interrogação na frente, como no Prolog definido por Clocksin e Mellish.

- Quando uma questão é feita, o Prolog realiza uma busca na Base de Conhecimento, procurando um Fato que seja igual ao da questão.
- Dizemos que dois fatos (ou um fato e uma questão) são unificam (são iguais) se:
    1. seus predicados são os mesmos,
    2. eles possuem o mesmo número de argumentos e,
    3. os argumentos são iguais.

- Se o Prolog encontra um fato que se iguala a questão, ele retorna “YES”, indicando que a questão tem resposta verdadeira; caso contrário, ele retorna “NO”.
- OBS: O Prolog retornará “NO” toda vez que não conseguir igualar a questão a um fato da Base de Conhecimento. Isso implica que se nada for declarado sobre um determinado problema, qualquer questão relacionada a ele retornara como “NO”.

2.3 UNIFICAÇÃO

Unificação é o nome dado ao processo que tenta casar um objetivo (*goal*) com uma cláusula, que inclui a busca, o casamento de padrões (*pattern-matching*) e as instanciações (Yin, 1987). A unificação é o elemento básico do interpretador ou compilador Prolog.

Basicamente, dizemos que a unificação teve resultado positivo se os objetos envolvidos na comparação são idênticos ou se as cláusulas possuem variáveis que podem ser instanciadas. Caso contrário, dizemos que a unificação falhou.

3. TURBO PROLOG

3.1 A ESTRUTURA DE UM PROGRAMA TURBO PROLOG

Um programa escrito em Turbo Prolog é dividido em várias partes, cada uma com um propósito específico. Estas partes são:

DOMAINS: parte onde são declaradas as diferentes classes dos objetos usados pelo programa. A declaração das classes dos objetos pode ser vista como uma definição dos tipos de todos os objetos a serem usados no programa.

Exemplo 2:

```
tipo_pessoa = symbol
```

As classes dos objetos em Prolog são chamadas de domínios, e a cada nova definição como a acima, estamos definindo um novo domínio para os objetos de um programa. Os domínios pré definidos pelo Turbo Prolog, no qual todo novo domínio deve ser baseado serão descritos à frente. Essa parte é necessária a todo programa.

O Turbo Prolog define alguns domínios de objetos que podem ser usados pelos programadores para definir os domínios em um programa. Uma relação desses domínios é dada na Tabela 1.

Domínio	Valores	Exemplos:
Char	todos os caracteres	‘a’, ‘b’, ‘@’
Integer	de -32768 à 32767	1, 2, -67, 7643
Real	de ± 1e(-307) à ± 1e(308)	6.02e(23), - 64556
String	uma seqüência de no máximo 256 caracteres, entre aspas duplas.	“Reinaldo Bianchi”, “1 2 3 4”, “Leopoldo”
symbol	uma seqüência de letras, dígitos, e traços, com o primeiro caracter em minúsculo. Também pode ser uma seqüência entre aspas duplas.	maria, jose, p1999, manoel, flores, maria_fumaça, “Alice”, “Anna Rillo”, dia_d.
File	um nome válido de um arquivo do DOS	exer01.pro, teste.txt

Tabela 1: Domínios do Turbo Prolog.

PREDICATES: nesta parte são declarados os predicados que poderão ser usados no programa, indicando o domínio de seus argumentos. A definição de cada predicado pode ser comparada com a declaração do protótipo da função que é feita em linguagem C. Um predicado deve ser definido para cada fato ou regra a ser utilizado no programa.

Exemplo 3:

```
gosta (tipo_pessoa, tipo_pessoa).
```

CLAUSES: nesta parte são declarados todos os fatos e todas as regras do programa.

Exemplo 4:

```
gosta (joão, maria).              /* um fato */
gosta (joão, X) :- X = mulher.    /* uma regra */
```

GOAL: é a chamada meta, o objetivo do programa. É uma questão que vai dar início ao programa. O objetivo pode ser interno ou externo. Ele é interno se for definido dentro do corpo do programa, e externo, quando é definido interativamente, através da formulação de questões na janela de diálogo.

Exemplo 5:

```
gosta (joão, maria).
```

DATABASE: parte usada para declarar os predicados usados por um banco de dados dinâmico em um programa. É necessária somente quando se utiliza bancos de dados dinâmicos.

NOTA: Comentários podem ser inseridos em um programa com os delimitadores /* e */.

As partes *Domains*, *Predicates* e *Clauses* são necessárias a todo programa e a declaração do objetivo interno é necessária somente quando se deseja compilar o programa.

O Turbo Prolog não diferencia letras maiúsculas de minúsculas, a não ser em símbolos, onde os iniciados por maiúscula são considerados variáveis. Para criar uma padronização, aconselha-se a escrever os programas usando sempre letra minúscula, deixando as letras maiúsculas somente para as variáveis.

3.2 PRIMEIRO PROGRAMA

Abaixo, temos um exemplo de um programa simples em Prolog. Note que ele possui as 3 partes básicas (*Domains*, *Predicates* e *Clauses*). Esse exemplo é muito conhecido e foi proposto no livro de Clocksin e Mellish (1984):

Exercício 1: Exemplo de Clocksin e Mellish (1984).

```
/*
 * EE - 097
 * Exercício 01
 */

domains
   tipo_nome = symbol

predicates
   gosta(tipo_nome, tipo_nome).

clauses
   gosta (josé, peixe).
   gosta (josé, maria).
   gosta (maria, livro).
   gosta (joão, livro).
```

Para iniciar o programa apresente as seguintes questões ao interpretador Prolog e anote as respostas:

```
gosta (josé, peixe).

gosta (josé, dinheiro).

gosta (maria, josé).

gosta (maria, livro).

possui (joão, livro).
```

Adicione outros predicados ao programa, como por exemplo “fica”, “ama”, “odeia”.

3.3 CONJUNÇÕES

Uma conjunção ("E" Lógico) é feita colocando-se uma vírgula entre os goals.

Exercício 2: Conjunção. Se quisermos saber se joão gosta de maria e maria gosta de joão, fazemos a seguinte questão para o Prolog:

```
gosta (joão, maria), gosta (maria, joão).
```

Verifique a resposta.

- A resposta programa ao goal total será "YES" se toda a seqüência de goals for satisfeita.
- O Prolog tenta satisfazer os goals da esquerda para a direita.

NOTA**:** Já podemos observar que o Prolog é basicamente uma busca em uma Base de Conhecimento por fatos que se igualem a questão.

3.4 VARIÁVEIS

Uma variável em Prolog sempre começa com uma letra maiúscula.

Uma variável pode estar instanciada ou não-instanciada. Dizemos que ela está instanciada quando estiver assumindo o valor de um objeto e, caso contrário, não-instanciada.

Exemplo 6: Considere a seguinte Base de Conhecimento:

```
gosta (joão, flores).
gosta (joão, maria).
gosta (paulo, maria).
```

Ao ser realizada a questão:

```
gosta (joão, X).
```

A variável X está inicialmente não-instanciada. O Prolog procura então na Base de Conhecimento por um fato que se iguale a questão, isto é, que tenha o mesmo predicado, o mesmo número de argumentos e que o primeiro argumento seja "joão". A procura é feira na ordem em que os fatos estão na Base de Conhecimento. Ao encontrar o primeiro fato, o Prolog verifica se todos os requisitos estão cumpridos, e instancia a variável X com o objeto "flores".

NOTA**:** No Turbo Prolog, se estivermos fazendo uma questão através da janela de diálogo (um goal externo), após encontrar o primeiro valor de X, o programa continua procurando na Base de Conhecimento por outros valores e, no final, apresenta uma lista com todas as respostas. Porém, se for um goal interno (dentro do programa), o Prolog só encontrará a primeira resposta.

Exercício 3: Lista de alunos.

```
/*
 * EE - 097
 * Exercício 03
 */

domains
   tipo_nome = symbol

predicates
   aluno(tipo_nome).

clauses
   aluno(benedicte).
   aluno(alice).
   aluno(marcelo).
   aluno(andré).
   aluno(roberto).
```

Para iniciar o programa, apresente uma questão ao interpretador Prolog digitando na janela de diálogo "aluno(X)."

O funcionamento do interpretador Prolog pode ser visto usando uma das ferramentas de depuração do Turbo Prolog, o Trace. Acione o Trace através do menu "Options - Compiler directives - Trace - Trace". Quando acionado, a tecla F10 fará o programa ser executado passo a passo (conhecido com *Step*).

Verifique o funcionamento do programa acima usando o Trace.

3.5 ARIDADE

O termo aridade é usado para a quantidade de objetos que o argumento de um predicado possui.

Exemplo 7:

Predicado	Aridade
gosta (maria, josé).	2
bebe (penélope, pinga, vodka, rum).	4

Tabela 2: Exemplos de aridade.

4. O RETROCESSO (BACKTRACKING)

O retrocesso é um mecanismo usado pelo Prolog para encontrar fatos ou regras adicionais que satisfaçam um objetivo (o goal) quando a tentativa corrente de unificação falha, em questões com sub-goals (cada pedaço de uma questão separada por vírgulas).

O Prolog utiliza marcadores para marcar os pontos onde uma nova tentativa de solução deve ser procurada.

Toda vez que é realizada uma questão, o Prolog tenta solucioná-la comparando a questão com os fatos na Base de Conhecimento. Quando a questão possui muitos sub-goals, a falha em uma busca pode acontecer. Neste momento, o Prolog precisa de uma maneira para "lembrar" os pontos de onde pode tentar modificar a solução corrente para procurar uma resposta certa. O retrocesso é a implementação desta "lembrança", e os pontos importantes para procurar a resposta certa são as variáveis: ele marca onde as variáveis estão no programa e quais foram as instanciações, para poder mudá-las caso necessite.

O Prolog tenta encontrares os "pares que casam" da esquerda para direita. Se uma procura falha, volta para o último momento onde instanciou uma variável e procura se existe outra possibilidade de instanciação.

O exemplo abaixo foi apresentado por Clocksin e Mellish (1984) e explica o processo de busca por uma solução para uma questão com dois sub-goals e uma variável.

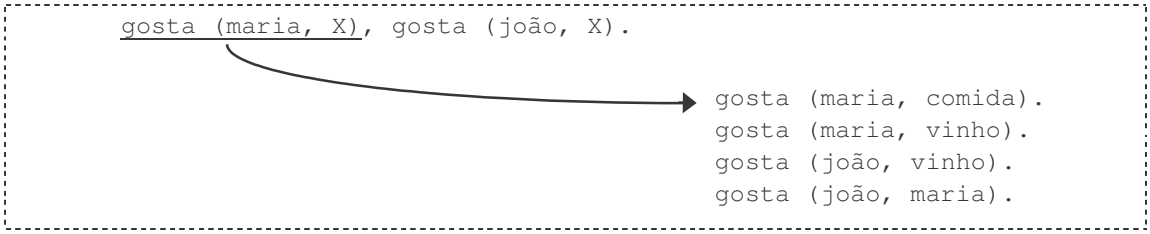
Exemplo 8: Considere a seguinte Base de Conhecimento:

```
gosta (maria, comida).
gosta (maria, vinho).
gosta (joão, vinho).
gosta (joão, maria).
```

É realizada a questão:

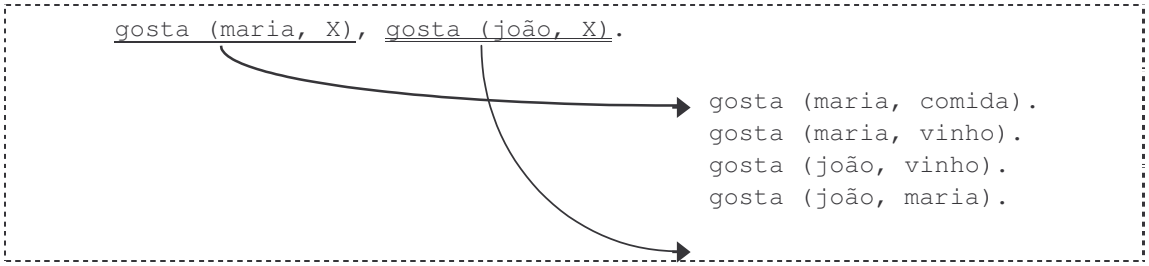
```
gosta (maria, X), gosta (joão, X).
```

Que significa "O quê Maria e João gostam ao mesmo tempo?".



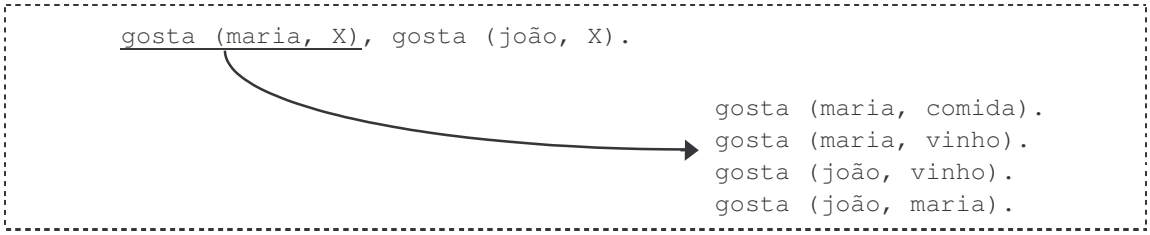
Inicialmente, a procura por uma solução para o primeiro sub-goal é bem sucedida, com a variável X sendo instanciada com o objeto "comida".

Em seguida, o Prolog tenta encontrar um fato que satisfaça o segundo sub-goal. Como a variável X já está instanciada, o fato que ele procura é: "gosta (joão, comida).".

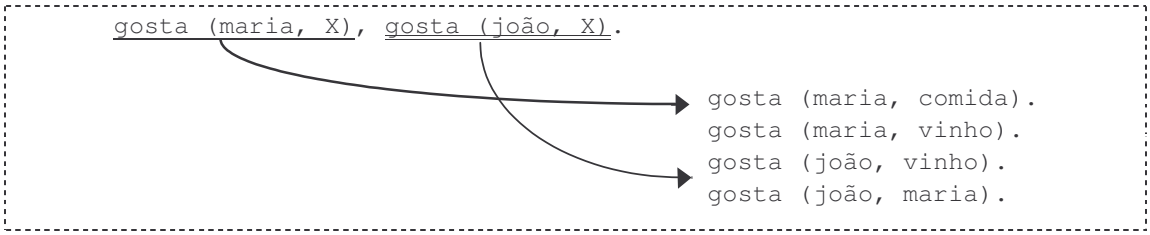


Neste momento a procura pelo segundo sub-goal falha.

Em seguida, o retrocesso é realizado: o valor instanciado na variável X é “esquecido’, e o Prolog procura outro valor para a variável. No caso, ele encontra o valor “vinho”.



Agora, o Prolog tenta satisfazer o segundo sub-goal, com X = “vinho”.



No final, o Prolog encontra na Base de Conhecimento um fato que satisfaz o segundo sub-goal e avisa ao usuário que ele foi encontrado.

5. REGRAS

As regras são utilizadas para construir relações entre fatos, explicitando as dependências entre eles.

Ao contrário dos fatos, que são incondicionais, as regras especificam coisas que podem ser verdadeiras se algumas condições forem satisfeitas.

As regras possuem duas partes:

- o corpo, que define as condições e se encontra na parte direita da regra, e
- a cabeça, que define a conclusão, e se encontra na parte esquerda da regra.

A cabeça e o corpo são separados pelo símbolo “**:-**”, que é lido como “se”.

Uma regra sempre é terminada com um ponto final.

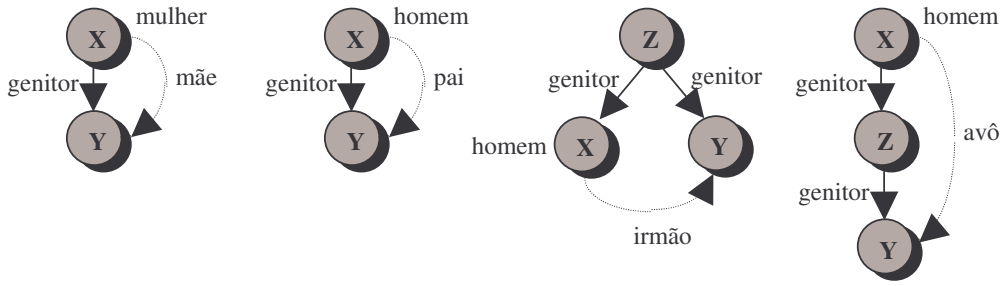
Exemplo 9:

```
gosta (joão, X) :- gosta (X, vinho).

gosta (joão, X) :- gosta (X, vinho), gosta (X, comida).

filho (X, Y) :- homem (X), genitor(Y, X).
```

Exercício 4: Baseado nos predicados:

```
homem (nome).
mulher (nome).
genitor(nome, nome).
```

E nos esquemas abaixo:



Defina as regras para:

Mãe:

Pai:

Avô:

Avó:

Irmão:

Irmã:

Outros predicados: primo, prima, tio, filho, sobrinho, sogro, sogra, genro, etc.

5.1 RESOLUÇÃO COM REGRAS

Dada a seguinte Base de Conhecimento em um programa Turbo Prolog:

```
/*
 * EE - 097
 * Exercício 05
 */

domains
   tipo_nome = symbol

predicates
   homem (tipo_nome).
   mulher (tipo_nome).
   genitor(tipo_nome, tipo_nome).
   irmã (tipo_nome, tipo_nome).

Clauses
   /* fatos */
   genitor (jaques, benedicte).
   genitor (miriam, benedicte).
   genitor (jaques, carolina).
   genitor (miriam, carolina).
   mulher (miriam).
   mulher (benedicte).
   mulher (carolina).
   homem (jaques).

   /* regras */
   irmã (X, Y) :- genitor (Z, X),
                  genitor (Z, Y),
                  mulher (X),
                  X <> Y.
```

Ao ser realizada a questão “irmã (benedicte, carolina).”, o Turbo Prolog inicialmente procura por um fato igual a questão. Após não encontrar o fato, encontra uma regra que declara como é a relação “irmã”. Neste momento, instancia as variáveis da regra:

X = “benedicte” e Y = “carolina”

Depois de instanciada, o Prolog tenta descobrir se a regra é valida, verificando se a parte condicional da regra é verdadeira. O goal original transforma-se em um conjunto de sub-goals dados pela regra. Se todos os sub-goals da regra forem verdadeiros, a cabeça da regra e o goal original também são, e o Prolog retorna “YES”.

Após encontrar uma regra e instanciar suas variáveis, os fatos do lado direito da regra se tornam o goal que o programa deve satisfazer e são verificados normalmente, da esquerda para a direita.

Exercício 5: Verifique com Trace o funcionamento do exemplo acima.

5.2 RETROCESSO (BACKTRACKING) COM REGRAS

Nas regras também ocorre o retrocesso: caso um sub-goal não seja satisfeito, o Prolog tenta modificar o valor das variáveis instanciadas para procurar uma resposta válida.

A diferença entre se fazer uma questão com valores especificados e uma questão com variáveis é que no segundo caso as variáveis da regra que correspondem às variáveis não instanciadas na questão, continuarão não instanciadas, permitindo o retrocesso.

Exercício 6: Para a mesma Base de Conhecimento acima, faça as seguintes questões:

```
irmã (benedicte, Quem).
```

Verifique o funcionamento do programa acima usando o Trace.

Verifique o funcionamento das regras para irmão, filho, filha, etc.

5.3 DISJUNÇÃO ("OU")

A disjunção (o "OU" lógico) é representada no Prolog quando se declara mais de uma regra para determinar uma mesma relação. Isso ocorre porque na busca de uma solução, o Prolog automaticamente procura as regras que podem ser aplicadas quando uma regra falha.

Exercício 7: Para a mesma Base de Conhecimento dos exercícios acima, construa uma regra que explicite a relação "meio-irmão", não incluindo irmão congênito.

6. ENTRADA E SAÍDA

O Turbo Prolog fornece ao programador os seguintes predicados para a entrada e saída de dados:

WRITE: usado para escrever uma mensagem no dispositivo padrão de saída, geralmente a janela de diálogo. O argumento do write pode ser tanto uma variável como um objeto (símbolo, inteiro, string, etc.).

Exemplo 10:

```
write("Que dia lindo!!!").
write(Nome).
```

READLN: usado para ler uma string ou um símbolo do dispositivo padrão de entrada, geralmente o teclado, e guardá-lo em uma variável. O Turbo Prolog possui vários predicados de leitura, um para cada tipo de objeto a ser lido. Os outros predicados usados são:

READINT: para a leitura de um número inteiro.

READCHAR: para a leitura de um caracter.

READREAL: para a leitura de um número em ponto flutuante.

Exemplo 11:

```
readln (Nome).
readint (Dia).
```

NL: este predicado é usado para forçar uma mudança de linha.

Exercício 8: Verifique o funcionamento do programa abaixo, que pergunta seu nome e o imprime.

```
/*
 * EE - 097
 * Exercício 08
 */

predicates
   dialogo.

clauses
   dialogo :- nl, nl,
              write ("Qual e' o seu nome?"),
              readln (Nome), nl,
              write("Ola', ", Nome, "."), nl.
```

Exercício 9: Adicione ao programa do exercício 03, uma regra que leia um nome digitado pelo usuário, procure o nome na Base de Conhecimentos e imprima "Encontrei" ou "Não encontrei.". Usar um *goal* interno para executar o programa.

Exercício 10: Imprima como resultado “Encontrei <nome> na Base de Conhecimentos” ou “Não encontrei <nome> na Base de Conhecimento”. Neste exercício, verifique o escopo das variáveis e a passagem de parâmetros entre predicados.

7. CONTROLE DO RETROCESSO

O b*acktracking*, ou retrocesso, é um mecanismo existente no Prolog que procura fatos ou regras adicionais que satisfaçam um objetivo em uma questão com vários objetivos (*sub-goals*) quando a instanciação corrente falhou.

O controle do *backtracking* é importante pois é uma das maneiras de provocar repetição, sendo outra maneira o uso da recursão. Existem dois predicados fornecidos pelo Prolog para o controle do *backtracking*: a falha (*fail*) e o corte (*cut*).

7.1 FALHA (FAIL)

O predicado fail (chamado de falha em português), pré-existente em toda implementação do Prolog, sempre retorna uma falha na unificação. Com isso, força o Prolog a fazer o *backtracking*, repetindo os predicados anteriores ao predicado fail.

Por ser muito importante, o uso do fail para forçar o *backtracking* é chamado de método “*Backtrack After Fail*” de controle.

Exercício 11: Dado a Base de Conhecimentos do exercício 03, construir uma regra que imprima todos os nomes dos alunos, usando o comando fail.

```
/*
 * EE - 097
 * Exercício 11
 */

domains
   tipo_nome = symbol

predicates
   aluno(tipo_nome).
   escreve_todos.

clauses
   /* fatos */
   aluno(benedicte).
   aluno(alice).
```

```
aluno(marcelo).
aluno(andre).
aluno(roberto).

/* regras */
escreve_todos :- aluno (Nome),
                 write (Nome),
                 nl,
                 fail.
```

Verifique o funcionamento do predicado fail usando o Trace e o F10. Retire o fail e verifique o resultado. Como fazer para que a regra "escreve_todos" resulte verdadeira, após terminar de escrever toda a lista? (i. e., como fazer para termos "Yes" em vez de "No" ao final das execução?)

Notar a diferença de quando se pergunta interativamente a questão "aluno(X)." e quando se faz uma regra. Interativamente, o *backtracking* é feito automaticamente e em uma regra, não.

Exercício 12: Adicione nomes repetidos na Base de Conhecimentos. Faça uma regra que receba um nome, procure-o na base de dados e imprima "Encontrei" todas as vezes que ele aparece.

7.2 CORTE (CUT)

O Prolog, na busca de uma solução, sempre faz automaticamente o *backtracking* após uma tentativa de unificação de um predicado que não resultou positiva.

O corte é um predicado existente em todas as implementações do Prolog, usado para evitar o *backtracking*. Ao usar o corte, o Prolog "esquece" todos os marcadores de *backtracking* que foram feitos ao se tentar satisfazer o sub-goal atual e não mais procura por outras possíveis unificações caso ocorra uma falha.

Com isso, o corte pode tornar um programa mais rápido, pois evita que o Prolog explore alternativas que, sabe-se de antemão, não irão contribuir para a solução do problema e ainda permite a economia de memória, pois elimina marcadores de *backtracking*.

O símbolo do corte é uma exclamação "!".

Apesar de ser usualmente comparado com o *break* existente em linguagens como Pascal e Linguagem C, o funcionamento do corte não é exatamente igual ao dele, existindo muitas

diferenças entre os dois. Por isso, o aluno não deve pensar no corte simplesmente como um *break*.

O corte pode ser visto como um diodo, que só permite a passagem da busca da esquerda para direita e não da direita para esquerda. Assim, as unificações feitas pelo Prolog à esquerda de um corte não podem ser desfeitas após a passagem da busca do Prolog para o lado direito do corte.

Por exemplo, na regra

teste :- a, b, c, !, d, e, f.

o Prolog tenta satisfazer os *sub-goals* a, b e c, podendo realizar o *backtracking* entre essas clausulas até que c seja satisfeita, o que causará a passagem da busca do Prolog da esquerda para direita do corte. Depois dessa passagem, o Prolog pode tentar satisfazer as clausulas d, e e f, tendo ou não que realizar o *backtracking* para isso, até o momento que a clausula d falha. Nessa hora, não é mais possível voltar para escolher outra unificação para a clausula c e toda a regra "teste" falha.

Os cortes são chamados de "verdes" quando não mudam o resultado da busca ou "vermelhos" quando mudam o resultado. O corte verde é usado para eliminar os marcadores do *backtracking*, ponteiros que ocupam espaço na memória.

Exercício 13: Corte verde. Os predicados abaixo definem a função:

$$f(x) = \begin{cases} 0 \text{ se } x < 3, \\ 2 \text{ se } x \geq 3 \text{ e } x < 6 \\ 4 \text{ se } x \geq 6 \end{cases}$$

```
/*
 * EE - 097
 * Exercício 13
 */

domains
   tipo_numero = integer

predicates
   f(tipo_numero, tipo_numero).

clauses
   /* regras */
   f(X,0) :- X < 3,!.
   f(X,2) :- 3 <= X, X < 6,!.
   f(X,4) :- 6 <= X,!.
```

Use a questão "f(2,Resposta), Resposta < 2." e verifique o resultado. Usando o trace e F10, verifique o funcionamento da função. Retire os cortes e verifique novamente. Como estes cortes são verdes, as respostas devem ser iguais com ou sem os cortes. Podemos notar

que com o uso do corte, eliminamos a passagem do programa por regras que sabemos de antemão que não serão úteis para resolver o problema.

Exercício 14: Corte vermelho. Para definir a mesma função f(x) usada no exercício anterior, usamos os predicados abaixo:

```
/*
 * EE - 097
 * Exercício 14
 */

domains
   tipo_numero = integer

predicates
   f(tipo_numero, tipo_numero).

clauses
   /* regras */
   f(X,0) :- X < 3,!.
   f(X,2) :- X < 6,!.
   f(X,4).
```

Use a questão "f(2,Resposta), Resposta < 2." e verifique a resposta. Usando o trace e F10, verifique o funcionamento da função. Retire os cortes e verifique novamente.

7.3 MÉTODO DO CORTE E FALHA (CUT AND FAIL)

Ao se utilizar o predicado de corte em conjunto com falhas, provocadas pelo predicado fail ou por uma regra, obtém-se um poderoso método de controle. Neste método, as falhas (do fail ou de uma regra) são usadas para forçar o *backtracking* até que uma condição específica seja encontrada, quando se usa o corte para terminá-lo.

Esse método lembra o *repeat-until* das linguagens procedimentais, porém é mais poderoso, pois o *until* não é apenas uma condição, mas um predicado que pode inclusive realizar operações.

Exercício 15: *Cut and Fail*.

```
/*
 * EE - 097
 * Exercício 15
 */

domains
   tipo_nome = symbol

predicates
   aluno(tipo_nome).
   escreve_ate(tipo_nome).
   controle_do_corte(tipo_nome, tipo_nome).
```

```
clauses
   /* fatos */
   aluno(benedicte).
   aluno(alice).
   aluno(marcelo).
   aluno(andre).
   aluno(roberto).

   /* regras */
   escreve_ate (Nome_final) :-
        aluno (Nome),
        write (Nome),
        nl,
        controle_do_corte(Nome, Nome_final),
        !.

   /* A regra abaixo falha provocando o backtracking,
      ate que o nome corrente da busca seja igual
      ao final, dado pelo usuário. Assim, ela substitui
      o predicado fail neste programa. */

   controle_do_corte(Nome_2, Nome_final_2) :-
     Nome_2 = Nome_final_2.
```

8. RECURSÃO

Outra método muito usado para causar a repetição em programas escritos em Prolog é a recursão. Toda recursão é composta por duas regras: a primeira, que define a condição de parada da recursão; e a segunda, onde é feita a chamada à função recursiva.

Exercício 16: Cálculo do Fatorial de um número.

```
/*
 * EE - 097
 * Exercicio 16
 */

domains
   tipo_numero = real

predicates
   fatorial (tipo_numero, tipo_numero).

clauses
   /* regras */
   /* A regra abaixo e' o teste da recursao.*/
   fatorial (0, Resultado) :- Resultado = 1,!.

   /* A regra abaixo gera as chamadas recursivas. */
   fatorial (Numero, Resultado) :-
                Numero_menos_1 = Numero - 1,
                fatorial(Numero_menos_1, Resultado_parcial),
                Resultado = Numero * Resultado_parcial.
```

Formule uma questão e utilize o Trace para verificar o funcionamento deste programa. Explique qual a finalidade do corte no final da primeira regra.

8.1 A VARIÁVEL ANÔNIMA "_"

O Prolog permite a definição de uma variável anônima (chamada "_") que pode ser usada quando uma regra necessita em sua cabeça de uma variável que não será usada em seu corpo. Essa variável é muito usada porque o Turbo Prolog avisa automaticamente quando uma variável é usada somente uma vez, e pode ser trocada pela variável anônima "_". Ao se trabalhar com listas, a variável anônima é útil e deve ser usada.

9. LISTAS

Uma lista é um conjunto ordenado de objetos, todos sendo do mesmo tipo de domínio.

Exemplo 12:

```
[1,2,3,4,5,6,7,8,9,10]            comprimento = 10
[alfa, beta, gama, delta]         comprimento = 4
[]                                lista vazia
```

Uma lista é delimitada pelos colchetes e seus elementos são separados por virgulas.

No Turbo Prolog, para que um fato ou uma regra possa ter uma lista como argumento, deve-se definir nos *Domains* quais listas serão usadas no programa e de que tipo de objetos essas listas serão compostas.

Exemplo 13: para definir um tipo que é uma lista de inteiros, escreve-se:

```
tipo_lista_de_inteiros = integer*
```

onde o * é que indica que lista_de_inteiros é uma lista, composta por elementos do tipo inteiro.

Exemplo 14: Se desejarmos criar uma matriz de inteiros, usamos:

```
tipo_vetor_de_inteiros = integer*
tipo_matriz_de_inteiros = tipo_vetor_de_inteiros*
```

Uma lista é guardada na memória como uma árvore binária, onde o primeiro elemento é guardado junto com um apontador para o resto da lista. Assim, uma lista é composta por duas partes: a primeira, que é o primeiro elemento da lista, chamado de cabeça; e a segunda, que é o resto da lista, chamado de cauda. A tabela abaixo apresenta algumas listas com suas cabeças e caudas.

Lista	Cabeça	Cauda
[]	erro	erro
[a, b, c, d]	a	[b, c, d]
[b, c, d]	b	[c, d]
[a]	a	[]

Tabela 3: Listas, suas cabeças e caudas.

Ao se unificar uma lista com uma variável, o Prolog trabalha normalmente, instanciando as variáveis como já foi visto. Duas listas unificam quando: possuem o mesmo número de objetos ou pelo menos o número suficiente para todas as variáveis serem instanciadas.

Exemplo 15: Unificação de uma lista com uma variável:

Lista 1	Variável	Unificação
[a, b, c]	X	X = [a, b, c]

Durante uma unificação de duas listas, a cabeça de uma lista pode ser separada de sua cauda com o uso do "|" (o caracter barra vertical, ou *pipe*).

Usando-se a lista [Cabeça | Cauda] (onde Cabeça e Cauda são duas variáveis) para receber uma lista em um argumento de uma cláusula, em vez de uma única variável, a variável Cabeça fica com o primeiro elemento da lista e a variável Cauda com o resto da lista. Essa separação é feita durante a unificação das duas listas (a que está sendo passada e a lista [Cabeça | Cauda]).

Exemplo 16: Unificação de duas listas.

Lista 1	Lista 2	Unificação
[a, b, c]	[X | Y]	X = a, Y = [b, c]
[casa, pata]	[X | Y]	X = casa, Y = [pata]
[a, b, c]	[X, Y]	Não unifica

Exercício 17: Unifique as listas abaixo:

Lista 1	Lista 2	Unificação
[[a, b] | Y]	[H | T]	
[a, b]	[X, Y | Z]	

[X, Y]	[[a, b], [c, d]]	

A recursão é muito usada ao se manipular listas, porque associada às listas com “l”, regras que varrem toda uma lista podem ser criadas.

Exercício 18: Imprimindo uma lista usando recursão. Verifique com o Trace o funcionamento do programa abaixo, que imprime uma lista .

```
/*
 * EE - 097
 * Exercicio 18
 */

domains
   tipo_nome = symbol
   tipo_lista_de_nomes = tipo_nome*

predicates
   alunos(tipo_lista_de_nomes).
   imprime_lista(tipo_lista_de_nomes).

clauses
   /* fatos */
   alunos([benedicte, alice, marcelo, andre, roberto]).

   /* regras */
   imprime_lista([]).
   imprime_lista([Cabeca|Cauda]) :- write (Cabeca),
                           nl,
                           imprime_lista(Cauda).

goal
   alunos (X),
   write("A lista com todos os alunos declarados e':"),
   nl,
   imprime_lista(X).
```

Exercício 19: Crie uma regra que procura um elemento em uma lista.

Exercício 20: O programa abaixo realiza a concatenação de duas listas.

```
/*
 * EE - 097
 * Exercicio 20
 */

domains
   tipo_nome = symbol
   tipo_lista_de_nomes = tipo_nome*

predicates
   concatena_listas
 (tipo_lista_de_nomes,tipo_lista_de_nomes,tipo_lista_de_nomes).

clauses
   /* regras */
   /* concatena a primeira lista + a segunda na terceira.*/
   concatena_listas ([],L,L).
   concatena_listas ([N|L1],L2,[N|L3]) :-
                  concatena_listas (L1,L2,L3).
```

Faça uma questão passando como argumentos duas listas de símbolos e uma variável. Como funciona a concatenação? Verifique com o Trace.

Exercício 21: Resolva o "Problema dos Missionários e Canibais".

Outros Exercícios: Implemente regras para o tratamento de listas que:

1.calcula o comprimento de uma lista;

2.insere um elemento em uma lista;

3.retira um elemento de uma lista;

4.substitui elementos em uma lista;

5.encontra o N-ésimo elemento de uma lista;

6.divide uma lista ao meio;

7.soma os elementos de uma lista numérica;

8.inverte uma lista.

10. ESTRUTURAS COMPOSTAS

Até agora foram vistos predicados do tipo:

gosta (reinaldo, o_iluminado).

Neste fato, os objetos "reinaldo" e "o_iluminado" são chamados de objetos simples e uma estrutura composta somente por objetos simples é chamada de estrutura simples.

Mas o fato acima não especifica se "o_iluminado" que "reinaldo" gosta é um livro ou um filme. Para poder fazer esta distinção, o Prolog permite a construção de objetos e predicados compostos.

gosta (reinaldo, livro (o_iluminado)).

gosta (reinaldo, filme (o_iluminado)).

Os predicados acima são chamados de estruturas compostas, pois são constituídas por objetos compostos, no caso, livro e filme.

Exemplo 17:

```
gosta (marcelo, bebidas_alcoolicas(pinga, whisky, vodka, jim)).
pai( jose, filhos (manoel, joaquim), filhas (maria, rita)).
```

As estruturas compostas lembram os registros (*records* do Pascal e *structs* de C) das linguagens procedimentais.

Dá-se o nome de funtor ao primeiro termo de um objeto composto. Nos exemplos acima, bebidas_alcoolicas , filhos e filhas são funtores.

Como qualquer outro objeto, os domínios dos objetos compostos devem ser declarados na parte *domains* do Turbo Prolog, onde deve ser descrito o número de argumentos e domínio dos argumentos do novo domínio de objetos.

Exercício 22:

```
/*
 * EE - 097
 * Exercicio 22
 */

domains
   tipo_data = data(tipo_dia, tipo_mes, tipo_ano)
   tipo_dia, tipo_ano = integer
   tipo_pessoa, tipo_mes = symbol

predicates
   aniversario (tipo_pessoa, tipo_data).

clauses
   aniversario (rita, data (21, fevereiro, 1976)).
   aniversario (reinaldo, data (26, julho, 1970)).
   aniversario (alice, data (23, setembro, 1974)).
   aniversario (mayra, data (08, junho, 1976)).
```

Verifique o funcionamento do programa acima, encontrando os seguintes objetivos externos: Quem nasceu em 1976? Quem nasceu após 1973?

As estruturas dinâmicas podem ser representadas graficamente através de diagramas. Por exemplo, para a estrutura data, temos:

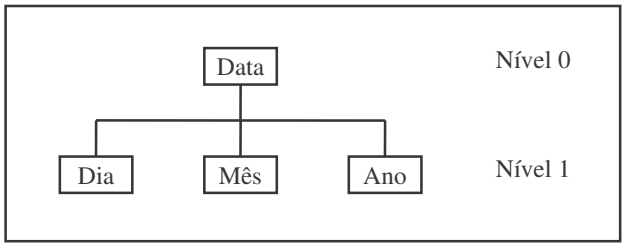


Figura 1: Diagrama do domínio da estrutura data.

No Turbo Prolog, geralmente se associa um nome diferente do funtor da estrutura para designar o domínio da estrutura. Porém, isto não é necessário e não é usado em outras implementações do Prolog.

A estrutura dos predicados do programa também pode ser representada graficamente. Para o programa 23, temos:

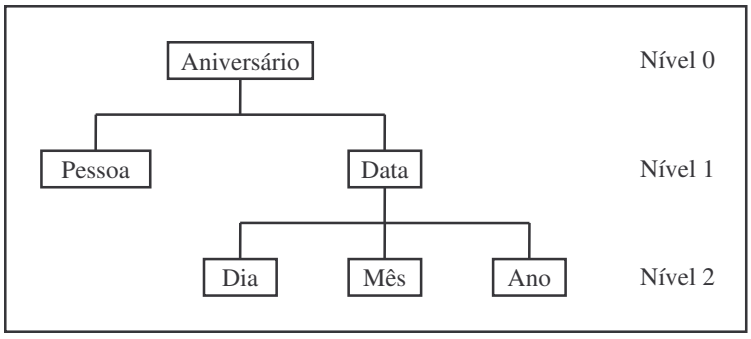


Figura 2: Diagrama da estrutura de predicados do exercício 15.

Exercício 23:

Crie uma estrutura para catalogar livros em uma biblioteca. A estrutura que define um livro deve ter como objetos autor, título, data de publicação e editora. Esse objeto composto deve ser parte de um predicado que contém ainda para quem o livro está emprestado. Desenhe os diagramas do domínio e de predicado do seu programa. Depois de criado, tente encontrar os seguintes objetivos externos: Quais são os títulos publicados em um determinado ano? Quais livros foram publicados após 1990? (Exercício proposto por Yin, 1987.)

Finalmente, o Prolog permite a criação de listas de estruturas, onde todos os elementos devem ser do mesmo tipo de domínio.

Exemplo 18:

```
[data(4, março, 1997), data(22, julho, 1960)]
```

11. BANCOS DE DADOS DINÂMICOS

O Turbo Prolog permite a construção de bancos de dados dinâmicos, isto é, que podem ser alterados e armazenados em disco para nova utilização. Esses bancos de dados podem conter somente fatos. Os predicados que definem os fatos usados no banco de dados dinâmico devem ser declarados na parte "*database*" do programa e não na "*predicates*". Assim, um mesmo programa pode ter fatos declarados na parte de "*predicates*", que só podem ser alterados com a modificação do código fonte do programa, e fatos guardados em um arquivo separado, que podem ser carregados, modificados e salvos.

O Turbo Prolog fornece ao programador os seguintes predicados para manipular bancos de dados dinâmicos:

ASSERTA: este predicado acrescenta um fato ao banco de dados armazenado na memória do computador. Ele acrescenta o novo fato no início das cláusulas de mesmo predicado.

```
asserta (gosta(maria, jose)).
```

ASSERTZ: este predicado é similar ao asserta, porém adiciona o novo fato após as cláusulas de mesmo predicado.

RETRACT: é um predicado usado para retirar um fato do banco de dados. Ele retirará o primeiro fato que for casado com o seu argumento.

```
retract (gosta( maria, jose)).
```

FINDALL: predicado que procura no banco de dados todos os fatos que combinem com seu primeiro argumento no predicado passado pelo seu segundo argumento, e os coloca em uma lista, que é seu terceiro argumento.

```
findall (Nome, gosta(Nome,_), Lista_resultante).
```

CONSULT: para carregar um banco de dados do disco para a memória é usado o predicado consult. Ele recebe como argumento o nome do arquivo a ser carregado.

```
consult ("meu_b_d.dba").
```

O nome dado a um arquivo pode ser qualquer, desde que obedeça o formato de 8 + 3 letras do DOS. A extensão do arquivo é definida pelo usuário, mas geralmente se usa .dba.

Uma observação que deve ser feita é que o consult carregará o banco de dados do disco concatenando-o com o da memória. Por exemplo, se o usuário carregar duas vezes o mesmo banco de dados, ele ficará na memória com duas cópias de todos os fatos carregados, uma após a outra.

SAVE: este predicado transfere todos os fatos existentes na memória para um arquivo. Ele recebe como argumento um nome que seja válido para um arquivo no DOS.

```
save ("meu_b_d.dba").
```

Exercício 24: Agenda de telefones.

```
/* EE - 097
 * Exercicio 24.
 */

domains
     tipo_nome = string.
     tipo_telefone = string.

database
     agenda(tipo_nome, tipo_telefone).
```

A partir da definição de domínio e banco de dados acima, adicione entradas na agenda com o asserta e use todos os outros predicados relacionados com bancos de dados apresentados.

Exercício 25: O programa abaixo implementa uma agenda de telefones com uma interface com o usuário através de menus de texto simples. Ele guarda um nome e um número para cada entrada no banco de dados.

```
/*
 * EE - 097
 * Exercicio 25.
 */

domains

     tipo_nome, tipo_telefone = string.

database
     agenda(tipo_nome, tipo_telefone).

predicates
     menu (integer)./* gerencia a interacao com o usuario.*/
     processa (integer)./* processa a escolha do usuario. */
     erro.    /* processa erro na escolha da opcao do menu. */
     adiciona.         /* adiciona novo item na agenda */
     deleta.           /* retira item da agenda */
     procura.          /* procura item na agenda */
     carrega.          /* carrega agenda de um arquivo */
     salva.            /* salva a agenda em um arquivo. */


clauses
     /* Clausula que controla a interacao com o usuario. */

     menu(Escolha):- Escolha = 6,
                write ("Terminado!").
     menu(_):- nl,
```

```
        write("Escolha uma opcao: "),nl,
        write("1. Adiciona um telefone ao Banco de Dados"),
         nl,
        write("2. Deleta um telefone do Banco de Dados"),
         nl,
        write("3. Procura um telefone"),nl,
        write("4. Carrega o arquivo do Banco de Dados"),nl,
        write("5. Salva o Banco de Dados em um arquivo"),
         nl,
        write("6. Termina. "),nl,
        write(" "),nl,
        readint(Escolha),
        processa (Escolha), menu(Escolha).


/* Clausulas que processam a escolha do usuario. */
processa (1) :- adiciona.
processa (2) :- deleta.
processa (3) :- procura.
processa (4) :- carrega.
processa (5) :- salva.

/* se a escolha foi 6, confirma saida. */
processa (6) :- write ("Tem certeza que deseja terminar?
                (s/n)"),
            readln(Resposta),
            frontchar (Resposta,'s',_).

processa(6) :- menu(0).

/* processa entradas invalidas */
processa (Escolha) :- Escolha < 1, erro.
processa (Escolha) :- Escolha > 6, erro.

erro :- write ("Por favor entre numeros entre 1 e 6."),
         nl,
        write ("Tecle algo para continuar."),
         readchar (_).

/* Clausulas que implementam as rotinas de manipulacao do DB.*/

/* adiciona um telefone */
adiciona :- write ("Adicionando um telefone ao B D."),
            nl,
            write ("Entre o nome: "),
            readln (Nome),
            write ("Entre o telefone dele:"),
            readln (Telefone),
            assertz (agenda (Nome, Telefone)),
            write ("Telefone adicionado ao B D."),
            nl.


/* deleta um telefone do BD */
deleta :-   write ("Deletando um telefone do B D."),
```

```
                nl,
                 write ("Entre o nome: "),
                readln (Nome),
                write ("Entre o telefone dele: "),
                readln (Telefone),
                retract (agenda (Nome, Telefone)),
                write ("Telefone retirado do B D."),
                nl.

    deleta :-   write ("Telefone nao encontrado!!!"), nl.


    /* procura um telefone */
    procura :-  write ("Procurando um telefone no B D."),
                nl,
                write ("Entre o nome: "),
                readln (Nome),
                agenda (Nome, Telefone),
                write ("O telefone do(a) ", Nome, " e' ",Telefone,"."),
                nl.

    procura :-  write ("Telefone nao encontrado!!!"), nl.


    /* carrega o arquivo com o banco de dados */
    carrega :-  write ("Carregando o arquivo com o B D."),
                nl,
                write ("Qual o nome do arquivo? "),
                readln(Nome_do_arquivo),
                consult(Nome_do_arquivo),
                write("Carregado!"),
                nl.

    /* salva o arquivo com o banco de dados */
    salva :-    write ("Salvando o Banco de Dados."), nl,
                write ("Qual o nome do arquivo? "),
                readln(Nome_do_arquivo),
                save(Nome_do_arquivo),
                write("Salvo!"),
                nl.


/* Inicia o programa */
goal

    menu(0).
```

Verifique o funcionamento do programa, adicionando, procurando e retirando telefones. Modifique o programa para que a agenda contenha também o endereço da pessoa.

Exercício 26: Implemente uma regra que imprima todos os telefones guardados na agenda.

12. BIBLIOGRAFIA

Esta apostila é um resumo sobre a linguagem Prolog e, portanto, características importantes dessa linguagem não foram abordadas. O estudante pode aprofundar seus conhecimentos na bibliografia abaixo, da qual vários exemplos e definições foram retirados.

YIN, K. M; SOLOMON, D. **Using Turbo Prolog**. Indianápolis, Que Corporation, 1987.

ROBINSON, P. R. **Turbo Prolog: Guia do Usuário.** São Paulo, McGraw-Hill, 1988.

CLOCKSIN, W. F.; MELLISH, C. S. **Programming in Prolog.** 2a. ed., Berlin, Springer-Verlag, 1984.

RILLO, M. Notas de aula da disciplina Introdução à Inteligência Artificial. (PEE-817) Escola Politécnica - USP.

Qualquer sugestão, comentários ou dúvidas, favor contatar: Reinaldo Bianchi (rbianchi@pcs.usp.br).

Projetos

Inicialmente, projetos propostos por Clocksin e Mellish:

1. Escreva um programa que calcula o intervalo em dias entre duas datas, expressas na forma Dia/Mês, assumindo que eles se referem ao mesmo ano. Por exemplo, a questão "intervalo (3, marco, 7, abril, 35)." deve ser unificada com sucesso.

2. Concordância é uma lista de palavras que são encontradas em um texto, listadas em ordem alfabética, cada palavra seguida pelo número de vezes que ela apareceu no texto. Escreva um programa que produz uma concordância de uma lista de palavras representada como uma string do Prolog. (Dica: Strings são listas de códigos ASCII).

3. O problema das N-rainhas é amplamente discutido em textos de programação. Implemente um programa que encontre todas as maneiras de colocar 4 rainhas em um tabuleiro de xadrez de 4 por 4 em posições que nenhuma rainha consiga atacar a outra.

4. Escreva procedimentos para inverter e multiplicar matrizes.

5. Escreva um programa que compreenda sentenças simples em Português, que possuam a seguinte forma:

_____ é um(a) _____ .

Um(a) _____ é um(a) _____ .

É _____ um(a) _____ ?

O programa deve dar a resposta apropriada (sim, não, o.k. e desconheço), com base nas sentenças apresentadas. Por exemplo, temos abaixo uma execução do programa:

```
João é um homem.
o.k.
Um homem é uma pessoa.
o.k.
É João uma pessoa?
sim
É Maria uma pessoa?
Desconheço
```

Cada sentença deve ser transformada em uma cláusula do Prolog, que deve então ser inserida no banco de dados (com asserta) ou executada. As traduções dos exemplos anteriores deve ser:

```
homem(joão).
pessoa(X) :- homem(X).
? pessoa (joão).
? pessoa (maria).
```

Esse exercício é um exemplo simples de programas para compreensão de Linguagem Natural.

Abaixo, projetos retirados de outras fontes:

1.Escreva os procedimentos para ordenar os elementos de uma lista por Quicksort, Bubllesort e Shellsort.

2.Escreva um programa que resolva o Problema das Jarras D'água.

3.Crie uma base de dados sobre livros, usando objetos compostos.

4.Implemente uma Rede Neuronal Associativa.

5. Crie um pequeno sistema especialista, que ajude uma pessoa a fazer uma escolha de compra. Por exemplo, um programa que ajude uma pessoa a encontrar uma cidade para viajar de férias, fazendo perguntas simples como "Você gosta de praia? (Sim/Não)", "Você gosta de sair a noite? (Sim/Não)", etc.

6.Crie um programa simples que utilize aprendizado, como o programa "Animais". Este programa tenta adivinhar em qual animal o usuário está pensando e, caso não consiga descobrir pois desconhece o animal, aprende a diferença entre o novo animal e os que ele conhece.