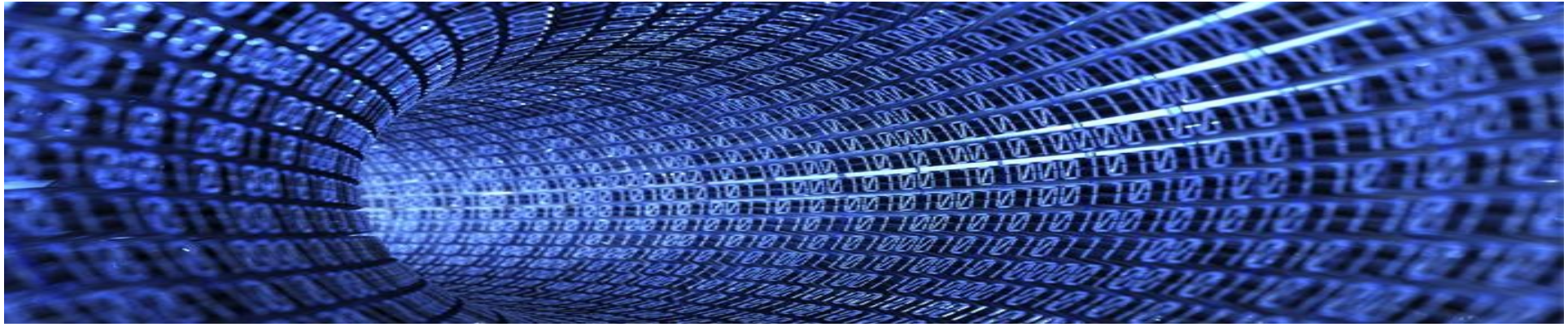
INSTITUTO MAUÁ DE TECNOLOGIA

MAUÁ

Curso de Engenharia de Computação
Linguagens Formais, Autômatos e Compiladores

Lógica e Prolog



Slides da disciplina ECM253 – Linguagens Formais, Autômatos e Compiladores
Curso de Engenharia de Computação
Instituto Mauá de Tecnologia – Escola de Engenharia Mauá
Prof. Marco Antonio Furlan de Souza

Prolog

- **Definição**
  - Na **lógica** de **predicados**, utilizam-se **regras** de **inferência** para se chegar a **teses** a **partir** das **hipóteses**;
  - Se uma **tese** tiver sido **demonstrada** como **consequência** de determinada **hipótese**, então, em uma **interpretação** na qual a **hipótese** seja verdadeira, a **tese** também será **verdadeira**.
  - A **linguagem** de programação **Prolog**, que significa ***PROgramming in LOGic***, também **ajuda** a **chegar** a **teses** a partir das **hipóteses**.
  - A **linguagem** inclui **predicados**, **conectivos lógicos** e **regras** de **inferência**. Ela permite a **descrição** de uma **interpretação**, ou melhor, de **hipóteses verdadeiras** em uma **interpretação**.

Prolog

- **Características**
  - As linguagens de programação mais conhecidas são **linguagens procedurais**. Nelas, o **programador**, portanto, **instrui** o **computador como resolver** um problema;
  - Já o **Prolog** é uma **linguagem declarativa** (também chamada de **linguagem descritiva**);
  - Um **programa Prolog** consiste em **declarações ou descrições** sobre uma **interpretação**, isto é, quais as **hipóteses** que são **verdadeiras** em uma **interpretação**. Este **conjunto** de **declarações** é também **chamado** de **base de dados** do **Prolog**;
  - Para **determinar se** uma dada **hipótese**, é ou não **verdadeira**, o usuário posta-a na forma de uma **pergunta** e, então, o **Prolog** usa sua **base** de **dados** e **aplica** suas **regras** de **inferências** (sem a necessidade de qualquer instrução por parte do programador).

Prolog

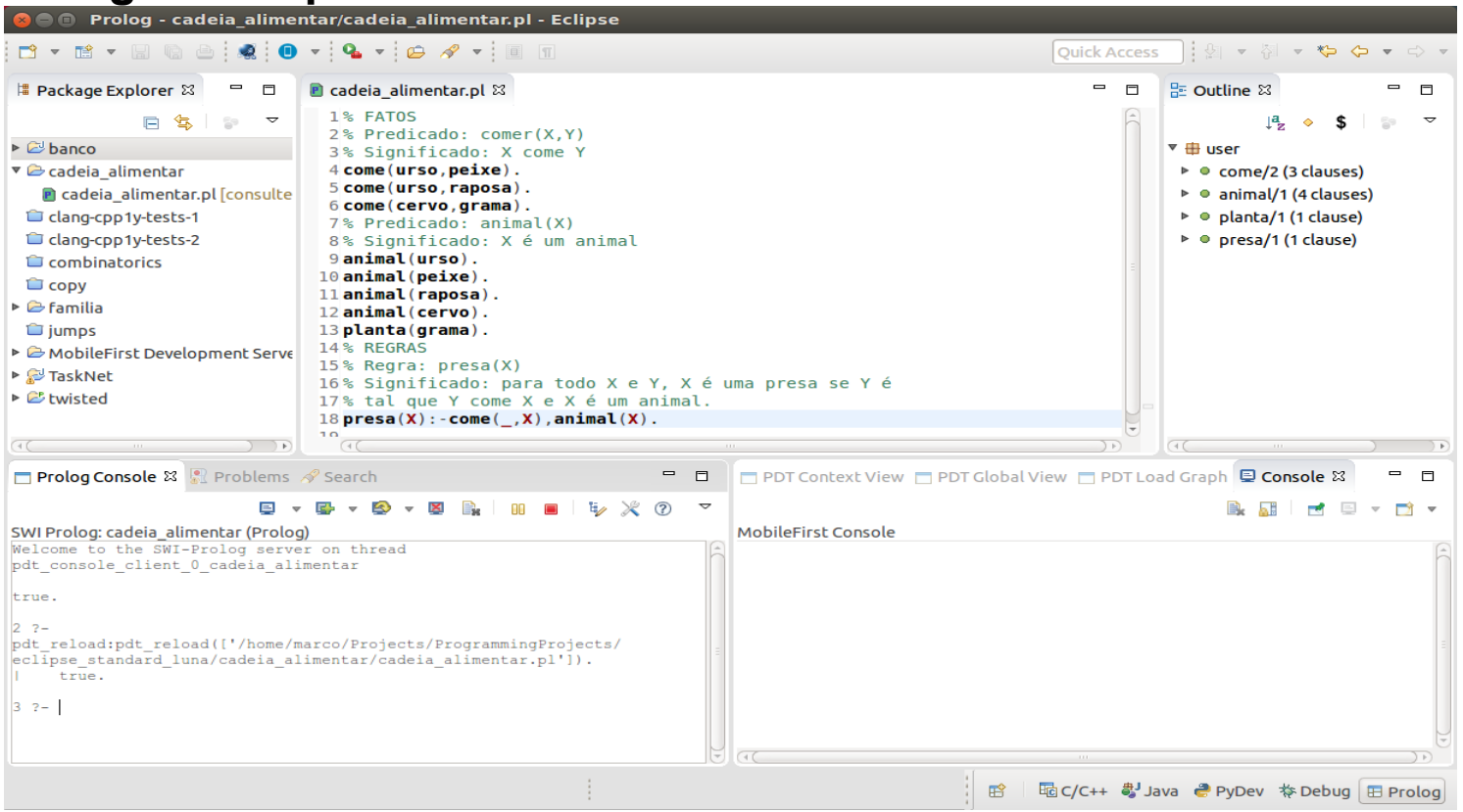
- **Interpretadores**
  - **SWI Prolog**
    - É a opção de código aberto e gratuito mais utilizada;
    - Existe desde 1987;
    - http://www.swi-prolog.org/
- **SWI Prolog no Eclipse**
  - **Prolog Development Tool** – PDT (mais completo!)
    - http://sewiki.iai.uni-bonn.de/research/pdt/docs/start
  - **Prolog Development Tools** – ProDT
    - http://prodevtools.sourceforge.net/

Prolog

- **SWI Prolog no Eclipse**
  - Instalar o plugin **PDT** (veja demonstração em sala);
  - **Utilização**
    - **Alterar** para a **perspectiva** “Prolog”;
    - Garantir que o **console** do **Prolog** está **visível** (Window → Show View → Prolog Console);
    - **Criar** um novo processo do Prolog: no console do Prolog, clicar sobre o botão com símbolo “+” verde;
    - **Criar** um **projeto genérico** do **Eclipse**: File → New → Project… → General → Project e depois **escolher** um nome (exemplo: “cadeia_alimentar”);
    - Clicar com o botão direito sobre o projeto criado e então criar um arquivo com extensão “.pl”. Por exemplo, `cadeia_alimentar.pl`;
    - Carregar este arquivo no interpretado: F9 ou botão direito sobre o arquivo no “Package Explorer” e então “Prolog Development Tools → (Re)consult”;
    - Na janela do console, digite suas consultas.

Prolog

- **SWI Prolog no Eclipse**



Prolog

- **Conceitos**
  - **Itens**
    - Os **itens** em uma **base** de **dados Prolog** pode ser de **dois tipos**:
      - **Fatos**: definem predicados;
      - **Regras**: é um tipo de fato envolvendo predicados e conectivos.
    - A **tradução** de **elementos** da **lógica** de **predicados** para **Prolog** pode ser realizada de acordo com a tabela seguir:

Lógica	Prolog
Constante	Cadeia em minúsculas
Variável	Cadeia em maiúsculas
Relação	Notação funcional
Conectivo E	Vírgula
se α **então** β	β :- α

Prolog

- **Exemplo**
  - **Fatos** sobre uma **cadeia alimentar**:
    - Declaram “coisas” que são sempre verdadeiras;
    - São escritos como **predicados** (ou **relações**) sobre um domínio de interpretação, terminados por ponto (“.”).

```
% FATOS
% Predicado: come(X,Y)
% Significado: X come Y
come(urso,peixe).
come(urso,raposa).
come(cervo,grama).
% Predicado: animal(X)
% Significado: X é um animal
animal(urso).
animal(peixe).
animal(raposa).
animal(cervo).
planta(grama).
```

Prolog

- **Exemplo**
  - **Regras** sobre uma **cadeia alimentar**:
    - Declaram “coisas” que são **verdadeiras dependendo** de uma dada **condição** – possuem variáveis que generalizam situações (quantificador universal).
    - São escritas no **formato** “`cabeça:-corpo.`”,onde `corpo` é uma lista de **metas** separadas por vírgula (“`,`”) – **conjunção**.

```
% REGRAS
% Regra: presa(X)
% Significado: para todo X e Y, X é uma presa se Y é
% tal que Y come X e X é um animal.
presa(X):-come(Y,X),animal(X).
```

    - **Outra versão**: já que `Y` aparece uma única vez na fórmula, pode-se utilizar uma variável anônima , “`_`”:

```
presa(X):-come(_,X),animal(X).
```

Prolog

- **Exemplo**
  - **Consultas** sobre uma **cadeia alimentar**:
    - São expressões passadas ao Prolog para que ele tente prová-las, resultando em uma “verdade” se isto for possível ou “falsidade” caso contrário.
    - **Exemplos**:
      - “O que o urso come?”

```
?- come(urso,X).
X = peixe ;
X = raposa.
```

      - “Quem come grama?”

```
?- come(X,grama).
X = cervo.
```

      - “Grama é um animal?”

```
?- animal(grama).
false.
```

Prolog

- **SWI Prolog em linha de comando**
  - Em um terminal, digitar: `swipl`
  - Aparecerá o *prompt* a seguir:

```
marco@NEPTUNE:~$ swipl
Welcome to SWI-Prolog (Multi-threaded, 64 bits, Version 7.1.32)
Copyright (c) 1990-2014 University of Amsterdam, VU Amsterdam
SWI-Prolog comes with ABSOLUTELY NO WARRANTY. This is free software,
and you are welcome to redistribute it under certain conditions.
Please visit http://www.swi-prolog.org for details.

For help, use ?- help(Topic). or ?- apropos(Word).

?-
```

  - Duas formas de trabalho:
    - Iterativo, no *prompt*;
    - Escrever um arquivo com a base, carregá-la e depois consultá-la (mais adequado).

Prolog

- **SWI Prolog em linha de comando**
  - Por exemplo, com um editor que suporte sintaxe destacada do Prolog (**Notepad++**, **gedit**, **geany**, **kate**), escrever a seguinte base de dados para o problema da cadeia alimentar e **salvar como** `/home/aluno/cadeia_alimentar.pl`:

```
% FATOS
% Predicado: comer(X,Y)
% Significado: X come Y
come(urso,peixe).
come(urso,raposa).
come(cervo,grama).
% Predicado: animal(X)
% Significado: X é um animal
animal(urso).
animal(peixe).
animal(raposa).
animal(cervo).
planta(grama).
% REGRAS
% Regra: presa(X)
% Significado: para todo X e Y, X é uma presa se Y é
% tal que Y come X e X é um animal.
presa(X):-come(_,X),animal(X).
```

Prolog

- **SWI Prolog em linha de comando**
  - Dentro do interpretador Prolog, **alterar** seu **diretório** corrente para aquele em que se encontra sua base de dados:

```
?- working_directory(_,'/home/aluno').
true.
```

  - Em seguia, **consultar** o **arquivo** (i.e. pedir para o Prolog carregá-lo na memória). Faça isso toda vez que alterar a base:

```
?- [cadeia_alimentar].
true.
```

    - Não é necessário informar a extensão.
  - Por fim, **fazer consultas** que quiser ("`;`" traz mais respostas):

```
?- come(urso,Y).
Y = peixe ;
Y = raposa.
```

  - Para **terminar** o Prolog:

```
?- halt.
```

Prolog

- **Inferências em Prolog**
  - Em **lógica** de **predicados**, a regra:
    - "Para x e y, se y come x e x é um animal, então x é uma presa."
  - Poderia ser assim **formulada**:

$$E(y, x) \wedge A(x) \rightarrow P(x)$$

  - Mas em **Prolog**, **todas** as **variáveis** são **universalmente quantificadas**, então esta fórmula é **entendida** como:

$$(\forall y)(\forall x)[E(y, x) \wedge A(x) \rightarrow P(x)]$$

Prolog

- **Inferências em Prolog**
  - Primeiramente, **Prolog** representa **fbfs** no formato de **cláusulas** de **Horn**:
    - É uma **fbf** composta de **predicados** ou **negações** de **predicados agregados** por **disjunções**, onde no **máximo um predicado não** deve ser **negado**;
    - Então a **fórmula**:

$$E(y, x) \land A(x) \rightarrow P(x)$$

    - Pode ser **reescrita** como **cláusulas** de **Horn** aplicando a **equivalência** da **implicação** e depois **DeMorgan**:

$$\neg E(y, x) \lor \neg A(x) \lor P(x)$$

Prolog

- **Inferências em Prolog**
  - Prolog utiliza uma **única regra** de **inferência**, denominada **resolução**;
  - O **princípio** é **simples**: com cláusulas de Horner, uma cláusula que contém um certo predicado, $A(a)$ por exemplo, pode "casar" com outra cláusula que contém sua negação, $\neg A(a)$:
  - De:

    $$A(a)$$

    $$\neg A(a) \vee B(b)$$
  - Conclui-se:

    $$B(b)$$
  - Que é a mesma conclusão de:

    $$A(a) \wedge (A(a) \rightarrow B(b))$$

Prolog

- **Inferências em Prolog**
  - Por exemplo, a consulta para saber que animais são presas:

```
presa(X).
```

  - Faz com que o Prolog **pesquise** sua base de **dados** por alguma **regra** com o **predicado** como **consequente** e encontra:

```
presa(X):-come(_,X),animal(X).
```

  - Então a **pesquisa segue procurando** por **outras cláusulas** que possam ser **resolvidas** com **esta** cláusula. A primeira é:

```
come(urso,peixe).
```

  - Esta unifica com o predicado `come(_,X)` (que em Horner é uma negação), anterior para o valor de `X` com a constante `peixe`. Daí, é necessário, ainda, provar `animal(X)`, que é obtido com:

```
animal(peixe).
```

Prolog

- **Inferências em Prolog**
  - Prolog **responde** à consulta assim:

```
?- presa(X).
X = peixe
```

  - Mas ainda podem haver **outras respostas**. Ao se **teclar** "*;*" no console, o Prolog realiza um processo de ***backtrack***, voltando à **pesquisa** de **outras soluções** possíveis:

```
4 ?- presa(X).
X = peixe ;
X = raposa ;
false.
```

  - Prolog utiliza, portanto, um **processo** de **busca primeiro em profundidade** (lembram?).

Prolog

- **Inferências em Prolog**
  - **Definições recursivas**
    - São regras em que um predicado é definido em termos dele próprio;
    - Por exemplo, um **predicado** que **atesta** que um **dois animais** são **parte** de uma **cadeia alimentar**, pode ser assim definido:

```
% Regra: na_cadeia_alimentar(X,Y)
% Significado: para todo X, Y e Z, X e Y fazem parte de uma cadeia
% se X come Y ou se X come Z e Z e Y fazem parte da cadeia.
na_cadeia_alimentar(X,Y):-come(X,Y).
na_cadeia_alimentar(X,Y):-come(X,Z),na_cadeia_alimentar(Z,Y).
```

    - Acrescentar a cláusula e **`come(peixe,alga).`** então perguntar quem faz parte da cadeia da alga:

```
?- na_cadeia_alimentar(X, alga).
```

Prolog

- **Depuração**
  - A **depuração** de programas Prolog pode ser feita com o **predicado** `trace`, **antes** do **objetivo** que se deseja depurar;
  - **No console**

```
?- trace, come(urso,X).
   Call: (7) come(urso, _G2669) ? creep
   Exit: (7) come(urso, peixe) ? creep
X = peixe ;
   Redo: (7) come(urso, _G2669) ? creep
   Exit: (7) come(urso, raposa) ? creep
X = raposa.
```

  - Para sair deste modo, digitar o predicado `nodebug`.

```
[trace]  ?- nodebug.
true.

?-
```

Prolog

- **Exercícios**
  - Elaborar em Prolog uma base de dados que contenha as seguintes regras:
    1. **Se** `X` é pai de `A` e `X` é pai de `B` e `A` e `B` são distintos, **então** `A` e `B` são irmãos. Em Prolog, para dizer que `A` é diferente de `B`: `A\=B`.
    2. **Se** `X` e `Y` são irmãos e se `X` é pai de `A` e `Y` é pai de `B`, então `A` e `B` são primos.
  - E os seguintes fatos:
    - Mary é um pai de Jane.
    - Jane é pai de Karen.
    - John é pai de Jim.
    - Mary é pai de John.
    - Jane é pai de Bill.
  - Pede-se: descobrir quem são primos.

Prolog

- **Sintaxe**
  - Um **programa** em **Prolog** é construído a partir de **termos** que podem ser:
    - **Constantes**
    - **Variáveis**
    - **Estruturas**

Prolog

- **Constantes**
  - **Nomeiam objetos específicos** ou **relações** específicas;
  - **Tipos** de constantes
    - **Átomos**, que podem ser:
      - **Nomes** de **objetos**: são cadeias de símbolos **iniciados** por **letra minúscula** e que podem conter letras, números e o símbolo "_" no seu interior. Se escrito entre "'" e "'" pode conter qualquer símbolo em seu interior (ex.: `maria`, `sensorA`, `nivel_tensao`, `'Alarme disparado'`)
      - **Símbolos especiais**: utilizados como símbolos reservados ou operadores (ex.: `=`, `:-`);
    - **Números**:escritos sob a forma convencional: `-2`, `3.14`, `6.02E23` etc.

Prolog

- **Variáveis**
  - São cadeias de símbolos iniciados por **maiúsculas** e que podem conter letras, números e o símbolo “_” no seu interior;
  - Variáveis são nomes que **antes** da **execução** do programa **não referenciam nenhum nome particular**, mas que **poderão fazê-lo** no **decorrer** da **execução** do programa;
  - Quando se necessita de uma variável, mas não se necessita nomeá-la, pode-se fazer uso da **variável anônima**, _:
  - Por **exemplo**, para saber se existe algum **vizinho** da **cidade** 'Santo André', mas **sem saber qual** é (são), pode-se escrever:

```
?- vizinho(_,'Santo André').⏎
true.
```

Prolog

- **Estruturas**
  - Uma **estrutura** é um **objeto composto** por uma coleção de outros **objetos** denominados de **componentes**;
  - Esses **componentes podem**, ainda, **ser** outras **estruturas** e assim por diante;
  - Estruturas **auxiliam** na **organização** do **programa** – permitem **agrupar informações** relacionadas e tratá-las como um objeto **único**;
  - Uma **estrutura** é **definida** por seu nome geral, o ***functor***, e por seus **componentes** escritos **dentro** de **parênteses**;
  - **Exemplo**:

```
triangulo( ponto(4,2), ponto(6,4), ponto(7,1) ).
```

functors

Functors são utilizados para definir fatos e predicados!

Prolog

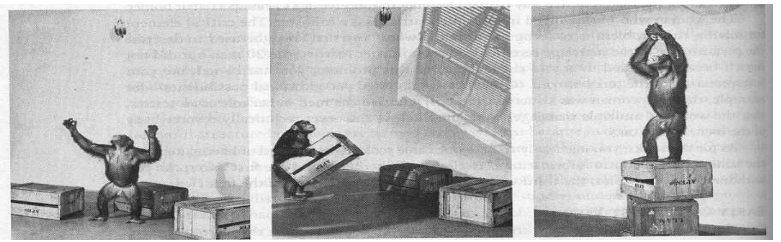
- **Estruturas**
  - **Exemplo**:

```
% Segmentos
segmento( ponto(1,1), ponto(5,1) ).
segmento( ponto(2,1), ponto(3,2) ).

% Versões para testar segmentos horizontais
eh_segmento_horizontal1( S ):-
    S = segmento(P1, P2),
    P1 = ponto(_, Y),
    P2 = ponto(_, Y).

eh_segmento_horizontal2( S ):-
    S = segmento( ponto(_, Y), ponto(_, Y) ).
```

Prolog



- **Estruturas**
  - **Exemplo**: problema do macaco e a banana

```
% move(State1, Move, State2): Move in State1
results in State2
% state(HorizPosition,VertPosition,
BoxPosition, HasBanana): configures
%    the current state of the problem
% canget( State ): indicates how the monkey can
get the banana
move(state(middle,onbox,middle,hasnot),
     grasp,
     state(middle,onbox,middle,has)).

move(state(P,onfloor,P,H),
     climb,
     state(P,onbox,P,H)).

move(state(P1,onfloor,P1,H),
     push(P1,P2),
     state(P2,onfloor,P2,H)).
```

```
move(state(P1,onfloor,B,H),
     walk(P1,P2),
     state(P2,onfloor,B,H)).

canget(state(_,_,_,has)).
canget(State1):-
    move(State1,Move,State2),
    canget(State2).
%
canget(state(atdoor,onfloor,atwindow,hasnot)).
```

Prolog

- **Operadores**
  - É conveniente que alguns ***functors*** sejam **escritos** no formato de **operadores**;
  - Isso ocorre com **expressões aritméticas**. Assim, as duas **expressões** a seguir são **idênticas** em Prolog:
    - `x+y*z` (operador)
    - `+(x,*(y,z))` (functor)
  - Salvo poucas exceções (`,`, `[]` e `{}`), Prolog permite redefinir seus operadores e também criar novos outros.
  - **Nota:** `3 + 4` não é para ser entendido como a aplicação do operador "`+`" sobre os números `3` e `4` resultando no número `7`. É para ser entendido como uma estrutura `+(3,4)` que pode ser interpretada posteriormente pelo predicado `is`.

Prolog

- **Posição dos operadores**
  - **Operador infixo**
    - É aquele que é **escrito entre** seus **argumentos**. Exemplos: `+`, `-`, `*`, `/`.
  - **Operador prefixo**
    - É aquele que é escrito antes de seus argumentos. Exemplo: `-` para representar negação.
  - **Operador pós-fixo**
    - É aquele que escrito após seus argumentos.

Prolog

- **Precedência e associatividade dos operadores**
  - Cada **operador** do Prolog pertence à uma **classe** de **precedência** – identificada por um **número inteiro** entre 0 e 1200 (SWI Prolog), sendo que a **maior precedência** se dá para **números menores**;
  - A **associatividade** se dá por meio de **regras** que baseadas no **tipo** do **operador** em questão: `xf`, `yf`, `xfx`, `xfy`, `yfx`, `fy` ou `fx`:
    - A letra "`f`" **indica** a **posição** do **functor**, enquanto que as letras "`x`" e "`y`" indicam a **posição** dos **argumentos**;
    - A letra "`y`" deve ser **interpretada** como "nesta posição deve ocorrer um **termo** com **precedência menor** ou **igual** ao **functor**";
    - De modo similar, a letra "`x`" representa a ocorrência de um **termo** de precedência **estritamente menor** que a do functor.

Prolog

- **Criação/redefinição de novos operadores**
  - Utiliza-se a **diretiva** `op`.
  - **Exemplo**:

```
:-op(500,xfx,was).
:-op(400,xfx,of).
:-op(300,fx,the).

diana was the secretary of the department.
```

  - **Testes**:

```
?- diana was What.⏎
What = the secretary of the department.
?- diana was the secretary of Where.⏎
Where = the department.
?- diana was What of Where.⏎
What = the secretary,
Where = the department.
```

Prolog

- **Igualdade e unificação**
  - O **operador infixo** (ou **predicado**) de **igualdade** é assim utilizado:

```
?- X = Y.
```

  - Nesta consulta, o Prolog executa um **processo** de **unificação**, que **consiste** em **tornar** `X` e `Y` **iguais**, assim:
    - Se `X` é uma **variável não instanciada** e `Y` é **instanciada** a **qualquer termo**, então `X` torna-se **instanciada** ao que `Y` está **instanciada**;
    - **Inteiros** e **átomos** são sempre iguais a eles próprios;
    - Duas **estruturas** são **iguais** se possuem o **mesmos functor** e **número** de **componentes** e os **componentes correspondentes** são **iguais**.

Prolog

- **Igualdade e unificação**
  - O **operador infixo** (ou **predicado**) para testar uma **desigualdade** é assim utilizado:

```
?- X \= Y.
```

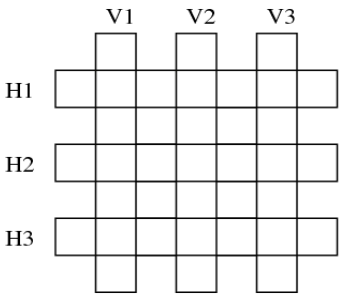
  - Este operador retorna **verdadeiro** se a **meta** em **questão** não **puder** ser **provada**;
  - **Exemplos** dos operadores de igualdade:

```
?- a(b, C, d(e, F, g(h, i, J))) = a(B, c, d(E, f, g(H, i, j))).⏎
C = c,
F = f,
J = j,
B = b,
E = e,
H = h.

?- pai(joao,maria) \= pai(maria,joao).⏎
true.
```

Prolog

- **Exercício**
  - São fornecidas seis palavras em italiano: `astante`, `astoria`, `baratto`, `cobalto`, `pistola`, `statale`. Elas devem ser arranjadas como em um jogo de palavras-cruzadas na grade a seguir a partir da base de dados a seguir (dicionário):




```
word(astante, a,s,t,a,n,t,e).
word(astoria, a,s,t,o,r,i,a).
word(baratto, b,a,r,a,t,t,o).
word(cobalto, c,o,b,a,l,t,o).
word(pistola, p,i,s,t,o,l,a).
word(statale, s,t,a,t,a,l,e).
```

  - Escrever um predicado denominado `crossword/6` (6-ário) que preencherá a grade apresentada. Os primeiros três argumentos do predicado devem ser as palavras verticais, da esquerda para a direita e os últimos três argumentos devem ser as palavras horizontais de cima para baixo.

Prolog

- **Conjunção e disjunção de metas**
  - Em Prolog, é **muito comum** que para uma **meta** a ser alcançada seja **necessário alcançar** uma **conjunção** de **submetas**:

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

  - O **operador** "**,**" representa o **operador** de **conjunção** (**AND**) para **metas**. Prolog também aceita **disjunção** de **metas** (**OR**), se for necessário, com o operador "**;**":

```
extenso(Numero,Texto):-
    Numero = 1, Texto = 'um';
    Numero = 2, Texto = 'dois';
    Numero = 3, Texto = 'três'.
```

Reescrever o predicado acima sem utilizar o operador ";"

Prolog

- **Aritmética**
  - **Predicados predefinidos** para **comparar** dois números:

Expressão	Significado
`X=:=Y`	`X` e `Y` referem-se ao mesmo número
`X=\=Y`	`X` e `Y` referem-se a números diferentes
`X<Y`	`X` é menor que `Y`
`X>Y`	`X` é maior que `Y`
`X=<Y`	`X` é menor ou igual a `Y`
`X>=Y`	`X` é maior ou igual a `Y`

  - **Não há** um **operador** para **atribuição**: utiliza-se o **operador infixo** `is` para **unificar** uma **expressão aritmética** avaliada como **segundo operando** à uma **variável** do **lado esquerdo**. **Exemplo**:

```
?- X is 7/4.⏎
X = 1.75.
```

Prolog

- **Aritmética**
  - **Predicados predefinidos** para **operações aritméticas**:

Expressão	Significado
`X+Y`	Soma de `X` e `Y`
`X-Y`	Diferença entre `X` e `Y`
`X*Y`	Produto entre `X e Y`
`X/Y`	Quociente de `X` dividido por `Y`
`X//Y`	Quociente inteiro de `X` dividido por `Y`
`X mod Y`	Resto da divisão de `X` dividido por `Y`

Prolog

- **Aritmética**
  - **Exemplo**:

```
%Fatos sobre populações de países (milhões de habitantes)
populacao(eua, 318).
populacao(brasil, 200).
populacao(china, 1357).
populacao(india, 1252).
%Fatos sobre áreas de países (milhões de km2)
area(eua, 9.8).
area(brasil, 8.5).
area(china, 9.5).
area(india, 3.3).
%Regra para densidade
densidade(X,Y):-
    populacao(X,P),
    area(X,A),
    Y is P/A.
```

Prolog

- **Listas**
  - Uma **lista** em Prolog é definida por uma **sequência** de **zero** ou **mais itens** dentro de "`[`" e "`]`";
  - Uma **lista vazia** é escrita como `[]`;
  - Uma lista pode ser decomposta como a concatenação de um termo que inicia a lista (**cabeça**) seguido do restante da lista (**cauda**), separado pelo operador "`|`"
  - Exemplos:

```
?- [a,c,d,e]=[X|Y].⏎
X = a,
Y = [c, d, e].
```

Prolog

- **Listas**
  - **Exemplos**
    - Pertinência

```
% X pertence à L se X é cabeça da lista L
member(X,[X|_]).
% Ou se X pertence à cauda da lista L
member(X,[_|Tail]):-
    member(X,Tail).
```

    - Concatenação

```
% Uma lista vazia concatenada com uma lista L resulta em L
conc([],L,L).
% Uma lista L que possua X como cabeça concatenada com outra
% resulta em uma lista concatenada com X como cabeça
conc([X|L1],L2,[X|L3]):-
conc(L1,L2,L3).
```

    - Como definir `member` a partir de `conc`?

Prolog

- **Listas**
  - **Exemplos**
    - Adição de elementos

```
% X é um elemento que adicionado à frente da lista L
% resulta na lista [X|L]
add(X, L, [X|L]).
```

    - Remoção de elementos

```
% Se X é a cabeça de uma lista, então o resultado de remover
% X desta lista é sua cauda.
del(X,[X|Tail],Tail).
% Se X estiver na cauda, então ele deve ser eliminado de lá.
del(X,[Head|Tail],[Head|TailWithoutX]):-
    del(X,Tail,TailWithoutX).
```

    - Inserção de elementos

```
% Dizer que inserir X à uma lista torna-a maior é o mesmo que
% dizer que remover X da lista maior resulta na lista original.
insert(X,List,BiggerList):-
    del(X,BiggerList,List).
```

Prolog

- **O operador `!`**
  - Sistema de ***backtracking*** do **Prolog** pode, muitas vezes, **gastar tempo** na **busca** de **soluções** que nitidamente **não contribuirão** com o resultado;
  - O **operador** `!` (“**cut**”) **fixa** com as **mudanças** feitas desde que a **meta** tenha sido **unificada** com o **lado esquerdo** da **cláusula** que contém este operador, “cortando” a possibilidade de *backtracking* em outros caminhos.
  - Exemplo, **testar** no programa a seguir `max(2,3)` com `trace`, com e sem `!`:

```
max(X,Y,Y):-
    X =< Y,!.
max(X,Y,X):-
    X > Y.
```

Prolog

- Exemplo de um sistema especialista simples

```
:-op(800,fx,if).
:-op(700,xfx,then).
:-op(300,xfy,or).
:-op(200,xfy,and).

is_true(P):-
     fact(P).

is_true(P):-
     if Condition then P,
     is_true(Condition).

is_true(P1 and P2):-
     is_true(P1),
     is_true(P2).

is_true(P1 or P2):-
     is_true(P1);
     is_true(P2).
```

Prolog

- Exemplo de um sistema especialista simples (cont.)

```
if hall_wet and kitchen_dry then
    leak_in_bathroom.

if hall_wet and bathroom_dry then
    problem_in_kitchen.

if window_closed or no_rain then
    no_water_from_outside.

forward :-
    new_derived_fact(P),
    !,
    write( 'Derived: '), write( P), nl,
    fact(P),
    forward
    ;
    write( 'No more facts').
```

Prolog

- Exemplo de um sistema especialista simples (cont.)

```
new_derived_fact(Concl):-
    if Cond then Concl,
     \+fact(Concl),
     composed_fact(Cond).

composed_fact(Cond):-
     fact( Cond).

composed_fact(Cond1 and Cond2):-
   composed_fact(Cond1),
   composed_fact(Cond2).

composed_fact(Cond1 or Cond2):-
   composed_fact(Cond1);
   composed_fact(Cond2).

% Some facts...
fact(hall_wet).
fact(window_closed).
fact(bathroom_dry).
% Execute: ?- forward.
```

Referências Bibliográficas

- BRATKO, I. **PROLOG Programming for Artificial Intelligence**. 3. ed. Harlow: Addison-Wesley, 2001.
- GERSTING, J.L. **Fundamentos matemáticos para a ciência da computação**. 4. ed. Rio de Janeiro, RJ: LTC, 2001. 538 p. ISBN 85-216-1263-X.