LINGUAGEM PROLOG

SANDRA CORTINOVI

PROLOG é uma linguagem de programação simples, embora poderosa, fundamentada na lógica simbólica. Foi desenvolvida na Universidade de Marselha, França, com o intuito de ser uma ferramenta prática para programação em lógica. De forma semelhante a LISP, PROLOG é uma linguagem interativa projetada primeiramente para processamento de dados simbólicos. Ambas as linguagens são ferramentas para desenvolvimento de aplicações da área de Inteligência Artificial, estando baseadas em sistemas matemáticos formais: LISP, baseada no cálculo lambda, é tipicamente usada para definição de funções; PROLOG, baseada em um subconjunto poderoso da lógica clássica, é usada para definição de relações. A linguagem LISP pura, de fato, pode ser vista como uma especialização da linguagem PROLOG.

A linguagem PROLOG é baseada em um provador de teoremas para cláusulas de Horn. A estratégia específica utilizada é uma forma restrita de resolução linear de entrada.

1. Linguagem PROLOG básica

PROLOG é uma linguagem de programação usada para solucionar problemas que envolvem objetos e relações entre objetos.

Programas em PROLOG consistem de:

- declaração de alguns fatos sobre objetos e seus relacionamentos;
- definição de regras sobre os objetos e seus relacionamentos;
- resposta a consultas sobre objetos e seus relacionamentos.

A programação em PROLOG permite ignorar a maioria dos detalhes referentes a como um programa é executado. A linguagem PROLOG foi projetada de forma a permitir que o utilizador forneça comandos sobre alguma coisa que seja logicamente verdadeira de modo que o interpretador PROLOG possa tirar conclusões; não requer que se especifique *como* um programa deve ser executado, mas que se especifique *com o que* as soluções de um problema se parecem.

PROLOG provêm de PROgramação em LÓGica. Em lógica, define-se, a grosso modo, um teorema e se pesquisa fatos e regras para verificar a validade do teorema. De forma análoga à lógica, PROLOG é usado para expressar fatos e relacionamentos entre esses fatos, e para inferir soluções para problemas.

Um fato expressa alguma verdade sobre um relacionamento. Por exemplo,

SÓCRATES É HOMEM.

é um fato que define que o indivíduo *SÓCRATES* pertence à classe *HOMEM*.

Uma regra, por outro lado, expressa um relacionamento entre fatos. Um relacionamento em uma regra é verdadeiro se os outros relacionamentos nessa regra também o são. Por exemplo,

SE ALGUÉM É UM HOMEM, ESSE ALGUÉM É MORTAL.

é uma regra que define que se um indivíduo pertence à classe *HOMEM*, esse indivíduo também pertence à classe *MORTAL*.

Uma inferência possível a partir desse fato e dessa regra é que *SÓCRATES É MORTAL*. Em PROLOG, esse fato e essa regra são expressos da seguinte forma:

```
homem(socrates).
mortal(X):-homem(X).
```

e a consulta: *É SÓCRATES UM MORTAL* é expressa e respondida da forma que segue:

```
?-mortal(socrates).
yes
```

1.1. Objetos manipulados pela linguagem PROLOG

PROLOG é uma linguagem de programação orientada para o processamento de símbolos. Símbolos não permitem operações convencionais; só permitem montar estruturas de símbolos. Os objetos manipulados pelos programas PROLOG são chamados termos. Termos podem ser <u>átomos</u> ou <u>estruturas</u>.

1.1.1. Átomos

Átomos podem ser constantes ou variáveis.

<u>Constantes</u> podem ser:

- identificadores que iniciam por letras minúsculas.

Exemplos: brasil, mt1, get_item

- cadeias de caracteres entre apóstrofes.

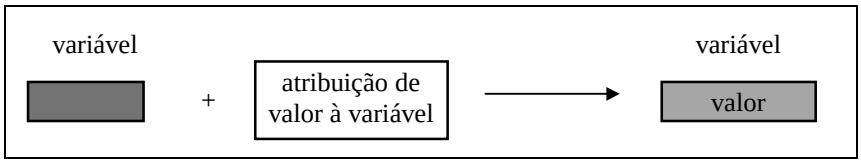
Exemplos: 'Socrates', '4º andar'

- números inteiros.

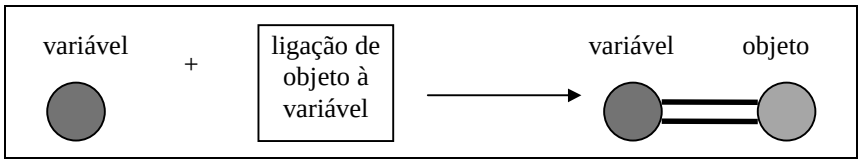
Exemplos: 146, 20

<u>Variáveis</u> assemelham-se a identificadores, exceto que começam com uma letra maiúscula ou um símbolo sublinha (_). Uma variável pode ser vista como representando algum objeto que não se é capaz de nomear.

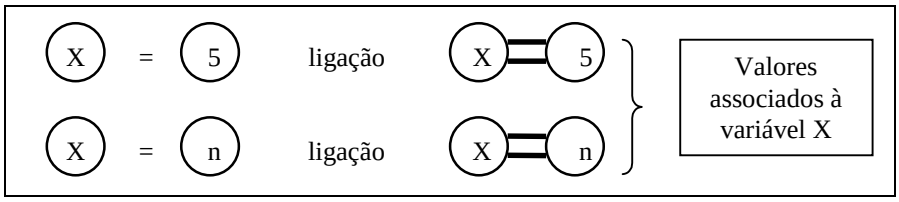
Em linguagens convencionais uma variável representa uma célula de memória.



Em PROLOG uma variável é associada a um objeto.



A vantagem da ligação sobre a atribuição está no fato de que qualquer tipo de objeto pode ser ligado a uma mesma variável; não se tem restrições às ligações, pois uma variável não possui estrutura interna.



Em PROLOG a ligação de uma variável a um termo (objeto) é chamada <u>substituição</u>. Diz-se que a variável é substituída pelo termo, pois onde a variável é usada tudo se passa como se o termo estivesse sendo usado. Portanto, uma variável pode estar sendo usada, simultaneamente, em vários pontos de um programa, mesmo que não esteja ligada a qualquer objeto. Quando essa ligação é estabelecida, ela repercute em todos os pontos em que a variável está sendo usada, tudo se passando a partir daí como se o objeto ligado à variável estivesse sendo usado naqueles pontos desde o início.

1.1.2. Estruturas

Uma estrutura é um objeto que possui organização interna. Em PROLOG estruturas são representadas por

símbolo funcional(lista de argumentos)

onde ***símbolo funcional*** é um identificador e ***lista de argumentos*** é uma lista de termos que pode ser vazia; isto é, um átomo é um caso particular de estrutura.

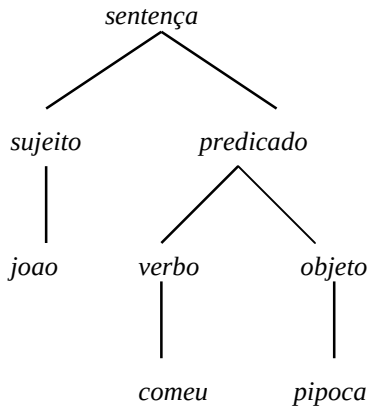
Exemplos:

livro(gabriela,autor(amado,jorge))
cidade(sao-paulo, pais(brasil))

Em PROLOG, os termos estruturados são vistos como árvores. Por exemplo, a estrutura

sentença(sujeito(joao),
predicado(verbo,comeu), objeto(pipoca)))

equivale à seguinte árvore:



1.2. Base de dados PROLOG

Usa-se o termo base de dados quando se tem um conjunto de fatos e regras, chamados genericamente de cláusulas, que são usados para solucionar um problema específico.

1.2.1. Fatos

Um fato é uma relação entre objetos da forma

relação(objeto$_1$, ..., objeto$_n$).

Os nomes dos objetos que encontram-se entre parênteses são chamados argumentos. O nome da relação é chamado predicado.

Exemplo de relação:

naturalidade(joao,cidade(sao-paulo,pais(brasil))).
naturalidade(jean,cidade(paris,pais(França))).
naturalidade(john,cidade(londres,pais(inglaterra))).

Uma relação é um conjunto de n-uplas de objetos. Cada comando PROLOG é chamado cláusula. Os comandos que definem uma base de dados são chamados fatos porque representam o fato de que uma determinada n-upla pertence a uma certa relação.

1.2.2. Consultas

Uma vez criada, uma base de dados PROLOG pode ser consultada. Uma consulta é uma pergunta ao sistema, o qual deve respondê-la conforme a natureza da pergunta. Se ela for fechada, isto é, não contiver variáveis, deve ser entendida como uma solicitação para verificar a veracidade de um fato, ou seja, se uma determinada n-upla pertence realmente a uma dada relação. Neste caso a resposta do sistema deve ser da forma *sim/não*.

Se a consulta for aberta, isto é, se contiver variáveis, deve ser entendida como uma solicitação para listar todos os valores que, substituídos nas variáveis, tornam um fato verdadeiro - fazem sua correspondente n-upla pertencer a uma dada relação. Neste caso, a resposta do sistema deve ser uma lista de substituições da forma

variável = valor

Uma consulta a uma base de dados PROLOG é feita na forma

?- predicado(argumentos).

Por exemplo, dada a base de dados

```
naturalidade(joao,cidade(sao-paulo,pais(brasil))).
naturalidade(jean,cidade(paris,pais(França))).
naturalidade(john,cidade(londres,pais(inglaterra))).
possui(andre,livro(gabriela,autor(amado,jorge))).
possui(andre,livro(zoeira,autor(verissimo,luis))).
possui(julia,livro(o-continente,autor(verissimo,erico))).
empregado(nr(1027),nome(silva,jose),dep(vendas)).
empregado(nr(1028),nome(silva,julio),dep(trocas)).
```

observe a forma como o PROLOG responde às seguintes consultas.

```
?- naturalidade(jean,X).
X=cidade(paris,pais(França))
?-naturalidade(X,cidade(Y,pais(Z))).
X=joao          Y=sao paulo     Z=brasil
?-;
X=jean          Y=paris         Z=França
?-;
X=john          Y=londres       Z=inglaterra
?-;
no
?-possui(andre,livro(zoeira,autor(verissimo,luis))).
yes
?-possui(X,livro(Y,autor(verissimo,Z))).
X=andre     Y=zoeira        Z=luis
?-;
X=julia     Y=o-continente  Z=erico
?-empregado(nr(1028),N,D).
N=(silva,julio)  D=dep(trocas)
```

A resposta a uma consulta fechada é um valor lógico que indica se uma determinada n-upla pertence à relação. A resposta a uma consulta aberta é um valor lógico que indica se há pelo menos uma n-upla que se enquadra no esquema definido pela consulta; em caso afirmativo, a resposta vem acompanhada da extensão do subconjunto da relação o qual satisfaz a consulta. Há, portanto, para cada consulta, uma extensão de resposta possível. Isso leva naturalmente à idéia de intersecção de extensões para obtenção de respostas caracterizadas por meio de diversas condições, isto é, obtenção de elementos que pertençam a diversas relações simultaneamente.

Observação: O símbolo ***;*** significa ***ou***. Quando de uma consulta, o PROLOG busca o primeiro fato da base de dados que a satisfaz, fornecendo a resposta. Um ***;*** faz com que o PROLOG pesquise na base, a partir da última solução encontrada, um próximo fato que satisfaça a consulta. Não o encontrando responde ***no***.

Exemplo:

```
/* base de dados */
amigo(antonio,andre).
amigo(joao,jose).
amigo(maria,julia).
vizinho(andre,pedro).
vizinho(julia,rui).
vizinho(julia,pedro).
/* consulta */
?-amigo(X,Y),vizinho(Y,pedro).
X=antonio          Y=andre
;
X=maria            Y=julia
;
no
```

1.2.3. Unificação de termos

Chama-se substituição de variável à operação de associação de uma variável a um termo qualquer.

Uma variável está livre se não está associada a nenhum termo. Uma variável só pode ser substituída por um termo quando estiver livre.

Uma variável substituída por outra variável fica substituída pelo objeto associado a essa última. Se a segunda variável estiver livre, diz-se que ambas ficam substituídas, uma pela outra. Neste caso, as duas continuam livres após a substituição.

Uma variável substituída só pode sofrer nova substituição após ter sido liberada do termo a que estava associada.

A unificação é a operação pela qual dois termos são tornados únicos, isto é, com o mesmo valor. O valor de um termo é a sua estrutura simbólica. Dois termos possuem o mesmo valor quando possuem a mesma estrutura simbólica.

A unificação é definida do seguinte modo:
1. Dois átomos constantes são unificáveis se são iguais.
2. Uma variável livre é unificável com qualquer termo - a unificação se faz pela substituição da variável pelo termo.
3. Uma variável associada a um objeto é unificável com um termo se o objeto for unificável com o termo.
4. Um termo estruturado é unificável com outro termo estruturado se a cada componente de um corresponder um componente do outro, e os componentes correspondentes forem unificáveis.

O resultado da unificação é um termo que substitui os dois termos iniciais.

1.2.4. Regras

Em PROLOG, regras são usadas quando se deseja dizer que um fato depende de uma conjunção de outros fatos. Uma regra é um comando geral sobre objetos e seus relacionamentos, na forma

cabeça :- corpo.

Em PROLOG, uma regra consiste em uma cabeça e um corpo, conectados pelo símbolo ***:-***, o qual significa ***se***. A cabeça de uma regra descreve o fato que esta pretende definir. O corpo descreve a conjunção de objetivos que devem ser satisfeitos, um após o outro.

Em geral um predicado será definido por uma mistura de fatos e regras. Estas são chamadas cláusulas de um predicado. Usa-se o termo cláusula quando se refere a um fato ou uma regra.

Cabe ressaltar que uma regra é uma definição geral por permitir que uma variável represente um objeto diferente para cada uso da regra. O PROLOG deve ser capaz de dizer se uma variável X qualquer representa objetos distintos em cláusulas distintas - isto é resolvido através do conhecimento do escopo de uma variável.

Por exemplo, dada a base de dados:

```
/* fatos */
fica-em(sao-paulo,brasil).
fica-em(paris,França).
fica-em(grenoble,França).
nasceu-em(joao,sao-paulo).
nasceu-em(jean,paris).
nasceu-em(louis,grenoble)
/* regra */
patria-de(X,Y):-nasceu-em(X,Z),fica-em(Z,Y).
```

as seguintes consultas têm como respostas:

```
?-patria-de(louis,P).
P=França
?-patria-de(joao,brasil).
yes
```

A regra estabelece que *patria-de* é uma relação dependente das relações *nasceu-em* e *fica-em*, e mostra como os ítens da relação *patria-de* se relacionam com os ítens das outras relações.

Por exemplo, dada a base de dados

```
amigo(antonio,andre).
amigo(andre,juliana).
amigo(juliana,andre).
amigo(juliana,jose).
amigo(X,Y):-amigo(Y,X).
amigo(X,Z):-amigo(X,Y),amigo(Y,Z).
```

observa-se que com esta definição da relação *amigo* existe uma regra definindo-a como reflexiva e outra definindo-a como transitiva.

Por exemplo, na base de dados

```
mae(antonio,maria).
mae(pedro,maria).
mae(jose,ana).
pai(antonio,carlos).
pai(pedro,jose).
pai(eva,jose).
irmao(X,Y):-pai(X,Z),pai(Y,Z).
irmao(X,Y):-mae(X,Z),mae(Y,Z).
```

observa-se que a relação *irmao* é definida por duas cláusulas.

2. Mecanismo de inferência da linguagem PROLOG

A máquina que executa os programas escritos em PROLOG põe em funcionamento um mecanismo de inferência lógica para procurar a resposta a qualquer pergunta que for colocada. O mecanismo usado pela máquina PROLOG baseia-se no princípio da resolução, que é um método de demonstração automático de teoremas.

A máquina PROLOG normalmente é uma máquina virtual simulada por um programa executado em um computador clássico: um programa emulador da máquina PROLOG. Durante sua execução, o emulador da máquina PROLOG cria uma árvore que representa o espaço de procura do problema em questão. A técnica de procura usada é a de pesquisa em profundidade simples.

O diagrama da página seguinte representa o funcionamento de um emulador para uma máquina PROLOG.

2.1. Como PROLOG responde a perguntas

Quando de uma consulta, a máquina PROLOG pesquisa a base de dados procurando cláusulas que se unifiquem com a consulta. Dois fatos se unificam se seus predicados são os mesmos (escritos da mesma forma), e se seus argumentos correspondentes são os mesmos. PROLOG responderá *sim* se localizar um fato que se unifique com a consulta; se o fato consultado não existir na base de dados, PROLOG responderá *não*.

Quando uma consulta contém variáveis, a máquina PROLOG realiza um pesquisa em todas as cláusulas para localizar os objetos que as variáveis podem representar.

Quando PROLOG responde à consulta

?-gosta(joao,X).

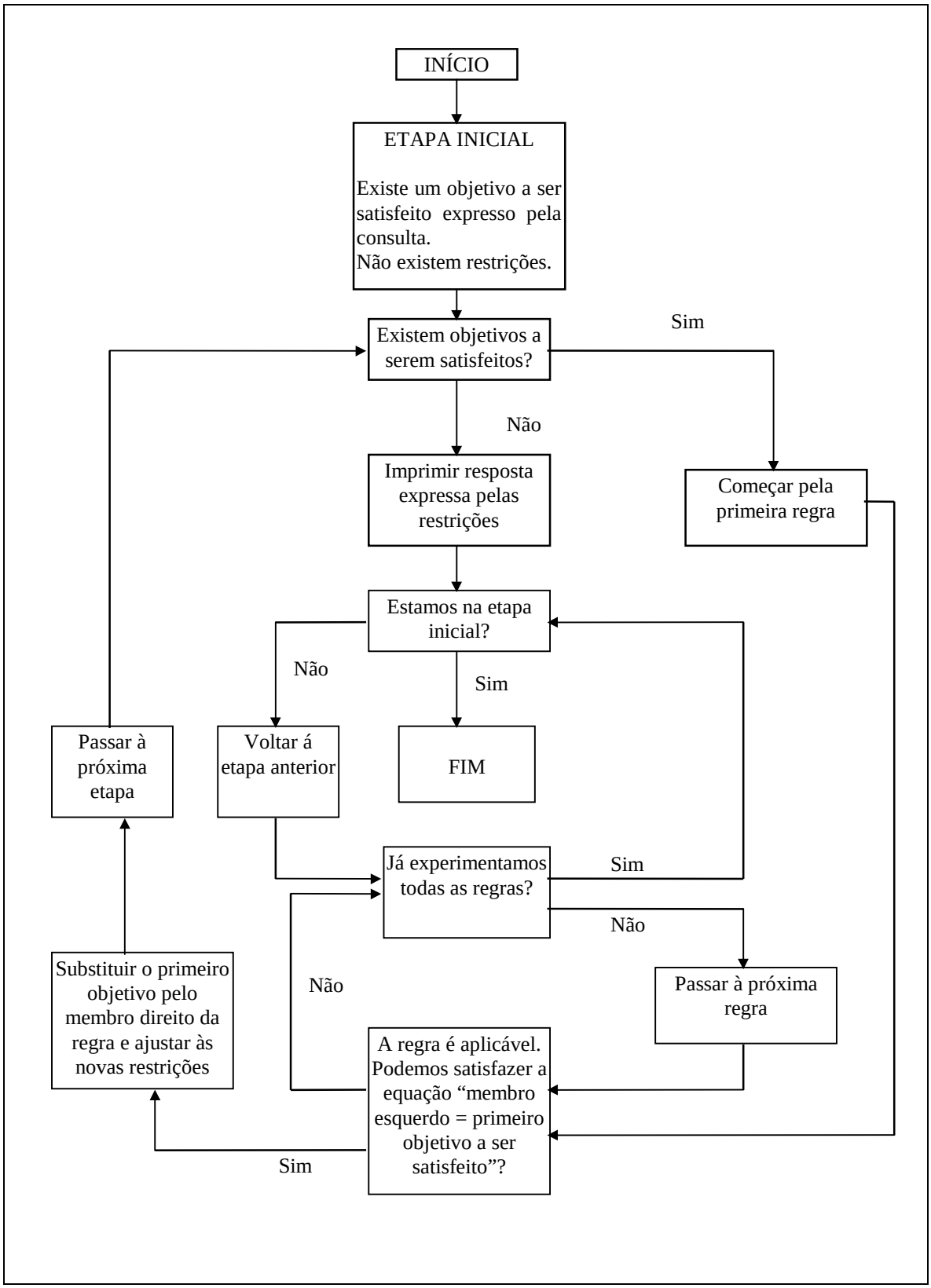
referente à base de dados

gosta(joao,flores).
gosta(joao,maria).
gosta(paulo,maria).

a variável *X* inicialmente não está instanciada, isto é, não representa nenhum objeto. O mecanismo de inferência da linguagem PROLOG pesquisa a base de dados em busca de uma cláusula que se unifique com a consulta; se uma variável não instanciada aparecer como argumento, a máquina PROLOG permitirá à variável associar-se com qualquer outro argumento na mesma posição da cláusula. Nesse caso específico, PROLOG pesquisa por qualquer cláusula cujo cabeçalho tenha como predicado *gosta* com primeiro argumento *joao*; o segundo argumento desse predicado pode ser qualquer coisa, pois a consulta contém uma variável não instanciada como segundo argumento. Quando uma cláusula dessas é encontrada, a variável *X* passa a representar o segundo argumento do predicado, qualquer que seja esse. A máquina PROLOG pesquisa a base de dados na ordem em que esta foi criada, sendo o fato *gosta(joao,flores)* localizado primeiro. A variável *X* então representará o objeto flores (diz-se que *X* está instanciada para *flores*), e PROLOG então marca o ponto na base de dados onde a unificação foi detectada.

Desde que a máquina PROLOG localize um fato que se unifique com a consulta, ela responde com o(s) objetos(s) que a(s) variável(is) representa(m); no exemplo anterior, somente a variável *X* associada ao objeto *flores*. Então o PROLOG espera por instruções adicionais: se o usuário transmitir um ponto-e-vírgula (;), o PROLOG retoma a pesquisa a partir do ponto na base de dados onde localizou a resposta anterior; se o usuário não transmitir um ponto-e-vírgula, que significa que está satisfeito com apenas uma resposta, o PROLOG interrompe a pesquisa. Quando o PROLOG inicia a pesquisa a partir de uma marca de reinício, ao invés do início da base de dados, diz-se que o PROLOG está tentando re-satisfazer a consulta.

O símbolo vírgula (,) corresponde à conjunção lógica, servindo para separar qualquer número de objetivos diferentes que devem ser satisfeitos a fim de responder a uma consulta. Quando uma seqüência de objetivos separados por vírgulas é fornecida, o PROLOG tenta satisfazer cada objetivo, e todos os objetivos devem ser satisfeitos para que a seqüência seja satisfeita.


INÍCIO
ETAPA INICIAL
Existe um objetivo a ser satisfeito expresso pela consulta.
Não existem restrições.
Existem objetivos a serem satisfeitos?
Sim
Não
Imprimir resposta expressa pelas restrições
Começar pela primeira regra
Estamos na etapa inicial?
Não
Sim
Passar à próxima etapa
Voltar á etapa anterior
FIM
Já experimentamos todas as regras?
Sim
Não
Substituir o primeiro objetivo pelo membro direito da regra e ajustar às novas restrições
Não
Passar à próxima regra
A regra é aplicável. Podemos satisfazer a equação "membro esquerdo = primeiro objetivo a ser satisfeito"?
Sim

Por exemplo, na consulta

?-gosta(paulo,X),gosta(joao,X).

o PROLOG procede da seguinte forma:

> *Se o primeiro objetivo for localizado na base de dados, então o PROLOG marca a cláusula que o satisfez e tenta satisfazer o segundo objetivo. Se o segundo objetivo for satisfeito, marca o ponto onde foi localizado e obtém uma solução que satisfaz ambos os objetivos. Se, entretanto, o segundo objetivo não for satisfeito, o PROLOG tentará re-satisfazer o primeiro objetivo, iniciando a pesquisa a partir da marca de parada desse objetivo.*

Nesse caso o PROLOG responderá:

X=maria

Se o usuário solicitar outra possibilidade de resposta, enviando

;

o PROLOG responderá *no*, significando que não existem outros fatos que satisfaçam a conjunção de objetivos expressa na consulta.

É importante ressaltar que:

- cada objetivo tem sua própria marca de reinício;
- o PROLOG pesquisa toda a base de dados para cada objetivo;
- se acontecer de uma cláusula da base de dados satisfazer um objetivo, o PROLOG marcará a localização dessa cláusula na base de dados para o caso de ter de resatisfazer esse objetivo; e
- quando um objetivo for re-satisfeito, o PROLOG iniciará a pesquisa a partir da marca de reinício do próprio objetivo, ao invés do início da base de dados.

2.2. Retrocesso

Retrocesso é o nome dado ao comportamento do mecanismo de inferência da linguagem PROLOG de tentar repetidamente satisfazer e re-satisfazer objetivos numa conjunção de objetivos.

Pode-se visualizar uma conjunção de objetivos como tendo os objetivos arranjados da esquerda para a direita, separados por vírgula. Cada objetivo pode ter um "vizinho" à esquerda e um "vizinho" à direita, exceto o objetivo na extrema esquerda (somente um "vizinho" à direita) e o objetivo na extrema direita (somente um "vizinho" à esquerda).

Quando manipula uma conjunção de objetivos, o PROLOG tenta satisfazer cada objetivo analisando-os da esquerda para a direita. Se um objetivo é satisfeito, o PROLOG deixa uma marca na cláusula (ponto da base de dados) que o satisfez. Deve-se entender essa marca como um apontador do objetivo para a cláusula onde se encontra a solução. Além disso, quaisquer variáveis previamente não instanciada tornam-se instanciadas. O PROLOG tenta então satisfazer seu objetivo "vizinho" à direita, partindo do inicio da base de dados.

Sempre que a tentativa de satisfação de um objetivo falhar, ou seja, não se possa detectar uma cláusula que se unifique com ele, o PROLOG volta atrás, torna não instanciadas as variáveis instanciadas pelo objetivo que falhou e tenta re-satisfazer o objetivo anterior ("vizinho" à esquerda), iniciando a pesquisa a partir da marca de reinício deste.

Se cada objetivo, após ter ingressado no seu "vizinho" à direita, não puder ser re-satisfeito, então as falhas causarão gradualmente o retorno a objetivos à esquerda a medida em cada objetivo falhar. Se o primeiro objetivo da seqüência de objetivos (objetivo na extrema esquerda da conjunção de objetivos) falhar, então a conjunção inteira falha, isto é, não pode ser satisfeita.

Em suma, retrocesso consiste em revisar o que tem sido feito, tentando re-satisfazer os objetivos através de formas alternativas para satisfazê-los. Além disso, se o usuário não está satisfeito com apenas uma resposta, pode iniciar um retrocesso digitando um ponto-e-vírgula quando o PROLOG informar a solução.

Dada a seguinte base de dados

```
mae(antonio, maria).                          /* fato1 */
mae(pedro, maria).                            /* fato2 */
mae(jose, ana).                               /* fato 3 */
pai(antonio,carlos).                          /* fato 4 */
pai(pedro,jose).                              /* fato5 */
pai(paulo,jose).                              /* fato6 */
irmao(X,Y):-pai(X,Z), pai(Y,Z), X\==Y.        /* regra1 */
irmao(X,Y):-mae(X,Z), mae(Y,Z), X\==Y.        /* regra2 */
```

a solução para a consulta

?-irmao(antonio,I)

é obtida através do processo explicado a seguir.

O PROLOG unifica o predicado *irmao(antonio,I)* contido na consulta com a "cabeça" da *regra1*, na forma

irmao(antonio,I):-pai(antonio,Z), pai(I,Z), antonio \== I.

Satisfazendo o primeiro objetivo dessa regra, o PROLOG encontra o *fato4*: *pai(antonio,carlos)*, associando o valor *carlos* à variável *Z*. Neste momento a *regra1* é vista da seguinte forma:

irmao(antonio,I):-pai(antonio,carlos),pai(I,carlos),antonio\==I.

O PROLOG pesquisa novamente desde o início da base de dados buscando um fato que satisfaça o objetivo *pai(I,carlos)*. Encontra o *fato4*: *pai(antonio,carlos)*, associando o valor *antonio* à variável *I*. A regra fica com a seguinte forma:

irmao(antonio,I):-pai(antonio,carlos),pai(I,carlos),antonio\==antonio.

Ao tentar satisfazer o terceiro e último objetivo da regra (*X\==I*), ocorre uma falha, pois *antonio* não é diferente de *antonio*. Com isso, o PROLOG tenta re-satisfazer o objetivo anterior *pai(I,carlos)*, pesquisando a base de dados a partir da posição imediatamente posterior ao *fato4*. Não detectando nenhum fato que se unifique com esse predicado, tenta re-satisfazer o primeiro objetivo *pai(antonio,Z)*, pesquisando a partir da posição imediatamente posterior ao *fato4*. Novamente ocorre uma falha, fazendo com que a *regra1* falhe.

O PROLOG busca então, a partir da posição seguinte à *regra1*, um predicado que se unifique com o da consulta *irmao(antonio,I)*, localizando a *regra2*, que assume a forma

irmao(antonio,I):-mae(antonio,Z),mae(I,Z),antonio\==I.

Satisfazendo o primeiro objetivo *mãe(antônio,Z)* o PROLOG localiza o *fato1*, associando a variável *Z* à *maria*.

O próximo objetivo a ser satisfeito é *mae(I,maria)*. O *fato1* é detectado, assumindo *I* o valor *antonio*. Ao ser avaliado o terceiro objetivo está na forma *antonio\==antonio*, que retorna o valor falso. Neste ponto ocorre um retrocesso, sendo que o PROLOG busca re-satisfazer o objetivo *mae(I,maria)*, detectando o *fato2* e associando o valor *pedro* à variável *I*. O terceiro ojetivo é novamente testado com a forma *antônio\==pedro*, devolvendo valor verdadeiro. A *regra2* é satisfeita e a resposta fornecida pelo PROLOG é *I=pedro*.

Observações:

- numa base de dados PROLOG o que estiver contido entre os símbolos /* e */ é considerado comentário; e
- o símbolo \== é um predicado de comparação que significa diferente: c *T1\==T2* é a comparação *T1 é diferente de T2.*

2.3. Ordem das cláusulas e objetivos

Como mostrado pelo exemplo anterior, a linguagem PROLOG é seqüencial: objetivos são pesquisados seqüencialmente, da esquerda para a direita, durante a verificação de uma regra; cláusulas são pesquisadas seqüencialmente, do início para o fim, durante a verificação de um objetivo.

PROLOG possui dois mecanismos principais de controle do processo de pesquisa: um mecanismo para controlar a pesquisa vertical (entre cláusulas) e outro para ontrolar a pesquisa horizontal (entre objetivos). Os dois mecanismos interagem fortemente, formando diversos esquemas de controle de ativação das cláusulas.

O mecanismo que controla a pesquisa vertical fundamenta-se na operação de unificação dos argumentos de uma chamada com os argumentos de uma cláusula a ser ativada. Pode ser visto como uma forma generalizada de ativação de procedimentos: ativação por unificação de padrões.

Dado o predicado *amigo(X,Y)* definido da seguinte forma

```
amigo(antonio,andre).
amigo(andre,juliana).
amigo(antonio,maria).
amigo(juliana,jose).
amigo(X,Y):-amigo(Y,X).
```

a ativação de uma entre as cinco cláusulas, quando de uma chamada, se faz em função do padrão (forma) dos argumentos: é a forma do par de argumentos que determina qual cláusula será ativada.

Uma chamada do tipo

```
?-amigo(X,Y)
```

produzirá como primeira resposta

```
X=antonio        Y=andre
```

Entretanto, se a base de dados desse exemplo estivesse arranjada da seguinte forma

```
amigo(X,Y):-amigo(Y,X).
amigo(antonio,andre).
amigo(andre,juliana).
amigo(antonio,maria).
amigo(juliana,jose).
```

a mesma consulta

```
?-amigo(X,Y)
```

faria com que o PROLOG ingressasse numa recursão incontrolável.

Um problema fundamental em definições recursivas é a recursão à esquerda. Isto surge quando uma regra causa a chamada de um objetivo que é essencialmente equivalente ao objetivo que levou ao uso da regra.

Não se assume que, apenas porque se forneceu todos os fatos e regras relevantes, o PROLOG sempre as encontrará. Deve-se ter em mente quando se escreve programas em PROLOG como o PROLOG pesquisa através da base de dados e quais variáveis serão instanciadas quando uma de suas regras é usada.

De uma forma geral, é uma boa idéia colocar fatos antes de regras sempre que possível. Algumas vezes as regras em uma determinada ordem serão úteis se forem usadas para resolver objetivos gerados de um modo, mas não se os objetivos forem gerados de outro.

Considerando-se a seguinte definição do predicado *fat*, que fornece o fatorial de um número

```
fat(N,F) :- N1 is N-1, fat(N1,F1), F is N*F1.
fat(0,1).
```

o programa entrará num “laço eterno”.

Uma definição melhor do predicado *fat* é a seguinte:

```
fat(0,1).
fat(N,F) :- N1 is N-1, fat(N1,F1), F is N*F1.
```

No entanto, o controle da pesquisa vertical pela forma dos argumentos não é suficiente para garantir a exclusão mútua de cláusulas alternativas. Assim, por exemplo, dada a última definição do fatorial de uma número, a chamada

```
?-fat(3,F).
```

produz uma primeira resposta

```
F=6
```

e, se lhe for permitido procurar novas respostas, o mecanismo de retrocesso forçará o programa a entrar em uma recursividade incontrolável, pois o argumento *N* passará a assumir valores negativos.

Portanto, o mecanismo de unificação de argumentos só é eficaz quando o controle da pesquisa vertical pode ser feito apenas com base na forma dos parâmetros. Porém, quando é preciso decidir com base na avaliação dos valores dos parâmetros, ele se torna insuficiente.

O controle da pesquisa horizontal é o controle da busca de valores alternativos para as substituições das variáveis de uma cláusula. Ele é, portanto, o controle do procedimento de retrocesso na pesquisa de cláusulas alternativas, durante a verificação de um objetivo.

3. Listas

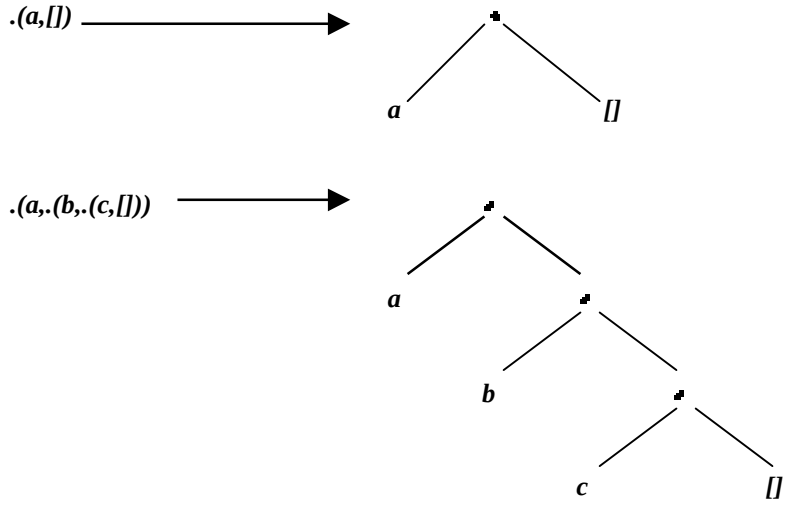
Uma lista é um termo estruturado que é operado como um conjunto ordenado de elementos. Os elementos podem ser átomos ou termos estruturados, inclusive listas.

Uma lista ou é uma lista vazia, não possuindo elementos, ou é uma estrutura com dois componentes: a cabeça e a cauda. O fim da lista é costumeiramente representado como uma cláusula que é composta pela lista vazia, a qual é expressa como **[]**. A cabeça e a cauda de uma lista são componentes do símbolo funcional "**.**".

Exemplos:

a) a lista contendo o único elemento ***a*** é ***.(a,[])***
a) a lista que consiste dos átomos **a**, ***b*** e ***c*** pode ser escrita ***.(a,.(b,.(c,[])))***

Listas sempre podem ser representadas por árvores binárias em que os nós intermediários são constiuídos pelos "**.**" e as folhas por átomos.



Listas são consideradas seqüências ordenadas, sendo a lista ***.(a,.(b,[]))*** diferente da lista ***.(b,.(a,[]))***.

A notaçao de lista consiste dos elementos da lista separados por vírgulas e a lista toda entre colchetes.

Exemplos: ***[]***
[a, b, c, [d, e, f], g]
[a, V1, b, [X, Y]]

Variáveis dentro de listas são tratadas da mesma forma que em qualquer outra estrutura. Podem tornar-se instanciadas a qualquer tempo, sendo que o uso sensato de variáveis pode fornecer uma forma de colocar "espaços" em listas que podem ser preenchidos posteriormente.

Listas são manipuladas por meio de sua divisão em uma cabeça e uma cauda. A cabeça de uma lista é o primeiro argumento do símbolo funcional "**.**", o qual é usado para construção de listas. A cauda de uma lista é o segundo argumento do símbolo funcional "**.** ", portanto é sempre uma lista.

Quando uma lista está na notação entre colchetes, a cabeça da lista é o primeiro elemento dessa e a cauda da lista consiste de quaisquer elementos exceto o primeiro.

Lista	Cabeça	Cauda
[a, b, c, d]	a	[b, c, d]
[a]	a	[]
[]	*falha*	*falha*
[[o, gato], caiu]	[o, gato]	[caiu]
[o, gato]	o	[gato]
[o, [gato, caiu]]	o	[[gato, caiu]]
[o, [gato,caiu], lá]	o	[[gato, caiu], lá]
[X+Y, x=y]	X + Y	[x = y]

A lista vazia não possui cabeça nem cauda

Uma operação comum sobre uma lista é dividi-la em sua cabeça e cauda. Há uma notação especial para representar *"a lista com cabeça X e cauda Y"*: **X|Y**.

Lista1	Lista2	Variáveis instanciadas quando da unificação de Lista1 com Lista2
[X, Y, Z]	[joão, come, peixe]	X=joão Y=come Z=peixe
[gato]	[X|Y]	X=gato Y=[]
[X, Y|Z]	[maria, bebe, vinho]	X=maria Y=bebe Z=[vinho]
[[o, Y]|Z]	[[X, lápis], [está, aqui]]	X=o Y=lápis Z=[[está, aqui]]
[golden|T]	[golden, norfolk]	T=norfolk
[vale, cavalo]	[cavalo, X]	*falha a unificação*
[branco, Q]	[P|cavalo]	P=branco Q=cavalo

A relação entre um objeto e uma lista que diz se este objeto é membro da lista é o predicado ***membro(X,L)***, que é verdaeiro se o objeto representado por ***X*** é um membro da lista representada por ***L***.

Existem duas condições que devem ser verificadas:

1) ***X*** é menbro da lista ***L*** se ***X*** é a cabeça da lista: ***membro(X, [X|_]).***

2) ***X*** é membro da lista ***L*** se ***X*** pertence à cauda da lista*:* ***membro(X,[_|Y]) :- membro(X,Y).***

Este dois predicados definem o predicado *membro* de uma lista, informando ao PROLOG para pesquisar a lista do início até o fim, procurando um determinado elemento. O ponto mais importante a lembrar, quando um predicado é definido recursivamente, é a procura por condições de limite do caso recursivo. Para o predicado *membro* há duas condições de limite, já que o objetivo procurado ou está na lista ou não está :

1. O membro é reconhecido pela primeira cláusula, que faz com que a pesquisa seja interrompida se o primeiro argumento do predicado membro unificar-se com a cabeça do segundo agumento.
2. O segundo argumento do predicado membro é a lista vazia (o primeiro argumento não é membro da lista).

Quanto ao caso recursivo (segunda cláusula) tem-se o seguinte: em algum ponto ou há uma unificação com a primeira cláusula ou o predicado terá a lista vazia como segundo argumento. Quando qualquer um desses fatos ocorre a recursividade dos objetivos *membro* termina. A primeira condição de limite é reconhecida por um fato, o qual não causa consideração de quaisquer subobjetivos adicionais. A segunda condição de limite não é reconhecida por qualquer cláusula do predicado *membro*, sendo que este falha.

Exemplos:

```
membro(X,[X|_]).
membro(X,[_|Y]):-membro(X,Y).
?-membro(d,[a,b,c,d,e,f,g]).
yes
?-membro(2,[3, a, 4, f]).
no
```

É importante ressaltar que cada vez que o predicado membro tem chamada a sua segunda cláusula, a linguagem PROLOG trata cada invocação como uma cópia diferente.

Observação: Os predicados apresentados nesta seção utilizaram diversas vezes variáveis anônimas (denotadas por um único símbolo sublinha) em suas cláusulas. Uma variável anônima é usada naqueles momentos em que se necessita usar uma variável, mas o objeto a ela associado não precisa se conhecido. Por exemplo, na primeira cláusula do predicado membro
membro(X,[X|_]).
somente a cabeça da lista que constitui o segundo argumento desse predicado é relevante; os objetos que compõem a cauda da lista não interessam nesta situação.

4. Controle do retrocesso

Recapitulando o que foi apresentado anteriormente sobre o que pode ocorrer a um objetivo numa conjunção de objetivos, tem-se:

1º) Uma tentativa pode ser feita para satisfazer um objetivo. Quando o PROLOG tenta satisfazer um objetivo, pesquisa a base de dados a partir de seu início, existindo duas situações possíveis de acontecer:

a) Um fato, ou cabeça de regra, unifica-se com o objetivo. O PROLOG marca o lugar desse fato ou regra na base de dados e instancia quaisquer variáveis previamente não instanciadas que tenham sido unificadas. Se a unificação se dá com uma regra, o PROLOG deve primeiro tentar satisfazer todos os objetivos que compõem a regra. Se o primeiro objetivo é satisfeito, então o PROLOG tenta satisfazer o próximo objetivo à direita da conjunção de objetivos.

b) Nenhum fato ou cabeça de regra que seja unificável com o objetivo é encontrado. Neste caso, diz-se que ocorreu uma falha. O PROLOG então tenta re-satisfazer o objetivo anterior à esquerda. É o que se chama retrocesso.

2º) O PROLOG pode tentar re-satisfazer um objetivo. Neste caso deve tornar não instanciadas quaisquer variáveis que se tornaram instanciadas quando este objetivo foi anteriormente satisfeito. Este é o significado de *desfazer* todo o trabalho feito previamente neste objetivo. A seguir o PROLOG retoma a pesquisa na base de dados a partir do lugar onde a marca de reinício do objetivo foi colocada. Como visto anteriormente, este novo objetivo *retrocedido* pode ter sucesso ou falhar, sendo que ou a situação em (a) ou a situação em (b) acima ocorrerá.

4.1. Corte

O corte é um mecanismo especial usado em programas PROLOG que permite informar ao sistema quais escolhas prévias não necessitam ser consideradas novamente quando de um retrocesso.

Sintaticamente, o uso do operador de corte em uma regra assemelha-se ao de um objetivo que tenha como predicado o símbolo **!** e nenhum argumento. A forma geral de uma cláusula que contém um operador de corte é a seguinte:

cabeça :- objetivo$_1$, ..., !, ..., objetivo$_n$.

sendo que o número de objetivos à direita e à esquerda do corte pode ser nulo.

Como objetivo, o operador de corte sempre sucede e não pode ser re-satisfeito. O seu efeito é remover as marcas de reinício de certos objetivos de forma que eles não possam ser re-satisfeitos.

Quando um operador de corte é encontrado como um objetivo de uma regra, o PROLOG imediatamente *congela* todas as escolhas feitas desde que o objetivo-pai foi invocado - por objetivo-pai entende-se o objetivo que causou o uso da regra que contém o corte. Todas as outras escolhas alternativas são descartadas. Dessa forma, quaisquer tentativas de re-satisfazer um objetivo entre o objetivo-pai e o corte falhará.

Há diversas formas de descrever o que acontece com as escolhas que são afetadas pelo corte. Pode-se dizer que as escolhas são congeladas, que o sistema se detém nas escolhas feitas ou que as alternativas são descartadas. Ou pode-se também ver o símbolo de corte como sendo uma barreira que separa objetivos.

O operador de corte, portanto, cancela as substituições feitas antes de ele ser ativado. Ele possibilita a pesquisa exaustiva de todas as alternativas de substituições para os objetivos que estão a sua direita. Porém, um retrocesso que tente buscar valores alternativos para o corte produz o retorno da chamada da cláusula como falha, o que significa que não há mais alternativas a procurar: os objetivos à direita do corte foram esgotados e os à esquerda do corte foram *congelados*.

Assim, o programa

```
trocar(eu,voce).
trocar(bebo,bebe).
trocar(X,X).
alterar([],[]).
alterar([P|R],[M|N]):-trocar(P,M),alterar(R,N).
```

produz as seguintes respostas à consulta:

```
?-alterar([eu,bebo,cerveja],F).
F=[voce,bebe,cerveja]
?-;
F=[voce,bebo,cerveja]
?-;
F=[eu,bebe,cerveja]
?-;
F=[eu,bebo,cerveja]
?-;
no
```

Tal ocorre porque o retrocesso sobre *trocar(P,M)* não está controlado, possibilitando o uso de todas as alternativas de substituição. Se for necessário fixar apenas uma alternativa, é preciso cortar o retrocesso sobre o predicado *trocar*. Esse corte pode ser obtido reescrevendo a regra da seguinte forma:

```
alterar([],[]).
alterar([P|R],[M|N]):-trocar(P,M),!,alterar(R,N).
```

A consulta, agora, produz apenas um resposta

```
?-alterar([eu,bebo,cerveja],F).
F=[voce,bebe,cerveja]
?-;
no
```

pois todas as tentativas de obter novas substituições para *M* são evitadas pela presença do corte.

4.2. Padrões de uso do corte

De forma geral, as cláusulas que contém o operador de corte podem ser interpretadas de três maneiras distintas, uma para cada modo de utilização deste operador:

a) o corte para implementar exclusão mútua incondicional entre cláusulas;
b) o corte para implementar exclusão mútua condicional entre cláusulas; e
c) o corte para produzir resposta única.

Na primeira forma de uso, exclusão mútua incondicional, a cláusula não contém as condições à esquerda do corte, ficando com a forma

cabeça:-!, objetivo$_2$, ..., objetivo$_n$.

Por exemplo, o programa de alteração de sentenças pode ser reescrito como:

```
trocar(eu,voce):-!.
trocar(bebo,bebe:-!.
alterar([],[]).
alterar([P|R],[M|N]):-trocar(P,M),alterar(R,N).
```

Nesse exemplo o corte implementa exclusão mútua incondicional entre as cláusulas do predicado *trocar*, e o programa só fornece uma resposta para cada consulta.

Na exclusão mútua condicional, as cláusulas tem a forma geral plena

cabeça :- objetivo$_1$, ..., !, ..., objetivo$_n$.

onde os objetivos à direita do corte podem ou não estar presentes.

O programa a seguir, que retorna o fatorial de um número, ilustra a exclusão mútua condicional.

```
fat(N,F):- N=0, !, F=1.
fat(N,F) :- N1 is N-1, fat(N1,F1), F is N*F1.
```

Na implementação de resposta única, os objetivos à direita do corte não existem, havendo apenas objetivos à esquerda do corte.

cabeça :- objetivo$_1$, ..., objetivo$_n$, !.

O programa a seguir exemplifica a implementação de resposta única.

```
primo(joao,jose).
primo(joao,julio).
primo(jaco,julio).
primo(jair,jorge).
tem-primo(X):-primo(X,Y),!.
```

4.3. Ciclo de repetição e falha

O predicado ***repeat*** constitui-se numa forma adicional de gerar múltiplas soluções através de retrocesso. Na maioria dos sistemas PROLOG encontra-se embutido, isto é, está pré-definido na implementação da linguagem, na forma:

```
repeat.              /*C1*/
repeat:-repeat.      /*C2*/
```

Qual é o efeito de colocar-se o predicado *repeat* como um objetivo em uma regra? Primeiro, o objetivo sempre sucede devido ao fato *C1*. Segundo, se o retrocesso retorna a este ponto novamente, o PROLOG será capaz de tentar uma alternativa, a regra presente em *C2*. Quando usa essa regra, outro objetivo *repeat* é gerado. Desde que esse unifica-se com o fato em *C1*, sucede novamente. Se por retrocesso voltar a este ponto, o PROLOG novamente usará a regra onde usa o fato anterior. Para satisfazer o objetivo extra, novamente tomará o fato como primeira opção. E assim sucessivamente. Portanto, o objetivo *repeat* será capaz de suceder muitas vezes por retrocesso. Cabe observar a importância da ordem da informação na definição do *repeat*: fato antes da regra.

Por que gerar objetivos que sempre sucederão no retrocesso? A razão é que permitem construir, fora de regras que não permitem alternativas de retrocesso nelas próprias, regras que permitem alternativas, podendo-se fazê-las gerar valores diferentes de cada vez.

O predicado pré-definido ***fail*** sempre falha. Um dos usos desse predicado é quando se deseja explicitamente que outro objetivo retroceda através de todas as soluções possíveis.

O ciclo ***repeat-fail*** assume a forma geral:

r:-repeat, q, (c;fail).

Essa combinação permite que um predicado **q** seja continuamente executado até que uma condição **c** se torne verdadeira.

Exemplo:

le-predicados:-repeat, read(P), (P='fim', !; analisa-predicado(P), fail).

4.4. Combinação de corte e falha

O operador de corte pode ser combinado com o predicado fail para produzir uma falha forçada. Uma conjunção de objetivos da forma

cabeça :- objetivo$_1$, ..., objetivo$_n$, !, fail.

é usada para informar ao PROLOG: *"se a execução chegou até esse ponto, então pode abandonar a tentativa de satisfazer essa regra"*. A conjunção falha devido ao fail, e o objetivo-pai falha devido ao corte.

A versão do programa do fatorial a seguir ilustra o emprego da falha forçada.

```
fat(N,F) :- N<0, !, fail.
fat(0,1) :- !.
fat(N,F) :- N1 is N-1, fat(N1,F1), F is N*F1.
```

Com a combinação de corte e falha implementa-se em PROLOG a negação lógica

```
not(X) :- X, !, fail.
not(X).
```

que tem a particularidade de ser uma negação definida pela impossibilidade se se provar *X*.

Com o predicado ***not***, a consulta

```
?-not(primo(jaco,jair)).
```

sobre a seguinte base de dados

```
primo(joao,jose).
primo(joao,julio).
primo(jaco,julio).
primo(jair,jorge).
tem-primo(X):-primo(X,Y),!.
```

produz a resposta *yes*.

5. Predicados Pré-definidos

O ambiente/linguagem Prolog apresenta um conjunto de predicados pré-definidos para entrada e sáida, manipulação da base da dados, manipulação de termos, operações aritméticas, meta-lógica, controle do retrocesso, depuração e gerência da área de trabalho.

A seguir apresenta-se alguns predicados que se encontram pré-definidos na maioria das implementações da linguagem Prolog.

5.1 Entrada e Saída

Há quatro tipos de objetos que podem ser lidos ou escritos: termos PROLOG, caracteres ASCII, *strings* e dados formatados. Cada tipo de objeto é manipulado por seus próprios predicados de entrada e saida.

A seguir apresenta-se alguns dos predicados para entrada e saída disponíveis no Arity Prolog.

5.1.1 Entrada e saida de termos

Os predicados de entrada e saida de termos lêem e escrevem termo Prolog válido, como átomos, variáveis, inteiros e cláusulas. Tanto a entrada como a saida de termos obedecem às seguintes normas:

- quando se lê um termo do teclado, o ponto (.) e o sinal de transmissão marcam o fim do termo;
- átomos que contenham espaços ou iniciam por letra maiúscula devem ser colocados entre apóstrofes; e
- quando um programa escreve um termo, variáveis não instanciadas são escritas como um sublinha seguido de um número hexadecimal que é a localização da variável.

read(Termo)	Lê um termo a partir do dispositivo padrão de entrada.
write(Termo)	Escreve um termo no dispositivo padrão de saida.
display(Termo)	Escreve um termo no dispositivo padrão de saida na forma prefixada.

Observação: Os predicados **read**, **write** e **display** não podem ser re-satisfeitos.

---

op(Precedência, Associatividade, Valor)

Define um operador, determinando como podem ser traduzidas espressões que envolvam o operador. Quando uma expressão é lida, a definição do operador determina como ela será traduzida na notação prefixada. Quando a expressão é escrita, determina como pode ser traduzida de volta para a notação convencional.

Posição determina a ordem na qual o operador aparece em relação ao seus operandos; pode ser:

prefixada ⇒ xfx
posfixada ⇒ fx
infixada ⇒ xf

Precedência determina a ordem na qual os operadores são traduzidos - qual operador é o mais externo na notação prefixada quando parênteses não estão presentes na expressão. Por exemplo, como o operador de multiplicação possui precedência menor que o operador de adição, a expressão

*a*b+c*

é traduzida como

+((a,b),c)*

de forma que o operador de adição torna-se o operador principal, e o termo *(a,b) o seu primeiro argumento.

Associatividade determina a ordem na qual operadores de igual precedência são traduzidos, podendo ser da esquerda para a direita, da direita para a esquerda ou não associativa (esta última significando que operadores não associativos não podem aparecer na mesma expressão).

Posição	Átomo	Associatividade
Infixada	xfx xfy yfx	não associativa direita para a esquerda esquerda para a direita
Prefixada	fx fy	não associativa esquerda para a direita
Posfixada	xf yf	não associativa direita para a esquerda

Observações:

- As regras de precedência e associatividade definidas para um operador podem sempre ser alteradas através do uso de parênteses.
- A precedência de um operador é definida através de um número entre 1 e 1200.
- Expressões envolvendo operadores de precedência igual ou superior a 1000 só podem ser usadas como argumento de um símbolo funcional se estiverem entre parênteses.
- Associar a um operador a precedência 0 implica em desabilitá-lo.

A tabela a seguir apresenta a definição de alguns operadores pré-definidos no Arity Prolog.

Operador	Associatividade	Precedência
:-	**xfx**	**1200**
:- ?-	**fx**	**1200**
;	**xfy**	**1100**
,	**xfy**	**1000**
\+	**fy**	**900**
=	**xfy**	**700**
is =.. \= == \==	**xfx**	**700**
< =< > >=	**yfx**	**600**
+ -	**yfx**	**500**
:	**xfy**	**500**
+ -	**fx**	**500**
*** / mod //**	**yfx**	**400**
^	**xfy**	**300**
..	**yfx**	**200**

5.1.2 Entrada e saída de caracteres

Os predicados para entrada e saída de termos exige que cada linha seja um termo (terminando com ponto seguido de "enter"), o que não é muito natural para entrada de dados. Os predicados de entrada e saida de caracteres fornecem meios mais naturais, lendo e/ou escrevendo um caracter por vez.

get0(Caracter)	Lê um caracter a partir do dispositivo de entrada padrão. Se Caracter estiver instanciado, o predicado get0 verifica se o próximo caracter lido é igual ao caracter especificado, retornando sucesso ou falha.
get(Caracter)	Semelhante ao predicado get0, exceto que não lê caracteres com código ASCII inferior a 32 (pula caracteres não visíveis).
put(Caracter)	Escreve no dispositivo padrão de saida o caracter especificado.
nl	Faz com que a próxima saida seja impressa na próxima linha.
tab(Número)	Escreve o número especificado de espaços no dispositivo padrão de saida.

5.1.3 Entrada e saída em arquivos

A entrada e saida em arquivos difere da entrada e saída padrão devido ao fato de ser necessário abrir ou criar o arquivo antes de lê-lo ou gravá-lo e fechar o arquivo quando todas as operações de entrada e saída forem completadas.

Sempre que um programa abre um arquivo, o Arity Prolog associa a ele um número, que é chamado de apontador de arquivo (AA). Esse número deve ser usado em todas as operações subseqüentes sobre o arquivo.

read(AA,Termo)	Lê termos do arquivo.
get0(AA,Termo)	Lê caracteres do arquivo.
get(AA,Caracter)	Lê somente caracteres com código ASCII maior que 32 do arquivo.
display(AA,Caracter)	Escreve termos no arquivo na notação pré-fixada.
write(AA,Termo)	Grava termos no arquivo.
put(AA,Termo)	Grava caracteres no arquivo.
nl(AA)	Grava um caracter "new-line" no arquivo.
tab(AA,Número)	Grava o número especificado de espaços no arquivo.
create(AA,Arquivo)	Cria um arquivo, abrindo-o para gravação; se o arquivo já existe, ele é esvaziado e regravado.
open(AA,Arquivo,Modo)	Abre o arquivo com o modo de acesso especificado, que pode ser: **r** (read), **w** (write), a (append), **rw** (read/write) e **ra** (read/append).
close(AA)	Fecha o arquivo.

5.2. Manipulação da base de dados

Os predicados a seguir possibilitam adicionar, revisar e remover cláusulas de uma base de dados Prolog.

consult(Arquivo)	Lê uma base de dados do disco, na forma fonte, acrescentando-a à base corrente.
reconsult(Arquivo)	Lê uma base de dados do disco, na forma fonte, substituindo as cláusulas duplicadas na base de dados corrente pelas cláusulas lidas.
save(Arquivo)	Grava a base de dados corrente em disco, na forma binária.
restore(Arquivo).	Lê uma base de dados do disco na forma binária, destruindo a base de dados corrente.
listing	Lista no dispositivo padrão de saida todos os predicados da base de dados.
listing(Nome/Grau,...)	Lista no dispositivo padrão de saida os predicados especificados, com símbolo funcional dado por **Nome** e número de argumentos dado por **Grau**.
clause(Cabeça,Corpo)	Retorna os objetivos associados a uma dada **Cabeça**. Se houver mais de uma cláusula associada, o Prolog escolherá a primeira. Se uma cláusula não possuir corpo, considera-se que tem **Corpo** com valor “true”.
asserta(Cláusula)	Acrescenta uma cláusula no incio da lista de predicados.
assertz(Cláusula)	Acrescenta uma cláusula no fim da lista de predicados.
retract(Cláusula)	remove uma cláusula da base de dados.

5.3 Manipulação de termos

var(X)	Sucede se X é uma variável não instanciada.
nonvar(X)	Determina se X é uma variável anônima.
atom(X)	Sucede quando X está representando um átomo Prolog.
integer(X)	Sucede quando X está representando um número inteiro.
atomic(X)	Sucede quando X está representando um átomo ou um inteiro.
X = Y	Sucede se X é unificável com Y.
X \= Y	Sucede se X não é unificável com Y.
X == Y	Sucede se X for equivalente a Y.
X \== Y	Sucede se X não for equivalente a Y.
Estrutura =.. Lista	Converte uma estrutura em uma lista e vice-versa. A lista é gerada a partir da estrutura usando o símbolo funcional como cabeça e os argumentos como cauda. A estrutura é montada a partir da lista usando a cabeça como símbolo funcional e a cauda como argumentos.
arg(Argumento,Termo,Valor)	Retorna o valor de um dado argumento de um termo.
functor(Estrutura,Nome,Grau)	Desmonta uma estrutura retornando seu nome e grau.
name(Átmo,Lista)	Converte uma lista em um átomo e vice-versa. Se Átomo é um átomo, a lista é gerada como uma seqüência com os códigos ASCII que compõem o nome do átomo. Se Átomo é um inteiro, a lista é gerada como uma seqüência de códigos ASCII para a impressão de valores inteiros.

5.4 Expressões aritméticas

Os seguintes predicados atuam sobre operadores aritméticos.

X > Y	Sucede se X for maior que Y.
X < Y	Sucede se X for menor que Y.
X >= Y	Sucede se X for maior ou igual a Y.
X =< Y	Sucede se X for menor ou igual a Y.
X =:= Y	Sucede se X for igual a Y.
X =\= Y	Sucede se X for diferente de Y.
X is Y	Avalia a expressão aritmética Y e unifica seu valor com X. Y deve estar instanciada para uma estrutura que possa ser interpretada como uma expressão aritmética. Primeiro, a estrutura instanciada por Y é avaliada para fornecer um inteiro, o resultado. Se X não estiver instanciada, então X torna-se instanciada para o resultado e o **is** sucede. Se X já estiver instanciada, então X e o resultado são comparados para igualdade, sendo que o is sucede ou falha dependendo do resultado do teste.

5.5 Depuração

trace	Ativa uma depuração exaustiva.
notrace	Desatva o modo de depuração.
spy(Nome/Grau)	Especifica um ponto de parada em um predicado.
nospy(Nome/Grau)	Remove um ponto de parada de um predicado.