UNIVERSIDADE FEDERAL DO PARÁ
INSTITUTO DE CIÊNCIAS EXATAS E NATURAIS
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIAS DA COMPUTAÇÃO

Prof. Eloi Luiz Favero
Versão 2012

SUMÁRIO

1. Programação com Listas 3
1.1. Operações sobre o tipo lista 3
1.2. O tipo de dados lista 4
1.3. Programação recursiva 5
1.3.1. Refatoração de código de cláusulas 5
1.4. Usando um acumulador 7
1.5. Seleção em Listas 9
1.6. Explorando o significado de um predicado 10
1.7. Construção de listas 10
1.8. Determinismo: fatoração com Corte 12
1.9. Trabalhando com ordem 14
1.10. Ordem alfanumérica 16
1.11. Predicados com computação aritmética 17
2. Refatoração: Otimização de predicados 18
2.1.1. Recursividade na cauda 19
2.1.2. Mais sobre inferências 20
2.2. Predicados para listas do SWI-Prolog 21
3. Programação procedural vs lógica (Opcional) 23

1. Programação com Listas

Este capítulo é básico e essencial para qualquer curso de Prolog. Ele apresenta um estudo detalhado sobre programação em lógica pura, com o tipo de dados lista, que é fundamental para o Prolog. Algumas técnicas básicas de programação são introduzidas no decorrer do capítulo, entre elas, programação recursiva, predicados reversíveis, fatoração de cláusulas e predicados, corte e técnica de acumuladores.

Neste capítulo, examinamos a definição de diversos predicados que são predefinidos nos sistemas Prolog. Entre outros, temos os predicados append, member, select. No momento de programá-los usamos um sufixo numérico ou um nome em português, por exemplo, append1, membro, select1, pois a tentativa de redefinir um predicado causa um erro do tipo "No permission to modify static procedure select/3". No final do capítulo apresentamos a relação de predicados para manipulação de listas que estão predefinidos no SWI-Prolog. Estes são utilizados nos programas apresentados nos capítulos seguintes.

Este capítulo está estruturado em torno de uma lista com aproximadamente 40 exercícios Exerc que foram criados para se treinar a **prática da Programação em Lógica** com o Prolog. Os Exerc podem ser utilizados para compor listas de exercícios, pequenos testes e/ou provas. Na apresentação do material é importante ter aulas teóricas onde os alunos vão tentar responder os exercícios no papel e depois num segundo momento as aulas práticas para o aluno verificar se os algoritmos executam, afinando as soluções. Remendamos 50% de aulas teóricas e 50% de aulas práticas; porém, sempre aulas intercaladas, por exemplo, 2 horas teóricas depois duas práticas.

1.1. Operações sobre o tipo lista

O tipo de dados lista é uma das estruturas fundamentais para a PL (e também em linguagens funcionais). Parte das operações sobre listas são apresentadas em quatro principais classes:

- construção: [], [|], append, zipper;
- seleção: select, member, sublist, last;
- ordem: sort, permutation;
- diversas: contagem, somatório, etc.

Na classe construção temos os construtores primitivos [], [|] e outras operações que têm listas como saída, entre elas o famoso append, que concatena duas listas. Na classe de seleção temos os predicados que selecionam elementos ou sublistas de listas.Uma das operações de seleção mais usadas é o member. A classe de ordem é subdividida em duas. A subclasse estrutural compreende operações como o reverse, que inverte uma lista. Já na subclasse linear, temos as operações que ordenam os elementos de uma lista seguindo critérios como a ordem aritmética ou lexicográfica. Na classe diversas coletamos todas as outras operações com listas, tais como operações aritméticas com listas que compreendem algum tipo de

cálculo sobre os valores das listas (ex. contagem, somatório, produto) e operações que envolvem mais de uma classe.

1.2. O tipo de dados lista

Em Prolog, listas são estruturas com representação interna recursiva. Uma lista é definida por dois construtores:

- [], que representa a lista vazia;
- [X|Xs], uma lista com o elemento X na cabeça e com a lista Xs como cauda.

Estes são os construtores básicos para recursivamente se construir listas de qualquer comprimento. Por exemplo, [a|[b|[c|[]]]] é uma lista com os elementos a,b,c. Para evitar esse excesso de colchetes aninhados, melhorando a leitura e escrita, o Prolog admite algumas simplificações, como segue:

Sintaxe básica:	Sintaxe simplificada:
[]	[]
[a| []]	[a]
[a|[b| []]]	[a,b]
[a|[b|[c| []]]]	[a,b,c]
[a|Xs]	[a|Xs]
[a|[b|Xs]]	[a,b|Xs]

Listas são manipuladas pela unificação. Por exemplo, para remover a cabeça da lista [a,b,c] basta unificá-la com uma lista que tenha uma variável na cabeça:

```
?- [X|Xs]=[a,b,c].  X=a, Xs=[b,c] Yes
```

Vale notar que a cabeça de uma lista é sempre um elemento, já a cauda é uma estrutura de lista. Assim, para manipular os elementos de uma lista devemos sempre trabalhar começando na cabeça da lista. Podemos pegar vários elementos da cabeça. No exemplo abaixo pegamos os três primeiros elementos de uma lista.

```
?- [X,Y,Z|_] = [1,3,5,7,9,11,13,15].
    X=1, Y=3, Z=5 Yes
?- [A,B,C|_]=[1,[a,b],f(5),4,f(bb),[6,t],7].
   A=1, B=[a,b], C=f(5)  Yes
```

Aqui, também vemos que uma lista pode conter elementos dos mais variados tipos, como inteiros, letras, átomos, outras listas e functores. Estas operações de pegar vários elementos de uma lista são baseadas na operação primitiva, que pega apenas um, por exemplo, abaixo pegamos os dois primeiros elementos em X e Y, usando apenas a construção primitiva de lista [|]:

```
?- L=[1,3,5,7],L=[X|Xs],Xs=[Y|_].
```

X=1, Y=3, L=[1,3,5,7], Xs = [3,5,7] Yes
?- L=[1,3,5,7],L=[X|[Y|_]].
X=1, Y=3, L=[1,3,5,7] Yes

Nestes exemplos usa-se a variável curinga anônima, underscore (_). Ela é uma variável sem nome. Assim quando o conteúdo de uma variável não é usado para nada, é melhor que a variável seja anônima.

1.3. Programação recursiva

A seguir, veremos uma versão lógica para o predicado compr, que calcula o comprimento de uma lista. No Prolog, este predicado é predefinido como lenght/2, como no teste que segue.

?- length([a,b],L).
L = 2

Para estudo de algoritmos e estruturas de dados devemos inicialmente desenvolver versões lógicas dos programas em Programação em Lógica (PL) pura. Estas versões posteriormente podem ser relaxadas, buscando um efeito mais procedural (ou funcional), por exemplo, substituindo-se expressões aritméticas por valores calculados. A definição de operações sobre listas faz referência à recursividade da estrutura de dados lista. Assim, o predicado comprimento compr/2 é definido por duas regras, uma para a lista com cabeça e cauda e a outra para a lista vazia. A primeira é chamada regra recursiva e a segunda regra base, que define quando termina a recursividade:

- regra recursiva: o comprimento da lista [X|Xs] é 1 mais o comprimento da cauda;
- regra base: o comprimento da lista vazia [] é 0.

Esta especificação traduzida como um predicado Prolog definido por duas regras:

```
compr(L,T):-L=[X|Xs], compr(Xs,T1),T=1+T1.
compr(L,T):-L=[], T=0.
```

1.3.1. Refatoração de código de cláusulas

Uma lição básica de programação é saber refatorar. Refatorar significa reescrer uma cláusula de um predicado com o objetivo de torna-la mais econômica, mais simples de leitura. Neste sentido, a PL é uma boa linguagem, pois ela define formalmente o conceito de refatoração de cláusulas, com as regras de reescrita. Dada uma cláusula, podemos encontrar uma instância mais simples com a mesma a semântica.

Vamos examinar o exemplo do predicado compr/2. A primeira dificuldade na programação de um predicado é encontrar uma versão inicial, que funcione como esperado, isto é, que seja uma especificação correta para o problema. Assumindo que a versão inicial do compr/2 é correta, podemos perguntar: existe um predicado equivalente e de leitura mais simples? Examinemos como simplificar a primeira cláusula em alguns passos:

Passo	**Descrição**	**Regra de reescrita /nro palavras**
1	compr(L,T):-L=[X|Xs], compr(Xs,T1), T=1+T1. "o compr de uma lista L é T se L é igual a uma lista com cabeça X e cauda Xs, e o compr da cauda Xs é T1, e T é igual a 1 mais T1"	= 37 palavras
2	compr([X|Xs],T):-[X|Xs]=[X|Xs], compr(Xs,T1), T=1+T1. compr([X|Xs],T):-true, compr(Xs,T1), T=1+T1. compr([X|Xs],T):-compr(Xs,T1), T=1+T1. "o compr de uma lista com cabeça X e cauda Xs é T se o compr da cauda Xs é T1, e T é igual T1 mais 1"	Regra a): Eliminação da variável L: substituição {L/[X|Xs]}; true no corpo da cláusula pode ser omitido; 28 palavras
3	compr([X|Xs],T):-compr(Xs,T1), T=1+T1. compr([X|Xs],1+T1):- compr(Xs,T1), 1+T1=1+T1. compr([X|Xs],1+T1):- compr(Xs,T1), true. compr([X|Xs],1+T1):- compr(Xs,T1). "o compr de uma lista com cabeça X e cauda Xs é 1 mais T1 se o compr da cauda Xs é T1"	Regra a) Eliminação da variável T: idem 23 palavras
4	compr([X|Xs],1+T1):- compr(Xs,T1). compr([X|Xs],1+C):- compr(Xs,C). "o compr de uma lista com cabeça X e cauda Xs é 1 mais C se o compr da cauda Xs é C"	Regra b) Substituição de variável, com nome mais simples "c" vs. "t um" 23 palavras
5	compr([_|Xs],1+C):- compr(Xs,C). "o comprimento de uma lista com cauda Xs é um mais C se o comprimento da cauda Xs é C"	Regra c) Substituição de variável por variável anônima: {_/X} 20 palavras

Para entendermos o efeito da refatoração para cada um dos passos fizemos uma leitura: iniciou com 37 palavras e terminou com 20 palavras. Em cada um destes passos obtivemos uma leitura mais simples, com menos palavras ou com palavras mais simples. Refatorar um predicado é sinônimo de torná-lo mais simples, porém mantendo o seu significado.

Segue a versão final do predicado incluindo as duas cláusulas; a segunda cláusula do predicado é refatorada pela eliminação das duas variáveis: compr([],0).

```
compr([_|Xs],1+T):-compr(Xs,T).
compr([],0).

?- compr([a,b,c],T). T=1+1+1+0 Yes
?- compr([],X). X=0 yes
```

Para simplificar cláusulas, evitando-se dar nomes a variáveis (indivíduos) que não se relacionam com ninguém, são usadas variáveis curingas anônimas, que têm uma semântica própria, por exemplo,

```
?- [A,A]=[ 3,X].        A=X=3      porém em
?- [_,_]=[3,X],X=4. X=4 Yes.
```

No primeiro caso o A é usado para passar o valor 3 para X, porque, se temos dois As, ambos assumem o mesmo valor; com variáveis anônimas isto não acontece, cada ocorrência é uma instância distinta. **Assim sempre que temos uma variável única (single) numa cláusula ela pode ser reescrita como anônima.**

Temos outros casos de possíveis regras de reescritas para fatoração. Segue abaixo a lista de regras exemplificadas:

Regra :	**Exemplo:**
Regra a): Eliminação da variável: eliminando L, substituindo {[X|Xs]/L}	cabeça(E1,L):-L=[X|Xs], X=E1, Xs=[A|B]. cabeça(E1,[X|Xs]):- X=E1, Xs=[A|B].
Regra b) Substituição de variável, com nome mais simples: substituindo E1 por E	cabeça(E,[X|Xs]):- X=E, Xs=[A|B].
Regra c) Substituição por variável anônima: quando que ocorre uma única vez na cláusula, B por _	cabeça(E,[X|Xs]):- X=E, Xs=[A|_].
Regra e) Transformação de regra em fato: quando tem ":-true." após a eliminação de X e Xs	cabeça(E,[E|[A|_]]):- true. cabeça(E,[E|[A|_]]).
Regra d) Eliminação de construtor de lista: [a|[b|Xs]]=[a,b|Xs] e [a|[]] = [a]	cabeça(E,[E,A|_]).

1. Exerc: Utilizando as regras de fatoração, simplifique a cláusula abaixo, indicando qual regra é utilizada em cada passo:
   pega3(E,L):-L=[B|A],A=[Y|Ys],Ys=[Z|D],Z=E.

1.4. Usando um acumulador

Muitos problemas de programação recursiva podem ser resolvidos por diferentes estratégias. Quando o resultado de uma computação é calculado passo a passo (por exemplo, somar ou contar uma unidade), podemos usar a técnica do acumulador. Esta técnica simula uma computação numa linguagem procedural. Por exemplo, um programa para calcular o comprimento de uma lista L em Pascal pode ser:

```
acc:=0;
while not eh_vazia(L)
   begin     L := cauda(L);
             acc := acc+1;   end;
resultado := acc;
```

O papel do acumulador acc é armazenar o resultado intermediário durante o cálculo. Quando termina o laço de processamento, a variável resultado recebe o valor do acumulador. Abaixo, temos uma versão Prolog do predicado comprimento usando um acumulador. Definimos um predicado auxiliar compr3, com um parâmetro a mais, para o acumulador. O predicado principal chamado auxiliar, inicializando o acumulador= 0 (equivale à linha antes do

while). Na regra base do predicado auxiliar compr3, o valor do acumulador é passado para predicado principal (equivale a regra após while).

```
comprAcc(L,C):-compr3(L,0,C).
compr3([X|Xs],Acc,C):-compr3(Xs,Acc+1,C).
compr3([],Acc,Result):-Result=Acc.
```

Para estudo, comparamos os dois predicados comprimento em Prolog. A principal diferença está na estratégia de como é calculado o valor - sem acumulador vs. com acumulador. No *trace* para compr sem acumulador, o valor é calculado de trás para frente – da chamada mais interna para a mais externa:

```
?-trace(compr,[call,exit]).
%       compr/2: [call, exit] Yes
[debug]  ?- compr([a,b,c],X).
 T Call: (6) compr([a, b, c], _G367)
  T Call: (7) compr([b, c], _G422)
   T Call: (8) compr([c], _G425)
    T Call: (9) compr([], _G428)
    T Exit: (9) compr([], 0)
   T Exit: (8) compr([c], 0+1)
  T Exit: (7) compr([b, c], 0+1+1)
 T Exit: (6) compr([a, b, c], 0+1+1+1)
X = 0+1+1+1
%
```

No *trace* para comprAcc com acumulador, ao contrário – o valor é calculado da chamada mais externo para a interna. O trace mostra em cada um dos passos da execução o que entra call e o que sai exit:

```
?-trace([compr3,comprAcc],[call,exit]).
%       compr3/3: [call, exit]
%       comprAcc/2: [call, exit]  Yes
 [debug]  ?- comprAcc([a,b,c],X).
 T Call: (7) comprAcc([a, b, c], _G391)
 T Call: (8) compr3([a, b, c], 0, _G391)
 T Call: (9) compr3([b, c], 0+1, _G391)
 T Call: (10) compr3([c], 0+1+1, _G391)
 T Call: (11) compr3([], 0+1+1+1, _G391)
 T Exit: (11) compr3([], 0+1+1+1, 0+1+1+1)
 T Exit: (10) compr3([c], 0+1+1, 0+1+1+1)
 T Exit: (9) compr3([b, c], 0+1, 0+1+1+1)
 T Exit: (8) compr3([a, b, c], 0, 0+1+1+1)
 T Exit: (7) comprAcc([a, b, c], 0+1+1+1)
X = 0+1+1+1
```

No cenário acima usamos o comando trace/2 para entrar no modo debug, controlando a exibição só para as portas call e exit.

2. Exerc: Com base no programa compr/2 faça um predicado sum/2 que soma uma lista de valores? Solução:

```
sum([],0).
sum([X|Xs],X+S) :- sum(Xs,S).
```

1.5. Seleção em Listas

Nesta seção, apresentaremos o predicado member/2, que é predefinido nos sistemas Prolog. Normalmente, não podemos redefinir um predicado predefinido no sistema; podemos criar o predicado com um novo nome, por exemplo, member1 ou membro.

O predicado membro/2, com duas regras, verifica se um elemento é membro de uma lista:

- (base) um elemento é membro de uma lista, se ele é a cabeça da lista;
- (recursiva) um elemento é membro de uma lista, se ele for membro da cauda da lista.

3. Exerc: Faça o predicado membro/2 baseado nestas duas regras. Solução:

```
membro(X,[X|_]).
membro(X,[_|Xs]):-membro(X,Xs).
```

Para executar roda-se

```
?- membro(a,[b,c]).  No.
?- membro(b,[a,b,c]).  Yes
?-membro(X,[a,c,c]). X=a Yes
?- membro(a,X).  X=[a] Yes
?- L=[a,B,c],membro(e,L).  L=[a,e,c], B=e   Yes
?- membro(X,Y).  X=_, Y=[X|_]; Yes
```

Como vemos, membro/2 é reversível. Se os dois parâmetros são termos fechados (preenchidos com constantes), o predicado funciona como um teste. Se nos parâmetros existem variáveis, o Prolog tentará inferir o valor delas. Este predicado tem vários usos:

- testar se um elemento está na lista;
- incluir um elemento na lista, quando a lista é uma variável ou quando a lista contém uma variável;
- retornar um elemento da lista (na pergunta, o elemento é uma variável).

Relacionado com este último uso, vale reforçar que ele retorna todos os elementos de um conjunto, um de cada vez, por retrocesso:

```
?- membro(X,[a,b,c]).
X = a ; X = b ; X = c ; No
```

Segue uma versão mais simples, chamada membro1/2, que somente testa se um elemento é membro de uma lista. Este predicado é definido por duas regras mutuamente exclusivas. A primeira para X==Y e a segunda negando a condição da primeira.

```
membro1(X,[Y| _]):- X ==Y.
membro1(X,[Y|Xs]):-X\==Y,membro1(X,Xs).
?- membro1(b,[a,b,c]).  Yes
?- membro1(a,X).  No
?- L=[a,B,c],membro1(x,L).  No.
```

1.6. Explorando o significado de um predicado

A diferença entre membro e membro1 está no significado dos operadores de igualdade == e unificação =. Apresentamos uma técnica para se conhecer mais sobre operadores e predicados. Um sistema Prolog, por ser um interpretador de comandos, nos oferece a facilidade de explorar, de imediato e interativamente, uma operação ou estrutura de dados. Submetendo-se pequenas perguntas e analisando-se as respostas,... reiterando com pequenas variações, até descobrir o uso de uma construção. Este processo pode e deve ser usado com os predicados que são novos para o programador - ou, às vezes, também para relembrar um significado de um predicado já conhecido. Veja alguns exemplos:

```
?- 1==1.  Yes
?- 'a'==a.  yes
?- a==B.  No
?- a\==B.  Yes
?- p(a,b)=p(a,X).  X = b  Yes
?- p(b)=p(c,D).  No
?- p(b)\=p(c,D).  D=_  Yes
?-p(X)=p(X+1). No
?- p(X)==p(X+1).
```

É importante diferenciar a unificação = da igualdade ==. A unificação de dois termos os torna iguais, se casam. Uma vez casados é como um casamento sem separação pelo resto da vida útil das variáveis, na instância da cláusula. Uma mesma variável em diferentes instâncias de cláusulas pode assumir diferentes valores, ver capítulo 2. Na verdade, diversas variáveis podem casar com um único valor, vejamos:

```
?- A=B, B=Z, C=A, C=5.   A=B=C=Z=5   Yes
```

O teste de igualdade simplesmente verifica se dois termos são iguais, e cada um dos termos mantém seus valores.

1.7. Construção de listas

Uma das operações sobre listas mais usadas é a concatenação, também conhecida como append/3, no Prolog. Esta operação concatena duas listas dadas em uma terceira:

- (base) concatenar uma lista vazia [] com uma lista Bs, o resultado é a lista Bs;
- (recursiva) concatenar a lista [A|As] com a lista Bs resulta na lista [A|AsBs], sendo AsBs uma lista formada pela concatenação de As com Bs.

4. Exerc: faça o predicado append1/3 baseado nestas duas regras.

```
append1([],Bs,Bs).
append1([A|As],Bs,[A|AsBs]):- append1(As,Bs,AsBs).
```

Em Prolog, esta especificação resulta num flexível predicado, como mostra a execução abaixo. Dados dois parâmetros é devolvido o terceiro. Chamamos de append1, pois o append já é predefinido no Prolog.

```
?- append1([a,b],[c,d],X).
  X=[a,b,c,d].
?- append1(X,[c,d],[a,b,c,d]).
  X=[a,b]
?-append1([a,b],X,[a,b,c,d]).
  X=[c,d].
```

O append é um dos predicados de manipulação de listas mais útil e reusado para definir novos predicados.

5. Exerc: Faça o predicado membro/2 usando append. X é membro de uma lista L, se está no meio da lista? Solução:

```
membro(X,L):-append(_,[X|_],L).
```

Outros dois predicados similares ao member são prefix e sufix que devolvem todos os prefixos ou sufixos de uma lista. Com eles é fácil fazer o predicado sublist/2 que testa se uma lista é sublista de outra e/ou, também, gera todas as sublistas.

6. Exerc: Faca o predicado sublist/2. Solução:

```
prefix(P,L):-append(P,_,L).
suffix(S,L):-append(_,S,L).
sublist(S,L):-prefix(P,L),suffix(S,P).

?- prefix(P,[a,b,c]).
P = [] ;
P = [a] ;
P = [a, b] ;
P = [a, b, c]
?- suffix(S,[a,b,c]).
S = [a, b, c] ;
```

```
S = [b, c] ;
S = [c] ;
S = []
?- sublist(L,[a,b]),L\=[].
L = [a] ;
L = [a, b] ;
L = [b]
```

Outra operação parente do append é o select/3. Ela tem a mesma estrutura recursiva do member/2, com a diferença que, dada uma lista, ela remove um elemento desta, devolvendo-a sem o elemento. Definimos um select1, pois o select é predefinido no Prolog.

```
select1(X,[X|Xs],Xs).
select1(X,[Y|Xs],[Y|Zs]):-select1(X,Xs,Zs).

?- L=[a,b,c,d], select1(b,L,Lo).
L = [a,b,c,d] ,  Lo = [a,c,d] Yes
?- L=[a,b,c,d], select1(X,L,Lo).
L = [a,b,c,d] , X = a ,Lo = [b,c,d] ;
L = [a,b,c,d] , X = b ,Lo = [a,c,d] ;
L = [a,b,c,d] , X = c ,Lo = [a,b,d] ;
L = [a,b,c,d] , X = d ,Lo = [a,b,c] ; no
```

Este predicado, por retrocesso, seleciona sistematicamente cada um dos elementos de uma lista, partindo do primeiro ao último.

1.8. Determinismo: fatoração com Corte

Um predicado Prolog pode retornar várias soluções, então um desafio é tornar um predicado determinístico. Seja o predicado max abaixo.

```
max(X,Y,X):- X>Y.
max(X,Y,Y).

?- max(3,2,M).
M=3 ;  M=2
```

Este programa retorna dois resultados, o segundo resultado é errado. Com o operador de corte podemos consertar a solução; este tipo de corte é chamado de **corte vermelho** pois concerta o predicado.

```
max1(X,Y,X):- X>Y,!.
max1(X,Y,Y).
?- max1(3,2,M).
M=3
```

Podemos ter outro uso do **corte verde**, pois não muda o significado do predicado. O corte verde deve vir sempre depois dos testes de cada cláusula.

```
max0(X,Y,M):- X>Y,!, M=X.
max0(X,Y,M):- X<Y,!, M=Y.
max0(X,Y,M):- X=Y,!, M=X.
```

```
?- max0(3,4,X).
X=4.
```

O corte pode ser útil também na programação com listas como veremos a seguir.

7. Exerc: Defina o predicado ultimo/2, que devolve o último elemento de uma lista?
   Solução:

```
ultimo(L,U):-L=[X|Xs],Xs=[],U=X.
ultimo(L,U):-L=[X|Xs],Xs\=[],ultimo(Xs,U).
```

Fatorando o predicado acima podemos remover a variável L e também podemos reescrever [X|Xs],Xs=[],U=X como [U]. Assim podemos criar a nova versão simplificada do predicado, como abaixo:

```
ultimo1([U],U).
ultimo1([_|Xs],U):-ultimo1(Xs,U).
?- ultimo1([a,b,c],U).
U=c ; false
```

Neste predicado o sistema não pode decidir por uma cláusula observando apenas os parâmetros, pois [U]=[U|[]]=[_|Xs] sempre que Xs=[]. Por isso após responder ele pensa que ainda tem outra resposta **(; false).** Mas podemos reescrever uma versão melhorada, diferenciando as duas cláusulas no primeiro parâmetro, considerando que [X,Y|Xs]\=[U].

```
ultimo2([U],U).
ultimo2([X,Y|Xs],U):-ultimo2([Y|Xs],U).
```

```
?- ultimo3([a,b,c],U).  U = c.
```

Nesta versão o sistema não permite tentar a outra alternativa com **(;).** Este efeito pode ser programado com o operador de corte. Colocando-se corte no final da primeira cláusula da versão ultimo1 teremos a versão ultimo3, abaixo.

```
ultimo3([U],U):-!.
Ultimo3([_|Xs],U):-ultimo3(Xs,U).
?- ultimo3([a,b,c],U).
U=c
```

Aqui o corte (!), corta o uso da cláusula que vem na linha seguinte. Não modifica o resultado do predicado mas corta um processamento inútil. Esta otimização deve ser feita sempre que um predicado é determinístico: retorna uma única resposta. Neste caso no sistema não deve-se responder **(; false).**

Assim o corte é usado para **refatoração de predicados** com mais de uma cláusula, cortando a execução das cláusulas seguintes (com o mesmo nome) dentro do mesmo predicado.

8. Exerc: Faça uma versão do predicado ultimo/2, usando append. Solução:
   ultimo(L,U):-append(_,[U],L).
9. Exerc: Defina o predicado remU/2 que devolve uma lista sem o último elemento, fazendo duas regras uma recursiva. Solução.
   remU(L,Lu):-L=[X|Xs],Xs=[],Lu=[].
   remU(L,Lu):-L=[X|Xs],Xs\=[],Lu=[X|Xu],remU(Xs,Xu).
10. Exerc: Escreva em linguagem natural as regras que definem o predicado remove último remU/2 com recursividade.
11. Exerc: Simplifique esta versão do predicado remU/2, para que não retorne **(; false),** sem usar corte.
12. Exerc: Simplifique esta versão do predicado remU/2, para não retornar **(; false)** tirando os testes e usando corte.
13. Exerc: Defina o predicado remU/2 usando append sem recursividade.
    remUa(L,Lu):-append(Lu,[U],L).
14. Exerc: Defina um predicado contiguos/1 que testa se uma lista tem dois elementos contíguos iguais, faça uma versão recursiva. Por exemplo ?-contiguos ([a,b,c,c,e]). Yes
15. Exerc: Defina um predicado contiguos/1, que testa se uma lista tem dois elementos contíguos iguais, usando append.
16. Exerc: Defina um predicado dupl/1, que é verdadeiro, se a lista tem elementos duplicados. Use o member e recursividade.
17. Exerc: Defina um predicado trocaPU/2 que devolve uma lista em que o primeiro e último elementos são trocados de posição. Use append.
18. Exerc: Defina um predicado remDupl/2 que remove os elementos duplicados de uma lista, use apenas o member e recursividade. Remova as cópias iniciais dos elementos da lista. Assim, ?-remDupl([a,b,a],X). X=[b,a]. Yes
19. Exerc: Defina um predicado remDupl1/2 que remove os elementos duplicados de uma lista, use apenas o member e o select/3. Remova as cópias finais dos elementos da lista. Assim, ?-remDupl1([a,b,a],X). X=[a,b]. Yes
20. Exerc: Faça um predicado remove/3 tal que: ?- remove(a,[a,b,a,d,x,a], X). X=[b,d,x] Yes
21. Exerc: Defina o predicado flatten/2 que devolve uma lista simples a partir de uma lista formada por listas aninhadas, mantendo a ordem de ocorrência dos elementos, como em
    ?- flatten1([1,[2],[],[2,[3,4],5],6],X). X = [1, 2, 2, 3, 4, 5, 6] Yes
    Solução:
    ```
    flatten1([],   []).
    flatten1(X,   [X]):-X\=[], X\=[_|_].
    flatten1([X|Xs],F):-flatten1(X,F1),flatten1(Xs,F2),append(F1,F2,F).
    ```

22. Exerc: Faça um predicado zipper/3" tal que: ?- zipper([a,b,c,d], [1,2,3,4], X). X=[a-1, b-2, c-3, d-4], yes Solução:
    ```
    zipper([X|Ys],[Y|Ys],[X-Y|XYs]):-zipper(Xs,Ys,XYs).
    zipper([],[],[]).
    ```

1.9. Trabalhando com ordem

Entre os elementos de uma lista, existem dois tipos de ordem. A ordem aritmética ou alfanumérica, usada para ordenar nomes ou valores numéricos. Este tipo de ordem leva em conta o valor ou o conteúdo de cada elemento. Por outro lado, a ordem estrutural dependente da posição física dos elementos na lista e não do valor de cada elemento. Este tipo de ordem é usado em problemas de permutação, por exemplo, permutar os elementos de uma lista. Abaixo, mostra-se a execução do predicado permutation/2, gerando as permutações da lista [a,b,c].

```
?- permutation([a,b,c],P).
P = [a,b,c] ;
P = [a,c,b] ;
P = [b,a,c] ;
P = [b,c,a] ;
P = [c,a,b] ;
P = [c,b,a] ; No
```

Este predicado, permutation, usa o predicado select/3 para extrair cada elemento de uma lista. Assim sendo, permutar uma lista consiste em:

- (base) se a lista é [], então retorna-se [];
- (recursiva) senão, dada uma lista, extrai-se um de seus elementos usando select/3, coloca-se como primeiro elemento da solução e chama-se recursivamente o predicado permutação para a lista sem o elemento.

23. Exerc: Defina este predicado permutation. Solução:

```
permutation(Xs,[Z|Zs]):-select(Z,Xs,Ys),permutation(Ys,Zs).
permutation([],[]).
```

Um predicado parente do permutation/2 é o reverse/2, que inverte a ordem dos elementos de uma lista. Podemos programá-lo em duas versões: um com acumulador e a outra sem acumulador.

24. Exerc: Defina o predicado reverse com acumulador. Solução:

```
reverse(L,R) :- reverse(L,[],R).
reverse([], R,R).
reverse([X|Xs],ACC,R) :- reverse(Xs,[X|ACC],R).
?- reverse([a,b,c],R).      R=[c,b,a]
```

25. Exerc: Defina o predicado reverse sem acumulador, usando append/3. Codifique as regras:
    (base) o reverso de uma lista vazia é a lista vazia;
    (recursiva) o reverso de uma lista é o reverso da cauda concatenado com a cabeça.
26. Exerc: Defina o predicado palindrome/1, que é verdadeiro se a lista é um palíndromo, por exemplo, [a,b,c,d,c,b,a]. Faça uma versão recursiva sem usar reverse.
27. Exerc: Defina o predicado palindrome/1, sem recursividade usando o reverse.
28. Exerc: Defina um predicado metIguais/1, que é verdadeiro se uma lista é formada por duas metades iguais. Use o append. Seguem dois exemplos de uso.

```
?-metIguais([a,b,c, a,b,c]).  Yes
?-metIguais([a,b,c, a,b,d]).  No
```

Assumindo que uma lista é um conjunto de elementos, assim [a,b] ou [b,a] são duas listas que representam o conjunto {a,b}.

29. Exerc: Defina o predicado subConjunto/2 que retorna todos os subcojuntos de um conjunto.

```
subConjunto([X|Xs],Y):- select(X,Y,Ys), subConjunto(Xs,Ys).
subConjunto([],Y).
```

30. Exerc: Defina o predicado interseção de dois conjuntos? Solução:

```
intersecao([],X,[]).
intersecao([X|Xs],Y,[X|Is]):-  member(X,Y),  intersecao(Xs,Y,Is).
intersecao([X|Xs],Y,   Is):- \+ member(X,Y),  intersecao(Xs,Y,Is).
```

31. Exerc: Defina o predicado união de dois conjuntos, onde na solução não pode ter elementos repetidos; faça uma versão recursiva?

1.10. Ordem alfanumérica

A ordem alfanumérica é usada para ordenar números e cadeias de caracteres, por exemplo, ordenar nomes na ordem alfabética. O predicado isOrdered/1 é verdadeiro se uma lista de inteiros está em ordem crescente.

32. Exerc: Defina o predicado isOrdered/1? Solução:

```
isOrdered([]).
isOrdered([_]).
isOrdered([X,Y|XYs]):-X=<Y, isOrdered([Y|XYs]).
?- isOrdered([1,5,5,11]). Yes
?- isOrdered([2,1]). No
```

33. Exerc: Sem usar acumulador, defina um predicado maxL(M,L) que retorna o valor máximo de uma lista de valores numéricos? (seleção e ordem) Solução:

```
maxL([X],X):-!.
maxL([X|Xs],X):-maxL(Xs,M),X >M.
maxL([X|Xs],M):-maxL(Xs,M),X=<M.
```

34. Exerc: Usando um acumulador, defina um predicado maxL(M,L). Inicie o acumulador com a cabeça da lista de entrada.
35. Exerc: Usando permutation e isOrdered defina um algoritmo de ordenação sortx/2, tal que: ?-sortx([1,11,5,2], X). X=[1, 2, 5, 11] Solução:

```
sortx(L,S):-permutation(L,S),isOrdered(S).
```

36. Exerc: Faça um predicado insOrd/3, que insere um elemento numa lista mantendo -a ordenada. Faça duas regras: uma base e uma recursiva. ?-insOrd(4,[2,3,5,7],L). L=[2,3,4,5,7] Yes Solução:

```
insord(X,[],[X]).
insOrd(X, [Y|Ys],[X,Y|Ys]):- X<Y.
InsOrd(X, [Y|Ys],[Y|XYs]):- X>=Y,insOrd(X,Ys,XYs).
```

37. Exerc: Faça um predicado insOrd/3, que trabalha também com caracteres, assim ?-insOrd(c,[a,b,d,e],L). L=[a,b,c,d,e] Yes Solução:

```
insord(X,[],[X]).
insOrd(X, [Y|Ys],[X,Y|Ys]):- X@<Y.
```

InsOrd(X, [Y|Ys],[Y|XYs]):- X@>=Y,insOrd(X,Ys,XYs).

38. Exerc: Faça o predicado de ordenação pelo método inserção direta, insDir/2. Use o predicado insOrd/3. Enquanto a lista dada de entrada não for vazia, pegue a cabeça e insira na lista ordenada, que é um acumulador (inicializado como lista vazia).
39. Exerc: Faça um predicado que particiona/3 uma lista em duas, de tamanho igual se o número de elementos for par, senão uma delas terá um elemento a mais. Tire dois elementos de uma lista (se possível) e ponha cada um em uma lista resultado.
    ?- particiona([a,b,c,1,2,3,4],A,B). A=[a,c,2,4], B=[b,1,3] Yes Solução:
    particiona([X,Y|XYs],[X|Xs],[Y|Ys]):-particiona(XYs,Xs,Ys).
    particiona([X],[X],[]).
    particiona([],[],[]).
40. Exerc: Faça o predicado merge/3, que junta duas listas ordenadas em uma terceira, mantendo a ordem. Solução:
    merge1([X|Xs],[Y|Ys],[X|XYs]):-X@<Y,!,merge1(Xs, [Y|Ys],XYs).
    merge1([X|Xs],[Y|Ys],[Y|XYs]):-X@>=Y,!,merge1([X|Xs], Ys,XYs).
    merge1([],Ys,Ys):-!.
    merge1(Xs,[],Xs):-!.
    ?- merge1([a,b,b,k,z], [c,m,n,o], X). X=[a,b,b,c,k,,m,n,o,z], Yes

Note que as cláusulas merge1([],Ys,Ys):-!. e merge1(Xs,[],Xs):-!. devem necessariamente possuir corte, pois, caso contrário a pergunta merge1([],[],X) terá duas soluções.

41. Exerc: Faça um predicado de ordenação mergeSort/2. Use o particiona/3 e o merge/3. Dada uma lista, particiona-se alista em duas, chama-se o mergeSort para ordenar cada partição e com o merge juntamos os dois sub-resultados no resultado final. Considere a condição deparada, uma lista vazia ou com um elemento. Assim ?-mergeSort([1,11,5,2], X). X=[1, 2, 5, 11] Yes Solução:
    mergeSort([],[]):-!.
    mergeSort([X],[X]):-!.
    mergeSort(L,S):-particiona(L,X,Y),mergesort(X,Xo),mergeSort(Y,Yo),merge1(Xo,Yo,S).

1.11. Predicados com computação aritmética

O operador “is” do Prolog é similar ao “:=” do Pascal. Por exemplo, X1:=X+1 se escreve em Prolog X1 is X+1. **Cuidado**, em Prolog X is X+1 que equivale a X:=X+1 **sempre será falso**, pois uma variável não casa com dois valores.

42. Exerc: Faça um predicado enesimo/3 tal que: ?- enesimo(3,[a,b,c,d], X). X=c, Yes Solução:
    enezimo(N,[X|Xs],X):-N<=1.
    enezimo(N,[X|Xs],Y):-N>1, N1 is N-1, enezimo(N1,Xs,Y).
43. Exerc: Faça um predicado compr2/2, similar ao compr/2 tal que: ?- compr2([a,b,c,d], X). X=4, Yes Solução:
    compr2(L,C):-L=[],C=0.

```
compr2(L,C):-L=[X|Xs],compr2(Xs,C1),C is C1+1.
```

2. Refatoração: Otimização de predicados

Neste capítulo vimos que existem diferentes técnicas para se fazer uma mesma solução, por exemplo, com e sem acumulador. Agora vamos explorar a técnica de refatoração (reescrever um código que já está funcionando buscando clareza, reuso ou desempenho). São discutidas duas técnicas: desempenho linear e recursão na cauda. Refatoração buscando desempenho linear. Um predicado para manipular listas deve ter, se possível, uma desempenho linear em relação ao comprimento da lista. Vamos mostrar como testar e indicar o caminho para obter esta performance linear.

```
maxL0([X],X).
maxL0([X|Xs],X):-maxL0(Xs,M),X >M,!.
maxL0([X|Xs],M):-maxL0(Xs,M),X=<M,!.
%%
maxL00([X],X).
maxL00([X|Xs],M):-maxL00(Xs,M1),(X>M1,M=X,!; X=<M1,M=M1).
%%
maxL([X|Xs],M):-maxL(Xs,X,M).
maxL([],M,M).
maxL([X|Xs],ACC,M):- ACC>X,!,maxL(Xs,ACC,M).
maxL([X|Xs],_,M):- maxL(Xs,X,M).
%%
gL(L,N):-!,findall(X,(between(1,N,I),X is random(1000)),L).
```

Aqui temos três versões para o predicado que retorna o valor máximo de uma lista. E temos um predicado que gera uma lista de valores randômicos de tamanho N, para testar as versões do predicado. Uma unidade de medida de tempo de predicados é o número de inferências, que é uma medida de tempo, independente de cpu, complementar ao tempo em segundos da CPU. Uma inferência é associada a uma operação de unificação, por exemplo, um teste de igualdade é uma inferência e a ligação de uma variável com um valor também é uma inferência. Associada a esta medida temos a unidade chamada Lips (logical inferences per second), que define a velocidade do Prolog numa dada máquina, dividindo o número de inferências pelo tempo em segundos.

A primeira versão, maxL0, tem um comportamento exponencial: vai crescendo exponencialmente em relação ao comprimento da lista. Pois para uma lista de 10 elementos executa com 416 inferências; para 20 salta para 467.004; para 30 dispara para 126 milhões; com os tempos de 0.0seg, 0.14seg e 39.69 segundos. Projetando estes valores num gráfico temos uma curva exponencial. Um predicado com comportamento exponencial se possível deve ser refatorado.

A segunda versão, maxL00, já tem um comportamento linear em relação ao tamanho da lista. Em média de 4,9 inferências por elemento. E a terceira, maxL que usa um acumulador tem o melhor tempo, com uma média de 2 inferências por elemento.

```
?- gL(L,10), time(maxL0(L,M)).
  416 inferences, 0.00 CPU in 0.00 seconds (?% CPU, Infinite Lips)
?- gL(L,20), time(maxL0(L,M)).
  467,004 inferences, 0.13 CPU in 0.14 seconds (88% CPU, 3736032 Lips)
?- gL(L,30), time(maxL0(L,M)).
  126,353,408 inferences, 34.97 CPU in 39.69 seconds (88% CPU, 3613324 Lips)
%%
?- gL(L,100), time(maxL00(L,M)).
  490 inferences, 0.00 CPU in 0.00 seconds (?% CPU, Infinite Lips)
?- gL(L,10000), time(maxL00(L,M)).
  49,990 inferences, 0.02 CPU in 0.02 seconds (98% CPU, 3199360 Lips)
%%
?- gL(L,100), time(maxL(L,M)).
  204 inferences, 0.00 CPU in 0.00 seconds (?% CPU, Infinite Lips)
?- gL(L,10000), time(maxL(L,M)).
  20,022 inferences, 0.00 CPU in 0.00 seconds (?% CPU, Infinite Lips)
```

2.1.1. Recursividade na cauda

Normalmente a versão de um programa com acumulador pode ser recursiva na cauda. Isto é, a cabeça da cláusula é chamada no final do corpo, daquela mesma cláusula; a chamada é a última parte do corpo. Das três versões acima só o maxL é recursivo na cauda. Um predicado recursivo na cauda utiliza menos recursos de memória para executar, tendo conseqüências positivas na redução do tempo. Veja abaixo os tempos em segundos quando aumentamos o tamanho da lista para 100 mil e depois para 1 milhão de elementos. Para 100 mil o maxL recursivo na cauda levou 0.5 segundos, enquanto que o maxL0 não recursivo na cauda levou 0.15 segundo. Já, para 1 milhão, o maxL levou 0.140 segundos, porém o maxL0 levou 2.813 segundos. Este tempo a mais é para gerenciar os recursos de pilha de processamento e coleta de lixo, que num predicado recursivo na cauda não existe.

```
9 ?- gL(L,1000000), time(maxL00(L,M)).
% 2,999,992 inferences, 2.609 CPU in 2.813 seconds (93% CPU, 1149698 Lips)
L = [710, 683, 123, 164, 181, 677, 757, 191, 848|...],
M = 999.
10 ?- gL(L,1000000), time(maxL(L,M)).
% 2,000,000 inferences, 0.141 CPU in 0.140 seconds (100% CPU, 14222222 Lips)
L = [729, 733, 924, 763, 529, 11, 649, 992, 651|...],
M = 999.
```

Em resumo um predicado recursivo na cauda consome menos recursos e é mais robusto. Portanto, sempre que um predicado é executado um grande número de vezes ou que faz parte de uma biblioteca importante, ele deve ser reescrito com recursão na cauda e deve passar no teste de performance linear.

2.1.2. Mais sobre inferências

As inferências são contabilizadas somente para os predicados que são executados na máquina abstrata do Prolog. Portanto, quando estamos chamando um predicado de uma biblioteca em C, o número de inferências não representará toda a realidade. Segue um código que é usado para ilustrar estes pontos.

```
rsort(X,S):-sort(X,S1),reverse(S1,S).
rmsort(X,S):-msort(X,S1),reverse(S1,S).
%%
front(N,L,Lo):-length(Lo,N),append(Lo,_,L).
 last(N,L,Lo):-length(Lo,N),append(_,Lo,L).
%%

?- sort([a,a,a,b,c,d,d,d,d],S).
  S = [a, b, c, d]
?- rsort([a,a,a,b,c,d,d,d,d],S).
  S = [d, c, b, a]
?- msort([a,a,a,b,c,d,d,d,d],S).
  S = [a, a, a, b, c, d, d, d, d]
?- rmsort([a,a,a,b,c,d,d,d,d],S).
  S = [d, d, d, d, c, b, a, a, a]
```

No SWI-Prolog, utilizamos muito dois predicados de ordenação de listas: sort e msort. Ambos ordenam uma lista na ordem crescente, o primeiro remove os elementos duplicados e o segundo não remove. Acima codificamos duas versões que retornam o resultado em ordem decrescente, simplesmente invertendo a lista. Veja as execuções abaixo.

Os predicados sort e msort são implementados numa biblioteca em C, portanto quando chamamos eles o número de inferências é apenas uma. Já o rsort usa o reverse da biblioteca do Prolog, portanto as inferências são contadas: uma inferência para cada elemento de uma lista. O mesmo vale para o msort.

```
gL(L,100000), time(sort(L,M)).
  1 inferences, 0.08 CPU in 0.08 seconds (98% CPU, 13 Lips)
?- gL(L,100000), time(rsort(L,M)),length(M,LM).
  1,004 inferences, 0.06 CPU in 0.06 seconds (99% CPU, 16064 Lips)
  LM = 1000
%%
?- gL(L,100000), time(msort(L,M)),length(M,LM).
  1 inferences, 0.25 CPU in 0.25 seconds (100% CPU, 4 Lips)
  L = [399, 335, 758, 172, 194, 503, 747, 234, 915|...]
  M = [0, 0, 0, 0, 0, 0, 0, 0, 0|...]
  LM = 100000
?- gL(L,100000), time(rmsort(L,M)),length(M,LM).
  100,004 inferences, 0.45 CPU in 0.45 seconds (100% CPU, 220698 Lips)
```

```
L = [259, 560, 928, 397, 887, 348, 373, 527, 191|...]
M = [999, 999, 999, 999, 999, 999, 999, 999, 999|...]
LM = 100000
```

As perguntas abaixo mostram a execução do front e last que pegam respectivamente N elementos iniciais ou finais de uma lista. Estes predicados usam o append. Vemos que pegar elementos na frente da lista é bem mais eficiente. Logo para otimizar programas devemos sempre que possível trabalhar a partir do início de uma lista.

```
gL(L,10000),time(front(1000,L,Lo)).
  1,004 inferences, 0.00 CPU in 0.00 seconds (?% CPU, Infinite Lips)
?- gL(L,10000),time(last(1000,L,Lo)).
  9,004 inferences, 0.45 CPU in 0.47 seconds (97% CPU, 19871 Lips)
```

Por fim temos o predicado gL que depende dos predicados findall e between. Abaixo temos o predicado forall(between(1,5,I), ...) que, com dois parâmetros, imita um predicado for de uma linguagem procedural. O findall(EXP,PRED,LISTA) é similar ao forall, porém tem três parâmetros; o primeiro é uma EXPressão que é coletada em LISTA a cada vez que o PREDicado é executado. O between(INIC,FIM,I) retorna por retrocesso todos os valores da faixa declarada. Seguem alguns testes.

```
?- between(1,3,X).
  X = 1 ; X = 2 ; X = 3 ; No
?- forall(between(1,5,I),(write(I*I),write(' '))).
  1*1 2*2 3*3 4*4 5*5
?- findall(Y*Y,(between(1,5,Y)),L).
  L = [1*1, 2*2, 3*3, 4*4, 5*5]
```

O funcionamento interno destes predicados é apresentado no capítulo sobre estruturas de controle.

2.2. Predicados para listas do SWI-Prolog

Existe uma convenção para definir a passagem de parâmetros em protótipos de predicados em Prolog. Por exemplo, em reverse(+List1, -List2), o ("+") indica parâmetro de entrada e o ("-") parâmetro de saída.

```
?- reverse([a,b],X).
  X = [b, a] ; No
?- reverse(X,[a,b]).
  X = [b, a] ;
  ERROR: Out of global stack
?- reverse(X,Y).
  X = [], Y = [] ;
```

```
  X = [_G228], Y = [_G228] ;
  X = [_G228, _G234], Y = [_G234, _G228] ;
...
```

Como vemos acima, podemos usar o predicado indevidamente e mesmo assim em muitos casos ele funciona. O uso correto, pela especificação do protótipo, é o da primeira pergunta. Outra anotação de passagem de parâmetros é o (?), que define que o parâmetro pode ser tanto de entrada como de saída, como length(?List, ?Int).

```
?- length([a,b],T).
  T = 2 Yes
?- length(L,2).
  L = [_G216, _G219] Yes
?- length(L,I).
  L = [], I = 0 ;
  L = [_G231], I = 1 ;
...
```

Segue a relação dos protótipos dos predicados de manipulação de listas, predefinidos no SWI-Prolog:

- is_list(+Term) retorna sucesso, se Term é uma lista, vazia ou com elementos. Se Term for uma variável, o predicado falha. Este teste é feito na primeira cláusula do predicado, que é definido abaixo.

```
is_list(X) :-var(X),!,fail.
is_list([]).
is_list([_|T]) :-is_list(T).
```

- memberchk(?Elem, +List) é similar ao member, mas retorna somente uma solução. List deve ser dada como entrada.
- delete(+List1, ?Elem, ?List2) remove todas as ocorrências do elemento Elem de List1 retornando List2. List1 deve ser dada como entrada.
- nth0(?Index, ?List, ?Elem) retorna o enésimo elemento de uma lista. A contagem começa no 0.
- nth1(?Index, ?List, ?Elem) idem, a contagem começa no 1.

Os predicados abaixo, são os mesmos estudados neste capítulo:

```
append(?List1, ?List2, ?List3)
member(?Elem, ?List)
select(?Elem, ?List, ?Rest)
last(?Elem, ?List)
reverse(+List1, -List2)
flatten(+List1, -List2)
length(?List, ?Int)
```

```
sumlist(?List, ?Int)
merge(+List1, +List2, -List3)
```

Ainda existe o predicado maplist que mapeia um predicado aritmético, no exemplo, plus/3, para todos os elementos de uma lista, retornando uma nova lista.

Segue um teste.

```
?- plus(4,5,X).
  X =9
?- maplist(plus(2),[1,2,3],L).
  L = [3, 4, 5]
```

O funcionamento interno deste predicado é apresentado no capítulo sobre estruturas de controle.

3. Programação procedural vs lógica (Opcional)

Esta seção é destinada a programadores experientes em programação imperativa, por exemplo, Pascal e C++.

Aqui fazemos um paralelo entre a programação procedural e lógica, aprofundando a discussão sobre a codificação de predicados usando a técnica dos acumuladores. Abaixo temos a versão em Prolog do programa que calcula o comprimento de uma lista, com e sem acumulador.

```
% sem acumulador
compr([],0).
compr([X|Xs],C):-compr(Xs,C1), C is C1+1.

% com acumulador
comprAcc(L,C):-compr3(L,0,C).
compr3([X|Xs],Acc,C):-Acc1 is Acc+1, compr3(Xs,Acc1,C).
compr3([],Acc,Acc).
```

Abaixo, codificamos estes predicados como procedimentos em Pascal. Inicialmente definimos um tipo Lista e as funções cauda e eh_vazia para podermos manipular a lista. Em seguida, temos três versões destes predicados. A primeira é para calcular o comprimento sem usar um acumulador, que é bem próxima da versão em Prolog, a diferença está no açúcar sintático que é necessário no Pascal.

A segunda corresponde ao predicado com acumulador, mas sem recursividade. Aqui a variável acc é local ao procedimento, e é atualizada a cada iteração. A iteração é executada por um while; após o while retornamos o valor acumulado. A terceira é programada de forma recursiva. Até no Pascal, ao usarmos recursividade, temos que passar o acumulador como um

parâmetro. Esta versão Pascal também é bem próxima do predicado Prolog, salvo pelo açúcar sintático.

```
Type  Lista = record cab:info; pcauda:pLista; end;
     pLista = pointer to Lista;
function cauda(L:pLista):pLista; begin cauda := L^.pcauda; end;
function eh_vazia(L:pLista):bool; begin eh_vazia := L=nil; end;
% sem acumulador recursiva
procedure compr(L:pLista; var resultC:int);
begin
  if   eh_vazia(L) then result:=0
  else begin compr(cauda(L),C1); resultC:=C1+1; end;
end;
% com while iterativa
procedure comprAccI(L:tLista; var result:int);
begin
acc:=0;
while not eh_vazia(L)
   begin  L := cauda(L); acc := acc+1; end
result := acc;
end;
% com acumulador recursiva
procedure comprAcc(L:tLista; var result:int);

begin  camprAcc0(L,0,result); end;
procedure comprAcc0(L:tLista; acc:int; var result:int);
begin
  if   eh_vazia(L) then result:=acc
  else begin acc1 := acc+1; comprAcc0(cauda(L),acc1, result); end;
end;
```

Podemos agora resumir as diferenças e similaridades entre os códigos Prolog e Pascal, em várias dimensões:

- predicado vs. procedure: cada predicado é definido como um procedimento imperativo;
- cláusulas vs. if: cada cláusula de um predicado corresponde a um caminho de um comando condicional if;
- parâmetros vs. acumuladores: num programa iterativo, a variável que acumula um resultado corresponde a um parâmetro num predicado Prolog (ou numa procedure recursiva);
- itens de dados: listas em Prolog são estruturas de dados com ponteiros em Pascal; variáveis simples e constantes são os mesmos objetos em ambas as linguagens;
- seqüência, seleção e repetição: seqüências de comandos em Pascal são separadas por (;), em Prolog por (,); um comando condicional if é codificado em Prolog por diferentes cláusulas de um mesmo predicado; as repetições while são codificadas por recursividade;

- atribuição vs unificação: A atribuição result:=acc em Pascal corresponde a uma combinação de passagem de parâmetros e unificação em Prolog: a cláusula compr3([],Acc,Acc) pode ser rescrita como compr3([],Acc,Result) :- Result = Acc.