Programas Prolog
para Processamento de Listas
e
Aplicações

Maria Carolina Monard
Universidade de São Paulo / ILTC
Instituto de Ciências Matemáticas de São Carlos
Departamento de Ciências de Computação e Estatística

Maria do Carmo Nicoletti
Universidade Federal de São Carlos / ILTC
Departamento de Computação

Janeiro 1993 - Versão 2.0

Contents

1 Prefácio

A linguagem de programação lógica Prolog surgiu na década de 70 e ganhou popularidade nos últimos anos através, principalmente, de seu uso em aplicações de computação simbólica.

Existem vários livros que abordam os fundamentos teóricos de programação lógica; não é difícil, entretanto, encontrar livros específicos da linguagem Prolog, onde a linguagem é introduzida através de problemas e de suas soluções em Prolog.

O aumento significativo de publicações que tratam de Prolog justifica-se também pelo aumento do número de cursos que utilizam esta linguagem, tanto a nível de graduação, quanto a nível de pós-graduação.

Lembrando que uma linguagem pode ser classificada como procedural, funcional e/ou lógica, ao analisarmos o curriculum de nossos cursos de computação, verificamos que primeiro são ensinadas várias linguagens procedimentais, tais como Pascal, Cobol, Fortran, etc., e depois linguagens do tipo Apl, Lisp, Prolog, etc.

Temos observado, após vários anos de experiência com ensino em cursos de computação, que o estilo procedimental de programação, talvez por ser o primeiro ensinado, interfere, de certa forma, na correta aprendizagem da linguagem Prolog por parte dos estudantes.

Constatamos, por outro lado, que tal interferência pode ser minimizada se durante o aprendizado de Prolog, os estudantes forem submetidos à resolução de conjuntos de problemas semelhantes. Motivados por tal constatação, decidimos preparar uma série de notas em programação Prolog, cada uma delas tratando classes de problemas semelhantes, e desenvolvendo para cada um deles a resolução em Prolog.

Já foram publicadas pelo Instituto de Lógica Filosofia e Teoria da Ciência – ILTC – as notas dedicadas a processamento de listas [Monard 88] e árvores [Monard 89]. Devido à grande receptividade dessas notas, resolvemos ampliá-las. Este fascículo é uma ampliação da Nota *Programas Prolog para Processamento de Listas*. Consideramos que a discussão de algumas aplicações que usam listas é motivante do ponto de vista didático, pois contribui para fixar a forma de programar em Prolog usando essa estrutura. Deve ser ressaltado que os problemas aqui apresentados se encontram espalhados nos vários livros citados nas referências. O que pretendemos nesta Nota é reunir e discutir esses numerosos problemas típicos, apresentando para cada um deles uma solução.

A fim de resolver os problemas aqui tratados, o leitor deverá ter um conhecimento básico da linguagem Prolog. É importante também que tente resolver os problemas e executar seus programas antes de consultar as soluções propostas nesta Nota.

A sintaxe do Prolog utilizada é a sintaxe de Edinburgh, especificamente utilizamos o Arity-Prolog [Arity 90]. Há implementações de Prolog que usam outras sintaxes. Como a sintaxe da linguagem é muito simples, torna-se fácil a expressão de um mesmo programa em diferentes sintaxes.

2 Introdução

Um programa Prolog consiste de:

- declaração de *fatos* a respeito de objetos de dados e suas relações;
- definição de *regras* a respeito de objetos de dados e suas relações;
- interrogação a respeito de objetos de dados e suas relações.

A figura 1 mostra a classificação dos objetos de dados em Prolog. O Prolog reconhece o tipo de um objeto de dado em um programa, pela sua forma sintática. Isso é possível devido ao fato da sintaxe do Prolog especificar formas diferentes para cada tipo de objeto de dado.

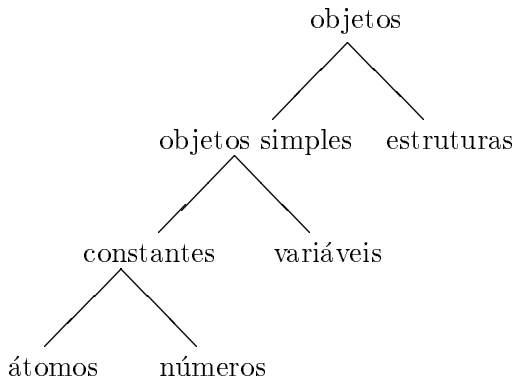


Figure 1: Objetos de Dados em Prolog

Átomos são cadeias compostas pelos seguintes caracteres:

- letras maiúsculas: `A`, `B`,...,`Z`
- letras minúsculas: `a`, `b`,..., `z`
- dígitos: `1`, `2`,..., `9`
- caracteres especiais, tais como: `*`, `+`, `-`, `>`, `<`, `_`, `=`, `:`, `.`, `&`, `~`.

Átomos podem ser construídos de três maneiras:

1. cadeias de letras, dígitos e o caractere `_`, começando com uma letra minúscula. Por exemplo: `x_y`, `maria`, `curso_de_IA`.
2. cadeias de caracteres especiais tais como: `<--->`, `::=`, `...`

   O uso de átomos desta forma exige algum cuidado, uma vez que alguma cadeia de caracteres especiais já têm um significado pré-definido, como é o caso do `::=`.

3. cadeias de caracteres entre apóstrofos. Isto é interessante quando se deseja, por exemplo, ter um átomo cuja primeira letra é a letra maiúscula. Colocando-o entre apóstrofos ele é diferenciado variáveis, por exemplo:
   `Maria`.

Números em Prolog incluem números inteiros e números reais. A sintaxe dos números inteiros é bem simples, como por exemplo: `1`, `-20`.

O tratamento de números reais depende da implementação Prolog utilizada; de qualquer maneira, eles podem ser tão simples quanto: `1.22`, `-0.2908`. É importante lembrar que o uso de números reais não é o forte de programação Prolog, uma vez que tal linguagem é tipicamente voltada para computação simbólica.

Variáveis são cadeias de letras, dígitos e caracteres `_`, sempre começando com letra maiúscula ou com o caractere `_`. Por exemplo: `X`, `Mapa_da_Mina`, `_nome`.

Estruturas ou objetos estruturados são objetos de dados que têm vários componentes, podendo cada um deles, por sua vez, ser uma estrutura.

Por exemplo, uma data pode ser vista como uma estrutura com três componentes: dia, mês e ano. Embora compostas por vários componentes, estruturas são tratadas no programa como objetos simples. A combinação dos componentes em um objeto simples é feita através de um *funtor*.

No caso específico da data, o funtor escolhido pode ser, por exemplo, `data` e pode-se escrever a data 7 de julho de 1953 como:

```
data(7,julho,1953)
```

Os componentes `7`, `julho` e `1953` são constantes (dois inteiros e um átomo).

Sintaticamente todos os objetos de dados em Prolog são termos. Tanto `julho` quando `data(7,julho,1953)` são termos.

Todos objetos estruturados podem ser representados como árvores, como mostrado na Figura 2, para a estrutura `data(7,julho,1953)`.

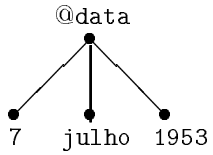


Figure 2: Representação em Árvore da Estrutura `data(7,julho,1953)`

A raiz da árvore é o funtor e os filhos da raiz são os componentes. Se um componente for também uma estrutura, ele será representado como uma sub-árvore da árvore que representa toda a estrutura. Todo objeto estruturado em Prolog é, pois, uma árvore representada no programa por um termo. Um funtor é definido por:

1. nome, cuja sintaxe é a mesma de átomos;

2. aridade, que corresponde ao número de argumentos

3 Listas

Uma das estruturas mais simples em Prolog é a estrutura de listas, muito comum em programação não numérica. Ela é uma seqüência ordenada de elementos e pode ter qualquer comprimento. É uma seqüência ordenada no sentido de que a ordem dos elementos na seqüência é importante, isto é, pode-se fazer referência ao primeiro elemento, ao i-ésimo ou ao último elemento de uma lista.

Como comentado anteriormente, todos os objetos estruturados em Prolog são árvores; listas não fogem à regra.

A cabeça de uma lista pode ser qualquer objeto Prolog (tal como uma árvore ou uma lista vazia), e a cauda deve ser uma lista. A cabeça e a cauda são combinadas numa estrutura através de um funtor especial. Este funtor depende da implementação Prolog; geralmente tal funtor é o ponto `.`

```
.(Cabeca,Cauda)
```

Desde que `Cauda` por sua vez é uma lista, ela ou é a lista vazia, ou tem sua própria cabeça e cauda.

É importante notar que a notação utilizando o funtor `.` pode produzir expressões difíceis de serem entendidas. Por esta razão, Prolog fornece uma notação alternativa para listas, o que possibilita uma lista ser escrita como uma sequência de itens entre conchetes.

3.1 Sintaxe para Listas

Na notação utilizando colchetes, a lista vazia é denotada por `[]`; a lista que tem cabeça `X` e cauda `Y` é denotada por `[X|Y]`. Dentro da lista, os elementos são separados por vírgulas. Por exemplo:

Lista	Cabeça	Cauda
`[gosto,de,vinho]`	`gosto`	`[de,vinho]`
`[[3],5,[2,7]]`	`[3]`	`[5,[2,7]]`
`[X,Y|Z]`	`X`	`[Y|Z]`
`[[o,gato]]`	`[o,gato]`	`[]`

É importante notar, contudo, que o uso de colchetes é apenas uma melhoria notacional, pois internamente listas são representadas como árvores binárias, como mostrado nas Figuras 3 e 4.

Quem programa em Prolog pode usar ambas notações; entretanto, a notação usando colchetes é normalmente preferida por ser mais simples — ela é apenas uma melhoria notacional, uma vez que internamente listas são representadas como árvores binárias.

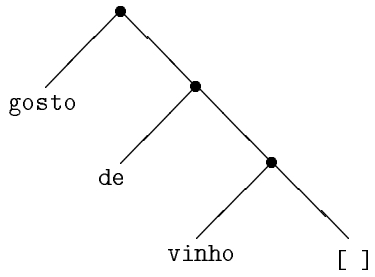


Figure 3: Representação em Árvore da Lista `[gosto,de,vinho]`

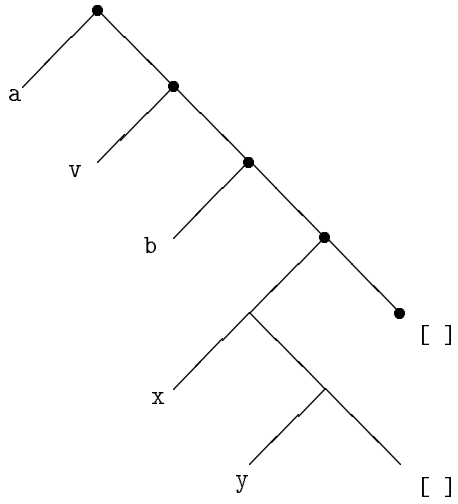


Figure 4: Representação em Árvore da Lista `[a,v,b,[x,y]]`

3.2 Unificação em Listas

Em Prolog, a mais importante operação envolvendo termos é conhecida como *unificação*[1].

Dados dois termos, diz-se que eles unificam se:

1. são indênticos, ou
2. as variáveis em ambos os termos podem ser instanciadas em objetos, de maneira que após a substituição das variáveis por esses objetos, os termos se tornam idênticos

As regras gerais que regem se dois termos **R** e **S** unificam são:

[1]Na realidade, o que está sendo chamado de unificação neste documento corresponde ao termo *matching*. Vários autores preferem o uso de matching e evitam a palavra unificação devido ao fato que o matching, por razões de eficiência, está implementado em sistemas Prolog de uma maneira que não corresponde exatamente ao algoritmo de unificação em Lógica.

1. se **S** e **T** são constantes, então **S** e **T** unificam se e só se são o mesmo objeto
2. se **S** for uma variável e **T** for qualquer termo, então unificam, e **S** é instanciado com T; vice-versa, com a variável **T** instanciada com **S**,
3. se **S** e **T** são estruturas, elas unificam se e só se **S** e **T** têm o mesmo funtor principal, e todos os elementos correspondentes unificam

Se os termos não unificam, diz-se que o processo de unificação falha. Entretanto, se eles unificam, diz-se que o processo é bem sucedido e as variáveis em ambos os termos são instanciadas com valores que tornam os termos idênticos.

A unificação de uma lista `[X|Y]`, onde `X` e `Y` são variáveis, com uma lista do tipo

`[a1,a2,a3,...,an]`

resulta na substituição:

`{X/a1,Y/[a2,a3, ... an]}` se n > 1

e na substituição:

`{X/a1,Y/[]}` se n $= 1$

A seguir são mostrados vários exemplos de unificação de listas. Sempre que houver dúvida com relação à unificação de termos, Prolog deve ser interrogado com

```
?- Termo1 = Termo2.
```

Por exemplo:

```
?- [X|Y] = [a,b,c].
X = a
Y = [b,c] ->;
no
```

Lista 1	Lista 2	Unificação
[mesa]	[X|Y]	X/mesa Y/[ ]
[a,b,c,d]	[X,Y|Z]	X/a Y/b Z/[c,d]
[[ana,Y]|Z]	[[X,foi],[ao,cinema]]	X/ana Y/foi Z/[[ao,cinema]]
[ano,bissexto]	[X,Y|Z]	X/ano Y/bissexto Z/[]
[ano,bissexto]	[X,Y,Z]	não unifica
[data(7,Z,W),hoje]	[X|Y]	X/data(7,Z,W) Y/hoje
[data(7,W,1993),hoje]	[data(7,X,Y),Z]	X/W Y/1993 Z/hoje

3.3 Busca Recursiva

Frequentemente é necessário procurar por algum termo que faz parte de uma estrutura Prolog; isto resulta em uma busca recursiva. Como exemplo, considere a lista dos nomes dos filhos de Júpiter, rei de Olimpo, e de sua esposa Juno: Ilítia, Marte, Hebe e Vulcano.

```
[ilitia,marte,hebe,vulcano]
```

Para verificar se um dado nome está numa lista, deve-se verificar se ele é a cabeça da lista, ou se ele está na cauda da lista. Se o final da lista for atingido, significa que a busca não foi bem sucedida.

Para escrever o programa Prolog que verifica se um elemento faz parte de uma lista, deve-se decidir qual o nome da relação entre o elemento e a lista de elementos. Chamando esta relação de `pertence`, é agora necessário definir o programa `pertence(X,Y)`, que será bem sucedido se `X` for um dos elementos da lista `Y`. O número de argumentos de uma relação (aqui referenciada como predicado ou programa) Prolog é denominado *aridade*. Assim, o predicado acima, pode ser referenciado como `pertence/2`.

A primeira condição que especifica que o elemento `X` pertence à lista que o tem como cabeça pode ser escrita como:

```
pertence(X,[X|_]).
```

onde a variável anônima `"_"` é usada para representar a cauda da lista, uma vez que não há interesse em saber qual é esta cauda. O uso da variável anônima não apenas torna os programas mais legíveis como também, em alguns casos, evita que o interpretador ou compilador Prolog realize tarefas desnecessárias. É importante entender bem

este conceito: uma variável anônima pode (e deve) ser usada quando é tentada uma unificação, mas não há necessidade de que Prolog lembre as instanciações das variáveis desta unificação.

Voltando ao exemplo, a segunda condição que especifica que o elemento `X` pertence à lista, caso ele pertença à sua cauda, pode ser escrita como:

```
pertence(X,[_|Y]) :- pertence(X,Y).
```

que mostra a recursão do programa Prolog `pertence`. Sempre que um programa recursivo é definido, deve-se procurar pelas condições limites — ou de parada — e pelo caso recursivo.

No predicado `pertence/2` existem duas condições limite: o elemento que se procura está ou não na lista.

A primeira condição de `pertence/2` é reconhecida pela primeira cláusula, o que terminará a busca se o primeiro argumento de `pertence/2` unifica com a cabeça do segundo argumento.

A segunda condição limite ocorre quando o segundo argumento é a lista vazia. Cada vez que a segunda cláusula de `pertence/2` é ativada, a lista na qual o elemento deve ser procurado é menor — a cauda de uma lista é sempre menor do que a lista original. Eventualmente, uma das duas situações seguintes acontecerá: ou a primeira cláusula será satisfeita ou `pertence/2` terá como segundo argumento uma lista vazia. Acontecendo uma dessas duas situações, a recorrência terminará. A primeira condição limite é reconhecida por um fato — primeira cláusula. A segunda condição limite não é reconhecida por nenhuma cláusula e, consequentemente, `pertence/2` irá falhar.

```
pertence(X,[X|_]).
pertence(X,[_|Y]) :- pertence(X,Y).
```

Após a definição do programa, é possível interrogá-lo. Por exemplo:

```
?- pertence(a,[l,a,b]).
yes

?- pertence(a,[hoje,ontem]).
no

?- pertence(X,[a,b,c]).
X = a ->;
X = b ->;
X = c ->;
no
```

Porém, se as interrogações forem:

```
?- pertence(a,X).
```
ou
```
?- pertence(X,Y).
```

deve-se observar que cada uma delas tem "infinitas" respostas, uma vez que existem infinitas listas que validam estas interrogações para o programa `pertence`.

O quadro abaixo mostra as possíveis instanciações dos dois argumentos do programa `pertence/2`. Nele são analisados se esses argumentos devem ou não estar instanciados quando o programa é ativado, a fim de que todas as soluções sejam obtidas, de maneira a garantir que o seu número seja finito.

<arg-1>	<arg-2>	pertence(<arg-1>,<arg-2>)
(+) instanciado	(+) instanciado	sim
(-) livre	(+) instanciado	sim
(+) instanciado	(-) livre	não
(-) livre	(-) livre	não

3.4 Contrôle do Retrocesso — *Backtracking*

O Prolog, na busca de uma solução, sempre faz automaticamente o retrocesso. O retrocesso automático é um poderoso recurso de programação, uma vez que libera o programador de codificá-lo explicitamente. Entretanto, o retrocesso não controlado, pode tornar um programa ineficiente. Consequentemente, algumas vezes torna-se necessário controlá-lo, ou mesmo, prevenir que aconteça. Isto é feito através o uso de um predicado especial, chamado **corte**, notado por `!`

Duas razões são determinantes para o uso do corte:

1. tornar o programa mais rápido. O uso do corte previne o Prolog de explorar alternativas que não contribuem para a solução, reduzindo o espaço de busca através de uma poda dinâmica da árvore de busca;

2. fazer com que o programa ocupe menos espaço de memória. Um uso mais econômico da memória pode ser feito se pontos para possíveis retrocessos não precisarem ser guardados para um exame posterior.

O corte deve ser usado com cuidado — a maioria das vezes o seu uso pode ser interpretado apenas proceduralmente, o que fere o estilo declarativo de programação a qual o Prolog se propõe.

Dada uma função $f(x)$ definida por:

$$f(x) = \begin{cases} 0 \; se \; x < 3 \\ 2 \; se \; x \geq 3 \; e \; x < 6 \\ 4 \; se \; x \geq \; 6 \end{cases} \tag{1}$$

ela pode ser implementada em Prolog da seguinte forma:

```
f(X,0):- X < 3.            % versao 1
f(X,2):- 3 =< X, X < 6.
f(X,4):- 6 =< X.
```

Ao se interrogar Prolog com:

```
?- f(1,Y),2<Y.
```

na prova de f(1,Y), Y é instanciado com 0 e a prova de $2 < 0$ falha. Antes de admitir que a interrogação não é satisfeita, o Prolog tenta, através do retrocesso, as outras duas alternativas, que o programador já sabe, pela própria definição de $f(x)$ (1), que não serão satisfeitas.

As três regras a respeito da relação `f` são mutuamente exclusivas — no máximo, uma delas será satisfeita.

A fim de prevenir retrocesso inútil, pode-se explicitamente direcionar o Prolog a não realizá-lo, através do uso do corte. O programa anterior reescrito com cortes fica:

```
f(X,0):- X < 3,!.          % versao 2
f(X,2):- 3 =< X, X < 6,!.
f(X,4):- 6 =< X.
```

Se a mesma interrogação anterior for feita, o programa irá falhar na tentativa de provar $2 < 0$.

Nessa segunda versão do programa, o Prolog irá tentar o retrocesso, mas não além do ponto marcado com !, ou seja, as duas cláusulas alternativas não serão ativadas. Isto faz com que a versão 2 seja mais eficiente.

Se os cortes forem removidos do programa, ele continuará ainda produzindo os mesmos resultados. Neste caso em particular, a introdução dos cortes mudou apenas o significado procedural do programa — o seu significado declarativo permaneceu o mesmo.

Na literatura, um corte com tais características é conhecido como *corte verde* — sua adição ou remoção não afeta o significado do programa. Na *leitura* de um programa Prolog, os cortes verdes podem ser ignorados.

Por outro lado, existem também os chamados *cortes vermelhos* que afetam o significado declarativo de um programa. Eles tornam um programa difícil de ser entendido e devem ser usados cuidadosamente. A introdução e a remoção de um corte vermelho em um programa mudam o significado do programa, como é mostrado a seguir.

A função $f(x)$ definida em (1) pg.9 poderia também ser especificada da seguinte forma:

bf se $x < 0$ **então** $f(x) = 0$
bf caso contrário se $x < 6$ **então** $f(x) = 2$
caso contário $f(x) = 4$

que pode ser traduzida diretamente para Prolog como:

```
f(X,0):- X < 3,!.          % versao 3
f(X,2):- X < 6,!.
f(X,4).
```

Este programa produz os mesmos resultados e é ainda mais eficiente que a versão 2. Entretanto, a introdução dos cortes alterou o significado declarativo do programa, uma vez que se eles forem retirados, o programa produz soluções não corretas.

Seja o programa `pertence/2` que estabelece se um elemento está ou não em uma lista:

```
pertence(X,[X|_]).
pertence(X,[_|Y]) :- pertence(X,Y).
```

Essa versão do pertence é *não determinística* e percorre toda a lista. Por exemplo, se um dado elemento ocorre várias vezes na lista, então todas suas ocorrências podem ser encontradas através do retrocesso. Entretanto, se o interesse for o de simplesmente verificar se o elemento pertence à lista, ou seja, encontrar a sua primeira ocorrência, basta evitar o retrocesso assim que a primeira cláusula for bem sucedida. A versão determinística do programa é:

```
pertence_deterministico(Elem,[Elem|_]):-!.
pertence_deterministico(Elem,[_|Cauda]):-
             pertence_deterministico(Elem,Cauda).
```

Este programa, quando interrogado, irá fornecer apenas uma solução [2]. Por exemplo:

```
?- pertence_deterministico(Elem,[a,b,c,d,e,f]).
Elem = a;
no
```

3.5 Modos de Chamada

Para documentar as formas corretas de interrogação dos programas tratados nesta Nota, será usada a seguinte notação:[3]

```
modo(<arg-1>,<arg-2>,... ,<arg-n>).
```

onde:

[2] É importante observar que se o primeiro argumento deste programa não estiver instanciado, o programa simplesmente encontra o primeiro elemento da lista.

[3] O `modo` de chamada apresentado antes de cada programa, está pronto para ser inserido como comentário do programa correspondente, razão pela qual as palavras aparecem sem acentuação.

`<arg-i> = +` se `<arg-i>` deve estar instanciado
`<arg-i> = -` se `<arg-i>` deve ser variável livre
`<arg-i> = ?` se `<arg-i>` for qualquer dos casos acima

Assim, os possíveis modos de chamada do programa `pertence/2`, com resposta correta e número finito de soluções, são:

```
modo(+,+).
modo(-,+).
```

notado por

```
modo(?,+).
```

Deve ser ressaltado que, dependendo do contexto em que o programa `pertence` é usado, é possível usar os outros modos de chamadas com "infinitas" soluções. Pretende-se aqui simplesmente determinar o modo de chamada dos programas a fim de assegurar, após retrocesso — *backtraking* —, um número finito de respostas.

Nas próximas seções são mostrados vários programas básicos para processamento de listas, bem como seus modos de chamada e exemplos de execução [4]

4 Programas

4.1 Elementos de uma Lista

```
modo(?,+)

pertence(<arg-1>,<arg-2>)
<arg-1> : elemento
<arg-2> : lista
```

1. se `Elem` é o primeiro elemento da lista, então `Elem` pertence à lista.
2. seja a lista `[_|Cauda]`; se `Elem` pertence à lista, então `Elem` deve pertencer à cauda `Cauda`.

```
pertence(Elem,[Elem|_]).

pertence(Elem,[_|Cauda]):- pertence(Elem,Cauda).
```

Exemplo 4.1.1 `modo(-,+)`

[4] No que se segue, as respostas Prolog às interrogações, quando necessário, foram reformatadas.

```
?- pertence(Elem,[a,b,c,d,e,f]).
Elem = a ->;
Elem = b ->;
Elem = c ->;
Elem = d ->;
Elem = e ->;
Elem = f ->;
no
```

Exemplo 4.1.2 `modo(+,+)`

```
?- pertence(a,[a,b,c,d,e,f]).
yes

?- pertence(h,[a,b,c,d,e,f]).
no
```

4.2 Último Elemento de uma Lista

```
modo(+,?)

ultimo(<arg-1>,<arg-2>)
<arg-1> : lista
<arg-2> : elemento
```

1. se a lista tem apenas um elemento, este elemento é seu último elemento.
2. o último elemento de uma lista com mais de um elemento é o último elemento da cauda da lista.

```
ultimo([Elem],Elem).

ultimo([_|Cauda],Elem):- ultimo(Cauda,Elem).
```

Exemplo 4.2.1 `modo(+,-)`

```
?- ultimo([casa,bola,carro,tv],X).
X = tv ->;
no
```

Exemplo 4.2.2 `modo(+,+)`

```
?- ultimo([a,b,d,e,f,g],g).
yes
```

4.3 Elementos Consecutivos em uma Lista

```
modo(?,?,+)

consecutivos(<arg-1>,<arg-2>,<arg-3>)
<arg-1> : primeiro elemento
<arg-2> : segundo elemento
 <arg-3> : lista
```

1. dois elementos `E1` e `E2` são consecutivos em uma lista se eles forem o primeiro e o segundo elementos da lista, ou
2. se forem consecutivos na cauda da lista.

```
consecutivos(E1,E2,[E1,E2|_]).

consecutivos(E1,E2,[_| Cauda]):-
                    consecutivos(E1,E2,Cauda).
```

Exemplo 4.3.1 `modo(+,+,+)`

```
?- consecutivos(a,b,[d,e,f,g,a,b]).
yes
```

Exemplo 4.3.2 `modo(-,-,+)`

```
?-consecutivos(X,Y,[d,e,f,g,a,b,g]).
X = d
Y = e ->;
X = e
Y = f ->;
X = f
Y = g ->;
X = g
Y = a ->;
X = a
Y = b ->;
X = b
Y = g ->;
no
```

4.4 Soma dos Elementos de uma Lista Numérica

```
modo(+,?)

soma(<arg-1>,<arg-2>)
<arg-1> : lista numerica
<arg-2> : soma dos elementos
```

1. se a lista for a lista vazia, a soma de seus elementos é zero.
2. se a lista for `[Elem|Cauda]`, a soma de seus elementos é a soma dos elementos da cauda `Cauda` mais o primeiro elemento `Elem`.

```
soma([],0).

soma([Elem|Cauda],S) :- soma(Cauda,S1),
                        S is S1+Elem.
```

Pode ser observado que embora seja logicamente equivalente escrever a segunda cláusula como:

```
soma([Elem|Cauda],S):- S is S1 + Elem,
                       soma(Cauda,S1).
```

isto provocará um erro quando da execução de `soma/2` porque Prolog exige que `S1` e `Elem` estejam instanciados, ao tentar provar `S is S1 + Elem`.

Exemplo 4.4.1 `modo(+,-)`

```
?- soma([1,2,3,4,5,6],S).
S = 21 ->;
no
```

4.5 N-ésimo Elemento de uma Lista

```
modo(?,?,+)

n_esimo(<arg-1>,<arg-2>,<arg-3>)
<arg-1> : numero
<arg-2> : elemento
<arg-3> : lista
```

1. o primeiro elemento de uma lista é a cabeça da lista.
2. o n-ésimo elemento de uma lista é o (n-1)-ésimo elemento da sua cauda.

```
n_esimo(1,Elem,[Elem|_]).

n_esimo(N,Elem,[_|Cauda]):- n_esimo(M,Elem,Cauda),
                            N is M + 1.
```

Exemplo 4.5.1 `modo(-,-,+)`

```
?- n_esimo(N,Elem,[bit,byte,bloco,registro,arquivo]).
N = 1
Elem = bit ->;
N = 2
Elem = byte ->;
N = 3
Elem = bloco ->;
N = 4
Elem = registro ->;
N = 5
Elem = arquivo ->;
no
```

Exemplo 4.5.2 `modo(+,-,+)`

```
?- n_esimo(5,Elem,[um,dois,tres,quatro,cinco,seis]).
 Elem = cinco->;
 no
```

4.6 Número de Elementos de uma Lista

```
modo(+,?)

no_elementos(<arg-1>,<arg-2>)
<arg-1> : lista
<arg-2> : numero de elementos
```

1. o número de elementos de uma lista vazia é zero.
2. o número de elementos de uma lista `[Elem|Cauda]` é o número de elementos da cauda `Cauda`, mais um.

```
no_elementos([],0).

no_elementos([Elem|Cauda],N) :- no_elementos(Cauda,N1),
                                N is N1 + 1.
```

Exemplo 4.6.1 `modo(+,-)`

```
no_elementos([um,dois,tres,quatro],N).
N = 4 ->;
no
```

4.7 Pegar Elementos de uma Lista dada a Lista de suas Posições

```
modo(+,+,?)

pegar(<arg-1>,<arg-2>,<arg-3>)
<arg-1> : lista de posicoes
<arg-2> : lista
<arg-3> : lista resultante
```

1. pegar nenhum elemento de uma lista é obter uma lista vazia.
2. seja a lista de posições `[M|N]` e a lista de elementos `L`; pegar os elementos de `L` especificados na lista de posições significa pegar o `M`-ésimo elemento de `L` e os elementos especificados na cauda `N` da lista de posições.

```
pegar([],_,[]).

pegar([M|N],L,[X|Y]):- n_esimo(M,X,L),
                        pegar(N,L,Y).
```

Exemplo 4.7.1 `modo(+,+,-)`

```
?- pegar([1,2,3,6],[a,b,c,d,e,f],L).
L = [a,b,c,f] ->;
no
```

4.8 Retirar uma Ocorrência de um Elemento de uma Lista

```
modo(?,+,?),(+,?,+)

retirar_elemento(<arg-1>,<arg-2>,<arg-3>)
<arg-1> : elemento
<arg-2> : lista
<arg-3> : lista sem o elemento
```

1. se o elemento `Elem` for a cabeça da lista, então o resultado será a cauda da lista.
2. se o elemento `Elem` pertence à cauda `Cauda`, retirar `Elem` de `Cauda`.

```
retirar_elemento(Elem,[Elem|Cauda],Cauda).

retirar_elemento(Elem,[Elem1|Cauda],[Elem1|Cauda1]):-
                    retirar_elemento(Elem,Cauda,Cauda1).
```

Exemplo 4.8.1 `modo(-,+,-)`

```
?- retirar_elemento(X,[a,b,a,d,e,f],L).
X = a
L = [b,a,d,e,f] ->;
X = b
L = [a,a,d,e,f] ->;
X = a
L = [a,b,d,e,f] ->;
X = d
L = [a,b,a,e,f] ->;
X = e
L = [a,b,a,d,f] ->;
X = f
L = [a,b,a,d,e] ->;
no
```

Exemplo 4.8.2 `modo(+,+,+)`

```
?- retirar_elemento(a,[a,b,c,d,e,f],[b,c,d,e,f]).
yes
```

4.9 Retirar todas as Ocorrências de um Elemento de uma Lista

```
modo(+,+,?)

retirar_todas(<arg-1>,<arg-2>,<arg-3>)
<arg-1> : elemento
<arg-2> : lista
<arg-3> : lista sem qualquer ocorrencia do elemento
```

1. se a lista for a lista vazia o resultado é a lista vazia.
2. retirar todas as ocorrências do elemento `Elem` de uma lista `[Elem|Cauda]` implica também em retirar `Elem` da cauda `Cauda`.
3. retirar o elemento `Elem` de uma lista `[Elem1|Cauda]`, com `Elem` diferente de `Elem1`, significa obter uma lista `[Elem1|Cauda1]` onde `Cauda1` é uma lista sem ocorrências de `Elem`.

```
retirar_todas(_,[],[]).

retirar_todas(Elem,[Elem|Cauda],L):-
                      retirar_todas(Elem,Cauda,L).

retirar_todas(Elem,[Elem1|Cauda],[Elem1|Cauda1]):-
                      Elem \== Elem1,
                      retirar_todas(Elem,Cauda,Cauda1).
```

Deve ser observado que o Prolog possui vários tipos de igualdade entre termos. Até agora foi introduzida a igualdade baseada em unificação, escrita como: `X = Y`, que é verdade se X e Y unificam.

Outro tipo de igualdade já utilizada é `X is Y`, que é verdade se o valor da expressão aritmética Y unifica com X.

Algumas vezes é de interesse um tipo de igualdade estrita entre dois termos, denominada *igualdade literal*. Este tipo de igualdade está implementada como um outro predicado pré-definido `==`, escrito como um operador infixo: `X == Y`, que é verdade se os termos X e Y são idênticos, ou seja, têm exatamente a mesma estrutura, e seus componentes correspondentes na estrutura também são idênticos. Em particular, deve ser ressaltado que os nomes das variáveis devem ser os mesmos. Por exemplo:

```
?- gosta(maria,jose) == gosta(maria,X).
no

?- gosta(maria,jose) == gosta(maria,jose).
yes

?- gosta(X,jose) == gosta(Y,jose).
no
```

A relação complementar à `==` é `\==`, que significa não idêntico. Por exemplo,

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

?- gosta(X,jose) \== gosta(Y,jose).
yes
```

Exemplo 4.9.1 `modo(+,+,-)`

```
?- retirar_todas(a,[c,a,s,a],L).
L = [c,s] ->;
no
```

4.10 Retirar Elementos Repetidos de uma Lista

```
modo(+,?)

retirar_rep(<arg-1>,<arg-2>)
<arg-1> : lista
<arg-2> : lista sem elementos repetidos
```

1. uma lista vazia não tem elementos repetidos.

2. retirar os elementos repetidos de uma lista `[Elem|Cauda]` significa criar uma nova lista que possua somente um elemento `Elem` e sua cauda não tenha elementos repetidos.

```
retirar_rep([],[]).

retirar_rep([Elem|Cauda],[Elem|Cauda1]):-
                    retirar_todas(Elem,Cauda,Lista),
                    retirar_rep(Lista,Cauda1).
```

Exemplo 4.10.1 `modo(+,-)`

```
?-retirar_rep([a,b,a,b,a,b,a,a,b,a],L).
L = [a,b] ->;
no
```

Exemplo 4.10.2 `modo(+,+)`

```
?-retirar_rep([a,b,c,d,a,a,a],[a,b,c,d]).
yes

?-retirar_rep([a,b,a,b,a,b,c],[c,b,a]).
no
```

4.11 Inserir um Elemento em uma Lista

Um elemento pode ser inserido em uma lista como primeiro elemento da lista, ou então em qualquer outra posição da lista. Isto dá origem a duas possíveis implementações vistas a seguir.

4.11.1 Inserção do Elemento na Primeira Posição da Lista

```
modo(?,?,?)

inserir_1(<arg-1>,<arg-2>,<arg-3>)
<arg-1> : elemento
<arg-2> : lista
<arg-3> : lista com o elemento inserido
```

1. inserir um elemento `Elem` na lista `Lista` significa criar uma nova lista na qual `Elem` é a cabeça e `Lista` é a cauda.

```
inserir_1(Elem,Lista,[Elem|Lista]).
```

Exemplo 4.11.1 `modo(+,+,-)`

```
?- inserir_1(ana,[pedro,maria,jose],L).
L = [ana,pedro,maria,jose] ->;
no
```

4.11.2 Inserção do Elemento em Qualquer Posição da Lista

Se o elemento é inserido em qualquer posição da lista, o programa pode ser definido em função do programa `retirar_elemento`, visto na seção 4.8, pg.17.

```
modo(?,?,+),(+,+,?)

inserir_2(<arg-1>,<arg-2>,<arg-3>)
<arg-1> : elemento
<arg-2> : lista
<arg-3> : lista com o elemento inserido em qualquer posicao
```

1. inserir um elemento `Elem` na lista `Lista` significa encontrar uma nova lista `Lista1` da qual retirando-se o elemento `Elem` obtém-se a lista `Lista`.

```
inserir_2(Elem,Lista,Lista1) :-
                    retirar_elemento(Elem,Lista1,Lista).
```

Exemplo 4.11.2 `Modo(+,+,-)`

```
?- inserir_2(ana,[pedro,maria,jose],L).
L = [ana,pedro,maria,jose] ->;
L = [pedro,ana,maria,jose] ->;
L = [pedro,maria,ana,jose] ->;
L = [pedro,maria,jose,ana] ->;
no
```

Exemplo 4.11.3 `modo(-,-,+)`

```
?- inserir_2(X,Y,[1,2,3,4]).
X = 1
Y = [2,3,4] ->;
X = 2
Y = [1,3,4] ->;
X = 3
Y = [1,2,4] ->;
X = 4
Y = [1,2,3] ->;
no
```

4.12 Substituir um Elemento de uma Lista por um Outro Elemento

```
modo(?,?,+,?)

substitui(<arg-1>,<arg-2>,<arg-3>,<arg-4>)
<arg-1> : elemento a ser substituido
<arg-2> : elemento que substitui
<arg-3> : lista original
<arg-4> : lista resultante
```

1. se lista for a lista vazia, o resultado é a lista vazia.
2. substituir um elemento `X` por outro elemento `Y` em uma lista que tem `X` por cabeça e `L` por cauda, implica em substituir a cabeça da lista por `Y`, e em substituir `X` por `Y` em `L`.
3. substituir um elemento `X` por outro elemento `Y` em uma lista que não tem `X` por cabeça, implica em substituir `X` por `Y` na cauda.

```
substitui(X,Y,[],[]).

substitui(X,Y,[X|L],[Y|L1]):- substitui(X,Y,L,L1).

substitui(X,Y,[Z|L],[Z|L1]):- X \== Z,
                               substitui(X,Y,L,L1).
```

Exemplo 4.12.1 `modo(+,+,+,-)`

```
?- substitui(2,1000,[0,4,2,6,90,2,2,88,2,6],L).
L = [0,4,1000,6,90,1000,1000,88,1000,6] ->;
no
```

4.13 Duplicar os Elementos de uma Lista

```
modo(+,?)

duplica(<arg-1>,<arg-2>)
<arg-1> : lista original
<arg-2> : lista com os elementos dupicados
```

1. se a lista for a lista vazia, o resultado é a lista vazia.
2. duplicar os elementos de uma lista cuja cabeça é `X` e cauda `L`, significa duplicar a cabeça, e duplicar os elementos de `L`.

```
duplica([],[]).

duplica([X|L],[X,X|L1]) :- duplica(L,L1).
```

Exemplo 4.13.1 `modo(+,-)`

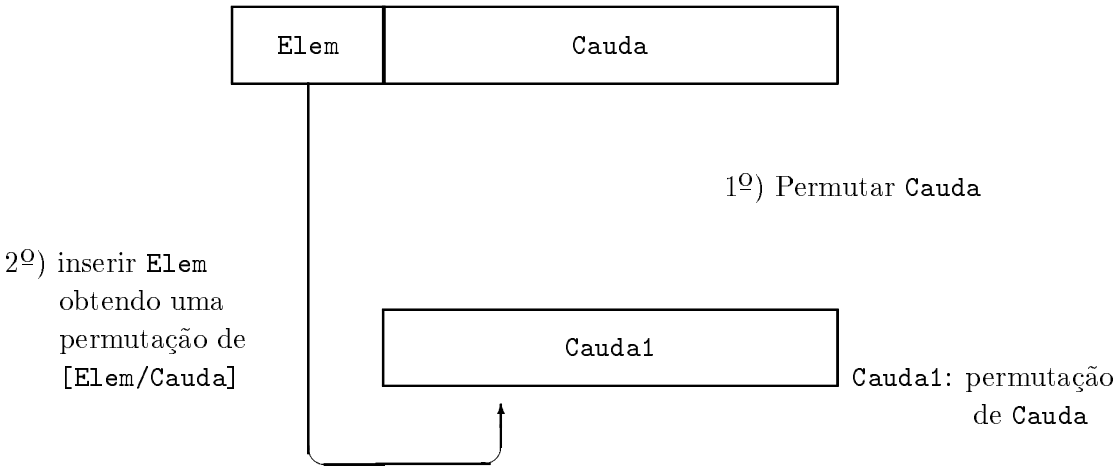
```
?- duplica([1,2,3],L).
L = [1,1,2,2,3,3] ->;
no
```

4.14 Permutar os Elementos de uma Lista

```
modo(+,?)

permutacao(<arg-1>,<arg-2>)
<arg-1> : lista
<arg-2> : permutacao dos elementos da lista
```

1. a permutação de uma lista vazia é a própria lista vazia.
2. se a lista a ser permutada é `[Elem|Cauda]`, primeiramente faz-se a permutação de `Cauda`; seja `Cauda1` a lista permutada. A seguir, insere-se o elemento `Elem` em qualquer posição da lista `Cauda1`, como mostra a figura a seguir.




```
permutacao([],[]).

permutacao([Elem|Cauda],Lista_perm) :-
                    permutacao(Cauda,Cauda1),
                    inserir_2(Elem,Cauda1,Lista_perm).
```

Exemplo 4.14.1 `modo(+,-)`

```
?- permutacao([casa,castelo,palacio],L).
L = [casa,castelo,palacio] ->;
L = [castelo,casa,palacio] ->;
L = [castelo,palacio,casa] ->;
L = [casa,palacio,castelo] ->;
L = [palacio,casa,castelo] ->;
L = [palacio,castelo,casa] ->;
no
```

4.15 Concatenar duas Listas

```
modo(?,?,+),(+,?,?)

concatenar(<arg-1>,<arg-2>,<arg-3>)
<arg-1> : primeira lista
<arg-2> : segunda lista
<arg-3> : lista concatenada
```

1. se o primeiro argumento for a lista vazia, então o segundo e o terceiro argumentos são a mesma lista.
2. se o primeiro argumento não for a lista vazia, então é do tipo `[Elem|Lista1]`; o resultado de sua concatenação com a lista `Lista2` é a lista `[Elem|Lista3]`, onde `Lista3` é a concatenação de `Lista1` e `Lista2`.

```
concatenar([],Lista,Lista).

concatenar([Elem|Lista1],Lista2,[Elem|Lista3]) :-
                    concatenar(Lista1,Lista2,Lista3).
```

Exemplo 4.15.1 `modo(+,+,-)`

```
?- concatenar([um,dois,tres],[quatro,cinco,seis],L).
L = [um,dois,tres,quatro,cinco,seis] ;
no
```

Exemplo 4.15.2 `modo(-,-,+)`
Com este modo de chamada, o programa é usado para decompor uma lista em duas listas.

```
?- concatenar(L1,L2,[ana,pedro,maria,jose]).
L1 = []
L2 = [ana,pedro,maria,jose] ->;
```

```
L1 = [ana]
L2 = [pedro,maria,jose] ->;
L1 = [ana,pedro]
L2 = [maria,jose] ->;
L1 = [ana,pedro,maria]
L2 = [jose] ->;
L1 = [ana,pedro,maria,jose]
L2 = [] ->;
no
```

Este exemplo mostra que é possível decompor a lista `[ana,pedro,maria,jose]` de cinco maneiras diferentes, e o programa encontra todas as soluções no retrocesso.

É possível também usar esse programa para procurar por um padrão em uma lista. Por exemplo, para encontrar os dias da semana que precedem e antecedem um dado dia, pode-se fazer a seguinte interrogação:

```
?- concatenar(Anterior,[quarta|Posterior],
              [segunda,terca,quarta,quinta,sexta,sabado,domingo]).
Anterior = [segunda,terca]
Posterior = [quinta,sexta,sabado,domingo] ->;
no
```

Para encontrar o dia anterior e posterior a um dado dia interroga-se com:

```
?- concatenar(X,[Dant,terca,Dpost|Y],
              [segunda,terca,quarta,quinta,sexta,sabado,domingo])
X = []
Dant = [segunda]
Dpost = [quarta]
Y = [quinta,sexta,sabado,domingo] -> ;
no

?- concatenar(X,[Dant,sexta,Dpost|Y],
              [segunda,terca,quarta,quinta,sexta,sabado,domingo]).
X = [segunda,terca,quarta]
Dant = [quinta]
Dpost = [sabado]
Y = [domingo] ->;
no
```

4.16 Sublistas de uma Lista

```
modo(?,+)

sublista(<arg-1>,<arg-2>)
<arg-1> : sublista
<arg-2> : lista
```

1. `Sub` é sublista da lista `Lista` se `Lista` pode ser decomposta em duas listas `L1` e `L2`, e `L2` por sua vez pode ser decomposta em duas outras listas, `Sub` e `L3`, como mostra a Figura 5 a seguir:

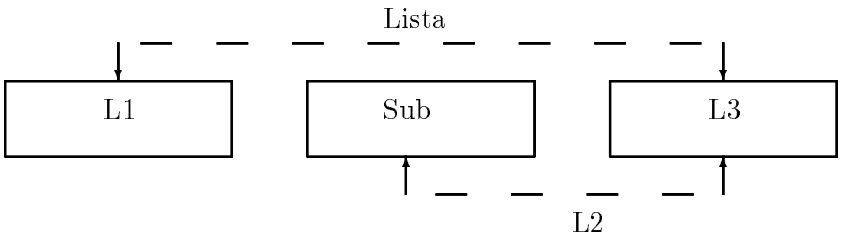


Figure 5: Sublista de uma Lista

```
sublista(Sub,Lista) :- concatenar(L1,L2,Lista),
                       concatenar(Sub,L3,L2).
```

Exemplo 4.16.1 `modo(-,+)`

```
?- sublista(X,[a,b,c,d]).
X = [];
X = [a] ->;
X = [a,b] ->;
X = [a,b,c] ->;
X = [a,b,c,d] ->;
X = [] ->;
X = [b] ->;
X = [b,c] ->;
X = [] ->;
X = [c] ->;
X = [] ->;
no
```

4.17 Inverter uma Lista

São apresentadas a seguir duas implementações para inverter uma dada lista; a segunda delas é a que deve ser efetivamente utilizada por ser a mais eficiente.

```
modo(?,+)

inverter_v1(<arg-1>,<arg-2>)
<arg-1> : lista
<arg-2> : lista invertida
```

1. a lista invertida da lista vazia é a lista vazia.
2. a lista invertida da lista `[Elem|Cauda]` é obtida invertendo-se `Cauda` e concatenando o resultado com a lista `[Elem]`.

```
inverter_v1([],[]).

inverter_v1([Elem|Cauda],Lista):-
                    inverter_v1(Cauda,Cauda1),
                    concatenar(Cauda1,[Elem],Lista).
```

Exemplo 4.17.1 `modo(+,-)`

```
?- inverter_v1([1,2,3,4,5,6,7,8,9,10],L).
L = [10,9,8,7,6,5,4,3,2,1] ->;
no
```

Pode-se observar que este programa, ainda que logicamente correto, é muito ineficiente, uma vez que para inverter a lista inverte a cauda remanescente de cada lista considerada para, só então, através de `concatenar/3`, construir a lista invertida. Segue uma versão mais eficiente deste programa.

```
inverter_v2(Lista,Lista_inv):- inv1(Lista,[],Lista_inv ).

inv1([],Lista,Lista).

inv1([Elem|Cauda],Lista_int,Lista_inv):-
                  inv1(Cauda,[Elem|Lista_int],Lista_inv).
```

Este processo de inverter a lista, construindo diretamente a lista invertida, torna o programa `inverter_v2/2` muito mais rápido do que o programa anterior.

O programa `inverter_v2/2` utiliza a técnica do uso de um acumulador como parâmetro, visando tornar o processo de inversão da lista mais eficiente. Esta técnica é frequentemente utilizada em programação Prolog. Quando um programa Prolog varre uma estrutura, o uso do acumulador objetiva representar *o resultado obtido até o momento.*

O programa `inverter_v2/2` faz uso deste artifício. Para tanto, um programa auxiliar `inv1/3` foi definido, com os mesmos parâmentros de `inverter_v2/2` (lista e sua lista invertida) e com o acumulador, inicializado com a lista vazia.

O programa `inv1/3` vai acumulando elemento a elemento da lista dada na lista acumulador. Quando a lista vazia for obtida, no acumulador está a lista invertida, situação tratada pela primeira cláusula de `inv1/3`, que instancia a lista invertida — terceiro argumento —, com o acumulador — segundo argumento.

A tabela abaixo mostra, aproximadamente, o tempo usado por ambos os programas para inverter listas de diversos tamanhos. As medidas foram realizadas sob o interpretador Prolog usando um micro computador PC/AT compatível[5].

No. Elementos	`inverter_v1`	`inverter_v2`
10	1.0t	0.2t
40	16.0t	0.7t
100	105.0t	1.6t

4.18 Maior Elemento de uma Lista Numérica

```
modo(+,?)

max(<arg-1>,<arg-2>)
<arg-1> : lista numerica
<arg-2> : maior elemento
```

1. o maior elemento de uma lista de um elemento, é o próprio elemento.
2. Seja a lista `[X,Y|Cauda]`; se `X` é maior ou igual que `Y`, então ache o maior elemento da lista `[X|Cauda]`.
3. caso contrário, por causa do corte em 2., `Y` é maior que `X`, então ache o maior elemento da lista `[Y|Cauda]`.

```
max([X],X).

max([X,Y|Cauda],Max):- X >= Y,
                       !,
                       max([X|Cauda],Max).

max([X,Y|Cauda],Max):- max([Y|Cauda],Max).
```

Exemplo 4.18.1 `modo(+,-)`

```
?- max([1,10,100,-1,20],M).
M = 100 ->;
no
```

[5] Ver programa no Apêndice

4.19 Dividir uma Lista Numérica em duas Sublistas que Contenham os Elementos Menores ou Iguais e Maiores que um Dado Elemento.

```
modo(+,+,?,?),(+,?,+,+)

dividir(<arg-1>,<arg-2>,<arg-3>,<arg-4>)
<arg-1> : numero
<arg-2> : lista
<arg-3> : lista com elementos menores ou iguais a numero
<arg-4> : lista com elementos maiores que numero
```

Dado um número M e uma lista `[H|T]`:

1. se a lista for a lista vazia, as sublistas serão também a lista vazia.
2. se a cabeça `H` da lista `[H|T]` for menor ou igual a `M`, então inserir `H` em `U1`, que contém os elementos menores ou iguais a `M` e dividir a lista `T`.
3. se a cabeça `H` da lista `[H|T]` for maior que `M`, então inserir `H` em `U2`, que contém os elementos maiores que `M` e dividir a lista `T`.

```
dividir(_,[],[],[]).

dividir(M,[H|T],[H|U1],U2) :- H =< M,
                              dividir(M,T,U1,U2).

dividir(M,[H|T],U1,[H|U2]):- H > M,
                             dividir(M,T,U1,U2).
```

Exemplo 4.19.1 `modo(+,+,-,-)`

```
?- dividir(5,[1,2,3,4,5,6,7,-1],L1,L2).
L1 = [1,2,3,4,5,-1]
L2 = [6,7] ->;
no
```

4.20 Verificar se um Termo é uma Lista

```
modo(+)

eh_lista(<arg-1>)
<arg-1> : lista
```

1. `X` é lista se `X` for a lista vazia.

2. `X` é lista se `X` não for uma variável e não for um átomo.

```
eh_lista([]).

eh_lista(X):- not var(X),
              not atomic(X).
```

onde `var/1` e `atomic/1` são predicados pré-definidos de Prolog. O predicado `var(X)` é bem sucedido se `X` for uma variável não instanciada; `atomic(X)` é bem sucedido se `X` for um tipo de dado atômico.

Exemplo 4.20.1 `modo(+)`

```
?- eh_lista([a,v,c,e,f]).
yes

?- eh_lista([]).
yes

?- eh_lista(1).
no
```

Exemplo 4.20.2 `modo(-)`

```
?- eh_lista(X).
no
```

4.21 Achatar uma Lista

```
modo(+,?)

achatar(<arg-1>,<arg-2>)
<arg-1> : lista
<arg-2> : lista achatada
```

1. achatar a lista vazia resulta na lista vazia.
2. para achatar uma lista que tem uma lista como cabeça, achata-se a cabeça, achata-se a cauda e concatena-se ambas para obter o resultado.
3. se a cabeça não for uma lista, então o resultado será a lista que tem a mesma cabeça e a cauda é a cauda achatada.

```
achatar([],[]).

achatar([Cab|Caud],ListaA):-
                    lista(Cab),
                    achatar(Cab,CabA),
                    achatar(Caud,CaudA),
                    concatenar(CabA,CaudA,ListaA).

achatar([Cab|Caud],[Cab|CaudA]) :- not lista(Cab),
                                   achatar(Caud,CaudA).
```

Exemplo 4.21.1 `modo(+,-)`

```
?- achatar([a,[b,[c,d]],[[e,f,g],h],i],L).
L = [a,b,c,d,e,f,g,h,i] ->;
no
```

4.22 Elementos de uma Lista que Possuem uma Dada Propriedade P

O modo de chamada do programa `map/3` depende da propriedade `P`. O seu primeiro argumento, que representa esta propriedade, deve estar sempre instanciado.

```
map(<arg-1>,<arg-2>,<arg-3>)
<arg-1> : propriedade
<arg-2> : lista
<arg-3> : lista dos elementos da lista que possuem
          a propriedade P
```

```
map(_,[],[]).

map(P,[A|B],[C|D]):- Q=.. [P,A,C],
                     call(Q),
                     map(P,B,D).
```

O predicado representado pelo símbolo `=..` chama-se *univ*. É uma meta que é bem sucedida se seu primeiro argumento é uma estrutura e seu segundo argumento é uma lista que tem como cabeça o funtor da estrutura e como cauda os argumentos da estrutura. Geralmente é usado para transformar uma lista em uma estrutura ou uma estrutura em uma lista. Por exemplo:

```
?- X =..[a,b,c].
X = a(b,c) ->;
no
```

```
?- f(g(c),b) =.. L.
L = [f,g(c),b] ->;
no
```

Considerando a cláusula

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

pode-se utilizar `map/3` da seguinte forma:

```
?- map(maior,[250,350,450],[100,200,300]).
yes
```

Ou, então, utilizando a cláusula:

```
desloc(X,Xdesloc):- Xdesloc is X + 2.
```

ele pode ser utilizado para criar uma nova lista com a propriedade definida por `desloc/3`.

```
?- map(desloc,[1,2,3,4,5],L).
L = [3,4,5,6,7] ->;
no
```

4.23 Imprimir uma Lista com Deslocamento

```
modo(+,?)

imprimir(<arg-1>,<arg-2>)
<arg-1> : lista
<arg-2> : deslocamento
```

```
imprimir([Elem|Cauda],I):- J is I  3, +
                           imprimir(Elem,J),
                           imprx(Cauda,J).

imprimir(Elem,I):- tab(I),
                   write(Elem),nl.

imprx([], _).
imprx([Elem|Cauda],I):- imprimir(Elem,I),
                        imprx(Cauda,I).
```

`tab(I)` é um predicado pré-definido de Prolog que escreve `I` brancos.

Exemplo 4.23.1 `modo(+,+)`

```
?- imprimir( [a,[b,[c,[d,[e],f],g],h],i],2).
a
   b
      c
         d
            e
         f
      g
   h
i
yes

?-imprimir([a,b,c,[d,e,f],g,h,i],2).
a
b
c
    d
    e
    f
g
h
i
yes
```

5 Operações em Conjuntos

Listas podem ser usadas para representar conjuntos, muito embora existam diferenças entre os dois conceitos já que a ordem dos elementos de um conjunto não é importante, enquanto que lista é um conjunto ordenado de elementos. Uma outra diferença é que um mesmo elemento pode aparecer várias vezes em uma lista, fato que não acontece em conjuntos.

Tendo em conta estas diferenças, é possível representar conjuntos usando a estrutura de lista. Alguns dos programas que foram definidos anteriormente são aplicáveis a operações em conjuntos. Entre eles:

- verificar se um elemento pertence a um conjunto que corresponde ao programa para verificar se um elemento pertence a uma lista.
- retirar um elemento de um conjunto que corresponde ao programa para retirar um elemento de uma lista, etc...

Na descrição dos argumentos dos programas que se seguem e que também manipulam conjuntos, os argumentos identificados por `conjunto`, são conjuntos representados por listas.

5.1 Verificar se a Intersecção entre Dois Conjuntos é não Vazia

```
modo(+,+)

nao_vazia(<arg-1>,<arg-2>)
<arg-1> : conjunto
<arg-2> : conjunto
```

1. a intersecção entre dois conjuntos é não vazia se existe algum elemento que pertença a ambos.

```
nao_vazia(L1,L2):- pertence(Elem,L1),
                   pertence(Elem,L2).
```

Exemplo 5.1.1 `modo(+,+)`

```
?- nao_vazia([a,b,c,d],[a,b]).
yes

?- nao_vazia([a,b,c],[d,e,f]).
no
```

5.2 Verificar se Dois Conjuntos são Disjuntos

Pode-se escrever este programa em função do programa anterior, com o mesmo modo de chamada, já que:

1. dois conjuntos são disjuntos se não têm elementos comuns.

```
conj_disjuntos(L1,L2):- not nao_vazia(L1,L2).
```

Exemplo 5.2.1 `modo(+,+)`

```
?-conj_disjuntos([a,b,c,d],[d,c,b,a]).
no
```

5.3 Verificar se Dois Conjuntos são Iguais

```
modo(+,+)

conj_iguais(<arg-1>,<arg-2>)
<arg-1> : conjunto
<arg-2> : conjunto
```

1. dois conjuntos vazios são iguais.

2. dois conjuntos são iguais se possuem os mesmos elementos.

```
conj_iguais([],[]).

conj_iguais([X|L1],L2):- remover(X,L2,L3),
                         conj_iguais(L1,L3).

remover(X,[X|Y],Y).

remover(X,[_|L1],[_|L2]):- remover(X,L1,L2).
```

Exemplo 5.3.1 `modo(+,+)`

```
?- conj_iguais([a,b,c],[a,b,d,e,f]).
no

?- conj_iguais([a,b,m,c,d],[b,a,m,d,c]).
yes
```

5.4 União de Dois Conjuntos

```
modo(+,+,?)

uniao_conj(<arg-1>,<arg-2>,<arg-3>)
<arg-1> : conjunto
<arg-2> : conjunto
<arg-3> : uniao dos conjuntos
```

1. a união de dois conjuntos `L1` e `L2` é o conjunto `U` que tem como elementos os elementos comuns a `L1` e `L2`.

```
uniao_conj(L1,L2,U):- concatenar(L1,L2,L3),
                      retirar_rep(L3,U).
```

Exemplo 5.4.1 `modo(+,+,-)`

```
?- uniao_conj([branco,vermelho],[amarelo,azul],L).
L = [branco,vermelho,amarelo,azul] ->;
no

?-uniao_conj([a,b,c], [a,b,c,d,e],L).
L = [a,b,c,d,e] ->;
no
```

5.5 Intersecção de Dois Conjuntos

```
modo(+,?,?)

intersec_conj(<arg-1>,<arg-2>,<arg-3>)
<arg-1> : conjunto
<arg-2> : conjunto
<arg-3> : interseccao dos conjuntos
```

1. a intersecção do conjunto vazio com um conjunto qualquer é o conjunto vazio.
2. a intersecção do conjunto representado pela lista `[Cab|Cauda]` com o representado pela lista `L` é o conjunto representado pela lista `[Cab|U]` se `Cab` pertencer a `L` e `U` for a intersecção de `Cauda` com `L`.
3. a intersecção do conjunto representado pela lista `[_|Cauda]` com o conjunto representado pela lista `L` é a intersecção de `Cauda` com `L`. Observe que o corte usado na cláusula 2 do programa dado a seguir, implica que a cláusula 3 somente será executada se o primeiro elemento da primeira lista não pertencer a `L`.

```
intersec_conj([],_,[]).

intersec_conj([Cab|Cauda],L,[Cab|U]):-
                            pertence(Cab,L),
                            !,
                            intersec_conj(Cauda,L,U).

intersec_conj([_|Cauda],L,U):- intersec_conj(Cauda,L,U).
```

Exemplo 5.5.1 `modo(+,+,_)`

```
?- intersec_conj([a,b,c],[a,b,d,e,f],L).
L = [a,b] ->;
no
```

Exemplo 5.5.2 `modo(+,-,-)`

```
?- intersec_conj([l,m,n],X,Y).
X = [l,m,n|_]
Y = [l,m,n] ->;
no
```

5.6 Diferença Simétrica entre Dois Conjuntos

```
modo(+,+,?)

conj_diferenca(<arg-1>,<arg-2>,<arg-3>)
<arg-1> : conjunto
<arg-2> : conjunto
<arg-3> : conjunto diferenca
```

1. a diferença simétrica entre dois conjuntos representados pelas listas `L1` e `L2` é a união dos conjuntos de elementos não comuns a ambos os conjuntos.

```
conj_diferenca(L1,L2,D):-
      findall(X,(pertence(X,L1),not pertence(X,L2)),D1),
      findall(Y,(pertence(Y,L2),not pertence(Y,L1)),D2),
      uniao_conj(D1,D2,D).
```

O predicado pré-definido `findall/3` é muito poderoso. Ele coleciona em uma lista os resultados de uma consulta. Por exemplo,

```
?- findall(Vars,M,L).
```

gera a lista `L`, das variáveis dadas em `Vars`, que verificam a meta `M`. As variáveis que aparecem na meta `M` e que não são mencionadas em `Vars` são consideradas como existencialmente quantificadas. A lista `L` resultante não está em ordem e pode conter elementos repetidos.

Exemplo 5.6.1 `modo(+,+,-)`

```
?- conj_diferenca([a,b,c],[a,b,d,e,f],L).
L = [c,d,e,f] ->;
no
```

Há outros dois predicados que também colecionam em uma lista os resultados de uma consulta. São eles:

`bagof(Vars,M,L)`
e
`setof(Vars,M,L)`

O predicado Prolog pré-definido `bagof(+Termo,+Meta,-Bag)` coleciona em uma lista — `Bag` — todas as instâncias de um termo — `Termo` — para as quais uma específica meta `Meta` foi satisfeita. `Termo` deve conter apenas variáveis que comparecem em `Meta`. Já `Meta` é um predicado que contém variáveis que comparecem também em `Termo`. Outras variáveis que comparecem em `Meta` ou são livres ou ligadas.

Uma variável é livre se:

1. está não instanciada, quando `bagof/3` é invocado;
2. não aparece no termo `Termo`;
3. não é existencialmente quantificada.

O quantificador existencial `^` é usado em variáveis não instanciadas para fazer com que `bagof/3` as trate como se fossem ligadas. Em outras palavras, em `X^P`, `X` é existencialmente ligada a `P`; em `X^Y^P`, tanto `X` quanto `Y` são existencialmente ligadas a `P`.

A lista que `bagof/3` constrói corresponde a cada instanciação de todas as variáveis livres de `Meta`. Para cada instanciação das variáveis livres, `bagof/3` retorna todas as instâncias de `Termo` numa lista não ordenada, que pode também conter duplicações. Por exemplo, seja uma base de dados contendo os seguintes fatos:

```
gosta(maria,cafe).
gosta(jose,cha).
gosta(ana,guarana).
gosta(pedro,vinho).
gosta(jose,coca).
gosta(maria,vinho).
```

`bagof/3` pode ser invocado para determinar quem gosta de qual bebida da seguinte forma:

```
?-bagof(X,gosta(X,Y),L).
X = _00E4
Y = cafe
L = [maria] ->;

X = _00E4
Y = cha
L = [jose] ->;

X = _00E4
Y = coca
L = [jose] ->;

X = _00E4
Y = guarana
L = [ana] ->;

X = _00E4
Y = vinho
L = [pedro,maria] ->;
no
```

No exemplo que se segue, o quantificador existencial estabelece que para qualquer instância de `Y` em `Meta`, `bagof/3` retornará todas as instâncias de `X`. Com a base de dados anterior, isto significa que serão colecionadas todas as instâncias de `X` (bebidas) que são gostadas por qualquer `Y` (pessoas), como segue:

```
?- bagof(X,Y^gosta(Y,X),L).
X = _00E4
Y = _00F4
L = [cafe,cha,guarana,vinho,coca,vinho]
yes
```

O predicado `setof(+Termo,+Meta,-Set)` comporta-se analogamente à `bagof/3`, com uma única diferença: a lista que retorna está classificada e não contém elementos duplicados. por exemplo, para a base de dados definida anteriormente tem-se:

```
?- setof(X,Y^gosta(Y,X),L).
X = _00E4
Y = _00F4
L = [cafe,cha,coca,guarana,vinho]
yes
```

Resumindo, o predicado `findall` é semelhante ao `bagof`, exceto que qualquer variável livre é assumida ser existencialmente quantificada. Isso significa que o `findall` procura na base por todas as ocorrências de `Termo` que satisfazem `Meta` e retorna as instâncias em uma lista não ordenada que eventualmente contém elementos duplicados. Deve ser observado que `findall` é mais eficiente que `bagof` e que `setof`.

5.7 Subconjuntos de um Dado Conjunto

```
modo(+,?)

subconjunto(<arg-1>,<arg-2>)
<arg-1> : conjunto
<arg-2> : subconjunto
```

1. se o conjunto é o conjunto vazio, o subconjunto é o próprio, caso contrário,
2. gerar os subconjuntos, retendo nos subconjuntos, o primeiro elemento do conjunto, ou
3. removendo, nos subconjuntos, o primeiro elemento do conjunto

```
subconjunto([],[]).

subconjunto([Prim|Resto],[Prim|Subconj]):-
                    subconjunto(Resto,Subconj).

subconjunto([Prim|Resto],Subconj) :-
                    subconjunto(Resto,Subconj).
```

Exemplo 5.7.1 `modo(+,-)`

```
?- subconjunto([a,b,c],C).
C = [a,b,c] ->;
C = [a,b] ->;
C = [a,c] ->;
C = [a] ->;
C = [b,c] ->;
C = [b] ->;
C = [c] ->;
C = [] ->;
no
```

6 Classificação

O problema de classificação — ou ordenação — consiste em rearranjar objetos em uma ordem especificada. Existem muitos algoritmos de classificação e a maioria deles depende da estrutura de dados utilizada. Será tratado aqui o processo de classificação quando os elementos a serem ordenados são elementos de uma lista residente na memória principal. Este processo é chamado de classificação interna.

Uma lista pode ser classificada se existe uma relação de ordem entre seus elementos. No que se segue, a relação de ordem entre os elementos da lista será dada pelo predicado:

`maior(X,Y)`

que é verdadeiro quando `X` for maior que `Y`.

Se os elementos da listas são números, o predicado `maior/2` pode ser definido como:

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

Se os elementos da lista forem estruturas, como por exemplo `nome('Ana','Silva')`, onde o primeiro e segundo argumentos denotam respectivamente o nome e o sobrenome de uma pessoa, então pode-se estabelecer como chave primária da relação de ordem — alfabética neste caso — o sobrenome. Em casos de empate, pode-se considerar a relação de ordem entre os primeiros nomes. Este novo predicado `maior/2` pode então ser definido como:

```
maior(nome(N1,S1),nome(N2,S2)):- S1 @> S2,!.

maior(nome(N1,S1),nome(N2,S2)):- S1 == S2,
                                 N1 @> N2.
```

Após ter definido esta relação entre os elementos de uma lista, é fácil verificar se os elementos da lista estão ordenados.

6.1 Verificar se os Elementos de uma Lista estão Ordenados

Uma lista está ordenada se:

1. possui somente um elemento.
2. a relação de ordem estabelecida se verifica entre todos os elementos consecutivos da lista.

```
modo(+)

ordenada(<arg-1>)
<arg-1> : lista
```

```
ordenada([X]).

ordenada([X,Y|Z]):- maior(Y,X),
                    ordenada([Y|Z]).
```

6.2 Ordenar os Elementos de uma Lista

O programa anterior apenas verifica se os elementos de uma lista estão ordenados. Para ordenar os elementos da lista deve-se definir uma relação

`ordenar(Lista,ListaOrdenada)`

onde `ListaOrdenada` deve possuir todos os elementos de `Lista` e, além disso, deve ser verdade que

`ordenada(ListaOrdenada)`

Há muitas formas de se definir a relação `ordenar(Lista,ListaOrdenada)`, sendo que cada uma delas corresponde a um algoritmo de classificação diferente. Uma possível implementação seria permutar os elementos da lista e verificar se esta permutação está ordenada, ou seja:

```
ordenar(Lista,ListaOrdenada):-
                    permutacao(Lista,ListaOrdenda),
                    ordenada(ListaOrdenada),
                    !.
```

onde o predicado `permutacao/2` é o definido na seção 4.14 pg.23.

Se a lista de entrada `Lista` possui n elementos diferentes, `permutacao/2` possui $n!$ soluções diferentes, e somente uma delas satifaz o programa `ordenar`. Pode-se ver que no melhor dos casos, (quando a lista de entrada está ordenada), o programa `permutacao` é executado somente uma vez. Entretanto, dependendo da lista de entrada, `permutacao/2` pode ser executado $n!$ no pior caso. Portanto, se for considerada como medida de eficiência do algoritmo o número de vezes que é executado o corpo da regra, temos que, no melhor caso é 1 e no pior caso $n!$.

Portanto, com esta medida de eficiência, o melhor caso deste algoritmo é $O(1)$ e o pior caso é $O(n!)$, onde O representa a ordem de complexidade.

Pelas considerações acima, é possível observar que tanto o pior quanto o melhor caso acontecem somente uma vez. É preciso, pois, analisar o comportamento médio deste algoritmo.

Se as $n!$ permutações da lista de entrada são igualmente possíveis, então o número médio de vezes que o corpo do programa é executado é dado por:

$$\sum_{i=1}^{n} i\ P_i$$

onde i é o número de vezes que o corpo do programa `ordenar` é executado e $P_i = \frac{1}{n!}$, isto é:

$$\sum_{i=1}^{n} i\ P_i = (1 + 2 + \ldots + (n-1)! + n!)\frac{1}{n!}$$

$$= \frac{n!(n!+1)}{2}\ \frac{1}{n!}$$

$$= \frac{n!}{2} + \frac{1}{2}$$

que é também $O(n!)$. Tem-se portanto um algoritmo exponencial. Na verdade, este algoritmo pode ser considerado como o pior algoritmo de classificação.

Classificação é um problema muito bem estudado na área de computação. Há na literatura inúmeros algoritmos projetados para realizar esta tarefa, e geralmente a publicação de um algoritmo desse tipo vem acompanhada de sua análise de complexidade. Um exame detalhado destes métodos foge aos objetivos deste trabalho e pode ser encontrado na maioria dos livros referentes a algoritmos e estruturas de dados.

É importante saber que tomando como medida de complexidade o número de comparações realizadas para ordenar n ítens, há algoritmos de classificação que requerem $O(n \log n)$, isto é, ordem $(n \log n)$ comparações para realizar a tarefa. É possível mostrar que realizando $O(n \log n)$ comparações pode-se ordenar qualquer permutação de n ítens, ou seja, há algoritmos que classificam n ítens realizando $O(n \log n)$ comparações no pior caso.

Um destes algoritmos chama-se *heapsort*, que, neste sentido, é um algoritmo ótimo, mas não é apropriado quando a estrutura de dados é a lista, razão pela qual não será tratado aqui.

Seguem-se as implementações em Prolog de alguns algoritmos de classificação interna, onde `Lista` denota a lista a ser ordenada e `ListaOrdenada` é uma lista que possui os mesmos elementos de `Lista` classificados em ordem decrescente. O modo de chamada para estes algoritmos é:

```
modo(+,?)

<arg-1>: lista a ser ordenada
<arg-2>: lista ordenada
```

6.3 Bubblesort

Este algoritmo baseia-se em:

1. encontrar dois elementos adjacentes `X` e `Y` em `Lista` tal que `maior(X,Y)` seja verdadeiro; trocar estes elementos em `Lista` obtendo `Lista1` e ordenar `Lista1`
2. se não há dois elementos adjacentes `X` e `Y` em `Lista`. tal que `maior(X,Y)` seja verdadeiro, então `Lista` está ordenada.

```
bubblesort_v1(Lista,ListaOrd):-
                    trocar(Lista,Lista1),!,
                    bubblesort_v1(Lista1,ListaOrd).

bubblesort_v1(ListaOrd,ListaOrd).

trocar([X,Y|Cau],[Y,X|Cau]):- maior(X,Y).

trocar([Z|Cau],[Z|Cau1]):- trocar(Cau,Cau1).
```

Exemplo 6.3.1 `modo(+,-)`

```
?- bubblesort_v1([9,8,7,6,5,4,3,2,1],L).
L = [1,2,3,4,5,6,7,8,9] ->;
no
```

Este algoritmo pode também ser implementado da seguinte forma:

```
bubblesort_v2(L1,L2):-
        concatenar(U,[A,B|V],L1),
        maior(A,B),
        concatenar(U,[B,A|V],M),
        bubblesort_v2(M,L2),!.

bubblesort_v2(L1,L1).
```

onde o predicado `concatenar/3` está definido na seção 4.15 pg.24.

6.4 Insertionsort

Este algoritmo baseia-se em:

1. ordenar a cauda da lista.
2. inserir a cabeça da lista na cauda ordenada, em uma posição tal que se obtenha a lista ordenada.

```
insertionsort([],[]).

insertionsort([Cab|Cauda],ListaOrd):-
                    insertionsort(Cauda,CaudaOrd),
                    inserir(Cab,CaudaOrd,ListaOrd).

inserir(X,[Y|CauOrd1],[Y|CauOrd2]):-
                    maior(X,Y),
                    !,
                    inserir(X,CauOrd1,CauOrd2).

inserir(X,CauOrd,[X|CauOrd]).
```

Exemplo 6.4.1 `modo(+,-)`

```
?- insertionsort([9,8,7,6,5,4,3,2,1],L).
L = [1,2,3,4,5,6,7,8,9] ->;
no
```

6.5 Quicksort

Os procedimentos de classificação *bubblesort* e *insertionsort* são simples mas ineficientes, já que o caso médio de ambos os algoritmos é $O(n^2)$.

O algoritmo *quicksort*, apresentado a seguir, foi desenvolvido por C.A.R. Hoare e tem complexidade $O(n \log n)$ se a lista for dividida em duas sublistas de aproximadamente o mesmo comprimento [Hoare 62]. Entretanto, se em cada passo a lista for dividida em duas listas tal que uma é muito maior que a outra, temos o pior caso de quicksort que é $O(n^2)$.

Todas as variações deste algoritmo concentram-se na eleição do elemento que é responsável pela divisão da lista, de maneira que a ocorrência do pior caso seja minimizada. Os passos deste algoritmo para classificar uma lista são:

```
quicksort([],[]).

quicksort([X|Cau],ListaOrd):-
                    particionar(X,Cau,Men,Mai),
                    quicksort(Men,MenOrd),
                    quicksort(Mai,MaiOrd),
                    concatenar(MenOrd,[X|MaiOrd],ListaOrd).

particionar(_,[],[],[]).

particionar(X,[Y|Cau],[Y|Men],Mai):-
                    maior(X,Y),
                    !,
                    particionar(X,Cau,Men,Mai).

particionar(X,[Y|Cau],Men,[Y|Mai]):-
                    particionar(X,Cau,Men,Mai).
```

Exemplo 6.5.1 `modo(+,-)`

```
?- quicksort([9,8,7,6,5,4,3,2,1],L).
L = [1,2,3,4,5,6,7,8,9] ->;
no
```

Observação: Em geral os interpretadores/compiladores Prolog disponíveis cormecialmente fornecem alguns predicados pré-definidos para classificar listas, de acordo com uma ordem padrão. No caso específico do Arity Prolog há dois pré-definidos descritos a seguir:

- `sort(+L1,-L2)`, que classifica a lista `L1` de acordo com a ordem padrão, faz a intercalação dos elementos duplicados e retorna a lista ordenada em `L2`. Por exemplo:

```
?- sort([q,a,f,a,q,f],L).
 L = [a,f,q]
 yes
```

- `keysort(+L1,-L2)` que classifica termos da forma Chave-Valor, não faz a intercalação de elementos duplicados e retorna a lista ordenada em `L2`. Por exemplo:

```
?- keysort([a-3,b-2,a-1,c-5,a-1],L).
L = [a-3,a-1,a-1,b-2,c-5]
```

É importante notar que existem mais algoritmos de classificação do que aqueles apresentados nesta Nota. Além disso, dado um algoritmo — por exemplo *bubblesort* —, há diversas maneiras de implementá-lo. Devido a esta variedade de algoritmos/implementações, é interessante que o leitor verifique os tempos de execução dessas diferentes implementações em função do número de elementos a serem classificados. Esses tempos devem ser comparados em igualdade de condições, i.e., versões interpretadas ou versões compiladas[6].

O programa apresentado no Apêndice pode ser usado para avaliar o desempenho das diversas implementações, quando da classificação de um mesmo conjunto de dados.

7 Problemas que Utilizam Listas e suas Soluções

Nesta seção serão apresentados diversos problemas que podem ser implementados usando a estrutura de lista.

7.1 Relação entre Duas Listas

Descrição: Dadas duas listas a e b, escrever um programa Prolog que implementa a relação `admissível` entre elas, que é bem sucedida quando

$$b_i = 2\ a_i \text{ e } a_{i+1} = 3\ b_i$$

sendo que a_i e b_i são os elementos da i-ésima posição das listas a e b respectivamente. Seguem duas implementações para a solução deste problema.

Versão 1

```
admissivel_v1(X,Y):- dobro(X,Y),
                     triplo(X,Y).

dobro([],[]).
dobro([X|Y],[M|N]):- M is 2*X,
```

[6]Predicados pré-definidos estão compilados.

```
                              dobro(Y,N).

triplo([],[]).
triplo([X],[Y]).
triplo([X,Y|Z],[M|N]):- Y is 3*M,
                         triplo([Y|Z],N).
```

Observação: A terceira cláusula de `triplo/2` necessita de discriminar o $i+1$-ésimo elemento da primeira lista, uma vez que este elemento deve ser o triplo do i-ésimo elemento da segunda lista para que a relação se verifique.

A segunda cláusula de `triplo/2` trata a situação em que as duas listas têm um elemento apenas, quando a segunda relação: $a_{i+1} = 3b_i$ não pode ser verificada. Para essa situação, fica valendo apenas que o i-ésimo elemento da lista b deve ser o dobro do i-ésimo da lista a.

Versão 2

```
admissivel_v2([],[]).
admissivel_v2([X,Y|Z],[M|N]):- M is 2*X,
                                Y is 3*M,
                                admissivel_v2([Y|Z],N).
admissivel_v2([X],[M]):- M is 2*X.
```

Um exemplo de execução:

```
?- admissivel_v1([1,6,36],[A,B,C]).
A = 2
B = 12
C = 72 ->;
no
```

É interessante notar que a interrogação:

```
?- admissivel_v1([A,B,C],[2,12,72])
```

provocará erro, uma vez que o programa procurará inicialmente verificar `2 is 2 * A` — que falha, uma vez que a variável `A` não está instanciada.

7.2 Teste de Verificação de Palavras que são Palíndromos

Descrição: Escrever um programa Prolog que verifique, a cada palavra dada, se a palavra é um palíndromo ou não. Palíndromos são palavras que têm a mesma leitura, quando lidas de trás para frente. São exemplos: *ovo*, *ana*, *arara*.

A interface do programa com o usuário é feita pelo procedimento `interface/1` — que lê uma palavra (átomo), através do predicado pré-definido Prolog `read/1` — e invoca

`testa_palindromo/1`, que será bem sucedido se a palavra lida for um palíndromo. O procedimento é repetido até o usuário digitar `pare`.

O procedimento `testa_palindromo/1` invoca o predicado pré-definido Prolog `name/2`, explicado no final desta seção, que transforma o seu primeiro argumento em uma lista — seu segundo argumento — e então invoca `palindromo/1`, que efetivamente verifica se a palavra é um palíndromo.

Se `palindromo/1` falhar, a segunda cláusula de `testa_palindromo/1` é invocada, e informa que a palavra não é um palíndromo.

O procedimento `palindromo/1` apenas verifica se a lista inversa da lista correspondente à palavra fornecida é a própria lista, o que significa que a palavra lida é um palíndromo.

```
interface:- write(' entre com a palavra: '),
            read(X),
            (X = pare;
            testa_palindromo(X),
            interface).

testa_palindromo(X):-
            name(X,Nx),
            palindromo(Nx), !,
            write(X),
            write(' e um palindromo'),
            nl.

testa_palindromo(X):-
            write(X),
            write(' nao e palindromo'),
            nl.

palindromo(X):- inverter_v2(X,X).
```

onde `inverter_v2/2` está definido na seção 4.17, pg. 26.

O predicado `name/2` é pré-definido do Prolog e relaciona átomos às suas codificações ASCII. Assim sendo, `name(A,L)` é verdade se `L` for a lista de códigos ASCII dos caracteres em `A`, ou vice-versa. Por exemplo,

```
 ?- name(ab27h,L).
 L = [97,98,50,55,104]
 yes

?-name(L,[116,117,100,111,95,98,101,109]).
 L = tudo_bem
 yes
```

Tipicamente o predicado `name` é usado para

1. dado um átomo, quebrá-lo em caracteres,
2. dado uma lista de caracteres, combiná-los em um átomo.

7.3 Classificação de Listas usando Intercalação

Descrição: Esta classificação é baseada na idéia de ir dividindo a lista, ordenando as listas menores obtidas e intercalando estas listas menores ordenadas, até obter a lista original ordenada [Bratko 90]. Ou seja, para classificar uma lista L:

1. dividir `L` em duas listas, `L1` e `L2`, de aproximadamente o mesmo tamanho;
2. ordenar `L1` e `L2`, obtendo `O1` e `O2`;
3. fazer a intercalação de `O1` e `O2`, obtendo a lista `L` ordenada

```
classifica([],[]).

classifica([X],[X]):-!.

classifica(L,O):- divide_lista(L,L1,L2),
                  classifica(L1,O1),
                  classifica(L2,O2),
                  intercalar(O1,O2,O).

divide_lista([X],[X],[]):-!.
divide_lista([X,Y],[X],[Y]):-!.
divide_lista(L,L1,L2):- no_elementos(L,M),
                        N is M // 2,
                        pega(L,N,L1,L2).

pega(L,0,[],L).
pega([X|Y],N,[X|L1],Y1):- M is N-1,
                          pega(Y,M,L1,Y1).
```

onde `no_elementos/2` está definido na seção 4.6, pg. 16 e `//` é um predicado pré-definido usado divisão inteira, dando como resultado um número inteiro.

É necessário agora definir `intercalar/3`, que a partir de duas listas ordenadas, constrói, intercalando os elementos das duas listas dadas, uma terceira lista ordenada. Uma possível implementação é:

```
intercalar(Xs,[],Xs) :- !.
```

```
intercalar([],Ys,Ys).

intercalar([X|Xs],[Y|Ys],[X|Zs]):-
        X < Y, !,
        intercalar(Xs,[Y|Ys],Zs).

intercalar([X|Xs],[Y|Ys],[X,Y|Zs]):-
        X = Y, !,
        intercalar(Xs,Ys,Zs).

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

Seguem exemplos de execução de `intercalar/3` e `classifica/3`:

```
?- intercalar([1,4,7,20],[0,4,8,15],L).
L = [0,1,4,4,7,8,15,20] ->;
no

?- classifica([2,5,3,9,19,0,7,9],L).
L = [0,2,3,5,7,9,9,19] ->;
no
```

7.4 Geração de Nomes Híbridos

Descrição: Gerar possíveis nomes híbridos entre nomes de dois animais, determinados da seguinte forma:

- quando duas ou mais letras no final do nome de um animal coincidem com as letras do início do nome de outro animal, um novo nome de animal é gerado — concatenação dos nomes fornecidos, sendo que a parte que coincide comparece apenas uma vez. Por exemplo: híbrido de cobra com rato é *cobrato*, de barata com tatu é *baratatu* e de galinha com nhambu é *galinhambu*
- quando não existe tal coincidência, o predicado não deve ser satisfeito

```
hibrido(X,Y,L):- name(X,LX),
                 name(Y,LY),
                 concatenar(PpioX,FimX,LX),
                 FimX \= [],
                 concatenar(FimX,FimY,LY),
                 FimY \= [],
                 concatenar(LX,FimY,LL),
                 name(L,LL).
```

onde `concatenar/3` está definido na seção 4.15, pg. 24.

Segue um exemplo de execução:

```
? - hibrido(minhoca,cavalo,L).
L = minhocavalo ->;
no
```

7.5 Determinar as Combinações dos Elementos de um Conjunto

O problema consiste na seleção de um subconjunto de um dado conjunto de elementos, onde a ordem em que esses elementos comparecem é irrelevante. Sob esse ponto de vista, por exemplo, as seguintes listas que representam subconjuntos[7] são equivalentes:

```
[1,2,3],[1,3,2],[2,1,3],[2,3,1],[3,1,2],[3,2,1]
```

Essas listas, consideradas como seis permutações diferentes, são tratadas como sendo apenas uma combinação. A fórmula que dá o número total de maneiras de selecionar r objetos, de um total de n objetos é:

$$C_{n,r} = \frac{n!}{r!(n-r)!} \tag{2}$$

Para um conjunto com n elementos $a, b, c, \ldots$, as $C_{n,r}$ combinações diferentes de r elementos que podem ser extraídas desse conjunto caem em duas categorias: aquelas que contém o elemento a e aquelas que não contém este elemento. Em todas as combinações que contém a, os outros $(r-1)$ elementos devem ter sido selecionados dos $(n-1)$ elementos restantes. O número de combinações que contém a deve então ser $C_{n-1,r-1}$.

No caso das combinações que não contém a, todos os r elementos devem ter sido selecionados dos $(n-1)$ elementos diferentes de a, ou seja, devem existir $C_{n-1,r}$ de tais casos. Pode-se pois escrever que:

$$C_{n,r} = C_{n-1,r-1} + C_{n-1,r} \; \forall n \geq r$$

Em qualquer processo recursivo, a recursão deve conduzir a algum caso limite. Na primeira parcela da soma acima, tanto n quanto r foram reduzidos a $(n-1)$ e $(r-1)$ respectivamente. Obviamente, durante o processo recursivo isso irá levar à uma situação onde o número de seleções a ser feita é 0. O número de combinações de 0 elementos selecionados é 1, uma vez que existe apenas uma maneira de deixar todos os objetos fora. Um caso limite é pois

[7] Não há elementos repetidos.

$$C_{n,0} = 1 \; \forall n \geq 0.$$

Na segunda parcela da soma, n é reduzido. Essa recursão deve levar à uma situação onde o número de elementos a ser selecionado é o mesmo que o número de objetos disponíveis — a única forma de fazer isso é selecionar todos os elementos. Isso resulta no segundo caso limite:

$$C_{r,r} = 1$$

As cláusulas Prolog que expressam as condições limite são:

```
nro_combinacoes(N,0,1):-!.
nro_combinacoes(N,N,1):-!.
```

O caso recursivo é tratado pela cláusula:

```
nro_combinacoes(N,R,Nro_Comb):-
        N > R,
        N1 is N - 1,
        R1 is R - 1,
        nro_combinacoes(N1,R1,Parcela_1),
        nro_combinacoes(N1,R,Parcela_2),
        Nro_Comb is Parcela_1 + Parcela_2.
```

Um exemplo de execução desse programa é:

```
?- nro_combinacoes(10,4,M).
M = 210
yes
```

É importante notar que o programa `nro_combinacoes/3` é muito ineficiente, uma vez que cálculos intermediários são redundantemente realizados durante o processo. Para a solução do problema em questão, a abordagem mais indicada é a utilização direta da equação (2).

O programa `nro_combinacao/3` dá apenas o número de combinações; as combinações podem ser obtidas pelo programa:

```
combinacoes(0,Conjunto,[]).
combinacoes(R,Conjunto,[Elem|T]):-
          R1 is R - 1,
          concatenar(Prim,[Elem|Resto],Conjunto),
          combinacoes(R1,Resto,T).
```

Segue um exemplo de execução desse programa.

```
?- combinacoes(5,[1,2,3,4,5,6],L).
L = [1,2,3,4,5] ->;

L = [1,2,3,4,6] ->;

L = [1,2,3,5,6] ->;

L = [1,2,4,5,6] ->;

L = [1,3,4,5,6] ->;

L = [2,3,4,5,6] ->;
no
```

7.6 Problema das Quatro Cores

Descrição: O problema das 4 cores se resume em colorir um mapa plano de tal forma que duas regiões vizinhas não tenham a mesma cor. Este problema foi uma questão em aberto por muito tempo. Em 1976 foi mostrado que 4 cores são suficientes para colorir qualquer mapa plano obedecendo a restrição de duas regiões vizinhas não terem a mesma cor.

Primeiramente será tratada a implementação proposta em [Sterling 86]. O programa implementará o seguinte algoritmo iterativo não-determinístico:

Para cada região do mapa:

- escolher uma cor do conjunto de cores disponíveis
- a partir do restante das cores, escolher as cores das regiões vizinhas

Será usado o funtor `mapa` para combinar numa estrutura o nome do mapa e o conjunto de regiões que fazem parte dele. Por exemplo, pode-se ter:

```
mapa(america_sul,Lista_Paises_Sul_Americanos)
mapa(sao_paulo,Lista_Vizinhos_Sao_Paulo)
mapa(europa_ocidental,Lista_de_Vizinhos)
```

Cada região, por sua vez, será representada por um funtor `regiao`, que combina numa única estrutura, o nome da região, sua cor e uma lista das cores de suas regiões vizinhas. Por exemplo:

```
regiao(espanha,azul,[Cor_Portugal,Cor_Franca])
```

Seguem as figuras de dois mapas, cujos nomes são `teste1` e `teste2`, e suas respectivas representações em Prolog. Nas figuras dos mapas, para identificar cada região, será usada a notação:

`x(Y)` — onde `x` indica o nome da região e `Y` a variável que contém a sua cor

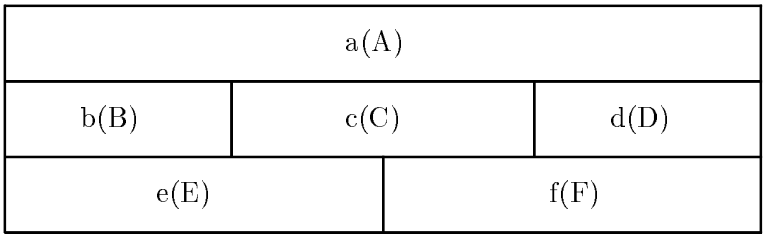


Figure 6: Problema das Quatro Cores: `teste1`

```
mapa(teste1,
     [regiao(a,A,[B,C,D]),
      regiao(b,B,[A,C,E]),
      regiao(c,C,[A,B,D,E,F]),
      regiao(d,D,[A,C,F]),
      regiao(e,E,[B,C,F]),
      regiao(f,F,[C,D,E])]).
```

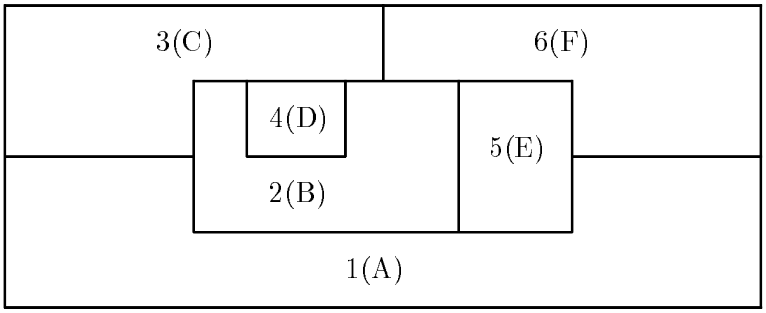


Figure 7: Problema das Quatro Cores: `teste2`

```
mapa(teste2,
     regiao(1,A,[B,C,E,F]),
     regiao(2,B,[A,C,D,E,F]),
     regiao(3,C,[A,B,D,F]),
     regiao(4,D,[B,C]),
     regiao(5,E,[A,B,F]),
     regiao(6,F,[A,B,C,E])]).
```

Nesta implementação, a relação de mais alto nível é `colore_mapa(Mapa,Cores)`, onde `Mapa` é uma estrutura de mapa previamente instanciada e `Cores` uma lista de cores,

com a qual se pretende colorir as várias regiões do mapa instanciado. Essa relação é expressa em Prolog como:

```
colore_mapa(mapa(_,[Regiao|Regioes]),Cores):-
                        colore_regiao(Regiao,Cores),
                        colore_mapa(mapa(_,Regioes),Cores).

colore_mapa(mapa(_,[]),Cores).
```

Como o procedimento `colore_mapa/2` é definido recursivamente sobre a lista de regiões, a condição de término será satisfeita quando não existirem mais regiões a serem coloridas. Isso é expresso em Prolog como:

```
colore_mapa(mapa(_,[]),Cores).
```

A parte central da implementação do algoritmo está na definição de `colore_regiao/2`:

```
colore_regiao(regiao(Nome_reg,Cor_reg,Cor_vizinhos),Cores):-
              colore(Cor_reg,Cores,Cor_restante),
              colore_vizinhos(Cor_vizinhos,Cor_restante).
```

Os predicados `colore/3` e `colore_vizinhos/2` atuam como geradores e também como verificadores de cores — a forma de atuação vai depender de seus argumentos estarem ou não instanciados. Se os argumentos não estiverem instanciados, eles instanciarão possíveis cores às variáveis que representam as cores da região. Na eventualidade dessas variáveis já estarem instanciadas, os predicados em questão atuam como verificadores — verificam se os valores instanciados satisfazem às restrições do problema. Os predicados são expressos em Prolog como:

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

colore_vizinhos([X|Xs],Ys):- pertence(X,Ys),
                             colore_vizinhos(Xs,Ys).
colore_vizinhos([ ],Ys).
```

onde `pertence/2` está definido na seção 4.1, pg. 12. Segue um exemplo de execução desse programa com os mapas de nome `teste1` e `teste2`, onde apenas algumas soluções são exibidas.

```
?- mapa(teste1,Regioes),
   colore_mapa(mapa(teste1,Regioes),[azul,verde,amarelo,preto]).

Regioes = [regiao(a,azul,[verde,amarelo,verde]),
           regiao(b,verde,[azul,amarelo,azul]),
```

```
            regiao(c,amarelo,[azul,verde,verde,azul,preto]),
            regiao(d,verde,[azul,amarelo,preto]),
            regiao(e,azul,[verde,amarelo,preto]),
            regiao(f,preto,[amarelo,verde,azul])] ->;
                              ...
Regioes = [regiao(a,verde,[azul,amarelo,azul]),
            regiao(b,azul,[verde,amarelo,verde]),
            regiao(c,amarelo,[verde,azul,azul,verde,preto]),
            regiao(d,azul,[verde,amarelo,preto]),
            regiao(e,verde,[azul,amarelo,preto]),
            regiao(f,preto,[amarelo,azul,verde])] ->;
                              ...
Regioes = [regiao(a,amarelo,[azul,verde,azul]),
            regiao(b,azul,[amarelo,verde,amarelo]),
            regiao(c,verde,[amarelo,azul,azul,amarelo,preto]),
            regiao(d,azul,[amarelo,verde,preto]),
            regiao(e,amarelo,[azul,verde,preto]),
            regiao(f,preto,[verde,azul,amarelo])] ->;
                              ...
Regioes = [regiao(a,preto,[azul,verde,azul]),
            regiao(b,azul,[preto,verde,amarelo]),
            regiao(c,verde,[preto,azul,azul,amarelo,preto]),
            regiao(d,azul,[preto,verde,preto]),
            regiao(e,amarelo,[azul,verde,preto]),
            regiao(f,preto,[verde,azul,amarelo])] ->;
                              ...

?- mapa(teste2,Regioes),
   colore_mapa(mapa(teste2,Regioes),[azul,verde,amarelo,preto]).

Regioes = [regiao(1,azul,[verde,amarelo,amarelo,preto]),
            regiao(2,verde,[azul,amarelo,azul,amarelo,preto]),
            regiao(3,amarelo,[azul,verde,azul,preto]),
            regiao(4,azul,[verde,amarelo]),
            regiao(5,amarelo,[azul,verde,preto]),
            regiao(6,preto,[azul,verde,amarelo,amarelo])] ->;
                              ...
Regioes = [regiao(1,verde,[azul,amarelo,amarelo,preto]),
            regiao(2,azul,[verde,amarelo,verde,amarelo,preto]),
            regiao(3,amarelo,[verde,azul,verde,preto]),
            regiao(4,verde,[azul,amarelo]),
            regiao(5,amarelo,[verde,azul,preto]),
            regiao(6,preto,[verde,azul,amarelo,amarelo])] ->;
                              ...
Regioes = [regiao(1,amarelo,[azul,verde,verde,preto]),
            regiao(2,azul,[amarelo,verde,amarelo,verde,preto]),
```

```
            regiao(3,verde,[amarelo,azul,amarelo,preto]),
            regiao(4,amarelo,[azul,verde]),
            regiao(5,verde,[amarelo,azul,preto]),
            regiao(6,preto,[amarelo,azul,verde,verde])] ->;
                              ...
Regioes = [regiao(1,preto,[azul,verde,verde,amarelo]),
            regiao(2,azul,[preto,verde,amarelo,verde,amarelo]),
            regiao(3,verde,[preto,azul,amarelo,amarelo]),
            regiao(4,amarelo,[azul,verde]),
            regiao(5,verde,[preto,azul,amarelo]),
            regiao(6,amarelo,[preto,azul,verde,verde])] ->;
                              ...
```

Uma outra abordagem para a resolução deste problema, encontrada em [Coelho 88], é feita através da discriminação de qual cor pode ficar perto de qual outra. No programa que se segue isso é realizado pelo procedimento `proxima/2`.

Neste programa, o procedimento `colorir/6` descreve o mapa — discriminando, através do procedimento `proxima/2`, cada uma das regioes e suas regiões vizinhas.

```
proxima(azul,verde).
proxima(azul,amarelo).
proxima(azul,preto).
proxima(verde,azul).
proxima(verde,amarelo).
proxima(verde,preto).
proxima(amarelo,azul).
proxima(amarelo,verde).
proxima(amarelo,preto).
proxima(preto,azul).
proxima(preto,verde).
proxima(preto,amarelo).

colorir(R1,R2,R3,R4,R5,R6):-
                    proxima(R1,R2),
                    proxima(R1,R3),
                    proxima(R1,R5),
                    proxima(R1,R6),
                    proxima(R2,R3),
                    proxima(R2,R4),
                    proxima(R2,R5),
                    proxima(R2,R6),
                    proxima(R3,R4),
                    proxima(R3,R6),
                    proxima(R5,R6).
```

Segue-se um exemplo de execução deste programa, onde apenas algumas das soluções

são exibidas.

```
?- colorir(A,B,C,D,E,F).
A = azul
B = verde
C = amarelo
D = azul
E = amarelo
F = preto ->;
   ...
A = verde
B = azul
C = amarelo
D = verde
E = amarelo
F = preto ->;
   ...
A = amarelo
B = azul
C = verde
D = amarelo
E = verde
F = preto ->;
  ...
A = preto
B = azul
C = verde
D = amarelo
E = verde
F = amarelo ->;
  ...
```

7.7 Caminhos em Grafos

Descrição: Seja o grafo (sem loops) da Figura 8, que exibe as distâncias existentes entre cidades adjacentes.

ele pode ser representado pelos seguintes fatos Prolog:

```
va_para(a,b,100).
va_para(a,e,50).
va_para(b,c,200).
va_para(c,d,20).
va_para(e,c,30).
va_para(e,d,70).
```

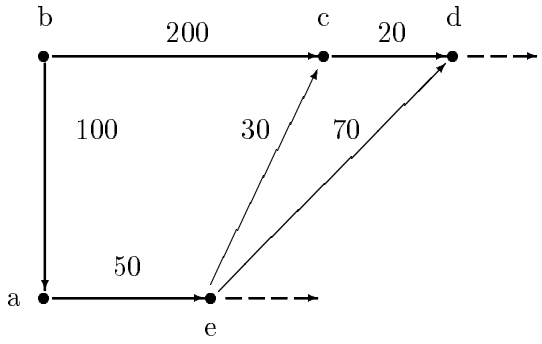


Figure 8: Distância entre Cidades

É simples definir um procedimento para encontrar todas as cidades que estão conectadas bem como a distância total entre elas. O programa é:

```
pode_ir(X,X,0).
pode_ir(Ini,Fim,Dist):- va_para(Ini,Adj,D1),
                        pode_ir(Adj,Fim,D2),
                       Dist is D1 + D2.
```

A primeira condição — condição de parada — especifica que a distância de uma cidade para ela mesma é zero. A segunda cláusula especifica que para ir de uma cidade `Ini` para uma outra `Fim` e distância total `Dist`, deve-se encontrar primeiro uma cidade `Adj` adjacente à cidade de partida `Ini` e a distância `D1` correspondente, para depois resolver recursivamente o problema de encontrar a distância `D2` partindo agora dessa cidade `Adj` até a cidade procurada `Fim`. Finalmente, a soma dessa duas distâncias é a distância total `Dist`. Segue um exemplo de interrogação:

```
?- pode_ir(a, c, X).
X = 300 ->;
X = 80 ->;
no
```

Entretanto, o procedimento acima não lembra o caminho percorrido. O seguinte programa resolve esse problema:

```
caminho(Ini,Fim,Dist,Cam):- caminho1(Ini,[Fim],0,Dist,Cam).
```

A idéia é que para determinar um caminho `Cam` da cidade `Ini` até a cidade `Fim`, o qual será uma lista `[Ini,Cid1,Cid2,...,Fim]` começa-se via `caminho1/5`, a construir um caminho auxiliar cuja única cidade presente inicialmente é a última cidade no caminho solicitado: `Fim`. Assim, o único nó que `caminho1/4` precisa ter como argumento é a cidade inicial que será usada para verificar a condição de parada. O programa é:

```
caminho1(Cid,[Cid|Cids],Dist,Dist,[Cid|Cids]).

caminho1(Ini,[Adj|Cids],Dist,DistF,CamF):-
                va_para(Interm,Adj,D1),
                D2 is Dist + D1,
                caminho1(Ini,[Interm,Adj|Cids],D2,DistF,CamF).
```

Segue um exemplo de interrogação.

```
?- caminho(a,d,Dist,Cam).
Dist = 320
Cam = [a,b,c,d] ->;

Dist = 100
Cam = [a,e,c,d] ->;

Dist = 120
Cam = [a,e,d] ->;
no
```

Deve ser observado que o programa acima insere as cidades adjacentes no caminho que está sendo construído sem verificar se elas já foram visitadas anteriormente; somente trabalha se não for possível visitar duas vezes a mesma cidade. Entretanto, se os seguintes fatos fossem adicionados

```
va_para(d,e,70).
va_para(b,a,100).
va_para(e,b,150).
```

este programa entraria em loop. É então necessário verificar se uma cidade candidata já faz parte do caminho até então construído, antes de inserí-la. Seguem o programa que realiza esta verificação e um exemplo de execução.

```
caminho_sl(Ini,Fim,Dist,Cam):-
        caminho1_sl(Ini,[Fim],0,Dist,Cam).

caminho1_sl(Cid,[Cid|Cids],Dist,Dist,[Cid|Cids]).

caminho1_sl(Ini,[Adj|Cids],Dist,DistF,CamF):-
         va_para(Interm,Adj,D1),
         not pertence1(Interm,[Adj|Cids]),
         D2 is Dist + D1,
         caminho1_sl(Ini,[Interm,Adj|Cids],D2,DistF,CamF).

pertence1(X,[X|_]) :- !.
```

```
pertence1(X,[_|Cau]) :- pertence1(X,Cau).
```

```
?- caminho_sl(d,X,Dist,Cam).
 X = d
 Dist = 0
 Cam = [d] ->;

 X = c
 Dist = 420
 Cam = [d,e,b,c] ->;

 X = c
 Dist = 100
 Cam = [d,e,c] ->;

 X = e
 Dist = 70
 Cam = [d,e] ->;

 X = a
 Dist = 320
 Cam = [d,e,b,a] ->;

 X = b
 Dist = 220
 Cam = [d,e,b] ->;
 no
```

7.8 Caminho do Cavalo no Tabuleiro de Xadrez

Descrição: O problema consiste em encontrar possíveis caminhos percorrido por um cavalo num tabuleiro de xadrez vazio, dado, pelo menos, a sua posição inicial.

A Figura 9 mostra os possíveis movimentos de um cavalo no jogo de xadrez.

	3		2	
4				1
		c		
5				8
	6		7	

Figure 9: Possíveis Movimentos de um Cavalo

ou seja, se o cavalo se encontra em uma dada posição (x, y), ele pode ir para as posições:

$$(x + \delta x, y + \delta y) \text{ ou } (x + \delta y, y + \delta x)$$

$$\text{onde: } (\delta x, \delta y) \in \{\ (2,1), (2,-1), (-2,1), (-2,-1)\}$$

que pode ser implementado pelo predicado `pode_ir/2`, onde `;` representa *ou*.

```
pode_ir([X,Y],[X1,Y1]):-
        (deltaxy(Dx,Dy);deltaxy(Dy,Dx)),
        X1 is X + Dx,
        dentro_tabuleiro(X1),
        Y1 is Y + Dy,
        dentro_tabuleiro(Y1).

deltaxy(2,1).
deltaxy(2,-1).
deltaxy(-2,1).
deltaxy(-2,-1).
```

Como o tabuleiro de xadrez é de 8×8 então:

```
dentro_tabuleiro(X):- 0 < X,
                      X < 9.
```

Segue exemplo de execução de `pode_ir/2`. Deve ser observado que o primeiro argumento deve sempre estar instanciado.

```
?- pode_ir([5,5],X).
 X = [7,6] ->;

 X = [7,4] ->;

 X = [3,6] ->;

 X = [3,4] ->;

 X = [6,7] ->;

 X = [4,7] ->;

 X = [6,3] ->;

 X = [4,3] ->;
 no
```

Assim, a idéia é definir o predicado `caminho_cavalo(I,F,Caminho)` que, dada a posição inicial do cavalo `I`, dentro do tabuleiro de xadrez, encontra os possíveis caminhos para chegar à posição `F` dentro do tabuleiro.

O programa, mostrado a seguir, é semelhante ao definido na seção 8, pg.59 onde também se encontra definido o predicado `pertence1/2`.

```
caminho_cavalo(I,F,Caminho):-
        caminho_cavalo1(F,[I],Caminho).

caminho_cavalo1(X,[X|L],[X|L]).

caminho_cavalo1(I,[E|Cam],Caminho):-
        pode_ir(E,E1),
        not pertence1(E1,Cam),
        caminho_cavalo1(I,[E1,E|Cam],Caminho).
```

Segue um exemplo de execução:

```
?- caminho_cavalo([2,3],X,Cam).
X = [2,3]
Cam = [[2,3]] ->;

X = [4,4]
Cam = [[4,4],[2,3]] ->;

X = [6,5]
Cam = [[6,5],[4,4],[2,3]] ->;
        .......
X = [7,8]
Cam = [[7,8],[5,7],[7,6],[8,8],[6,7],[8,6],[6,5],[4,4],[2,3]]
        .......
```

7.9 Problema das 8 Rainhas

Descrição: O problema consiste em posicionar 8 rainhas em um tabuleiro de xadrez, inicialmente vazio, tal que não há ataque entre elas [Bratko 90], como mostra a Figura 10.

Uma possível forma de representar o tabuleiro é através da lista:

```
[[X1,Y1],[X2,Y2],..,[X8,Y8]]
```

onde cada um dos pares de coordenadas deve ser um inteiro num intervalo 1..8. Para que não aconteça ataque entre as rainhas é necessário que cada uma delas esteja em uma fila diferente do tabuleiro. Assim, a idéia é definir um predicado:

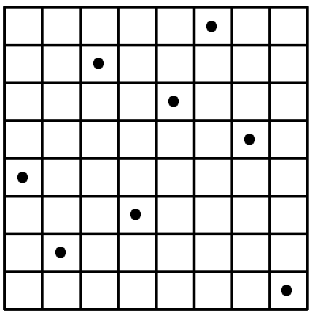


Figure 10: Uma Solução do Problema das Oito Rainhas

```
rainhas([[1,Y1],[2,Y2],...[8,Y8]])
```

que encontra os possíveis valores das colunas do tabuleiro onde cada rainha pode estar posicionada, de maneira que uma não ataque a outra. O procedimento é o seguinte:

```
rainhas([]).
rainhas([[X,Y]|Outras]) :-
                    rainhas(Outras),
                    pertence(Y,[1,2,3,4,5,6,7,8]),
                    nao_ataca([X,Y],Outras).
```

onde o programa `pertence/2` está definido na seção 4.1 pg. 12. A primeira cláusula `rainhas/1` verifica que a lista das posições das rainhas está vazia, e consequentemente, não há ataque entre elas. A segunda cláusula trata da lista não vazia do tipo `[[X,Y]|Outras]`. Neste caso, `Outras` deve ser uma solução para o problema, `Y` deve estar no tabuleiro e `[X,Y]` deve ser uma nova posição válida considerando a solução fornecida em `Outras`. Assim tem-se:

```
nao_ataca(_,[]).
nao_ataca([X,Y],[[X1,Y1]|Outras]) :-
                     Y \= Y1,
                     T is Y1-Y, T2 is X1-X, T \= T2,
                     T3 is X-X1, T \=T3,
                     nao_ataca([X,Y],Outras).
```

No exemplo de execução dado a seguir, o precedimento `t1(X)` simplesmente instancia `X`; sua definição é:

```
t1([[1,Y1],[2,Y2],[3,Y3],[4,Y4],[5,Y5],[6,Y6],[7,Y7],[8,Y8]]).
```

```
?- t1(X),rainhas(X).
```

```
X = [[1,4],[2,2],[3,7],[4,3],[5,6],[6,8],[7,5],[8,1]] ->;
X = [[1,5],[2,2],[3,4],[4,7],[5,3],[6,8],[7,6],[8,1]] ->;
X = [[1,3],[2,5],[3,2],[4,8],[5,6],[6,4],[7,7],[8,1]] ->;
                    ....
X = [[1,5],[2,7],[3,2],[4,6],[5,3],[6,1],[7,4],[8,8]] ->;
no
```

Em [Bratko 90], pp. 114-123 pode ser encontrada uma excelente descrição de diferentes versões para a solução deste problema visando diminuir o espaço de busca.

7.10 Alocação de Escritórios de uma Firma

Seja o predicado `pegue(N,B,L1,L2)` que fornece informações sobre o número de elementos que um conjunto de caixas pode conter, onde:

- `N` é o número de ítens a serem selecionados;
- `B` é o número que identifica a caixa;
- `L1` é uma lista de inteiros não negativos que indicam a quantidade de elementos em cada caixa;
- `L2` é uma lista obtida a partir de `L1`, da seguinte forma: têm os mesmos elementos de `L1`, exceto pelo elemento que ocupa a posição `B`, cujo valor é o valor anterior, menos `N`.

Por exemplo, o resultado da interrogação

```
?- pegue(3,3,[0,2,7,4], L2).
L2 = [0,2,4,4] ->;
no
```

significa que foram retirados `N = 3` elementos da caixa `B = 3`, que possui 7 elementos, de acordo com a lista `L1=[0,2,7,4]`, dando como resultado a nova configuração do número de elementos de cada caixa, através de `L2=[0,2,4,4]`.

Considerando que `N` e `L1` sejam dados — B e L2 podem ser variáveis não instanciadas — a implementação do programa é:

```
pegue(N,1,[X|Y],[X1|Y]):- X1 is X-N,
                          X1 >= 0.

pegue(N,B,[X|Y1],[X|Y2]):- pegue(N,B1,Y1,Y2),
                           B is B1+1.
```

Esse programa será utilizado como programa auxiliar na resolução do problema descrito a seguir.

Descrição: Uma firma está mudando para os 3 andares de um novo prédio. Cada andar tem 40 escritórios. A tabela a seguir fornece o nome de cada um dos departamentos da firma e o número de escritórios necessários a cada um deles [Rowe 88].

Departamento	Nro. Escritórios
m 43	09
d 48	06
m 77	11
m 54	13
j 39	09
j 76	12
m 20	11
j 92	11
j 83	07
j 64	10
m 06	11
j 48	10

Após a mudança, todos os escritórios de um mesmo departamento devem estar localizados num mesmo andar. Além disso, escritórios que interagem estreitamente devem também estar no mesmo andar. Eles são:

m 43	e	m 77
d 48	e	m 54
j 92	e	j 83

A solução deste problema pode ser oferecida em forma de triplas `[D,N,A]` onde:

- `D` é o identificador de departamento
- `N` é o número de escritórios necessários para o departamento `D`
- `A` é o andar designado para o departamento `D`

`D` e `N` são dados, e `A` deve ser determinado.

A solução do problema é dada pelo predicado `alocar(L,Disponivel)` onde `L` é a lista de triplas para cada departamento e `Disponivel` dá o número de escritórios disponíveis para cada andar. Usando a versão de `pegue/4` definida anteriormente, `alocar/2` requer somente uma regra recursiva e uma regra incondicional que nada faz quando `L = [ ]`. O programa é:

```
alocar(L,Disp):-inicializa(L),
                aloc(L,Disp).

aloc([],Disp).
aloc([(Dep,Num,Andar)|Cauda],Disp):-
                pegue(Num,Andar,Disp,Prox_disp),
                aloc(Cauda,Prox_disp).
```

onde o predicado `inicializa/1` simplesmente instancia `L` com a configuração descrita na tabela acima. Ele é definido como:

```
inicializa([(m43,9,A1),(d48,6,A2),(m77,11,A1),
            (m54,13,A2),(j39,9,A3),(j76,12,A4),
            (m20,11,A5),(j92,11,A6),(j83,7,A6),
            (j64,10,A7),(m06,11,A8),(j48,10,A9)]).
```

A solução do problema é obtida através da seguinte interrogação:

```
?- alocar(L,[40,40,40]).
L = [(m43 , 9 , 1),(d48 , 6 , 2),(m77 , 11 , 1),(m54 , 13 , 2),
     (j39 , 9 , 1),(j76 , 12 , 3),(m20 , 11 , 1),(j92 , 11 , 3),
     (j83 , 7 , 3),(j64 , 10 , 2),(m06 , 11 , 2),(j48 , 10 , 3)] ->;

L = [(m43 , 9 , 1),(d48 , 6 , 2),(m77 , 11 , 1),(m54 , 13 , 2),
     (j39 , 9 , 1),(j76 , 12 , 3),(m20 , 11 , 1),(j92 , 11 , 3),
     (j83 , 7 , 3),(j64 , 10 , 3),(m06 , 11 , 2),(j48 , 10 , 2)] ->;
                                  ....
L = [(m43 , 9 , 3),(d48 , 6 , 2),(m77 , 11 , 3),(m54 , 13 , 2),
     (j39 , 9 , 3),(j76 , 12 , 1),(m20 , 11 , 3),(j92 , 11 , 1),
     (j83 , 7 , 1),(j64 , 10 , 2),(m06 , 11 , 2),(j48 , 10 , 1)]->;
no
```

Apenas são mostradas a primeira, a segunda e a última solução — de um total de 30.

8 Conclusão

Foram mostradas neste trabalho a implementação de diversos programas para processamento de listas que são considerados necessários para uma aprendizagem inicial da linguagem lógica Prolog.

Deve ser salientado que é possível melhorar consideravelmente a eficiência de alguns destes programas que manipulam listas usando uma estrutura de dados alternativa para representar uma sequência de elementos, chamada listas-diferença. Esta estrutura é descrita em vários dos livros de texto refenciados, por exemplo [Bratko 90], [Clocksin 87], [Sterling 86], [O'Keefe 90], etc.

Um exercício interessante é verificar quais dos programas se adequam ao uso de listas-diferenças e então, escrever uma segunda versão do mesmo, usando tal estrutura. Para evidenciar a eficiência de lista-diferenças, é importante que se compare os tempos de execução das duas versões.

References

[Amble 87] Amble T. *Logic Programming and Knowledge Engineering.* Addison-Wesley, 1987.

[Araribóia 89] Araribóia, G. *Inteligência Artificial: Um Enfoque Prático.* LTC, 1989.

[Arity 90] Arity Corporation. *The Arity/Prolog Programming Language.* 1986.

[Balbin 84] Balbin, I.; Lecot, K. *Logic Programming A Classifed Bibliography.* Wilgrass Books Pty Ltd, Australia, 1984.

[Bharath 86] Bharath, R. *An Introduction to Prolog.* TAB Books Inc., 1986.

[Bharath 89] Bharath, R. *Prolog: Sophisticated Applications in Artificial Intelligence.* Windcrest Books, 1989.

[Bharath 89] Bharath, R. *Prolog: Sophisticated Applications in Artificial Intelligence.* Windcrest Books, 1989.

[Bratko 90] Bratko, I. *Prolog Programming for Artificial Intelligence.* Addison-Wesley, (2nd Ed.), 1990.

[Burnham 85] Burnham, W.D.; Hall, A.R. *Prolog Programming and Applications.* MacMillan Publishing Company, 1985.

[Casanova 87] Casanova, M.A.; Giorno, F.A.C.; Furtado, A.L. *Programação em Lógica e a Linguagem Prolog.* Editora Blücher Ltda, São Paulo, 1987.

[Clark 84] Clark, K.L.; McCabe, K.G. *Micro-PROLOG: Programming in Logic.* (2nd Ed.). Prentice-Hall, 1984.

[Clocksin 87] Clocksin, W.F.; Mellish, C.S. *Programming in Prolog.* (3rd Ed.) Springer-Verlag, 1987.

[Coelho 88] Coelho, H.; Cotta, J.C. *Prolog by Examples.* Springer-Verlag, 1988.

[Dodd 90] Dodd, T. *Prolog: a Logical Approach.* Oxford University Press, 1990.

[Ford 87] Ford, N. *How Machines Think.* John Wiley & Sons, 1987.

[Ford 89] Ford, N. *Prolog Programming.* John Wiley & Sons, 1989.

[Garavaglia 87] Garavaglia, S. *Prolog: Programming Techniques and Applications.* Harper & Row, 1987.

[Hoare 62] Hoare, C.A.R. *Quicksort.* Computer Journal 5, pp 10-15, 1962.

[Hogger 84] Hogger, C.J. *Introduction to Logic Programming.* Academic Press, 1984.

[Kluzniak 85] Kluzniak, F.;Szpakowicz, S. *Prolog for Programmers.* Academic Press, 1985.

[Kowalski 79] Kowalski, R.A. *Logic for Problem Solving.* Artificial Intelligence Series, Vol 7, Elsevier-North Holland, 1979.

[Lloyds 84] Lloyds, J.W. *Foundations of Logic Programming.* Springer-Verlag, 1984.

[Malpas 87] Malpas, J. *Prolog:A Relational Language and its Applications.* Prentice-Hall, 1987.

[Marcus 86] Marcus, C. *Prolog Programming: Aplications for Database System, Expert Systems and Natural Languages System.* Addison-Wesley, 1986.

[Monard 88] Monard, M.C.; Lima Vidal, R. *Programas Prolog para Processamento de Listas.* ILTC, 52 pg., 1988.

[Monard 89] Monard, M.C.; Lima Vidal, R.; Nicoletti, M.C. *Programas Prolog para Processamento de Árvores.* ILTC, 74 pg., 1988.

[O'Keefe 90] O'Keefe, R.A. *The Craft of Prolog.* The MIT Press, 1990.

[Rogers 86] Rogers, J.B. *A Prolog Primer.* Addison-Wesley, 1986.

[Rowe 88] Rowe, N.C. *Artificial Intelligence Through Prolog.* Prentice Hall, 1988.

[Sterling 86] Sterling, L.; Shapiro, E. *The Art of Prolog.* The MIT Press, 1986.

[Walker 87] Walker, A.(Ed); McCord, M.; Souza, J.F.; Wilson, W.G. *Knowledge Systems and Prolog.* Addison-Wesley, 1987.

9 Apéndice

O predicado `tempo_exec(Pred,Nvezes,Tempo)` pode ser utilizado para medir o tempo de execução — `Tempo` — de um predicado — `Pred`.

Para fazer isto, `temp_exec/3` executa `Nvezes` o predicado `Pred`, e calcula a média dos tempos de execução obtidos. Como o próprio predicado `tempo_exec/3` tem também seu tempo de execução, tal tempo não deve ser contabilizado e é, consequentemente, subtraído do tempo total obtido, relativo às execuções de `Pred`.

O tempo de execução do predicado `tempo_exec/3` é simulado através da execução de `loop_repeat(Nvezes)`.

Deve ser observado que `Tempo` é expresso com precisão de até centésimos de segundo. Isto faz com que `temp_exec/3` possa ser usado apenas para medir tempos de execução de predicados que executam em tempo maior do que 1 centésimo de segundo.

```
% tempo_exec(+Pred, +Nvezes, ?Tempo)
tempo_exec(Pred,Nvezes,Tempo):-
        inteiro_positivo(Nvezes),    % verifica Nvezes eh inteiro > 0
        time(T1),                    % tempo inicial
        execute(Pred,Nvezes),
        time(T2),                    % tempo final total
        loop_repeat(Nvezes),         % executa o loop repeat
        time(T3),
        tempo(T2-T1,T12),            % T12 tempo total
        tempo(T3-T2,T23),            % T23 tempo usado pelo loop_repeat
        Tempo is T12 - T23,          % tempo real da execucao de Pred
        escreva1(Pred,Nvezes,Tempo), % imprima tempo medio
        !.

inteiro_positivo(N):- integer(N),
                      N > 0, !.

inteiro_positivo(_):-
        nl, write('ERRO: <arg-2> deve ser um numero inteiro > 0'),nl,
        fail.

% execute(+Pred, +Nvezes)
% executa o predicado Pred Nvezes dentro de um repeat loop

execute(Pred,Nvezes):- ctr_set(0,1),
                       repeat,
                       Pred,
                       ctr_inc(0,X),
                       X == Nvezes,
                       !.
```

```
% loop_repeat(+Nvezes)
% executa um "dummy" loop_repead Nvezes

loop_repeat(Nvezes):- ctr_set(0,1),
                      repeat,
                      ctr_inc(0,X),
                      X == Nvezes,
                      !.

% tempo(+F-I, ?T)
% calcula T = F - I onde F e I sao time(H,M,S,Cs)

tempo(time(Hf,Mf,Sf,Csf) - time(Hi,Mi,Si,Csi), T) :-
                      H is (Hf - Hi) * 360000,
                      M is (Mf - Mi) * 6000,
                      S is (Sf - Si) * 100,
                      Cs is Csf - Csi,
                      T is H + M + S + Cs.

escreva1(Pred,Nvezes,Tempo):-
        nl,tab(10),write('Predicado = '),write(Pred),nl,
        tab(10),write('No_vezes = '),write(Nvezes),nl,
        tab(10),write('Tempo = '),write(Tempo),nl,
        Tmedio is Tempo / Nvezes,
        tab(10),write('Tmedio = '),write(Tmedio),nl.
```

Segue um exemplo de cálculo do tempo de execução do procedimento `classifica/2`, definido na seção 7.3, pg.49.

```
 ?- tempo_exec(classifica([10,2,9,11,-3,15,1,4,13,-5,0,16,7,3,-1,5,14,
                           -4,8,6,-2,17,12],20,L).

 Predicado = classifica([10,2,9,11,-3,15,1,4,13,-5,0,16,7,3,-1,5,14,
                         -4,8,6,-2,17,12]
 No_vezes = 20
 Tempo = 1335
 Tmedio = 66.75
```