Exercícios:

1. Defina a relação `flatten/2` (que lineariza uma lista) de forma a que, por exemplo:

```
| ?- flatten([a,b,[c,d],[],[[[e,f]],g],h],X).
X = [a,b,c,d,e,f,g,h] ?
yes
```

2. Escreva um programa para reconhecer se uma fórmula da lógica proposicional está na forma normal conjuntiva, ou seja, é uma conjunção de disjunções de literais. Um literal é um símbolo proposicional ou a sua negação.

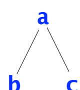
   Considere a declaração das seguintes conectivas lógicas:

```
:- op(500,yfx,/\).
:- op(500,yfx,\/).
:- op(300,fy,~).
```

3. Defina o predicado `conta_ocorr/3` para contar quantas vezes uma constante ocorre numa lista. (Sugestão: usar `atomic/1`).

4. Suponha que tem factos da forma *quadrado(Lado)*.
   Defina o predicado `zoom(+X,?Y,+F)` tal que `Y` é o quadrado que resulta de multiplicar pelo factor `F` os lados do quadrado `X`. (Sugestão: usar `=..`).

Um exemplo com árvores binárias



Esta árvore é representada pelo termo

nodo(a, nodo(b,vazia,vazia), nodo(c,vazia,vazia))

Exemplo:

```
arv_bin(vazia).
arv_bin(nodo(X,Esq,Dir)) :- arv_bin(Esq), arv_bin(Dir).

na_arv(X,nodo(X,_,_)).
na_arv(X,nodo(Y,Esq,_)) :- na_arv(X,Esq).
na_arv(X,nodo(Y,_,Dir)) :- na_arv(X,Dir).
```

Exercícios:

1. Defina predicados que permitam fazer as travessias *preorder*, *inorder* e *postorder*.
2. Defina um predicado `search_tree/1` que teste se uma dada árvore é uma árvore binária de procura.
3. Defina a relação `insert_tree(+X,+T1,?T2)` que sucede se `T2` é uma árvore binária de procura resultado da inserção de `X` na árvore binária de procura `T1`.
4. Defina a relação `path(+X,+Tree,?Path)` que sucede se `Path` é o caminho da raiz da árvore binária de procura `Tree` até `X`.

O *cut* !

O *cut* ! não deve ser visto como um predicado lógico. Apenas interfere na semântica procedimental do programa. A sua acção é a seguinte:

Durante o processo de prova, a 1ª passagem pelo cut é sempre verdadeira (com sucesso). Se por backtracking se voltar ao cut, então o cut faz falhar o predicado que está na cabeça da regra.

O cut “corta” ramos da árvore de procura. Os predicados antes do *cut* são apenas instanciados uma vez.

Exemplo:

```
x :- p, !, q.
x :- r.
```

`x` tem um comportamento semelhante a

`if p then q else r`

Green Cut

Chamam-se ***green cuts*** aos *cuts* que só alteram a semântica procedimental do programa, mas não alteram o significado do predicado. Este *cuts* são usados apenas para melhorar a eficiência. Se forem retirados serão produzidos os mesmos resultados.

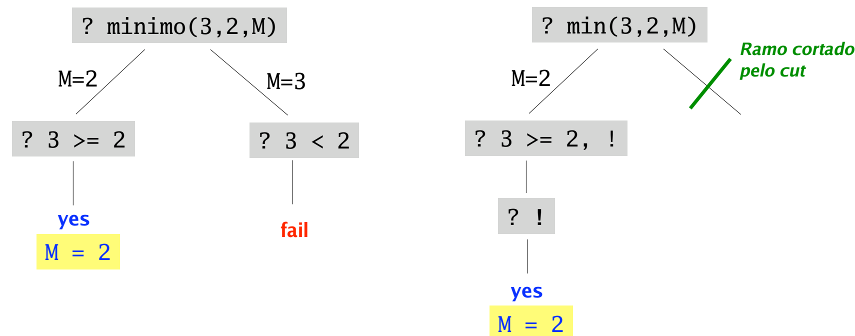
Red Cut

Chamam-se ***red cuts*** aos *cuts* que alteram não só a semântica procedimental do programa, como também o seu significado declarativo. Se forem retirados serão produzidos resultados diferentes. A utilização destes *cuts* deve ser evitada.

Exemplos: `minimo` e `min` são predicados equivalentes. O cut só está a cortar ramos que falham.

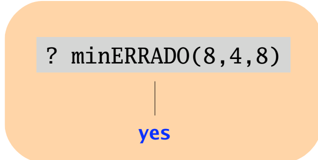
```
minimo(X,Y,Y) :- X >= Y.
minimo(X,Y,X) :- X < Y.
```

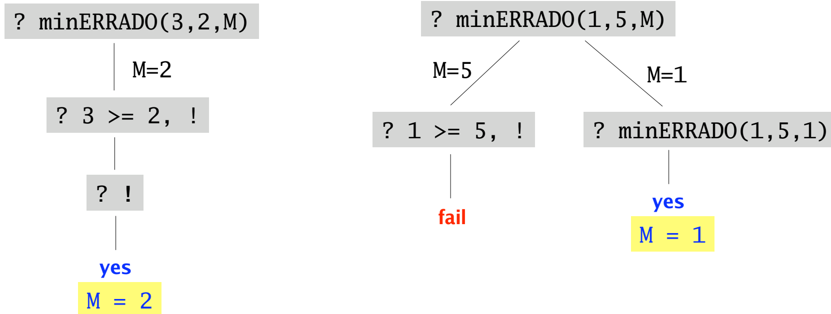
```
min(X,Y,Y) :- X >= Y, !.
min(X,Y,X) :- X < Y.
```



Exemplo de utilização *errada* do cut.

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





Exercícios:

1. Defina um predicado `no_dupl/2` que remova os duplicados de uma lista.

2. Defina um predicado `enumerar(+N,+M,+P,?L)` que gera a lista `L` de números entre `N` e `M`, a passo `P`. Por exemplo:

```
| ?- enumerar(3,10,2,L).
L = [3,5,7,9] ?
yes
```

3. Defina um programa que faça a ordenação de uma lista pelo algoritmo *merge sort*. Use o *cut* para implementar o predicado `merge` de forma mais eficiente.

Negação por falha

O Prolog tem pré-definidos os predicados **`true`** e **`fail`**: **`true`** sucede sempre, **`fail`** falha sempre

Utilizando o *cut* e o *fail* é possível implementar a negação.

Exemplo:

```
:- op(700,fy,nao).

nao X :- X, !, fail.
nao X.
```

`nao X` falha se o predicado X tiver solução, i.e., for verdadeiro.

No *Sicstus Prolog* o predicado de negação (por falha) é o `\+` .

Exemplos:

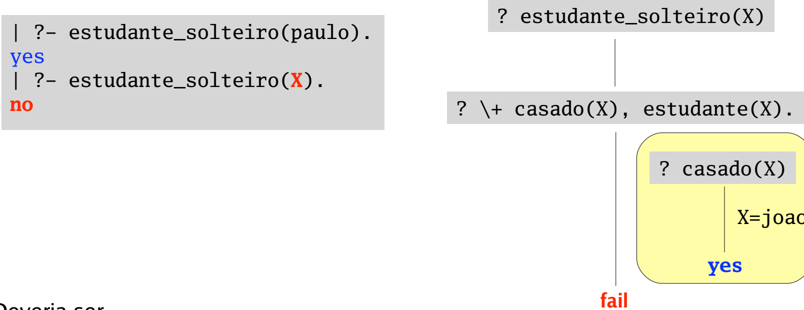
```
| ?- fib(1,1).
yes
| ?- nao fib(1,1).
no
| ?- \+ fib(1,1).
no
```

```
| ?- nao nao fib(1,1).
yes

| ?- \+ \+ fib(1,1).
yes
```


Problemas com a negação por falha

Exemplo:

```
estudante(paulo).
casado(joao).

estudante_solteiro(X) :- \+ casado(X), estudante(X).
```



Deveria ser

```
estSolt(X) :-  estudante(X), \+ casado(X).
```

```
| ?- estSolt(X).
X = paulo ?
yes
```

Nas chamadas ao not, \+, não devem ocorrer variáveis.

Exemplo:

```
| ?- \+ X=2, X=1.
no
```

```
? \+ X = 2, X = 1.
    ? X = 2
      X=2
      yes
fail
```

```
| ?- X=1, \+ X=2.
X = 1 ?
yes
```

```
? X = 1, \+ X = 2.
   X=1
? \+ 1 = 2.
    ? 1 = 2
      fail
yes
```


Outros predicados de controlo

Para além da *conjunção* de predicados (representada por **,**), também é possível combinar predicados pela *disjunção* (representada por **;**).

Exemplo:

```
progenitor(A,B) :- pai(A,B).
progenitor(A,B) :- mae(A,B).

avo(X,Y) :- progenitor(X,Z), progenitor(Z,Y).
```

```
progenitor(A,B) :- pai(A,B) ; mae(A,B).
```

```
tio(X,Y) :- (pai(A,Y) ; mae(A,Y)), irmao(X,A).
```

O Sicstus Prolog disponibiliza mais alguns predicados de control *(ver User's Manual)*.
Por exemplo:

`+P -> +Q; +R`	equivalente a	`(P -> Q; R) :- P, !, Q.` `(P -> Q; R) :- R.`
`+P -> +Q`	equivalente a	`(P -> Q; fail)`
`once(+P)`	equivalente a	`(P -> true; fail)`

Programação de 2ª ordem

Existem meta-predicados que permitem coleccionar todas as soluções para um dado objectivo de prova (ver *User's Manual*).

`findall(?Template,:Goal,?Bag)`

Bag é a lista de instâncias de *Template* encontradas nas provas de *Goal*.
A ordem da lista corresponde à ordem em que são encontradas as respostas.
Se não existirem instanciações para *Template*, *Bag* unifica com a lista vazia.

`bagof(?Template,:Goal,?Bag)`

Semelhante a `findall`, mas se *Goal* falhar, `bagof` falha.

`setof(?Template,:Goal,?Set)`

Semelhante a `bagof`, mas a lista é ordenada e sem repetições.

Exemplo: Considere o seguinte programa

```
amigo(ana,rui).
amigo(pedro,rui).
amigo(maria,helena).
amigo(pedro,ana).
amigo(maria,rui).

gosta(ana,cinema).
gosta(ana,pintura).
gosta(ana,ler).
gosta(rui,ler).
gosta(rui,musica).
gosta(maria,ler).
gosta(pedro,pintura).
gosta(pedro,ler).

compativeis(A,B) :- amigo(A,B), gosta(A,X), gosta(B,X).
compativeis(A,B) :- amigo(B,A), gosta(A,X), gosta(B,X).
```

```
| ?- compativeis(ana,X).
X = rui ? ;
X = pedro ? ;
X = pedro ? ;
no
```

```
| ?- findall(X,compativeis(ana,X),L).
L = [rui,pedro,pedro] ?
yes
| ?- bagof(X,compativeis(ana,X),L).
L = [rui,pedro,pedro] ?
yes
| ?- setof(X,compativeis(ana,X),L).
L = [pedro,rui] ?
yes
```

```
| ?- compativeis(helena,X).
no
```

```
| ?- findall(X,compativeis(helena,X),L).
L = [] ?
yes
| ?- bagof(X,compativeis(helena,X),L).
no
| ?- setof(X,compativeis(helena,X),L).
no
```

```
| ?- setof(comp(X,Y),compativeis(X,Y),L).
L = [comp(ana,pedro),comp(ana,rui),comp(maria,rui),comp(pedro,ana),
comp(pedro,rui),comp(rui,ana),comp(rui,maria),comp(rui,pedro)] ?
yes
```


Exercícios:

1. Defina o predicado `subconj(-S,+C)` onde S e C são duas listas que representam dois conjuntos. Este predicado deve gerar, por backtracking, todos os subconjuntos possíveis de C.

2. Defina o predicado `partes(+C,-P)`, que dado um conjunto C (implementado como lista) dá em P o conjunto de todos os subconjuntos de C.

3. Considere a linguagem proposicional gerada pelos simbolos proposicionais (átomos) e as conectivas: `falso`, `verdade`, `~`, `/\`, `\/`, `=>`. Relembre que um modelo é um conjunto de simbolos proposicionais.

```
:- op(600,xfy,=>).
:- op(500,yfx,/\).
:- op(500,yfx,\/).
:- op(300,fy,~).
```

Defina o predicado `atrib(+M,+P,?V)` que sucede se V é o valor de verdade da proposição P no modelo M. (Relembre a função μ da aulas teóricas).
Pode utilizar o predicado `simp(+E,?V)` que faz o cálculo do valor de uma expressão na álgebra booleana $\mathbb{Z}_2$ (ver slide seguinte).


```
simp(X+X,0).
simp(X*X,X).
simp(X+0,X).
simp(0+X,X).
simp(X*0,0).
simp(0*X,0).

simp(X+Y,Z) :- simp(X,X1), simp(Y,Y1), simp(X1+Y1,Z).
simp(X*Y,Z) :- simp(X,X1), simp(Y,Y1), simp(X1*Y1,Z).
```

4. Defina os predicados `formula_valida(+M,+P)` e `teoria_valida(+M,+T)` que sucedem se a proposição P e teoria T é válida no modelo M.

```
| ?- formula_valida([p,q], p/\q => q\/r => ~r /\ ~ ~p).
yes
```

5. Defina o predicado `consequencia(+T,+P)` sucede se a proposição P é uma consequência (semântica) da teoria T. (Sugestão: gere primeiro todos os modelos possíveis com os simbolos proposicionais das fórmulas envolvidas.)

6. Defina `tautologia/1` que testa se uma fórmula é uma tautologia.

7. Defina `inconsistente/1` que testa se uma teoria é inconsistente.

8. Defina os predicados `soma/3` e `produto/3` que implementam as operações de soma e produto de conjuntos de conjuntos de literais (ver apontamento da aulas teóricas).

9. Defina o predicado `fnn(+P,-FNN)` que dada uma proposição (da linguagem do exercício 3.) gera a forma normal negativa que lhe é semanticamente equivalente.

10. Defina o predicado `fnc(+FNN,-L)` que dada uma forma normal negativa, produz em L a forma normal conjuntiva equivalente, representada por conjuntos de conjuntos de literais.

11. Defina o predicado `constroi_fnc(+L,-P)` que sucede se P é a proposição que o conjunto de conjuntos de literarais L representa.

12. Defina o predicado `gera_fnc(+P,-FNC)` que dada uma proposição P produz FNC uma proposição na forma normal conjuntiva, semanticamente equivalente a P.