# Módulos

Para além dos predicados pré-definidos, o Sicstus Prolog possui um vasto conjunto de **módulos** onde estão implementados muitos predicados que poderão ser úteis em diversas aplicações. Para poder utilizar estes predicados é necessário carregar os módulos onde eles estão definidos.

Os módulos estão organizados por temas. Alguns exemplos de módulos:

- **lists** – fornece os predicados de manipulação de listas
- **terms** – fornece os predicados de manipulação de termos
- **queues** – fornece uma implementação de filas de espera
- **random** – fornece um gerador de números aleatóreos
- **system** – fornece primitivas para aceder ao sistema operativo
- **ugraphs** – fornece uma implementação de grafos sem informação nos arcos
- **wgraphs** – fornece uma implementação de grafos com informação nos arcos
- **tcltk** – fornece um interface para o Tcl/Tk
- **vbsp** – fornece um interface para o Visual Basic
- **jasper** – fornece um interface para o Java
- ...

Para mais detalhes sobre módulos, consulte o *Sicstus User's Manual*.



# Declaração de Módulos

É possível declarar novos módulos, colocando no início do ficheiro uma directiva da forma:

```
:- module(ModuleName, ExportList).
```

Nome a dar ao módulo

Lista dos predicados a exportar pelo módulo

```
:- module(familia, [mae/2,pai/2,avo/2]).

mae(sofia,ana).
mae(ana,maria).

pai(rui,luis).
pai(luis,pedro).
pai(luis,maria).

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

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



# Exemplos de utilização de módulos

O carregamento / importação de módulos pode ser feita através dos predicados:

```
use_module(library(Package)).
use_module(ModuleName).
use_module(library(Package), ImportList).
```

**Exemplo:**

```
| ?- use_module(library(random)).

| ?- random(X).
X = 0.2163110752346893 ?
yes
| ?- random(X).
X = 0.6344657121210742 ?
yes
| ?- random(20,50,X).
X = 31 ?
yes
| ?- randseq(3, 100, L).
L = [24,34,85] ?
yes
```



**Exemplo:**

```
| ?- use_module(library(lists),[sublist/2,remove_duplicates/2]).

| ?- remove_duplicates([2,3,4,3,5,6,5,3,2],L).
L = [2,3,4,5,6] ?
yes
| ?- append([1,2,3],[6,7],L).
! Existence error in user: : /2
! procedure library(_95):append/3 does not exist
! goal:  library(_95):append([1,2,3],[6,7],_93)
| ?- sublist(S,[1,2,3]).
S = [1,2,3] ? ;
S = [2,3] ? ;
S = [3] ? ;
S = [] ? ;
S = [2] ? ;
S = [1,3] ? ;
S = [1] ? ;
S = [1,2] ? ;
no
```

**Exemplo:** Considere que o seguinte programs está gravado no ficheiro ***exemplo.pl***

```
:- use_module(familia).
:- use_module(library(lists),[append/3, member/2,
                              remove_duplicates/2]).

avos([],[]).
avos([H|T],L) :- findall(A,avo(A,H),L1), avos(T,L2),
                 append(L1,L2,L3), remove_duplicates(L3,L).

ocorre(_,[],[]).
ocorre(X,[L|T],[L|L1]) :- member(X,L), !, ocorre(X,T,L1).
ocorre(X,[L|T],L1) :- \+ member(X,L), ocorre(X,T,L1).
```

```
| ?- [exemplo].
| ?- avos([pedro,maria],X).
X = [rui,sofia] ?
yes
| ?- progenitor(Z,pedro).
! Existence error in user: : /2
! procedure library(_84):progenitor/2 does not exist
! goal:  library(_84):progenitor(_81,pedro)
| ?- ocorre(2,[[3,2,4],[4,5],[a,d,2,e],[4,3,3,a]],L).
L = [[3,2,4],[a,d,2,e]] ?
yes
```

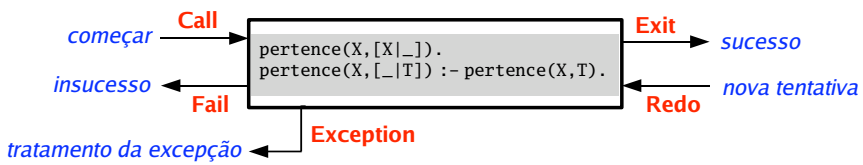


# Debugging

O interpretador Prolog possui um mecanismo de *debug*, que permite ter informação sobre os vários passos de execução (de prova) de um objectivo de prova.
A utilização deste mecanismo pode ser uma ferramenta preciosa na detecção e correcção de erros.

Uma forma de visualizar o fluxo de control de uma execução (incluindo o *backtracking*), é ver cada predicado/procedimento como uma caixa com as seguintes *"portas"*:

**Call** – lança um predicado para ser provado;
**Exit** – termina a prova do predicado com sucesso;
**Fail** – não consegue fazer a prova do predicado;
**Redo** – tenta construir uma nova prova para o predicado, forçada por backtracking;
**Exception** – ocorreu uma excepção.






# Tracing

A base do mecanismo de *debug* é a traçagem. A traçagem de um objectivo de prova, vai dando informação sobre os sucessivos passos da construção da prova (quais os predicados que vão sendo invocados e os argumentos da invocação).

Para activar o modo de traçagem faz-se, no interpretador, a invocação do predicado **trace**.
Quando o trace está activo o interpretador pára sempre uma das *"portas"* (do slide anterior) é activada.

```
| ?- trace.
% The debugger will first creep -- showing everything (trace)
yes
| ?- pertence(2,[1,2,3]).
       1      1 Call: pertence(2,[1,2,3]) ?
       2      2 Call: pertence(2,[2,3]) ?
?      2      2 Exit: pertence(2,[2,3]) ?
?      1      1 Exit: pertence(2,[1,2,3]) ?
yes
```



Alguns comandos úteis do modo *trace*:

| | | |
|---|---|---|
| h | help | lista todos os comandos disponíveis. |
| c | creep | avança mais um passo na prova (basta fazer *enter*). |
| l | leap | avança sempre até encontrar um *spypoint*. |
| s | skip | válido só para a porta Call ou Redo, faz com que não se desça mais no detalhe da prova desse *subgoal*. |
| f | fail | força a falha do *subgoal*. Passa o controle para a porta Fail. |
| r | retry | passa de novo o controle para a porta Call. |
| a | abort | Aborta a prova do predicado (e da traçagem). |

...

Para mais informações sobre debugging consulte o *Sicstus User's Manual*.

Para sair do modo de traçagem invoque, no interpretador, o prediado **notrace**.

```
| ?- notrace.
% The debugger is switched off
yes
```

**Exemplos:**

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

```
| ?- pertence(X,[1,2,3]).
        1      1 Call: pertence(_430,[1,2,3]) ? c
?       1      1 Exit: pertence(1,[1,2,3]) ?
X = 1 ? ;
        1      1 Redo: pertence(1,[1,2,3]) ? c
        2      2 Call: pertence(_430,[2,3]) ?
?       2      2 Exit: pertence(2,[2,3]) ?
?       1      1 Exit: pertence(2,[1,2,3]) ?
X = 2 ? ;
        1      1 Redo: pertence(2,[1,2,3]) ? s
?       1      1 Exit: pertence(3,[1,2,3]) ?
X = 3 ? ;
        1      1 Redo: pertence(3,[1,2,3]) ?
        2      2 Redo: pertence(3,[2,3]) ?
        3      3 Call: pertence(_430,[]) ?
        3      3 Fail: pertence(_430,[]) ?
        2      2 Fail: pertence(_430,[2,3]) ?
        1      1 Fail: pertence(_430,[1,2,3]) ?
no
```



```
| ?- pertence(X,[1,2,3]).
        1      1 Call: pertence(_430,[1,2,3]) ?
?       1      1 Exit: pertence(1,[1,2,3]) ?
X = 1 ? ;
        1      1 Redo: pertence(1,[1,2,3]) ? s
?       1      1 Exit: pertence(2,[1,2,3]) ?
X = 2 ? ;
        1      1 Redo: pertence(2,[1,2,3]) ? f
        1      1 Fail: pertence(_430,[1,2,3]) ?
no
| ?- pertence(X,[1,2,3]).
        1      1 Call: pertence(_430,[1,2,3]) ? l
X = 1 ? ;
X = 2 ? ;
X = 3 ? ;
no
```



**Exemplo:**

```
isort([],[]).
isort([H|T],L) :- isort(T,T1), ins(H,T1,L).

ins(X,[],[X]).
ins(X,[Y|Ys],[Y|Zs]) :- X > Y, ins(X,Ys,Zs).
ins(X,[Y|Ys],[X,Y|Ys]) :- X =< Y.
```

```
| ?- isort([3,2,8],L).
        1      1 Call: isort([3,2,8],_476) ?
        2      2 Call: isort([2,8],_1010) ? s
?       2      2 Exit: isort([2,8],[2,8]) ?
        3      2 Call: ins(3,[2,8],_476) ?
        4      3 Call: 3>2 ?
        4      3 Exit: 3>2 ?
        5      3 Call: ins(3,[8],_2316) ?
        6      4 Call: 3>8 ?
        6      4 Fail: 3>8 ?
        7      4 Call: 3=<8 ?
        7      4 Exit: 3=<8 ?
        5      3 Exit: ins(3,[8],[3,8]) ?
?       3      2 Exit: ins(3,[2,8],[2,3,8]) ?
?       1      1 Exit: isort([3,2,8],[2,3,8]) ?
L = [2,3,8] ? ;
        1      1 Redo: isort([3,2,8],[2,3,8]) ?
        3      2 Redo: ins(3,[2,8],[2,3,8]) ?
        8      3 Call: 3=<2 ?
        8      3 Fail: 3=<2 ?
        3      2 Fail: ins(3,[2,8],_476) ?
        2      2 Redo: isort([2,8],[2,8]) ?
        2      2 Fail: isort([2,8],_1010) ?
        1      1 Fail: isort([3,2,8],_476) ?
no
```



**Exercícios:**

1. Use a traçagem para confirmar a construção das árvores de procura que foram apresentas ao longo dos slides anteriores.

2. A seguinte definição pretende contar o número de ocorrências de um elemento numa lista, usando um parâmetro de acumulação.

```
conta_errado(X,L,N) :- contaAC(X,L,0,N).

contaAC(_,[],Ac,Ac).
contaAC(X,[H|T],Ac,N) :- X==H, Ac1 is Ac+1, contaAC(X,T,Ac1,N).
contaAC(X,[_|T],Ac,N) :- contaAC(X,T,Ac,N).
```

Mas este predicado não está correctamente definido. Por exemplo:

```
| ?- conta_errado(3,[3,2,3,4],N).
N = 2 ? ;
N = 1 ? ;
N = 1 ? ;
N = 0 ? ;
no
```

Faça debugging deste predicado para detectar o erro, e corrija-o.