**Predicado pré-definido** ->:

```
Teste -> Literais_if
```

ou

```
Teste -> Literais_if ; Literais_else
```

**Predicado** `nota` (usando o predicado ->):

```
nota_final(Nota) :-
   Nota>=10 -> write('Aprovado com '),
               write(Nota),
               writeln(' valores.')
               ;
               writeln('Reprovado.').
```

# 7.17 O PROLOG como linguagem de programação

**Predicados pré-definidos para a manipulação de termos:**

`functor(T_c, F, Ar)` afirma que

o termo composto `T_c` utiliza o functor `F` com aridade `Ar`.

**Exemplo 7.17.4**

```
?- functor(ad(marge, bart), ad, 2).
true.

?- functor(ad(marge, bart), F, Ar).
F = ad,
Ar = 2.

?- functor(T, ad, 2).
T = ad(_G313, _G314).

?- functor(a,F,A).
F = a,
A = 0.
```

# 7.17 O PROLOG como linguagem de programação

**Predicados pré-definidos para a manipulação de termos(cont.):**

`arg(N, T_c, Arg)` afirma que
`Arg` é o `N`-ésimo argumento do termo composto `T_c`.

**Exemplo 7.17.5**

```
?- arg(1, ad(marge, bart), marge).
true.

?- arg(2, ad(marge, bart), Arg2).
Arg2 = bart.

?- arg(1, ad(X, Y), marge).
X = marge.
```

**Predicados pré-definidos para a manipulação de termos(cont.):**

`T_c =.. Lst` afirma que
o 1º elemento da lista `Lst` corresponde ao functor do termo composto `T_c`,
o resto da lista `Lst` corresponde aos argumentos do termo composto `T_c`.

**Exemplo 7.17.6**

```
?- T =.. [ad, marge, bart].
T = ad(marge, bart).

?- ad(marge, bart) =.. [P | R].
P = ad,
R = [marge, bart].
```

# 7.17 O PROLOG como linguagem de programação

**Predicado pré-definido:** `call/1`

`call(O)` tem sucesso apenas se o objectivo correspondente ao seu argumento tem sucesso.

Útil quando se criam literais recorrendo aos predicados
`functor`, `arg` e `=...`

Exemplo:

```
?- P = membro, O =.. [P,5,[1,2]], call(O).
false.
```

Permite a definição de predicados de ordem superior, isto é, predicados que têm outros predicados como argumentos.

# 7.17 O PROLOG como linguagem de programação

**Exemplo: predicado** `filtra/3`

`filtra(L1, Tst, L2)` significa que
a lista L2 é constituída pelos elementos da lista L1
que satisfazem o teste `Tst`.

**Exemplo: predicado** `filtro/3`

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

filtra([P | R], Tst, L) :-
     Lit =.. [Tst, P],
     call(Lit),
     !,
     L = [P | S],
     filtra(R, Tst, S).

filtra([_ | R], Tst, S) :-
      filtra(R, Tst, S).
```

# 7.17 O PROLOG como linguagem de programação

Predicados podem ser estáticos ou dinâmicos.

Um predicado *dinâmico* pode ser alterado durante a execução de um programa.

Todos os predicados pré-definidos são estáticos.

A indicação de que um predicado é dinâmico é feita através do comando `dynamic/2`:

`:- dynamic` ⟨nome do predicado⟩`/`⟨aridade⟩`.`

# 7.17 O PROLOG como linguagem de programação

**Predicados pré-definidos** `asserta/1`, `assertz/1` **e** `retract/1`.

Seja `C` uma cláusula e `pred` o predicado (dinâmico) da cabeça de `C`:

`asserta(C)` adiciona a cláusula `C` no início da definição de `pred`.

`assertz(C)` adiciona a cláusula `C` no fim da definição de `pred`.

`retract(C)` remove a cláusula `C` da definição de `pred`.

# 7.17 O PROLOG como linguagem de programação

**Exemplo**

Dado o programa

```
:- dynamic liga/2.
liga(a,b).
liga(b,c).
```

# 7.17 O PROLOG como linguagem de programação

Obtemos a seguinte interacção

```
?- assertz(liga(c,d)).
true.
?- listing(liga).
:- dynamic liga/2.
liga(a, b).
liga(b, c).
liga(c, d).
true.
?- retract(liga(b,c)).
true.
?- listing(liga).
:- dynamic liga/2.
liga(a, b).
liga(c, d).
true.
```

## Exemplo:

Consideremos a definição de número de Fibonacci:

$$fib(n) = \begin{cases} 0 & \text{se } n = 0 \\ 1 & \text{se } n = 1 \\ fib(n-1) + fib(n-2) & \text{se } n > 1 \end{cases}$$

Programa:

```
fib(0, 0) :- !.
fib(1, 1) :- !.
fib(N, Fib_N) :-
     N > 1,
     N_menos_1 is N - 1,
     N_menos_2 is N - 2,
     fib(N_menos_1,Fib_N_menos_1),
     fib(N_menos_2,Fib_N_menos_2),
     Fib_N is Fib_N_menos_1 + Fib_N_menos_2.
```

# 7.17 O PROLOG como linguagem de programação

Programa correcto, mas efectua muitos cálculos repetidos.

Podemos ir memorizando os valores calculados.

```
:- dynamic(fib_mem/2).

fib_mem(0, 0) :- !.

fib_mem(1, 1) :- !.

fib_mem(N, Fib_N) :-
     N > 1,
     N_menos_1 is N - 1,
     N_menos_2 is N - 2,
     fib_mem(N_menos_1,Fib_N_menos_1),
     fib_mem(N_menos_2,Fib_N_menos_2),
     Fib_N is Fib_N_menos_1 + Fib_N_menos_2,
     memoriza(fib_mem(N,Fib_N)).

memoriza(Lit) :- asserta(Lit :- !).
```

# 7.17 O PROLOG como linguagem de programação

Cortes nas afirmações adicionadas são necessários?

Sim, para evitar que um valor que já é conhecido volte a ser calculado.

```
?- listing(fib_mem).
:- dynamic fib_mem/2.

fib_mem(0, 0) :- !.
fib_mem(1, 1) :- !.
fib_mem(A, D) :-
        A>1,
        B is A+ -1,
        C is A+ -2,
        fib_mem(B, E),
        fib_mem(C, F),
        D is E+F,
        memoriza(fib_mem(A, D)).
```

# 7.17 O PROLOG como linguagem de programação

```
?- fib_mem(6,F).
F = 8.

?- listing(fib_mem).
:- dynamic fib_mem/2.

fib_mem(6, 8) :- !.
fib_mem(5, 5) :- !.
fib_mem(4, 3) :- !.
fib_mem(3, 2) :- !.
fib_mem(2, 1) :- !.
fib_mem(0, 0) :- !.
fib_mem(1, 1) :- !.
fib_mem(A, D) :-
        A>1,
        B is A+ -1,
        C is A+ -2,
        ...
```