Predicates as procedures, and arguments as i/o

```
% increment(+X,-Y)
increment(X,Y) :- Y is X+1.
```

Predicates as procedures, and arguments as i/o

```
% increment(+X,-Y)
increment(X,Y) :- Y is X+1.

% incr(-X,+Y)
incr(X,Y) :- X is Y-1.
```

Predicates as procedures, and arguments as i/o

```
% increment(+X,-Y)
increment(X,Y) :- Y is X+1.

% incr(-X,+Y)
incr(X,Y) :- X is Y-1.

% incr2(+X,+Y)
incr2(X,Y) :- X =:= Y-1.
```

Predicates as procedures, and arguments as i/o

```
% increment(+X,-Y)
increment(X,Y) :- Y is X+1.

% incr(-X,+Y)
incr(X,Y) :- X is Y-1.

% incr2(+X,+Y)
incr2(X,Y) :- X =:= Y-1.
```

From SWI Prolog documentation:

$-$*Number* is $+$*Expr*

True when *Number* is the value to which *Expr* evaluates.

Predicates as procedures, and arguments as i/o

```
% increment(+X,-Y)
increment(X,Y) :- Y is X+1.

% incr(-X,+Y)
incr(X,Y) :- X is Y-1.

% incr2(+X,+Y)
incr2(X,Y) :- X =:= Y-1.
```

From SWI Prolog documentation:

$-$*Number* is $+$*Expr*

True when *Number* is the value to which *Expr* evaluates.

$+$*Expr1* $=:=$ $+$*Expr2*

True if expression *Expr1* evaluates to a number equal to *Expr2*.

Mode indicators

+ input (known)
− output (unknown)

From SWI Prolog documentation

> An argument mode indicator gives information about the intended direction in which information carried by a predicate argument is supposed to flow. Mode indicators (and types) are not a formal part of the Prolog language but help in explaining intended semantics to the programmer. There is no complete agreement on argument mode indicators in the Prolog community.

Mode indicators

+ input (known)
− output (unknown)

From SWI Prolog documentation

> An argument mode indicator gives information about the intended direction in which information carried by a predicate argument is supposed to flow. Mode indicators (and types) are not a formal part of the Prolog language but help in explaining intended semantics to the programmer. There is no complete agreement on argument mode indicators in the Prolog community.

? uncommitted (don't care ≈ unknown)

```
% successor(?X,?Y)
successor(X,succ(X)).
```

Mode indicators

+ input (known)
− output (unknown)

From SWI Prolog documentation

> An argument mode indicator gives information about the intended direction in which information carried by a predicate argument is supposed to flow. Mode indicators (and types) are not a formal part of the Prolog language but help in explaining intended semantics to the programmer. There is no complete agreement on argument mode indicators in the Prolog community.

? uncommitted (don't care ≈ unknown)

```
% successor(?X,?Y)
successor(X,succ(X)).     % :- numeral(X).

% numeral(?X)
numeral(0).
numeral(succ(X)) :- numeral(X).
```

Reversibility with ?

?*Term1* = ?*Term2*

Unify *Term1* with *Term2*. True if the unification succeeds.

Reversibility with ?

?*Term1* = ?*Term2*

Unify *Term1* with *Term2*. True if the unification succeeds.

member(?Elem,?List)

True if *Elem* is a member of *List*.

Reversibility with ?

?*Term1* = ?*Term2*

Unify *Term1* with *Term2*. True if the unification succeeds.

member(?Elem,?List)

True if *Elem* is a member of *List*.

```
?- member(1,[1]).
true.
```

Reversibility with ?

?*Term1* = ?*Term2*

Unify *Term1* with *Term2*. True if the unification succeeds.

member(?Elem,?List)

True if *Elem* is a member of *List*.

```
?- member(1,[1]).
true.

?- member(X,[1]).
X = 1 .
```

Reversibility with ?

?*Term1* = ?*Term2*

Unify *Term1* with *Term2*. True if the unification succeeds.

member(?Elem,?List)

True if *Elem* is a member of *List*.

```
?- member(1,[1]).
true.

?- member(X,[1]).
X = 1 .

?- member(1,List).
List = [1|_] ;
List = [_,1|_] ;
...
```

Reversibility with ?

?*Term1* = ?*Term2*

Unify *Term1* with *Term2*. True if the unification succeeds.

member(?Elem,?List)

True if *Elem* is a member of *List*.

```
?- member(1,[1]).
true.

?- member(X,[1]).
X = 1 .

?- member(1,List).
List = [1|_] ;
List = [_,1|_] ;
...

?- member(X,List).
List = [X|_] ;
List = [_,X|_] ;
...
```

Two more mode indicators

@ argument will *not* be further instantiated

@*Term1* == @*Term2*
: True if *Term1* is equivalent to *Term2*.

var(@*Term*)
: True if *Term* currently is a free variable.

Two more mode indicators

@ argument will *not* be further instantiated

@*Term1* == @*Term2*
: True if *Term1* is equivalent to *Term2*.

var(@*Term*)
: True if *Term* currently is a free variable.

: meta-argument that can be called as goal

\+ :*Goal*
: True if *Goal* cannot be proven

call(:*Goal1*)
: Call *Goal*.

On swipl

```
if(A,B,C) :- (A,!,B) ; C.

neg(A) :- if(A,fail,true).
```

On swipl

```
if(A,B,C) :- (A,!,B) ; C.

neg(A) :- if(A,fail,true).

?- listing(if).
if(A, B, C) :-
        ( call(A), !, call(B) ; call(C) ).
true.
```

On swipl

```
if(A,B,C) :- (A,!,B) ; C.

neg(A) :- if(A,fail,true).


?- listing(if).
if(A, B, C) :-
        ( call(A), !, call(B) ; call(C) ).
true.

?- if(0=0,X=1,X=2).
X = 1.

?- if(0=1,X=1,X=2).
X = 2.
```

On swipl

```
if(A,B,C) :- (A,!,B) ; C.

neg(A) :- if(A,fail,true).


?- listing(if).
if(A, B, C) :-
        ( call(A), !, call(B) ; call(C) ).
true.

?- if(0=0,X=1,X=2).
X = 1.

?- if(0=1,X=1,X=2).
X = 2.

?- 0=0 -> X=0.
X = 0.

?- 0=1 -> X=1.
false.
```