Mode indicator	Description	Information flows into predicate?	Information flows out of predicate (via further instantiation)?	Examples
++	At call time, the argument must be **ground:** the argument must be a fully instantiated term with no unbound variables. This mode indicates that the argument "**provides input to the predicate**". Indeed, being ground, the argument cannot be further instantiated.	exclusively	no	
+	At call time, the argument must be **instantiated:** the argument can be any term (possibly ground) but must not be an unbound variable. For example, the term `[_]` is instantiated and is a list, although the single list member is the anonymous variable, which is always unbound (and thus nonground). Generally the term is expected to satisfy some (informal) type specification. This mode indicates that the argument is "**provides input to the predicate**": although the argument might be nonground and thus could be further instantiated, the predicate promises to leave it as it is. On the other hand, sharing of unbound variables with between a "mode +" argument and an output argument might render that promise untenable, as in `max_member(X,[1,X,1,1,1,1])`.	yes	may (but should not)	`foldl(:Goal, +List, +V0, -V)`

?	At call time, the argument is **unconstrained:** the argument can be any term. It could be instantiated (ground or not) or uninstantiated (an unbound variable). This is also called "being a partial term".
	
	Generally the term is expected to satisfy some (informal) type specification.
	
	This mode indicates that the argument is used to **provide input to the predicate or accept output from the predicate or even both**. For example, in `length(?List,?Length)`, the `List` argument may provide input and the `Length` argument output, or the converse. Or both may be used for input if the predicate is used to verify the length of a list.
		yes	yes	`reverse(?List1, ?List2)`
				
				`length(?List, ?Length)`

-	At call time, the argument is **unconstrained**, as for mode '?': the argument can be a partial term.
	
	Unlike for mode '?', the argument should be considered as only **accepting output from the predicate**. It would generally be an uninstantiated variable.
	
	If the argument is instantiated at call time, the predicate behaves as if it had been called with that argument uninstantiated, and the predicate call were followed by unification of the instantiated (input) argument and the computed (output) argument, possibly causing failure. Sometimes the this is what the implementation actually does.
	
	The predicate thus exhibits **steadfast behaviour** (**steadfastness**): instantiation of an output argument at call-time does not change the predicate semantics (although optimizations performed based on the additional information available may cause changes in side-effects).
	
	Calling a predicate with an instantiated output argument is good style. For example, there is nothing wrong with `findall(X, Goal, [T])`. It is equivalent to `findall(X, Goal, Xs), Xs = [T]`. In effect, the first call restricts the output domain early.
	
	Note that any determinism specification, e.g., **det** ("succeed exactly once, leave no choicepoint") only applies if the relevant argument is indeed uninstantiated. For the case where the argument is instantiated or involved in constraints and thus more information is available and exploitable by the predicate, **det** effectively becomes **semidet** ("succeeds at most once, leaves no choicepoint"). Similarly, **multi** ("succeed at least once, but otherwise the succeed/fail behaviour is left unspecified") effectively becomes **nondet** ("the succeed/fail behaviour is left unspecified")
		may
		(to restrict output domain)
			yes	`max_member(-Max, +List)`
				
				`max_list(+List,-Max)`

--	At call time, the argument must be **uninstantiated** (an unbound variable). This is typically used by predicates that create 'something' and return a handle to the created object, such as open/3, which creates a stream.	no	exclusively	`open(+SrcDest, +Mode, --Stream)`
:	Argument is a meta-argument, for example a term that can be called as goal. The predicate is thus a meta-predicate. This flag implies +.	yes	may (but should not)	`foldl(:Goal, +List, +V0, -V)`
@	Argument will not be further instantiated than it is at call-time: any variable in the passed argument stays a variable. Typically used for type tests where unbound variables become themselves the objects of analysis.	yes	no	`var(@Term)`
!	Argument contains a mutable structure that may be modified using setarg/3 or nb_setarg/3.	mutated in place		