Programming Languages

Prolog

CSCI-GA.2110-001
Spring 2023

Prolog overview

- Stands for **Pro**gramming in **Log**ic.
- Invented in approximately 1972.
- Belongs to the logical & declarative paradigms.
- Based on first order predicate calculus.
- Used for artificial intelligence, theorem proving, expert systems, and natural language processing.
- Used as a standalone language or complements traditional languages.
- Radically different than most other languages.
- Each program consists of 2 components:
  - database (*program*): contains facts and rules
  - query : ask questions about relations

Propositional Logic

- Basis of predicate logic.
- Sentences have truth values.
- Sentences fragments are represented by Boolean variables, P, Q, etc.
- Example: P might mean “it is sunny out.”
- Logical connectives are used: $P \wedge Q$, $P \vee Q$, $\neg P$, etc.
- Inference rules (e.g. $P \rightarrow Q$) expressed using logical connectives.
- Logical formulas over the variables and connectives yield truth values.

First Order Predicate Logic

- Used by Prolog
- Establishes facts about and relationships between items in a universe.
- Relationships are expressed using predicates.
- Predicate `happy(john)` means "`john` is happy."
- Predicate `likes(john,mary)` means "`john` likes `mary`."
- Quantification allows us to reason more generally: $\forall$, $\exists$

Examples:

$\forall x : \mathtt{loves}(\mathtt{x}, \mathtt{mary})$ means "everyone loves mary."

$\exists x : \mathtt{loves}(\mathtt{x}, \mathtt{mary})$ means "someone loves mary."

$\forall x \exists y : \mathtt{loves}(\mathtt{x}, \mathtt{y})$ means "everyone loves someone."

$\exists y \forall x : \mathtt{loves}(\mathtt{x}, \mathtt{y})$ means "someone is loved by everyone."

$\exists x \exists y : \mathtt{loves}(\mathtt{x}, \mathtt{y})$ means "someone loves someone."

Stating Facts

Two ways to state facts:

```
?- [user].                                  consult user
sunny.                                      state the fact
% user://1 compiled 0.00 sec, 408 bytes
true.
```

(same as `?- consult(user).`)

Or:

```
?- assert(sunny).   state the fact
true.
```

Stating Facts 2

What facts can we describe?

1. Items `?- assert(sunny).`
2. Relationships between atoms:
   `?- assert(likes(john,mary)).`

Query the database:

```
?- likes(john,mary).
true.
?- likes(mary,john).
false.
?- likes(john,sue).
false.
```

Prolog Terminology

- Functors : an atom (defined below) with arity. e.g., `likes/2`
- Arguments can be legal Prolog *terms* : integer, atom, variable, structure.
- Atoms: lowercase characters, digits, underscore (not first), graphic characters (e.g. #,&,@) or *anything* in quotes.
  - Legal: hello, hi123, two_words, @pl, "G_1)!#)@blah"
  - Illegal: Hello, 123hi, _hello, two-words
- Variables: Any word beginning with a capital letter.
- Structures: Atoms with a list of arguments. e.g., `likes(john,mary)`

Structures are also known as *relations*, *compound terms*, and *predicates*.

Like functional languages, variables bind to values (not memory locations). Unlike functional languages, there is no clear notion of input and output.

```
?- likes(john,Who).
Who = mary
```

Prolog will display one *instantiation* at a time. Type a semicolon for more.

More Relations

All satisfying `likes` relations:

```
?- likes(Who1,Who2).
```

Who1 = john; Who2 = mary

Constrain queries using variables:

```
?- likes(Who,Who).
```

false.

(People who like themselves.)

Use wild card to determine if some instantiation exists:

```
?- likes(john,_).
```

true.

(That is, john likes *someone*—we don't care who.)

Wild cards can be used in conjunction with variables:

```
?- likes(Who,_).
```

Who = john

Rules

Rules express conditional statements about our world.
Consider the assertion: "All men are mortal."

Expressible as modus ponens: $\texttt{human} \rightarrow \texttt{mortal}$ ("human implies mortal.")

`mortal` is a *goal* (or *head*), and `human` is a *subgoal* (or *body*).

In Prolog, we write it in the following form:
$\texttt{mortal} \leftarrow \texttt{human.}$

Or more generally,
$\texttt{goal} \leftarrow \texttt{subgoal.}$

There can be multiple subgoals. Example:
$\texttt{goal} \leftarrow \texttt{subgoal}_1, \ldots, \texttt{subgoal}_n\texttt{.}$

This form is called a *Horn clause*.

Rules Example

```
?- assert(mortal(X) :- human(X)).
true.
?- assert(human(socrates)).
true.
```

Now we query:

```
?- mortal(socrates).
true.
```

You can also ask who is mortal:

```
?- mortal(X).
X = socrates
```

Closed World Assumption

Prolog relies on everything it is told being true: both facts and rules.

e.g., if you tell Prolog the sky is green, it won't argue with you.

```
?- assert(sky_color(green)).
true.
```

This is called a *closed world assumption*.

Conjunction and Disjunction

Conjunction is expressed using commas:

```
?- fun(X) :- red(X), car(X).
```

A red car.

Disjunction is expressed with semicolons or separate clauses:

```
?- fun(X) :- red(X); car(X).
```

Something red or a car.

. . . is the same as

```
?- fun(X) :- red(X).
?- fun(X) :- car(X).
```

Order of rules matters!

Subgoal `red(X)` will be attempted first, then `car(X)`.

Multi-Variable Rules

```
daughter(X,Y) :- parent(Y,X), female(X).

grandfather(X,Y) :- male(X), parent(X,Z), parent(Z,Y).
```

Quantification:

- Variables appearing in the goal are *universally* quantified.
- Variables appearing only in the subgoal are *existentially* quantified.

The `grandfather` goal reads as:

$$\forall_{\mathtt{X},\mathtt{Y}} \exists_{\mathtt{Z}} : \mathtt{grandfather}(\mathtt{X}, \mathtt{Y}) \leftarrow \mathtt{male}(\mathtt{X}), \mathtt{parent}(\mathtt{X}, \mathtt{Z}), \mathtt{parent}(\mathtt{Z}, \mathtt{Y}).$$

Unification

Prolog variables take on values by means of **unification**. Unification is a binding between any of the following:

- A variable and a value.
- A variable and another variable.
- A value and a value.

Examples of unification:

- (Prolog) The rule `happy(X) :- go_walking(X)` and query `happy(joe)` (`X = joe`).
  Result: `happy(joe) :- go_walking(joe)`
- (ML) Function signature `'a -> 'b -> int -> 'a` and input pattern `int -> real -> 'c -> 'c`.
  Result: `int -> real -> int -> int`

Unification generally happens "behind-the-scenes," although one may explicitly unify variables in Prolog also.

Explicit Unification

```
?- a=a.
true.
?- a=b.
false.
?- foo(a,b) = foo(a,b).
true.
?- foo(a,X) = foo(a,b).
X=b.
?- X=a.
X=a.
?- A=B.
A=B.
?- A=B, A=a, B=Y.
A=a; B=a; Y=a.
```

Unification Algorithm

1. Constants: any constant unifies with itself.
2. Structures: same functor, same arity, arguments unify recursively.
3. Variables: unify with anything.
   (a) Value: variable takes on the value.
   (b) Another Variable: unify by reference.

Some examples:

Operand 1	Operand 2	Result
`21`	`21`	`21`
`8`	`15`	error
`X`	`5`	`X=5`
`X`	`Y`	`X=Y`
`love(X,me)`	`love(you,Y)`	`X=you,Y=me`
`love(X,Y)`	`love(you,Y)`	`X=you,Y=Y`
`love(X,Y)`	`foobar(you,Y)`	error
`c(X,c(Y,c(Z,n)))`	`c(he, c(she, c(it,n)))`	`X=he, Y=she, Z=it`
`love(X,Y)`	`love(you,f(Y))`	`X=you,Y=??`

Occurs Check

Consider:

`equal(Y, f(Y)).`

Let's try unifying `Y=f(Y)`. We have:

`equal(Y, f(Y))`	no match
`equal(f(Y), f(f(Y)))`	no match
`equal(f(f(Y)), f(f(f(Y))))`	no match
`equal(f(f(f(Y))), f(f(f(f(Y)))))`	no match

Infinite recursion!

When two expressions are unified and a variable X in one expression "occurs" in the other expression, unsound logic may result.

- Prolog will **not** report any warning or error by default.
- This situation can be caught with an *occurs check*.
- The occurs check is not run by default, for performance reasons.

More on Occurs Check

When attempting to unify variable v and structure s, an *occurs check* determines whether v is contained within s. If so, unification fails.

- Prevents infinite loops or unsoundness.
- Inefficient to implement (linear in the size of the largest term).
- Most implementations of Prolog (like SWI Prolog) omit it.

Therefore, in SWI Prolog:

```
?- equal(Y, f(Y)).
Y = f(Y).
```

If you insist on the occurs check, you can force it in SWI:

```
?- unify_with_occurs_check(X,f(X)).
false.
```

Execution Order

There are two ways to answer a query:

1. *Forward chaining*: start with given facts and work forward to find a goal.
2. *Backward chaining*: start with goal and work backward toward the facts. (Used by Prolog).

If the body of a rule unifies with the heads of other rules in some particular order, it can be expressed as a tree.

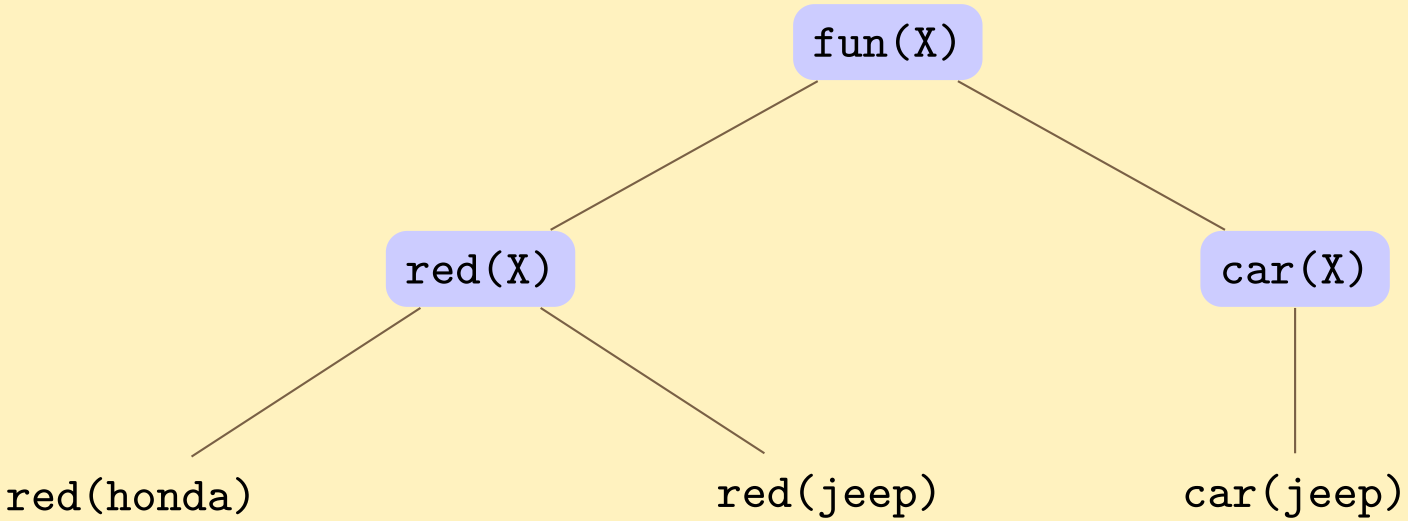
- Forward chaining: most suitable for: many rules, few facts.
  - Few facts reduces the possible fact combinations, yielding fewer rules to test
- Backward chaining: most suitable for: few rules, many facts.
  - Fewer rules mean fewer possible trees, yielding fewer facts to test

Execution Tree

Consider:

```
fun(X) :- red(X).    red(jeep).
fun(X) :- car(X).    car(jeep).
red(honda).          ?- fun(X).
```

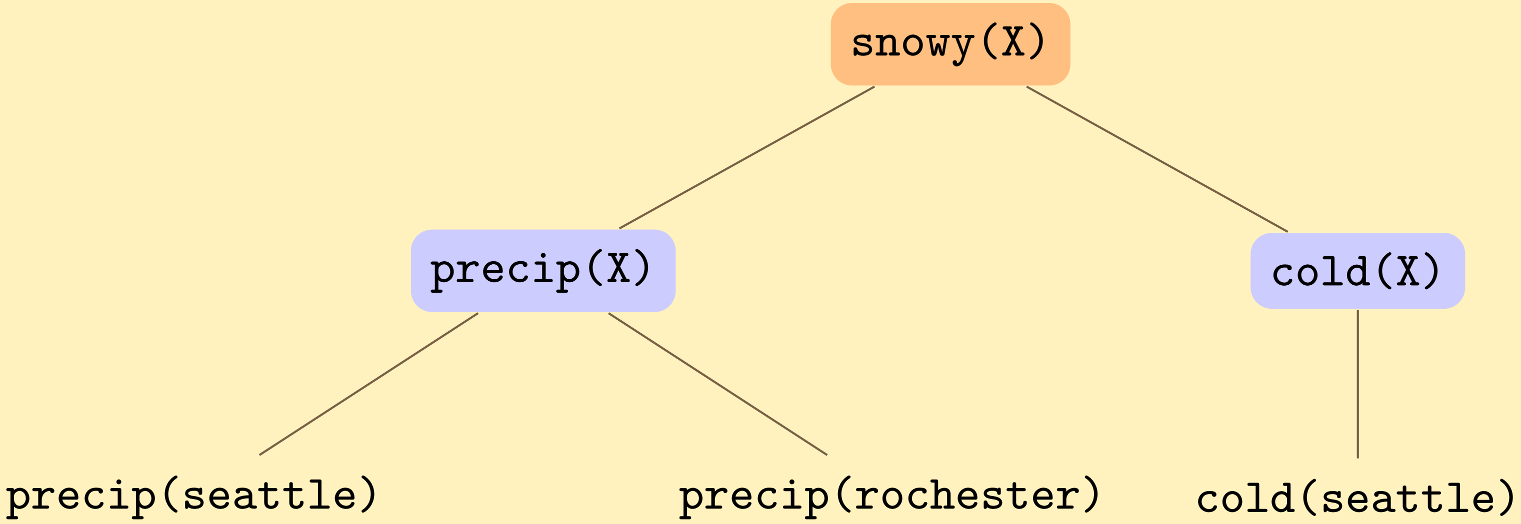
This produces an AND/OR tree:



Execution Order

Consider:

```
precip(seattle).
precip(rochester).
cold(seattle).
snowy(X) :- precip(X), cold(X).
?- snowy(X).
```



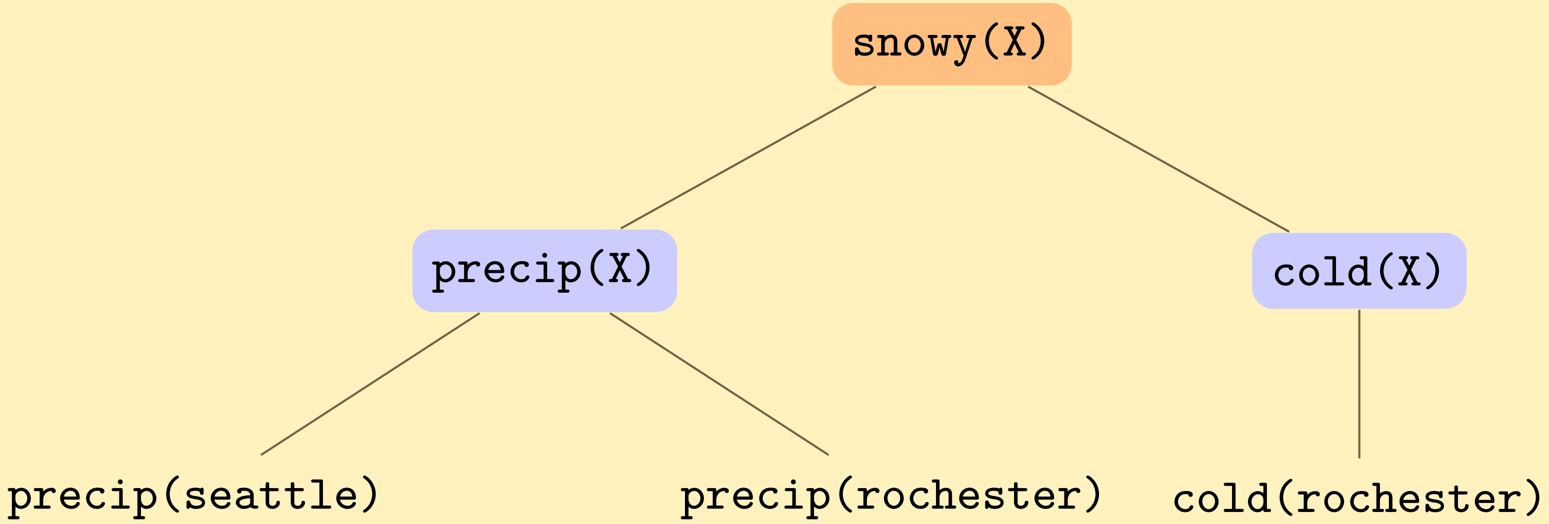
Backtracking

- Prolog maintains a list of goals to be satisfied.
- When a goal is queried, all *subgoals* of the goal are added to the list.
  - `goal(X,Y) :- subgoal1(X), subgoal2(Y).`
- Prolog will try to satisfy *all* subgoals.
- If a subgoal cannot be satisfied, Prolog will try another way.
- This is called *backtracking*.

Backtracking Example

Consider:

```
precip(seattle).
precip(rochester).
cold(rochester).
snowy(X) :- precip(X), cold(X).
?- snowy(X).
```



Backtracking in Prolog

```
?- precip(seattle).   ?- precip(rochester).
?- cold(rochester).   ?- snowy(X) :- precip(X), cold(X).
```

Print the backtrace by invoking `trace.`, then `snowy(X).`

```
Call: (6) snowy(_G466) ? creep
Call: (7) precip(_G466) ? creep
Exit: (7) precip(seattle) ? creep
Call: (7) cold(seattle) ? creep
Fail: (7) cold(seattle) ? creep
Redo: (7) precip(_G466) ? creep
Exit: (7) precip(rochester) ? creep
Call: (7) cold(rochester) ? creep
Exit: (7) cold(rochester) ? creep
Exit: (6) snowy(rochester) ? creep
X = rochester
```

Reflexive Transitive Closure

More than one "application" of a rule:

```
connect(Node,Node).
connect(N1,N2) :- edge(N1,Link), connect(Link,N2).
```

Now add some edges:

```
?- assert(edge(a,b)).   ?- assert(edge(c,d)).
?- assert(edge(a,c)).   ?- assert(edge(d,e)).
?- assert(edge(b,d)).   ?- assert(edge(f,g)).
```

```
?- connect(a,e).
true.
```

```
connect(a,e) :- edge(a,b), connect(b,e)
connect(b,e) :- edge(b,d), connect(d,e)
connect(d,e) :- edge(d,e), connect(e,e)
```

```
?- connect(d,f).
false.
```

Lists

Lists are denoted by `[ a, b, c ]`.

A *cons pair* is denoted `[X|Y]` where `X` is the *head* and `Y` is the *tail*.

Rules for testing list membership:

```
?- assert(member(X, [X|Xs])).
?- assert(member(X, [Y|Ys]) :- member(X,Ys)).
```

Testing membership:

```
?- member(b,[a,b,c]).
true.
?- member(b,[a,c]).
false.
```

You can also extract list membership:

```
?- member(X,[a,b,c]).
X = a; X = b; X = c.
```

Reversing Lists

Consider a list reverse rule:

```
reverse([],[]).
reverse([X|Xs],Zs) :- reverse(Xs,Ys), append(Ys,[X],Zs).
```

Reverse-accumulate:

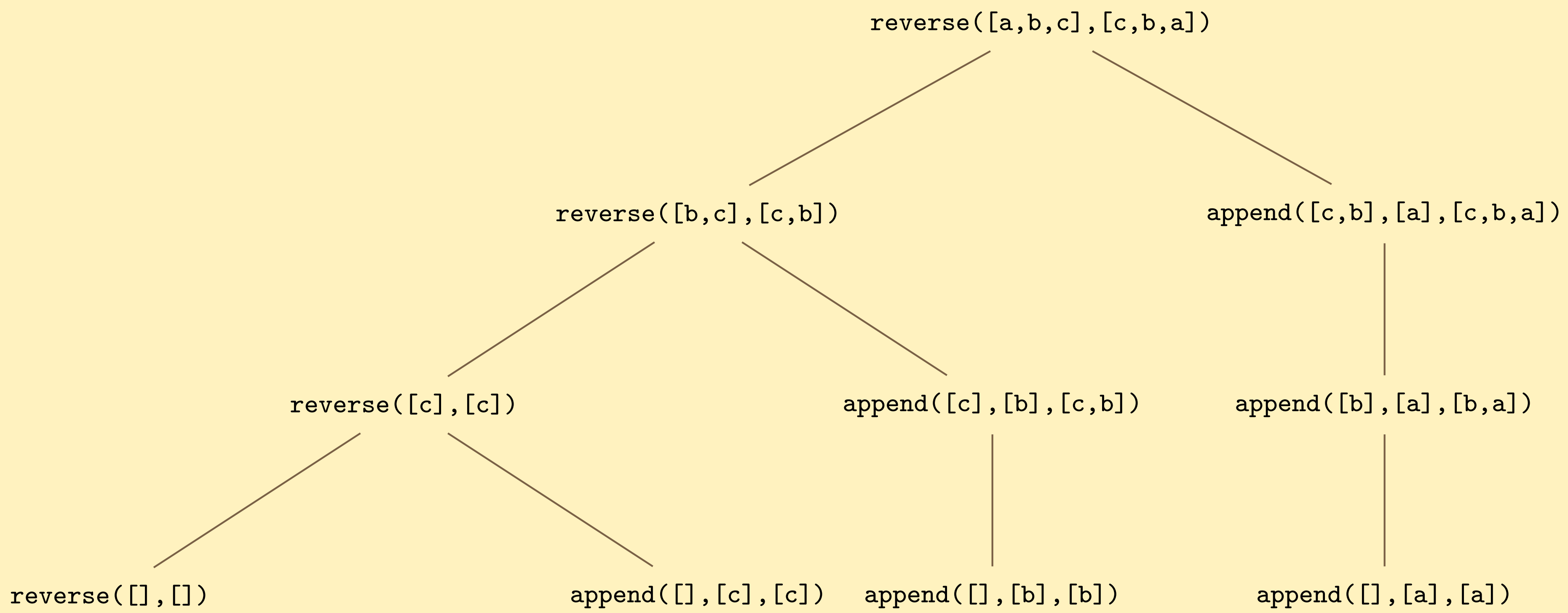
```
reverse(Xs,Ys) :- reverse(Xs,[],Ys).
reverse([X|Xs],Acc,Ys) :- reverse(Xs,[X|Acc],Ys).
reverse([],Ys,Ys).
```

Invoking the reverse rule:

```
?- reverse([a,b,c], X).
X = [c, b, a].
?- reverse([a,b,c], [a,c,b]).
false.
```

Tree for Reverse

The `reverse` rule at work:



Cut Operator

You can tell Prolog to stop backtracking using the *cut* operator, `!`.

- Used to "commit" all unifications up to the point of the `!`
- Prevents backtracking through any subgoal to the left of `!`. Bindings remain in place.
- Works across multiple rules (same goal).

Consider:

```
max(A,B,B) :- A < B.
max(A,B,A).
?- max(3,4,M).
```

M = 4; M = 3 (not correct)

Try instead:

```
max(A,B,B) :- A < B,!.
max(A,B,A).
?- max(3,4,M).
```

M = 4

More on Cut

The cut operator can also serve as an if-then-else construct:

```
statement :- condition, !, then_part.
statement :- else_part.
```

- Cut prevents the condition from being retested.
- If `condition` is true, subgoal `then_part` will be attempted.
- If `then_part` fails, the system will not backtrack into the condition.
- Second rule will not be called if `then_part` is called and fails.
- If first goal fails (meaning the condition failed), `else_part` will be called.

Negation

Definition of `not` (also known as `\+`):

```
not(Goal) :- call(Goal), !, fail.
not(Goal).
```

- Predicate `fail` unconditionally fails (known as the cut-fail pattern).
- Predicate `call` treats the input term as a goal and attempts to satisfy it.

Example:

```
female(Person) :- \+ male(Person).
```

Note: A true $\backslash +$ expression generally indicates *inability to prove*—**not** falsehood. (It could be that a particular `Person` is male, but no such fact was ever asserted.)

For the semantics of the `not` to mean *falsehood*, the universe of facts must be *complete* (everything that is true has been asserted accordingly.)