Negation

Negation in Prolog

Prolog does not have a connective for *classical negation*.

It has a special operator \+ read as

"fail (finitely) to prove"

The operational meaning of \+ is

\+ P succeeds iff P fails finitely

\+ P fails iff P succeeds

Negation in Prolog

In Prolog's negation \+ is allowed only in queries and in the bodies of rules (<u>not in the heads of rules!</u>).

EXAMPLE

Program:
```
happy(X) :-
    owns_a_house(X),
    \+ has_mortgage(X).
```

Query:
```
?- owns_a_house(X), \+ has_mortgage(X).
```

see: pt3_s3.pl

Negation As Failure (NAF) Rule

- \+Q is proved if all evaluation paths for the query Q end in failure.
- Proof of \+Q will not generate any bindings for variables in Q.
- If Q contains variables $X_1,..,X_k$ at the time it is evaluated, it behaves like:

$$\neg \exists X_1 \ldots \exists X_k \, Q$$

Connected to the '*Closed World Assumption*' --- anything that is not *known* to be true is assumed to be *false*.

Negation As Failure (NAF) Rule: Example

Program:

```
student(john).
student(mary).

gets_grant(john).
```

Query:

```
?- student(X), \+ gets_grant(X).
```

Answer:

```
X = mary
```

mary is a student and Prolog cannot *prove* that mary gets a grant.

see: pt3_s5.pl

Negation As Failure (NAF) Rule: Example 2

Program:

```
dragon(puff).
dragon(macy).
dragon(timothy).

magic(puff).
vegetarian(macy).

lives_forever(X)  :-  magic(X).
lives_forever(X)  :-  vegetarian(X).
```

Query:

```
?-  dragon(X), \+ lives_forever(X).
```

EXERCISE

Construct the Prolog evaluation to see how it finds the answers.

see: pt3_s6.pl

Negation As Failure (NAF) Rule: Example 3

Assuming a set of male/1 and parent/2 facts.

Who are the males with no children:

```
?- male(P), \+ parent(P,_).
```

P is a male who has no sons:

```
no_sons(P)  :-  male(P),
                \+ (parent(P,C),  male(C)).
```

P is a male who has no daughters

```
no_daughters(P)  :-  male(P),
                     \+ (parent(P,C), \+ male(C)).
```

see: pt3_s7.pl

NAF with unbounded conditions

Variables in negative conditions can give the wrong answers

```
?-  dragon(X), \+ lives_forever(X).
```

has answers. But

```
?-  \+ lives_forever(X), dragon(X).
```

has no answers. Why?

Apply the NAF inference rule to first condition: what is the result?

Some Prologs (not Sicstus) require \+ P to be ground at the instant it is selected for evaluation.

see: pt3_s6.pl

NAF with unbounded conditions: Example

```
person(bob).             likes(bob, frank).
person(chris).
person(frank).

sad(X)  :-
   person(X),
   person(Y),
   X \= Y,
   \+  likes(Y, X).
```

"X is sad if someone else doesn't like X"

see: pt3_s9.pl

In the example, bob, chris and frank are sad.

NAF with unbounded conditions: Example 2

```
person(bob).            likes(bob, frank).
person(chris).
person(frank).

very_sad(X)  :-
   person(X),
   \+ ( person(Y),
        X \= Y,
        likes(Y, X)
      ).
```

"X is very sad if no one else likes X"

see: pt3_s9.pl

In the example, bob and chris are very sad.

NAF and classical negation

\+ is not the same as classical negation ¬.

EXAMPLE

p ← ¬ p classically implies p

but

p :- \+ p cannot solve ?- p
(it has to fail *finitely*)

So p is a *logical* consequence in the first case, but it is not a *computable* consequence in the second.

NAF and classical negation: Example

Every person X is happy if all friends of X like logic.

In classical logic we could write

$$happy(X) \leftarrow person(X) \wedge \\ \forall Y \; (friend(X,Y) \rightarrow likes(Y, logic))$$

or equivalently

$$happy(X) \leftarrow person(X) \wedge \\ \neg \exists Y \; (friend(X,Y) \wedge \neg \; likes(Y,logic))$$

NAF and classical negation: Example

$$happy(X) \leftarrow person(X) \wedge \neg \exists Y\, (friend(X,Y) \wedge \neg\, likes(Y, logic))$$

In Prolog we can write:

```
happy(X)  :-
    person(X),
    \+ ( friend(X, Y), \+ likes(Y, logic) ).
```

see: pt3_s9.pl

Controlling Search

Controlling search: Example

```
send(Cust, Balance, Message)  :-
      Balance =< 0,
      warning(Cust, Message).

send(Cust, Balance, Message)  :-
      Balance > 0,
      Balance=< 50000,
        credit_card_info(Cust, Message).

send(Cust, Balance, Message)  :-
      Balance > 50000,
      investment_offer(Cust, Message).
```

needless search

see: pt3_s15.pl

Controlling search: Example

For a condition/call:

send(frank, -10, Message)

in a query for which all solutions are being sought, Prolog will try to use second and third clause after an answer has been found using the first clause.

Clearly this search is pointless.

Controlling search: The ‘cut’ primitive

- Cut, denoted by ‘!’, is a Prolog evaluation control primitive.
- It is "extra-logical": it is used to control the search for solutions and prune the search space.
- It always succeeds, but cannot be backtracked past. It is used to prevent unwanted backtracking.
- The cut can only be understood procedurally, in contrast to the declarative style that logic programming encourages.
- But used carefully, it can significantly improve efficiency without compromising clarity too much.

The 'cut' primitive: Example

```
send(Cust, Balance, Message)  :-
        Balance =< 0, !,
        warning(Cust, Message).

send(Cust, Balance, Message)  :-
        Balance=< 50000, !,
        credit_card_info(Cust, Message).

send(Cust, Balance, Message)  :-
       investment_offer(Cust, Message).
```

see: pt3_s18.pl

Effect of 'cut' primitive

p(...) :- T_1, ..., T_k, !, B_1, ..., B_n.
p(...) :- ...
p(...) :- ...

In trying to solve a call:

p(...)

if the first clause is applicable, and T_1, ..., T_k all succeed, then on *backtracking*:

- *do not try* to find an alternative solution for T_1, ..., T_k and
- *do not try* to use any other clauses for the call p(...).

Effect of ‘cut’ primitive: Example

Program:

```
comment(X)  :-  number(X), !, write(yes).
comment(X)  :-  write(no).
```

This program tests a term X and prints a comment.

The intention is that if X is a number then the comment is yes, and otherwise is no.

Will this program work correctly (assuming X is ground)?

see: pt3_s20.pl

Effect of ‘cut’ primitive: Example 2

Define least(X, Y, M) to mean “M is the least of X and Y”

Program:
```
least(X, Y, X)  :-  X < Y, !.
least(X, Y, Y).
```

Query: ?- least(1, 2, M)   correctly succeeds, with M = 1
Query: ?- least(2, 1, M)   correctly succeeds, with M = 1

BUT …

Query: ?- least(1, 2, 2)   wrongly succeeds

see: pt3_s21.pl

EXERCISE: Fix this.

Cut in built-in predicates: Length/2

Recall length(L, N) means "the length of list L is is N".

?- length(L, 2)

fails if we ask for a *second* solution with L unbound.

But evaluation of len(L, 2) where:

```
len([ ], 0).
len([_|L], N)  :-  len(L, M), N is M+1.
```

goes into an infinite loop if we ask for a *second* solution.

see: pt3_s22.pl

Why the difference?

Cut in built-in predicates: Length/2

Sicstus length/2 has a definition that uses !.

That definition is equivalent to:

```
length(L, N) :-
        number(N),
        len(L ,N), !.
length(L, N) :-
        len(L, N).
```

The cut ! in the first clause prevents Prolog from backtracking to try to find more solutions to len/2 call and prevents use of the second clause.

see: pt3_s22.pl

Cut in built-in predicates: NAF/1

```
\+(P) :- P, !, fail.
\+(_).
```

fail is a Prolog primitive that always fails.

Controlling search: Prolog conditional

Related to the !, is the Prolog conditional test:

(Test -> P ; Q)

Each of Test, P, Q can be a general Prolog query.

If Test succeeds then evaluate P else evaluate Q --- but don't backtrack for more solutions to Test if P fails.

Prolog conditional: Example

```
student_fees(S, F)  :-
      student(S),
      (eu(S)  ->   F=3000  ;  F=19000 ).
```

Equivalent to:

```
student_fees(S, F)  :-
      student(S),
      fees_for(S, F).

fees_for(S, F) :-
      eu(S), !, F = 3000.
fees_for(S, F)  :-  F=19000.
```

see: pt3_s26.pl

Prolog conditional: Example 2

```
send(Cust, Balance, Message)  :-
         (
        Balance =< 0
        ->
             warning(Cust, Message)
        ;
         Balance =< 50000
        ->
            credit_card_info(Cust, Message)
        ;
            % otherwise
             investment_offer(Cust, Message)
        ).
```

see: pt3_s27.pl

Prolog conditional: Example 3

We want to print out all the friends of X.

```
print_friends(X) :-
   write( ‘The friends of ‘), write(X), write( ‘:’ ), nl,
   friend(X, Y),
   write(‘ ‘), write(Y),
   nl,
   fail.

print_friends(_) :-
   write( ‘Done’ ),
   nl.
```

see: pt3_s28.pl

Prolog conditional: Example 3

```
print_all_friends(X)  :-
    person(X),
    friend(X, _), !,
    print_friends(X).    % as above

print_all_friends(X)  :-
    write(X),
    (person(X)
      ->  write( ' has no friends!' )
      ;     write( ' is not a person!' )
    ),
    nl.
```

see: pt3_s28.pl

Aggregation

Aggregation: findall/3 primitive

Often we want to collect into a single list all those items satisfying some property.

Prolog supplies a convenient primitive for this:

```
findall(Term, Goal, List)
```

List is the list of solutions. *It may contain duplicates*.

findall/3: Example

Program: likes(frank, chris).

likes(chris, bob).

likes(chris, frank).

To find all those whom chris likes:

Query: ?- findall(X, likes(chris, X), L).

Answer: L = [bob, frank]

Another example:

Query: ?- findall(X, likes(X, _), L).

Answer: L = [frank, chris, chris]

see: pt3_s32.pl

findall/3: Examples 2

To find all sublists of [a, b, c] having length 2:

Query: ?- findall([X, Y], sublist([X, Y], [a, b, c]), S).

Answer: S = [[b, c], [a, c], [a, b]]

see: pt3_s33.pl

findall/3: More Examples

Query: ?- findall(X-Y, append(X, Y, [a, b, c]), S).

Answer: S = [[]-[a,b,c], [a]-[b,c], [a,b]-[c], [a,b,c]-[]]

Query: ?- findall(p(X, [X], X) , member(X, [a, b, c]), S).

Answer: S = [p(a,[a],a), p(b,[b],b), p(c,[c],c)]

Query: ?- findall(g , member(X, [a, b, c]), S).

Answer: S = [g, g, g]

findall/3: More Examples

The list of children of a mother M and a father F:

```
children_of(M, F, Children) :-
    findall( C,  ( mother_of(M, C), father_of(F, C ) ), Children ).
```

see: pt3_s35a.pl

A list L of pairs (X, F) where X is a person and F is a list of all the friends of X:

```
friend_list(L) :-
   findall( (X, F), ( person(X), findall(Y, friend(X, Y), F) ), L ).
```

see: pt3_s35b.pl

(So in the latter we have a findall inside a findall)

findall/3: More Examples

A list L of pairs (X, N) where X is a person and N is the number of friends of X:

```
friend_number_list(L) :-
 findall( (X, N),
         ( person(X),
          findall(Y, friend(X, Y), F),
          remove_duplicates(F, Fx),
          length(Fx, N) ),
         L ).
```

see: pt3_s35b.pl

Aggregation: setof/3 primitive

setof(Term, Goal, List)

This is more powerful than findall/3.

It removes duplicates.

It also automatically orders the answer list using the predefined term ordering (=<) -- the normal numeric ordering for numbers and lexical ordering for constants.

There are also some important differences concerning variables in Goal.

setof/3: Example

Program: admires(jane, peter). admires(jane, amy).
admires(jane, bill). admires(kate, john).
admires(kate, mary).

Query: ?- findall(X, admires(M, X), L).

Answer: X = [peter, amy, bill, john, mary]

Here M is existentially quantified. Equivalent to:

Query: ?- findall(X, admires(_, X), L).

BUT

Query: ?- setof(X, admires(M, X), L).

Answer: M = jane, L = [amy, bill, peter] ;

Answer: M = kate, L = [john, mary]

see: pt3_s38.pl

setof/3: Exercise

Compare

Query: ?- setof(X, admires(M, X), L).

Answer: M = jane, L = [amy, bill, peter] ;

Answer: M = kate, L = [john, mary]

Query: ?- setof(X, admires(_, X), L).

Answer: L = [amy, bill, peter] ;

Answer: L = [john, mary]

Query: ?- setof(X, M^admires(M, X), L).

Answer: L = [amy, bill, john, mary, peter] ;

no

see: pt3_s38.pl

(like findall/3 but sorted)