Examples

1. The following program shows that constructive negation can finitely fails whereas the "delay technique" produces infinite computation

```
p(a).                                          % File delay.pnt in programs.zip
p(f(X)):-     p(X).
r(X):-  ! s(X).
s(g(X)).
```

Both questions : `!r(X),p(X).` and `p(X),!r(X).` produce an immediate fail in the BCN prototype. However, the "the delay technique" checks if each answer for `p(X)` is a finite failure of `r(X)` (which, indeed, is the case), producing an infinite number of checks.

2. The following program is a typical example of recursion through negation:

```
even(0).                                       % File even.pnt in programs.zip
even(s(X)):-  ! even(X).
```

Both questions `!even(Z).` and `!even(Z),even(s(Z)).`, conducted by the BCN prototype, produce the answers:

- `Z = s(0);`
- `Z = s(s(s(0)));`
- `Z = s(s(s(s(s(0)))));`
- … and so on

However, the question `!even(Z),!even(s(Z))`. produces infinite computation (without answers). This is the expected behaviour if we consider the declarative semantics given by the three-valued logical consequences of program completion.

3. A similar behaviour is shown by the following definite program:

```
less(0,s(Y)).                                  %File less.pnt in programs.zip
less(s(X),s(Y)):-  less(X,Y).
```

The BCN-answers for the question `!less(X,Y)` are:

- `X = 0, Y = _A, _A ≠ s(*B);`
- `X = s(_A), Y = _B, _B ≠ s(*C);`
- `X = s(s(_A)), Y = s(_B), _B ≠ s(*C);`
- `X = s(0), Y = s(_A), _A ≠ s(*B);`
- `X = s(s(s(_A))), Y = s(s(_B)), _B ≠ s(*C);`
- `X = s(s(0)), Y = s(s(_A)), _A ≠ s(*B);`
- `X = s(s(s(s(_A)))), Y = s(s(s(_B))), _B ≠ s(*C);`
- … and so on

whereas the question `!less(X,s(X))` produces (as expected) infinite computation (without answers).

4. The following is a definite program to compute even (and odd) numbers in a different way:

```
sum(0,X,X).                          % File evenbysum.pnt in programs.zip
sum(s(X),Y,s(Z)):- sum(X,Y,Z).
even_by_sum(X):-   sum(Y,Y,X).
```

The question `!sum(Z1,Z2,Z3)` gives:

- `Z1 = 0, Z2 = _A, Z3 = _B, _B ≠ _A;`
- `Z1 = s(_A), Z3 = _B, _B ≠ s(*C);`
- `Z1 = s(s(_A)), Z3 = s(_B), _B ≠ s(*C);`
- `Z1 = s(0), Z2 = _A, Z3 = s(_B), _B ≠ _A;`
- `Z1 = s(s(s(_A))), Z3 = s(s(_B)), _B ≠ s(*C);`
- `Z1 = s(s(0)), Z2 = _A, Z3 = s(s(_B)), _B ≠ _A;`
- `Z1 = s(s(s(s(_A)))), Z3 = s(s(s(_B))), _B ≠ s(*C);`
- … and so on

and the question `!even_by_sum(Z)` gives:

- `Z = _A, _A ≠ s(s(*B)), _A ≠ 0;` (equivalent to `Z=s(0);`)
- `Z = s(s(_A)), _A ≠ 0, _A ≠ s(s(*B));` (equivalent to `Z=s(s(s(0)));`)
- … and so on

5. The following program produces very big computation trees:

```
evennumf(a)                          % File evennumf.pnt in programs.zip
evennumf(f(X,Y)):- evennumf(X),
                   ! evennumf(Y).
evennumf(f(X,Y)):- ! evennumf(X),
                   evennumf(Y).
```

The question `!evennumf(Z)` has infinitely many answers. The BCN prototype produces them as follows:

- `Z = f(a,a);`
- `Z = f(f(a,a),f(a,a));`
- `Z = f(f(a,a),f(f(a,a),f(a,a)));`
- `Z = f(f(a,f(a,a)),a);`
- `Z = f(f(f(a,a),a),a);`
- `Z = f(a,f(f(a,a),a));`
- `Z = f(a,f(a,f(a,a)));`
- `Z = f(f(f(a,a),f(a,a)),f(a,a));`
- `Z = f(f(f(a,a),f(a,a)),f(f(a,a),f(a,a)));`
- `Z = f(f(f(a,a),a),f(f(a,a),a));`
- … and so on

6. The following program was introduced in
W. Drabent., *What is failure? An approach to constructive negation,*
Acta Informatica, 32:27-59,1995.

```
r:-     ! p(X),! q(X).                          % File Drabent.pnt in programs.zip
p(X):- p(X).
p(a).
q(a):- q(a).
q(X):- ! s(X).
s(a).
w(f(X)). % We have added this clause to extend the signature with f/1, because
          % the BCN prototype infers the signature from the program and the goal.
```

It can be checked that:

- `!r.` One answer : `True;`
- `r.` fails

7. The following program was introduced in
R.Barták, Constructive Negation in CLP(H) , Technical Report No 98/6,
Dept. of Theoretical Computer Science, Charles Univ., Prague, July 1998.
(http://kti.ms.mff.cuni.cz/ ~bartak/html/negation.html)
where you can find a prototype for constructive negation restricted to the case of finite computation trees:

```
p(a,f(Z)):-   t(Z).                           % File Bartak.pnt in programs.zip
p(f(Z),b):-   t(Z).
t(c).
```

The question `!p(X, Y).` in the BCN prototype produces the following five answers:

- `X = a, Y = _A, _A ≠ f(*B);`
- `X = f(_A), Y = _B, _B ≠ b;`
- `X = _A, _A ≠ f(*B), _A ≠ a;`
- `X = a, Y = f(_A), _A ≠ c;`
- `X = f(_A), Y = b, _A ≠ c;`

whose disjunction is equivalent to the disjunction of the four answers produced by Barták's prototype.
Our answers are more general than Barták's ones. This is a consequence of the fact that our procedural method works for infinite computation trees, as well as for finite ones.

8. The following program was introduced in
A. Bossi, M. Fabris and M .C. Meo, *A bottom-up semantics for constructive negation,* In P.~V. Hentenryck, editor, *Proceedings of the 11th International Conference on Logic Programming (ICLP '94)*, pages 520-534, MIT Press, 1994.

```
p(f(V), Y):-      q(V).       % File in Bossietal.pnt in programs.zip
p(X, g(W)):-      ! r(W).
q(a).
r(b):-      r(b).
r(c):-      r(c).
r(c).
```

It can be checked that the BCN prototype behaves as follows:

- `!p(f(a),Y).` fails
- `!p(a,c).` One answer : `True`
- `!r(c).` fails

- `!r(b).` loops infinitely
- `!r(X).` One answer: `X = _A, _A ≠ b, _A ≠ c;` and then, it loops infinitely
- `!p(X,Y).` Four answers (the last two infinitely repeated):
  - `X = _A, Y = _B, _B ≠ g(*C), _A ≠ f(*D);`
  - `X = f(_A), Y = _B, _B ≠ g(*C), _A ≠ a;`
  - `X = _A, Y = g(c), _A ≠ f(*B);`
  - `X = f(_A), Y = g(c), _A ≠ a;`
- `!q(Z).` One answer: `Z = _A, _A ≠ a;`
- `!p(g(Z),f(Z)),q(Z).` One answer: `Z = a;`
- `p(X, g(Y)), q(Y).` Two answers:
  - `X = f(a), Y = a;`
  - `Y = a;`
- `!p(X,Y),r(Y).` Two answers (infinitely repeated):
  - `X = _A, Y = c, _A ≠ f(*B);`
  - `X = f(_A), Y = c, _A ≠ a;`

9. The following is a definite program to compute symmetric (non-symmetric) terms (trees) of the signature `{a/0,f/2,g/1}` is:

```
symmetric(a).                          % File symmetric.pnt in programs.zip
symmetric(g(X)):-  symmetric(X).
symmetric(f(X,Y)):-mirror(X,Y).
mirror(a,a).
mirror(g(X),g(Y)):-mirror(X,Y).
mirror(f(X,Y),f(Z,W)):-  mirror(X,W),mirror(Y,Z).
```

The BCN-answers for the question `!symmetric(Z)` are:

- `Z = f(a,_A), _A ≠ a;`
- `Z = f(f(_A,_B),_C), _C ≠ f(*E,*D);`
- `Z = f(g(_A),_B), _B ≠ g(*C);`
- `Z = g(f(a,_A)), _A ≠ a;`
- `Z = g(f(f(_A,_B),_C)), _C ≠ f(*E,*D);`
- `Z = g(f(g(_A),_B)), _B ≠ g(*C);`
- `Z = f(g(a),g(_A)), _A ≠ a;`
- `Z = f(g(f(_A,_B)),g(_C)), _C ≠ f(*E,*D);`
- `Z = f(g(g(_A)),g(_B)), _B ≠ g(*C);`
- … and so on

The question `!symmetric(f(X,X))` also produces an infinite number of answers, whereas the question `!symmetric(f(X,X)),mirror(X,X)` produces (as expected) an infinite computation (without answers).

10. The following example is a variant of a program introduced in

    D.Chan, Constructive Negation Based On The Completed Database, in R. A. Kowalski and K. A. Bowen, editors, *Proceedings of the Fifth International Conference and Symposium on Logic Programming*, pages 111-125, Seatle, 1988. The MIT press.

```
has_duplicates([X|Y]):-  member(X,Y).      % File Chan.pnt in programs.zip
has_duplicates([_|Y]):-  has_duplicates(Y).
member(X,[X|_]).
member(X,[_|Y]):-  member(X,Y).
list_of_digits([]).
list_of_digits([X|Y]):-  digit(X), list_of_digits(Y).
digit(1).
digit(2).
digit(3).
digit(4).
digit(5).
```

    The goal `! has_duplicates([X1,X2,X3]), list_of_digits([X1,X2,X3]).` asks for all the list of three digits without repetitions and the prototype gives the 60 possible combinations and then fails:

    - `X1 = 5, X2 = 2, X3 = 1;`
    - `X1 = 4, X2 = 2, X3 = 1;`
    - `X1 = 3, X2 = 2, X3 = 1;`
    - `X1 = 5, X2 = 3, X3 = 1;`
    - … and so on

11. The following is a program that computes pairs of disjoint lists of numbers:

```
list([]).                              % File disjoint.pnt in programs.zip
list([E|L]):-element(E), list(L).
element(0).
element(s(E)):-    element(E).
member(X,[X|_]).
member(X,[_|L]):-  member(X,L).
disjoint([],_).
disjoint([E|L1],L2):-    ! member(E,L2), disjoint(L1,L2).
```

    The goal `! disjoint(L1,L2), list(L1), list(L2).` asks for the pairs of disjoint lists of numbers:

    - `L1 = [0], L2 = [0];`
    - `L1 = [s(0)], L2 = [s(0)];`
    - `L1 = [0,0], L2 = [0];`
    - `L1 = [0,0], L2 = [0,0];`
    - … and so on

12. The following is the classical insertion sort program:

```
greater(s(_),0).                        % File list_sort.pnt in programs.zip
greater(s(E1),s(E2)):-   greater(E1,E2).
insert_sort(E,[],[E]).
insert_sort(E1,[E2|L],[E1|[E2|L]]):- ! greater(E1,E2).
insert_sort(E1,[E2|L],[E2|NL]):-     greater(E1,E2), insert_sort(E1,L,NL).
list_sort(L,NL):-  sort(L,[],NL).
sort([],L,L).
sort([E|L1],L2,NL):-     insert_sort(E,L2,LAux), sort(L1,LAux,NL).
list([]).
list([E|L]):-element(E), list(L).
element(0).
element(s(E)):- element(E).
```

Notice that the variable `LAux` in the predicate `sort/3` doesn't appear in the head, hence it produces.universal quantification

The goal `! list_sort(L1,L2), list(L1), list(L2).` gives all the pairs of number lists such that the second list is not the ordering of the first one:

- `L1 = [], L2 = [0];`
- `L1 = [s(0)], L2 = [];`
- `L1 = [s(0)], L2 = [0];`
- `L1 = [], L2 = [s(0)];`
- … and so on

13. This program returns the maximum of a list of numbers:

```
greater(0,E,E).                         % File max_list.pnt in programs.zip
greater(E,0,E).
greater(s(E1),s(E2),s(E3)):-   greater(E1,E2,E3).
max_list([],0).
max_list([E|L],NE):-     max_list(L,EAux), greater(E,EAux,NE).
list([]).
list([E|L]):-element(E), list(L).
element(0).
element(s(E)):-    element(E).
```

Notice also in this program that the variable `EAux` in the predicate `max_list/2` produces universal quantification.

The goal `! max_list(L,E), list(L), element(E).` gives all the pairs (list of numbers, number) such that the number is not the maximum of the list:

- `L = [], E = s(0);`
- `L = [0], E = s(0);`
- `L = [], E = s(s(0));`
- `L = [0], E = s(s(0));`
- … and so on

14. The following program computes the list difference operation:

```
remove_list(L,[],L).                    % File remove_list.pnt in programs.zip
remove_list(L1,[E|L2],NL):-   remove_element(L1,E,LAux),
                              remove_list(LAux,L2,NL).
remove_element([],_,[]).
remove_element([E1|L],E2,NL):-      equal(E1,E2), remove_element(L,E2,NL).
remove_element([E1|L],E2,[E1|NL]):- ! equal(E1,E2),
                                    remove_element(L,E2,NL).
equal(E,E).
list([]).
list([E|L]):-     element(E), list(L).
element(0).
element(s(E)):-   element(E).
```

Notice that the variable `LAux` in the predicate `remove_list/3` produces universal quantification.

The goal `! remove_list(L1,L2,L3), list(L1), list(L2), list(L3).` gives all the 3-tuples (L1,L2,L3) such that L3 is not L1 - L2.

- `L1 = [], L2 = [0], L3 = [0];`
- `L1 = [], L2 = [], L3 = [0];`
- `L1 = [0], L2 = [], L3 = [];`
- `L1 = [], L2 = [s(0)], L3 = [s(0)];`
- … and so on

15. This program computes the sum of a list:

```
sum_element(0,E,E).                 % File sum_list.pnt in programs.zip
sum_element(s(E1),E2,s(E3)):- sum_element(E1,E2,E3).
sum_list([],0).
sum_list([E|L],NE):-    sum_list(L,EAux), sum_element(E,EAux,NE).
list([]).
list([E|L]):-     element(E), list(L).
element(0).
element(s(E)):-   element(E).
```

Notice that the variable `EAux` in the predicate `sum_list/2` produces universal quantification.

The goal `! sum_list(L,E), list(L), element(E).` gives all the pairs (lists of numbers, number) such that the number is not the sum of the list values:

- `L = [], E = s(0);`
- `L = [0], E = s(0);`
- `L = [], E = s(s(0));`
- `L = [0], E = s(s(0));`
- … and so on

16. The following program computes the joint of two lists:

```
union([],L,L).                          % File union.pnt in programs.zip
union([E|L1],L2,NL):-   member(E,L2), union(L1,L2,NL).
union([E|L1],L2,[E|NL]):-     ! member(E,L2), union(L1,L2,NL).
member(E,[E|_]).
member(E,[_|L]):- member(E,L).
element(0).
element(s(E)):-   element(E).
list([]).
list([E|L]):-     element(E), list(L).
```

The goal `! union([0,s(0),s(s(0))],L1,L2), list(L1), list(L2).` gives all the pairs (L1,L2) such that L2 is not the joint of L1 and the list [0,s(0),s(s(0))]:

- `L1 = [], L2 = [];`
- `L1 = [s(0)], L2 = [s(0)];`
- `L1 = [s(0)], L2 = [];`
- `L1 = [], L2 = [s(0)];`
- `L1 = [], L2 = [0,0];`
- `L1 = [s(0)], L2 = [s(0),0];`
- `L1 = [], L2 = [s(0),0];`
- … and so on