Declarative Programming in Prolog and Beyond

- Declarative (logic) programming:
  - inherent power of Prolog
  - when not (properly) used: lengthy, buggy programs result
- Procedural programming, needed for:
  - efficiency reasons
  - termination guarantee

Terminology (1)

- From logic (Prolog as declarative language):

```
nat(0).
nat(s(X)) :- nat(X).
```

  - predicate symbol: nat (unary)
  - function symbol: s (unary)
  - term: 0, s(X), X
  - constant: 0 (= nullary function symbol)
  - variable: X
  - (positive) literal = atom: nat(0), nat(s(X)), nat(X)

Terminology (2)

- From programming languages (Prolog as procedural language):

```
nat(0).
nat(s(X)) :- nat(X).
```

  - term: nat(0), nat(s(X)), nat(X), :-(nat(s(X)), nat(X)), s(X), 0, X
  - functor: s, nat, :-
  - principal functor: nat in nat(s(X)), :- in :-(nat(s(X)), nat(X)), s in s(X)
  - number: 0
  - variable: X

Inversion of Computation (1)

- **Example:** concatenation of lists
  U = V ° W
  with U, V, W lists and ° concatenation operator
- Usage:
  - [a, b] = [a] ° **W** ⇒ **W** = [b]
  - [a, b] = **V** ° [b] ⇒ **V** = [a]
  - **U** = [a] ° [b] ⇒ **U** = [a, b]
  - [a, b] = **V ° W?**

Inversion of Computation (2)

- Prolog concatenation of lists:

```
concat([], U, U).
concat([X|U], V, [X|W]) :-
    concat(U, V, W).
```

- concat as constructor:

```
?- concat([a, b], [c, d], X).
   X = [a, b, c, d]
```

- concat used for decomposition:

```
?- concat(X, Y, [a, b, c, d]).
   X = []
   Y = [a, b, c, d]
```

Inversion of Computation (3)

- concat used for decomposition:

```
?- concat(X, Y, [a, b, c, d]).
   X = []
   Y = [a, b, c, d];
   X = [a]
   Y = [b, c, d];
   X = [a, b]
   Y = [c, d];
   ...
```

Order of Clauses (1)

- LP: order is irrelevant
- Prolog: order may be relevant
- Example:

```
member(X,[_|Y]) :-
    member(X,Y).
member(X,[X|_]).
:- member(a,[b,a,c]).
```

Order of Clauses (2)

```
/*1*/  member(X, [_|Y]) :-
           member(X, Y).
/*2*/  member(X, [X|_]).

?- member(a, [a,b]).
   X = a, Y = [b]          match with 1
   ?- member(a, [b]).      next call
      X' = a, Y' = []      match with 1
      ?- member(a, []).    fail 1 and 2
   fail 1 and 2
   fail 1, backtracking to 2
   X = a                   match 2
yes! (but not efficient)
```

Order of Clauses (3)

```
/*1*/  member(X, [_|Y]) :-
           member(X, Y).
/*2*/  member(X, [X|_]).

?- member(X, [a, b]).
   X' = X, Y = [b]         match with 1
   ?- member(X', [b]).     next call
      X'' = X', Y' = []    match with 1
      ?- member(X'', []).  fail 1 and 2
   X'  = b                 fail 1, match 2
   X = b;                  backtracking
   X = a                   match 2
yes! (but not efficient)
```

Order of Clauses (4)

```
/*1*/  member(X, [_|Y]) :-
           member(X, Y).
/*2*/  member(X, [X|_]).

?- member(a, Z).
   X = a, Z = [_|Y]        match 1
   ?- member(a, Y).        next call
      X' = a, Y= [_|Y']    match 1
      ?- member(a, Y').    next call
            .
            .
            .
```

Stackoverflow 

Conclusions Order of Clauses

- LP: order clauses is irrelevant
- Prolog:
  - Order has effect on efficiency of program
  - Order may affect termination: terminating program + order change ≠ terminating program

Order of Conditions (1)

- Length of list with successor function
  $s : N \rightarrow N$, with $s(x) = x + 1$
- Program:

```
/*1*/ length([], 0).
/*2*/ length([_|X], N) :-
         length(X, M),
         N = s(M).
```

- Use:

```
?- length([a, b], N).
N = s(s(0))
```

Order of Conditions (2)

- Program:

```
/*1*/ length([], 0).
/*2*/ length([_|X], N) :-
         length(X, M),
         N = s(M).
```

- Use:

```
?- length(L, s(0)).
L = [_A];
```

Stack overflow

Order of Conditions (3)

- Trace:

```
/*1*/ length([], 0).
/*2*/ length([_|X], N) :-
         length(X, M),
         N = s(M).
```

```
?- length(L, s(0)).
   L = [_A|X], N = s(0)            match 2
   ?- length(X, M), s(0) = s(M).   subcall
      X = [], M = 0                match 1
   ?- s(0) = s(0).                 match
   L = [_A];                       backtracks
      ...                          (1 fails)
```

Order of Conditions (4)

- Trace:

```
/*1*/ length([], 0).
/*2*/ length([_|X], N) :-
         length(X, M),
         N = s(M).
```

```
?- length(L, s(0)).
   L = [_A|X], N = s(0)            match 2
   ?- length(X, M), s(0) = s(M).   subcall
      X = [_B|X'], N = M           match 2
   ?- length(X', M'), M = s(M'),   subcall
      s(0) = s(M).
         ...
```

Order of Conditions (5)

- Program:

```
/*1*/ length([], 0).
/*2*/ length([_|X], N) :-
         N = s(M),
         length(X, M).
```

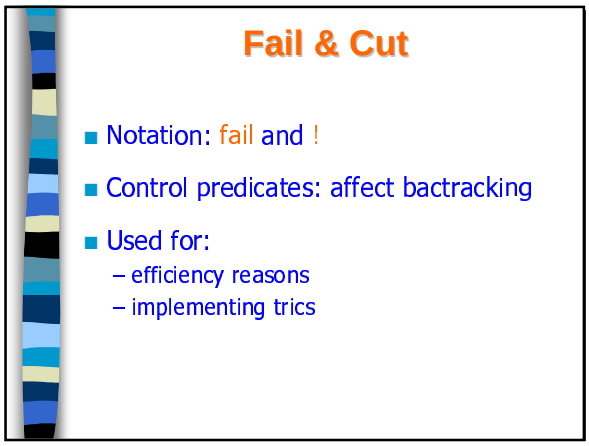
- Use:

```
?- length(L, s(0)).
L = [_A];
```

no

Declarative vs Procedural

- Order of clauses and conditions in clauses in Prolog progams may be changed, but
- This may be at the expense of:
  - loss of termination
  - compromised efficiency
- Schema for procedural programming:
  - special case first (top, left)
  - general case (e.g. including a recursive call) last (bottom, right)


Fail & Cut
Notation: fail and !
Control predicates: affect bactracking
Used for:
– efficiency reasons
– implementing trics