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

Inversion of Computation (1)

- *Example*: concatenation of lists
  - \( U = V \circ W \)
  - with \( U, V, W \) lists and \( \circ \) concatenation operator
- *Usage*:
  - \([a, b] = [a] \circ W \Rightarrow W = [b]\)
  - \([a, b] = V \circ [b] \Rightarrow V = [a]\)
  - \(U = [a] \circ [b] \Rightarrow U = [a, b]\)
  - \([a, b] = V \circ W ?\)

Inversion of Computation (2)

- *Prolog concatenation of lists*:
  - \(\text{concat}([U], U, U)\)
  - \(\text{concat}([X|U], V, [X|W]) :-\)
    \(\text{concat}(U, V, W)\)
  - \(\text{concat as constructor}:
    \text{?} - \text{concat}([a, b], [c, d], X).
    X = [a, b, c, d]\)
  - \(\text{concat used for decomposition}:
    \text{?} - \text{concat}(X, Y, [a, b, c, d]).
    X = []
    Y = [a, b, c, d]\)
Inversion of Computation (3)

- concat used for decomposition:

```prolog
?- 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:

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

Order of Clauses (2)

```prolog
/*1*/ member(X, []), member(X, [Y]).
/*2*/ member(X, [X]).

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

Order of Clauses (3)

```prolog
/*1*/ member(X, []), member(X, [Y]).
/*2*/ member(X, [X]).

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

Order of Clauses (4)

```prolog
/*1*/ member(X, []), member(X, [Y]).
/*2*/ member(X, [X]).

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

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
  \( s : N \to N \), 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([a, b], N).
  N = s(s(0))
  ```

Order of Conditions (3)
- Trace:
  ```
  /*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 (4)
- Trace:
  ```
  /*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 (5)
- Program:
  ```
  /*1*/ length([], 0).
  /*2*/ length([X], N) :-
    length(X, M),
    N = s(M).
  ```
- Use:
  ```
  ?- length([a, b], N).
  N = s(s(0))
  ```

Declarative vs Procedural
- Order of clauses and conditions in clauses in Prolog programs 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 backtracking
- Used for:
  - efficiency reasons
  - implementing trics