BAHASA PEMROGRAMAN

Pertemuan 4

REVIEW: PROGRAM MAKSIMUM DUA BILANGAN

Cara pertama:

```
maks(A, B, X) :- A >= B, X is A.
maks(A, B, X) :- A <  B, X is B.
```

Cara yang lebih singkat:

```
maks(A, B, A) :- A >= B.
maks(A, B, B) :- A <  B.
```

Apa bedanya?

Pada cara yang kedua, unifikasi langsung dilakukan antara variabel input dan output.

REVIEW: PROGRAM MAKSIMUM DUA BILANGAN

Cara lain dengan operator yang "jarang digunakan".
Dengan operator "atau" (titik koma):

```
maks(A, B, X) :- (A >= B, X is A) ; (A < B, X is B).
```

Dengan operator "if - then - else" (tanda panah, titik koma)

```
maks(A, B, X) :- A >= B -> X is A ; X is B.
```

RECURSIVE RULES

RECURSIVE RULES

- A procedure/program which calls itself typically until some final point is reached
- repeatedly perform some operation either over a whole data-structure, or until a certain point is reached.
- in Prolog, Recursive Rules mean :
  - we have a first fact that acts as some stopping condition
  - followed up by some rule(s) that performs some operation before reinvoking itself.

RECURSION

- parent(john,paul). /* paul is john's parent */

  parent(paul,tom). /* tom is paul's parent */

  parent(tom,mary). /* mary is tom's parent */

- Define rules of ancestor(X,Y)!
  - ancestor(X,Y):- parent(X,Y).
- Remember: *ancestor is not limited only to direct parent but also parent from parent*
  - ancestor(X,Y):- parent(X,Z), /* somebody is your ancestor if they are the parent */

    ancestor(Z,Y). /* of someone who is your ancestor */

RECURSIVE RULES

Make sure recursive rules have two components:

- Stop condition, when the recursion stop
- Recursive condition, when it calls itself

From rule below, which one is stop condition and recursive condition?

```
ancestor(A, B) :- parent(A, B).
ancestor(A, B) :- parent(A, X), ancestor(X, B).
```

WHICH OF THESE RULES ARE RECURSIVE?

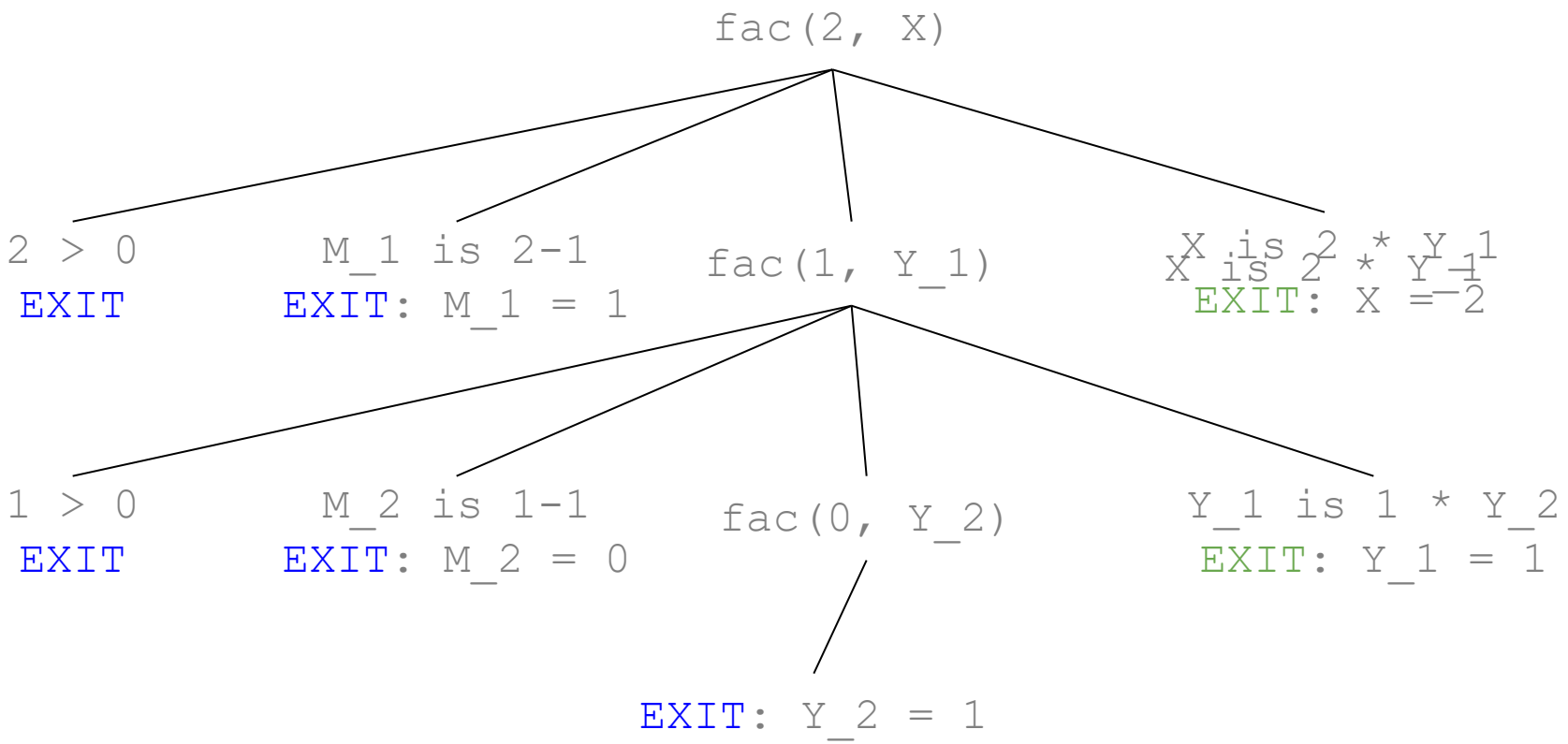
1. a(X):- b(X,Y), a(X). YES
2. go_home(no_12). NO
   go_home(X):- get_next_house(X,Y), home(Y).
3. foo(X):- bar(X). NO
4. lonely(X):- no_friends(X).
   no_friends(X):- totally_barmy(X). NO
5. has_flu(rebecca).
   has_flu(john). YES
   has_flu(X):- kisses(X,Y),has_flu(Y).
6. search(end). YES
   search(X):- path(X,Y), search(Y).

EXAMPLE : FACTORIAL

```
fac(0, 1).
fac(N, X) :- N > 0, M is N - 1, fac(M, Y), X is N * Y.
```

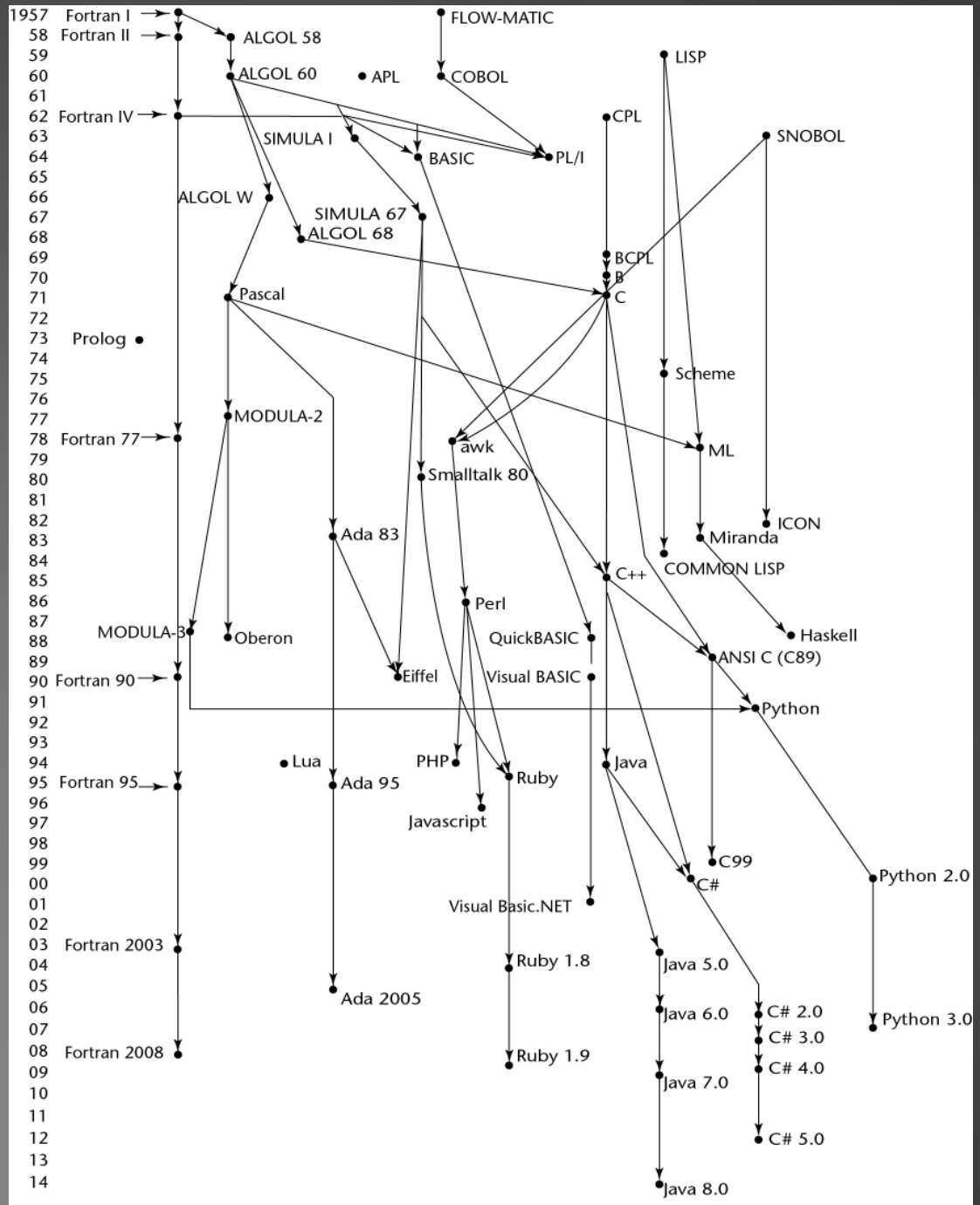
which one is stop condition and recursive condition?

Search tree for goal: `fac(2, X).`



EX: FAMILY TREE OF PROGRAMMING LANGUAGE

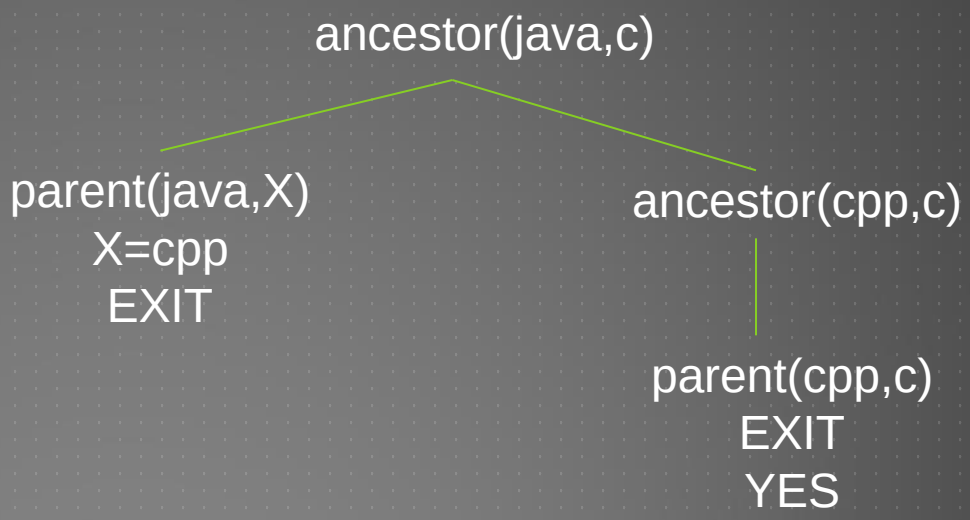
```
parent(c, b).
parent(c, algol).
parent(cpp, c).
parent(cpp, simula).
parent(java, cpp).
parent(csharp, java).
parent(csharp, cpp).
parent(awk, c).
parent(perl, awk).
parent(php, perl).
parent(javascript, perl).
parent(ruby, perl).
parent(scheme, lisp).
parent(basic, fortran).
parent(quickbasic, basic).
parent(visualbasic, quickbasic).
ancestor(A, B) :- parent(A, B).
ancestor(A, B) :- parent(A, X),
                  ancestor(X, B).
```



FAMILY TREE OF PROGRAMMING LANGUAGE

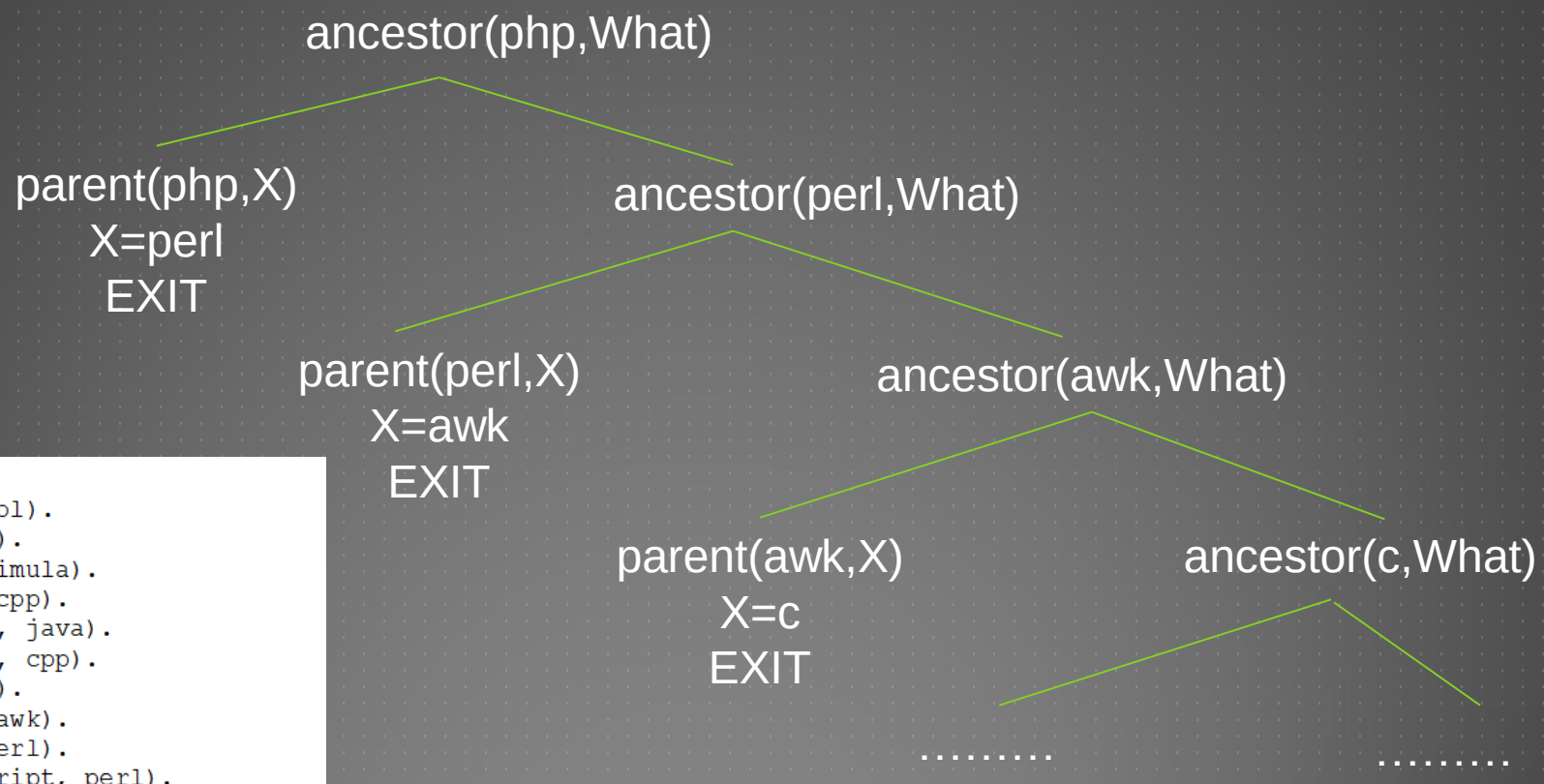
- Illustrate a search tree for goal: `ancestor(java, c).`

```
parent(c, b).
parent(c, algol).
parent(cpp, c).
parent(cpp, simula).
parent(java, cpp).
parent(csharp, java).
parent(csharp, cpp).
parent(awk, c).
parent(perl, awk).
parent(php, perl).
parent(javascript, perl).
parent(ruby, perl).
parent(scheme, lisp).
parent(basic, fortran).
parent(quickbasic, basic).
parent(visualbasic, quickbasic).

ancestor(A, B) :- parent(A, B).

ancestor(A, B) :- parent(A, X),
                  ancestor(X, B).
```



FAMILY TREE OF PROGRAMMING LANGUAGE

- Illustrate a search tree for goal: `ancestor(php, What).`




```
parent(c, b).
parent(c, algol).
parent(cpp, c).
parent(cpp, simula).
parent(java, cpp).
parent(csharp, java).
parent(csharp, cpp).
parent(awk, c).
parent(perl, awk).
parent(php, perl).
parent(javascript, perl).
parent(ruby, perl).
parent(scheme, lisp).
parent(basic, fortran).
parent(quickbasic, basic).
parent(visualbasic, quickbasic).

ancestor(A, B) :- parent(A, B).

ancestor(A, B) :- parent(A, X),
                  ancestor(X, B).
```

NON-LOGICAL FEATURE : CUT

"CUT" PREDICATE

- The **cut**, in Prolog, is a goal, written as !, which always succeeds, but cannot be backtracked.
- It is best used to prevent unwanted backtracking, including the finding of extra solutions by Prolog and to avoid unnecessary computations.
- There are two kinds of CUT:
  - ***Green cut***
  - ***Red cut***

GREEN CUT

- Does not change flow of program logic
- Only for searching efficiency

Without cut:

```
maks(A, B, A) :- A >= B.
maks(A, B, B) :- A <  B.
```

With cut:

```
maks(A, B, A) :- A >= B, !.
maks(A, B, B) :- A <  B.
```

without cut:

```
| ?- maks(10, 5, X).

X = 10 ? ;
no
```

with cut:

```
| ?- maks(10, 5, X).

X = 10
(1 ms) yes
```

Program maks has been given "cut" in the first rule
So it doesn't have to check second rule if first rule is succeeded

RED CUT

- Changes flow of program logic
- If cut is removed, program behaviour will change or will produce different output

```
hurufmutu(N, a) :- N >= 90, !.
hurufmutu(N, b) :- N >= 80, !.
hurufmutu(N, c) :- N >= 70, !.
hurufmutu(N, d) :- N >= 60, !.
hurufmutu(_, e).
```

```
hurufmutu(N, a) :- N >= 90.
hurufmutu(N, b) :- N >= 80.
hurufmutu(N, c) :- N >= 70.
hurufmutu(N, d) :- N >= 60.
hurufmutu(_, e).
```

```
{trace}
| ?- hurufmutu(77,H).
      1    1  Call: hurufmutu(77,_23) ?
      2    2  Call: 77>=90 ?
      2    2  Fail: 77>=90 ?
      2    2  Call: 77>=80 ?
      2    2  Fail: 77>=80 ?
      2    2  Call: 77>=70 ?
      2    2  Exit: 77>=70 ?
      1    1  Exit: hurufmutu(77,c) ?

H = c

(16 ms) yes
```

```
{trace}
| ?- hurufmutu(77,H).
      1    1  Call: hurufmutu(77,_23) ?
      2    2  Call: 77>=90 ?
      2    2  Fail: 77>=90 ?
      2    2  Call: 77>=80 ?
      2    2  Fail: 77>=80 ?
      2    2  Call: 77>=70 ?
      2    2  Exit: 77>=70 ?
      1    1  Exit: hurufmutu(77,c) ?

H = c ? ;
      1    1  Redo: hurufmutu(77,c) ?
      2    2  Call: 77>=60 ?
      2    2  Exit: 77>=60 ?
      1    1  Exit: hurufmutu(77,d) ?

H = d ? ;
      1    1  Redo: hurufmutu(77,d) ?
      1    1  Exit: hurufmutu(77,e) ?

H = e

(15 ms) yes
```

REVERSIBLE & IRREVERSIBLE PROGRAM

REVERSIBLE

- Example : kuadrat/2
- A database consist of :

```
kuadrat(1, 1).
kuadrat(2, 4).
kuadrat(3, 9).
kuadrat(4, 16).
kuadrat(5, 25).
kuadrat(6, 36).
```

- The predicate says that the second argument is square of the first argument
- There are four possible queries :

```
kuadrat(2, X).
kuadrat(X, 5).
kuadrat(X, Y).
kuadrat(2, 3).
```

- What is the square of this number (2) ?
- What is the root square of this number (5)?
- What numbers are related in square relationship?
- Are these numbers are related in square relationship?

```
| ?- kuadrat(2,X).
      1    1  Call: kuadrat(2,_23) ?
      1    1  Exit: kuadrat(2,4) ?

X = 4

yes
{trace}
| ?- kuadrat(X,5).
      1    1  Call: kuadrat(_23,5) ?
      1    1  Fail: kuadrat(_23,5) ?

no
{trace}
| ?- kuadrat(X,Y).
      1    1  Call: kuadrat(_23,_24) ?
      1    1  Exit: kuadrat(1,1) ?

X = 1
Y = 1 ?

(16 ms) yes
{trace}
| ?- kuadrat(2,3).
      1    1  Call: kuadrat(2,3) ?
      1    1  Fail: kuadrat(2,3) ?

no
```

IRREVERSIBLE

- Example : setelah/2
- A database consist of :

```
setelah(2, X).
setelah(X, 5).
setelah(X, Y).
setelah(2, 3).
```

```
setelah(A, B) :-
        B is A + 1.
```

```
{trace}
| ?- setelah(2,X).
      1    1  Call: setelah(2,_23) ?
      2    2  Call: _23 is 2+1 ?
      2    2  Exit: 3 is 2+1 ?
      1    1  Exit: setelah(2,3) ?

X = 3

yes
{trace}
| ?- setelah(X,5).
      1    1  Call: setelah(_23,5) ?
      2    2  Call: 5 is _23+1 ?
      2    2  Exception: 5 is _23+1 ?
      1    1  Exception: setelah(_23,5) ?
uncaught exception: error(instantiation_error
{trace}
| ?- setelah(X,Y).
      1    1  Call: setelah(_23,_24) ?
      2    2  Call: _24 is _23+1 ?
      2    2  Exception: _24 is _23+1 ?
      1    1  Exception: setelah(_23,_24) ?
uncaught exception: error(instantiation_error
{trace}
| ?- setelah(2,5).
      1    1  Call: setelah(2,5) ?
      2    2  Call: 5 is 2+1 ?
      2    2  Fail: 5 is 2+1 ?
      1    1  Fail: setelah(2,5) ?

no
```

MODE (CALLING PATTERN)

- For any given predicate with arity greater than 0, each argument may be intended to have one of three calling patterns:
  - Input : +
  - Output : -
  - Indeterminate : ?
- It is important to declare mode of argument in documentation

```
%% kuadrat(?,?)
%% setelah(+, ?)
```

LIST

LIST

- Built-in data structure in Prolog.
- List element could be anything, including list also.
- Example :
  - A = [ ]. %empty list
  - A = [1, 2, 3].
  - A = .(1, .(2, .(3,[ ]))).

HEAD & TAIL

- Each list has head and tail (except an empty list).
- List Append/Separate → (./2.) (|)
  - [1, 2, 3] = .(1, [2, 3]).
  - [1, 2, 3] = .(H, T).
  - [1, 2, 3] = [H|T].

EXAMPLE

MEMBER

- member(X, [X| _]).
- member(X, [_ |T]) : - member(X,T).

APPEND

- append([ ], B, B).
- append([H|At], B, [H|Ct]) : - append(At, B, Ct).

EXERCISE

Find the output of these queries!

1. member(2, [1, 2, 3]).
2. member(X, [1, 2, 3]).
3. member(X, L).
4. append([1, 2], [3, 4], [1, 2, 3]).
5. append([1, 2], B, C).
6. append(A, B, [1, 2, 3]).
7. append(A, [3, 4], C).
8. append(A, B, C).

BUILT-IN PREDICATES FOR LIST MANIPULATION

- append/3: append two lists
- length/2: get the length of a list
- member/2: check member of list
- reverse/2: reverse order of list
- last/2: get the last element
- first/2: get the first element
- second/2: get second element
- ….build the implementations (Homework)

DAFTAR PUSTAKA

- http://www.doc.gold.ac.uk/~mas02gw/prolog_tutorial/prologpages/recursion.html
- http://www.learnprolognow.org/lpnpage.php?pagetype=html&pageid=lpn-htmlse43
- http://www.cs.union.edu/~striegnk/learn-prolog-now/html/node88.html