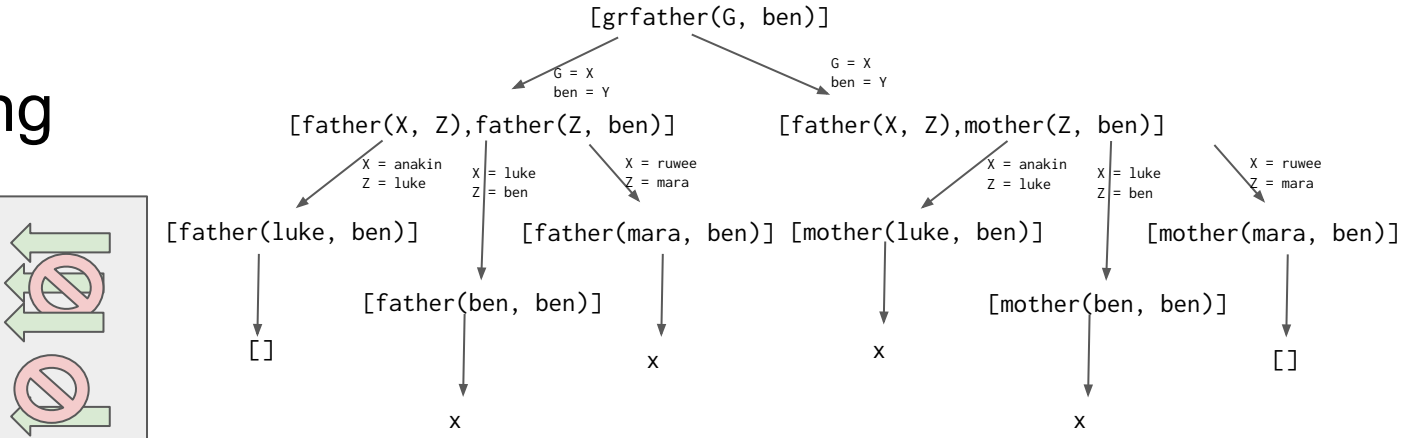
Prolog (3)

Backtracking

```
father(anakin, luke).
father(luke, ben).
father(ruwee, mara)

mother(padme, luke).
mother(mara, ben).

grfather(X, Y) :-
      father(X, Z),
      father(Z, Y).

grfather(X, Y) :-
      father(X, Z),
      mother(Z, Y).
```



Zie toledo voor tutorial!

is/2 en =/2

- =/2 is voor **unification**.
- is/2 is voor **evaluatie** en **assignment**

```
?- X is 1 + 2.
```

```
X = 3
```

```
?- X = 1 + 2.
```

```
X = 1 + 2
```

```
?- Y is 1, X is Y + 2.
```

```
Y = 1, X = 3
```

```
?- X = Y + 1, Y is 2.
```

```
Y = 2, X = 2 + 1
```

```
?- X is Y + 1, Y is 2.
```

```
?- X is 3, X is X + 1.
```

```
false
```

is/2 en =/2

- =/2 is voor **unification**.
- is/2 is voor **evaluatie** en **assignment**

Pas op: unificatie

```
sum(X, Y, R) :-
     R = X + Y.
```

```
?- sum(3, 2, R).
R = 3 + 2

?- sum(1 + 1, 2, R).
R = 1 + 1 + 2
```

```
sum(X, Y, R) :-
     R is X + Y.
```

```
?- sum(3, 2, R).
R = 5

?- sum(1+1, 2, R).
R = 4
```

```
sum(X, Y, X + Y).
```

```
?- sum(3, 2, R).
R = 3 + 2

?- sum(1 + 1, 2, R).
R = 1 + 1 + 2
```

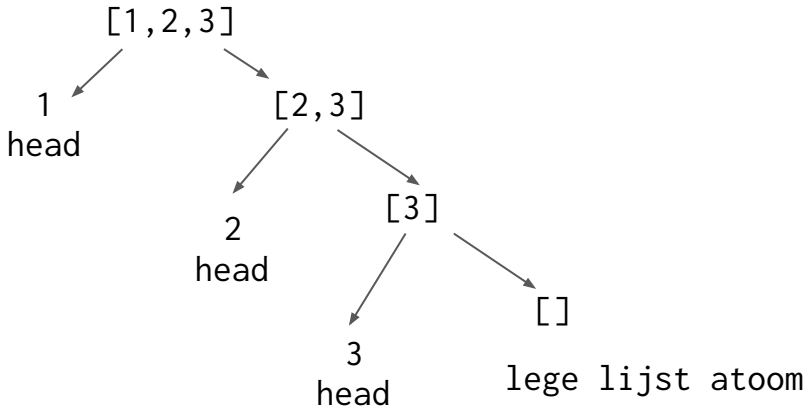
Lijsten

```
X = [1,2,3]
```

```
?- [X|Xs] = [1,2,3]
   ⇒  X = 1, Xs = [2,3]
```

```
?- [X1, X2 |Xs] = [1,2,3]
   ⇒ X1 = 1, X2 = 2, Xs = [3]
```

```
?- [X1, X2 |Xs] = [[a, b],2,[]]
   ⇒ X1 = [a, b], X2 = 2, Xs = [[]]
```



Lijsten

Lijsten loopen:

```
sum([], 0).
sum([E|Es], R) :-
     sum(Es, Rs),
     R is Rs + E.
```

Lijsten maken:

```
rrange(0, []).
rrange(N, [N|Rest]) :-
     N > 0,
     N1 is N - 1,
     rrange(N1, Rest).
```

Beiden:

```
square([], []).
square([E|Es], [R|Rs]) :-
     R is E * E,
     square(Es, Rs).
```

Lijst operaties

library(lists): List Manipulation

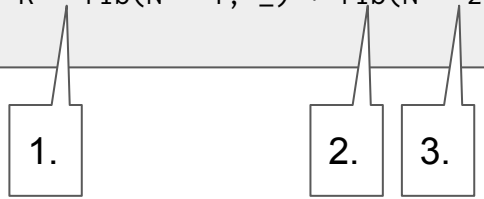
- member/2
- append/3
- append/2
- prefix/2
- select/3
- selectchk/3
- select/4
- selectchk/4
- nextto/3
- delete/3
- nth0/3
- nth1/3
- nth0/4
- nth1/4
- last/2
- proper_length/2
- same_length/2
- reverse/2
- permutation/2
- flatten/2
- clumped/2
- max_member/2
- min_member/2
- max_member/3
- min_member/3
- sum_list/2
- max_list/2
- min_list/2

Lijsten zijn essentiële data structuren!
→ Neem de built-ins eens door (kunnen van pas komen)
→ Probeer zelf een paar built-ins te schrijven

Belangrijke predicaten:
→ `member/2`
→ `select/3`
→ `append/3`
→ `permutation/2`

Veel voorkomende fouten (1).

```
fib(1, 0).
fib(2, 1).
fib(N, R) :-
      N > 2,
      R = fib(N - 1, _) + fib(N - 2, _).
```




```
fib(1, 0).
fib(2, 1).
fib(N, R) :-
      N > 2,
      N1 is N - 1,
      N2 is N - 2,
      fib(N1, F1),
      fib(N2, F2),
      R is F1 + F2.
```

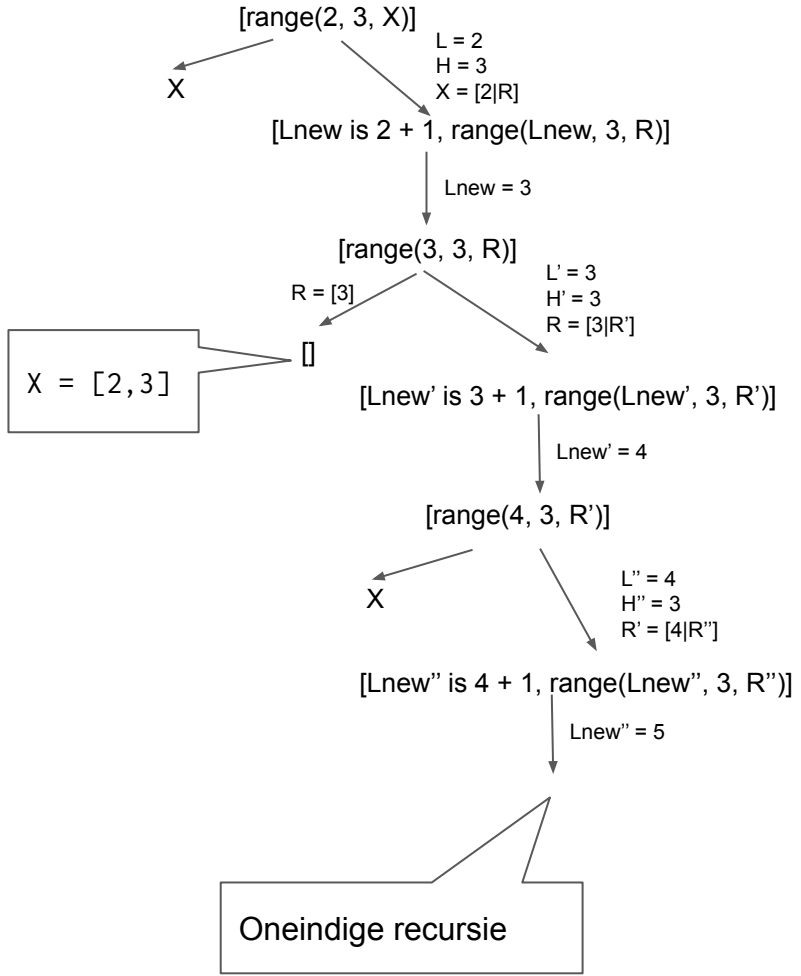
1. Unificatie =/= assignment
2. Predicaten zijn geen functies (hebben geen output).
3. N - 1 en N - 2 zijn prolog **termen**

Veel voorkomende fouten (2).

```
range(H, H, [H]).
range(L, H, [L|R]) :-
      Lnew is L + 1,
      range(Lnew, H, R).
```

```
range(H, H, [H]).
range(L, H, [L|R]) :-
      L < H,
      Lnew is L + 1,
      range(Lnew, H, R).
```

Oneindige recursie → Is je **base case** ook uitgesloten?



Oefening 1.

Tips:

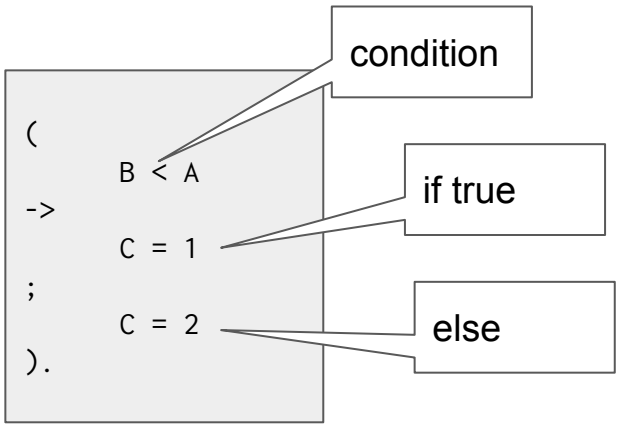
1. `balanced/1` → +- letterlijk vertalen
2. `add_to/3` → Keep it simple

Hence, the expression `t(t(nil,2,nil),3,nil)` represents a tree. A tree is balanced if the depths of the left and right subtree differ by at most one, and both subtrees are balanced as well.

Oefening 2.

Tips:

1. 3 ifs→ 3 rules
2. Een If-else in prolog:




```
alpha_beta(tree, a, b, maximize) → Value

        // base case. Example leaf(4, ‘hello’)
        if tree == leaf(Score, V):
                return Score

        // maximizing node
        if maximize:
                child_max = - inf
                for child in tree:
                        // recursive call
                        child_v ← alpha_beta(child, a, b, false)

                        // update max values
                        child_max ←  max(child_max, child_v)
                        a ← max(child_max, a)

                        // pruning condition
                        if b < a:
                                break

                return child_max


        // minimizing node
        if not maximize:
                ...
```

Oefening 3.

Naive way

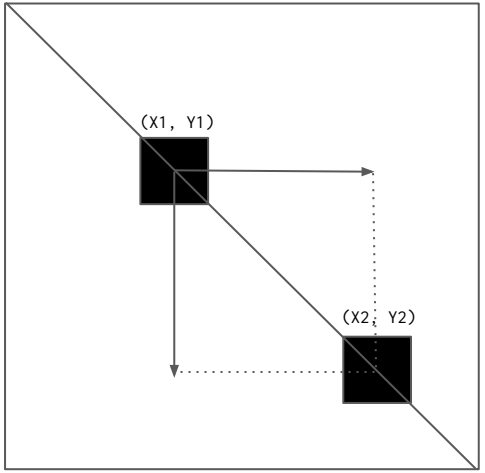
```
queens(N, Solution) :-
       generate(N, Solution),
       test(N, Solution).

generate(N, Configuration) :-
       % generates a potential correct
       % n-queens configuration.
       ...

test(Configuration) :-
       % checks if the given configuration
       % is a valid n-queens solution.
       ...
```

somewhat better

```
queens(N, Solution) :-
       % Recursively do:
       % 1. select row of queen i
       % 2. check if queen attacks any other queen
```

Tips:

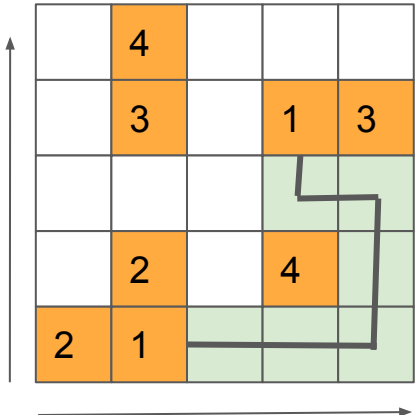
1. Lees het stukje over board configurations
2. Implementeer zelf een `range/3` en `permutation/2` (of gebruik built-in)
3. Gegeven (X1, Y1) en (X2, Y2), wanneer zijn ze op dezelfde diagonaal?



Oefening 4.

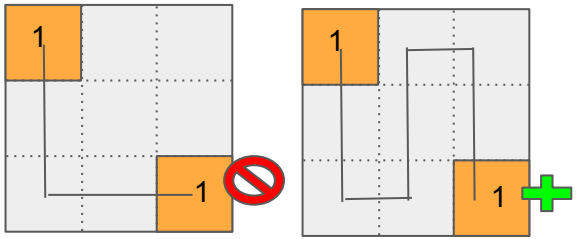
Used = Numbers + Paths

```
connect(Grid, Links, Used, Solution) :-
        % Grid = grid(Breadth, Height)
        % Links = [link(N, P1, P2), ...]
        % Used = [pos(P1, P2), ...]
        % Solution = [connects(N, [...]), ...]

        % 1. Find a path for the first link
        % 2. Add new found path to Used
        % 3. Find a path for other links

find_path(G, P1, P2, Used, PathSoFar, NewPath) :-
        % helper predicate
```

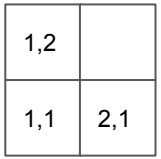


Tips:

1. Een oplossing is enkel valid als elk vakje van de grid in een pad voorkomt.



2. Maak gebruik van `is_valid_edge(G, P1, P2)` om buren te genereren.




```
is_valid_edge(grid(2,2), pos(1,1), R)
   -     R = pos(2, 1)
   -     R = pos(1, 2)
```

3. `find_path/5` is een complexere versie van `path/3` van oefenzitting 2.