Programowanie w logice

języki programowania

imperatywne

deklaratywne

funkcyjne

logiczne

Java
C++
Pascal
Fortran
⋮

Lisp
Haskell

Prolog

jak obliczymy?
↓
instrukcje
↓
Model von Neumann'a

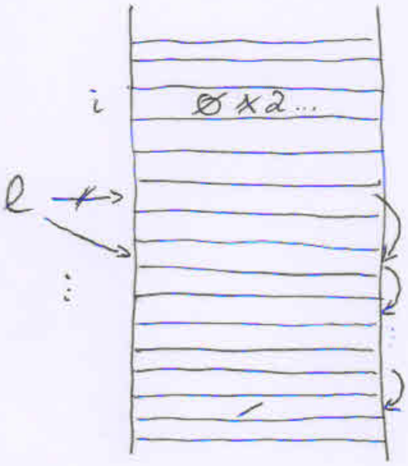
co obliczymy?
↓
właściwości
↓
Model?

Przykład: Długość listy

(i) wersja imperatywna

"manipuluje wartości zmiennych w pamięci"

```
int length (lista l){
  int i := 0;
  while l.next ≠ NULL {
    i := i+1;
    l := l.next; }
  return i; }
```



(ii) wersja logiczna

'używa właściwości length'

- długość pustej listy, to 0.
- długość niepustej listy, to jeden plus długość ogona listy

↳ predykat (relacja)

uwaga: Właściwości te (też) w wersji imperatywnej pokazują poprawność programu.

```
length([], 0).
length([X|L],N) :- length(L,M), N = M+1.
```

Interpretacja deklaratywna

$K :- B_1, \ldots B_m \quad (m \geq 0)$

k poprawny, jeżeli $B_1, \ldots B_m$ poprawne,
tzn. $(B_1 \wedge \ldots \wedge B_m) \Rightarrow K$

Interpretacja proceduralna

$K :- B_1, \ldots B_m \quad (m \geq 0)$

Aby pokazać k, pokazuj $B_1, \ldots B_m$ (w tej kolejności)

length([1,2], 2)

| X = 1, L = [2], N = 2

length([2], M), 2 = M + 1

| X' = 2, L' = [], M = 2

length([], P), M = P + 1, 2 = M + 1

| P = 0

M = 0 + 1, 2 = M + 1

| M = 1

2 = 1 + 1

|

True

length([1,2], 1)

| X = 1, L = [2], N = 1

length([2], M), 1 = M + 1

|

⋮

M = 0 + 1, 1 = M + 1

| M = 1

1 = 1 + 1

|

False

↳ Prolog sprawdza, czy predykat jest poprawny – nie obliczy!

"Obliczenie" przez wartości zmiennych:

length([1,2], N) poprawny, jeżeli

$\exists N_0$: length([1,2], N_0). – Taki(e) wartość(i) jest (są) wynikiem "obliczenia length([1,2], N).

↳ "wyniki obliczenia" w Prologu nie muszą być jednoznacznie!

↳ predykaty (a nie funkcji)

length([1,2], N)

| $X = 1$, $L = [2]$

length([2], M), N = M + 1

| $X' = 2$, $L' = []$

length([], P), M = P + 1, N = M + 1

| $P = 0$

M = 0 + 1, N = M + 1

| $M = 1$

N = 1 + 1

|

True

↳ Prolog odpowiada "N = 2"

?- length([1,2], 2)

True

?- length([1,2], 1)

False

?- length([1,2], N)

N = 2

Literatura

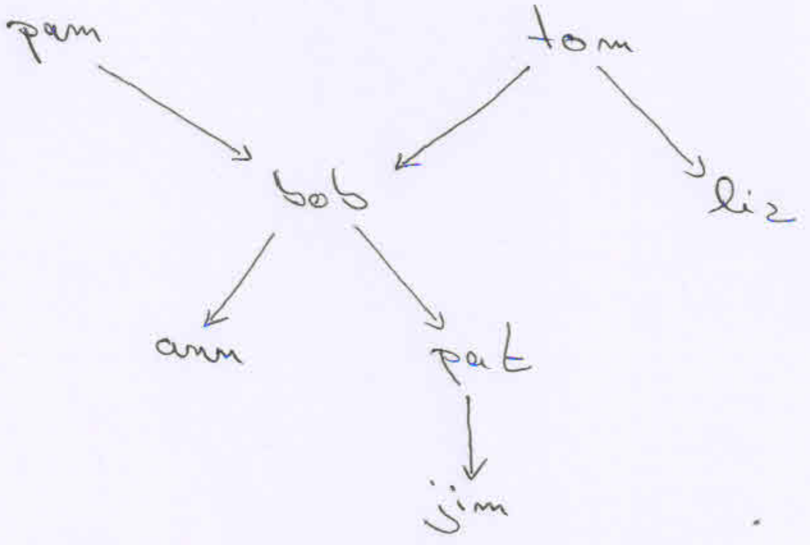
- Bratko; Prolog - Programming for Artificial Intelligence
- "Logic for Computer Science"
  (PL1, unifikacja, rezolucja)

§1 Wprowadzenie do Prologu

- obiekty
- relacje między obiektami (predykaty)
- Fakty, Reguły, Pytania

Fakty:

```
parent(tom, bob).
parent(tom, liz).
parent(pam, bob).
```



Pytania (queries):

```
?- parent(tom, bob).
True

?- parent(liz, pat).
False
```

?- parent(bob, X).

X = ann ;

X = pat ;

False

?- parent(Y, jim), parent(X, Y)

| Y = pat

parent(X, pat)

| X = bob

True

↳ Y = pat, X = bob

<u>uwaga:</u> Fakt parent(tom, X) przewiduje, że tom jest rodzicem "wszystkiego"!

matka(pam, bob).

ojciec(tom, bob).

ojciec(tom, liz).

⋮

jest złym rozszerzeniem faktów (bazy)

<u>lepiej:</u>

```
female (pam).
male (tom).
male (bob).
  ⋮
```

Requête:

```
mother (X, Y) :- female (X), parent (X, Y).
grandparent (X, Y) :- parent (X, Z),
                      parent (Z, Y).

predecessor (X, Z) :- parent (X, Z).
predecessor (X, Z) :- parent (X, Y),
                      predecessor (Y, Z).
```

Jak Prolog odpowiada na pytanie?

Program P (= definicje predykatów)

?- $p_1(X), \ldots p_n(X)$.

Prolog spróbuje spełnić wszystkie $p_i(X)$ jednocześnie - pod założeniem, że P jest spełniony, tzn. $\forall i = 1, \ldots n: M \models p_i$

Przykład:

$$\left.\begin{array}{l} \text{fallible}(X) \text{ :- } \text{man}(X). \\ \text{man}(\text{socrates}). \end{array}\right\} \text{program } P$$

?- fallible(socrates).

P spełnione, czyli $\forall X.\ \text{man}(X) \Rightarrow \text{fallible}(X)$, i dla X = socrates

$$\text{man}(\text{socrates}) \Rightarrow \text{fallible}(\text{socrates}).$$

Ponieważ man(socrates) $\in$ P jest spełnione, teraz wynika, że

$$\text{fallible}(\text{socrates})$$

musi być spełnione, tzn.

$$P \models \text{fallible}(\text{socrates})$$

i Prolog odpowiada "True".

w Prologu używa się

- $P \vdash q \iff P \cup \{\neg q\}$ nie spełnialny
- $H \text{:- } B_1, \ldots B_n.$ "$=$" $(B_1 \wedge \ldots \wedge B_n) \Rightarrow H$

$$\models\!\!\dashv \quad H \vee \neg B_1 \vee \ldots \vee \neg B_n$$

czyli

$\forall X \; fallible(X) \vee \neg man(X)$

$man(socrates)$

$\neg fallible(socrates)$

obserwacja:

jeżeli $(A \vee \neg B) \wedge (B \vee C)$ spełniona,
to też $A \vee C$ spełniona.

piszemy

$$\frac{A \vee \neg B \qquad B \vee C}{A \vee C}$$

więc

$$\frac{\dfrac{\begin{array}{l} fallible(socrates) \\ \vee \neg man(socrates) \end{array} \qquad man(socrates)}{fallible(socrates)} \qquad \neg fallible(socrates)}{\Box}$$

Przykład:

pre(X, Y) :- parent(X, Y). R1
pre(X, Y) :- parent(X, Z), pre(Z, Y). R2
parent(tom, liz). F1
parent(tom, bob). F2
parent(bob, pat). F3

?- pre(tom, pat).

1. parent(bob, pat) F3
2. pre(bob, pat) R1(1.)
3. parent(tom, bob) F2
4. pre(tom, pat) R2(3., 2.)

↳ forward-reasoning
wszystko, co generujemy jest poprawne, tzn. konsekwencją programu

backward-reasoning

pre(tom, pat)

| R2

parent(tom, bob), pre(bob, pat)

✓F2 | R1

parent(bob, pat)

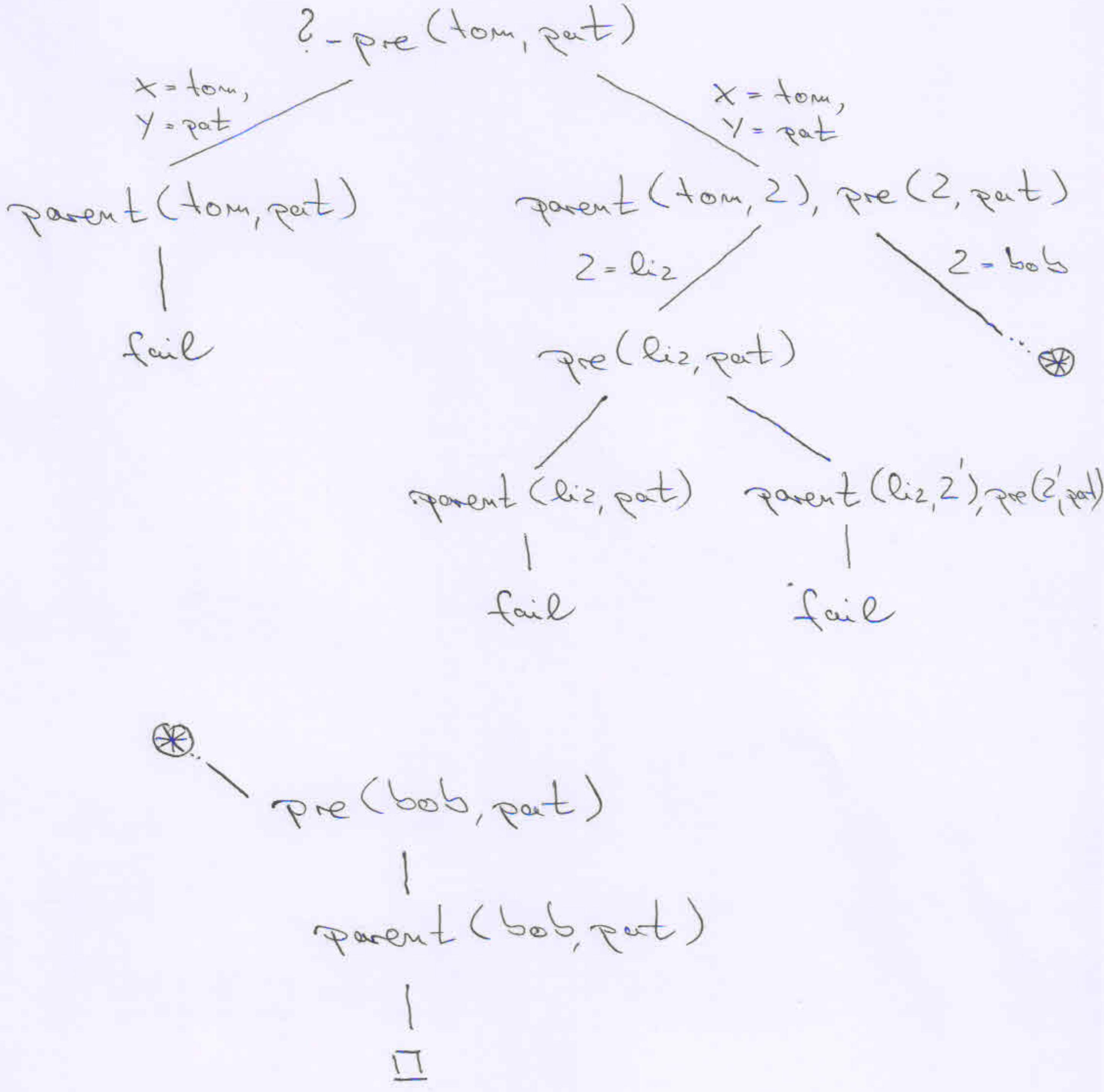
✓F3

ale: Dlaczego R2, a nie R1?

Dlaczego F2, a nie F1 lub F3?

↳ Prolog szuka:

- cele spełnia od lewej do prawej
- fakty i reguły używa od góry



Definicja

(i) formułę $H :- B_1, \dots B_n$, $n \geq 0$ nazywamy klauzulą (clause).

(ii) H, to głowa (head) a $B_1, \dots B_n$, to ciało (body) klauzuli.

(iii) funkcję σ, która przydziela wartości zmiennym, nazywamy substytucję.

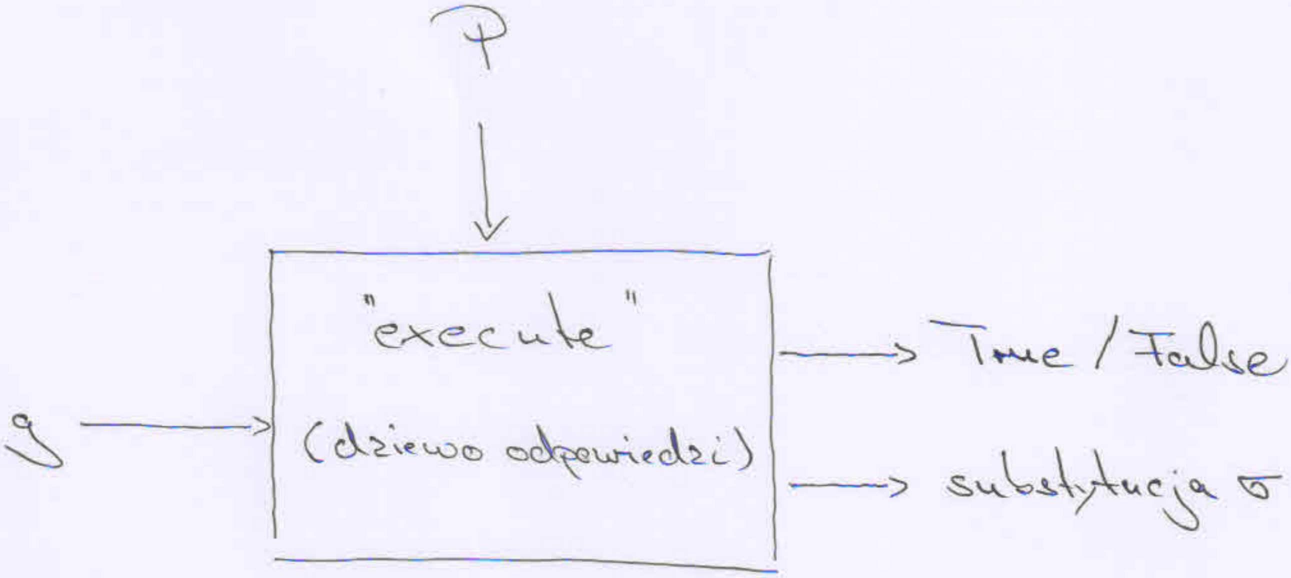
Program P, pytanie g

Deklaratywna Interpretacja

$$?- g \text{ "="} \ True \ :\Longleftrightarrow$$

$$\exists \text{ klauzula } C \in P$$
$$\exists \text{ substytucja } \sigma :$$
$$\sigma(head(C)) = \sigma(g) \ \wedge$$
$$\forall B_i \in body(C) : \ ?- \sigma(B_i) \text{ "="} \ True$$

Proceduralna Interpretacja



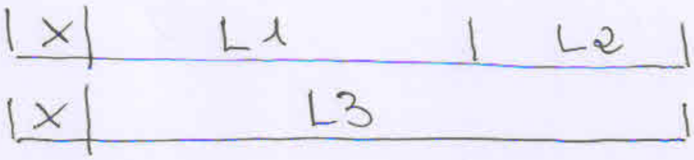
Listy

[] pusta lista

[|] dołączy element do listy - z lewej

$[1 | [2 | [a | []]]] = [1, 2, a]$

```
append([], L, L).
append([X|L1], L2, [X|L3]) :- append(L1, L2, L3).
```



?- append([1,2], [3,4], L).

$\quad | \; X = 1,\ L_1 = [2],\ L_2 = [3,4],\ L = [X | L_3]$

append([2], [3,4], L_3)

$\quad | \; X' = 2,\ L_1' = [],\ L_2' = [3,4],\ L_3 \doteq [X' | L_3']$

append([], [3,4], L_3')

$\quad | \; L_3' = [3,4]$

True

$L = [X | L_3] = [1 | [X' | L_3']] = [1 | [2 | [3,4]]]$

$= [1, 2, 3, 4]$

?- append([1,2], [3,4], [1,3,4]).

| $X = 1,\ L_1 = [2],\ L_2 = [3,4],\ L_3 = [3,4]$

append([2], [3,4], [3,4])

| $X' = 2,\ L_1' = [\,],\ L_2' = [3,4]$

$[3,4] \overset{!}{=} [X' | L_3'] \rightsquigarrow X' = 3$ ↯ $X' = 2$

False

?- append([1], [Y], [1,2])

| $X = 1,\ L_1 = [\,],\ L_2 = [Y],\ L_3 = [2]$

append([], [Y], [2])

| $L = [Y] = [2] \rightsquigarrow Y = 2$

True

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

member(2, [1,2,3])

2 = 1, _ = [2,3] → True

2 = X, _ = 1, L = [2,3] → member(2, [2,3])

2 = 2, _ = [3] → True

2 = X, _ = 2, L = [3] → member(2, [3])

2 = 3, _ = [] → True

2 = X, _ = 3, L = [] → member(2, [])

False

↳ ?- member(2, [1,2,3]).

2 = 1 ;

2 = 2 ;

2 = 3 ;

False

$L = \boxed{\quad L_1 \quad | X | \quad L_2 \quad}$

↳ member(X, L) :- append(L_1, [X | L_2], L).

Arytmetyka

```
length([],0).
length([X|L],N) :- length(L,M), N = M+1.
```

```
?- length([1,2,3],N).
N = ((0+1)+1)+1
```

```
length([1],N)
   | X=1, L=[]
length([],M), N = M+1
   | M=0
N = 0+1
   |
True                σ(N) = 0+1
```

```
?- X = 1+2.
X = 1+2
?- X is 1+2.
X = 3
?- X is Y+2.
```

↯ Y nie ma wartości

```
?- X=3, Y is X+2.
X = 3,
Y = 5
```

```
length([],0).
length([X|L],N) :- length(L,M), N is M+1.

?- length([1,2,3],N).
N = 3
```

uwaga:

```
length([],0).
length([X|L],N) :- N is M+1, length(L,M).
```

```
length([1,2,3],N)
      / X=1, L=[2,3]
N is M+1, length([2,3],M)
```

↯ M nie ma wartości

Termy

(i) proste
- stałe (atomy, liczby)
- zmienne

(ii) strukturalne

```
date(1, may, 1983).

?- date(Day, may, 1983).
Day = 1
```

Termy reprezentują obiekty!

tom

date (Day, june, 2019)

family (tom, bob, peter)

point (1, 1)

point (1, 2, 3)

Przykład: Listy

stała nil: pusta lista ("=" [])

funktor cons: dołączy element do listy ("=" [|])

czyli cons(1, cons(2, cons(3, nil))) "=" [1, 2, 3]

```
length(nil, 0).
length(cons(X, L), N) :- length(L, M), N is M+1.
```

length(cons(1, cons(2, nil)), N)

/ X = 1, L = cons(2, nil)

length(cons(2, nil), M), N is M+1

/ X' = 2, L' = nil

length(nil, M'), M is M'+1, N is M+1

/ M' = 0

M is 0+1, N is M+1

/ M = 1

N is 1+1

/ N = 2

True

Przykład: Figury geometryczne

```
point(1,1).
point(2,3).
seg(point(1,1), point(2,3)).
triangle(point(4,2), point(6,4), point(7,1)).

?- triangle(X, point(X1,X2), point(7,1)).
X = point(4,2),
X1 = 6,
X2 = 4

?- triangle(point(1,1), P, point(2,3)),
   triangle(X, point(4,Y), point(2,Z)).
X = point(1,1),
P = point(4,Y),
Z = 3


vertical(seg(point(X,Y1), point(X,Y2))).
horizontal(seg(point(X1,Y), point(X2,Y))).

?- vertical(seg(point(1,1), point(1,3))).
True
?- vertical(seg(point(1,1), point(2,Y))).
False
?- horizontal(seg(point(1,1), point(2,Y))).
Y = 1
```

?- vertical(S), horizontal(S).

/ S = seg(point(X, Y_1), point(X, Y_2))

horizontal(seg(point(X, Y_1), point(X, Y_2)))

/ $X = X_1$, $Y_1 = Y$
$X = X_2$, $Y_2 = Y$

True

↳ S = seg(point(X,Y), point(X,Y))

<u>Przykład:</u> Drzewa binarne

nil
drzewo(X, L, R)

```
height(nil, 0).
height(drzewo(X, L, R), N) :-
        height(L, H1),
        height(R, H2),
        H1 =< H2,
        N is H2 + 1.
height(drzewo(X, L, R), N) :-
        height(L, H1),
        height(R, H2),
        H1 > H2,
        N is H1 + 1.
```

```
height(nil, 0).
height(drzewo(X, L, R), N) :-
        height(L, H1),
        height(R, H2),
        (H1 >= H2, N is H1 + 1); (H1 < H2, N is H2 + 1).

sum(nil, 0).
sum(drzewo(X, L, R), N) :-
        sum(L, N1), sum(R, N2), N is X + N1 + N2

times(N, nil, nil).
times(N, drzewo(X1, L1, R1), drzewo(X2, L2, R2)) :-
        X2 is N * X1,
        times(N, L1, L2), times(N, R1, R2).


times(5, drzewo(3, drzewo(4, nil, nil),
                   drzewo(1, nil
                            drzewo(7, nil, nil))), D)

  | N = 5, X1 = 3,
  |     L1 = drzewo(4, nil, nil),
  |     R1 = drzewo(1, nil, drzewo(7, nil, nil))
  | D = drzewo(X2, L2, R2)

X2 is 5 * 3, times(5, drzewo(4, nil, nil), L2),
             times(5, drzewo(1, nil, drzewo(7, nil, nil)), R2)

  | X2 = 15

times(5, drzewo(4, nil, nil), L2),
times(5, drzewo(1, nil, drzewo(7, nil, nil)), R2)
```

$N = 5,\ X1' = 4,$
$L1' = nil,\ R1' = nil$
$L2 = drzewo(X2', L2', R2')$

$X2' = 5 * 4$, times$(5, nil, L2')$, times$(5, nil, R2')$,
times$(5$, drzewo$(1, nil$, drzewo$(7, nil, nil)), R2)$

$X2' = 20$

times$(5, nil, L2')$, times$(5, nil, R2')$,
times$(5$, drzewo$(1, nil$, drzewo$(7, nil, nil)), R2)$

$L2' = nil,\ R2' = nil$

times$(5$, drzewo$(1, nil$, drzewo$(7, nil, nil)), R2)$

⋮

do tej pory:

$D =$ drzewo$(X2, L2, R2)$
$=$ drzewo$(15$, drzewo$(X2', L1', R1'), R2)$
$=$ drzewo$(15$, drzewo$(20, nil, nil), R2)$

dalej analogicznie:

$R2 =$ drzewo$(5, nil$, drzewo$(35, nil, nil))$