Programowanie w Logice

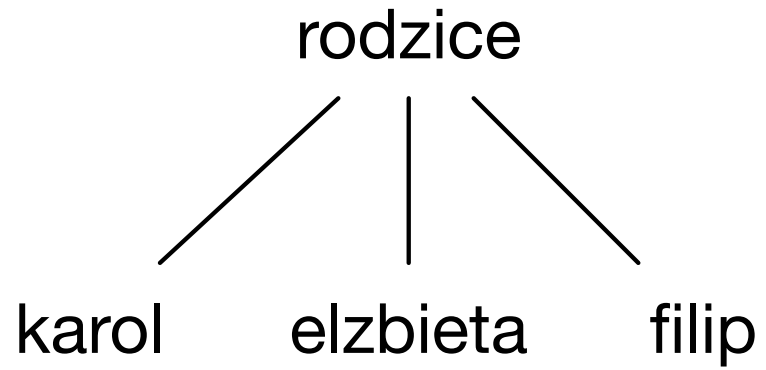
Struktury danych

Przemysław Kobylański
na podstawie [CM2003]

Struktury danych

Struktury a drzewa

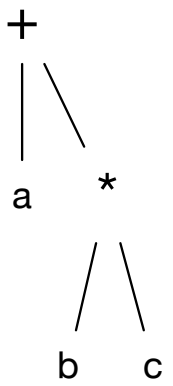
```
rodzice(karol, elżbieta, filip)
```



Struktury danych

Struktury a drzewa

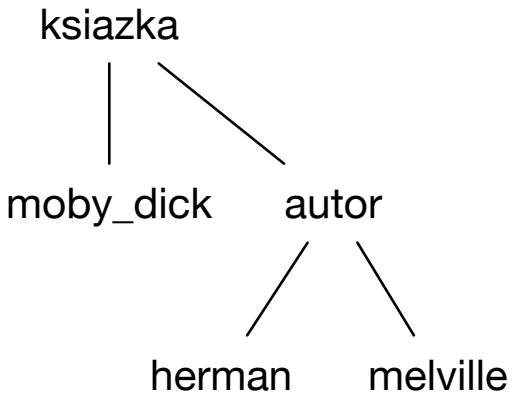
```
a + b * c = +(a, *(b, c))
```



Struktury danych

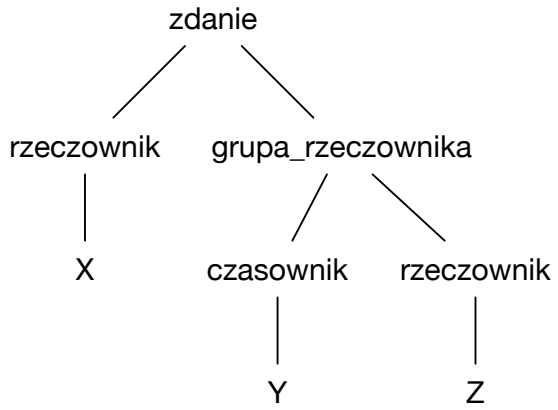
Struktury a drzewa

```
książka(moby_dick, autor(herman, melville))
```



Struktury danych
Struktury a drzewa

```
zdanie(rzeczownik(X), grupa_rzeczownika(czasownik(Y),
                                        rzeczownik(Z)))
```



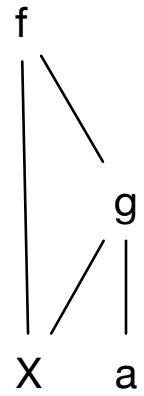
Struktury danych
Struktury a drzewa

"Jan lubi Marię"

Struktury danych
Struktury a drzewa

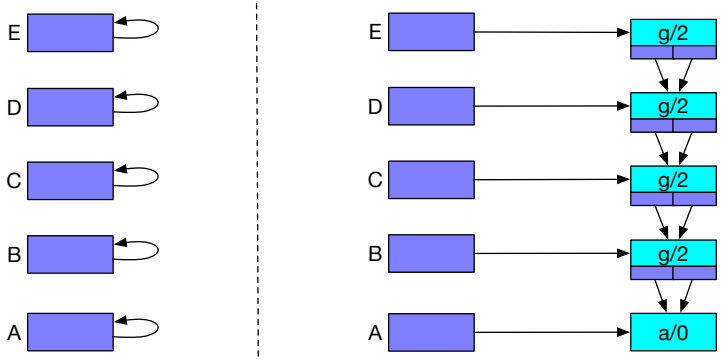
```
f(X, g(X, a))
```

Reprezentacja w postaci DAG (ang. direct acyclic graph):



Struktury danych
Reprezentacja termów

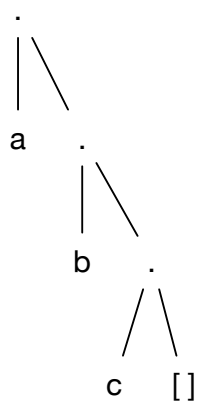
?- f(A, B, C, D, E) = f(a, g(A, A), g(B, B), g(C, C), g(D, D)).
A = a,
B = g(a, a),
C = g(g(a, a), g(a, a)),
D = g(g(g(a, a), g(a, a)), g(g(a, a), g(a, a))),
E = g(g(g(g(a, a), g(a, a)), g(g(a, a), g(a, a))), g(g(g(a, a), g(a, a)), g(g(a, a), g(a, a)))).



Struktury danych
Listy

Funktor kropka łączy głowę listy z jej ogonem.

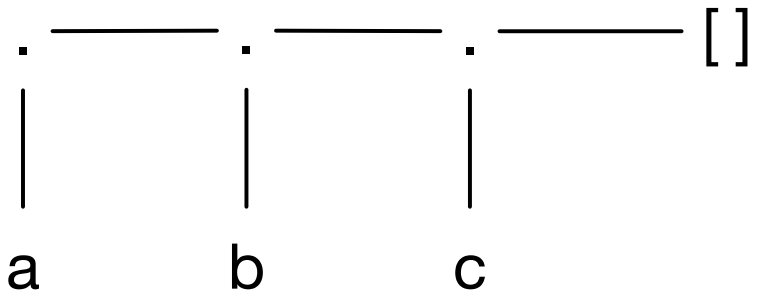
```
.(a, .(b, .(c, [])))
```



Struktury danych
Listy

```
.(a, .(b, .(c, [])))
```

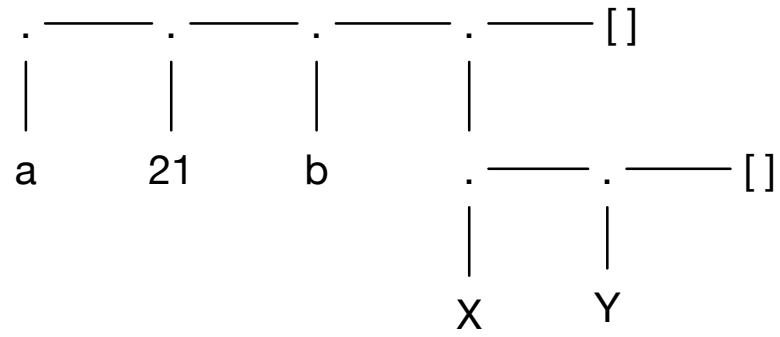
Poziomy zapis listy w postaci „winnej latorośli”:



Struktury danych
Listy

Wygodniej zapisywać elementy listy między kwadratowymi nawiasami oddzielając je przecinkami.

```
[a, 21, b, [X, Y]]
```



Struktury danych
Listy

Przykładowe listy, ich głowy i ogony

Lista	Głowa	Ogon
`[a, b, c]`	`a`	`[b, c]`
`[]`	brak	brak
`[[bury, kot], mruczy]`	`[bury, kot]`	`[mruczy]`
`[bury, [kot, mruczy]]`	`bury`	`[[kot, mruczy]]`
`[bury, [kot, mruczy], cicho]`	`bury`	`[[kot, mruczy], cicho]`
`[X+Y, x+y]`	`X+Y`	`[x+y]`

Struktury danych
Listy

Pionową kreską oddziela się początkowe elementy listy od listy jej pozostałych elementów.

```
p([1, 2, 3]).
p([bury, kot, mruczy, [sobie, pod, nosem]]).

?- p([X | Y]).
X = 1, Y = [2, 3] ;
X = bury, Y = [kot, mruczy, [sobie, pod, nosem]]
?- p([_, _, _, [_ | X]]).
X = [pod, nosem]
```

Struktury danych
Rozkładanie i składanie struktur

Załóżmy, że struktura składa się z funktora `f` i n argumentów $t_1, t_2, \ldots, t_n$:

$$f(t_1, t_2, \ldots, t_n).$$

Za pomocą dwuargumentowego predykatu `=..` można strukturę taką rozkładać na jej składowe elementy:

$$f(t_1, t_2, \ldots, t_n) =.. [f, t_1, t_2, \ldots, t_n]$$

```
?- para(jacek, barbara) =.. X.
X = [para, jacek, barbara].
?- para(jacek, barbara) =.. [Funktor | _].
Funktor = para.
?- para(jacek, barbara) =.. [_ | Argumenty].
Argumenty = [jacek, barbara].
?- para(jacek, barbara) =.. [_, _, DrugiArgument | _].
DrugiArgument = barbara.
?- a =.. [Fun | Args].
Fun = a, Args = [].
```

Struktury danych
Rozkładanie i składanie struktur

Predykat `=..` można również użyć do składania struktur.

```
?- X =.. [f, X].
X = f(X).
?- X =.. [f, Y, Z].
X = f(Y, Z).
?- X =.. [f, a, b].
X = f(a, b).
?- X =.. [F, a, b].
ERROR: Arguments are not sufficiently instantiated
ERROR: In:
ERROR:     [8] _8778=..[_8784,a|...]
ERROR:     [7] <user>
```

Struktury danych
Przeszukiwanie rekurencyjne

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

?- member(d, [a, b, c, d, e, f, g]).
true ;
false.

?- member(2, [3, a, 4, f]).
false.

?- member(X, [a, b, c]).
X = a ;
X = b ;
X = c.
```

Struktury danych

Przeszukiwanie rekurencyjne

```
?- member(a, X).
X = [a|_5758] ;
X = [_5756, a|_5764] ;
X = [_5756, _5762, a|_5770] ;
X = [_5756, _5762, _5768, a|_5776] ;
...
```

Struktury danych

Przeszukiwanie rekurencyjne

Prawda na jeden sposób:

```
true.

?- true.
true.
```

Prawda na nieskończenie wiele sposobów:

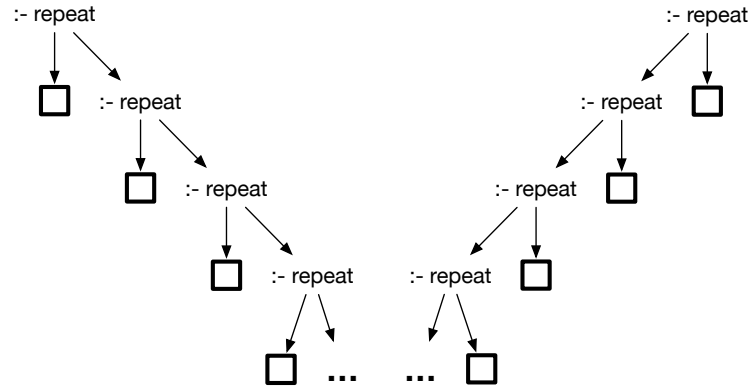
```
repeat.
repeat :- repeat.

?- repeat.
true ;
true ;
true ;
true ;
...      % nieskończenie wiele odpowiedzi twierdzących
```

Struktury danych

Przeszukiwanie rekurencyjne

Istotna jest kolejność klauzul.

```
repeat.
repeat :- repeat.
```




```
repeat :- repeat.
repeat.
```


Struktury danych

Przeszukiwanie rekurencyjne

```
repeat :- repeat.
repeat.

?- repeat.
ERROR: Out of local stack
   Exception: (1,970,845) repeat ? abort
% Execution Aborted
```

Staraj się stosować następującą kolejność klauzul definiujących predykat:

1. fakty
2. reguły, które nie odwołują się do definiowanego predykatu
3. reguły rekurencyjne

Struktury danych
Przeszukiwanie rekurencyjne

```
jest_listą([A | B]) :- jest_listą(B).
jest_listą([]).

?- jest_listą(X).
ERROR: Out of local stack
   Exception: (1,763,388) jest_listą(_G5290152) ? abort
% Execution Aborted
```

Struktury danych
Przeszukiwanie rekurencyjne

```
słaba_jest_listą([]).
słaba_jest_listą([_ | _]).
```

Ta wersja nie wpadnie w nieskończoną pętlę ale przepuści niepoprawne listy:

```
?- słaba_jest_listą([a | b]).
true.
```

Struktury danych
Akumulatory

```
dllisty([], 0).
dllisty([G | O], N) :-
    dllisty(O, N1),
    N is N1+1.
```

Powyższy predykat nie jest w postaci rekurencji ogonowej.

Struktury danych
Akumulatory

```
dllisty2(L, N) :-
    listaakum(L, 0, N).

listaakum([], A, A).
listaakum([G | O], A, N) :-
    A1 is A+1,
    listaakum(O, A1, N).
```

Struktury danych
Akumulatory

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

Struktury danych
Akumulatory

```
[trace]  ?- reverse([1,2, 3], X).
   Call: (7) reverse([1, 2, 3], _G5299633) ? creep
   Call: (8) reverse([2, 3], _G5299717) ? creep
   Call: (9) reverse([3], _G5299717) ? creep
   Call: (10) reverse([], _G5299717) ? creep
   Exit: (10) reverse([], []) ? creep
   Call: (10) lists:append([], [3], _G5299721) ? creep
   Exit: (10) lists:append([], [3], [3]) ? creep
   Exit: (9) reverse([3], [3]) ? creep
   Call: (9) lists:append([3], [2], _G5299724) ? creep
   Exit: (9) lists:append([3], [2], [3, 2]) ? creep
   Exit: (8) reverse([2, 3], [3, 2]) ? creep
   Call: (8) lists:append([3, 2], [1], _G5299633) ? creep
   Exit: (8) lists:append([3, 2], [1], [3, 2, 1]) ? creep
   Exit: (7) reverse([1, 2, 3], [3, 2, 1]) ? creep
X = [3, 2, 1].
```

Struktury danych
Akumulatory

```
reverse(X, Y) :-
    reverse(X, [], Y).

reverse([], S, S).
reverse([X | Y], S, R) :-
    reverse(Y, [X | S], R).
```

Struktury danych
Akumulatory

```
[trace]  ?- reverse([1, 2, 3], X).
   Call: (7) reverse([1, 2, 3], _G1097) ? creep
   Call: (8) reverse([1, 2, 3], [], _G1097) ? creep
   Call: (9) reverse([2, 3], [1], _G1097) ? creep
   Call: (10) reverse([3], [2, 1], _G1097) ? creep
   Call: (11) reverse([], [3, 2, 1], _G1097) ? creep
   Exit: (11) reverse([], [3, 2, 1], [3, 2, 1]) ? creep
   Exit: (10) reverse([3], [2, 1], [3, 2, 1]) ? creep
   Exit: (9) reverse([2, 3], [1], [3, 2, 1]) ? creep
   Exit: (8) reverse([1, 2, 3], [], [3, 2, 1]) ? creep
   Exit: (7) reverse([1, 2, 3], [3, 2, 1]) ? creep
X = [3, 2, 1].
```

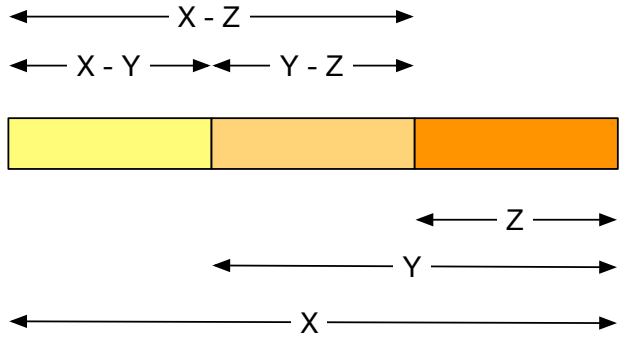
Struktury danych

Listy różnicowe

- Listą różnicową jest struktura danych `L1 - L2`, gdzie `L1` i `L2` są listami.
- Elementami listy różnicowej `L1 - L2` są elementy listy `L1` bez elementów listy `L2`.
- Lista różnicowa `[a, b, c | X] - X` składa się z trzech elementów `a, b, c`.
- Listę pustą reprezentujemy jako `X - X`.

Struktury danych

Listy różnicowe




```
app(X-Y, Y-Z, X-Z).
```

```
?- app([1,2,3|A]-A, [4, 5|B]-B, C).
A = [4, 5|B],
C = [1, 2, 3, 4, 5|B]-B.      % elementy 1, 2, 3, 4, 5
```

Uwaga: tylko łączy a nie umie rozerwać!