Programowanie - notatki

Thrandvil

2009

Spis treści

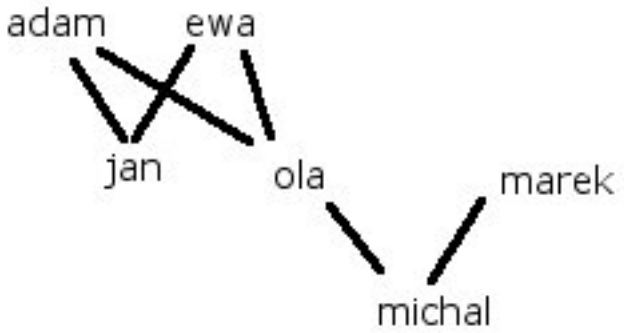
1 26.02 - Prologowe 'Hello World'

‘

1.1 Powiązania rodzinne

Tak, jak dla większości języków imperatywnych pierwszym przykładem jest program drukujący napis 'Hello World', tak w języku Prolog takim przykładem jest program ilustrujący powiązania rodzinne.

W całym przykładzie używamy bardzo prostego modelu rodziny.



Dla opisania powiązań w tej rodzinie użyjemy dwuargumentowego predykatu rodzic (dwuargumentowy predykat możemy zapisać jako rodzic/2). rodzic(a,b) oznacza, że a jest rodzicem b.

```
rodzic(adam, jan).
rodzic(ewa, jan).
rodzic(adam, ola).
rodzic(ewa, ola).
rodzic(ola, michal).
rodzic(marek, michal).
```

Co prawda możemy używać znaków unicode, ale staramy się tego unikać.

Mając tak przygotowaną 'bazę danych' możemy zadać podstawowe pytania:

Możemy zapytać bezpośrednio, czy dwie osoby są w relacji rodzic:

```
? - rodzic(adam,jan).
Yes
? - rodzic(ewa, michal).
No
```

Możemy również użyć zmiennej w miejscu jednego z argumentów. Wtedy Prolog dopasuje wyrażenie:

```
? - rodzic(X, ola).
X = adam
```

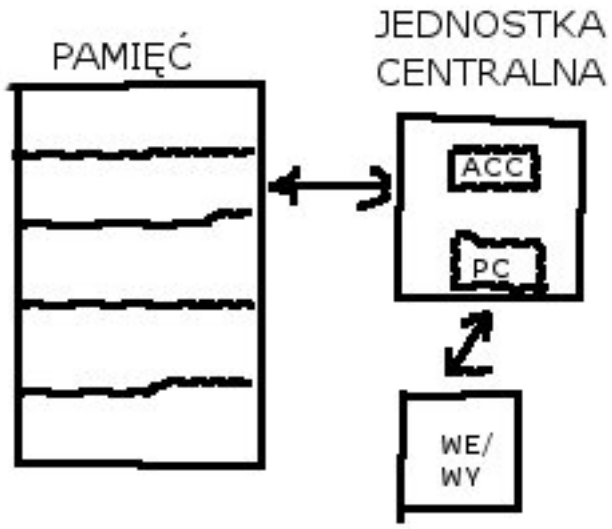
W tym wypadku możemy zaakceptować wynik (przeważnie klawiszem enter), albo kazać Prologowi szukać innego, pasującego faktu. Aby to osiągnąć wciskamy ';'

```
X = adam;
X = ewa;
No
```

Gdy odrzucimy wszystkie odpowiedzi, Prolog odpowie 'No'. Ponieważ nie może dobrać nowego wyniku.

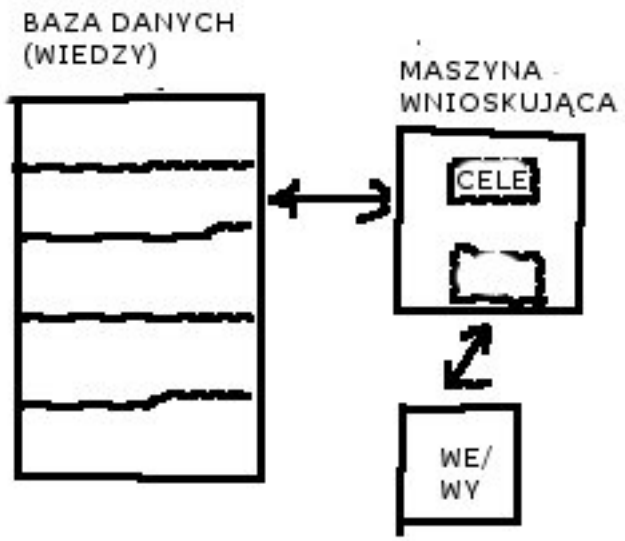
1.2 Jak to działa?

Przeważnie rozpatrujemy komputer według modelu przedstawionego na rysunku:



Tak przedstawiony komputer składa się z pamięci, jednostki centralnej (nie, to nie jest monitor!) oraz urządzeń wejścia/wyjścia. Wewnątrz jednostki centralnej znajdują się rejestry (jak znany z maszyny RAM akumulator, albo rejestr licznika programu wskazujący miejsce w pamięci).

W podobny sposób można przedstawić 'maszynę' Prologu, która przypomina w pewnym stopniu bazę danych.

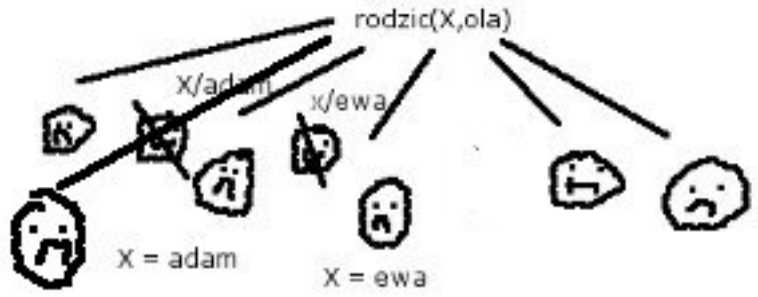


Wewnątrz bazy danych znajdują się klauzule, na których operuje maszyna wnioskująca. Klauzulą może być predykat o dowolnej liczbie argumentów (w tym 0 argumentowy - fakt).

Po otrzymaniu zapytania maszyna wnioskująca jako cel przyjmuje treść zapytania i stara się go spełnić korzystając z bazy wiedzy. Istotne jest, że fakty przeglądane są w podanej kolejności (z góry na dół). Maszyna wnioskująca zapamiętuje również miejsce w 'pamięci' w którym się aktualnie znajduje. Sposób, w jaki przeszukiwana jest baza wiedzy, można zobrazować na drzewie poszukiwań Prologu.

1.3 Proste drzewo poszukiwań

Przykładowe drzewo poszukiwań dla zapytania rodzic(X, ola) wygląda następująco:



Sprawdzane jest wszystkie 6 faktów w bazie wiedzy. Jeśli fakt nie pasuje do zapytania, Prolog natychmiast wraca do zapytania starając się spełnić je przy pomocy następnego w kolejności faktu. Jeżeli fakt pasuje do zapytania, Prolog kończy poszukiwania i wypisuje ukonkretnione wartości zmiennych z pierwotnego celu. Wciśnięcie klawisza ';' zmusza Prolog do odrzucenia znalezionego rozwiązania i dalszego przeszukiwania bazy wiedzy.

Proces powrotu do poprzedniego celu, gdy fakt nie pasuje do pytania (lub użytkownik odrzuci odpowiedź), nazywamy nawracaniem. W trakcie nawracania wszystkie zmienne, które zostały ukonkretnione w danej gałęzi, tracą swoją wartość i mogą zostać ukonkretnione innymi.

1.4 Bardziej skomplikowany predykat - potomek/2

Prolog pozwala oczywiście na konstruowanie zdecydowanie bardziej skomplikowanych programów niż ten pokazany wyżej, w którym przeszukiwana jest lista faktów podanych wcześniej przez użytkownika.

Sposób w jaki powstają bardziej skomplikowane predykaty możemy prześledzić na przykładzie predykatu potomek/2, który dla potomek(a, b) jest prawdziwy, jeśli a jest potomkiem b.

Warunek bycia potomkiem możemy zapisać słownie następująco:
Jeżeli X jest rodzicem Y to Y jest potomkiem X
Jeżeli X jest rodzicem Z, a Y jest potomkiem Z to Y jest potomkiem X

Możemy łatwo stworzyć predykat mówiący o byciu potomkiem przy pomocy wcześniej utworzonej bazy wiedzy.

W Prologu przyjęło się następującą kolejność zapisu:

potomek(Y,X) $\Leftarrow$ rodzic(X,Y)

Dodatkowo, z powodów praktycznych, rezygnuje się z zapisu $\Leftarrow$ i predykat języka Prolog wygląda następująco:

```
potomek(Y,X):- rodzic(X,Y).
potomek(Y,X):-
        rodzic(X,Z),
        potomek(Y,Z).
```

W tym zapisie koniunkcja jest przedstawiona przy pomocy znaku ','.

Jak widać, taki zapis jest dużo czytelniejszy, niż taki, który mielibyśmy utworzyć w języku imperatywnym (nie wymaga od nas dokładnego podania sposobu poszukiwania - podajemy tylko warunki które musi spełnić Y, aby być potomkiem X).

2 2.03 - Listy

2.1 Jeszcze o składni Prologa

Wracając na chwilę do przykładu definicji potomka.

Stwierdzenie o byciu potomkiem możemy zapisać dokładnie przy pomocy formuły logicznej jako:

$$\forall x, y[(\exists z(rodzic(y, z) \land potomek(x, z))) \Rightarrow potomek(x, y)]$$

Możemy wyciągnąć przed nawias kwantyfikator szczegółowy zmieniając go na ogólny (bo wyciągamy z implikacji - zaprzeczenie).

$$\forall x, y, z[(rodzic(y, z) \land potomek(x, z)) \Rightarrow potomek(x, y)]$$

Usunąć nawiasy przed implikacją (bo koniunkcja łączy silniej).

$$\forall x, y, z[rodzic(y, z) \land potomek(x, z) \Rightarrow potomek(x, y)]$$

Oraz (jak zwykliśmy to w logice robić) usunąć kwantyfikator ogólny - ponieważ prawdziwość wyrażenia dla wszystkich zmiennych jest w pewien sposób domyślna.

$$rodzic(y, z) \land potomek(x, z) \Rightarrow potomek(x, y)$$

Tak zapisane wyrażenie (w formie znanej z logiki) może już być naturalnie zapisane jako predykat Prologu:

```
potomek(Y,X):-
        rodzic(X,Z),
        potomek(Y,Z).
```

2.2 Klauzule hornowskie (poziom rozszerzony)

Klauzulą nazywamy:

$$\bigvee_{j=1}^{m} l_j j = 1, ..., m$$

gdzie l_j są literałami.

Czyli inaczej mówiąc klauzulą nazywamy alternatywę literałów.

Typową klauzulę możemy zapisać jako

$$l_1 \lor l_2 \lor l_3 \lor ... \lor l_n \lor \neg k_1 \lor \neg k_2 \lor \neg k_3 \lor ... \lor \neg k_m$$

negacje możemy przepisać jako implikację.

$$l_1 \lor l_2 \lor l_3, ..., \lor l_n \Leftarrow k_1 \land k_2 \land k_3 \land ... \land k_m$$

$\lor$ zamieniła się na $\land$ po prawej stronie, ponieważ wyłączyliśmy przed całość negację.

Taką składnią ciężko się operuje, ponieważ w momencie, gdy ustalimy, że prawa strona jest prawdziwa, daje nam to jedynie informacje o prawdziwości

jednej z alternatyw. W związku z tym wolimy używać klauzul, które po prawej stronie mają tylko jeden literał.

Takie klauzule, które mają co najwyżej jeden niezanegowany literał (w praktyce po prawej stronie), nazywamy klauzulami hornowskimi. Zgodnie z definicją przyjmują one postać:

$$l \Leftarrow k_1 \wedge k_2 \wedge ... \wedge k_m \bot \Leftarrow k_1 \wedge k_2 \wedge ... \wedge k_m$$

Warto zauważyć, że nie wszystkie fakty da się przedstawić przy pomocy klauzul hornowskich, ale znaczna część faktów może być przedstawiona w tej postaci.

2.3 Działania na liczbach - krótka informacja

Prolog jest tak skonstruowany, że otrzymując zapytanie:

```
?- 1 + 2 = 3.
```

odpowie:

```
No.
```

Co będzie efektem próby unifikacji +(1,2) z 3, która naturalnie się nie powiedzie. Aby zmusić Prolog do wykonywania obliczeń używamy innego predykatu:

```
?- 1 + 2 is 3.
Yes.
```

Ale więcej o tym w późniejszym terminie.

2.4 Struktury

Strukturą w Prologu nazywamy term nad sygnaturą złożoną ze stałych (atomów), zmiennych oraz symboli funkcyjnych (funktorów - identyfikatorów pisanych małymi literami).

Przy pomocy struktur można w unikatowy sposób opisywać nawet nieskończenie wiele obiektów. Jako przykład możemy rozpatrzyć strukturę, która mogłaby opisywać każdego urodzonego na Ziemi człowieka. Struktura ta składałaby się z liczby naturalnej - mówiącej którym z kolei dzieckiem swoich rodziców jest człowiek, oraz matki i ojca. Przykładem takiego zapisu może być:

```
dziecko(1,adam,ewa) %kain
dziecko(2,adam,ewa) %abel
dziecko(1,dziecko(1,adam,ewa), ewa) %bardzo dziwny twór ;)
```

2.5 Listy

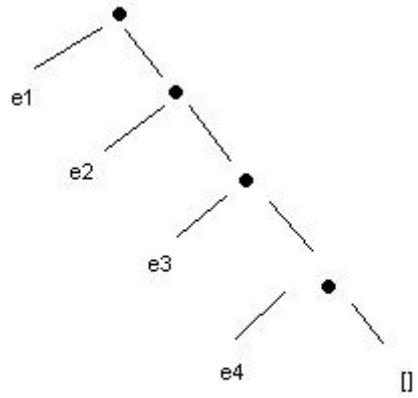
Listą nazywamy strukturę danych składającą się z nieokreślonej liczby elementów, w której mamy dostęp jedynie do pierwszego elementu (tzw. głowy/head) oraz pozostałych elementów (bez głowy/ogon/tail).

W Prologu listy zaimplementowane są przy pomocy dwóch wbudowanych i charakterystycznych predykatów: []/0 oraz ./2.

Prostą listę możemy zapisać jako:

```
.(e1, .(e2, .(e3, .(e4, [])))).
```

i przedstawić przy pomocy drzewa.



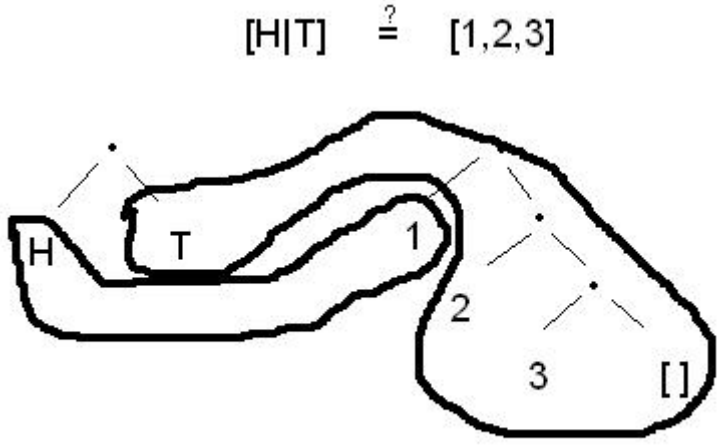
Można to drzewo zapisać również jako:

```
[e1,e2,e3,e4].
```

co jest jedynie cukrem syntaktycznym odpowiadającym wcześniejszemu wyrażeniu.

Aby otrzymać pierwszy element, możemy zunifikować listę z

```
[H|T]
%czyli tak naprawdę
.(H,T)
```



Korzystając z notacji — możemy zapisać predykat mówiący o tym, że jakiś element jest głową listy:

```
head(H, [H|_]).
```

Gdzie _ oznacza nienazwaną zmienną - informujemy Prolog, że ta część wyniku nas nie interesuje.

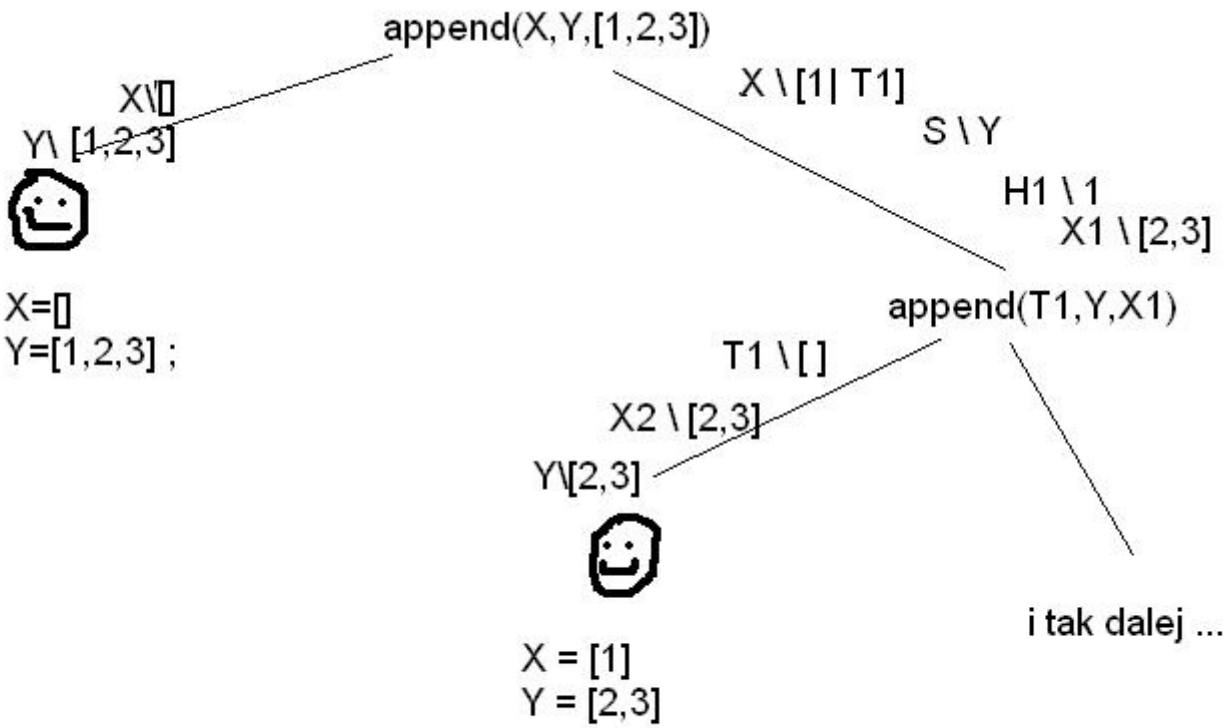
Aby połączyć ze sobą dwie listy, musimy 'przykleić' pierwszą listę na początek drugiej (struktury danych w Prologu nie mogą być zmienione po zdefiniowaniu), w ten sposób tworząc nową listę. Do tego zadania służy (wbudowany) predykat append/3:

```
append([],X,X).
append([H|T],S,[H|X]) :-
        append(T,S,X).
```

Przykładowe wywołanie:

```
append(X,Y,[1,2,3]).
```

można jak zwykle przedstawić w formie drzewa.


append(X,Y,[1,2,3])
X\[]
Y\ [1,2,3]
X=[]
Y=[1,2,3] ;
X \ [1| T1]
S \ Y
H1 \ 1
X1 \ [2,3]
append(T1,Y,X1)
T1 \ [ ]
X2 \ [2,3]
Y\[2,3]
X = [1]
Y = [2,3]
i tak dalej ...

3 4.03 5.03 - Struktury otwarte, odcięcia, rekursja ogonowa

3.1 Garbage collector

Pierwsza część wykładu dotyczyła (między innymi) Garbage Collectora - jednak z uwagi na lenistwo autora notatek pozostaje jedynie artykuł w Wikipedii : http://en.wikipedia.org/wiki/Garbage_collection_(computer_science).

3.2 Struktury otwarte

Strukturą otwartą nazywamy listę (dla przykładu - może to być też inna struktura danych), której część jest nieukonkretnioną zmienną. Przykładem takiej struktury może być

```
[a,b,c\|X]
```

W odróżnieniu od struktury zamkniętej, ta może się zmienić i można dołączyć element na jej koniec bez dużych nakładów związanych z kopiowaniem.

Listę otwartą możemy również zdefiniować jako:

```
L([1,2,3|X], X)
```

Co daje nam łatwy dostęp do elementów struktury.

Używając takiego zapisu możemy zdefiniować w Prologu kolejkę razem z podstawowymi metodami:

```
pusta(l(X,X)).
glowa(E, l([E|_], _).
ogon(l(T,X), l([_|T, X])).
na_poczatek(E, l(L,X), l([E|L], X)).
```

3.3 Operator

Listę otwartą zapisujemy też często jako :

```
[1,2|X] -- X
```

Taki dziwny zapis wynika z definicji – jako operatora. Definicję taką przeprowadzamy przy pomocy wbudowanego predykatu op/3

```
op(--,300,xfy).
```

Pierwszy argument to nazwa operatora, drugi to priorytet, natomiast trzeci to miejsce, w którym ma być umieszczony operator w wyrażeniu.

3.4 Wbudowane predykaty

Do tej pory poznaliśmy kilka wbudowanych predykatów. Tutaj zebrane są wszystkie razem z ich definicjami w Prologu.

```
%x/2
%unifikacja
X=X.
```

```
%fail/0
%zawsze zawodzi

%brak predykatów - wiec żaden nie jest znaleziony
```

```
%true/0
%sukces
true.
```

```
%repeat/0
%zawsze spełniony - niezależnie od liczby nawrotów.
repeat.
repeat :-
        repeat.
```

Dostępne jest również kilka predykatów, które służą do obsługi wejścia i wyjścia:

read/1 - unifikacja argumentu z termem z wejścia

write/1 - wypisuje term na wyjście

nl/0 - wypisuje znak nowej linii na wyjście

3.5 Dalej o odcięciach

Mechanizm odcięć można porównać do znanej z języka C instrukcji longjump. Sposób zapamiętywania swojego rodzica w drzewie poszukiwań może być utożsamiany z zapisywaniem stanu stosu, który jest później odczytywany odrzucając zmiany które się w międzyczasie pojawiły.

Podobnie jak wspomniany mechanizm longjump, odcięcia mają olbrzymi potencjał, ale jednocześnie bardzo łatwo ich użyć niepoprawnie. Znacznie zmniejszają także czytelność programu. W związku z tym należy używać ich z wielką uwagą.

Odcięcia dzielimy na dwa rodzaje:

Odcięcia zielone - takie, które swoim działaniem usuwają gałąź drzewa poszukiwań w której wiemy, że Prolog nie odnajdzie żadnego sukcesu. Takie odcięcia nie zmieniają odpowiedzi które udziela Prolog, a jedynie przyspieszają działanie programu. Ich cechą charakterystyczną jest to, że czytając program możemy całkiem je ignorować, a mimo to analiza programu będzie poprawna.

Odcięcia czerwone - mają wpływ na sposób, w jaki Prolog udziela odpowiedzi. Są pomocne w pisaniu programów, ale mogą znacznie utrudnić czytanie kodu. Należy używać ich z podwojoną ostrożnością, jedynie w momentach gdy wiemy co robimy (i będziemy wiedzieć to również po 2 godzinach czytając ponownie kod). Często używamy ich do odcięcia gałęzi drzewa poszukiwań, w której znajduje się nieskończenie wiele porażek (czyli takiej, w której Prolog się zapętla).

Sposób użycia odcięć możemy prześledzić na przykładzie predykatu, który mając daną listę dzieli ją na podlisty o elementach ujemnych i nieujemnych.

Kod bez odcięć wygląda następująco:

```
split([],[],[]).
split([H|T],[H|X],Y) :-
        H < 0,
        split(T,X,Y).
split([H|T],X,[H|Y]) :-
        H >= 0,
        split(T,X,Y).
```

Predykat porównuje kolejne głowy listy z 0 i łączy je z odpowiednimi listami wywołując się rekurencyjnie - po chwili zastanowienia działanie programu staje się oczywiste.

Warto zauważyć, że w momencie, gdy stwierdzamy, iż H ¡ 0, w drugiej klauzuli wiemy już, że przy ewentualnym nawrocie i przejściu do 3 klauzuli na pewno nastąpi porażka - zawiedzie cel H ¿= 0. W związku z tym po H ¡ 0 w 2 klauzuli możemy uznać, że nawracanie nie ma sensu. W ten sposób odcinamy gałąź drzewa poszukiwań, w której są same porażki - używamy zielonego odcięcia.

```
split([],[],[]).
split([H|T],[H|X],Y) :-
        H < 0, !,
        split(T,X,Y).
split([H|T],X,[H|Y]) :-
        H >= 0,
        split(T,X,Y).
```

Jednak skoro zastosowaliśmy odcięcie, wiemy, że gdy H ¡ 0, nie nastąpi już nawrót dla tego elementu. W związku z tym przez 2 klauzulę przejdzie jedynie taki element który nie jest mniejszy od 0. Wiedząc o tym, możemy w 3 klauzuli nie sprawdzać warunku H ¿= 0. Definicja nadal jest poprawna, ale wcześniej dodane odcięcie staje się czerwone - jego usunięcie doprowadziłoby do katastrofy i nieprawidłowej pracy programu.

```
split([],[],[]).
split([H|T],[H|X],Y) :-
        H < 0, !,
        split(T,X,Y).
split([H|T],X,[H|Y]) :-
        split(T,X,Y).
```

3.6 Rekursja ogonowa

Rekursją ogonową jest nazywana taka rekursja, w której nie ma potrzeby powrotu do funkcji po wywołaniu rekurencyjnym. Tak zapisane funkcje mogą zostać zoptymalizowane przez kompilator do postaci iteracyjnej.

Aby zobaczyć jak w praktyce może działać rekursja ogonowa, zastanówmy się jeszcze raz nad bardzo częstym problemem funkcji liczącej silnię. (wyjątkowo, dla czytelności przykładu, będziemy posługiwać się językiem C)

Iteracyjnie możemy zapisać funkcję liczącą silnię jako:

```
int silnia(int x)
{
        int s = 1;
        while(x)
                s *= x--;
        return s;
}
```

Tak zapisany program jest bardzo mało czytelny, ale za to działa najszybciej jak jest to możliwe.

Pierwszym, co przyjdzie na myśl, gdy rozważamy rekurencyjną definicję silni, może być:

```
int silnia(int x)
{
        if(x < 1)
                return 1;
        else
                return x*silnia(x-1);
}
```

Jednak taka definicja silni jest zła. Rekursja nie jest ogonowa (wykonywane najpierw jest wywołanie rekurencyjne, a dopiero później mnożenie), więc nie zostanie zoptymalizowana przez kompilator.

Rekursję ogonową możemy uzyskać rozwiązując bardziej ogólne zadanie (i wprowadzając przy okazji do funkcji akumulator). Funkcja licząca wartość s * x! może mieć postać:

```
int silnia(int x, int s)
{
        return (x <= 1) ? 1 : silnia(x-1, s*x);
}
```

Tym sposobem wartość s jest liczona przed wywołaniem rekurencyjnym i kompilator może zamienić kod do postaci iteracyjnej.

Rekursja ogonowa ma bardzo duże znaczenie w językach deklaratywnych, gdzie rekurencja jest jedynym sposobem na uzyskanie iteracji i ważne jest, aby nie tracić na niej prędkości działania programu.

3.7 Arytmetyka

Termy składające się z funktorów arytmetycznych (+ , - , * , / , // (dzielenie z wynikiem całkowitym) , sin , mod ...) oraz literałów całkowitych nazywamy termami arytmetycznymi.

Znaczenie arytmetyczne ma term 1 + 2, ale już nie możemy użyć dowolnego innego termu, czyli znaczenia arytmetycznego pozbawione jest wyrażenie: ala + 1. Dodatkowo, używane termy muszą być ukonkretnione, w związku z tym niepoprawne jest również 1 + X.

Żeby przypisać wartość wyrażenia arytmetycznego do zmiennej, zdefiniowany jest wbudowany predykat (infixowy operator) is/2. Po lewej stronie może występować dowolny term (także zmienna), a po prawej term arytmetyczny.

Sposób działania predykatu is/2 możemy zobaczyć na przykładach:

```
5 is 1 + 4.
Yes.

3 is 1 + 4.
No.

Y is 1 + 4.
Y = 5.

5 is 1 + Y.
ERROR: is/2: Arguments are not sufficiently instantiated.
```

Warto zwrócić uwagę na ostatni przykład, w którym Prolog informuje nas o błędzie (co powoduje natychmiastowe przerwanie pracy programu).

Do porównywania wartości korzystamy z wbudowanych predykatów </2, >/2, >=/2, =</2 oraz ==, =:= itd.

Przy pomocy predykatów arytmetycznych możemy zdefiniować predykat length, który podaje nam długość listy zadanej jako argument (akumulator jest niezbędny do osiągnięcia rekursji ogonowej):

```
length(L,N) :-
        length(L,0,N).
length([],A,A).
length([_|T],A,N) :-
        A1 is A + 1,
        length(T,A1,N).
```

3.8 Negacja

Do zapisania negacji w Prologu możemy używać wbudowanego predykatu \+, który mógłby być zdefiniowany jako:

```
\+ p :- p, !, fail.
\+ p.
```

4 25.03 26.03 - Haskell, podstawy

4.1 Listy w Haskellu

Typy danych w Haskellu przypominają te, które znamy z Prologu, w szczególności oznacza to, ze struktury takie jak listy są trwałe.

Podstawową strukturą danych jest lista.

Predykatowi ./2 z Prologu odpowiada funkcja (:):: a -¿ [a] -¿[a]. (Wyrażenie po czterech kropkach to typ funkcji. W tym przypadku mówi nam, że funkcja bierze jako argument element jakiegoś typu i listę elementów tego samego typu zwracając listę elementów tego samego typu). Natomiast []/0 odpowiada []::[a] - lista pusta traktowana jako lista dowolnego typu.

Listy w Prologu oraz Haskellu mają dwie różnice. Po pierwsze w Haskellu nie ma struktur otwartych, a po drugie listy w Haskellu składają się jedynie z elementów jednego typu. Znaczy to, że dozwolona w Prologu lista postaci [1, 2, [1,2]] w Haskellu zostanie potraktowana jako błąd (liczba całkowita i lista liczb całkowitych to całkiem różne typy).

4.2 Operacje na listach

Biblioteka standardowa Haskella dostarcza nam bardzo dużo narzędzi do operacji na listach, wśród nich:

(++)::[a]-¿[a]-¿[a] odpowiadający predykatowi append z Prologu. Funkcja taka może być zaimplementowana w następujący sposób:

```
[] ++ ys = ys
(x:xs) ++ ys = x:(xs++ys)
```

Implementacja ta jest bardzo podobna do tej, którą napisalibyśmy w Prologu. Jednocześnie warto zauważyć, że w tym przypadku rekursja jest ogonowa. Ponieważ Haskell jest 'leniwy' x:(xs++ys) zostanie zamienione na wywołanie funkcji (:) z parametrami x oraz xs++ys i takie pseudo wywołanie zostanie zwrócone przez funkcję (więc po rekurencyjnym wywołaniu nic już nie robimy) i policzone dopiero w momencie gdy będziemy chcieli wykorzystać jego wynik. Warto zauważyć że odwrotnie niż w Prologu zmienne zaczynamy małymi literami, natomiast typy zaczynać będziemy wielkimi.

Kolejną istotną funkcją wykorzystywaną do wykonywania operacji na listach jest funkcja map :: (a -¿ b) -¿ [a] -¿ [b]. Już w definicji typu widać, że funkcja ta jest 'dziwna'. Jako swój pierwszy argument map przyjmuję funkcję która jako argument bierze element typu a i zwraca element typu b, drugim argumentem funkcji map jest lista elementów typu a, a wynikiem lista elementów typu b. Funkcje które przyjmują inne funkcję jako swoje argumenty nazywamy funkcjami wyższego rzędu. Warto przyzwyczaić się do takich funkcji, bo w Haskellu spotykamy się z nimi na każdym kroku.

Funkcję map można zdefiniować jako

```
map _ [] = []
map g (x:xs) = g x : map g xs
```

_ ma bardzo podobne znaczenie do znanego z Prologu i w tym wypadku mówi nam o przypadku dla dowolnej funkcji.

4.3 Krótko o typach

Jak już wspomniałem w Haskellu każde wyrażenie ma przypisany sobie typ np True ma typ Bool, co zapisujemy True::Bool.

Jeśli chcemy dodać nowy typ danych piszemy (dla przykładu redefiniując typ Bool)

```
data Bool = True | False
```

Jeśli chcielibyśmy zaimplementować podobne do Prologowych relacje rodzinne tutaj również powinniśmy zadbać o typy

```
data Mezczyzna = Adam | Syn Integer Mezczyzna Kobieta
data Kobieta = Ewa | Corka Integer Mezczyzna Kobieta
```

W tym przypadku używamy typów rekurencyjnie do wzajemnej definicji.

Przy odpowiednio zdefiniowanych typach kompilator Haskella pilnuje za nas poprawności programu pod tym względem (w powyższym przypadku pomaga nam to nie definiować istot których pochodzenie nie byłoby możliwy z biologicznego punktu widzenia).

Typy w Haskellu dzielimy na monomorficzne i polimorficzne. Przykładem funkcji która ma monomorficzny typ może być

```
f :: Int -> Char -> Int
f x y = x + ord y
```

Funkcja ta działa na określonych typach (Przyjmuje Int i Char, zwraca Int). (Przy okazji warto zauważyć, że typ Int to całkiem inny typ niż Integer. Int to znany nam 32 (lub 64) bitowa wartość całkowita, natomiast wielkość wartości Integer jest ograniczona jedynie pamięcią naszego komputera, co w praktyce znaczy, że może przechowywać bardzo duże liczby, a w związku z tym operacje na niej bywają bardziej kosztowne. Ale wróćmy do typów polimorficznych ...)

Natomiast przykładem funkcji o polimorficznym typie może być

```
length :: [a] -> Int
length [] = 0
lentth (_:xs) = 1 + length xs
```

Zmienna a w typie funkcji sugeruje, że działa ona dla listy zawierającej elementy dowolnego typu, więc będzie działała równie dobrze dla listy liczb całkowitych, listy znaków czy listy list zawierających listy liczb zespolonych. W każdym z tych przypadków zwróci liczbę elementów (przy czym warto zauważyć, że w ostatnim przypadku będzie to liczba list zawierających listy liczb zespolonych, a zdecydowanie nie liczba liczb zespolonych, które są gdzieś głęboko ukryte)

4.4 Krótko o operatorach

Operatory w Haskellu są traktowane jak normalne funkcje, tyle, że mogą być pisane infixowo. Dla przykładu wywołanie

```
[1] ++ [2,3]
```

Jest równoważne wywołaniu (zapis w nawiasach informuje, że chcemy użyć notacji prefixowej)

```
(++) [1] [2,3]
```

Funkcje pisane przeważnie prefixowo możemy też zapisywać infixowo i zaprezentowane za chwilę wywołania są równoważne

```
mod(1,2)
1 `mod` 2
```

W tym wypadku stosujemy notację zamykającą nazwę funkcji w parze znaków ‘ ‘

Dodatkowo z operatorów (czyli i funkcji możemy tworzyć funkcję częściowe, których jeden z argumentów jest już ustalony.

```
-- funkcja dodająca 5
(5+)
```

Wyjątkiem od reguły jest zapis (-1), który oznacza ujemną wartość, a nie funkcję odejmującą.

4.5 Leniwość

Chociaż ciężko w to uwierzyć Haskell jest bardziej leniwy niż większość studentów, a już na pewno większość powszechnie wykorzystywanych języków programowania.

W językach takich jak C czy Java (takie podejście nazywamy gorliwym) wywołanie

```
funkcja( a , b )
```

Spowoduje najpierw obliczenie wartości wyrażenie a, później wartości wyrażenia b, a dopiero później wywołanie funkcji z obliczonymi wartościami jako argumentami. Haskell działa inaczej. Podobne wywołanie sprawi, że wyrażenia a i b zostaną zapamiętane i obliczone dopiero w momencie, gdy okaże się, że są one naprawdę potrzebne.

Przykładem funkcji w której może to mieć znaczenie jest np

```
int f (int x int y)
{
                if ( y > 0)
                        return y;
                        else return x;
}
```

W przypadku, gdy y ¿ 0 nie interesuje nas w ogóle wartość x, więc nie ma żadnego rozsądnego powodu, żeby ją w ogóle liczyć.

Na dodatek możemy do tego dodać funkcję

```
int g(y)
{
        while(y++>0);
        return 0;
}
```

Funkcja jest określona jedynie dla niedodatnich wartości, w innym przypadku się podrostu zapętla.

Wywołanie

```
h = f(g(y), y)
```

Przy braku leniwości sprawi, że najpierw będą liczone argumenty i funkcja h będzie miala określoną dziedzine jedynie dla wartości niedodatnich. Jednak jeżeli

wyrażenia wywoływane są leniwie wszystko jest w porządku i funkcja określona jest dla każdej wartości.

Ciekawe jest, że z matematycznego punktu widzenia dziedziną tej funkcji jest jedynie zbiór liczb niedodatnich. Jak widać matematyka, w przeciwieństwie do Haskella jest gorliwa.

Leniwość ma jeszcze jedną bardzo istotną zalete. Pozwala nam bez żadnych problemów na budowę nieskończonych struktur danych (np list). Całkiem poprawną definicją jest

```
inf :: Integer -> [Integer]
inf n = n : inf (n + 1)
```

Mogło by się z początku wydawać, że taka funkcja się po prostu zapętli, ale ponieważ Haskell jest leniwy elementy które znajdują się w liście nie muszą być liczone i funkcja zwraca (bardzo szybko !) wynik. Dodatkowo korzystając z wyniku możemy z niego pobrać np 5 pierwszych elementów i tylko one zostaną wygenerowane, dzięki czemu możemy spokojnie pracować na nieskończonych listach.

Bardzo naiwne podejście do leniwości od razu naprowadza na bardzo pesymistyczną myśl 'W takim razie wyrażenia są liczone wiele razy! To jest jeszcze gorsze!'. Spójrzmy na następujący przykład funkcji:

```
f x = x * x
```

Mogło by się wydawać, że x jest liczony dwa razy, co bardzo skutecznie zabiło by wszelką efektywność. Na szczęście problem ten ma bardzo proste rozwiązanie. Zmienna x jest tak na prawdę odwołaniem do komórki pamięci w której przy pierwszym liczeniu zapisywana jest wartość wyrażenia i następne odwołanie jedynie wydostaje z tej komórki wcześniej obliczoną wartość. Takie rozwiązanie nazywamy spamiętywaniem (memorization)

4.6 Haskell a świat rzeczywisty

Haskell jako język czysto deklaratywny stara się być tak oderwany od świata zewnętrznego jak to tylko możliwe. Tak naprawdę jedynie kontakt ze światem zewnętrznym i prezentacja wyników wymaga od nas policzenia jakichkolwiek wyrażeń. Dodatkowo działanie funkcji ma być całkiem niezależne od warunków zewnętrznych i zależeć jedynie od podanych argumentów. W związku z tym komunikacja między Haskellem a światem zewnętrznym odbywa się w dość skomplikowany sposób.

Haskell dostarcza nam typ abstrakcyjny IO(). Funkcję, które mają komunikować się z użytkownikiem zwracają obiekty tego typu, co nie oznacza że komunikacja zachodzi dokładnie w tej chwili. Leniwość sprawia, że nie możemy ustalić momentu w której ona zachodzi.

Jednym ze sposobów na wymuszenie komunikacji jest użycie specjalnej funkcji main. W ten sposób możemy napisać w Haskellu typowe 'Hello World'

```
main :: IO()
main = putStrLn 'Hello World'
```

Chociaż intuicyjnie wiemy jak działa ten program wyjaśnienie tego nie jest trywialne.