PROGRAMOWANIE DEKLARATYWNE

HASKELL (w PIGUŁCE mini, v1.8)

::Paweł Idzikowski

Podstawowe informacje:

- skrót $$ pozwala odwołać się do ostatnio wyznaczonej wartości np.
2+3=5
2 ^ $$ da w wyniku 32 (uwaga na odstępy!!!)

- Funkcje i operatory arytmetyczne możemy stosować w postaci infiksowej lub prefiksowej
postać **prefiksowa**: div x y , (-) 3 2, (+) 6 7
postać **infiksowa**: x `div` y, y `mod` z, 5 * 2 , 5 `mod` 2

- dwa znaki **==** oznaczają równość, np. 2==3 zwróci False bo 2 nie jest równe 3.

- **WAŻNE** – mod jako funkcja ma wyższy priorytet niż jako operator `mod` , przykład:
mod 5 2 ^ 3 = 1 (tu mod użyte jako funkcja)
5 `mod` 2 ^ 3 = 5 (tu `mod` jako operator)

- pamiętaj ,że **Haskell** interpretuje skrajny lewy brzeg notatnika jako „nowa funkcja”

- Należy **uważać** na kolejność wykonywania działań ponieważ w HASKELL operatory mają swoje priorytety(im wyższy tym silniejszy) i kierunek łączności. Przedstawia to poniższa tabelka:
L-lewostronna, P-prawostronna, N-brak łączności

Operator	Działanie	Priorytet	Łączność
<funkcja>	wywołanie funkcji	10	brak
!!	operator indeksowania	9	L
^ , ^^, **	potęgowanie	8	P
* , / , `mod`, `div`	mnożnie, dzielenie, modulo, dzielenie całkowite *(pamiętaj, że jeśli kilka takich operatorów mamy to zgodnie z zasadą: od lewej do prawej)* np. 2 * 3 * 5 / 6	7	L

	no to po kolei		
+ , -	dodawanie, odejmowanie	6	L
: , ++	: - dołączanie do istniejącej listy ++ - konkatenacja list (łączenie list)	5	P
>, <, >=, <=, == , /=	kolejno: większy, mniejszy, większy-równy, mniejszy-równy, równy, różny	4	N
&&	AND (koniunkcja)	3	P
||	OR (alternatywa)	2	P
..	specyfikacja zasięgu listy	brak	brak

- **UWAGA, ważne!** Priorytet **10** jest zarezerwowany dla **funkcji**, zatem wywołanie funkcji ma największy priorytet!!
- oczywiście wiemy, że pierwej działania w nawiasach np. (2*3+1)+(3-2)=(7)+(1)=8

Ćwiczenie

Podaj wyniki poniższych wyrażeń korzystając z tabelki powyżej a następnie sprawdź w HASKELL:

10 `mod` 6 `mod` 4 = ???

(10 `mod` 6) `mod` 4 = ???

10 `mod` (6 `mod` 4) = ???

16 `div` 3 `div` 2 = ???

(16 `div` 3) `div` 2 = ???

16 `div` (3 `div` 2) = ???

-- tak oznaczamy komentarz jednowierszowy

{- .... -} tak oznaczamy komentarz wielowierszowy

Zapis taki ogólny funkcji w **HASKELL** możemy przedstawić tak:

nazwa_funkcji argument(y) = definicja funkcji

Przykładowo funkcje **f(x) = x + 3** zapiszemy w HASKELL tak: **f x = x + 3**

funkcja kwadrat mogłaby wyglądać tak: kwadrat x = x * x

Typy jakimi dysponujemy w **HASKELL** to:

Całkowite:
- Int
- Integer

Rzeczywiste:
- Float
- Double

Znakowy:
- Char

Boolowski (prawda/fałsz)
- Bool (należy pamiętać, że zapisujemy False / True – pierwsza litera duża!!!)

- **uwaga:** Typ **string** to tak naprawdę lista znaków czyli zapisujemy ją tak: **[Char]**

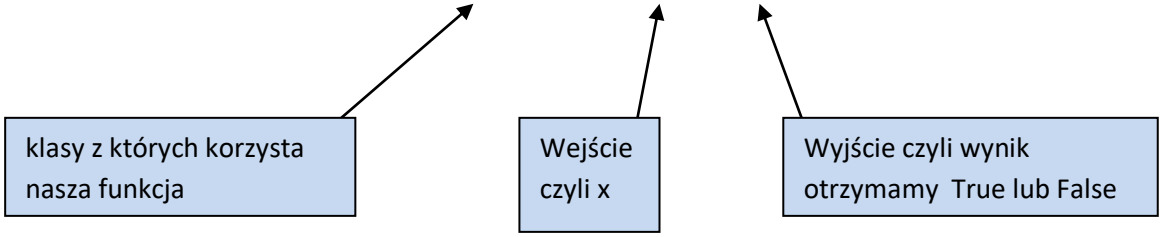
Poszczególne typy są przechowywane przez tzw. **klasy**
Klasa Eq – gdy użyjemy operatora == lub /= w naszej funkcji
Klasa Ord – gdy użyjemy operatora > , < , <= , >= w naszej funkcji
Klasa Num – (od Numeric) gdy wykonujemy operacje na liczbach
Klasa Floating – gdy wykonujemy operacje na liczbach rzeczywistych (wtedy nie musimy dawać **Num**) gdy mamy np. funkcje z dzieleniem i sinusem to należy pamiętać, że dzielenie wykonywane jest na liczbach wymiernych **(fractional)** a sinus na rzeczywistych zatem możemy użyć albo Double albo Float czyli wartość musi pochodzić z klasy: **Floating**
Klasa Enum – gdy brany jest następnik, poprzednik
Klasa Integral – należy do niej typ danych Int i typ danych Integer,
UWAGA: do klasy Integral należą np. funkcje even i odd, GDY taka funkcja będzie użyta to w typie dowolnym musimy napisać Integral a nie Num !!

Należy zwrócić uwagę, że każda funkcja, operator ma tzw. „**typ ogólny**" , czyli taki, który informuje nas o tym z jakich klas korzysta, co dajemy na wejście oraz co otrzymujemy na wyjściu(wynik).
Przykładowo typ ogólny dla funkcji kwadrat x = x * x wygląda następująco:

```
kwadrat :: Num a => a -> a
```

inny przykład **typu ogólnego** dla funkcji czy_dodatnie:

```
czy_dodatnie :: (Num a, Ord a) => a -> Bool
```



funkcja **czy_dodatnie** wygląda następująco:

```
kwadrat x | x * x > 0 = True
          | x == 0 = False
```

My możemy zdefiniować dla funkcji tzw. „**typ konkretny**”, czyli taki, który wymusza użycie konkretnego rodzaju danych np. możemy zrobić aby nasza funkcja przyjmowała tylko typ Int, po podaniu innego typu funkcja zwróci błąd. W typie konkretnym nie piszemy klas z których korzystamy.

Po zapisaniu typu konkretnego możemy uruchomić Haskell i sprawdzić czy go „zaakceptował”. Jeżeli pojawił się błąd to znaczy, że źle zdefiniowaliśmy nasz typ konkretny.

```
kwadrat :: Integer -> Integer
```

przećwiczmy podanie **typu ogólnego**:
mamy funkcję f x = x + 3 + 6
to nasz **typ ogólny** będzie wyglądał następująco: **f :: Num a => a -> a**

mamy funkcję f xs = map sqrt xs
to nasz **typ ogólny** będzie taki: **f :: Floating a => [a] -> [a]**

mamy funkcję wykorzystując dopasowanie do wzorca:
lub1 False True = True
lub1 False False = False
to nasz **typ ogólny** będzie taki: **lub1 :: Bool -> Bool -> Bool**

przećwiczmy teraz podanie **typu konkretnego**:
mamy funkcję f x = x + 3 + 6
to skoro ta funkcja działa na liczbach i nie mamy tu ani dzielenia ani funkcji trygonometrycznych to możemy skorzystać z typów Int,Integer,Float lub Double
zatem nasz **typ konkretny** będzie taki: **f :: Int -> Int**
lub np. taki może być **f :: Integer -> Integer**

mamy funkcję f xs = map sqrt xs
zauważ, że powyższa funkcja ma zmienną xs – oznacza ona listę, w wyniku też otrzymamy listę… przykładowy **typ konkretny** naszej funkcji może wyglądać tak:
f :: [Float] -> [Float] czyli podaliśmy listę i otrzymujemy w wyniku listę…

uwaga do powyższego typu: jeżeli sprawdzisz sobie jak działa sqrt, np. podamy w Haskell sqrt 3 to wynik będzie liczbą po przecinku – wniosek? Nie możemy tutaj skorzystać z liczb całkowitych(czyli Int lub Integer) bo otrzymalibyśmy błąd tuż po włączeniu skryptu.

UWAGA UWAGA UWAGA: Przy podawaniu typu konkretnego musimy uważać na to jaki typ danych, może być użyty w naszej funkcji zarówno na wejściu jak i wyjściu.

Warto pamiętać także: Haskell dysponuje wbudowaną funkcją: **compare**, która pozwala ocenić czy liczba jest mniejsza**(LT),** większa**(GT)** czy równa**(EQ)** od podanej drugiej liczby np.
compare 2 3 zwróci LT bo przecież 2 < 3
LT = Less Than czyli odpowiada to symbolowi <
GT = Greather Than czyli symbol >
EQ = Equals czyli symbol ==

Uwaga, gdy podajemy liczby np. 5 , 6 dla Haskella są one tzw. **dowolnego typu numerycznego** zatem np. polecenie sqrt 5 zadziała, gdyby jednak zdefiniować rozmiar = 5 to gdy napiszemy sqrt rozmiar to Haskell zwróci błąd, że sqrt chce liczbe z klasy Floating (czyli Float lub Double)

Definicje lokalne:

Definicje lokalne możemy powiedzieć, że są to definicje działające tylko w obrębie funkcji, w których je zdefiniowaliśmy, dla pozostałej części naszego skryptu są niewidoczne. Mamy dwa sposoby użycia definicji lokalnych. Możemy użyć definicji **where** lub **let…in…**

1) Konstrukcja z **where**:
   Słownie: najpierw podajemy normalnie nazwe funkcji, argumenty i działanie a potem (najlepiej w nowej linii) umieszczamy słówko where… w następnych liniach umieszczamy funkcje bądź stałe, które są wykorzystywane w naszym działaniu:

   np.

```
dodaj_odejmij x = x -  a + b
                       where
                             a = 5
                             b = 6
```

   Należy zwrócić uwagę, że to co definiujemy po ‘where’ musi być w jednej kolumnie!!!

możemy też po where zdefiniować kolejną funkcję, ale należy pamiętać, że jeśli chcemy wywołać w funkcji inną funkcję to nie możemy zapomnieć o przekazaniu jej argumentów!! (robimy to przemyślanie). Pokazane to zostało na poniższym przykładzie:

```
dodaj_ip x z = x * funkcja z
                    where
                        funkcja z = z*z+z
```

Warto też pamiętać o tym, że jeżeli chcemy taki wynik wyrażenia dodatkowo np. przemnożyć przez 2 to musimy otoczyć nawiasami tylko część „**definicji funkcji**" a nie całość!!! (to znaczy – bez where) dla powyższej funkcji dodaj_ip wyglądałoby to tak:

```
dodaj_ip x z = (x * funkcja z)*2
                    where
                        funkcja z = z*z+z
```

2) Do dyspozycji mamy również konstrukcję **let….in…** różni się między innymi tym od where, że jest bardziej naturalne od where no bo najpierw podajemy tak jakby dane a potem wyrażenie.. w where zauważ jest na odwrót. Należy pamiętać, że definiowane funkcje muszą być w jednej kolumnie! Let i in nie musi ale warto zrobić w tej samej żeby było przejrzyście…

przykładowo:

```
ob_kuli4 r =
        let a=4/3
            sz r = r*r*r
            funk x = x^2
         in  a*pi*sz r*funk x
```

Gdybyśmy chcieli naszą powyższą funkcje dodatkowo np. przemnożyć przez 2 to musielibyśmy objąć w nawias całość począwszy od let a kończąc na x czyli tak:

```
ob_kuli4 r =
        (let a=4/3
            sz r = r*r*r
            funk x = x^2
         in  a*pi*sz r*funk x)*2
```

Należy zapamiętać, że zmienne lub funkcje, które definiujemy lokalnie np. po słówku where lub po słówku let są **LOKALNE** zatem nie będzie ich widać **NA ZEWNĄTRZ**!!!

Żeby to zobrazować, załóżmy, że teraz chcemy napisać funkcję taką: **f x = 2 * a + 3** , w tym samym notatniku co powyższa funkcja **ob_kuli4** , no to jak uruchomimy ten notatnik to dostaniemy błąd ponieważ Haskell nie wie co ma podstawić za **a** w naszej funkcji:
f x = 2 * a + 3 ..pod a nie zostanie podstawione 4/3 z ob_kuli4 ponieważ a=4/3 jest zdefiniowane lokalnie i jest to widoczne tylko i wyłącznie w funkcji ob_kuli4

Kilka informacji istotnych do pamiętania:
-1) dzielenie (znakiem /) oczekuje liczby Fractional(wymiernej)
-2) funkcje trygonometryczne np. sinus oczekują liczby Floating(zmiennoprzecinkowej) czyli Double lub Float
-3) Floating to klasa, a Float to typ
-4) Integral to klasa, a Integer to typ

Przykładowy **typ ogólny**:
-- f :: Floating a => a->a->a

Wybieramy jedną z 3 klas jeśli pracujemy na liczbach – **Num** lub **Floating** lub **Integral**
Klasa Floating(typ Float, typ Double) –funkcje trygonometryczne, sqrt itd.
Klasa Integral(typ Int, typ Integer) – gdy mamy funkcje even, odd, div, mod (TU div, mod to funkcje!!)
Klasa Num – gdy możemy swobodnie pracować i na liczbach całkowitych i na liczbach przecinkowych

Wybieramy **Ord** jeśli w naszej funkcji pojawiło się: > , < , >=, <=

Przykładowy **typ konkretny**:
W typie konkretnym nie podajemy klas!!!
-- f :: Float->Float->Double

Definicje warunkowe:

Definicja if….then….else

konstrukcja definicji:

if <warunek> then <true-value> else <false-value>

Uwagi:

nie ma konstrukcji if ... then!!!

warto miedzy warunkami lamac linie

if..then..else gdy jeden warunek mamy to dośc fajnie się przydaje ☺

przykład:

```
znak1 x = if x < 0 then (-1) else
          if x == 0 then 0 else 1
```

Definicja guard(strażnika)

*znakiem | oznaczamy straznika

np. znak2 x | x < 0 oznacza „przepusc x jesli x < 0”

* konstrukcja strażnika: nazwa_funkcji argumenty | wyrazenie

*nazwe funkcji i argumenty podajemy tylko raz.

*Strażnicy muszą być w tej samej kolumnie!!!

przykladowy program:

```
znak2 x | x<0 = (-1)
        | x==0 = 0
        | x>0  = 1
```

albo tak moglibyśmy zapisać powyższy kod:

```
znak2 x | x < 0 = (-1)
        | x == 0 = 0
        | otherwise = 1
```

otherwise oznacza „w przeciwnym wypadku”

Strażnik jest bardzo fajną definicją warunkową – jest prosty i możemy np. w wyniku wywołać inną funkcję jak to robiliśmy np. z pierwiastkami, że jeżeli delta > 0 to wtedy wywołaliśmy funkcję **licz2pierw a b c** , tylko pamiętamy, że musieliśmy tej funkcji przekazać argumenty **a b c** żeby miała ona na czym liczyć te pierwiastki ….

W strażniku śmiało możemy wykorzystać definicję lokalną **where**, where jest ze strażnikiem lepsze ponieważ zadziała dla całej konstrukcji funkcji ze strażnikiem natomiast let nie mogłoby otoczyć wszystkiego… zatem jeśli chesz użyć definicji lokalnej ze strażnikiem to tylko where

przykład:

```
znak2 x | omega < 0 = (-1)
        | x == 0 = 0
        | otherwise = 1
              where
                    omega = x + x - x
```

```
wyrazenie case, wyznacza pewne wartosci, ale musi miec skonczona liczbe wzorcow
zademonstrujmy wyrażenie case podobna do funkcji do znak, nazwiemy ja liczba
case do wyliczeniowych typow stosujemy np. dni tygodnia (poniedzialek np. 1)
czasem jest wygodne, gdy mamy jakieś przypadki :)
zba x = case znak2 x of
            (-1) -> "ujemna"
            0 -> "zero"
            1 -> "dodatnia"
zero musi byc pod nawiasem, koniecznie!
```

Kolejna definicją warunkową jest tzw. **definiowanie dopasowując do wzorca**
Chodzi w niej o to jak w tytule: dopasowujemy do wzorca ☺
Czyli gdy jakieś określone argumenty na funkcji naszej będą spełnione
to ma się coś konkretnego zadziać np. wypisać True albo False
plusiki i uwagi dopasowywania do wzorca ☺
+czasami jest wygodniejsza niż guard
+należy pamiętać że dopasowanie do wzorca sprawdzane jest od góry do dołu
+z powyższego wynika to, żeby uważać jak się stawia określone warunki!!

przykład dopasowania do wzorca dla np. funkcji logicznej p || q:

```
lub1 False False = False
lub1 False True = True
lub1 True False = True
lub1 True True = True
```

Czyli pisząc do **Haskella** np. lub1 False True powinniśmy na ekran otrzymać: True

Warto też pamiętać o znaku _ którego nazywamy **wild card**(dzika karta). W grach „dzika karta" może być wszystkim, dlatego ten znak tak się nazywa, może być wszystkim, zastosowaliśmy go do poniższego przykładu upraszczając funkcję lub do minimum ☺

```
lub2 False False = False
lub2 _ _ = True
```

Zauważmy, że lub(OR) **jest tylko fałszywe gdy mamy 0 lub 0** czyli False lub False zatem możemy tak zrobić. Tylko tu należy pamiętać o kolejności tych wzorców… Gdybyśmy lub2 False False dali niżej to zawsze działałby ten lub2 _ _ czyli dla False False dostalibyśmy True – a to by było źle ☺

Ważną rzeczą do zapamiętania jest to, że możemy zastosować funkcję **error** żeby użytkownikowi coś napisać, zastosowaliśmy to w poniższym przykładzie:

```
silnia2 :: Integer -> Integer
silnia2 0 = 1
silnia2 n | n > 0 = n * silnia(n-1)
          | otherwise = error "ujemny arg"
```

A tak do zapamiętania, bo zapomniałem wcześniej umieścic:
funkcja **f x y = x + y** (tu podajemy dwa argumenty)
a funkcja **f (x,y) = x + y** (tu podajemy jeden argument będący parą)
To dwie różne funkcje!!
Funkcja f x y działa najpierw na x a potem dopiero na y
Funkcja f (x,y) działa jednocześnie operując na x i y

Listy i krotki:

Lista – rozmiar listy nie jest określony, składa się z elementów, które są tego samego typu, możemy do niej dołączać elementy za pomocą operatora : (ma priorytet 5 ten operator)
Przykładowe listy: [1,2,3] , ['a','b','c'] , [('a'),('b')]

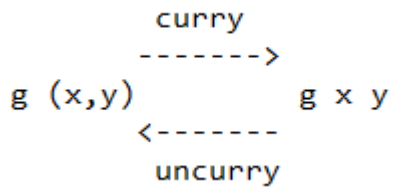
Krotka – rozmiar krotki jest określony, może mieć mieszane elementy np. liczby i znaki jednocześnie
Przykładowe krotki: (1,2,"abba") , (5,'a','c')

```
krotka zlozona np.
Osoba = (imie, wiek, plec)
zdefiniujmy sobie typ: "Osoba"
type Osoba :: (String, Int, Char)
```

Wbudowane funkcje, które operują na parach:

```
fst - wyznacza pierwszy element pary
fst :: (a,b) -> a
snd - wyznacza drugi element pary
snd :: (a,b) -> b
```

W **Haskell** mamy tzw. funkcje curry i uncurry, są one po to aby móc zastosować np. określoną funkcję w której powinniśmy dać parę argumentów – dwa oddzielne argumenty i vice versa ☺ , przedstawia to poniższa grafika:



Czyli mówiąc, krótko:
curry rozdziela,
uncurry łączy

UWAGA! Jeżeli chcesz zapisać argument, który jest listą to musimy dodać literkę s do argumentu np.
xs – lista
xss – lista listy
xsss – lista list list

Trzeba tego przestrzegać – wtedy funkcja jest bardziej zrozumiała ☺

Dysponujemy kilkoma operatorami do list… i są to:
!! – operator indeksowania (możemy wyświetlić element spod określonego indeksu, należy pamiętać że ten operator liczy od 0)
np. [1,2,3,4,5] !! 2 zwróci nam 3

++ - operator konkatenacji (do łączenia list)
np. [1,2,3] ++ [3,4] ++ [6,7] zwróci nam [1,2,3,3,4,6,7]

: - operator służący do dołączania elementów do listy
np. 5:[3,4,5] zwróci nam [5,3,4,5]

! uwaga, mamy też tzw. **operator zasięgu**, nie musimy definiować całej zawartości listy, możemy ustalić zakres od … do którego mają być elementy i przykładowo:

```
[1,3..10] -> [1,3,5,7,9]   :: [Integer]
['a'..'k'] -> "abcdefghijk" :: [Char]
[10,8..0] -> [10,8,6,4,2,0] :: [Integer]
```

Zauważmy np. też że gdy mamy listę **[1,4..6]** to nasza lista tak będzie wyglądać: **[1,4]**

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
  PODSTAWOWE FUNKCJE NA LISTACH:
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
head [1,2,3] -> [1]
tail [1,2,3] -> [2,3]
last [1,2,3] -> 3
init [1,2,3] -> [1,2]      (zwraca liste bez ostatniego elementu)
length [1..8] -> 8
null [1,2,3] -> False      (sprawdza czy lista jest pusta)
reverse [1,2,3] -> [3,2,1]
take 2 [1,2,3,4] -> [1,2]
take 5 [2,7,..]  -> [2,7,12,17,22]
drop 5 [1,2] -> []
drop 2 [1,2,3,4] -> [3,4]
uwaga: drop nie używamy z listą nieskończoną!!
bo jak usuniemy np. 2 pierwsze elementy to nadal
lista nieskończona zostanie
minimum [8,4,2,1,5,6] -> 1
maximum [1,9,2,3,4] -> 9
sum [5,2,1,6,3,2,5,7]      (zsumuje zawartośc listy)
product [6,2,1,2]          (iloczny zawartości listy)
elem 4 [3,4,5]             (sprawdza czy 4 jest elementem listy)
```

PS: elem jeśli znajdzie podany element w liście to zwróci oczywiście True a jeśli nie to False

Listy też są wygodne do … tworzenia **funkcji rekurencyjnych na listach** ☺
Musimy mieć świadomość jak robić takie funkcje rekurencyjne i umieć Ewaluować(pokazywać) na przykładowych listach jak działa nasza funkcja.

Zacznijmy od przykładu….

```
-- member(x,xs) - należenie elementu do listy
-- member :: Eq a => a -> [a] -> Bool

member x [] = False
member x (y:ys)
                | x==y = True
                | otherwise = member x ys

-- na kolokwium takie coś jest że np. wyznaczyć wartość
-- dla przykładowego argumentu

-- Ewaluacja
-- member 2 [3,1,2,4]
-- member 2 [1,2,4]
-- member 2 [2,4]
-- TRUE
```

W funkcji rekurencyjnej musimy **zacząć od warunku zakończenia rekurencji**, tutaj tym jest: **member x [] = False**, czyli musimy sobie wyobrazić, że jeśli nasza funkcja dojdzie do momentu, że lista będzie pusta to funkcja ma zwrócić False. Proste? Proste. No a dlaczego ma być False w tym przypadku? No twoja funkcja member ma sprawdzić czy element podany x należy do listy, jeżeli lista jest pusta to od razu wiadomo, że takiego elementu nie ma, prawda?

aha… i **warunek zakończenia rekurencji** jest tylko w funkcji rekurencyjnej, nigdzie więcej ☺

no i gdy przechodzimy do naszej rekurencji to jedyne co musimy pamiętać to

1) Rozpisać listę na głowę i ogon np. [] rozpisujemy (y:ys)
2) Użyć w rekurencji naszej funkcji, którą definiujemy.. no bo rekurencja to odwoływanie się do samej siebie…..

No … i to tak naprawdę tyle ogólnie można powiedzieć o **funkcjach rekurencyjnych z użyciem list**… naprawdę, reszta zależy tak naprawdę od zadania jakie masz , ale to o czym napisaliśmy to jest w każdej funkcji rekurencyjnej na listach czyli warunek i rekurencja. Nie ma co się bać tego – to jest proste :)

A co do ewaluacji to trzeba pamiętać o tym aby

1) Robić nawiasy
2) Uważać na priorytety !!

PS: Z funkcjami rekurencyjnymi na listach warto zajrzeć do tych co zrobiliśmy i do zadanek i spróbować je zrozumieć. Potem przysiąść do kolokwium i poćwiczyć kreatywność.

List Comprehensions:

Nie ma to niestety polskiego odpowiednika ale to nic innego jak zapisywanie zbioru w liście, takim list comprehension będzie np.:
[x*x | x <- [1..10], even x]
Pamiętamy, że **,** oznacza koniunkcje
even – funkcja która sprawdza czy podany element jest liczbą parzystą
odd – funkcja, która sprawdza czy podany element jest liczbą nieparzystą
np. odd 5 zwróci True

no i tutaj zauważ stosujemy **strażnik(|)** jednak w takim celu aby oddzielić część tego co chcesz wyświetlić na ekran od tego jakie warunki musi spełniać to co chcesz właśnie wyświetlić czyli tłumacząc zbiór:

[x*x | x <- [1..10], even x]
wyświetlimy listę elementów x*x taką, że x należy do listy od 1 do 10 i x jest liczbą parzystą czyli w wyniku otrzymamy [4,16,36,64,100]

inne przykłady:

```
[2*x | x <- [1..5]]
[y `mod` 3 | y <- [5..10]]
[ a*b | (a,b) <- [ (1,2), (2,3), (3,4)]]
[(x,y) | x <- [1,2], y <- [3,4]]
[x | x <- [1..12], y<- [1..12], x*y == 12]
[x | x <- [-5,2,3,-2], x>0 ]
```

Musisz wiedzieć co one wyświetlą i jak to opisać, przykładowo opiszmy pierwszy zbiór:
Lista [2,4,6,8,10] wszystkich liczb 2*x takich że x jest elementem listy [1..5]

Map:

Map stosuje funkcje do każdego elementu listy np. map sqrt [4,9,81] wezmie pierwiastek z kazdego elementu listy bo gdybsmy zrobili sqrt [4,9,81] to bedzie **blad**... bo sqrt samo dziala tylko na jednej wartosci!!

konstrukcja map -> -> -> **map funkcja lista**

warto tutaj wspomnieć, że funkcja nie musi być wbudowana, może też być autorska(czyli twoja, napisana przez ciebie….)

Pamiętaj, gdy piszesz funkcje musisz podać argumenty(argument) dla której ma ona działać!!! Tutaj mamy map f xs

xs to nasz argument czyli lista!!

przykład:

```
f x = 3*x-2

f_lista3 xs = map f xs
```

czyli pisząc np. **f_lista3 [4,5]** otrzymamy **[10,13]** ☺

ale w powyższym przykładzie naszą funkcję f x napisaliśmy zewnętrznie , to oznacza, że funkcja f x będzie widoczna wszędzie. Pamiętasz o definicjach lokalnych? Jeśli tak to wiesz pewnie, że możemy napisać powyższą funkcję f x lokalnie przy użyciu where TAK ABY funkcja f x była tylko na potrzeby funkcji f_lista!!!! ,
będzie to wyglądać tak:

```
f_lista4 xs = map f1 xs
                where
                    f1 x = 3*x-2
```

Na co warto zwrócić uwagę?

Że f1 ma argument x czyli nie listę! Dlaczego? Funkcja map działa na każdym elemencie listy, zatem nie musimy nazywać argumentu funkcji f1 jako lista(czyli xs) ponieważ ta funkcja ma działać na **pojedynczym elemencie**!!

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
             FUNKCJE ANONIMOWE
nie musimy im nadawac żadnej nazwy!!
lambda w Haskell to \        (backslash)
kropka w Haskell to ->       (strzałka)
nie definiujemy funkcji tylko wpisujemy postać
funkcji odrazu do Haskell wprost

przykład funkcji anonimowej:
(\x->3*x-2) 7

gdyby chcieć dla całej listy powyższe zastosować to
map (\x->3*x-2) [1,5]
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Funkcje anonimowe możemy pisać od razu w Haskell, i przy okazji – **pamiętasz, że funkcja ma priorytet 10 tak**? ☺

Przykład funkcji anonimowej, którą zapisaliśmy w skrypcie(pliku hs):

```
f_lista6 xs = map (\x->3*x-2) xs
```

czyli dla np. f_lista6 [1,2,3] otrzymamy wynik: [1,4,7]

Przykład funkcji anonimowej, którą od razu zapiszemy w Haskell:

(\x->x*x) 2 , w wyniku otrzymamy oczywiście 4

Przykład:

```
Funkcja anonimowa, która przekształca elementy listy
według funkcji a_elem

a_elem3 xs = map (\x->a_elem x) xs
```

Komentarz:

Musieliśmy zastosować funkcję map, żeby policzyć dla każdego elementu listy wynik w oparciu o funkcje a_elem