Instrukcja warunkowa
Ogólna postać

```
--  if warunek then instrukcja1 else isnrukcja2
--warunek jest typu Boolean
```

Przykład: wartość bezwzględna n

```
-- |n| -- zawsze powinno być else
wbezwzgledna :: Int -> Int
wbezwzgledna n = if n >= 0 then n else -n
```

Przykład: znak liczby n

```
--sgn(n)--instukcje If ... można zagnieżdżać
znak :: Int -> Int
znak n = if n < 0 then -1 else
     if n == 0 then 0 else 1
```

Wyrażenia warunkowe - *guared equations*
Ogólna postać

```
--Ogólna postać
-- warunek jest typu Boolean
-- znak | czytaj {\em takie, że}
fun t | warunek_1 = cialo_1 --If warunek_1 jest True, to cialo_1 i koniec.
fun t | warunek_2 = cialo_2 --If warunek_2 jest True, to cialo_2 i koniec.
fun t | warunek_3 = cialo_3 --If warunek_3 jest True, to cialo_3 i koniec.
fun t = cialo_4
```

Przykład: wartość bezwzględna n

```
wbezwzgledna n | n>=0 = n
wbezwzgledna n | otherwise = -n
----przyklad 1 |n| - krócej
wbezwzgledna n | n>=0 = n
               | otherwise = -n
```

Przykład: znak liczby n

```
znak n | n>0 = 1
       | n==0 = 0
       | otherwise = -1
    | otherwise = -n
----przyklad 1 |n| - krócej
wbezwzgledna n | n>=0 = n
               | otherwise = -n
```

Działania na krotkach
Pattern matching
– Znak '-' oznacza *pattern matching*; zastępuje element

```
myfst :: (a,b) -> a
myfst (x,_) = x
mysnd :: (a,b) -> b
mysnd (_,y) = y
```

Działania na listach

Pattern matching

Funkcja `test` sprawdza czy lista ma dokładnie trzy znaki i pierwszy znak jest `'a'`

```
test :: [Char]-> Boolean
test ['a',_,_] = True
test _ = False
```

Funkcja `test` sprawdza czy lista ma pierwszy znak `'a'`

```
test :: [Char]-> Boolean
test ('a': _ ) = True
test _ = False
```

Napisz funkcje `myhead` i `mytail`, która wykorzytuje *pattern matching*. Napisz funkcję `mytrzeci`, która zwraca trzeci element listy (krotki) trzema sposobami:

1. wykorzystaj `tail` i `head`
2. pattern matching
3. operator !!

List comprehensions

Zdefiniuj listy, której która składają się z elementów zbioru A, B, C i D gdzie

1. $A = \{x^2 : x = 1, \ldots, 10\}$
2. $B = \{(x, y) : x = 1, \ldots, 3; y = 4, 5\}$
3. $C = \{(x, y) : x = 1, \ldots, 3; y = x, \ldots, 3\}$
4. $D = \{(x, y, z) : x = 1, 3; y = x, \ldots, 3; z = 6\}$

```
A=[x^2 | x <- [1..5]]
B=[(x,y) | x <- [1,2,3], y <- [4,5]]
C=[(x,y) | x <- [1..3], y <- [x..3]]
```

Zdefiniuj funkcję, która dla danej listy listę list tworzy listę składającą się ze wszystkich elementów list w wejściowej liście

```
myconcat :: [[a]] -> [a]
myconcat xss = [x | xs <- xss, x <- xs]
```

Zdefiniuj funkcję dla danej listy krotek dwuelementowych zwraca listę składjącą się z pierwszych elementów każdej krotki w wejściowej liście

```
firsts :: [(a,b)] -> [a]
firsts ps = [x | (x,_) <- ps]
```

Co oblicza poniższa funkcja?

```
cooblicza :: [a] -> Int
cooblicza xs = sum [1 | _ <- xs]
```

List comprehensions z *guards*
Jeśli wyrażenie guards jest True, to dodajemy elementy do listy wyjściowej
Co obliczają poniższe funkcje?

```
co1 = [x | x <- [1..10], even x]
co2 n = [x | x <- [1..n], mod n x == 0]
co3 n = co2 n == [1,n]
co4 n = [x | x <- [2..n], co3 x]
```

Funkcja zip

```
zip :: [a] -> [b] -> [(a,b)]
--przykład 1
zip ['a','b','c'] [1,2,3,4]
--przykład 2
pairs :: [a] -> [(a,a)]
pairs xs = zip xs (tail xs)
```

Zdefiniuj funkcję dla danej listy xs i elementu x zwraca listę, która zawiera wszystkie pozycje, na której wystepuje elemet x w wejściowej liscie.

```
positions :: Eq a => a -> [a] -> [Int]
positions x xs = [i | (x',i) <- zip xs [0..], x == x']
```

Funkcja map
Jak działa funkcja map?

```
map :: (a -> b) -> [a] -> [b]
map f xs = [f x | x <- xs]
```

Sprawdź co obliczy?

```
map (+1) [1,3,5,7]
map even [1,2,3,4]
map reverse ["abc","def","ghi"]
map (map (+1)) [[1,2,3],[4,5]]
```