Haskell - przydatne wywołania

Haskell jest funkcyjnym językiem programowania.

GHC - Glorious/Glasgow Haskell Compiler - kompilator języka Haskell
GHCi - to tylko interaktywne środowisko do wykonywania poleceń Haskella

```
succ 1/10 -- returns 0.2
succ (1/10) -- return 1.1
```

Funkcje

Nazwa fukncji nie może się zaczynać od wielkiej litery ale za do z jakiegoś powodu można w jej nazwie używać **'**.

```
conanO'Brien = "It's a-me, Conan O'Brien!"
```

Kiedy funkcja nie przyjmuje żadnych parametrów mówimy, że jest to definicja (lub nazwa); Ponieważ nie możemy zmienić tego, co oznacza dana nazwa (i funkcja) to od momentu ich zdefiniowania, nazwa fukcji *conanO'Brien* i napis *It's a-me, Conan O'Brien* mogą być stosowane wymiennie.

Listy

Żeby zdefiniowac listę wystarczy użyć polecenia:

```
let lostNumbers = [4,8,15,16,23,42]
lostNumbers -- shows lists in command line
```

Zmienne typu *string* są w Haskellu traktowane jako listy więc wszystkie polecenia, które można wywołać na listach, można też wywołać na stringach.

Operacje na listach

```
Prelude> [1,2,3,4] ++ [9,10,11,12] -- łączenie list
[1,2,3,4,9,10,11,12]
Prelude> "hello" ++ " " ++ "world"
"hello world"
Prelude> 'A':" SMALL CAT" -- wstawianie elementu na początek listy
"A SMALL CAT"
Prelude> "Steve Buscemi" !! 6 -- pobranie elementu spod indeksu = 6
'B'
Prelude> head [5,4,3,2,1]
5
Prelude> tail [5,4,3,2,1]
[4,3,2,1]
Prelude> last [5,4,3,2,1]
1
Prelude> init [5,4,3,2,1]
[5,4,3,2]
Prelude> length [5,4,3,2,1]
5
Prelude> null [1,2,3]
False
Prelude> null []
True
Prelude> reverse [5,4,3,2,1]
[1,2,3,4,5]
Prelude> take 3 [5,4,3,2,1]
[5,4,3]
Prelude> drop 3 [8,4,2,1,5,6]
[1,5,6]
Prelude> minimum [8,4,2,1,5,6]
1
Prelude> maximum [1,9,2,3,4]
9
Prelude> sum [5,2,1,6,3,2,5,7]
31
Prelude> product [6,2,1,2]
24
Prelude> 4 `elem` [3,4,5,6]
True
Prelude> 10 `elem` [3,4,5,6]
False
```

Używając któregoś z poleceń: **head, tail, last, init** trzeba uważać żeby nie stosować ich na pustych listach bo wyrzucają błąd ale nie na etapie kompilacji a dopiero w runtimie.

Porównywanie list

```
Prelude> [3,2,1] > [2,1,0]
True
Prelude> [3,2,1] > [2,10,100]
True
```

```
Prelude> [3,4,2] > [3,4]
True
Prelude> [3,4,2] > [2,4]
True
Prelude> [3,4,2] == [3,4,2]
True
```

Porównywanie list odbywa się dla poszczególnych elemntów, tj. porównywany jest najpierw pierwszy element pierwszej listy z pierwszym elementem drugiej itd aż do momentu, że warunek nie będzie spełniony ale skończy się któraś z list.

Generowanie list dla podanych zakresów

W sumie to jest całkiem intuicyjne ale trzeba uważać na definiowanie list, których wartości maleją, nie działa wywołanie [20..1] (które tak btw zwraca []), trzeba zapisać to w taki sposób [20,19..1]. Trzeba też uważać na generowanie list dla flotów bo może się wysypać precyzja, tj. lista zostanie utworzona ale mogą być dziwne cyfry po przecinku itp.

```
Prelude> [1..20]
[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20]
Prelude> ['a'..'z']
"abcdefghijklmnopqrstuvwxyz"
Prelude> [2,4..20]
[2,4,6,8,10,12,14,16,18,20]
Prelude> [20,19..1]
[20,19,18,17,16,15,14,13,12,11,10,9,8,7,6,5,4,3,2,1]
Prelude> take 10 (cycle [1,2,3])
[1,2,3,1,2,3,1,2,3,1]
Prelude> take 12 (cycle "LOL ")
"LOL LOL LOL "
Prelude> take 10 (repeat 5)
[5,5,5,5,5,5,5,5,5,5]
Prelude> replicate 3 10
[10,10,10]
```

Ciekawe wywołanie:

```
Prelude> take 24 [13,26..]
```

Ponieważ Haskell jest *"leniwym"* językiem to nie wygeneruje nieskończonej listy tylko ogarnie, że chcesz 24 element i taką liste zwróci.

List comprehension

Nie wiem jak to przetłumaczyć na polski, coś jak lista wyrażeń…

```
Prelude> [x*2 | x <- [1..10]]
[2,4,6,8,10,12,14,16,18,20]
Prelude> [x*2 | x <- [1..10], x*2 >= 12]
[12,14,16,18,20]
Prelude> boomBangs xs = [ if x < 10 then "BOOM!" else "BANG!" | x <- xs, odd x]
Prelude> boomBangs [7..13]
["BOOM!","BOOM!","BANG!","BANG!"]
Prelude> let xxs = [[1,3,5,2,3,1,2,4,5],[1,2,3,4,5,6,7,8,9],[1,2,4,2,1,6,3,1,3,2,3,
Prelude> [ [ x | x <- xs, even x ] | xs <- xxs]
[[2,2,4],[2,4,6,8],[2,4,2,6,2,6]]
```

Typy danych i ich klasy

Haskell ma statyczne typowanie co znaczy, że typ każdej zmiennej musi być znany na etapie kompialcji projekctu. Co więcej, Haskell ma mechanizm wnioskowania typu na podstawie przekazanych wartości.

W Haskellu są dwa typy całkowite: **Int** i **Integer**. Pierwszy jest ograniczony dolnym i górnym limitem, ktory różni się w zależności od maszyny, na której to sprawdzamy. Za to Integer nie ma żadnych limitów ale za to jest mniej wydajny.

klasy typów:

- **Eq** - używana dla typów, które wspierają sprawdzanie równości.
- **Ord** - jest używana dla typów, które wspierają sortowanie
- **Show** - typy tej klasy mogą być reprezentowane jako string
- **Read** - przeciwstawne działanie do Show, ogranicza się do typów które mogą być przekształcone ze stringa na konkretny typ, np. Int
- **Enum** - typ może być ponumerowany
- **Bounded** - typ ma zdefiniowane limity, zarowno dolny jak i górny
- **Num** - typ może być traktowany jak liczby

W przypadku klasy read niekiedy konieczne będzie zdefiniowanie na jaki typ ma być skonwertowany string, np. read 5 :: Int. W przypadku, gdybyśmy wynik read wykorzystywali w innej funkcji, która oczekuje, np. Float to read by się domyślił, że ma przekształcić wartość na float. Ogólnie czasami warto jest po prostu zdefiniować, że oczekujemy konkretnej na wypadek, gdyby kompilator nie był się w stanie domyślić.