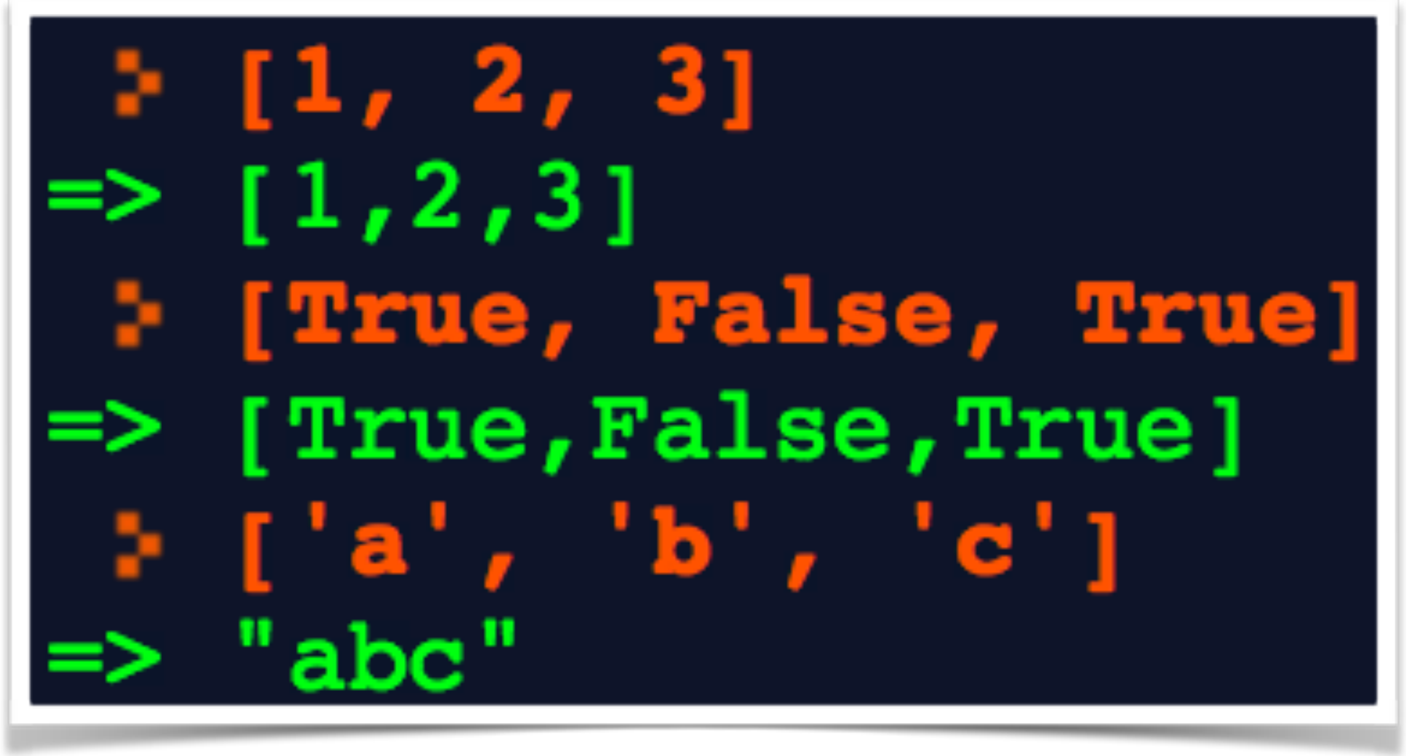
Functional programming

Working with lists

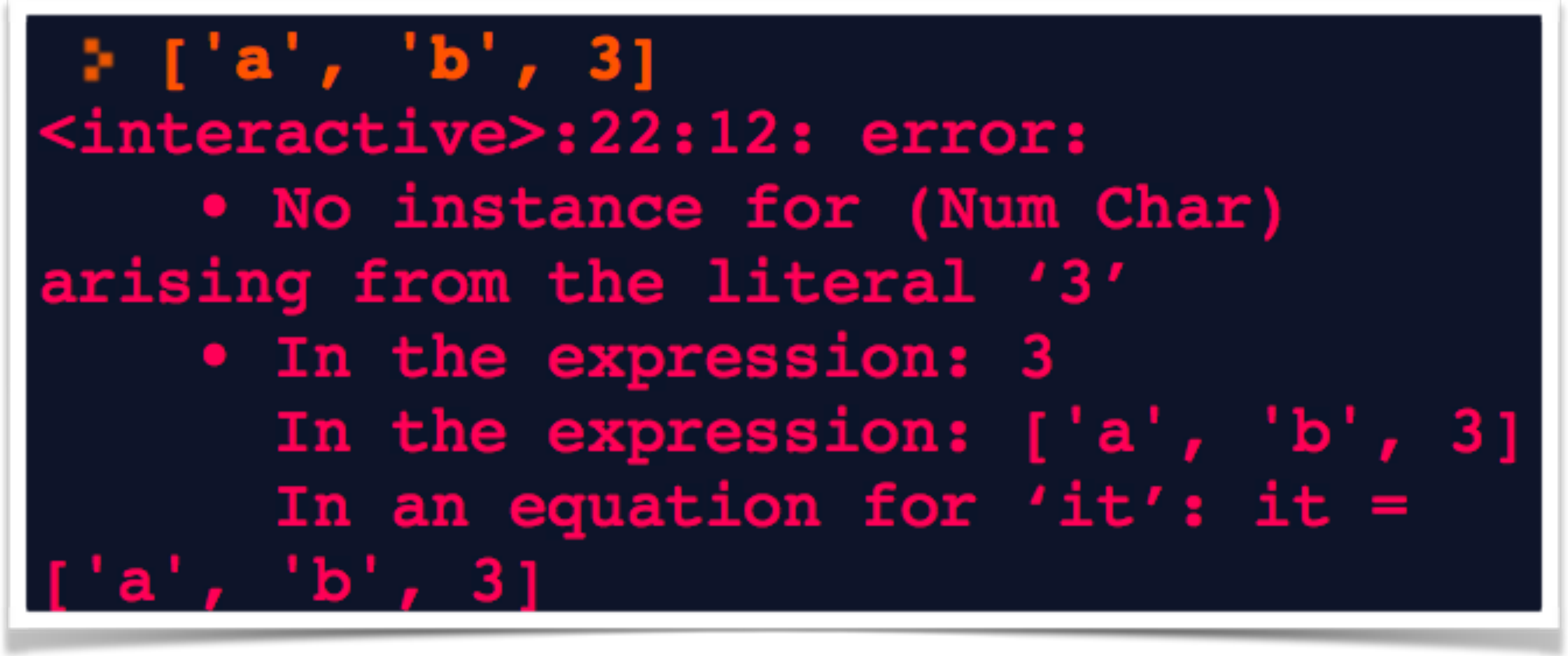
Representation

- Lists store elements of the same type
- Delimited with **[** and **]**
- Elements are separated by **,**

Is ***[ ]*** *also a list?*



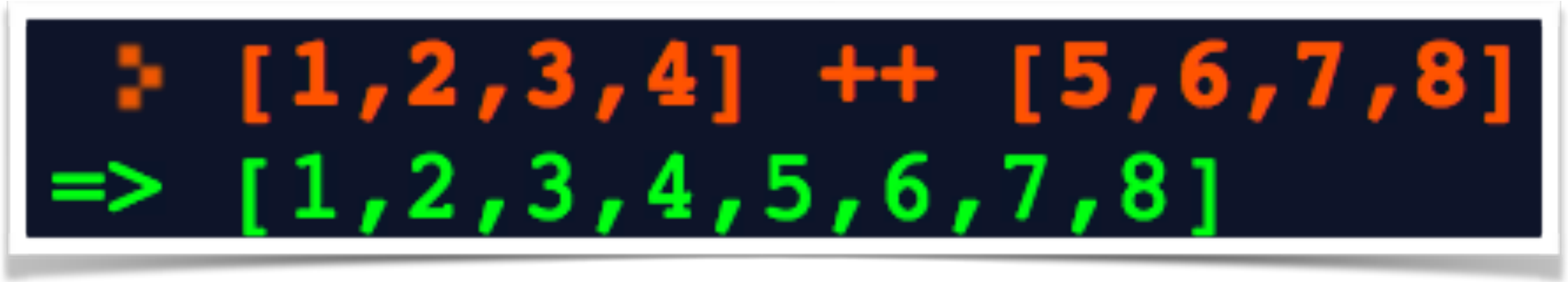
```
 > [1, 2, 3]
=> [1,2,3]
 > [True, False, True]
=> [True,False,True]
 > ['a', 'b', 'c']
=> "abc"
```



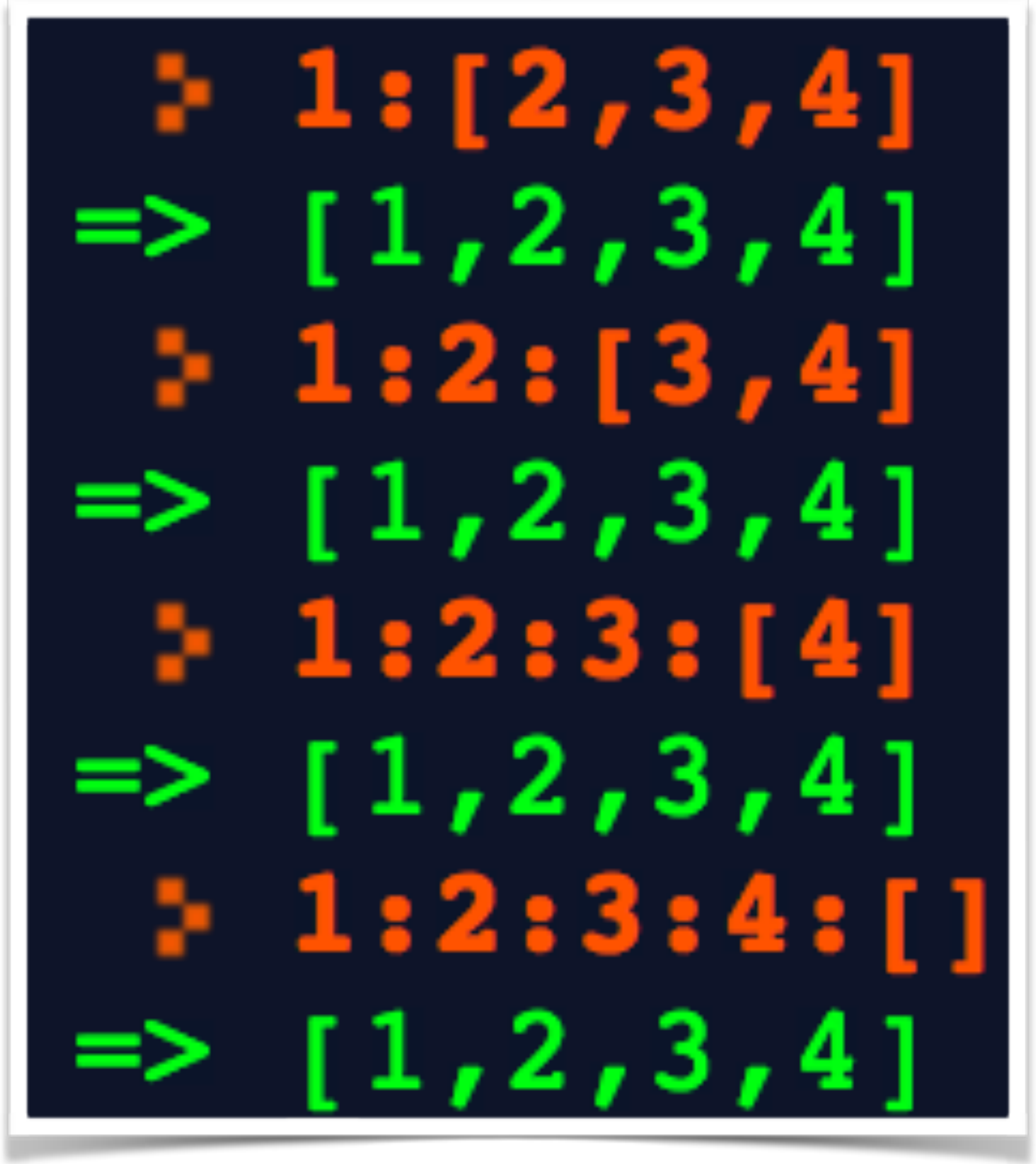
```
 > ['a', 'b', 3]
<interactive>:22:12: error:
    • No instance for (Num Char)
arising from the literal ‘3’
    • In the expression: 3
      In the expression: ['a', 'b', 3]
      In an equation for ‘it’: it =
['a', 'b', 3]
```

Concatenation

- Use **++** operator to concatenate two lists
- Haskell will walk through the whole list on the left side. Be careful with big lists
- To add element to the head of list use **:** operator (pronounced “cons")
- Adding element to the head of list is fast



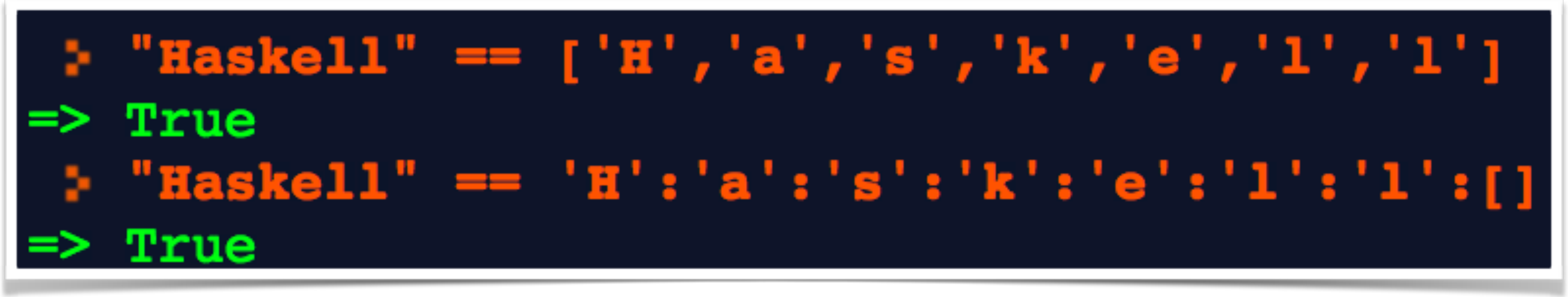
```
 > [1,2,3,4] ++ [5,6,7,8]
=> [1,2,3,4,5,6,7,8]
```



```
 > 1:[2,3,4]
=> [1,2,3,4]
 > 1:2:[3,4]
=> [1,2,3,4]
 > 1:2:3:[4]
=> [1,2,3,4]
 > 1:2:3:4:[]
=> [1,2,3,4]
```

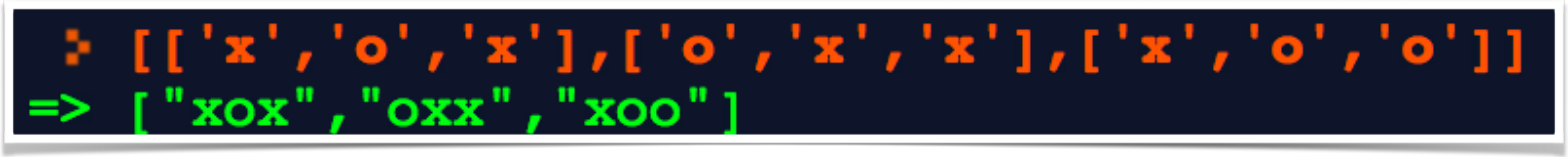
More list things

- Strings are lists of chars



```
 > "Haskell" == ['H','a','s','k','e','l','l']
=> True
 > "Haskell" == 'H':'a':'s':'k':'e':'l':'l':[]
=> True
```

- How can you represent a 2D array in Haskell?



```
 > [['x','o','x'],['o','x','x'],['x','o','o']]
=> ["xox","oxx","xoo"]
```

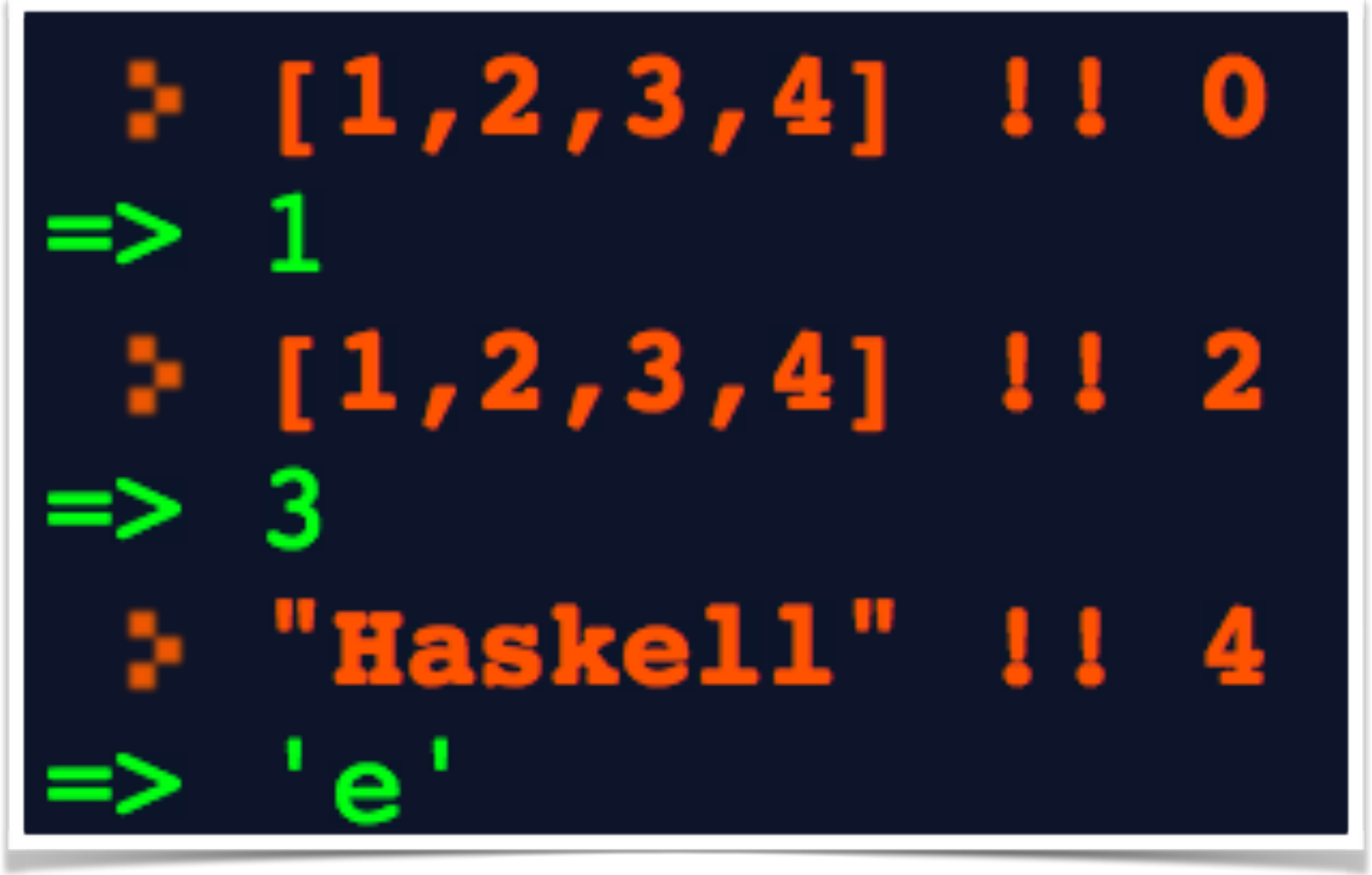
Exercise

- Which of the following list declarations are valid in Haskell?

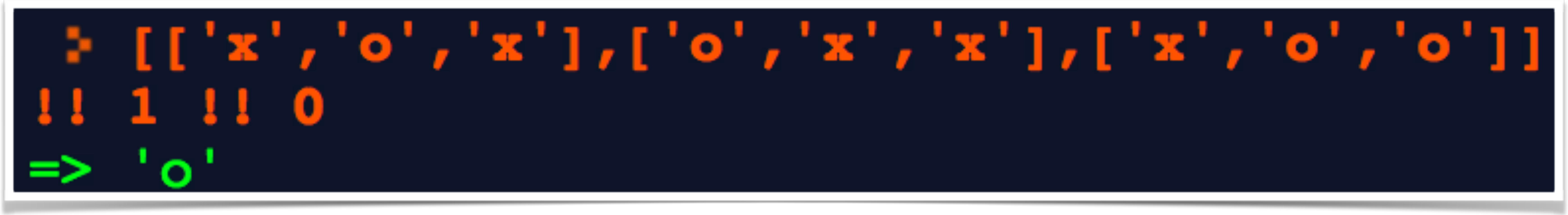
[1, 2, 3, [ ]]	?
[[1, 2, 3], [ ]]	?
[ ] : [[1, 2, 3], [4, 5, 6]]	?
[[ ]] : [[1, 2, 3], [4, 5, 6]]	?
[ ] : [ ]	?
[ ] : [[ ]]	?

Accessing elements

- **!!** operator is used to access list elements by index
- indexes start with **0**



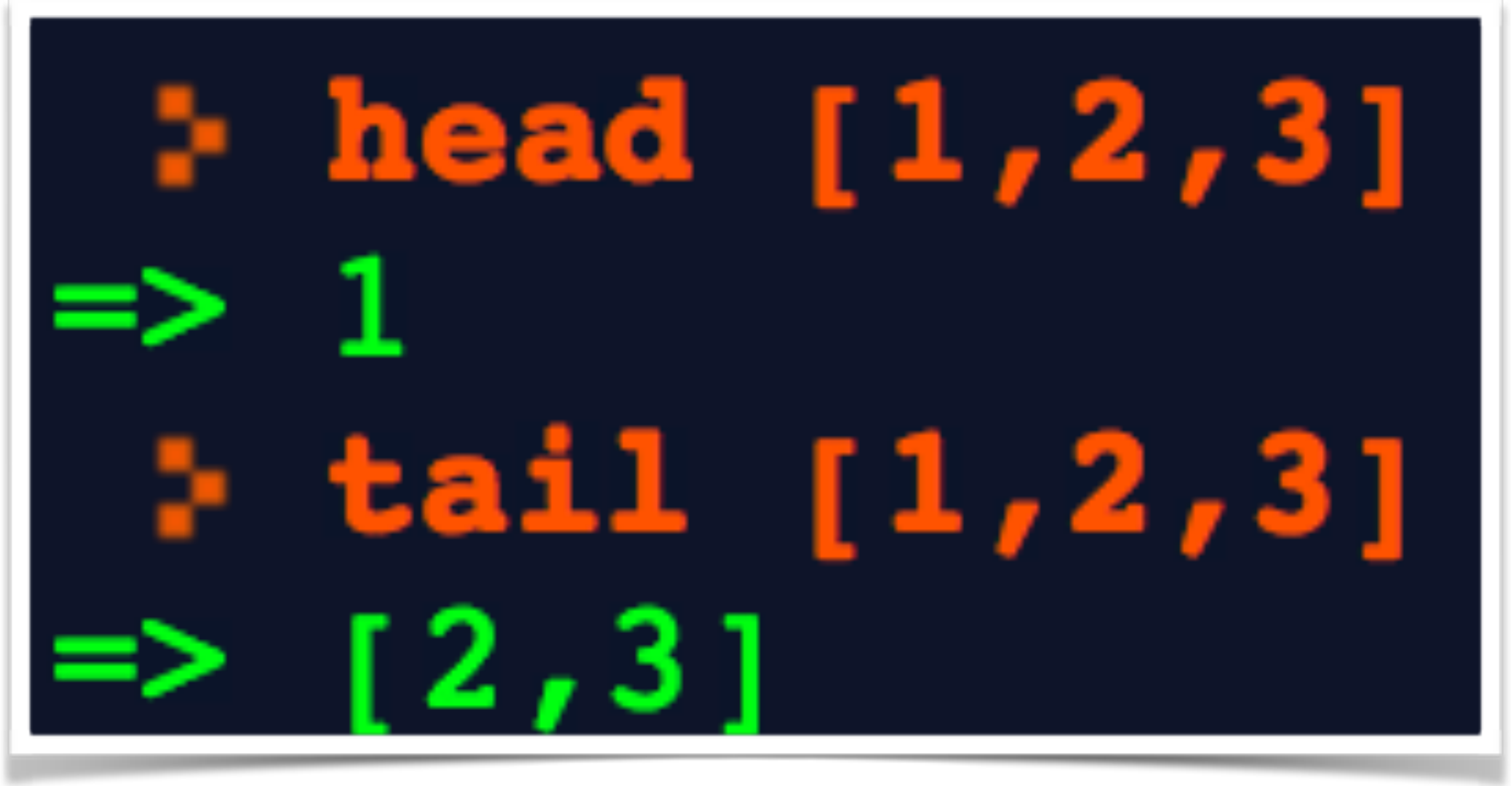
```
 > [1,2,3,4] !! 0
=> 1
 > [1,2,3,4] !! 2
=> 3
 > "Haskell" !! 4
=> 'e'
```



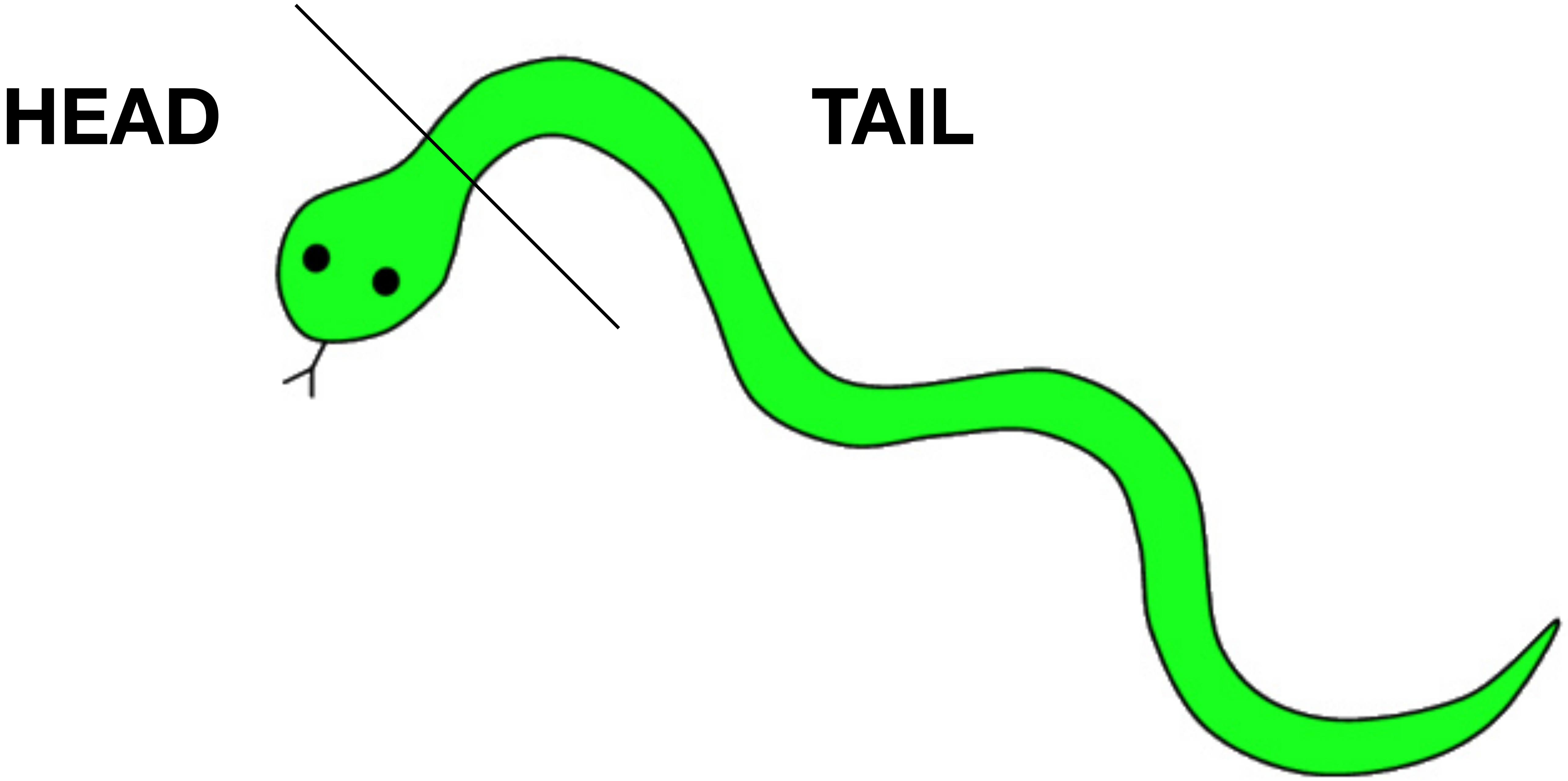
```
 > [['x','o','x'],['o','x','x'],['x','o','o']]
!! 1 !! 0
=> 'o'
```

Accessing elements

A list can be separated into **head** and **tail**

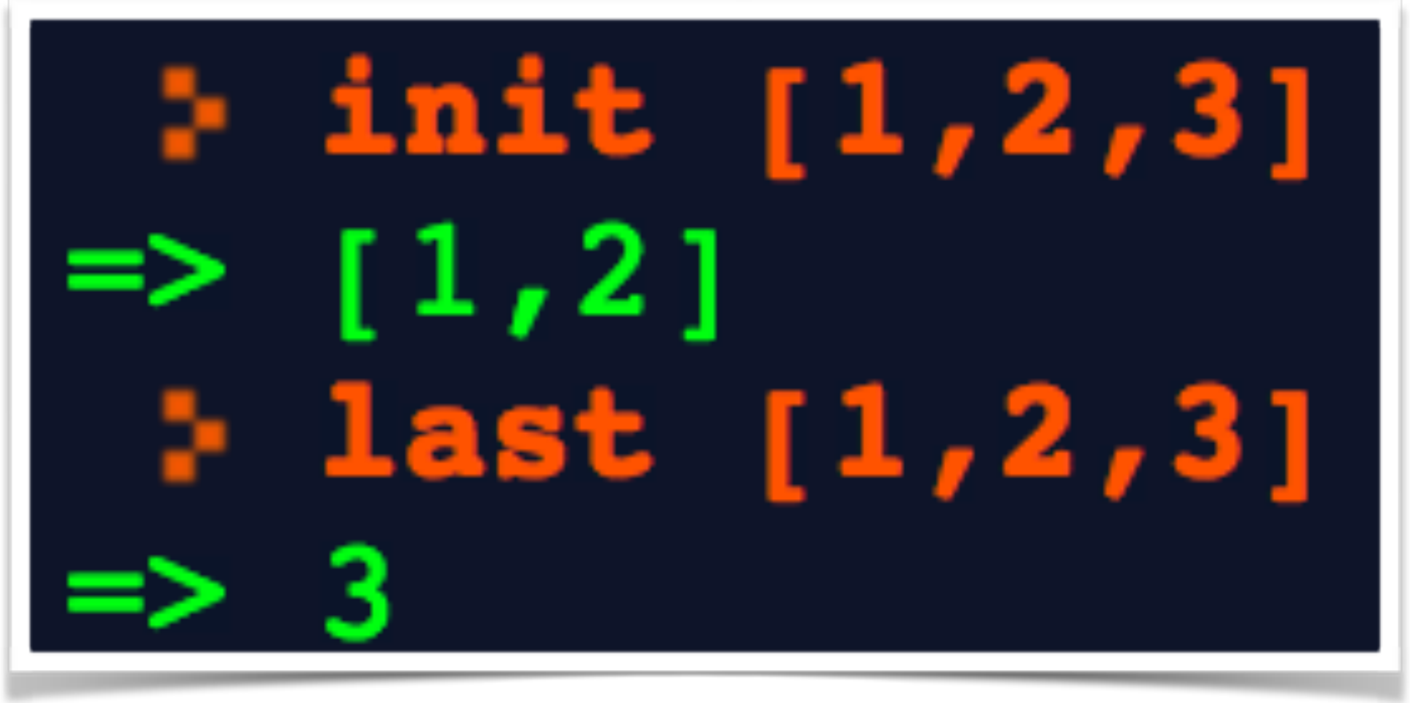


```
 > head [1,2,3]
=> 1
 > tail [1,2,3]
=> [2,3]
```

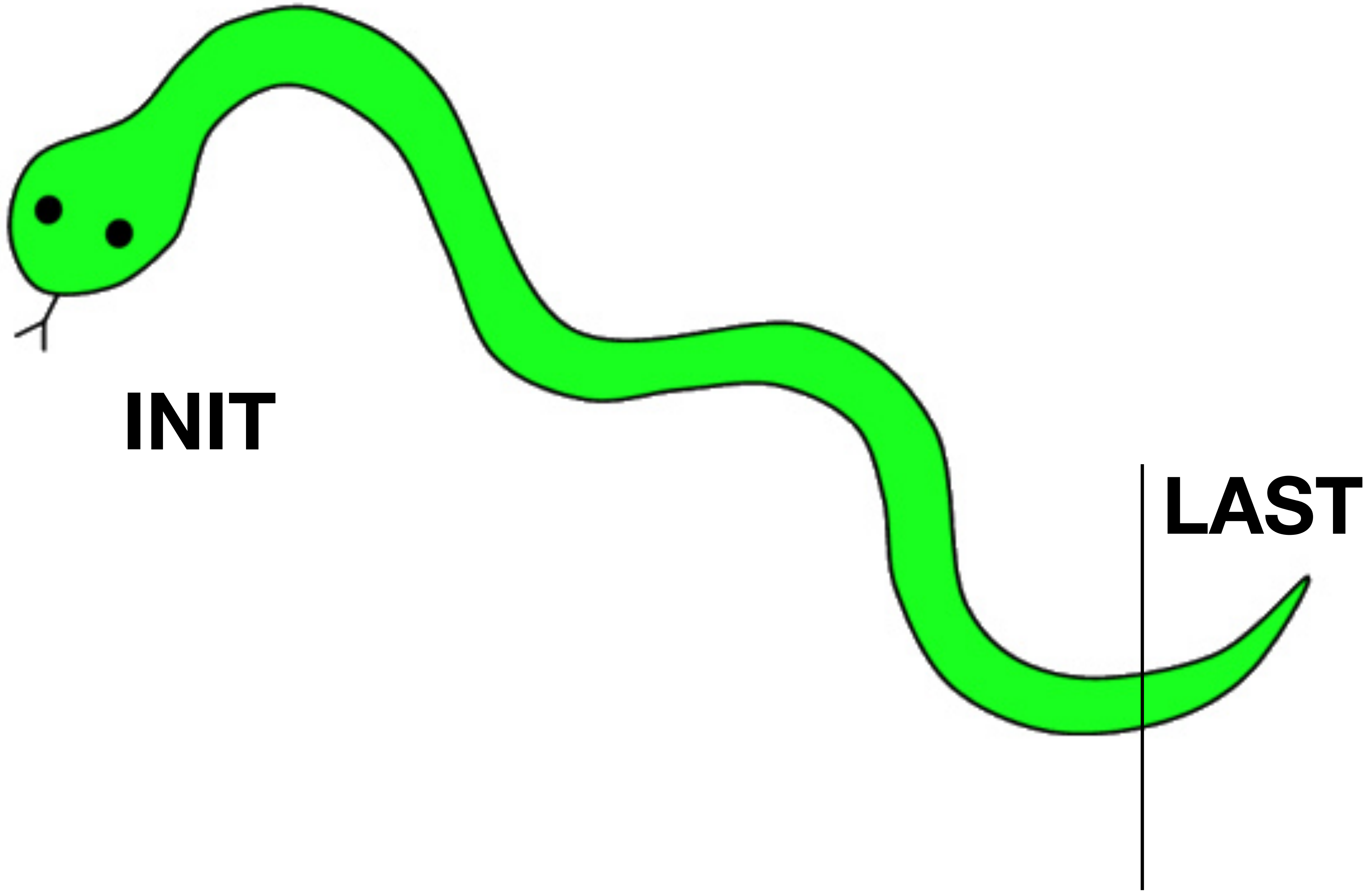


Accessing elements

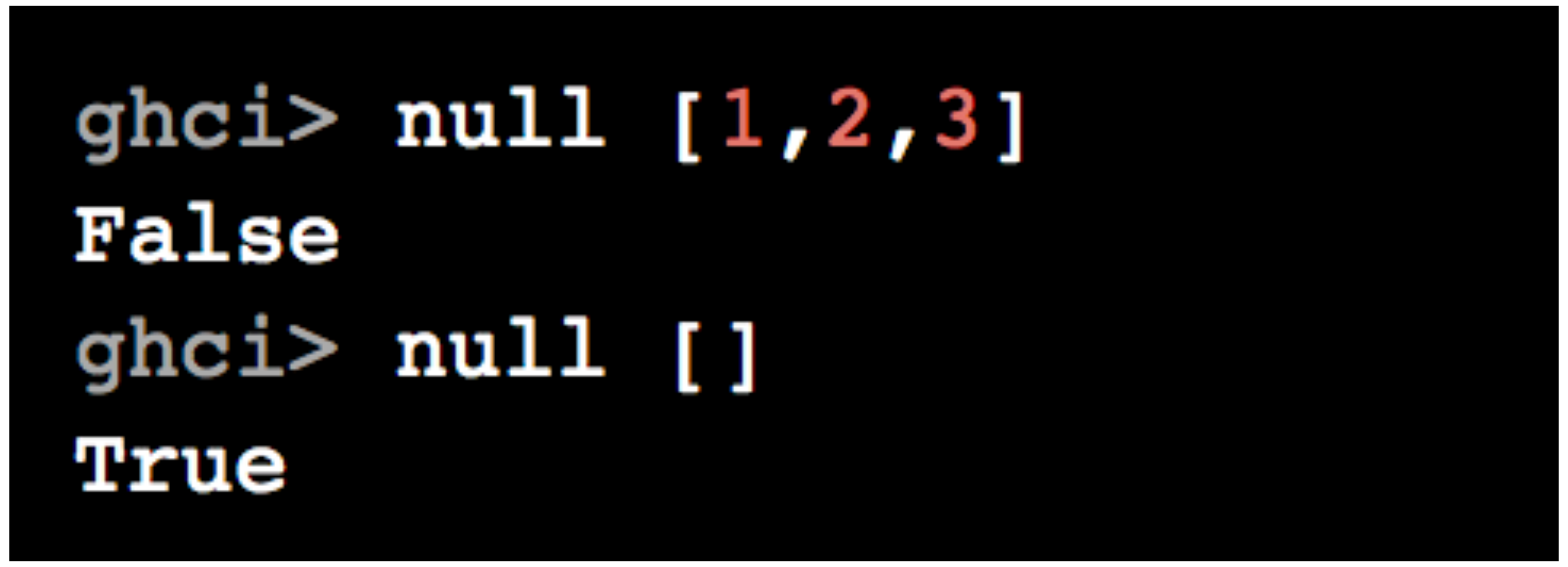
A list can be separated into **init** and **last**



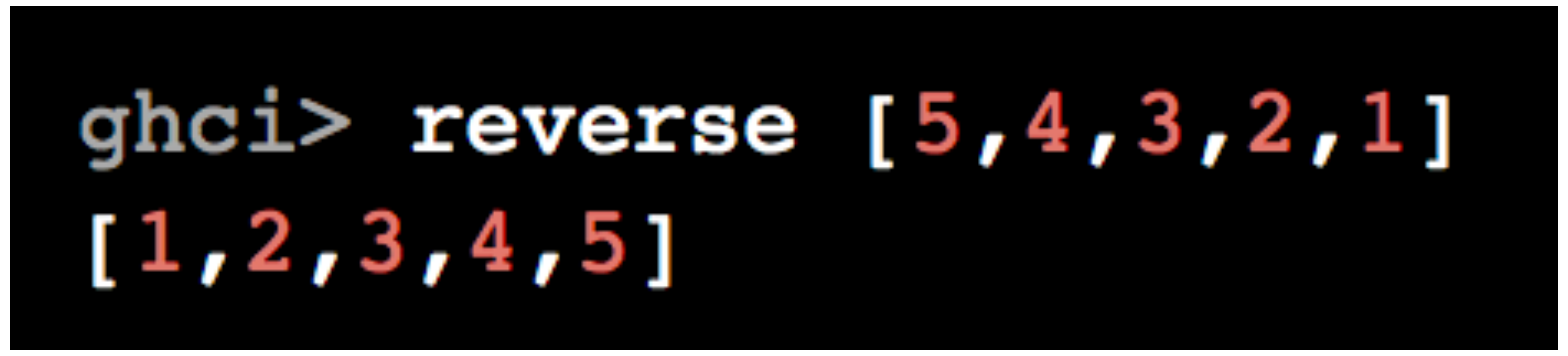
```
 > init [1,2,3]
=> [1,2]
 > last [1,2,3]
=> 3
```



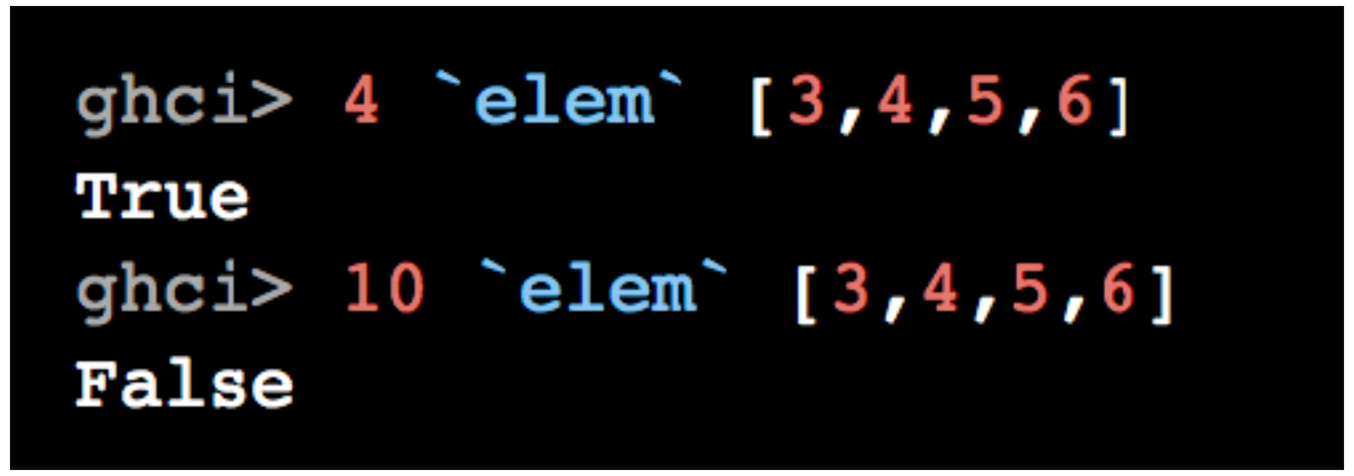
Additional Functions



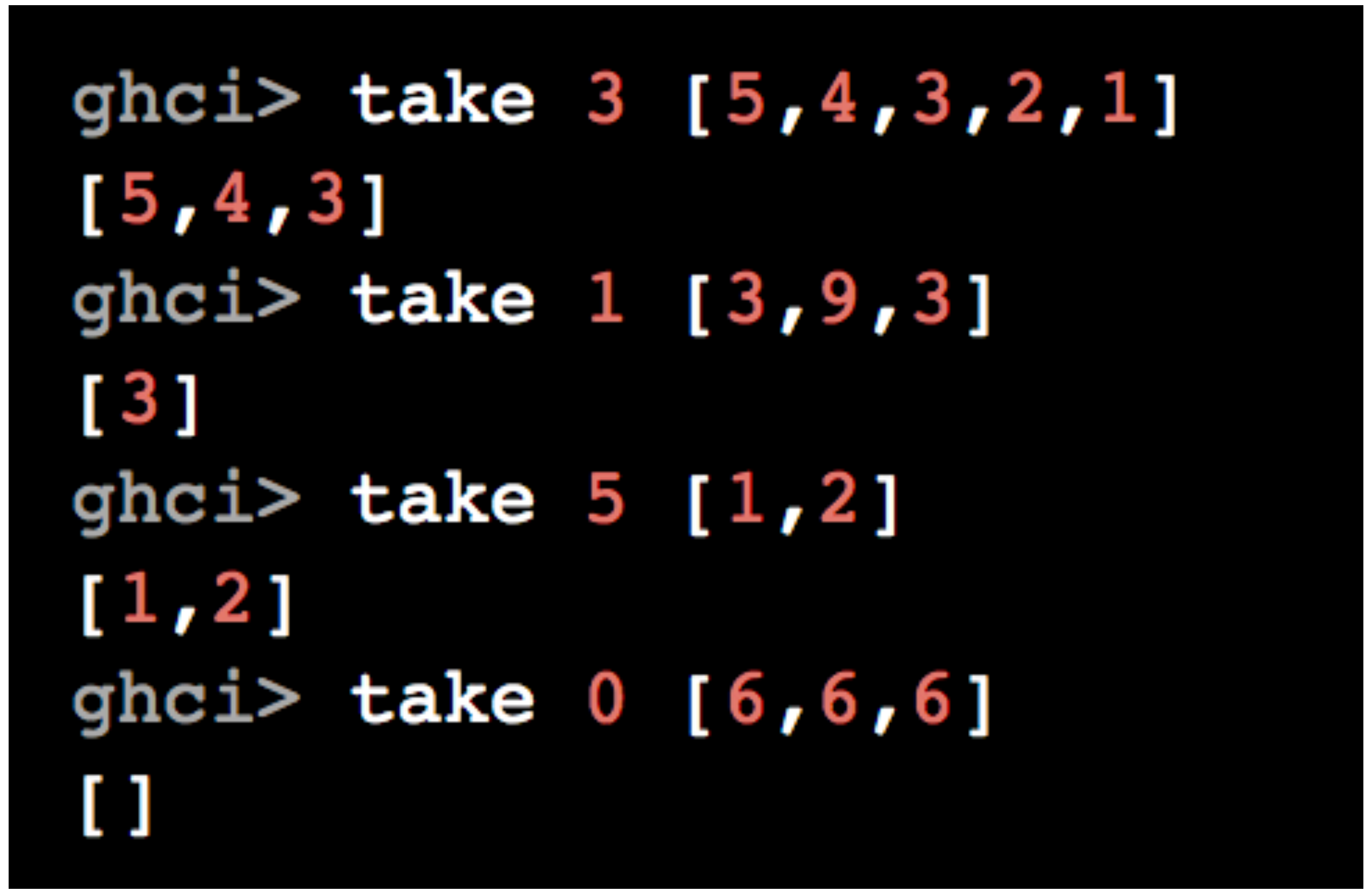
```
ghci> null [1,2,3]
False
ghci> null []
True
```



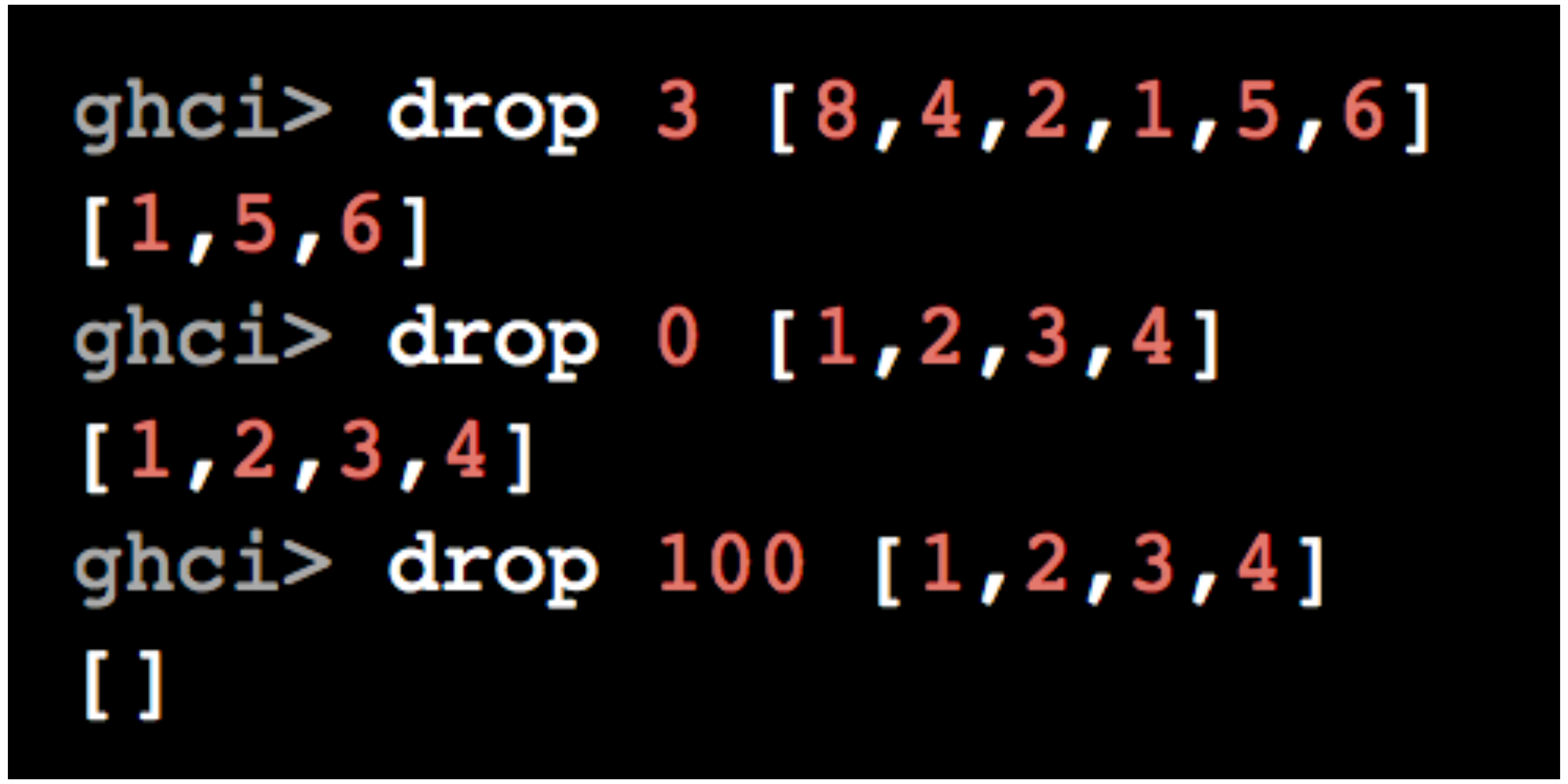
```
ghci> reverse [5,4,3,2,1]
[1,2,3,4,5]
```



```
ghci> 4 `elem` [3,4,5,6]
True
ghci> 10 `elem` [3,4,5,6]
False
```

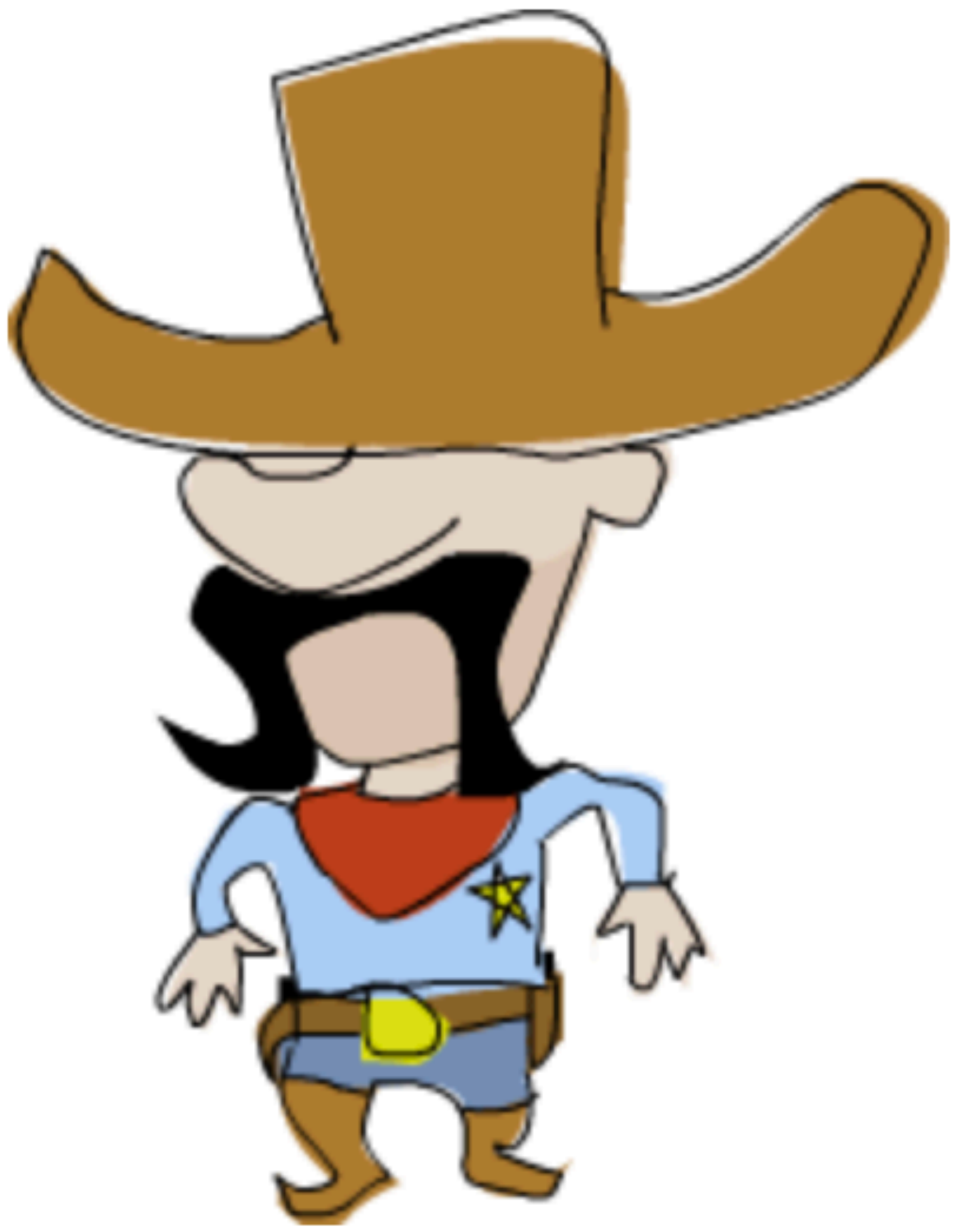


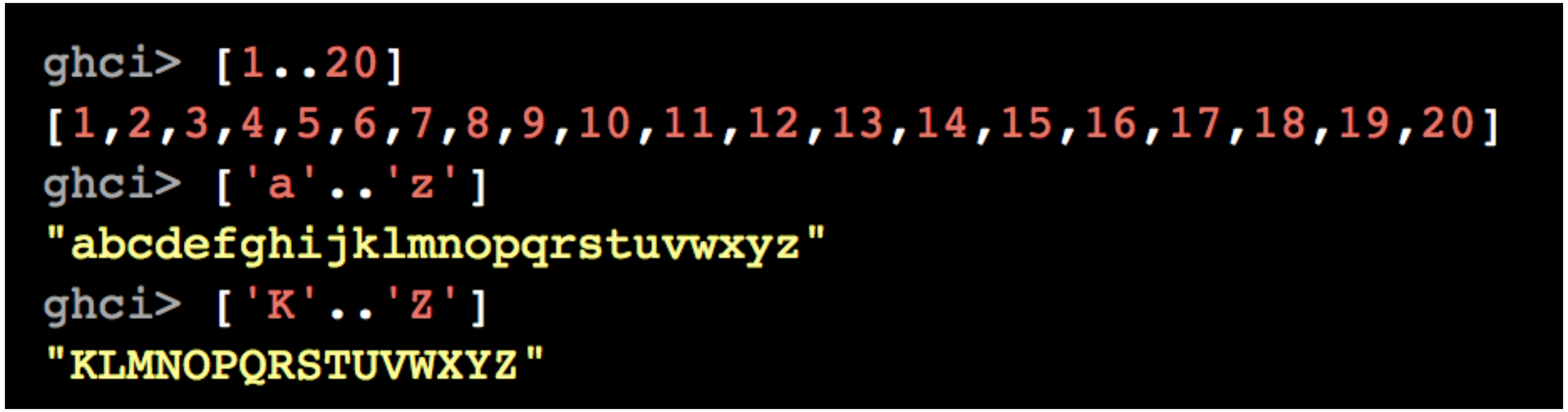
```
ghci> take 3 [5,4,3,2,1]
[5,4,3]
ghci> take 1 [3,9,3]
[3]
ghci> take 5 [1,2]
[1,2]
ghci> take 0 [6,6,6]
[]
```



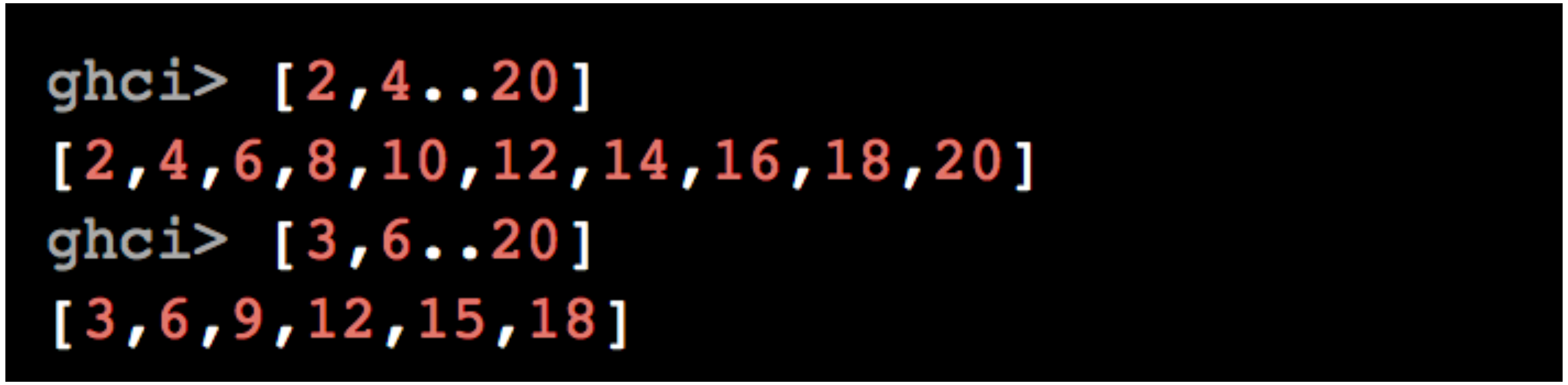
```
ghci> drop 3 [8,4,2,1,5,6]
[1,5,6]
ghci> drop 0 [1,2,3,4]
[1,2,3,4]
ghci> drop 100 [1,2,3,4]
[]
```

Ranges



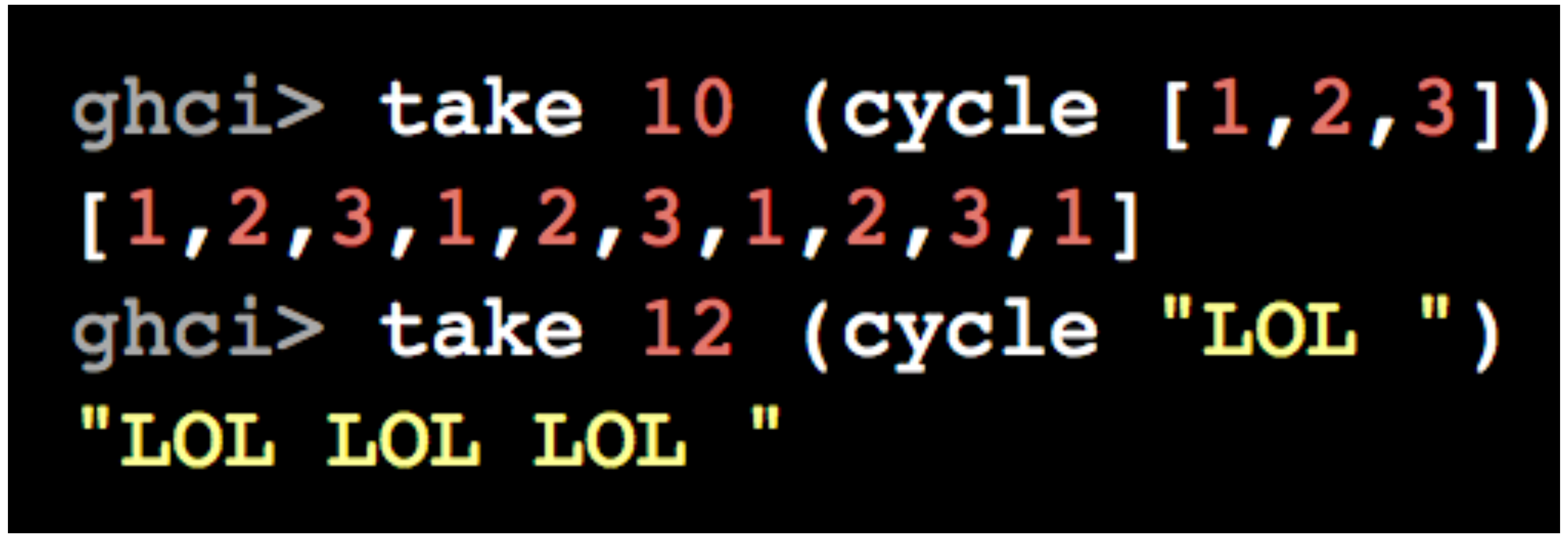


```
ghci> [1..20]
[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20]
ghci> ['a'..'z']
"abcdefghijklmnopqrstuvwxyz"
ghci> ['K'..'Z']
"KLMNOPQRSTUVWXYZ"
```

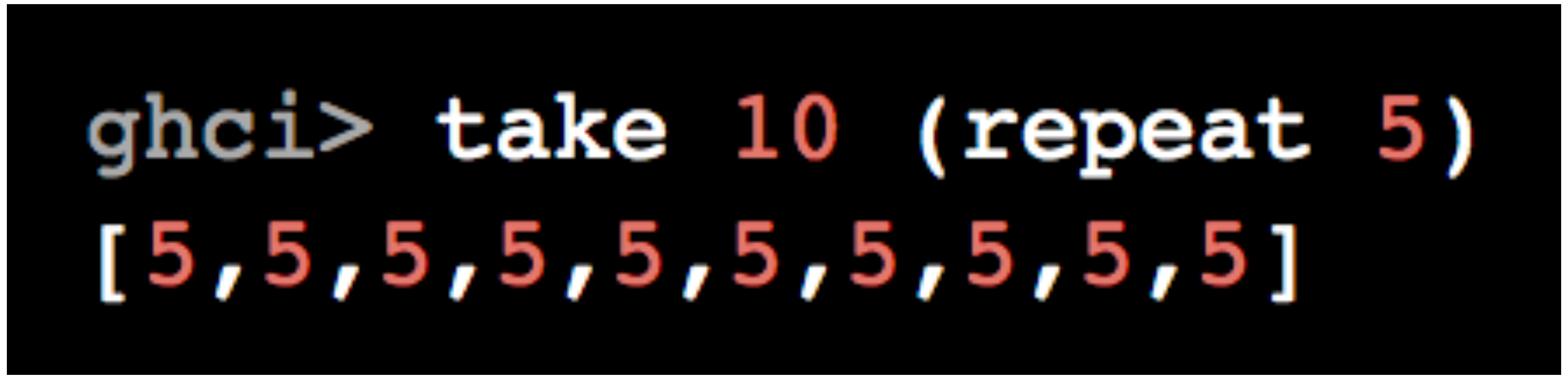


```
ghci> [2,4..20]
[2,4,6,8,10,12,14,16,18,20]
ghci> [3,6..20]
[3,6,9,12,15,18]
```

Cycle and Repeat

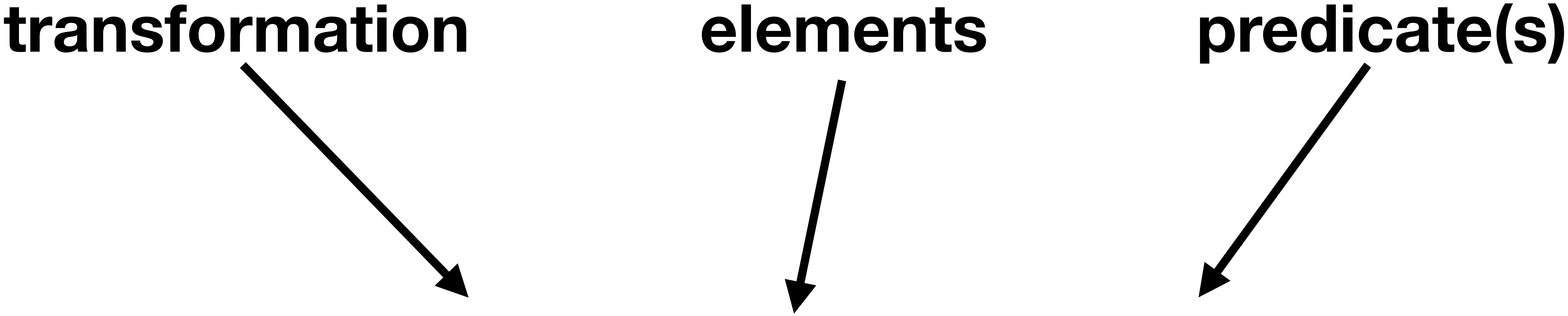


```
ghci> take 10 (cycle [1,2,3])
[1,2,3,1,2,3,1,2,3,1]
ghci> take 12 (cycle "LOL ")
"LOL LOL LOL "
```



```
ghci> take 10 (repeat 5)
[5,5,5,5,5,5,5,5,5,5]
```

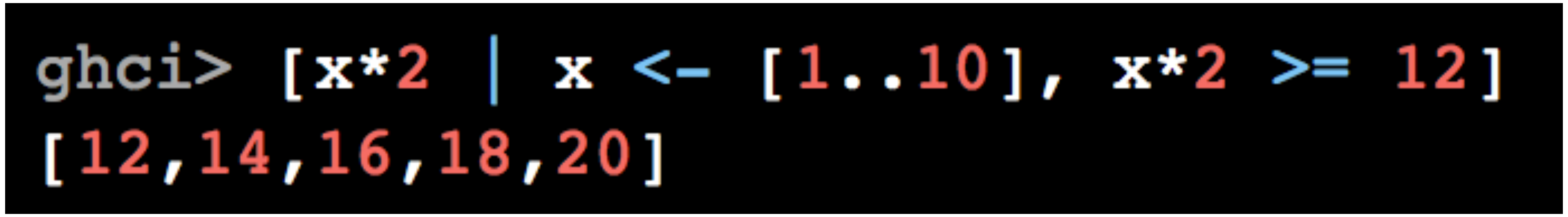
List comprehension



transformation

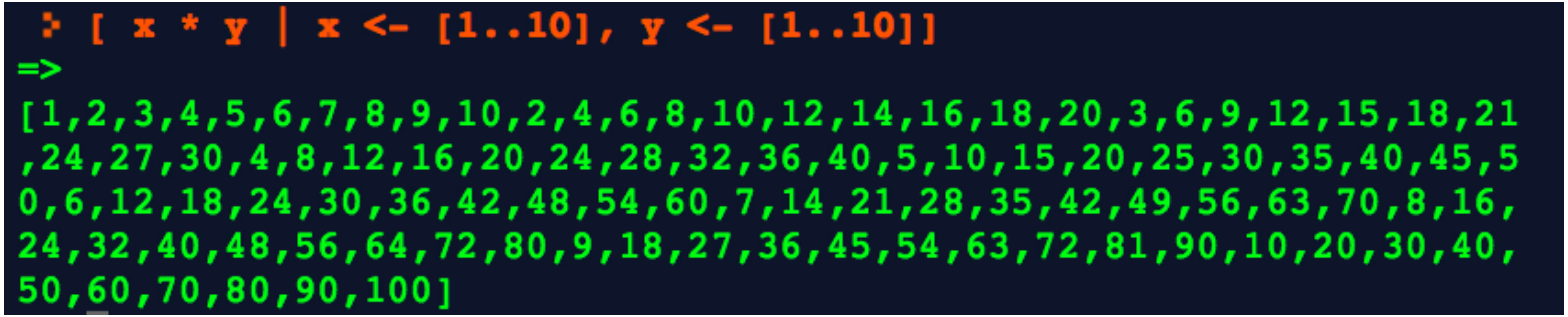
elements

predicate(s)

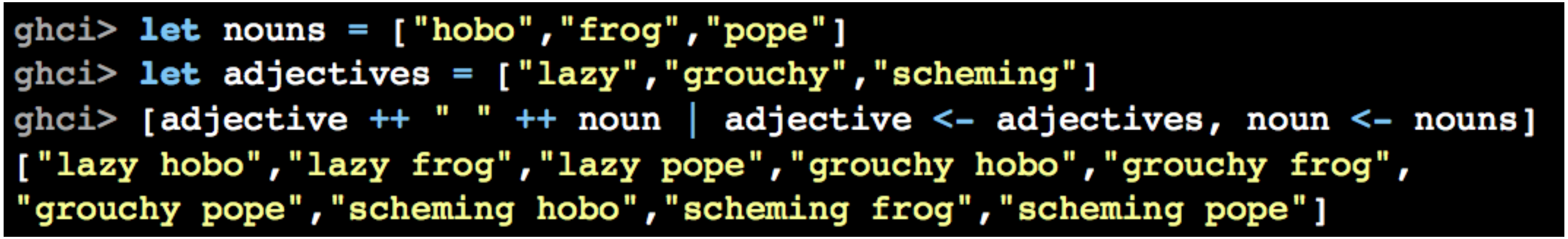


```
ghci> [x*2 | x <- [1..10], x*2 >= 12]
[12,14,16,18,20]
```

List comprehension

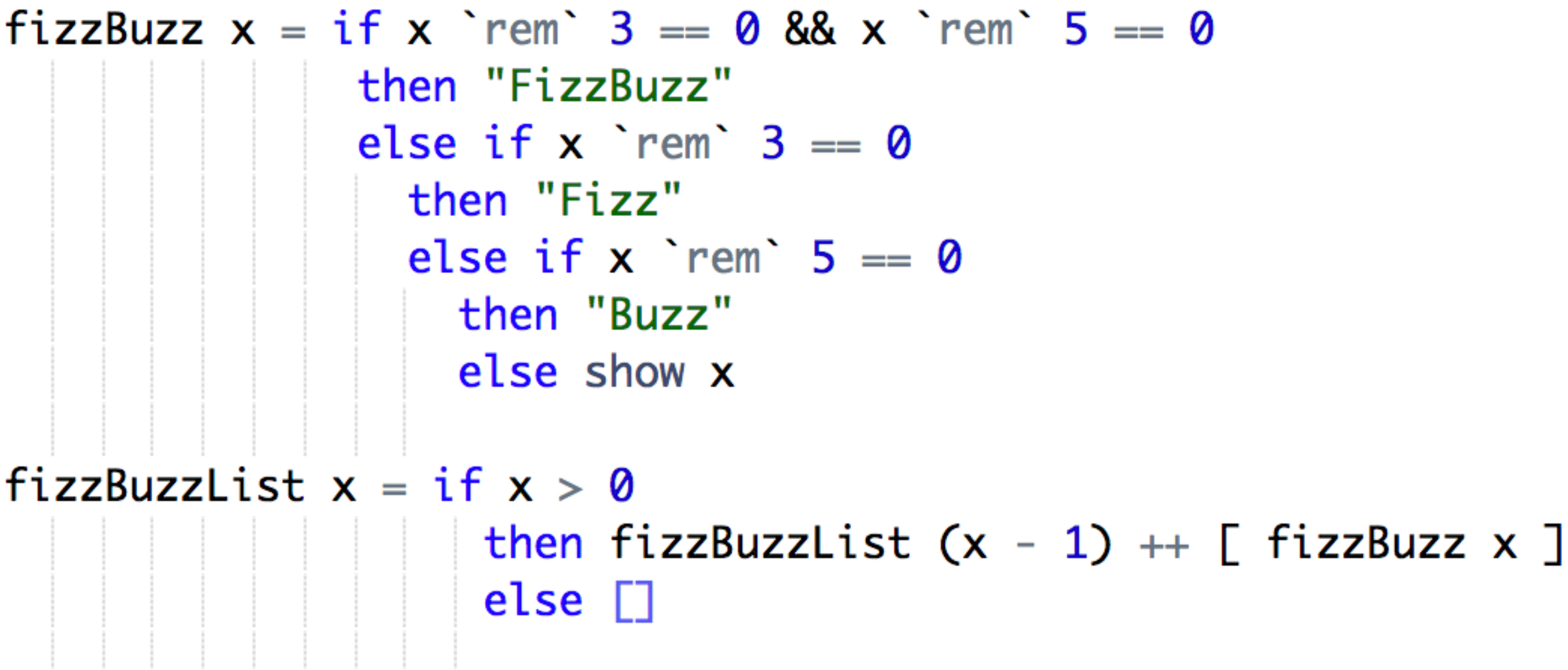


```
 > [ x * y | x <- [1..10], y <- [1..10]]
=>
[1,2,3,4,5,6,7,8,9,10,2,4,6,8,10,12,14,16,18,20,3,6,9,12,15,18,21
,24,27,30,4,8,12,16,20,24,28,32,36,40,5,10,15,20,25,30,35,40,45,5
0,6,12,18,24,30,36,42,48,54,60,7,14,21,28,35,42,49,56,63,70,8,16,
24,32,40,48,56,64,72,80,9,18,27,36,45,54,63,72,81,90,10,20,30,40,
50,60,70,80,90,100]
```



```
ghci> let nouns = ["hobo","frog","pope"]
ghci> let adjectives = ["lazy","grouchy","scheming"]
ghci> [adjective ++ " " ++ noun | adjective <- adjectives, noun <- nouns]
["lazy hobo","lazy frog","lazy pope","grouchy hobo","grouchy frog",
"grouchy pope","scheming hobo","scheming frog","scheming pope"]
```

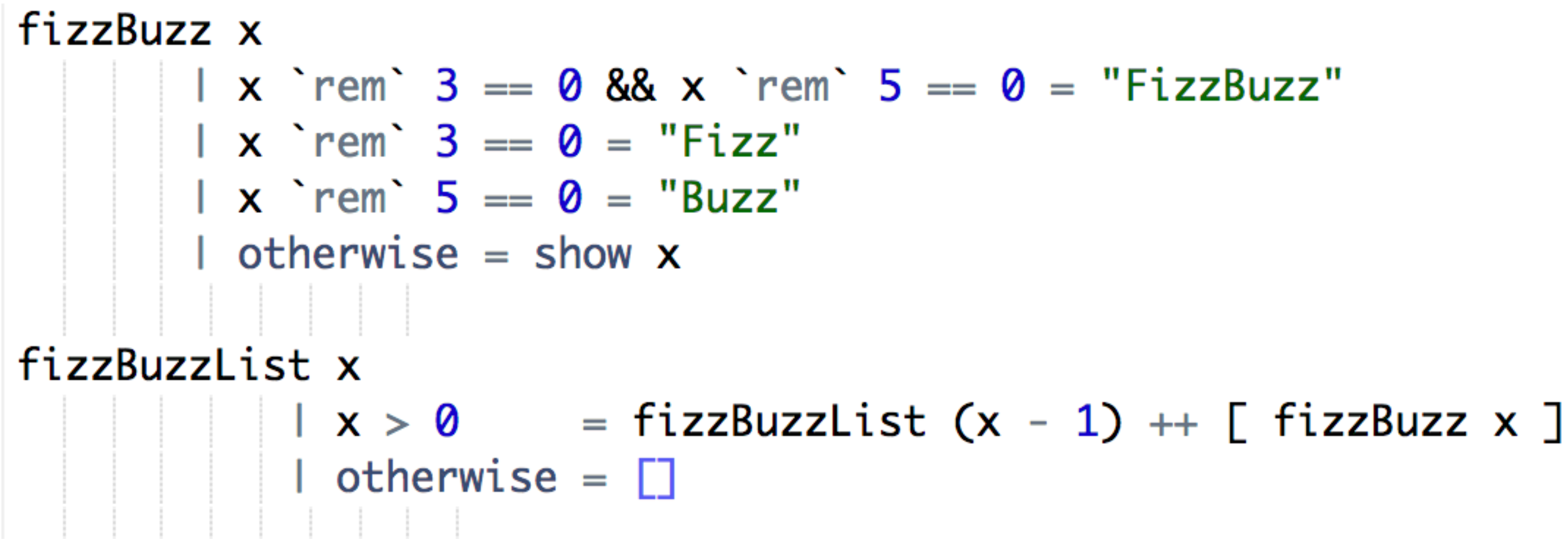
FizzBuzz



```haskell
fizzBuzz x = if x `rem` 3 == 0 && x `rem` 5 == 0
              then "FizzBuzz"
              else if x `rem` 3 == 0
                then "Fizz"
                else if x `rem` 5 == 0
                  then "Buzz"
                  else show x

fizzBuzzList x = if x > 0
                   then fizzBuzzList (x - 1) ++ [ fizzBuzz x ]
                   else []
```

FizzBuzz with Guards



```
fizzBuzz x
       | x `rem` 3 == 0 && x `rem` 5 == 0 = "FizzBuzz"
       | x `rem` 3 == 0 = "Fizz"
       | x `rem` 5 == 0 = "Buzz"
       | otherwise = show x

fizzBuzzList x
           | x > 0     = fizzBuzzList (x - 1) ++ [ fizzBuzz x ]
           | otherwise = []
```