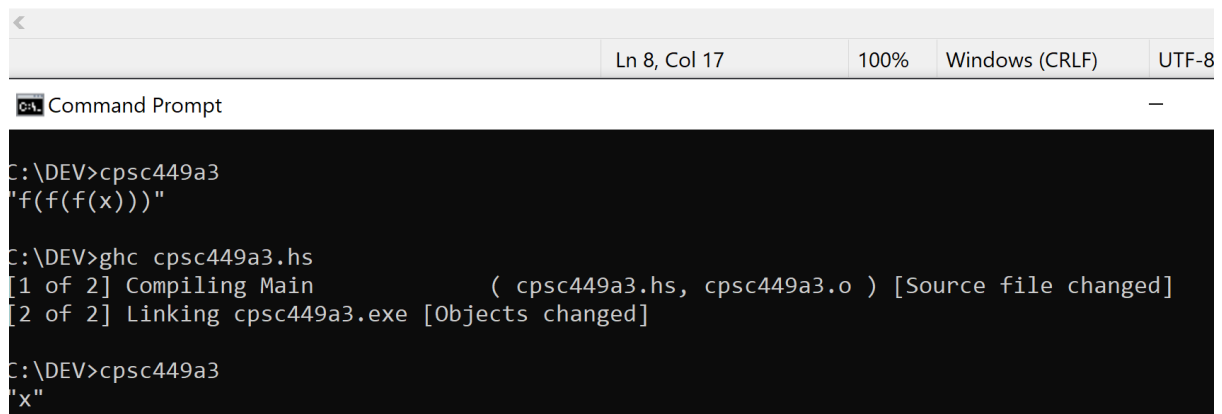


1

```
iter :: Int -> (a -> a) -> a -> a
iter 0 _ x = x
iter n f x = f (iter (n-1) f x)

main :: IO ()
main = do
    let result = iter 0 (\x -> "f(" ++ x ++ ")") "x"
    print result
```



```
<
Ln 8, Col 17    100%  Windows (CRLF)  UTF-8

Command Prompt

C:\DEV>cpssc449a3
f(f(f(x)))

C:\DEV>ghc cpssc449a3.hs
[1 of 2] Compiling Main             ( cpssc449a3.hs, cpssc449a3.o ) [Source file changed]
[2 of 2] Linking cpssc449a3.exe [Objects changed]

C:\DEV>cpssc449a3
"x"
```

This is higher order function that it takes integer and function takes value of any type and return the value with that type.

If it has value of 0 then it returns x

If $n - 1$ is bigger than 0 then it will recursion until it reaches $n - 1 = 0$.

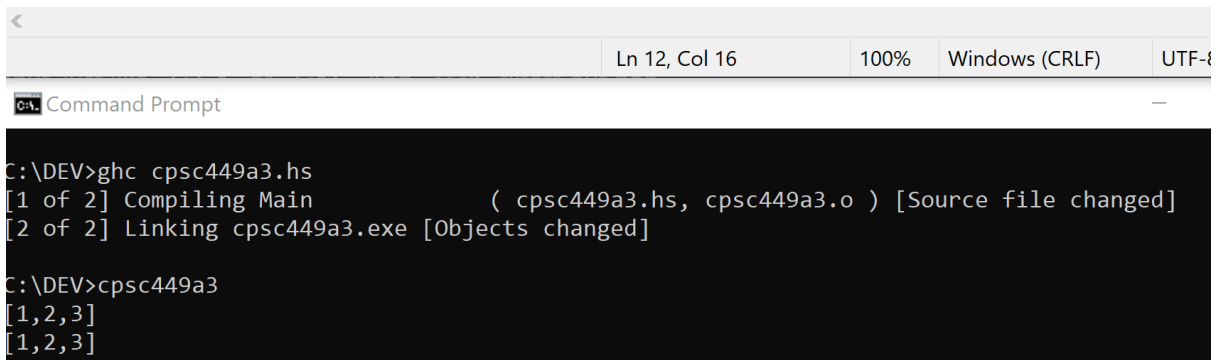
If I put $n = 0$ in the iter then it will just print x. Otherwise, it will print f() until $n - 1 = 0$ so at the last recursion, we will be putting x in the f().

2.

```
sec1 = (+1)
sec2 = (>=0)

fil1 = map sec1 . filter sec2
fil2 = filter (>0) . map(+1)

main = do
  let x = [-2, -1, 0, 1, 2]
  let result1 = fil1 x
  let result2 = fil2 x
  print result1
  print result2
```



```
C:\DEV>ghc cpssc449a3.hs
[1 of 2] Compiling Main ( cpssc449a3.hs, cpssc449a3.o ) [Source file changed]
[2 of 2] Linking cpssc449a3.exe [Objects changed]

C:\DEV>cpssc449a3
[1,2,3]
[1,2,3]
```

```
filterPositive :: [Integer] -> [Integer]
filterPositive = filter (> 0)
```

This is function that it filter out negative number so we can have positive number.

filter (>0) . map (+1), this filter out the negative numbers in the list and add 1 to the each element in the list.

Sec1 has operator section of (+1) and sec2 has operator section of (>=0).

Fil1 removes number that is less than 0 and only keeps the number greater or equal to 0. Then add 1 to the element in the list.

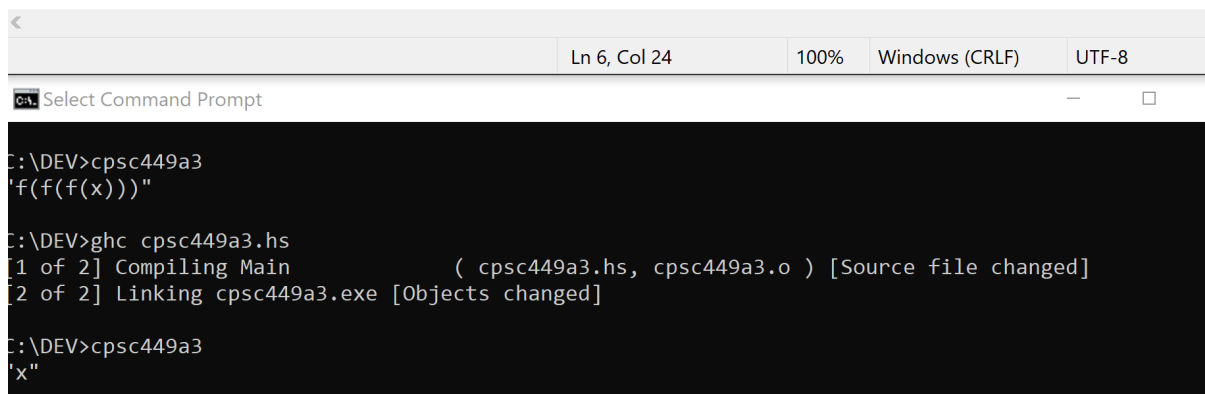
`[-2,-1,0,1,2] -> [0,1,2] -> [1,2,3]`

Fil2 add 1 to the each of element in the list and then removes number less than 0 and only keeps positive number.

`[-2,-1,0,1,2] -> [-1,0,1,2,3] -> [1,2,3]`

```
iter :: Int -> (a -> a) -> a -> a
iter n f x = foldr (.) id (replicate n f) x

main :: IO ()
main = do
    let result = iter 3 (\x -> "f(" ++ x ++ ")") "x"
    print result
```



The screenshot shows a Windows command prompt window titled "Select Command Prompt". The window has a status bar at the top indicating "Ln 6, Col 24", "100%", "Windows (CRLF)", and "UTF-8". The command prompt shows the following sequence of commands and output:

```
C:\DEV>cpsc449a3
'f(f(f(x)))'

C:\DEV>ghc cpsc449a3.hs
[1 of 2] Compiling Main             ( cpsc449a3.hs, cpsc449a3.o ) [Source file changed]
[2 of 2] Linking cpsc449a3.exe [Objects changed]

C:\DEV>cpsc449a3
'x'
```

First create the list of n copies of f by using `replicate` and used `foldr` to compose n copies of f . `[f.f...f]` created the list. We used `'.'` to the list that replicate n f will be done right to left. Then composition will become $n - 1$ copies of f so that n copies of f is then applied to x . Then will be equal to the composition $f . f . \dots . f$ applied n times

```

multiplyCurry :: (Int, Int, Int) -> Int
multiplyCurry (x, y, z) = x * y * z

multiplyUncurry :: Int -> Int -> Int -> Int
multiplyUncurry x y z = x * y * z

multiply :: Int -> Int -> Int
multiply x y = x * y

uncurry' :: (a -> b -> c) -> ((a, b) -> c)
uncurry' g = \ (x, y) -> g x y

curry3 :: ((a, b, c) -> d) -> (a -> b -> c -> d)
curry3 g = \ x y z -> g (x,y,z)

uncurry3 :: (a -> b -> c -> d) -> ((a, b, c) -> d)
uncurry3 g = \ (x, y, z) -> g x y z

uncurriedMult :: (Int, Int, Int) -> Int
uncurriedMult = uncurry3 multiplyUncurry

curried3Mult :: Int -> Int -> Int -> Int
curried3Mult = curry3 multiplyCurry

uncurried2Mult :: (Int, Int) -> Int
uncurried2Mult = uncurry' multiply

main = do
  print (curried3Mult 3 4 5)
  print (uncurriedMult (3, 4, 5))
  print (uncurried2Mult (uncurry' multiply (3,4), 5))

```

```

^ [1 of 2] Compiling Main ( cpsec449a3
[2 of 2] Linking cpsec449a3.exe [Objects changed]

C:\DEV>cpsec449a3
60
60
60

C:\DEV>

```

```

curry' :: ((a, b) -> c) -> (a -> b -> c)
curry' g x y = g (x, y)

```

```

uncurry' :: (a -> b -> c) -> ((a, b) -> c)
uncurry' g (x, y) = g x y

```

Curry and uncurry works with 2 arguments, curry3 and uncurry3 will work with 3 arguments.

```

Curry3 :: ((a,b,c) -> d) -> (a -> b -> c -> d)
Curry3 g = \x y z -> g (x,y,z)

```

Curry3 takes function g and takes three argument a, b, c and then return value d.

```

Uncurry3 :: (a -> b -> c -> d) -> ((a,b,c) -> d)
Uncurry3 g = \ (x, y, z) -> g x y z

```

Uncurry3 takes curried function g that takes three arguments of type of a, b, c and return d.

Since curry3 is changing uncurry to curry so we need to have uncurry multiply to print the result. And uncurry3 is changing curry to uncurry so need another curry multiply function to print the result

Curry3 converts a function with three arguments into curried form.

Uncurry3 converts a function with three argument into uncurried form.

Both perform the analogue of curry and uncurry since way of design is same.

We can use curry and uncurry in curry3 and uncurry3 definition since I used curry two times to get curry3.

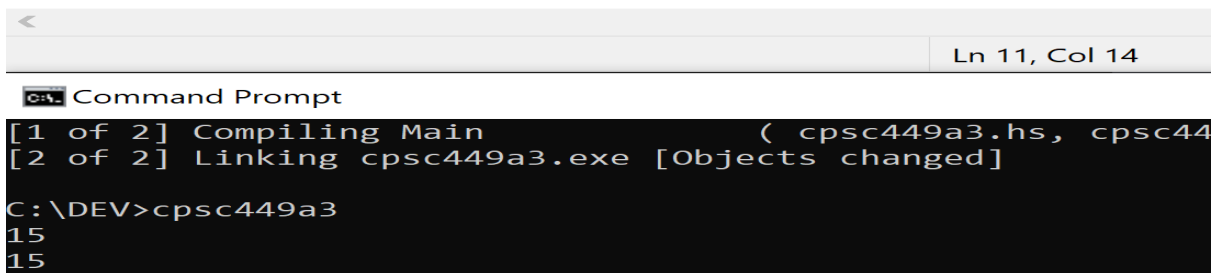
5.

```
curryList :: ([a] -> d) -> (a -> [a] -> d)
curryList f x xs = f (x:xs)
```

```
sumList :: [Int] -> Int
sumList = foldl (+) 0
```

```
sumCurried :: Int -> [Int] -> Int
sumCurried = curryList . sumList
```

```
main :: IO ()
main = do
    print $ sumList [1, 2, 3, 4, 5]
    print $ sumCurried 1 [2, 3, 4, 5]
```

A screenshot of a Haskell IDE window. The top bar shows a back arrow and "Ln 11, Col 14". Below the bar, the title bar says "C:\DEV Command Prompt". The main area shows the output of the program: "[1 of 2] Compiling Main (cpsc449a3.hs, cpsc449a3.o)" and "[2 of 2] Linking cpsc449a3.exe [Objects changed]". Below this, the command prompt shows "C:\DEV>cpsc449a3" followed by the output "15" and "15".

```
< Ln 11, Col 14
C:\DEV Command Prompt
[1 of 2] Compiling Main ( cpsc449a3.hs, cpsc449a3.o )
[2 of 2] Linking cpsc449a3.exe [Objects changed]
C:\DEV>cpsc449a3
15
15
```

It takes function of list of type a and return type d. And it returns function that takes a and list a and return d.

We can use the curry and uncurry since we can take list as argument and return it as list.