

Background – Functions (1C)

Copyright (c) 2016 - 2018 Young W. Lim.

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

Please send corrections (or suggestions) to youngwlim@hotmail.com.

This document was produced by using LibreOffice.

Based on

<http://learnyouahaskell.com/making-our-own-types-and-typeclasses#the-functor-typeclass>

<http://learnyouahaskell.com/functors-applicative-functors-and-monoids>

Haskell in 5 steps

https://wiki.haskell.org/Haskell_in_5_steps

Function Definition

Function Definition I.

```
square x = x * x
```

- **function type** is inferred → not efficient

Type Inference

Function Definition II.

```
square :: Double -> Double
```

```
square x = x * x
```

– **function type declaration**

– **function definition**

- **function type** **declaration**
- **function** **definition**

<http://www.toves.org/books/hsfun/>

Type Declaration

Type Declaration

the declaration of an identifier's type

identifier name :: type name ...

identifier names
(including function
identifiers) must
always begin with a
lower-case letter

type names in
Haskell always
begin with a
capital letter

<http://www.toves.org/books/hsfun/>

Function Types and Type Classes

Function Definition I.

```
square x = x * x
```

function definition

```
=
```

Function Definition II.

```
square :: Double -> Double
```

```
square x = x * x
```

function definition

- **function type declaration**

```
=
```

type class – a set of types

- **function type 1**
- **function type 2**
-
- **function type n**

Requirements

Subclasses

<http://www.toves.org/books/hsfun/>

Curry & Uncurry

$f :: a \rightarrow b \rightarrow c$ the curried form of $g :: (a, b) \rightarrow c$

$f = \text{curry } g$

$g = \text{uncurry } f$

$f \ x \ y = g \ (x, y)$

the curried form is usually more convenient because it allows **partial application**.

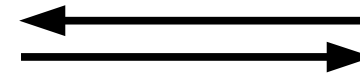
all functions are considered **curried**

all functions take **just one argument**

the curried form

$f :: a \rightarrow b \rightarrow c$

currying



$g :: (a, b) \rightarrow c$

uncurrying

$f \ x \ y$

$g \ (x, y)$

<https://wiki.haskell.org/Currying>

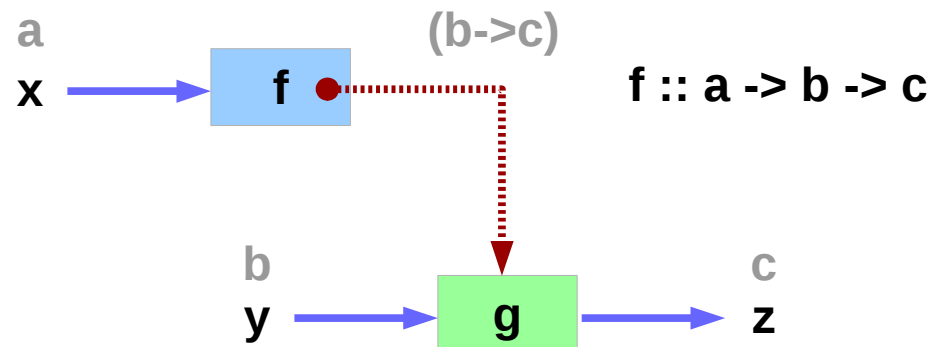
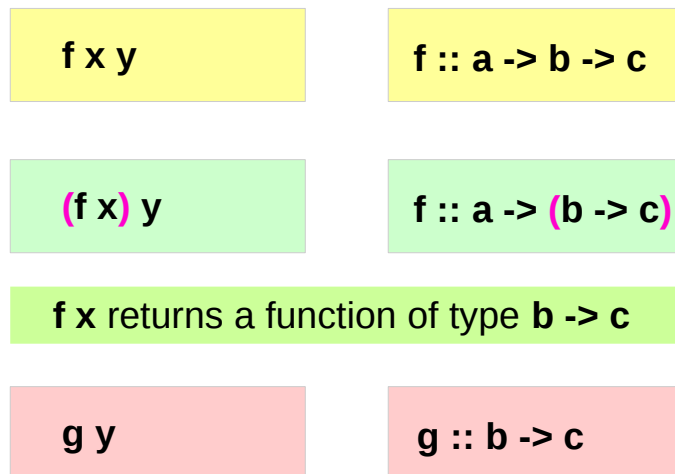
Functions : First-class Data Types

functions are **first-class data types**

Haskell **treats functions as regular data**,

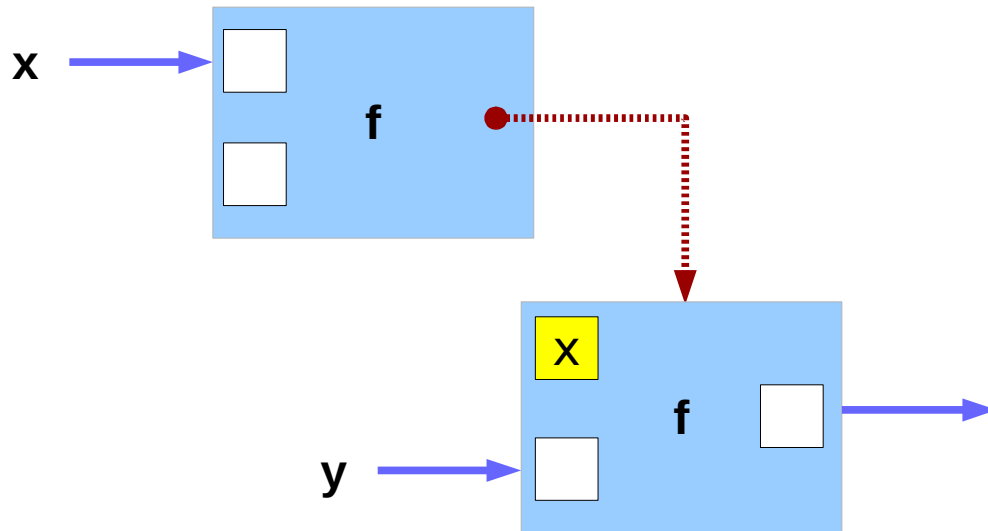
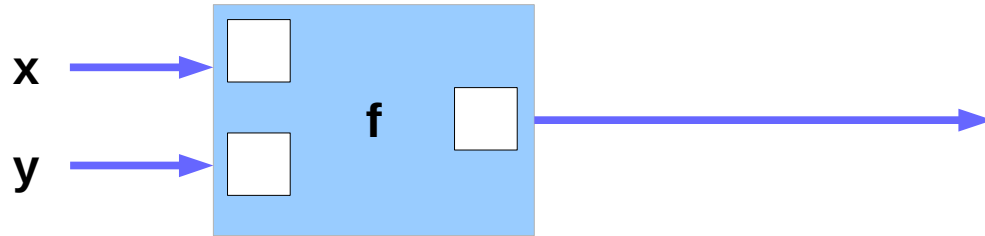
just like integers, or floating-point values, or other types.

- a function can take other functions as **parameters**
- a function takes a **parameter** and produces **another function** (curried function)



<http://www.toves.org/books/hsfun/>

Currying Examples



<http://learnyouahaskell.com/functors-applicative-functors-and-monoids>

Polymorphic Functions

specific types vs. arbitrary types

a **polymorphic** functions – an **abstract** type
each type variable is generally a **lower-case letter**.

Example) A translate function

takes a function **f** and a distance **d**

returns a new function **g**

that is **f** "translated" **d** units to the right

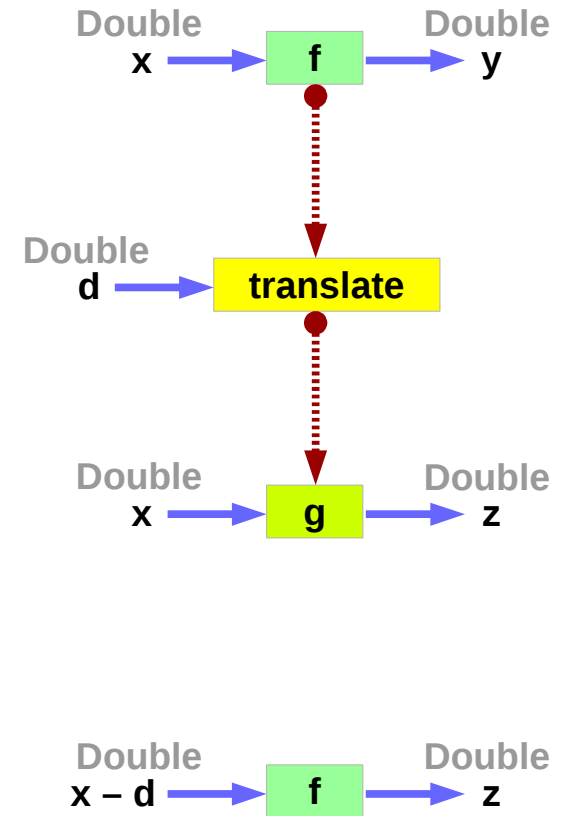
<http://www.toves.org/books/hsfun/>

Polymorphic Function Examples

`translate :: (Double -> Double) -> Double -> (Double -> Double)`

`translate f d = g` where `g x = f (x - d)`

`translate :: (Double -> a) -> Double -> (Double -> a)`



<http://www.toves.org/books/hsfun/>

Currying

Currying recursively transforms
a function that takes multiple arguments
into a function that takes just a single argument and
returns another function if any arguments are still needed.

$f :: a \rightarrow b \rightarrow c$

$f\ x\ y$

$f :: a \rightarrow b \rightarrow c$

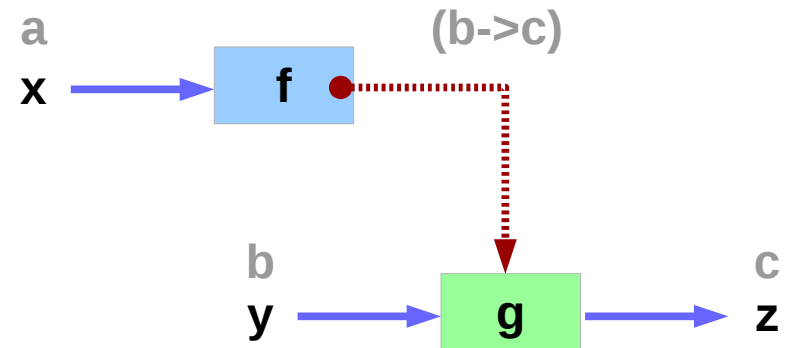
$(f\ x)\ y$

$f :: a \rightarrow (b \rightarrow c)$

$g\ y$

$g :: b \rightarrow c$

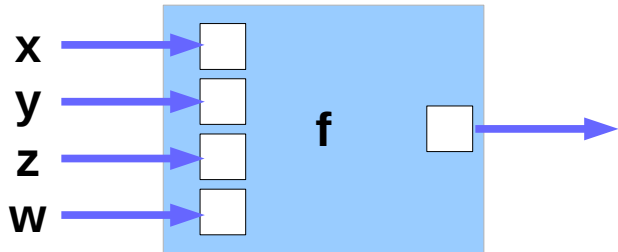
$f :: a \rightarrow b \rightarrow c$



<https://wiki.haskell.org/Currying>

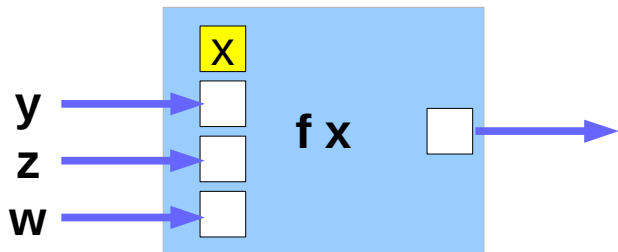
<http://learnyouahaskell.com/functors-applicative-functors-and-monoids>

Partially Applied Functions – f , $(f\ x)$



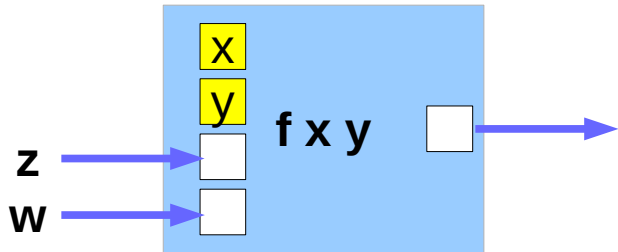
$f :: a \rightarrow b \rightarrow c \rightarrow d \rightarrow e$
 $f\ x\ y\ z\ w = \dots$

$(f\ x)\ y\ z\ w$



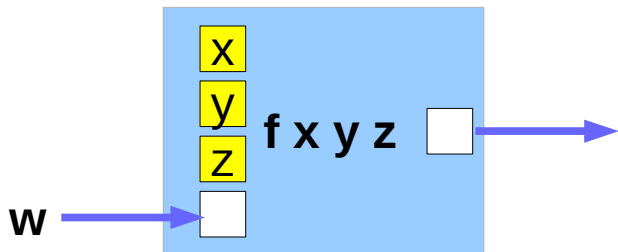
$g1 :: b \rightarrow c \rightarrow d \rightarrow e$
 $g1\ y\ z\ w = \dots$

Partially Applied Functions – (f x y), (f x y z)



(f x y) z w

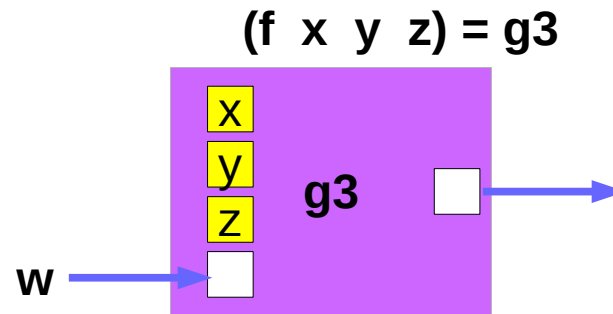
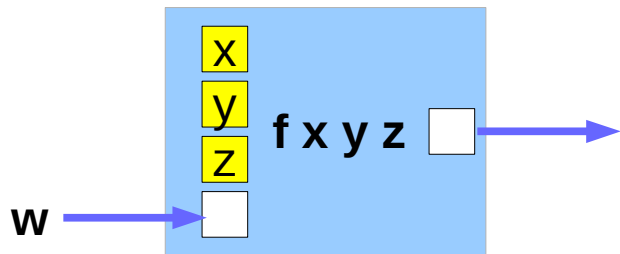
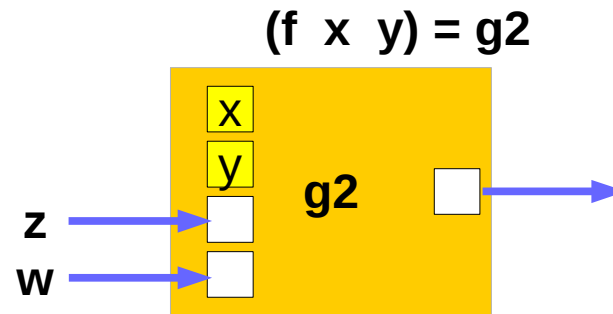
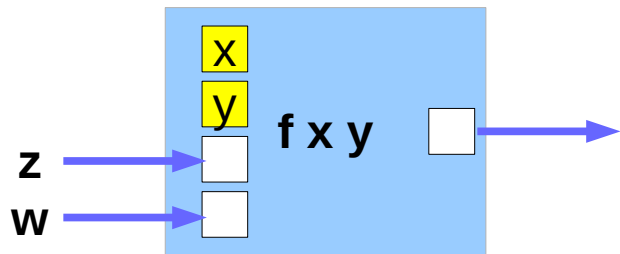
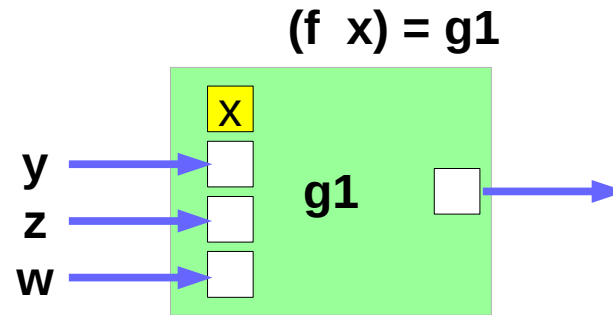
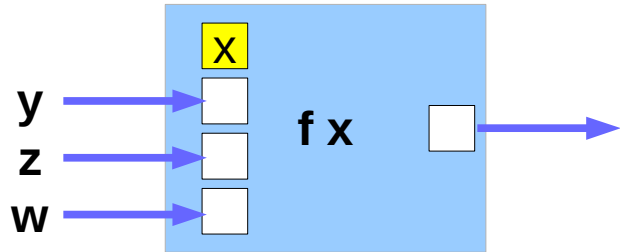
$g2 :: c \rightarrow d \rightarrow e$
 $g2\ z\ w = \dots$



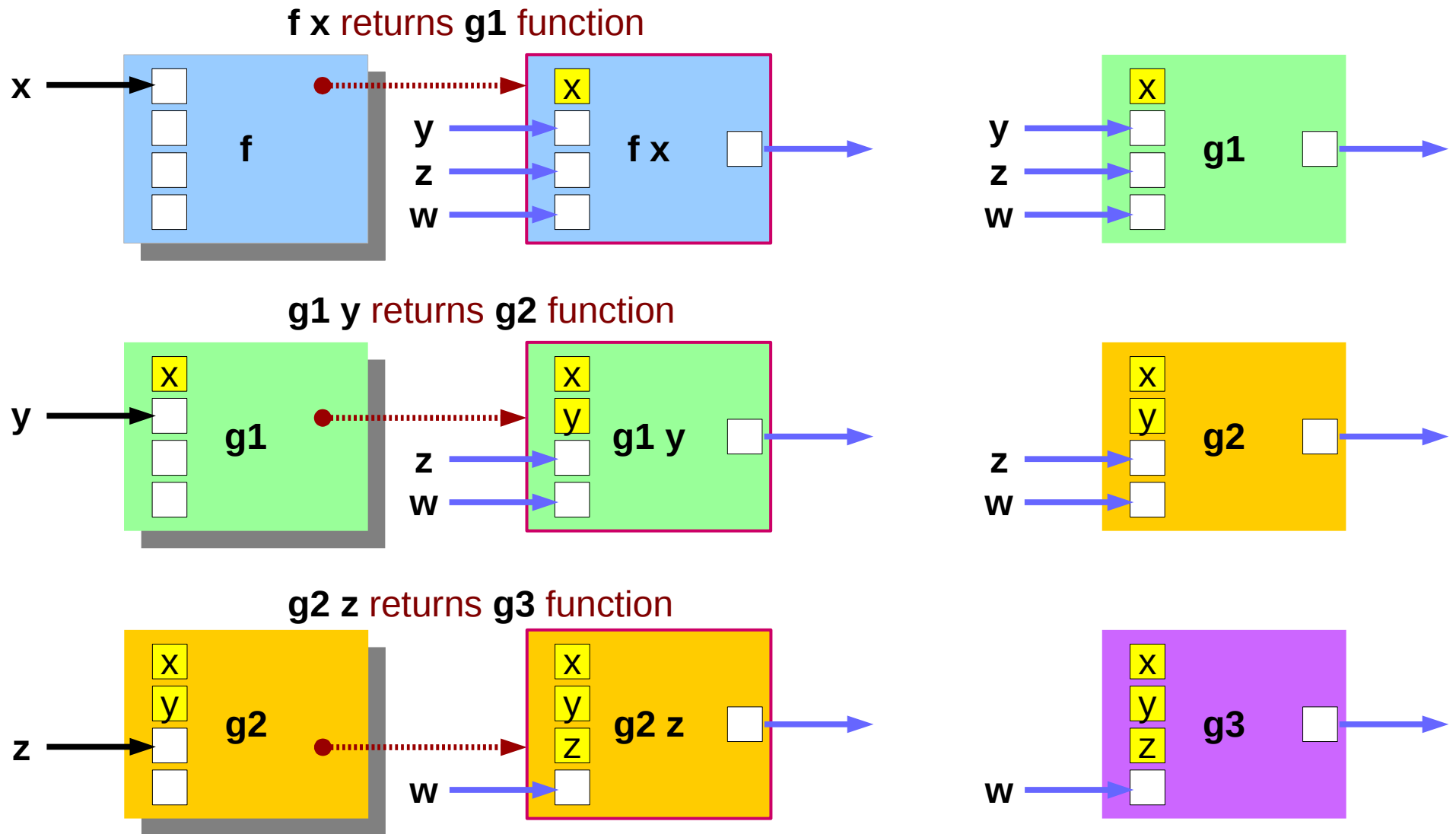
(f x y z) w

$g3 :: d \rightarrow e$
 $g3\ w = \dots$

Partially Applied Functions – g1, g2, g3



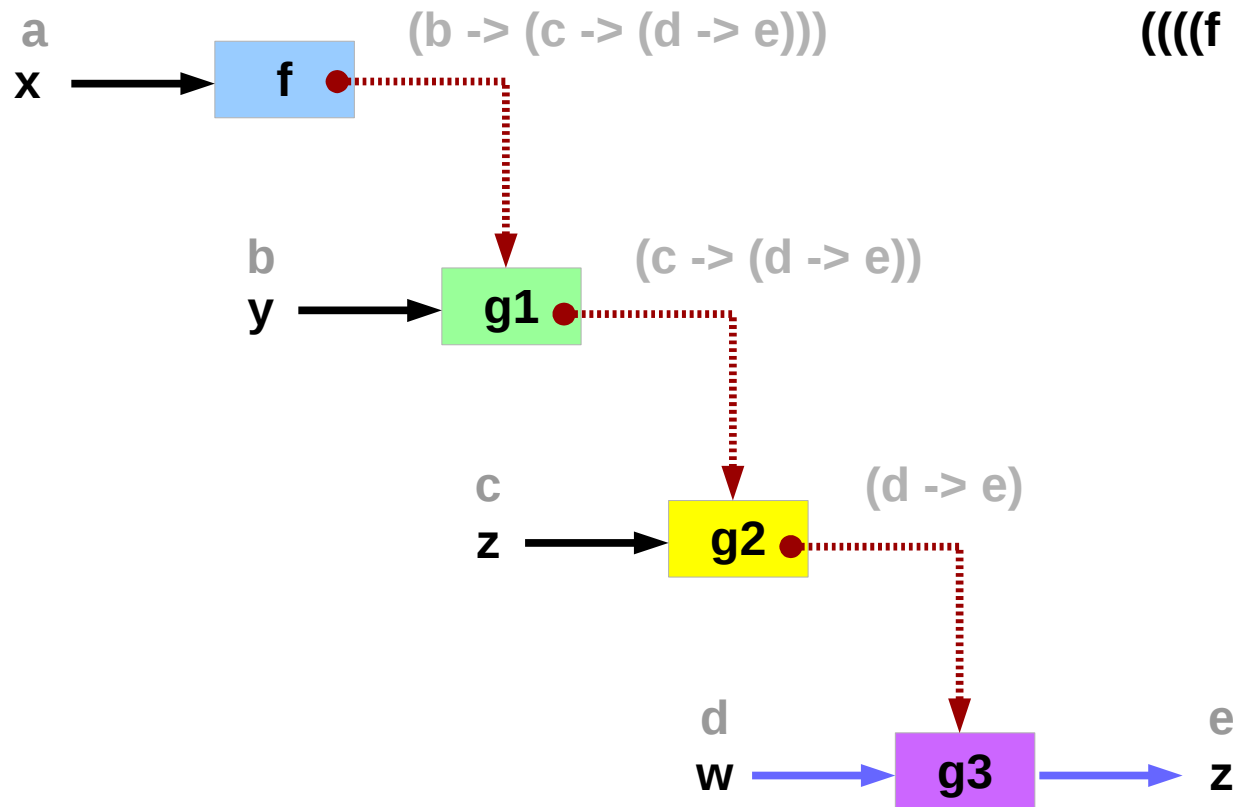
Returning Functions



Currying Examples

$f :: a \rightarrow b \rightarrow c \rightarrow d \rightarrow e$

$f :: a \rightarrow (b \rightarrow (c \rightarrow (d \rightarrow e)))$



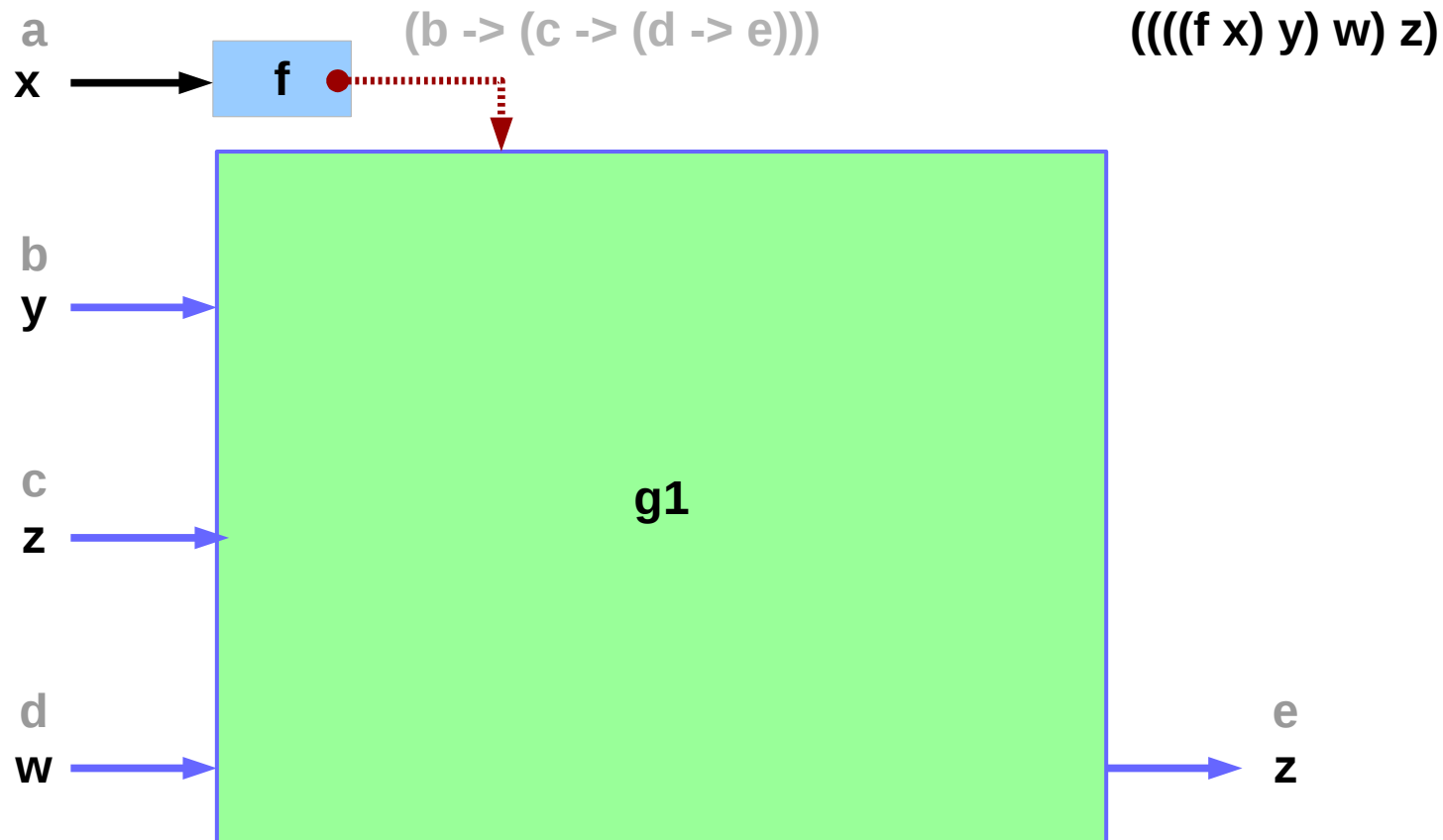
$((((f\ x)\ y)\ z)\ w)$

<http://learnyouahaskell.com/functors-applicative-functors-and-monoids>

Currying Examples

$f :: a \rightarrow b \rightarrow c \rightarrow d \rightarrow e$

$f :: a \rightarrow (b \rightarrow (c \rightarrow (d \rightarrow e)))$



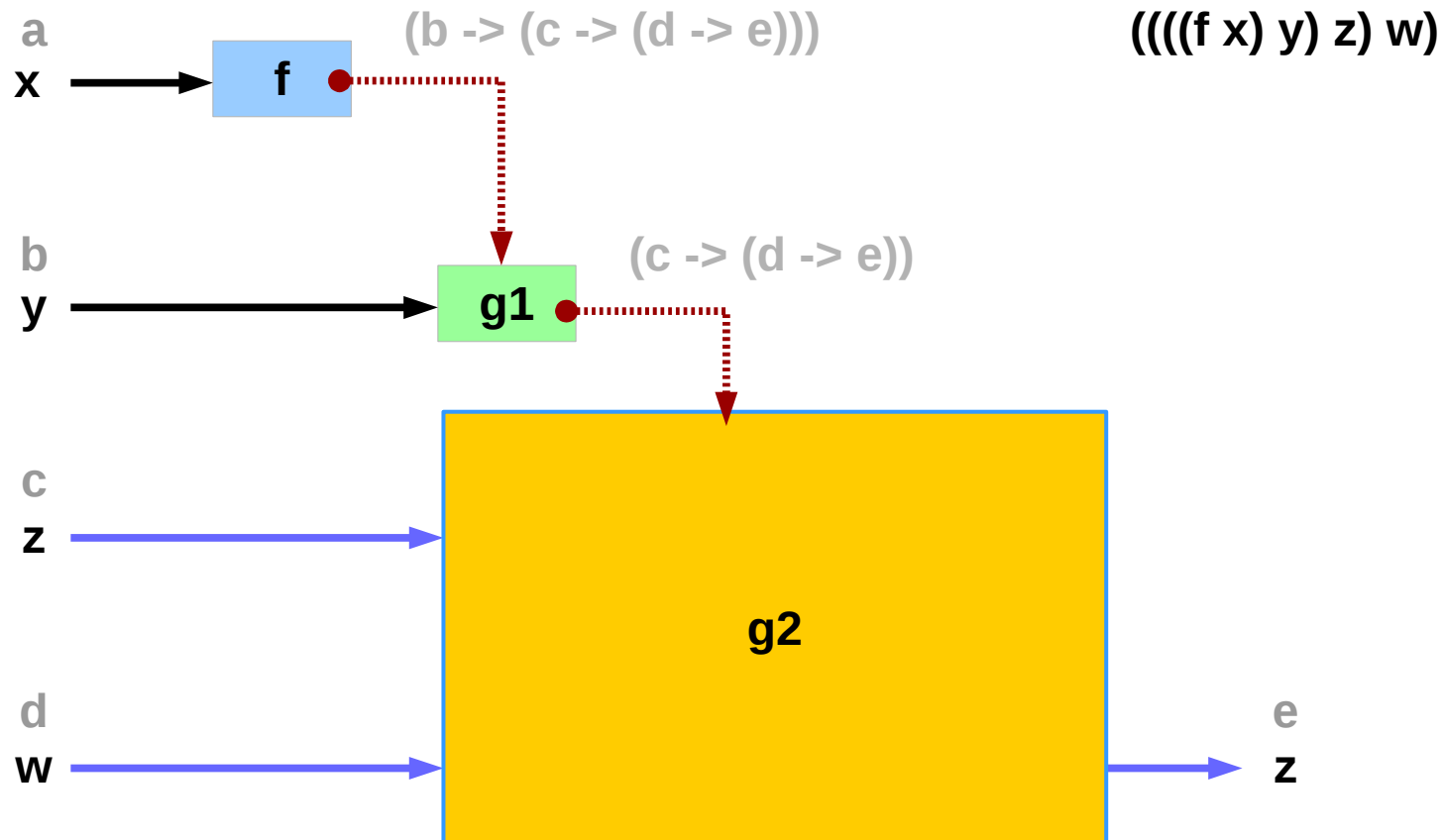
$((((f\ x)\ y)\ w)\ z)$

<http://learnyouahaskell.com/functors-applicative-functors-and-monoids>

Currying Examples

$f :: a \rightarrow b \rightarrow c \rightarrow d \rightarrow e$

$f :: a \rightarrow (b \rightarrow (c \rightarrow (d \rightarrow e)))$



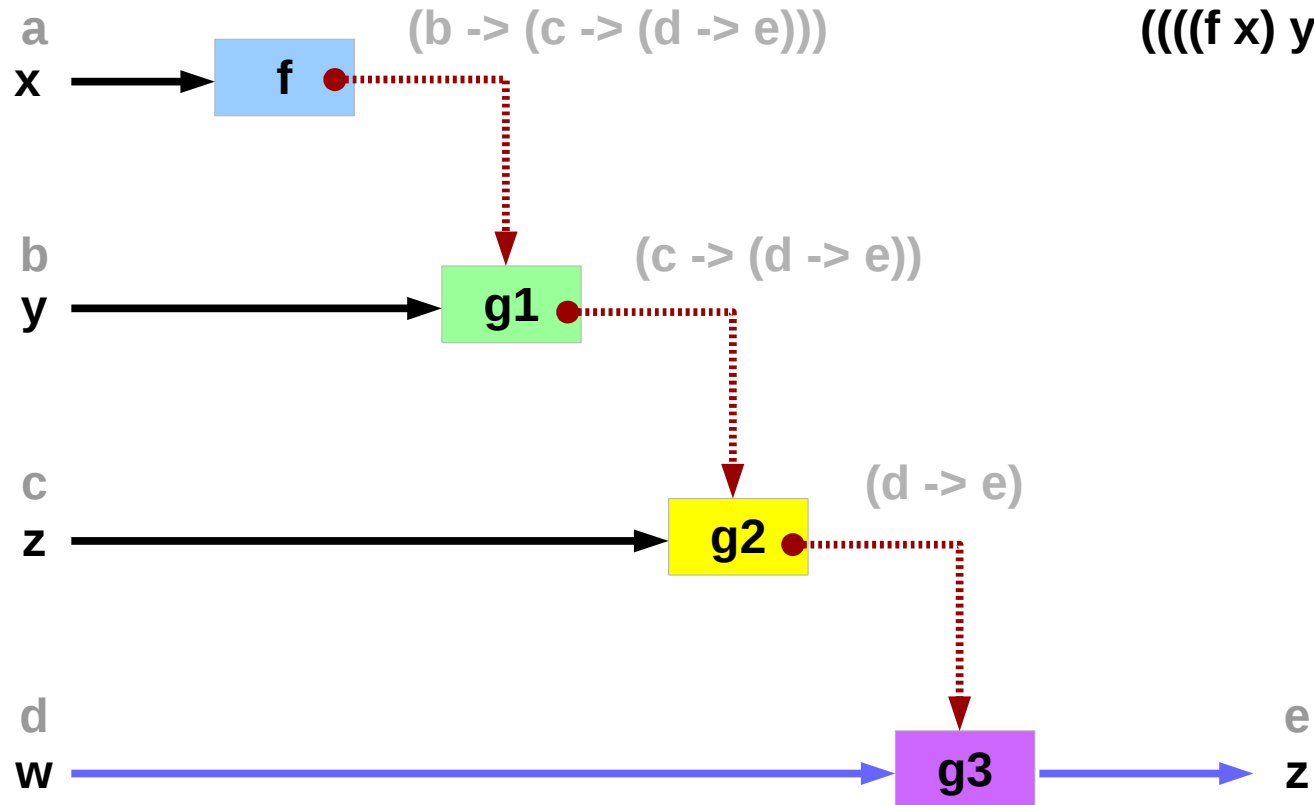
$((((f\ x)\ y)\ z)\ w)$

<http://learnyouahaskell.com/functors-applicative-functors-and-monoids>

Currying Examples

$f :: a \rightarrow b \rightarrow c \rightarrow d \rightarrow e$

$f :: a \rightarrow (b \rightarrow (c \rightarrow (d \rightarrow e)))$



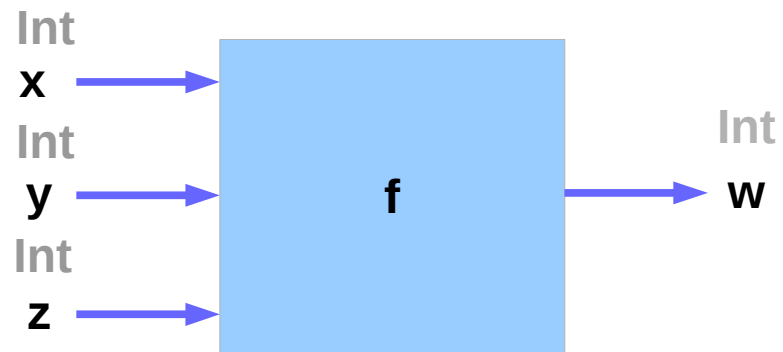
$(((((f\ x)\ y)\ z)\ w))$

<http://learnyouahaskell.com/functors-applicative-functors-and-monoids>

Currying Examples

`mult :: Int -> Int -> Int -> Int`
`((mult x) y) z`

`f :: a -> (b -> (c -> d))`
`((f x) y) z`



<http://learnyouahaskell.com/functors-applicative-functors-and-monoids>

Partial Applications

`mult :: Int -> Int -> Int -> Int`

`mult x y z`

`mult a1 y z = g1 y z`

`mult a1 a2 z = g2 z`

`mult a1 a2 a3 constants`

`f :: Int -> (Int -> (Int -> Int))`

`f :: Int -> (Int -> (Int -> Int))`

`f x y z`

`f x :: Int -> (Int -> Int)`

`g1 :: Int -> (Int -> Int)`

`g1 y z`

`f x y :: Int -> Int`

`g2 :: Int -> Int`

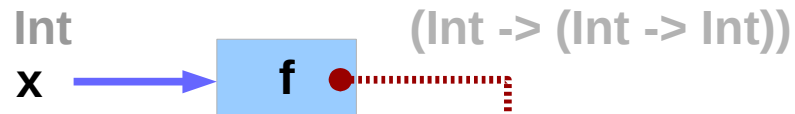
`g2 z`

<http://learnyouahaskell.com/functors-applicative-functors-and-monoids>

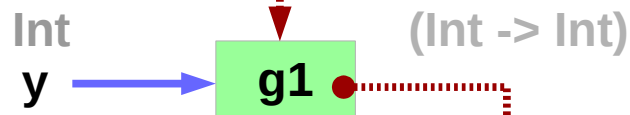
Returning Functions

`mult :: Int -> Int -> Int -> Int`

`mult x y z`



`mult a1 y z`



`mult a1 a2 z`

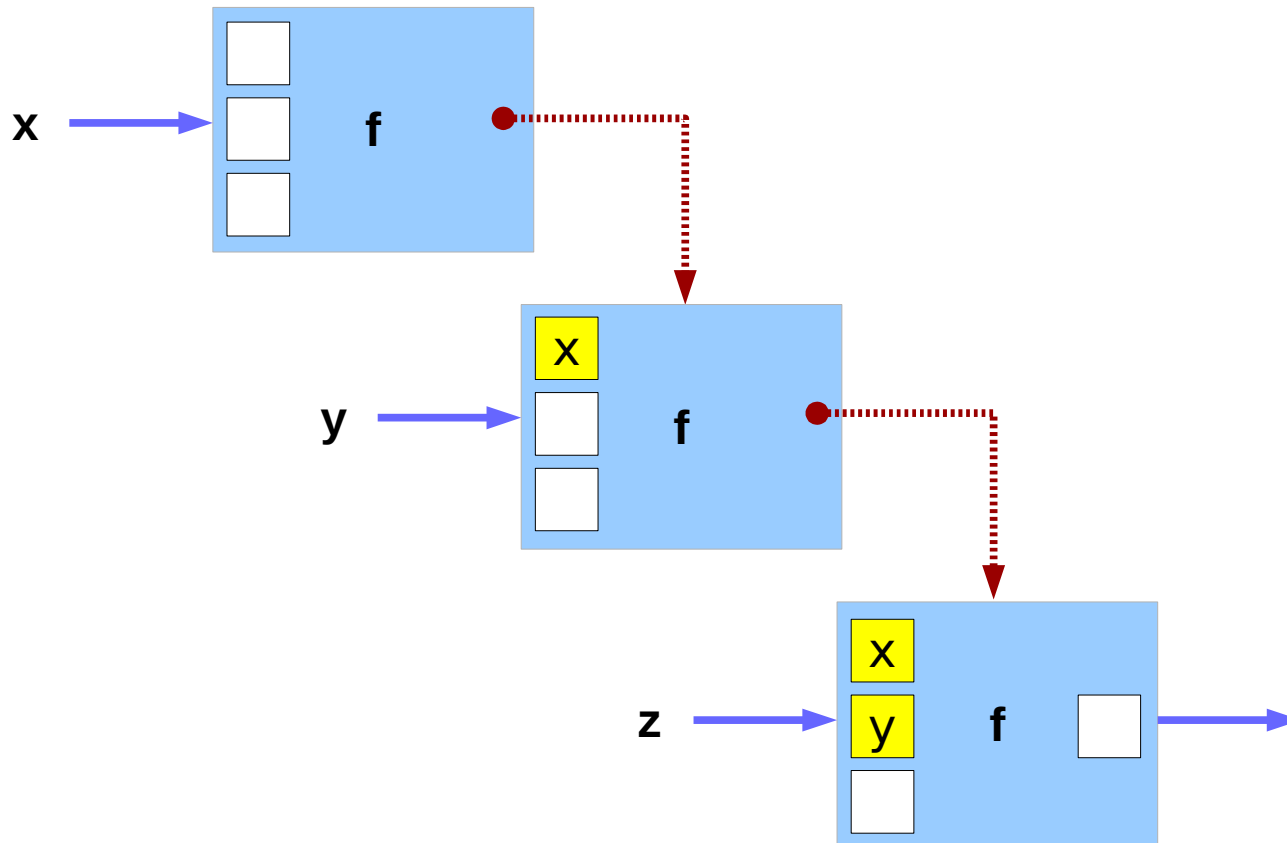


`mult a1 a2 a3`

<http://learnyouahaskell.com/functors-applicative-functors-and-monoids>

Currying Examples

`mult :: Int -> Int -> Int -> Int`



`mult x y z`

`mult a1 y z`

`mult a1 a2 z`

`mult a1 a2 a3`

<http://learnyouahaskell.com/functors-applicative-functors-and-monoids>

Anonymous Function

```
\x -> x + 1
```

```
(\x -> x + 1) 4
```

```
5 :: Integer
```

```
(\x y -> x + y) 3 5
```

```
8 :: Integer
```

```
addOne = \x -> x + 1
```

Lambda Expression

https://wiki.haskell.org/Anonymous_function

let ... in ...

```
cylinder :: (RealFloat a) => a -> a -> a
```

```
cylinder r h =
```

```
  let sideArea = 2 * pi * r * h
```

```
      topArea = pi * r ^2
```

```
  in sideArea + 2 * topArea
```

The form is **let** <bindings> **in** <expression>.

The names that you define in the **let** part are **accessible** to the expression after the **in** part.

Notice that the names are also aligned in a single column.

For now it just seems that **let** puts the bindings first and the expression that uses them later **whereas** where is the other way around.

<http://learnyouahaskell.com/syntax-in-functions>

\$ Function Application

$(\$)$:: $(a \rightarrow b) \rightarrow a \rightarrow b$

$f \$ x = f x$

$(a \rightarrow b)$: left function

a : right value

b : result

Function application with a space

- high precedence

- left-associative

$f x$

$f a b c = ((f a) b) c$

Function application with \$

- the lowest precedence

- right associative

$f \$ x$

$f \$ a \$ b \$ c = f (a (b c))$

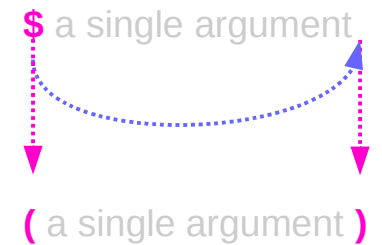
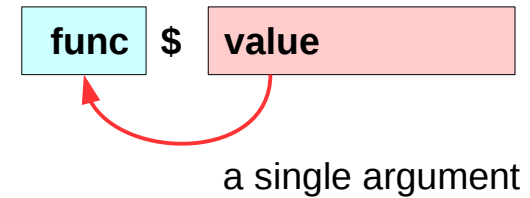
<http://learnyouahaskell.com/higher-order-functions>

\$ a single argument

\$ a convenience function that eliminates many parentheses.

When a \$ is encountered, the expression on its right is applied as the parameter to the function on its left.

writing an opening parentheses (and then writing a closing one) on the far right side of the expression.



far right side

<http://learnyouahaskell.com/higher-order-functions>

\$ Function Application Examples

```
sum (map sqrt [1..130])
```

due to a low precedence

```
sum $ map sqrt [1..130]
```

```
sqrt 3 + 4 + 9
```

```
((sqrt 3) + (4 + 9))
```

```
sqrt (3 + 4 + 9)
```

```
sqrt $ 3 + 4 + 9
```

<http://learnyouahaskell.com/higher-order-functions>

\$ Right Associative Examples

because \$ is right-associative

f (g (z x))

f \$ g \$ z x

sum (filter (> 10) (map (*2) [2..10]))

sum \$ filter (> 10) \$ map (*2) [2..10]

<http://learnyouahaskell.com/higher-order-functions>

\$ Map Function Application Examples

But apart from getting rid of parentheses,
\$ means that function application
can be treated just like another function.

map function application over a list of functions.

```
map ($ 3) [(4+), (10*), (^2), sqrt]
```

```
[(4+ $ 3), (10* $ 3), (^2 $ 3), sqrt $ 3]
```

```
[7.0, 30.0, 9.0, 1.7320508075688772]
```

<http://learnyouahaskell.com/higher-order-functions>

const function

```
const x _ = x
```

```
Prelude> const 3 333
```

```
3
```

```
Prelude> const 3 99999
```

```
3
```

useful for passing to higher-order functions
when you don't need all their flexibility.

For example, the monadic sequence operator `>>`
can be defined in terms of the monadic bind operator as

```
 $x \gg y = x \gg= \mathbf{const} y$ 
```

```
 $(\gg) = (. \mathbf{const}) . (\gg=)$ 
```

<https://stackoverflow.com/questions/7402528/whats-the-point-of-const-in-the-haskell-prelude>

read function

```
Prelude> let x = read "True"
```

```
Prelude> :t x
```

```
x :: Read a => a
```

x doesn't have a concrete type.

x is sort of an expression

that can provide a value of a concrete type,
when we ask for it.

ask x to be an **Int** or a **Bool** or anything

```
Prelude> x :: Bool
```

```
True
```

```
Input: read "12"::Int
```

```
Output: 12
```

```
Input: read "12"::Double
```

```
Output: 12.0
```

```
Input: read "1.22"::Double
```

```
Output: 1.22
```

<https://stackoverflow.com/questions/7402528/whats-the-point-of-const-in-the-haskell-prelude>
http://zvon.org/other/haskell/Outputprelude/read_f.html

replicate, take, repeat, cycle

replicate `Int -> a -> [a]`

creates a list of **length** given by the first argument
and the items having **value** of the second argument

take `Int -> [a] -> [a]`

creates a list, the first argument determines,
how many **items** should be taken from the list passed
as the second argument

repeat `a -> [a]`

it creates an **infinite** list where all items are the first argument

cycle `[a] -> [a]`

it creates a **circular list** from a finite one

http://zvon.org/other/haskell/Outputprelude/cycle_f.html

replicate, take, repeat, cycle examples

Input: **replicate** 3 5

Output: [5,5,5]

Input: **replicate** 4 "aa"

Output: ["aa","aa","aa","aa"]

Input: **replicate** 5 'a'

Output: "aaaaa"

Input: **take** 5 [1,2,3,4,5,6,7]

Output: [1,2,3,4,5]

Input: **take** 5 [1,2]

Output: [1,2]

Input: **take** 0 [1,2,3,4,5,6,7]

Output: []

Input: **take** 5 (**repeat** 3)

Output: [3,3,3,3,3]

Input: **take** 7 (**iterate** (2*) 1)

Output: [1,2,4,8,16,32,64]

Input: **take** 10 (**cycle** [1,2,3])

Output: [1,2,3,1,2,3,1,2,3,1]

Input: **take** 4 (**repeat** 3)

Output: [3,3,3,3]

Input: **take** 6 (**repeat** 'A')

Output: "AAAAAA"

Input: **take** 5 (**repeat** "A")

Output: ["A","A","A","A","A"]

Input: **take** 10 (**cycle** [1,2,3])

Output: [1,2,3,1,2,3,1,2,3,1]

Input: **take** 10 (**cycle** "ABC")

Output: "ABCABCABCA"

http://zvon.org/other/haskell/Outputprelude/cycle_f.html

References

- [1] <ftp://ftp.geoinfo.tuwien.ac.at/navratil/HaskellTutorial.pdf>
- [2] <https://www.umiacs.umd.edu/~hal/docs/daume02yaht.pdf>