Background – Functions (1C)

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

This document was produced by using OpenOffice.

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 Type Declaration

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

Type Declaration

the declaration of an identifier's type

the identifier name :: the type name ...

type names in Haskell always begin with a capital letter,

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

Functions : First-class Data Types

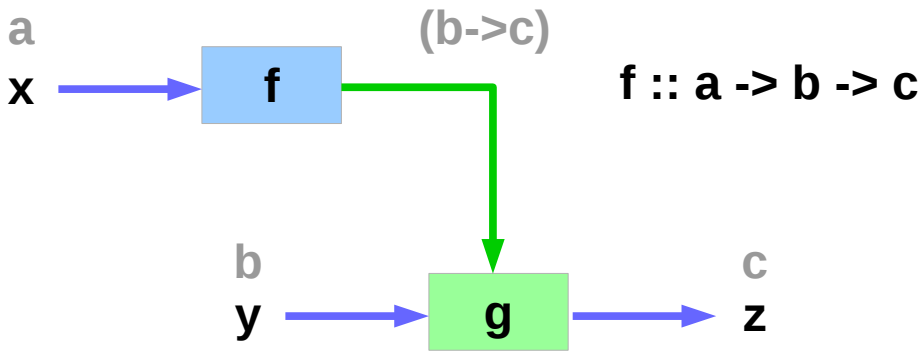
functions are **first-class data types**

Haskell treats functions as regular data,

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

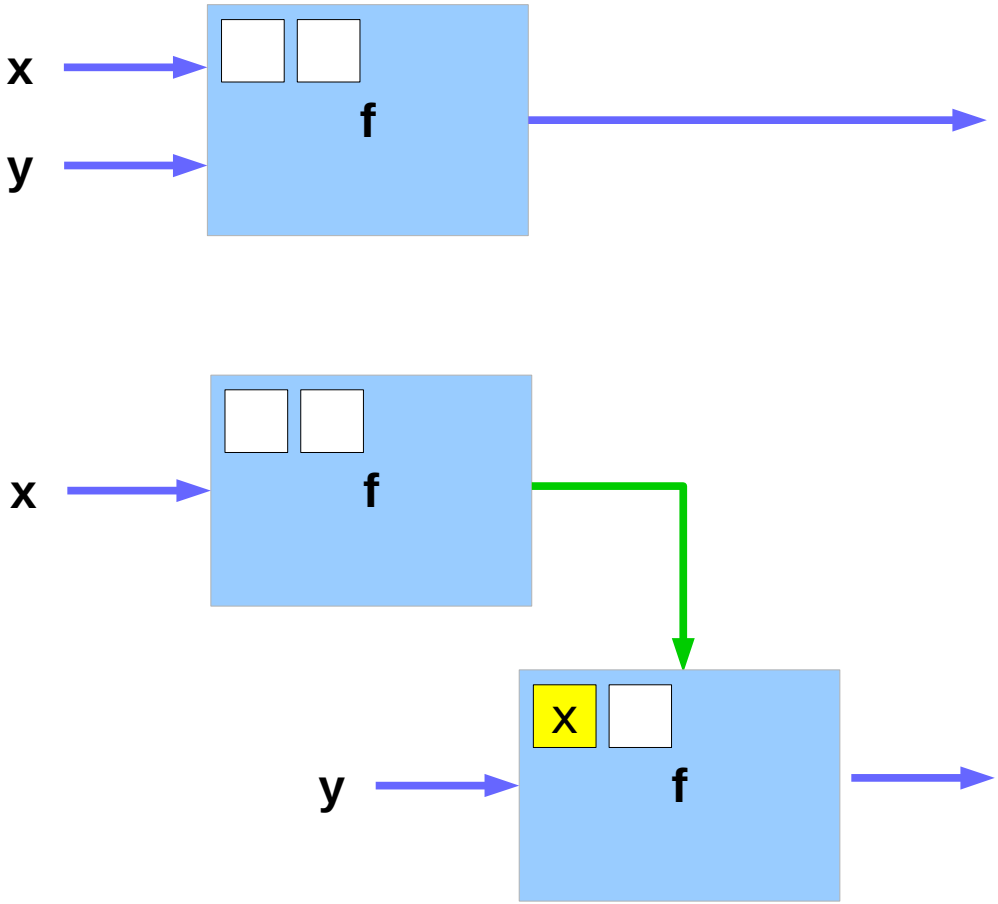
- a function can take other functions as parameters
- a function takes a parameter and produces another function (curried function)

f x y	**f :: a -> b -> c**
(f x) y	**f :: a -> (b -> c)**
f x returns a function of type **b -> c**	
g y	**g :: b -> c**



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

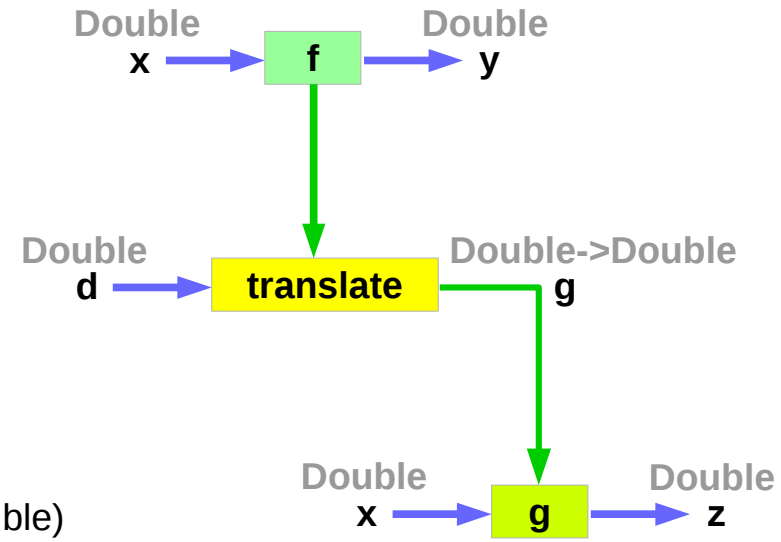
takes a function f and a distance d

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/

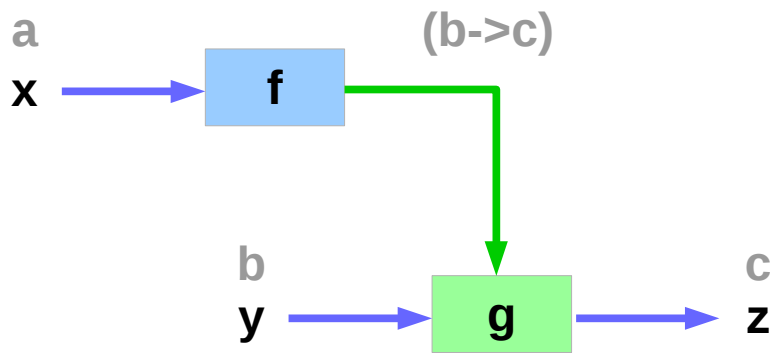
Currying

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 -> b -> c

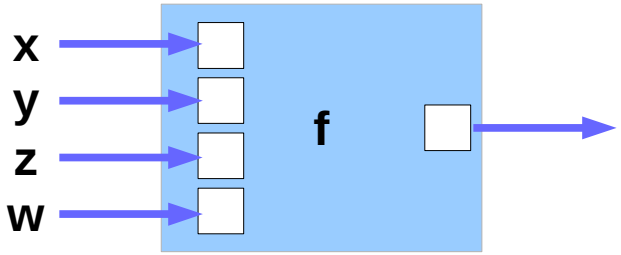
f x y	**f :: a -> b -> c**
(f x) y	**f :: a -> (b -> c)**
g y	**g :: b -> c**

f :: a -> b -> c



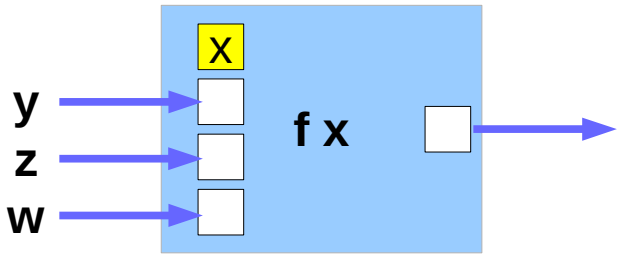
https://wiki.haskell.org/Currying
http://learnyouahaskell.com/functors-applicative-functors-and-monoids

Partially Applied Functions – f, (f x)



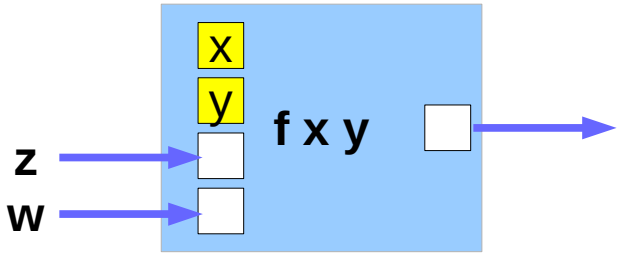

```
f :: a -> b -> c -> d -> e
f  x  y  z  w  = ...
```

```
(f   x)   y   z   w
```



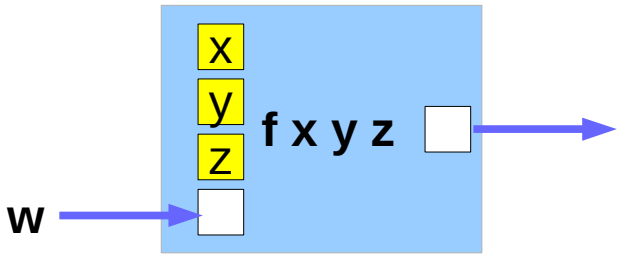

```
g1 :: b -> c -> d -> e
g1  y  z  w  = ...
```

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



(f x y) z w

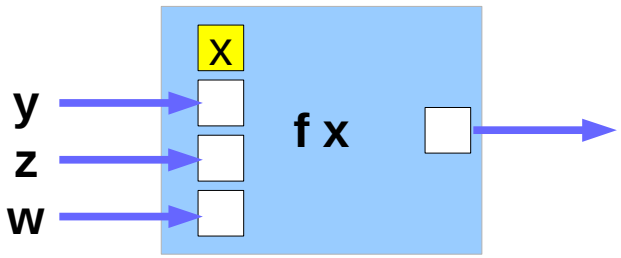
```
g2 :: c -> d -> e
g2  z  w  = ...
```

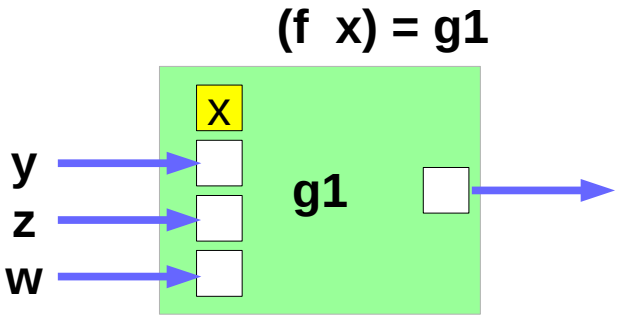


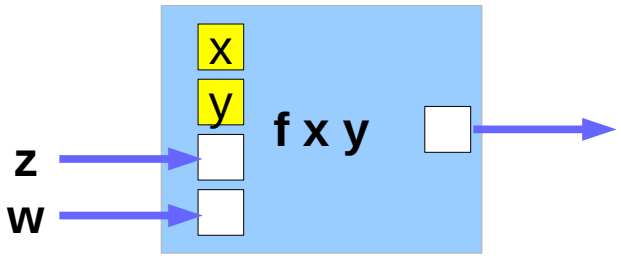
(f x y z) w

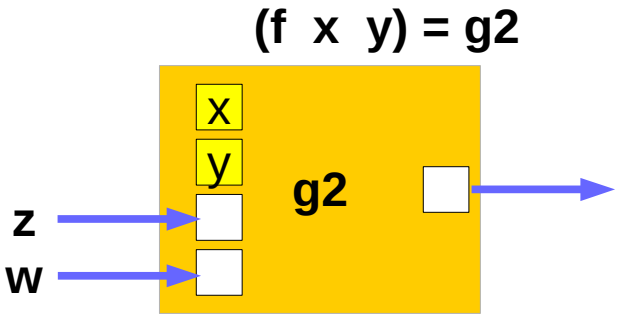
```
g3 :: d -> e
g3  w  = ...
```

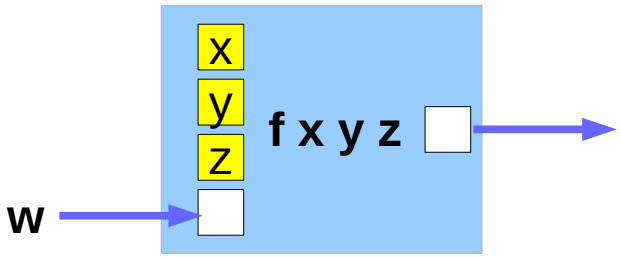
Partially Applied Functions – g1, g2, g3

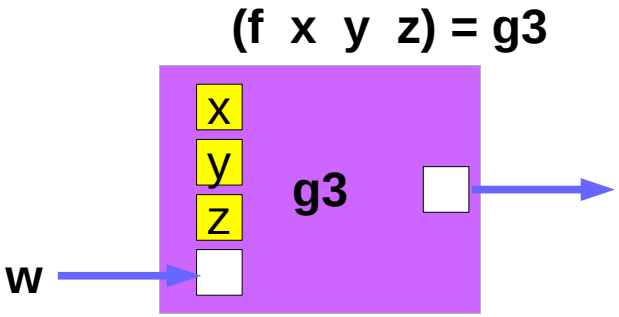




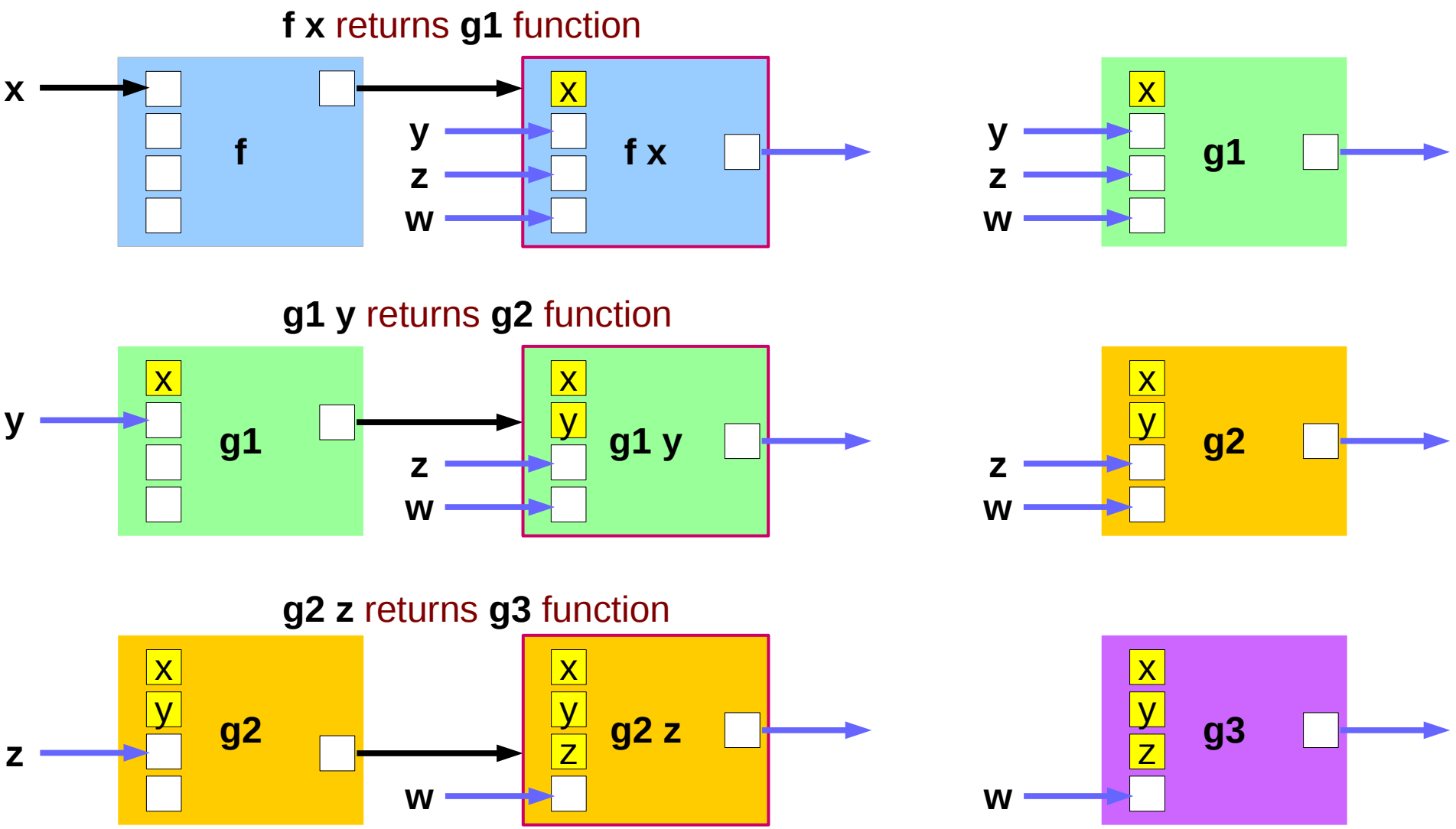








Returning Functions

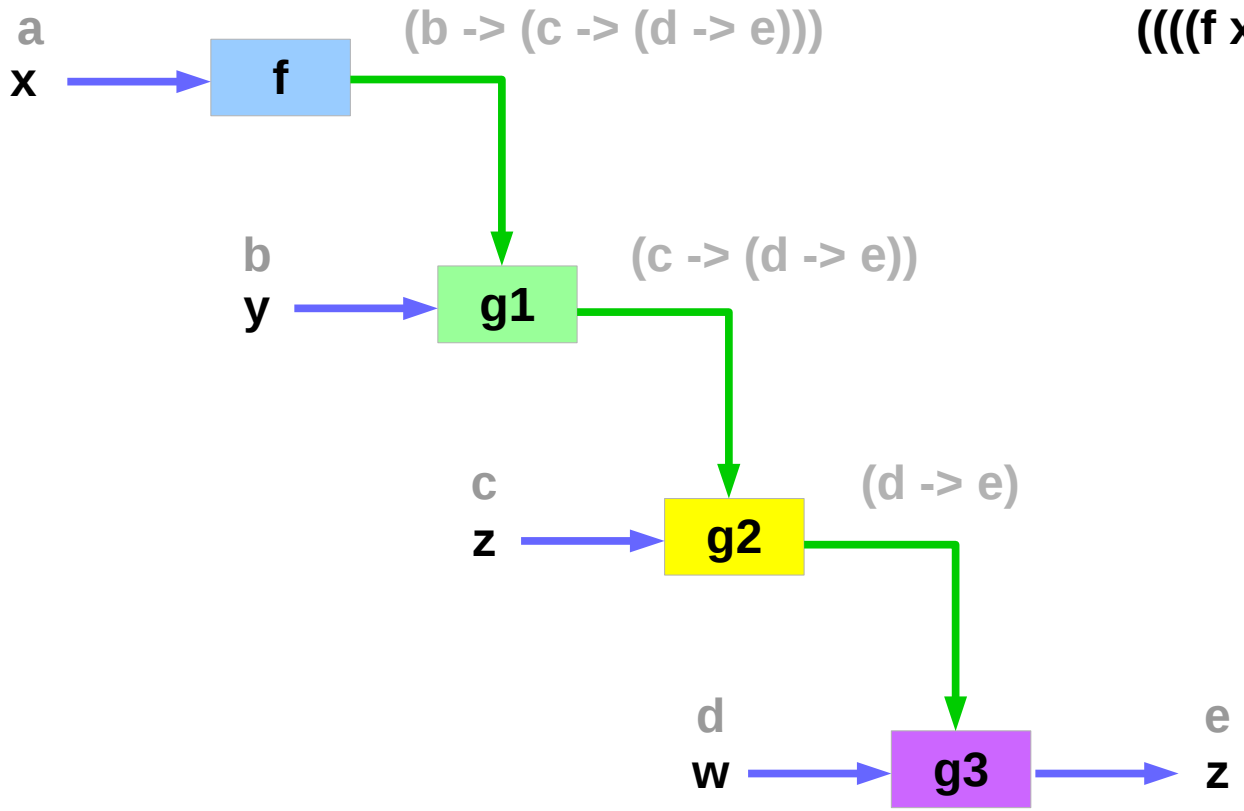


Currying Examples

f :: a -> b -> c -> d -> e

f :: a -> (b -> (c -> (d -> e)))

((((f x) y) z) w)

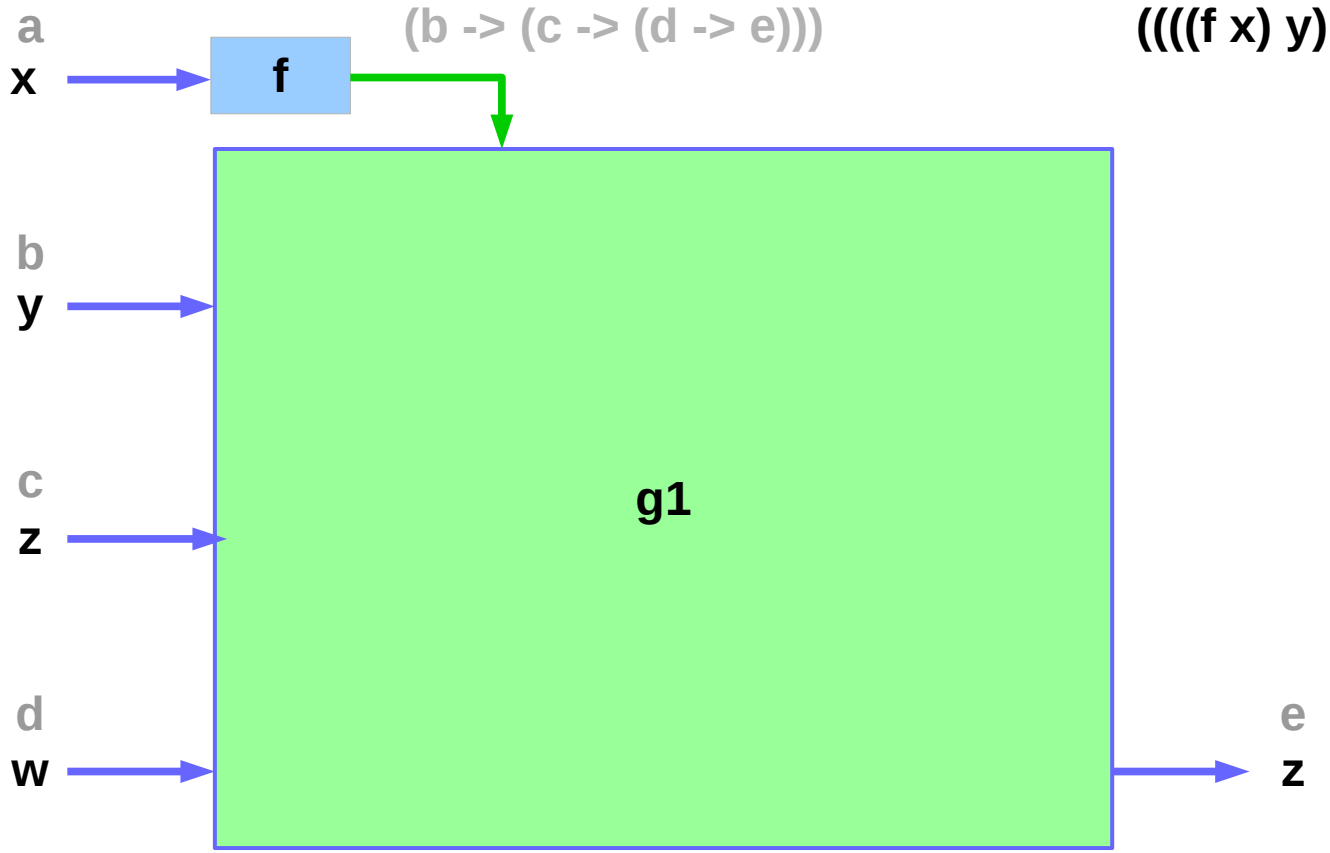


http://learnyouahaskell.com/functors-applicative-functors-and-monoids

Currying Examples

f :: a -> b -> c -> d -> e

f :: a -> (b -> (c -> (d -> e)))



((((f x) y) w) z)

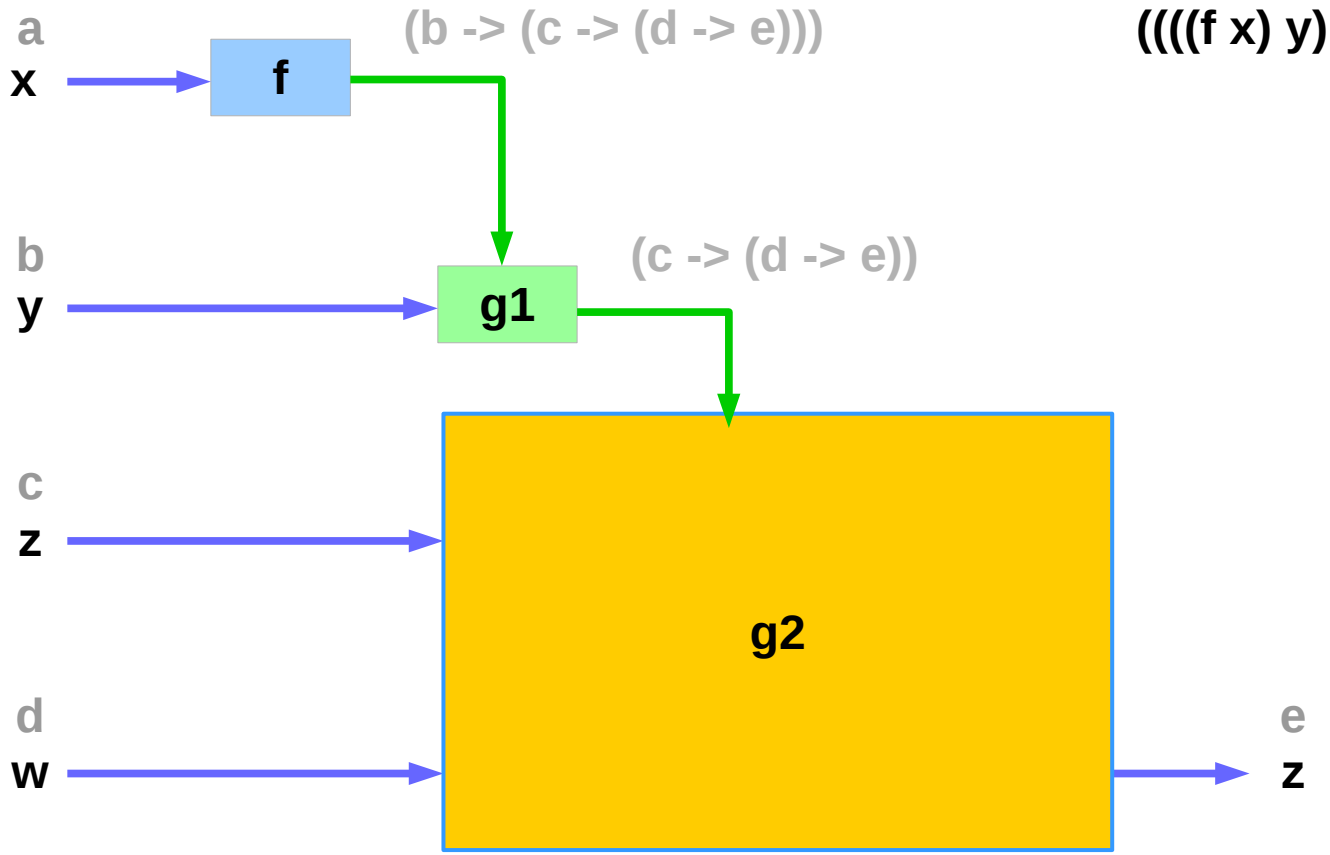
http://learnyouahaskell.com/functors-applicative-functors-and-monoids

Currying Examples

f :: a -> b -> c -> d -> e

f :: a -> (b -> (c -> (d -> e)))

((((f x) y) z) w)



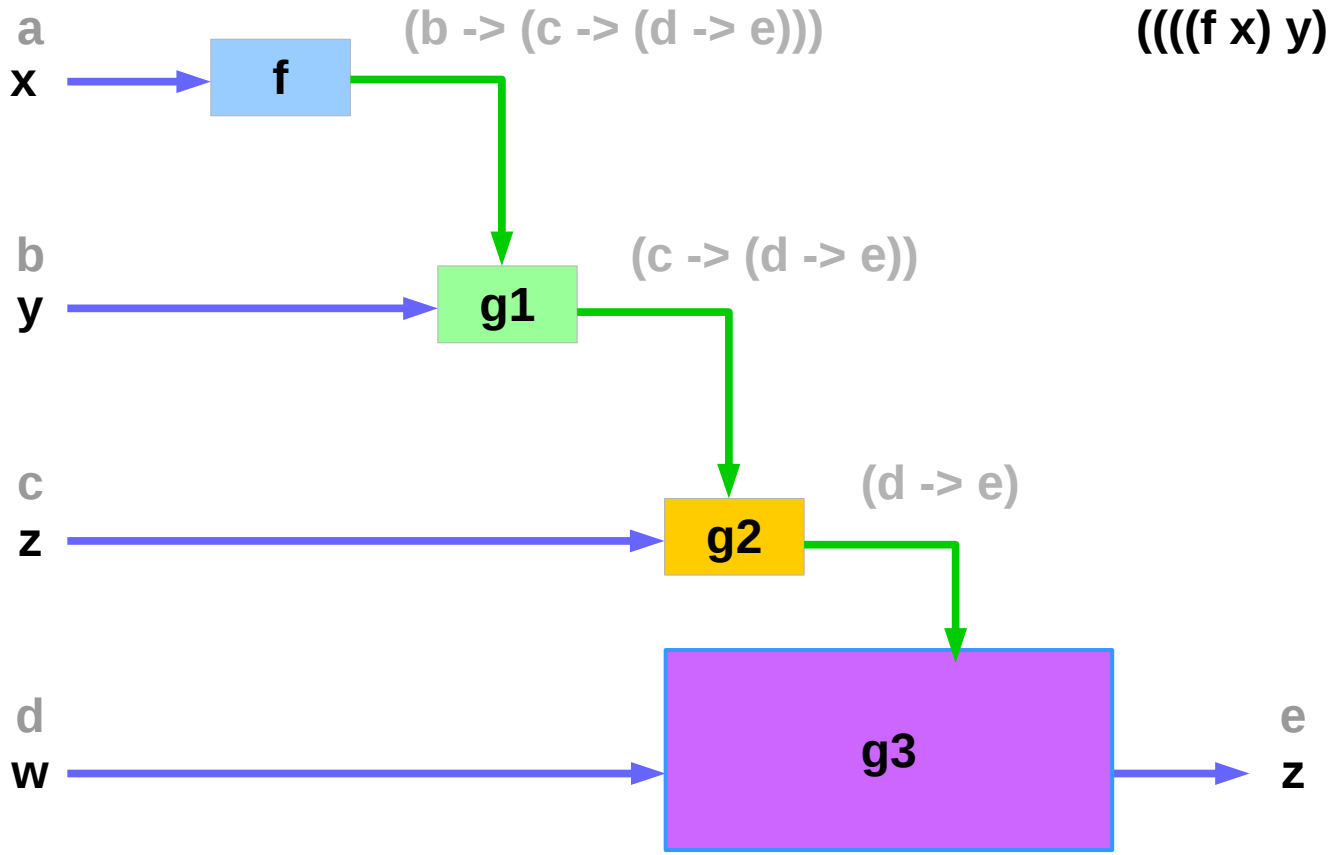
http://learnyouahaskell.com/functors-applicative-functors-and-monoids

Currying Examples

f :: a -> b -> c -> d -> e

f :: a -> (b -> (c -> (d -> e)))

((((f x) y) z) w)

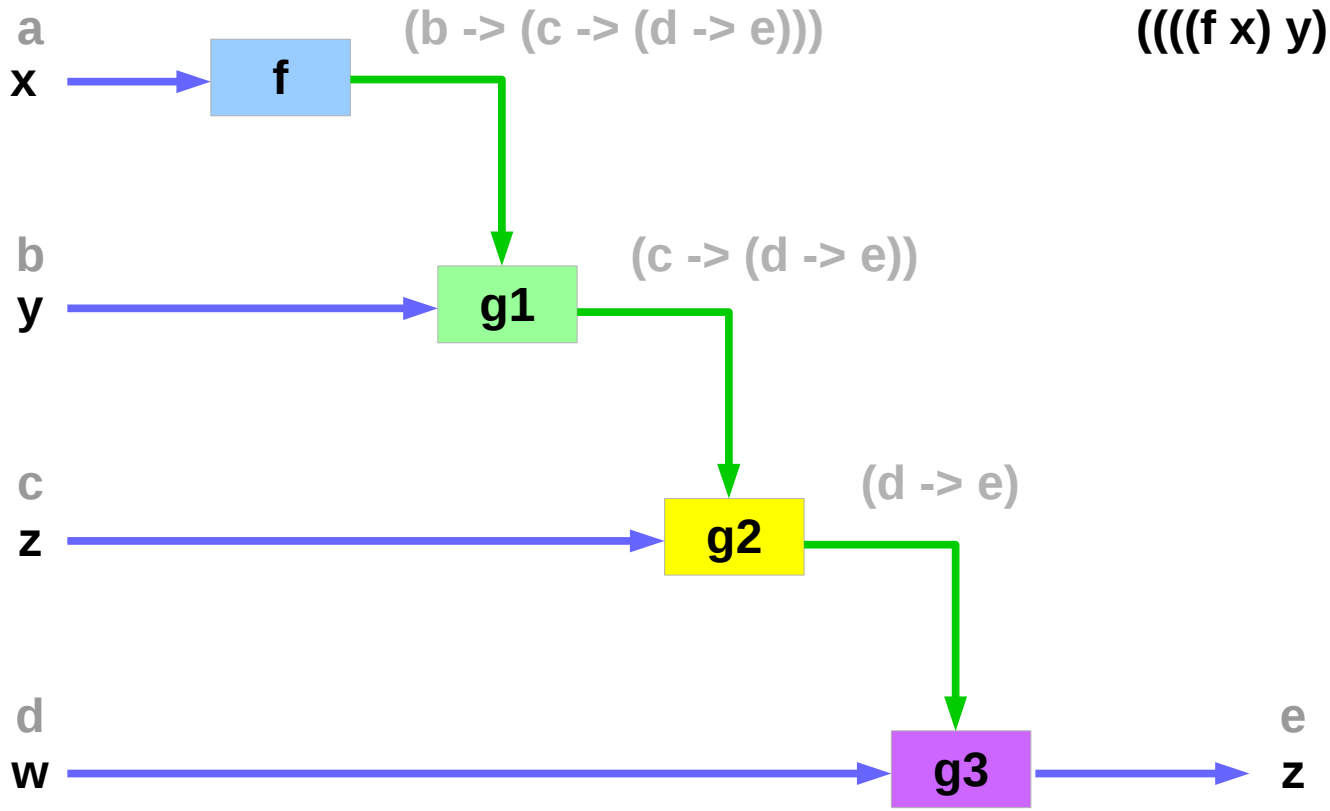


http://learnyouahaskell.com/functors-applicative-functors-and-monoids

Currying Examples

f :: a -> b -> c -> d -> e

f :: a -> (b -> (c -> (d -> e)))



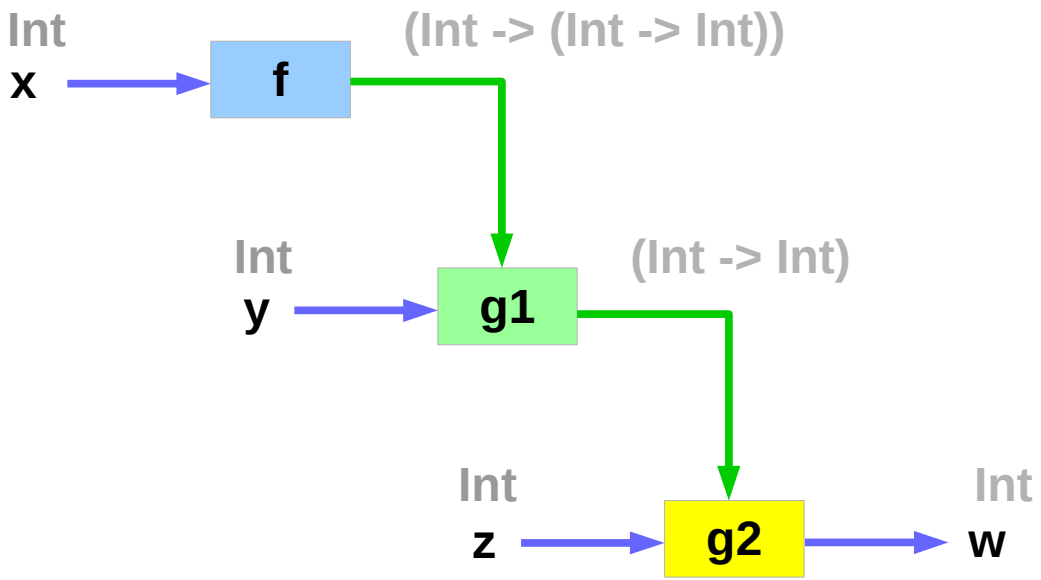
((((f x) y) z) w)

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

Currying Examples

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

f :: a -> (b -> (c -> (d -> e)))

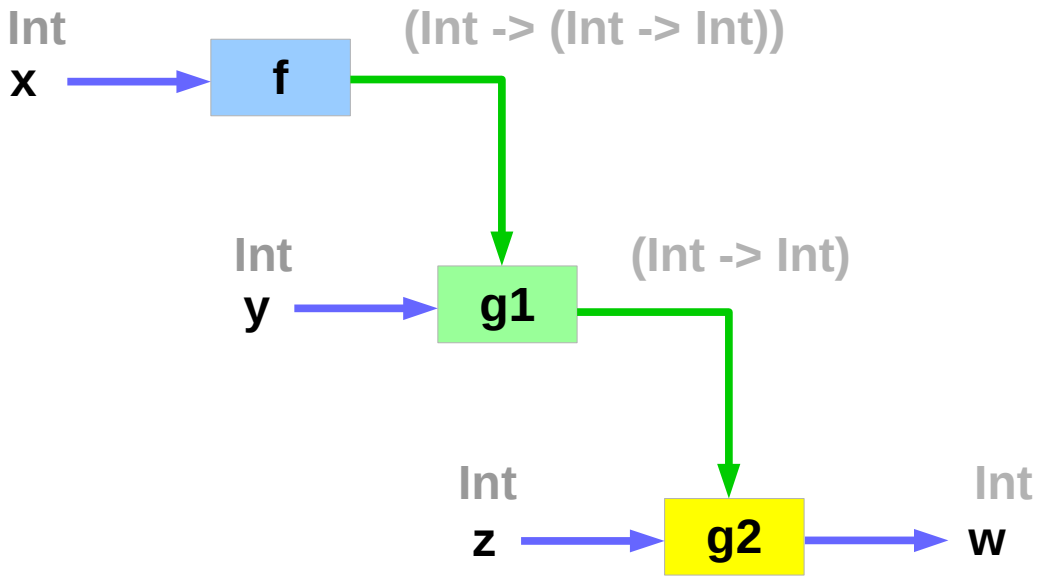


(((mult x) y) z)

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

Currying Examples

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



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

f x :: Int -> (Int -> Int)
g1 :: Int -> (Int -> Int)

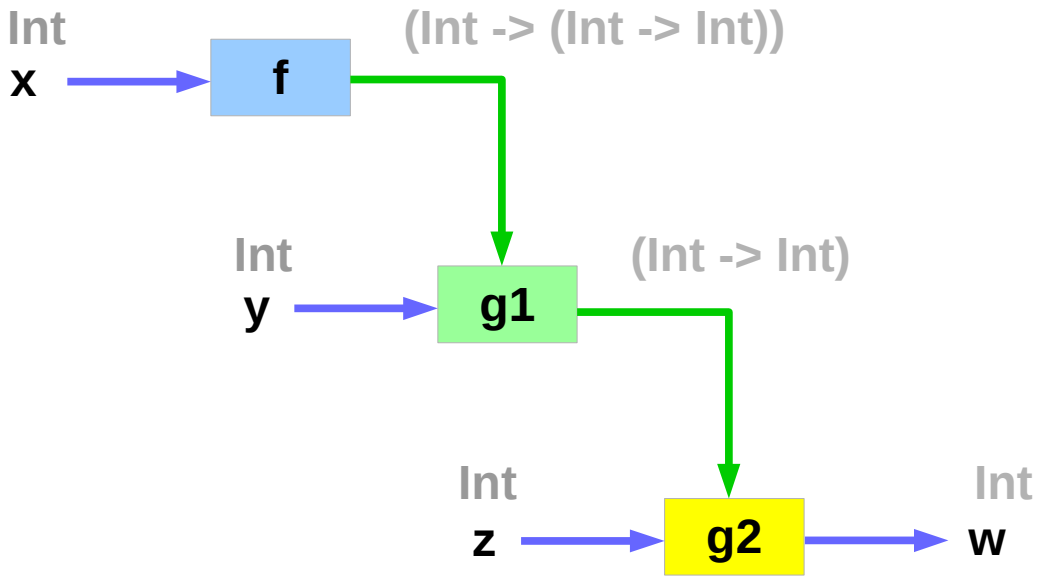
f x y :: Int -> Int
g2 :: Int -> Int

f x y z :: Int

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

Currying Examples

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



mult x y z

mult a_1 y z

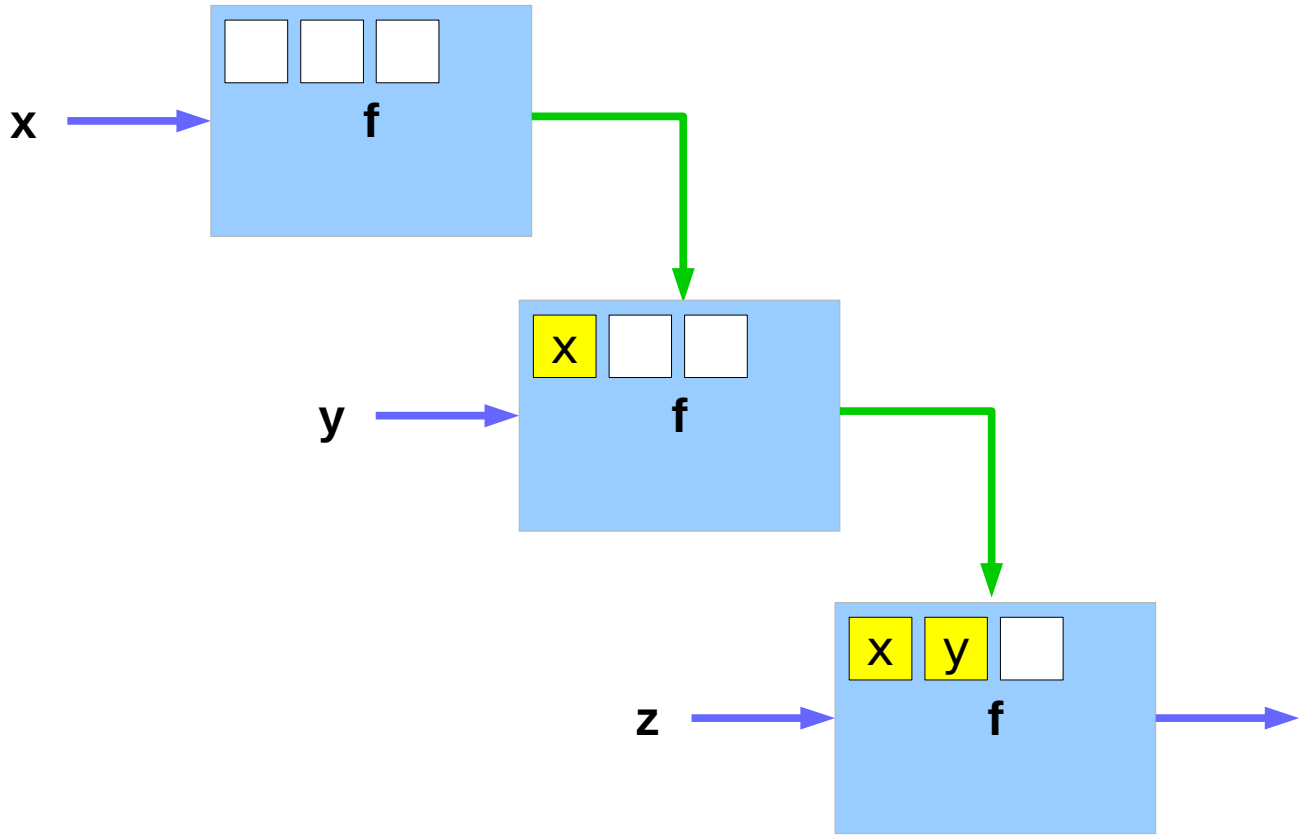
mult a_1 a_2 z

mult a_1 a_2 a_3

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

Currying Examples

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



mult x y z

mult a_1 y z

mult a_1 a_2 z

mult a_1 a_2 a_3

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

Curry & Uncurry

f :: a -> b -> c the curried form of g :: (a, b) -> c

f = curry g
g = uncurry f

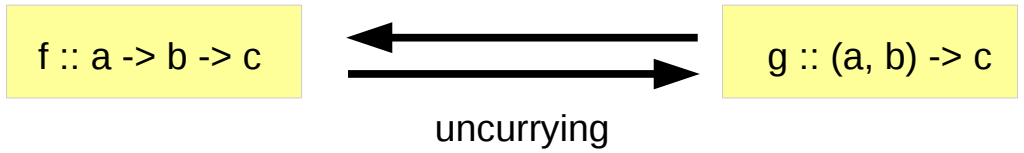
f x y = g (x,y)

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 x y

g (x,y)

https://wiki.haskell.org/Currying

References

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