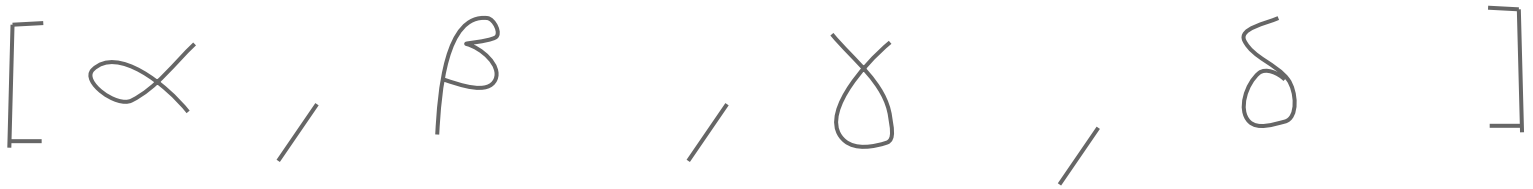
Folding 101

Simplified folding

foldl1

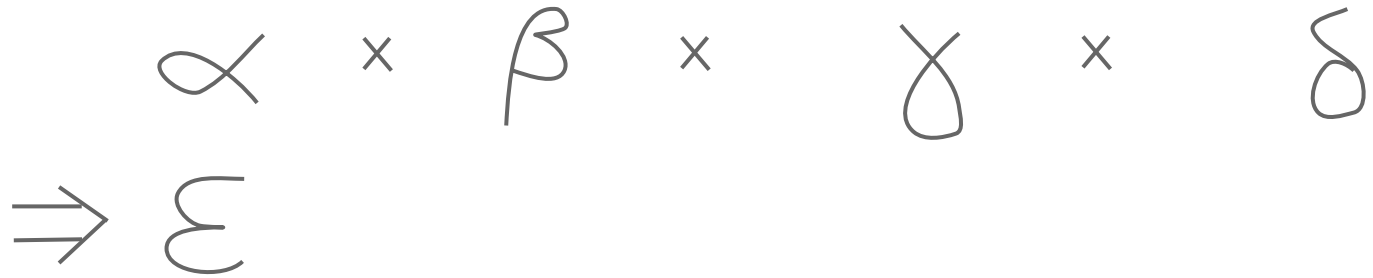
- **Visualise a list of elements**



- **Visualise a list of elements**
- **Remove outer square brackets**

- **Visualise a list of elements**
- **Remove outer square brackets**
- **Replace commas with a single binary operator**

- **Visualise a list of elements**
- **Remove outer square brackets**
- **Replace commas with a single binary operator**
- **Evaluate expression**



Examples

- Summation - foldl1 (+)
- Logical And - foldl1 (&&)
- Bananas - foldl1 mash

Visualise Elements (as when printed)

[1,2,3,4]

[True,True,False]

[Banana,Banana,Banana]

Remove outer square Brackets

1,2,3,4

True,True,False

Banana,Banana,Banana

Replace commas with an operation

1+2+3+4

True && True && False

Banana `mash` Banana `mash` Banana

Evaluate (in this case left to right)

1+2+3+4

=> 10

True && True && False

=> False

Banana `mash` Banana `mash` Banana

=> MashedBananas

Associativity

foldl1 and foldr1

What does this equal?

2 ^ 3 ^ 4

In what order will this evaluate?

2 ^ 3 ^ 4

?

(2 ^ 3) ^ 4 ?

or

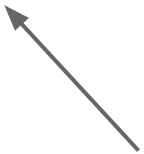
2 ^ (3 ^ 4) ?

Clue

```
Prelude> :i (^)
(^) :: (Num a, Integral b) => a -> b -> a     -- Defined in ‘GHC.Real’
infixr 8 ^
```

Clue

```
Prelude> :i (^)
(^) :: (Num a, Integral b) => a -> b -> a     -- Defined in ‘GHC.Real’
infixr 8 ^
```



r stands for right associative

In what order will this evaluate?

(2 ^ 3) ^ 4 ?

or

2 ^ (3 ^ 4) ?

In what order will this evaluate?

(8) ^ 4 ? = 4096 NO!

or

2 ^ (3 ^ 4) ?

In what order will this evaluate?

(8) ^ 4 ? = 4096 NO!

or

2 ^ (81) =

241785163922925834941235 2 YES!

or according to “numerals”

```
Prelude Text.Numeral.Language.EN Text.Numeral.Grammar.Reified>
gb_cardinal defaultInflection (2^3^4)
```

```
Just "two septillion four hundred and seventeen sextillion eight hundred
and fifty-one quintillion six hundred and thirty-nine quadrillion two
hundred and twenty-nine trillion two hundred and fifty-eight billion three
hundred and forty-nine million four hundred and twelve thousand three
hundred and fifty-two"
```

Common associativities

Left	Right
+ * - /	`function` ^ ** : ++ && ||

Fold left vs fold right

```
foldl1 (^) [2,3,4] == (2 ^ 3) ^ 4

foldr1 (^) [2,3,4] == 2 ^ (3 ^ 4)
```

Fold left vs fold right *Continued...*

foldl1 (-) [2,3,4,5,6] == ((2-3)-4)-5)-6

foldr1 (-) [2,3,4,5,6] == (2-(3-(4-(5-6)))))

Fold left

Smallest most inner(Highest precedence) sub-expression on the left.

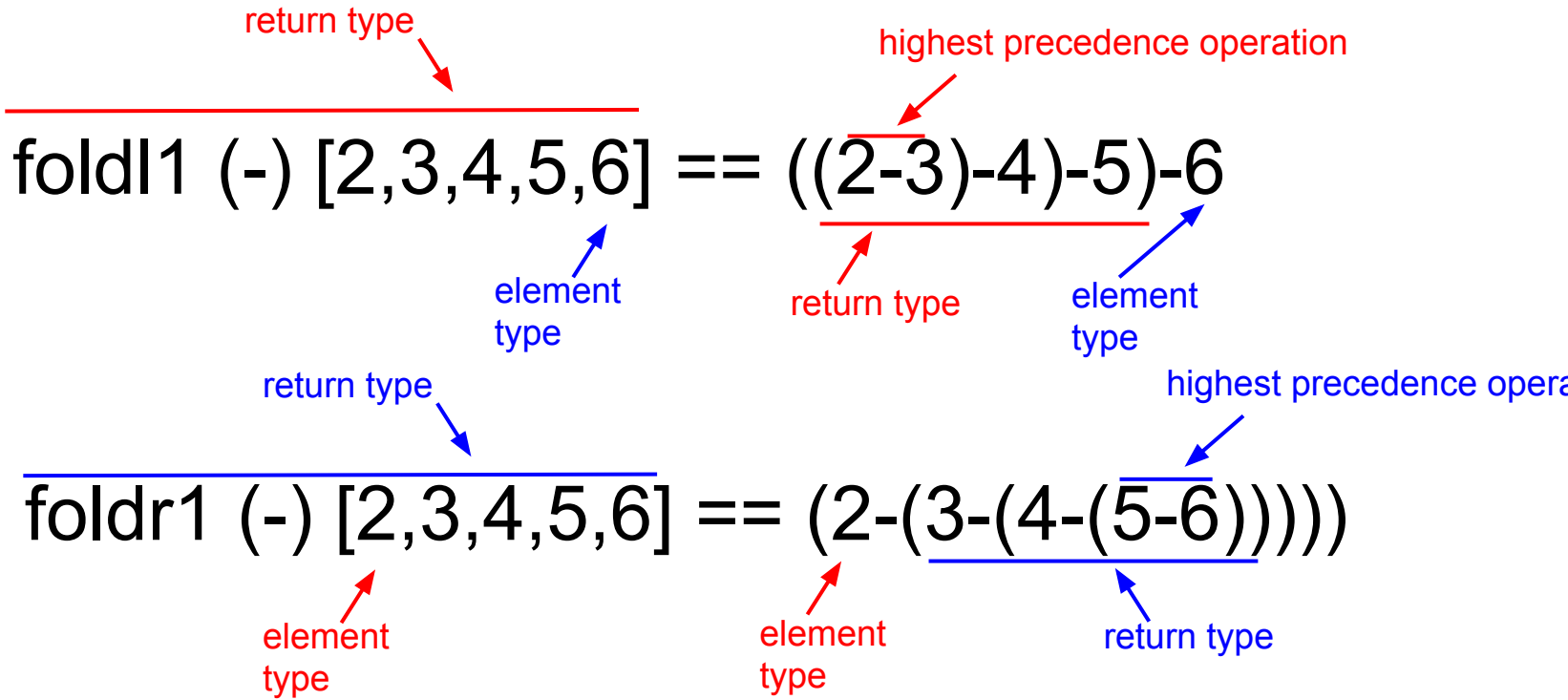
Largest top level sub expression on the left.

Fold right

Smallest most inner(Highest precedence) sub-expression on the right.

Largest top level sub expression on the right.

Fold left vs fold right *Continued...*



Misnomers

foldr f v xs /= foldl f v (reverse xs)

foldr f v xs /= foldl (flip f) v xs

Reasons for choice

Fold right - faster when pattern matching part of result. In the case of infinite list necessary.

Fold left - faster when strictly evaluated.

How to remember

foldx :: ? -> ? -> ? -> *Result*

- Function
- Initial accumulator
- Structure to fold

In general arguments that are subjectively least likely to be parameterised usually go first to make it easier to use partial application.

Heterogeneous Folding

foldl :: (b -> a -> b) -> b -> [a] -> b

return type is left argument

foldr :: (a -> b -> b) -> b -> [a] -> b

return type is right argument