Week 4: Function Expressions and Higher Order Functions

ECS713

Functional Programming

Dr. Paulo Oliva / Prof. Edmund Robinson

Queen Mary University of London

This week (learning objectives)

- understand how to use function application and function composition
- be able to work with functions without “naming” them, using sections, partial application, and anonymous functions
- understand the concept of a higher-order function
- be familiar with using common higher-order functions such as fold, map, zipWith and filter.

This week (core)

- Function Expressions:
  - **Partial application**
  - **Sections**
  - **Anonymous functions** (aka lambda expression)
- Higher Order Functions
  - **What is a Higher Order Function?**
  - **map, filter and zipWith**
  - **foldl and foldr**
  - **Application and composition**

This week (additional)

- Function Expressions:
  - **Partial application**
  - **Sections**
  - **Anonymous functions** (aka lambda expression)
- Higher Order Functions
  - **What is a Higher Order Function?**
  - **map, filter and zipWith**
  - **foldl and foldr**
  - **Application and composition**

Language Uniformity: functions as first-class values

Language Uniformity: declarations and expressions

map-reduce

foldr and recursion

foldl and iterators

Language Uniformity

- Languages should not have special cases:
  - **You should not have constructs that behave in one way on one class of things and differently on others**
- Uniformity across constructs:
  - **If you have two "equivalent" ways of doing something, they should actually work the same.**

Functions as first-class values

- There is no distinction between functions and things of other (basic) types in terms of what you can do with them.

- In particular:
  - **There are expressions that are interpreted as functions (not just names introduced by declarations)**
  - **You can return a function as the result of an operation (another function)**
  - **You can pass a function as a parameter to another operation (a higher order function)**

Function Expressions

ECS713

Functional Programming

Dr. Paulo Oliva / Prof. Edmund Robinson

Queen Mary
University of London

Function Expressions

- An expression that has type **a -> b** is called a **function expression**, because it has a "function" type
- Some useful ways of building function expressions:
  - **Partial application**
  - **Sections**
  - **Anonymous functions** (aka lambda expression)

Partial Application

We can give just some of the arguments of a function in order to define a new function


```
Prelude> let f name age = "Hi " ++ name ++ " you are " ++ show(age)
Prelude> :type f
f :: String -> Int -> String
Prelude> f "John" 40
"Hi John you are 40"
Prelude> let greetJohn = f "John"
Prelude> :type greetJohn
greetJohn :: Int -> String
Prelude> greetJohn 20
"Hi John you are 20"
```


f takes two arguments

When we give it one argument ("John"), we get a function that is waiting for the second argument

We would call this a **partial application**

Infix versus Prefix

Some binary functions are more naturally written in **infix** notation, while most are written in **prefix** notation


```
Prelude> 2 + 3
5
Prelude> div 8 2
4
Prelude> (+) 2 3
5
Prelude> 8 `div` 2
4
```

One can convert from one to the other using ( ) and ` `

Sections

We can also do **partial application** in infix binary functions, these are called **sections**

```
Prelude> let f = (+3)
Prelude> :type f
Int -> Int
Prelude> f 10
13
Prelude> let g = (2^)
Prelude> map g [1..5]
[2,4,8,16,32]
Prelude> let h = (^2)
Prelude> map h [1..5]
[1,4,9,16,25]
```

We call this the "**right section**" of addition +

This is the "**left section**" of the exponentiation function

The **left** and **right** sections could be very different functions

Reading the type

-> types can be read in two different ways

```
Prelude> let f name age = "Hi " ++ name ++ " you are " ++ show(age)
Prelude> :type f
f :: String -> Int -> String
```


We can read this as a single type constructor:

<> -> <> -> <>

applied to three types

But technically it is a compound construction, built from two instances of

<> -> <>

Reading the type

-> types can be read in two different ways: but technically all functions in Haskell have only a single argument

```
Prelude> let f name age = "Hi " ++ name ++ " you are " ++ show(age)
Prelude> :type f
f :: String -> Int -> String
```


`String -> Int -> String`

brackets as

`String -> (Int -> String)`

This fits with

`f “John” 40`

bracketing as

`(f “John”) 40`

The first argument of `f` is a `String`

Partially applying `f` to this `String` produces a function `Int -> String`

Using sections

We can use sections in some complicated ways:


```
*Main> filter ((==2).(`mod` 3)) [1..20]
[2,5,8,11,14,17,20]
*Main> filter (`elem` "aeiouAEIOU") "The cat sat on the mat"
"eaaoea"
*Main> filter (not.(`elem` "aeiouAEIOU")) "The cat sat on the
mat"
"Th ct st n th mt"
*Main>
```

Anonymous functions

We can use the "**lambda notation**" to define a function without giving it a name

```
Prelude> let f n = n + 1
Prelude> f 10
11
Prelude> (\n -> n + 1) 10
11
Prelude> map (\n -> 2 * n) [1..5]
[2,4,6,8,10]
Prelude> map (2*) [1..5]
[2,4,6,8,10]
```

This is called a "**lambda expression**" because mathematically we would write this as $\lambda n \rightarrow n + 1$

Sometimes a "section" is sufficient, these two function expressions are equivalent

Using lambda expressions

We can use lambda expressions to make quick anonymous functions:

```
*Main> filter ((==2).(`mod` 3)) [1..20]
[2,5,8,11,14,17,20]
*Main> filter (\x -> x `mod` 3 == 2) [1..20]
[2,5,8,11,14,17,20]
*Main> filter (`elem` "aeiouAEIOU") "The cat sat on the mat"
"eaaoea"
*Main> filter (not.(`elem` "aeiouAEIOU")) "The cat sat on the mat"
"Th ct st n th mt”
*Main> filter (\x -> not (elem x “aeiouAEIOU")) "The cat sat on the mat"
"Th ct st n th mt
*Main>
```

Functions as first-class values

- There is no distinction between functions and things of other (basic) types in terms of what you can do with them.

- In particular:
  - **There are expressions that are interpreted as functions (not just names introduced by declarations)**
  - **You can return a function as the result of an operation (another function)**
  - **You can pass a function as a parameter to another operation (a higher order function)**

Language uniformity

Using lambda expressions, any function declaration is equivalent to a simple value declaration.

```
-- standard pattern-matching declaration - pattern = expr
f :: String -> Int -> String
f name age = "Hi "++name++" you are "++show(age)

-- equivalent simple declarations - name = expr
g :: String -> Int -> String
g = \ name age -> "Hi "++name++" you are "++show(age)

h :: String -> Int -> String
h = \ name -> (\age -> "Hi "++name++" you are "++show(age))
```

Language uniformity

Using lambda expressions, any function declaration is equivalent to a simple value declaration.

```
– standard pattern-matching declaration – pattern = expr
sum' :: [Int] -> Int
sum' [] = 0
sum' (x:xs) = x + sum' xs

– equivalent simple declaration - name = expr
sum'' :: [Int] -> Int
sum'' = \ xs -> case xs of
    [] -> 0
    (y:ys) -> y + sum'' ys
```

Language Uniformity

- Languages should not have special cases:
  - ✦ **You should not have constructs that behave in one way on one class of things and differently on others**
- Uniformity across constructs:
  - ✦ **If you have two "equivalent" ways of doing something, they should actually work the same.**

Higher Order Functions

ECS713

Functional Programming

Dr. Paulo Oliva / Prof. Edmund Robinson

Queen Mary University of London

Higher Order Functions

`Functions that take other functions as arguments!`


```
Prelude> :type map
(a -> b) -> [a] -> [b]
Prelude> :type filter
(a -> Bool) -> [a] -> [a]
Prelude> :type foldl
(b -> a -> b) -> b -> [a] -> b
Prelude> :type foldr
(a -> b -> b) -> b -> [a] -> b
```

map, **filter** and **fold** are extremely useful higher order functions!

Map

Function to be “mapped”

List of inputs

List of results

```
Prelude> :type map
map :: (a -> b) -> [a] -> [b]
Prelude> map (2*) [1..5]
[2,4,6,8,10]
Prelude> map (\x -> x + 17) [1..5]
[18,19,20,21,22]
Prelude> map not [True, False, True]
[False, True, False]
Prelude> :module Data.Char
Prelude Data.Char> map toUpper "coffee"
"COFFEE"
```

Filter

A “test”

Input list

Sub-list, with elements that “pass the test”

```
Prelude> :type filter
filter :: (a -> Bool) -> [a] -> [a]
Prelude> filter odd [1..10]
[1,3,5,7,9]
Prelude> filter (<3.1415) [4.3,1.1,0.2,5.7]
[1.1,0.2]
Prelude> :module Data.Char
Prelude Data.Char> filter isDigit "17 Mandela Blvd"
"17"
Prelude Data.Char> filter isAlpha "17 Mandela Blvd"
"MandelaBlvd"
```

map and **filter** are built in functions
but actually they have simple recursive definitions


```
map f [] = []
map f (x:xs) = (f x) : (map f xs)

filter p [] = []
filter p (x:xs) = if p x then x : rest else rest
    where rest = filter p xs
```

map and **filter** have recursive definitions, by recursion on the input list xs

zip and zipWith

```
Prelude> :type zip
zip :: [a] -> [b] -> [(a, b)]
Prelude> zip ['a'..'d'] [1..5]
[('a',1),('b',2),('c',3),('d',4)]
Prelude> :type zipWith
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
Prelude> zipWith (*) [1..4] [2..6]
[2,6,12,20]
```

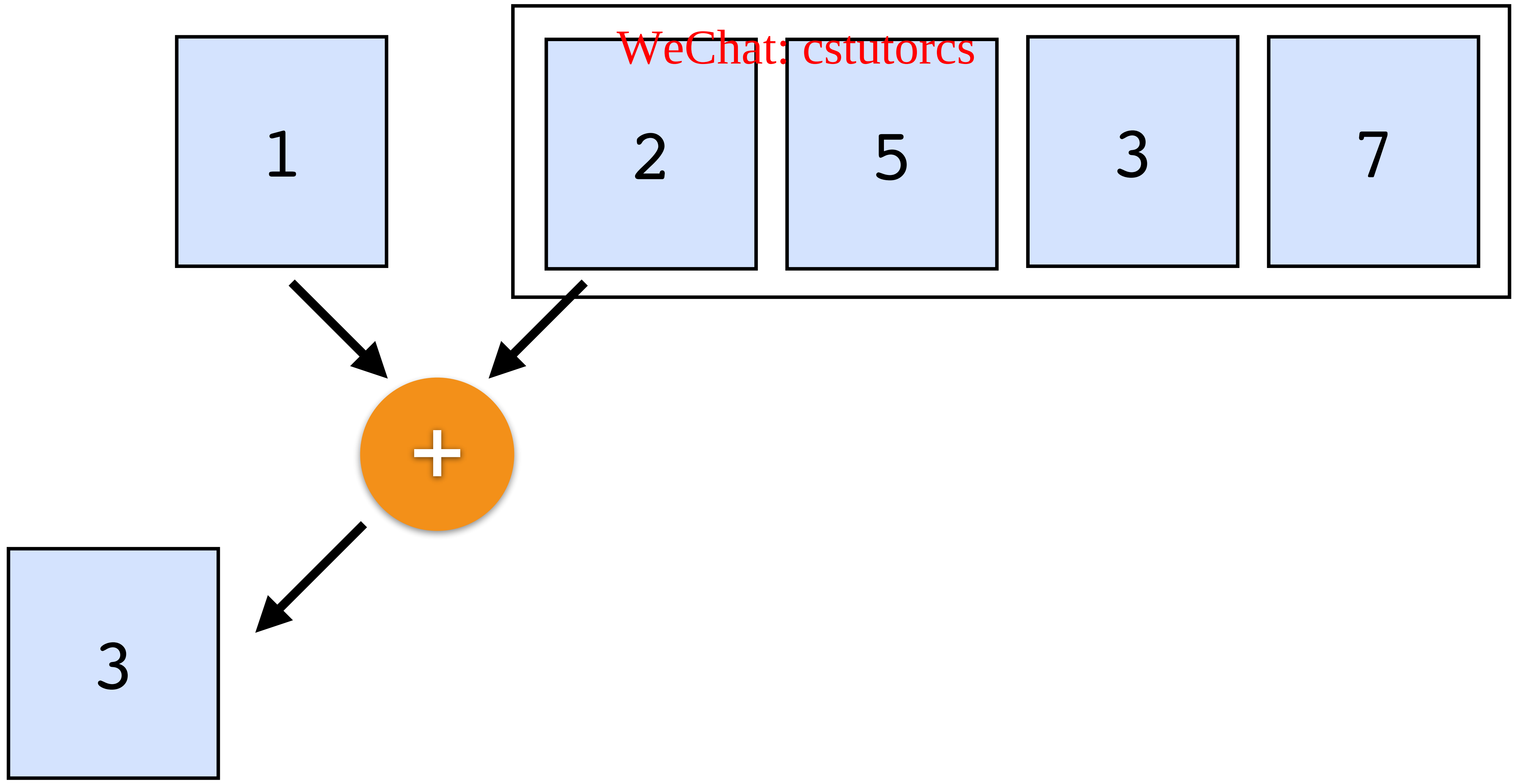
Fold Left

```
Prelude> foldl (+) 1 [2,5,3,7]
18
```

A binary function

Starting value

List to be “folded”



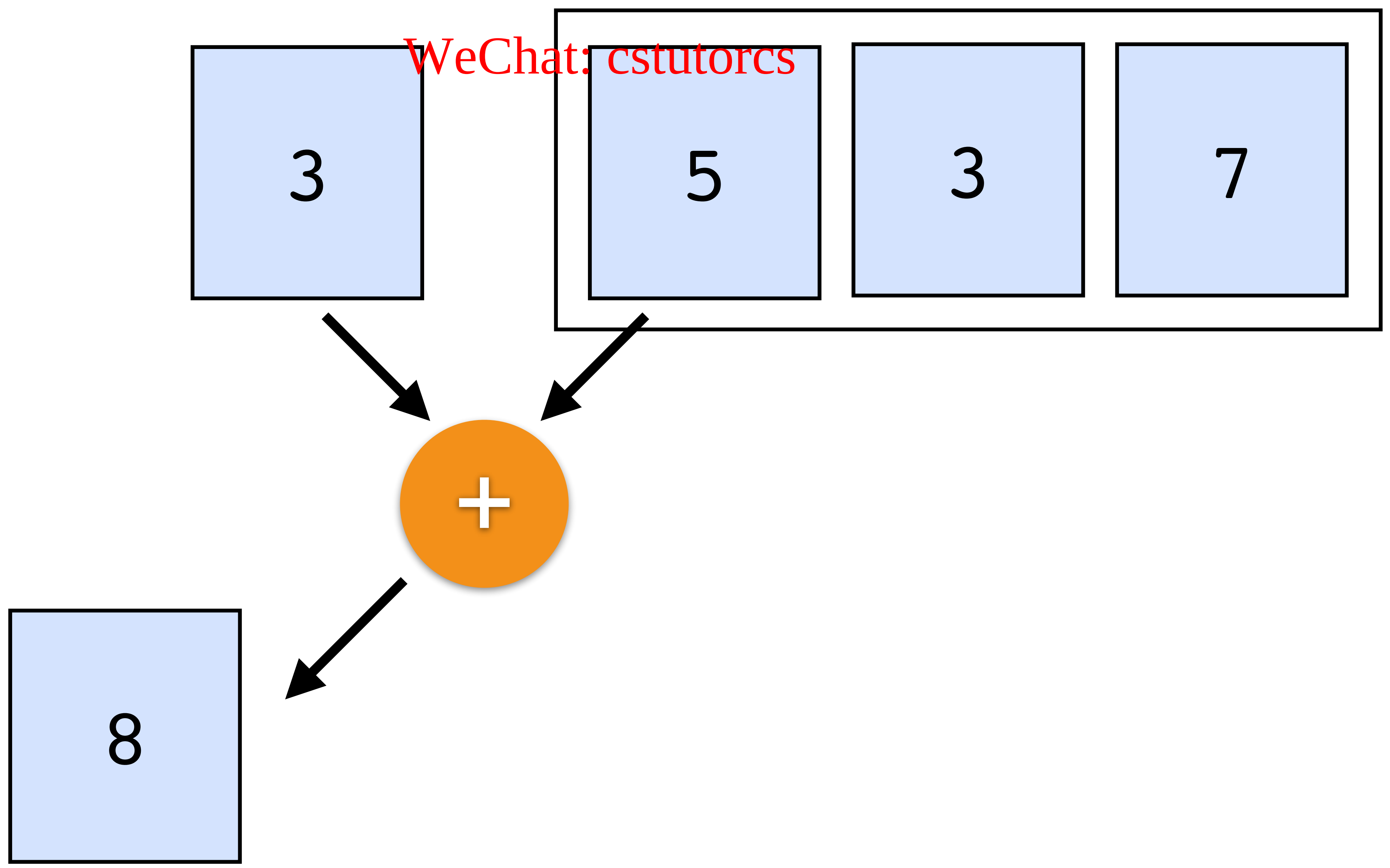
Fold Left

```
Prelude> foldl (+) 1 [2,5,3,7]
18
```

A binary function

Starting value

List to be "folded"



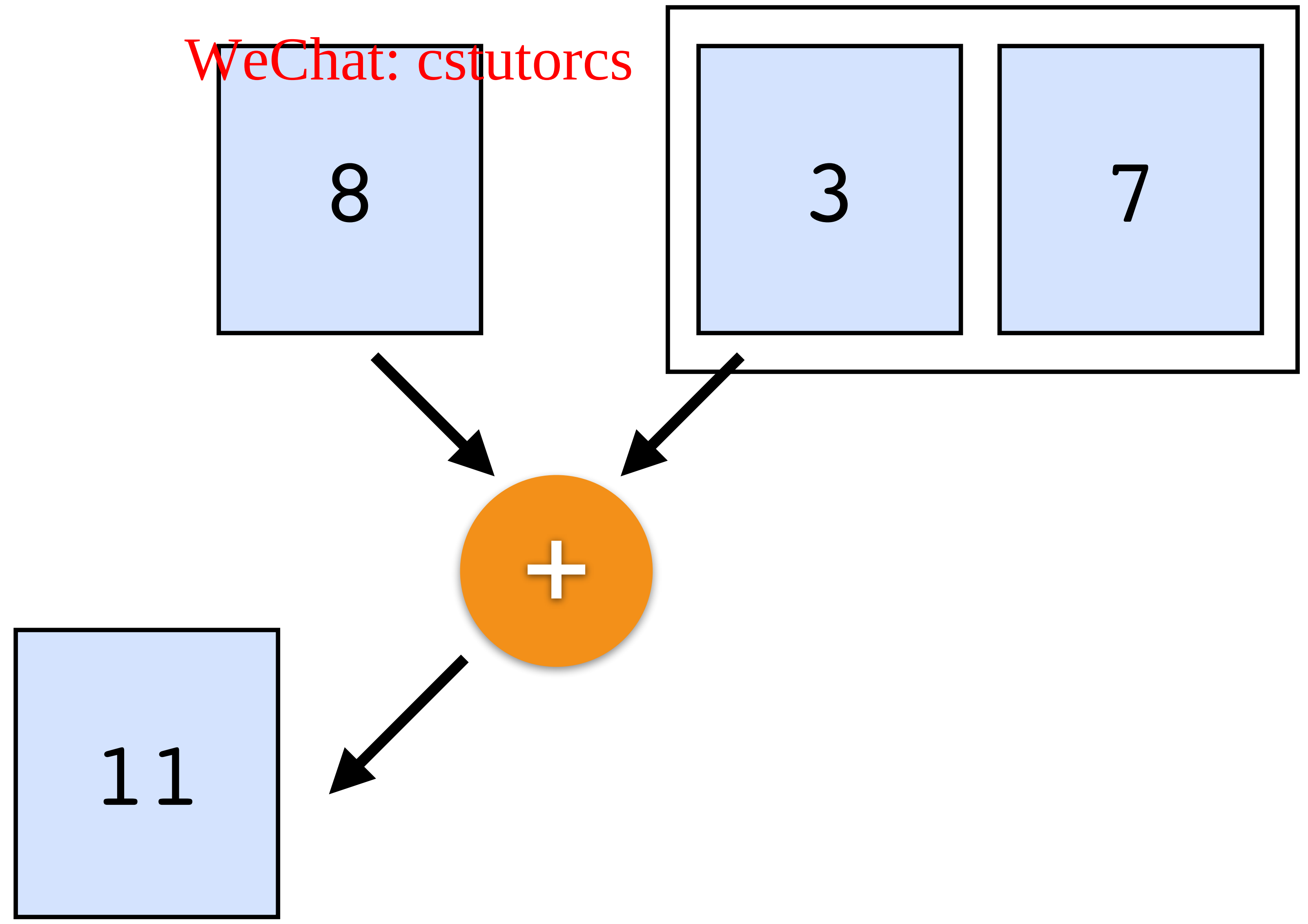
Fold Left

```
Prelude> foldl (+) 1 [2,5,3,7]
18
```

A binary function

Starting value

List to be “folded”



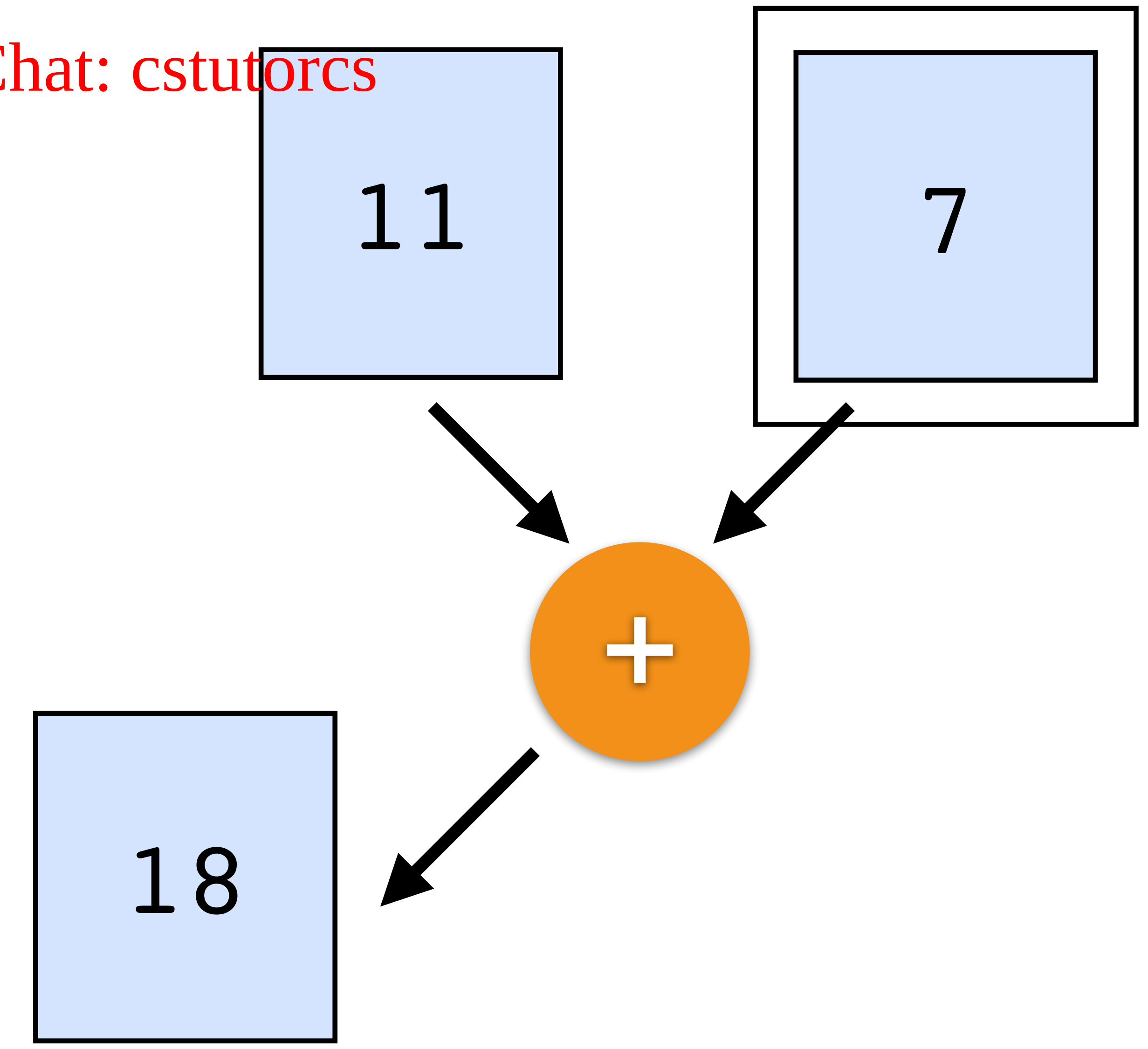
Fold Left

```
Prelude> foldl (+) 1 [2,5,3,7]
18
```

A binary function

Starting value

List to be “folded”



Fold Left

```
Prelude> foldl (+) 1 [2,5,3,7]
18
```

List to be "folded"

A binary function

Starting value

18

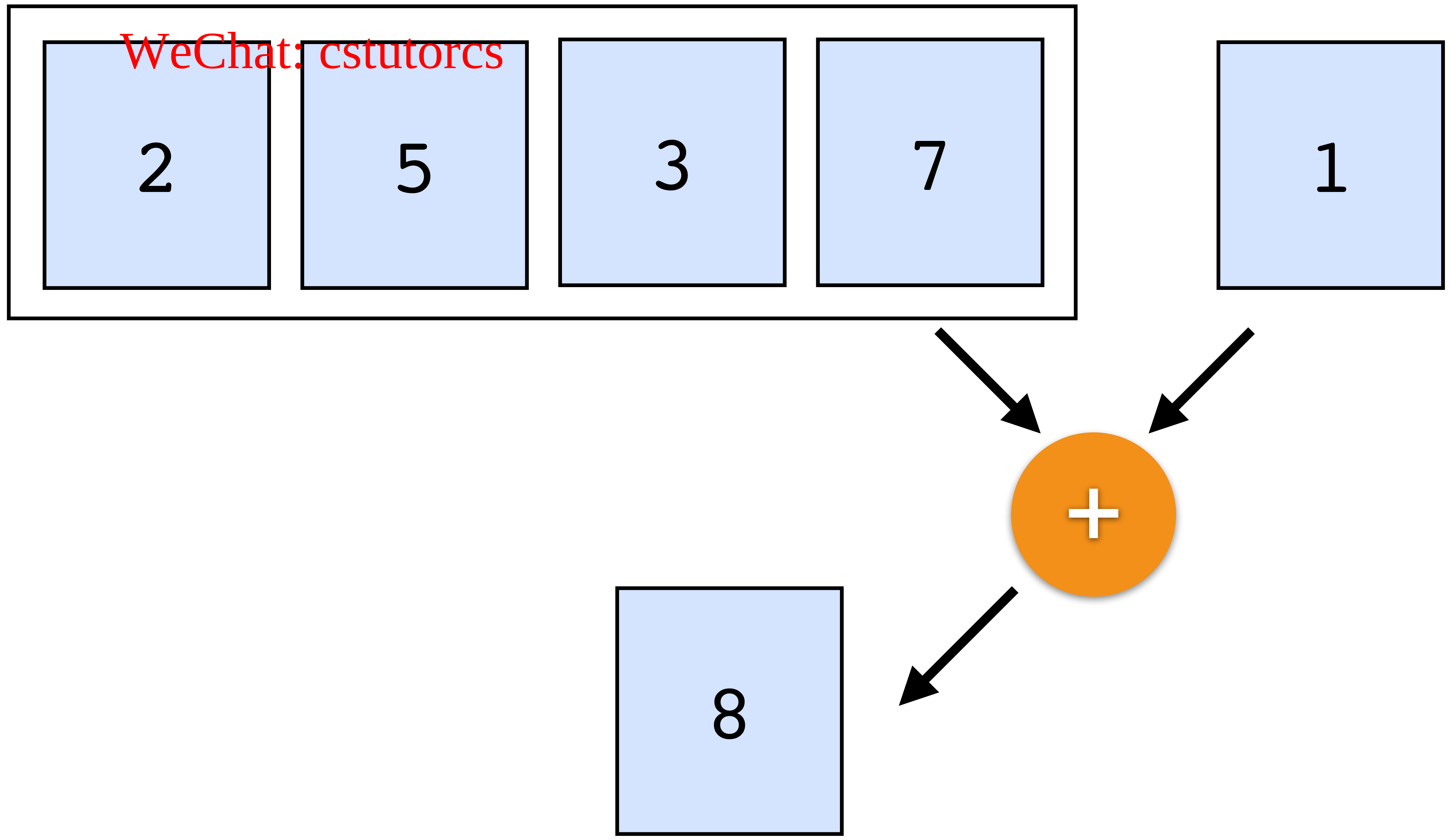
Fold Right

```
Prelude> foldr (+) 1 [2,5,3,7]
18
```

A binary function

Starting value

List to be "folded"



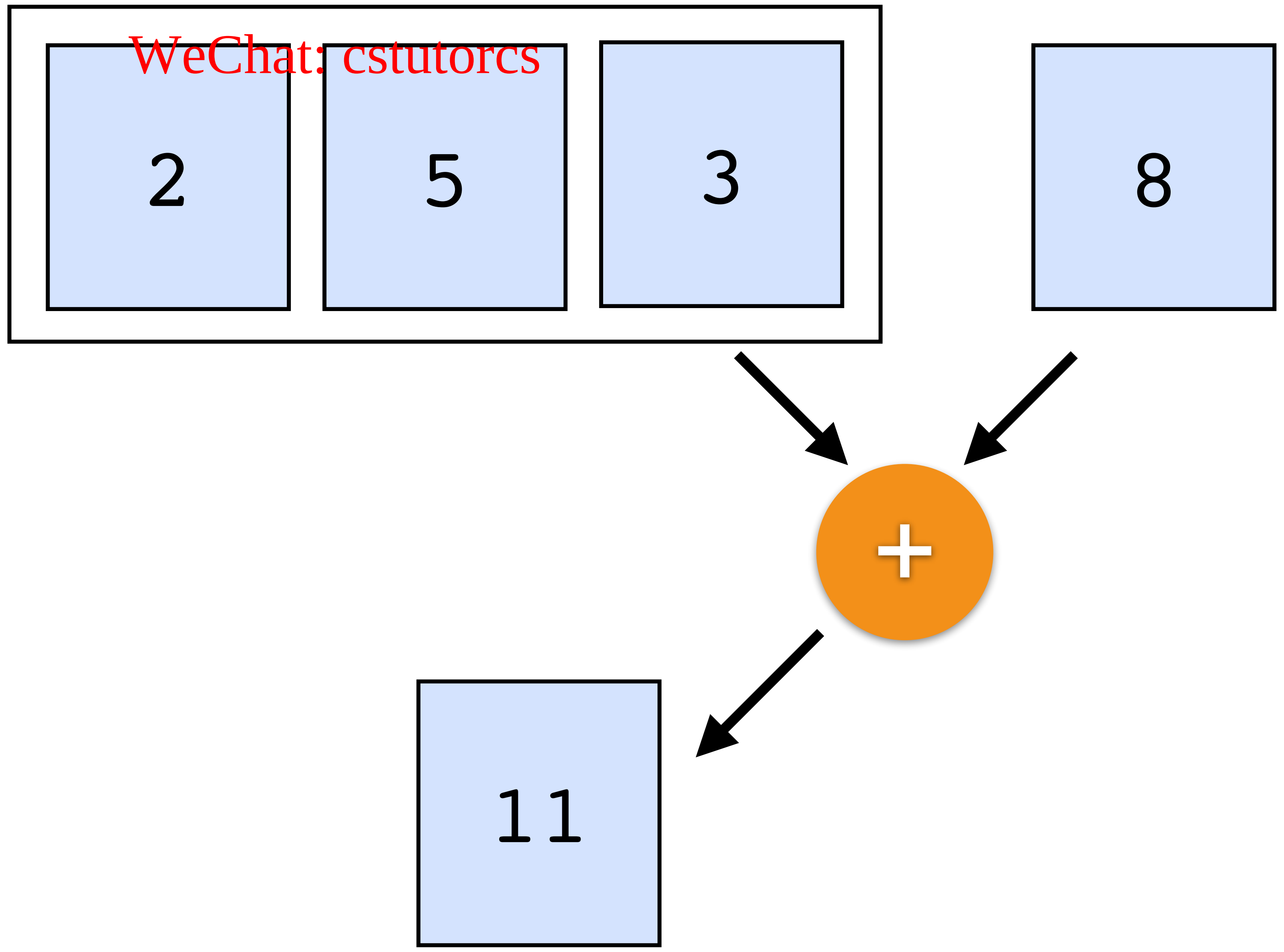
Fold Right

```
Prelude> foldr (+) 1 [2,5,3,7]
18
```

List to be “folded”

A binary function

Starting value



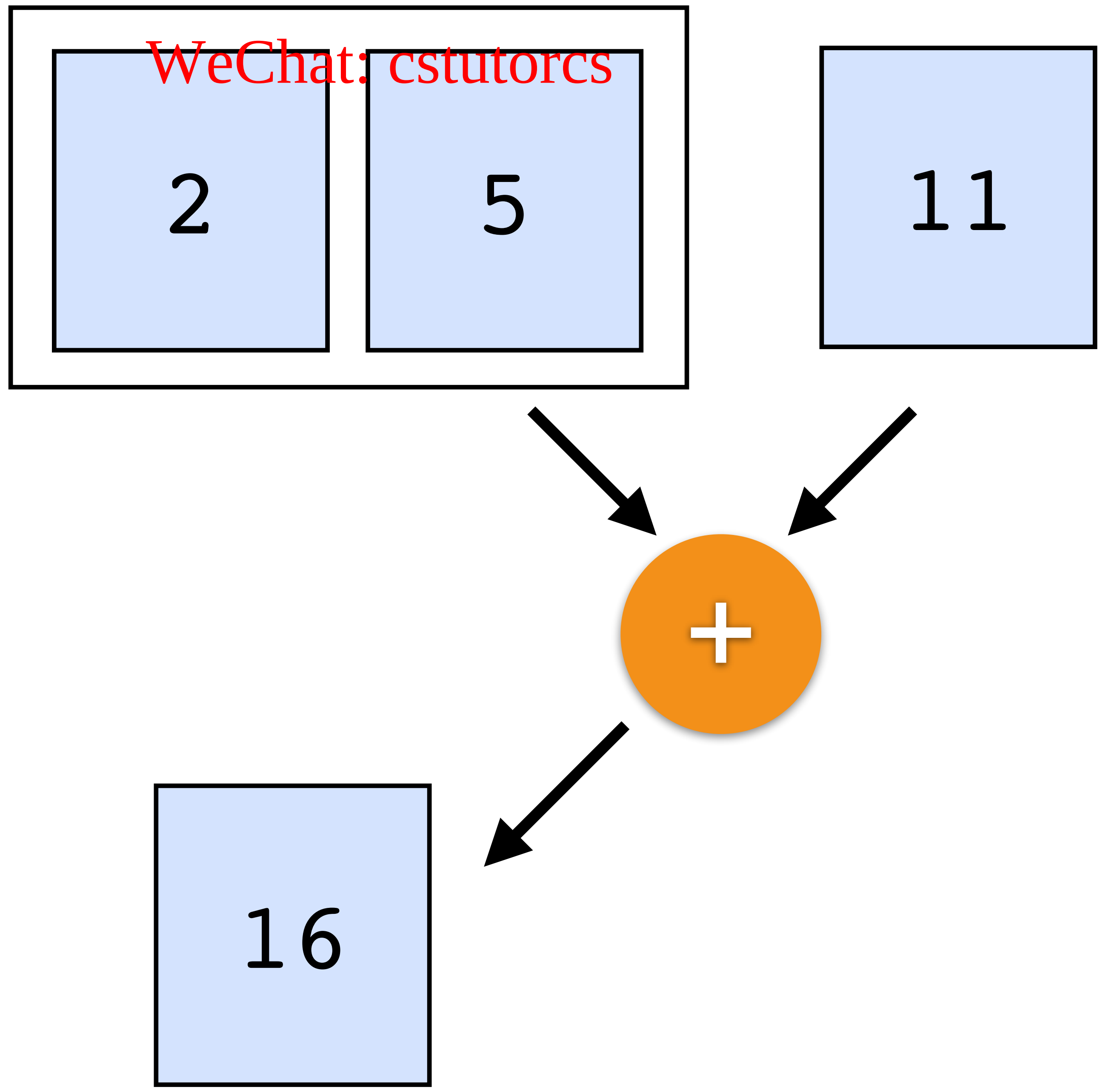
Fold Right

```
Prelude> foldr (+) 1 [2,5,3,7]
18
```

A binary function

Starting value

List to be "folded"



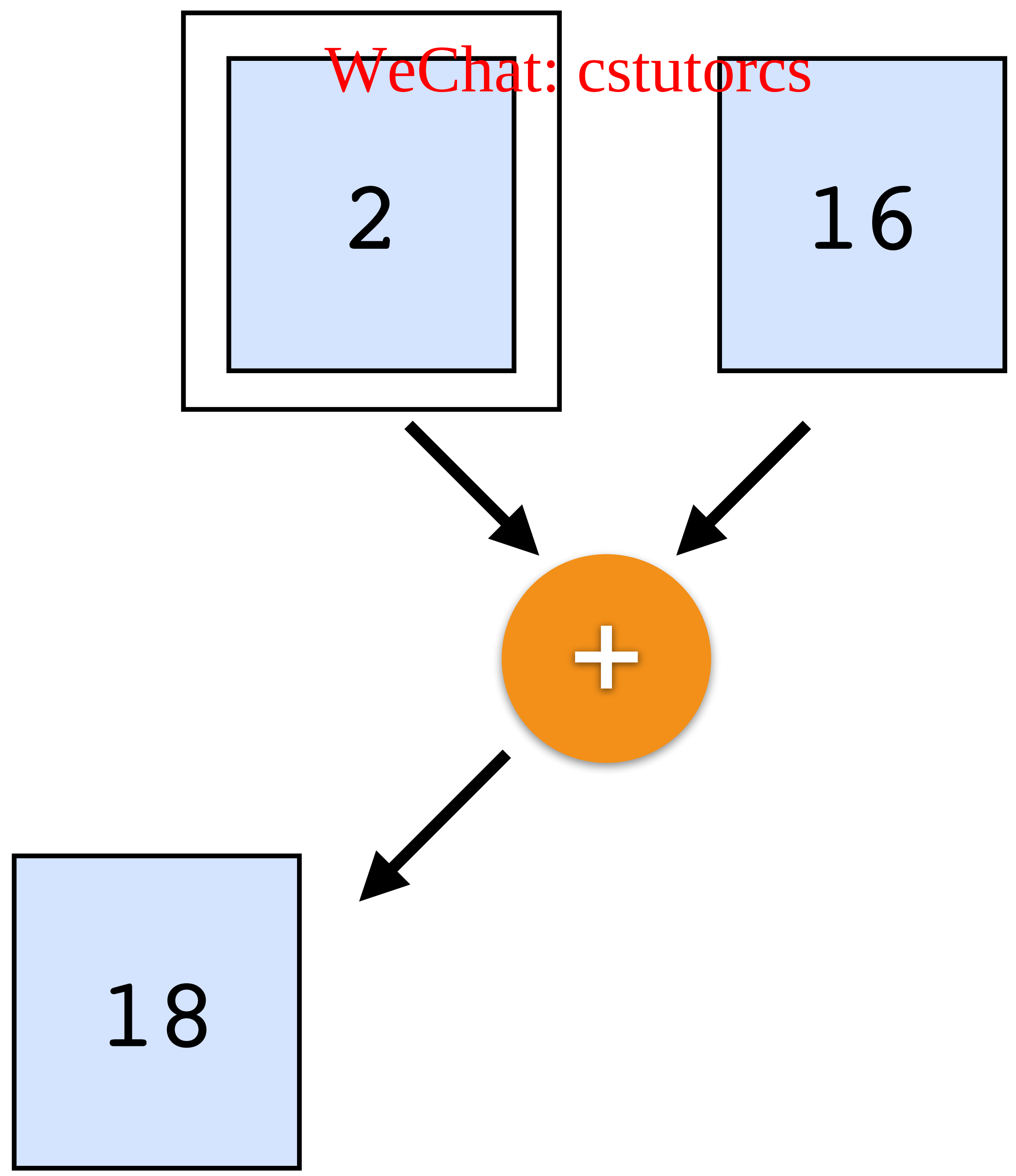
Fold Right

```
Prelude> foldr (+) 1 [2,5,3,7]
18
```

List to be "folded"

A binary function

Starting value



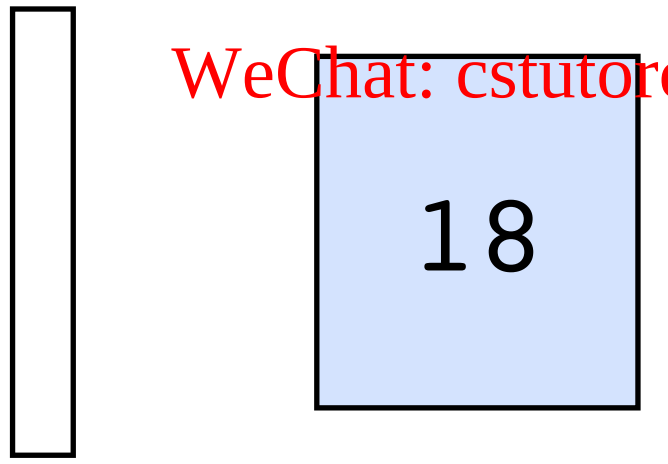
Fold Right

```
Prelude> foldr (+) 1 [2,5,3,7]
18
```

A binary function

Starting value

List to be "folded"



Fold Left / Right

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

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



```
Prelude> foldl (*) 1 [1..5]
120
Prelude> foldr (*) 1 [1..5]
120
Prelude> foldl (^) 2 [1..3]
64
Prelude> foldr (^) 2 [1..3]
1
```

((2^1)^2)^3 = 64

1^(2^(3^2)) = 1

foldl and **foldr** are built in functions
but actually they have simple recursive definitions


```
foldl f b [] = b
foldl f b (x:xs) = foldl f (f b x) xs

foldr f b [] = []
foldr f b (x:xs) = f x (foldr b xs)
```

foldl and **foldr** have recursive definitions, by recursion on the input list xs

Fold Right / Recursion

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

```
concat' [] = []
concat' (xs:xss) = xs ++ concat' xss

– refactor
concat' [] = []
concat' (xs:xss) = foo xs (concat’ xss)
      where foo xs a = xs ++ a

– rewrite with foldr
concat’ = foldr foo []
     where foo xs a = xs ++ a
```

foo xs (concat’ xss)

make the form explicit

Fold Left / Iterators

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

```
# Python
>>> y=0
>>> for x in [2,5,6]:
...     y = y+x
...
>>> y
13
```

Iterator: for loop

Block makes y function of y and x


```
– Haskell
foo y x = y+x
foldl foo 0 [2,5,6]
13
```

Block makes y function of y and x

Iterator: for loop

Google's MapReduce

(http://code.google.com/edu/parallel/mapreduce-tutorial.html#MapReduce)

(*although link is no longer available*)

Now that we have seen some basic examples of parallel programming, we can look at the MapReduce programming model. This model derives from the map and reduce combinators from a **functional language** like Lisp.

In Lisp, **a map takes as input a function and a sequence of values. It then applies the function to each value in the sequence. A reduce combines all the elements of a sequence using a binary operation.** For example, it can use "+" to add up all the elements in the sequence.

MapReduce is inspired by these concepts. It was developed within Google as a mechanism for processing large amounts of raw data, for example, crawled documents or web request logs. This data is so large, it must be distributed across thousands of machines in order to be processed in a reasonable time. This distribution implies parallel computing since the same computations are performed on each CPU, but with a different dataset. MapReduce is an abstraction that allows Google engineers to perform simple computations while hiding the details of parallelization, data distribution, load balancing and fault tolerance.

Map, written by a user of the MapReduce library, takes an input pair and produces a set of intermediate key/value pairs. The MapReduce library groups together all intermediate values associated with the same intermediate key I and passes them to the reduce function.

The reduce function, also written by the user, accepts an intermediate key I and a set of values for that key. It merges together these values to form a possibly smaller set of values.

Divide and conquer

Divide and conquer

- Classic algorithmic technique

- Split your problem into smaller ones
- Keep splitting until solution is trivial
- Combine solutions to get solution for large problem.

Divide and conquer: sorting

- Split list into smaller lists.
- Sort smaller lists.
- Combine sorted lists.

Divide and conquer

- Split list into smaller lists.

- Sort smaller lists.
- Combine sorted lists.
- Divide and conquer is always a candidate for **recursion**.

Divide and conquer: mergesort

- Split list into smaller lists - put no effort into this, but ideally into two halves.

- Sort smaller lists.
- Combine sorted lists: work done here using merge.

Divide and conquer: quicksort

- Split list into smaller lists - put effort into this: make sure everything in one list is smaller than anything in the other.

- Sort smaller lists.

- Combine sorted lists: now trivial since you can just concatenate.

Divide and conquer: functional mergesort

- Function to split list: simple like dealing cards


```
halve :: [a] -> ([a],[a])
halve [] = ([],[])
halve [a] = ([a],[])
halve (a1:a2:as) = let (t1,t2) = halve as in
   (a1:t1,a2:t2)
```

Divide and conquer: functional mergesort

- Function to merge sorted lists:


```
merge :: Ord a => [a] -> [a] -> [a]
merge as [] = as
merge [] (b:bs) = b:bs
merge (a:as) (b:bs) | a <= b = a:(merge as (b:bs))
merge (a:as) (b:bs) | otherwise = b:(merge (a:as) bs)
```

Divide and conquer: functional mergesort

- Put them together with trivial cases


```
mergesort :: [a] -> [a] -> [a]
mergesort [] = []
mergesort [a] = [a]
mergesort (a1:a2:as) = let (t1,t2) = halve (a1:a2:as) in
  merge (mergesort t1) (mergesort t2)
```

Divide and conquer: functional quicksort


```
quicksort :: Ord a => [a] -> [a] -> [a]
quicksort [] = []
quicksort (a:as) = let t1 = [x | x <- as, x<=a]
                       t2 = [x | x <- as, a<x]
                       in
                       (quicksort t1)++[a]++(quicksort t2)
```

https://cloudsurvey.co.uk/live/quiz/224526/

224 526