Staging (Higher-Order Functions in Action)

15-150

Lecture 11: October 3, 2024

Stephanie Balzer
Carnegie Mellon University

Can we generalize map and fold?

So fare we have considered map and fold exclusively for lists.

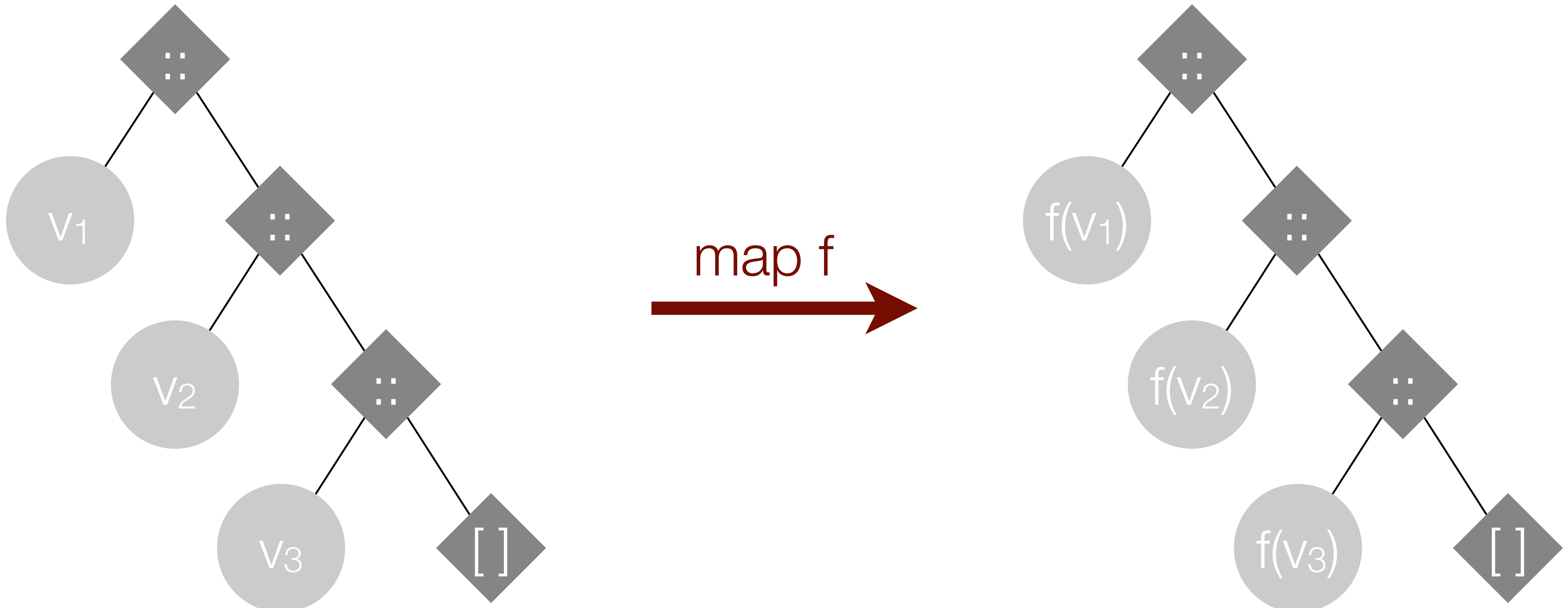
- map: transform elements in a list, given a transformation function
- fold: combining elements in a list, given a binary operation and base value

Can we generalize map and fold to, for example, binary trees?

- Yes! Let's work it out.
- It may be helpful to visualize map and fold for lists diagrammatically first, to capture the underlying pattern.

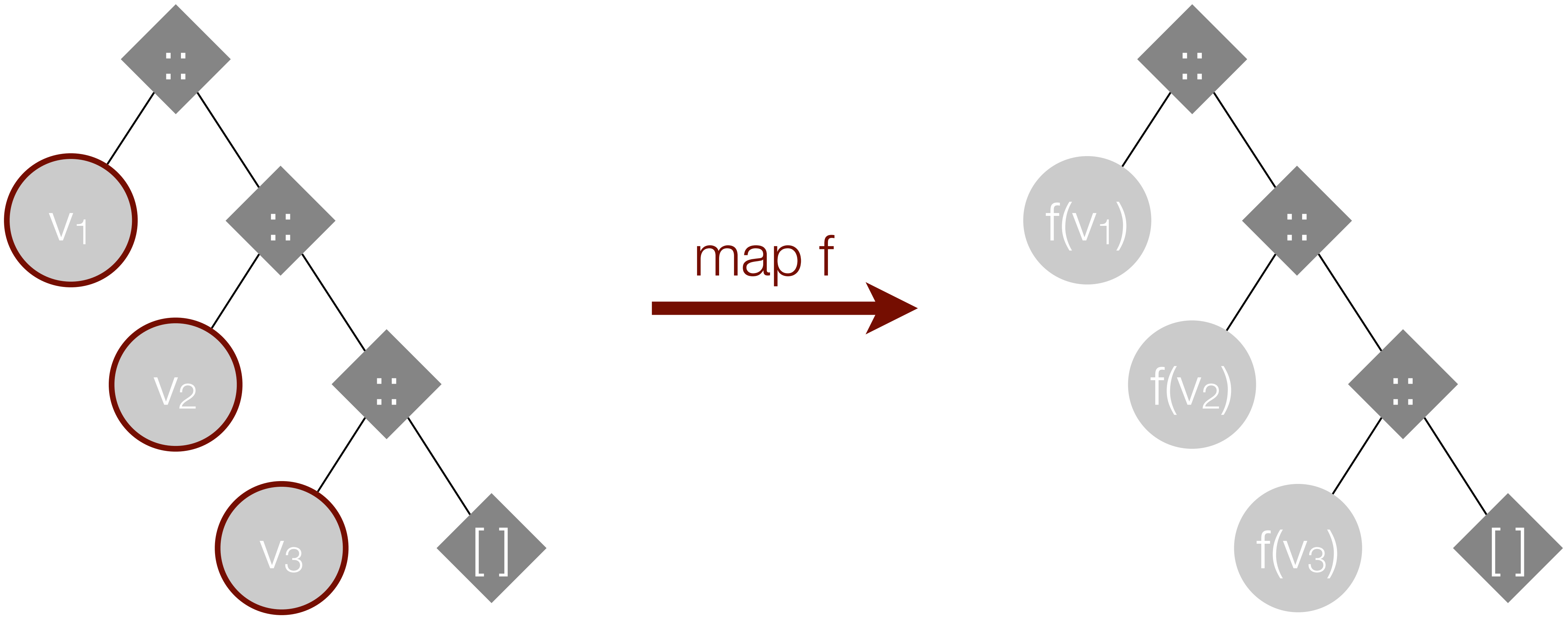
The “pattern” underlying map

```
(* map: ('a -> 'b) -> 'a list -> 'b list *)
```



The “pattern” underlying map

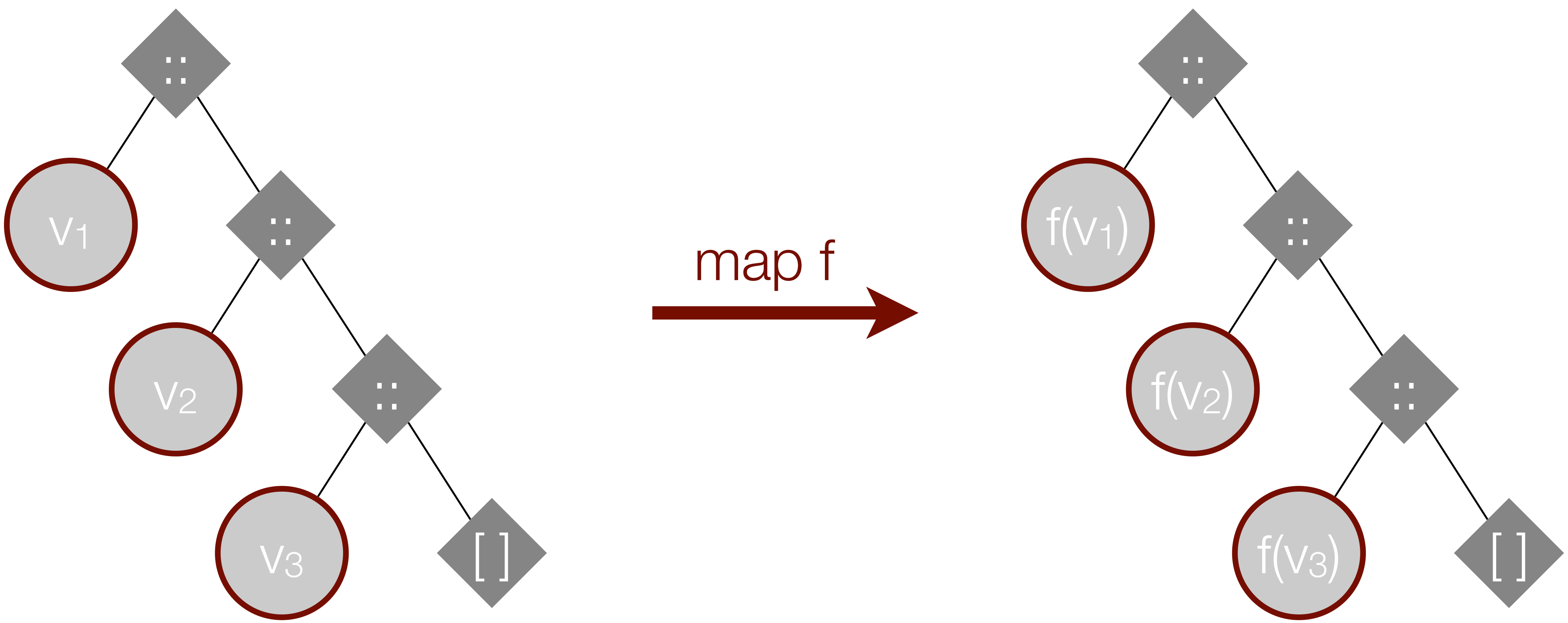
```
(* map: ('a -> 'b) -> 'a list -> 'b list *)
```



Replace every element value v_i with its transformed value $f(v_i)$.

The “pattern” underlying map

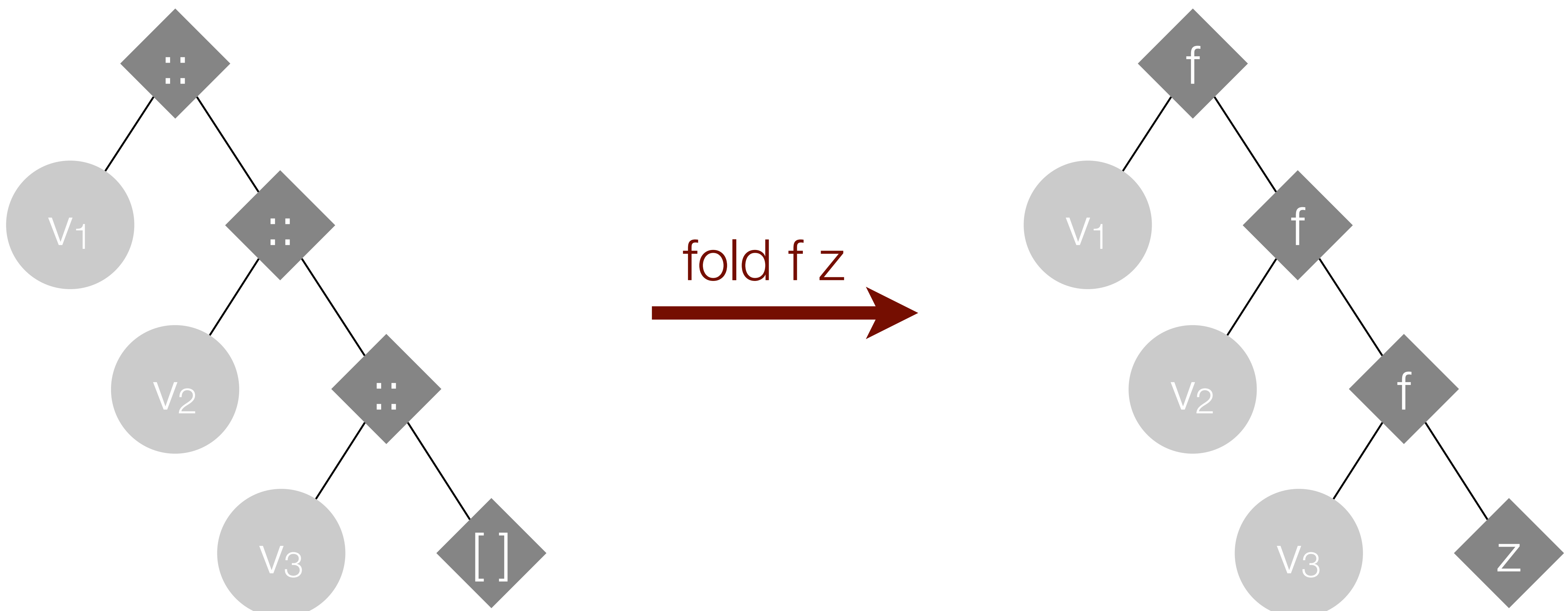
```
(* map: ('a -> 'b) -> 'a list -> 'b list *)
```



Replace every element value v_i with its transformed value $f(v_i)$.

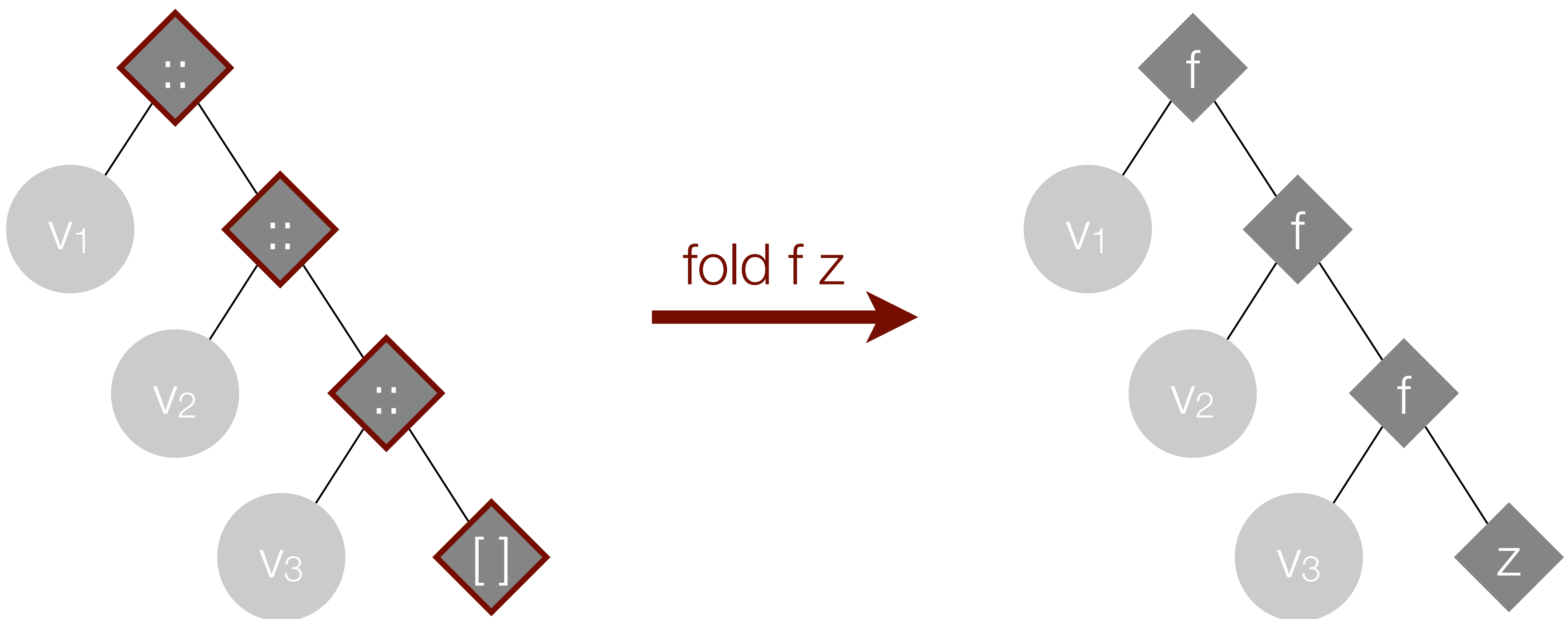
The “pattern” underlying fold

```
(* fold: ('a * 'b -> 'b) -> 'b -> 'a list -> 'b *)
```



The “pattern” underlying fold

```
(* fold: ('a * 'b -> 'b) -> 'b -> 'a list -> 'b *)
```



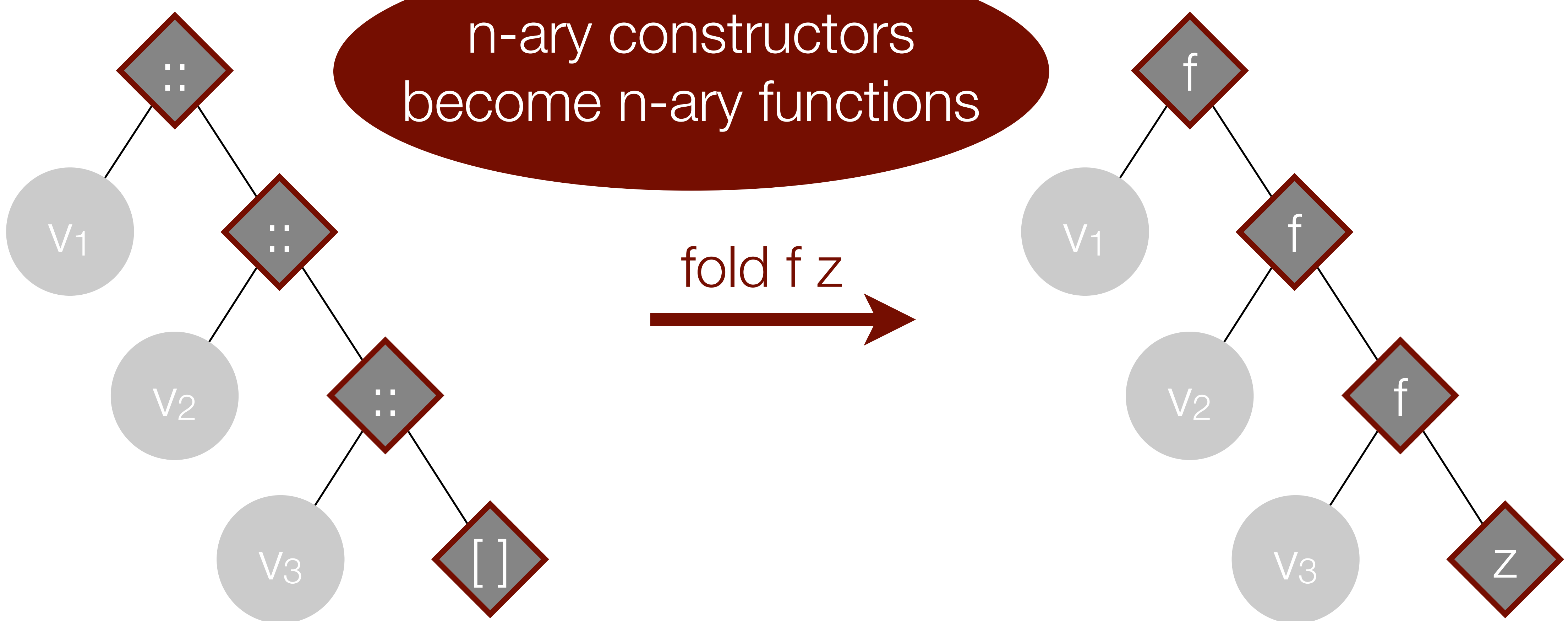
Replace every constructor with a function or value.

The “pattern” underlying fold

catamorphism

```
(* fold: ('a * 'b -> 'b) -> 'b -> 'a list -> 'b *)
```

n-ary constructors
become n-ary functions



Replace every constructor with a function or value.

Map and fold for binary trees

```
datatype 'a tree = Empty | Node of 'a tree * 'a * 'a tree
```

Map and fold for binary trees

```
datatype 'a tree = Empty | Node of 'a tree * 'a * 'a tree

(* tmap : ('a -> 'b) -> 'a tree -> 'b tree *)
```

Map and fold for binary trees

```
datatype 'a tree = Empty | Node of 'a tree * 'a * 'a tree

(* tmap : ('a -> 'b) -> 'a tree -> 'b tree *)
fun tmap f Empty =
```

Map and fold for binary trees

```
datatype 'a tree = Empty | Node of 'a tree * 'a * 'a tree

(* tmap : ('a -> 'b) -> 'a tree -> 'b tree *)
fun tmap f Empty = Empty
```

Map and fold for binary trees

```
datatype 'a tree = Empty | Node of 'a tree * 'a * 'a tree

(* tmap : ('a -> 'b) -> 'a tree -> 'b tree *)
fun tmap f Empty = Empty
  | tmap f (Node(l,x,r)) =
```

Map and fold for binary trees

```
datatype 'a tree = Empty | Node of 'a tree * 'a * 'a tree

(* tmap : ('a -> 'b) -> 'a tree -> 'b tree *)
fun tmap f Empty = Empty
  | tmap f (Node(l,x,r)) = Node(                          )
```

Map and fold for binary trees

```
datatype 'a tree = Empty | Node of 'a tree * 'a * 'a tree

(* tmap : ('a -> 'b) -> 'a tree -> 'b tree *)
fun tmap f Empty = Empty
  | tmap f (Node(l,x,r)) = Node(tmap f l, f x, tmap f r)
```

Map and fold for binary trees

```
datatype 'a tree = Empty | Node of 'a tree * 'a * 'a tree

(* tmap : ('a -> 'b) -> 'a tree -> 'b tree *)
fun tmap f Empty = Empty
  | tmap f (Node(l,x,r))                  x, tmap f r)

(* tfold : ('b * 'a * 'b -> 'b) -> 'b -> 'a tree -> 'b *)
```

same number of arguments as constructor

result of fold of left subtree

result of fold of right subtree

base value for empty

Map and fold for binary trees

```
datatype 'a tree = Empty | Node of 'a tree * 'a * 'a tree

(* tmap : ('a -> 'b) -> 'a tree -> 'b tree *)
fun tmap f Empty = Empty
  | tmap f (Node(l,x,r)) = Node(tmap f l, f x, tmap f r)

(* tfold : ('b * 'a * 'b -> 'b) -> 'b -> 'a tree -> 'b *)
```

Map and fold for binary trees

```
datatype 'a tree = Empty | Node of 'a tree * 'a * 'a tree

(* tmap : ('a -> 'b) -> 'a tree -> 'b tree *)
fun tmap f Empty = Empty
  | tmap f (Node(l,x,r)) = Node(tmap f l, f x, tmap f r)

(* tfold : ('b * 'a * 'b -> 'b) -> 'b -> 'a tree -> 'b *)
fun tfold f z Empty =
```

Map and fold for binary trees

```
datatype 'a tree = Empty | Node of 'a tree * 'a * 'a tree

(* tmap : ('a -> 'b) -> 'a tree -> 'b tree *)
fun tmap f Empty = Empty
  | tmap f (Node(l,x,r)) = Node(tmap f l, f x, tmap f r)

(* tfold : ('b * 'a * 'b -> 'b) -> 'b -> 'a tree -> 'b *)
fun tfold f z Empty = z
```

Map and fold for binary trees

```
datatype 'a tree = Empty | Node of 'a tree * 'a * 'a tree

(* tmap : ('a -> 'b) -> 'a tree -> 'b tree *)
fun tmap f Empty = Empty
  | tmap f (Node(l,x,r)) = Node(tmap f l, f x, tmap f r)

(* tfold : ('b * 'a * 'b -> 'b) -> 'b -> 'a tree -> 'b *)
fun tfold f z Empty = z
  | tfold f z (Node (l, x, r)) =
```

Map and fold for binary trees

```
datatype 'a tree = Empty | Node of 'a tree * 'a * 'a tree

(* tmap : ('a -> 'b) -> 'a tree -> 'b tree *)
fun tmap f Empty = Empty
  | tmap f (Node(l,x,r)) = Node(tmap f l, f x, tmap f r)

(* tfold : ('b * 'a * 'b -> 'b) -> 'b -> 'a tree -> 'b *)
fun tfold f z Empty = z
   | tfold f z (Node (l, x, r)) =
     f (                             )
```

Map and fold for binary trees

```
datatype 'a tree = Empty | Node of 'a tree * 'a * 'a tree

(* tmap : ('a -> 'b) -> 'a tree -> 'b tree *)
fun tmap f Empty = Empty
  | tmap f (Node(l,x,r)) = Node(tmap f l, f x, tmap f r)

(* tfold : ('b * 'a * 'b -> 'b) -> 'b -> 'a tree -> 'b *)
fun tfold f z Empty = z
   | tfold f z (Node (l, x, r)) =
     f (tfold f z l, x, tfold f z r)
```

Examples for tmap and tfold

```
val stringify = tmap Int.toString

val treesum = tfold (fn (a,x,b) => a+x+b) 0
```

Examples for tmap and tfold

```
val stringify = tmap Int.toString
```

```
val treesum = tfold (fn (a,x,b) => a+x+b) 0
```

What are the types of `stringify` and `treesum`?

Examples for tmap and tfold

```
val stringify = tmap Int.toString
```

```
val treesum = tfold (fn (a,x,b) => a+x+b) 0
```

What are the types of `stringify` and `treesum`?

```
(* stringify :                          *)
```

Examples for tmap and tfold

```
(* tmap : ('a -> 'b) -> 'a tree -> 'b tree *)
val stringify = tmap Int.toString


val treesum = tfold (fn (a,x,b) => a+x+b) 0
```

What are the types of `stringify` and `treesum`?

```
(* stringify :                          *)
```

Examples for tmap and tfold

```
(* tmap : ('a -> 'b) -> 'a tree -> 'b tree *)
val stringify = tmap Int.toString


val treesum = tfold (fn (a,x,b) => a+x+b) 0
```

What are the types of `stringify` and `treesum`?

```
(* stringify : int tree -> string tree *)
```

Examples for tmap and tfold

```
(* tmap : ('a -> 'b) -> 'a tree -> 'b tree *)
val stringify = tmap Int.toString


val treesum = tfold (fn (a,x,b) => a+x+b) 0
```

What are the types of `stringify` and `treesum`?

```
(* stringify : int tree -> string tree *)
(* treesum :                       *)
```

Examples for tmap and tfold

```
(* tmap : ('a -> 'b) -> 'a tree -> 'b tree *)
val stringify = tmap Int.toString

(* tfold : ('b * 'a * 'b -> 'b) -> 'b -> 'a tree -> 'b *)
val treesum = tfold (fn (a,x,b) => a+x+b) 0
```

What are the types of `stringify` and `treesum`?

```
(* stringify : int tree -> string tree *)

(* treesum :                           *)
```

Examples for tmap and tfold

```
(* tmap : ('a -> 'b) -> 'a tree -> 'b tree *)

val stringify = tmap Int.toString

(* tfold : ('b * 'a * 'b -> 'b) -> 'b -> 'a tree -> 'b *)
val treesum = tfold (fn (a,x,b) => a+x+b) 0
```

What are the types of `stringify` and `treesum`?

```
(* stringify : int tree -> string tree *)

(* treesum : int tree -> int *)
```

Map and fold for leafy binary trees

```
datatype 'a leafy = Leaf of ‘a
                  | Node of 'a leafy * 'a leafy
```

Map and fold for leafy binary trees

```
datatype 'a leafy = Leaf of ‘a
                  | Node of 'a leafy * 'a leafy


(* lmap: ('a -> 'b) -> 'a leafy -> 'b leafy *)
```

Map and fold for leafy binary trees

```
datatype 'a leafy = Leaf of ‘a
                  | Node of 'a leafy * 'a leafy
```

```
(* lmap: ('a -> 'b) -> 'a leafy -> 'b leafy *)
fun lmap f Leaf(x) =
```

Map and fold for leafy binary trees

```
datatype 'a leafy = Leaf of ‘a
                  | Node of 'a leafy * 'a leafy

(* lmap: ('a -> 'b) -> 'a leafy -> 'b leafy *)
fun lmap f Leaf(x) = Leaf(f x)
```

Map and fold for leafy binary trees

```
datatype 'a leafy = Leaf of ‘a
                  | Node of 'a leafy * 'a leafy

(* lmap: ('a -> 'b) -> 'a leafy -> 'b leafy *)
fun lmap f Leaf(x) = Leaf(f x)
  | lmap f (Node(l,r)) =
```

Map and fold for leafy binary trees

```
datatype 'a leafy = Leaf of ‘a
                  | Node of 'a leafy * 'a leafy

(* lmap: ('a -> 'b) -> 'a leafy -> 'b leafy *)
fun lmap f Leaf(x) = Leaf(f x)
  | lmap f (Node(l,r)) = Node(lmap f l, lmap f r)
```

Map and fold for leafy binary trees

```
datatype 'a leafy = Leaf of ‘a
                  | Node of 'a leafy * 'a leafy

(* lmap: ('a -> 'b) -> 'a leafy -> 'b leafy *)
fun lmap f Leaf(x) = Leaf(f x)
  | lmap f (Node(l,r)) = Node(lmap f l, lmap f r)

(* lfold: ('b*'b -> 'b) -> ('a->'b) -> 'a leafy -> 'b *)
```

Map and fold for leafy binary trees

```
datatype 'a leafy = Leaf of ‘a
                  | Node of 'a leafy * 'a leafy

(* lmap: ('a -> 'b) -> 'a leafy -> 'b leafy *)
fun lmap f Leaf(x) = Leaf(f x)
  | lmap f (Node(l,r)) = Node(lmap f l, lmap f r)

(* lfold: ('b*'b -> 'b) -> ('a->'b) -> 'a leafy -> 'b *)
fun lfold f g Leaf(x) =
```

Map and fold for leafy binary trees

```
datatype 'a leafy = Leaf of ‘a
                  | Node of 'a leafy * 'a leafy

(* lmap: ('a -> 'b) -> 'a leafy -> 'b leafy *)
fun lmap f Leaf(x) = Leaf(f x)
  | lmap f (Node(l,r)) = Node(lmap f l, lmap f r)

(* lfold: ('b*'b -> 'b) -> ('a->'b) -> 'a leafy -> 'b *)
fun lfold f g Leaf(x) = g(x)
```

Map and fold for leafy binary trees

```
datatype 'a leafy = Leaf of ‘a
                  | Node of 'a leafy * 'a leafy

(* lmap: ('a -> 'b) -> 'a leafy -> 'b leafy *)
fun lmap f Leaf(x) = Leaf(f x)
  | lmap f (Node(l,r)) = Node(lmap f l, lmap f r)

(* lfold: ('b*'b -> 'b) -> ('a->'b) -> 'a leafy -> 'b *)
fun lfold f g Leaf(x) = g(x)
  | lfold f g (Node (l, r)) =
```

Map and fold for leafy binary trees

```
datatype 'a leafy = Leaf of ‘a
                  | Node of 'a leafy * 'a leafy

(* lmap: ('a -> 'b) -> 'a leafy -> 'b leafy *)
fun lmap f Leaf(x) = Leaf(f x)
  | lmap f (Node(l,r)) = Node(lmap f l, lmap f r)

(* lfold: ('b*'b -> 'b) -> ('a->'b) -> 'a leafy -> 'b *)
fun lfold f g Leaf(x) = g(x)
   | lfold f g (Node (l, r)) =
     f (lfold f g l, lfold f g r)
```

Examples for lmap and lfold

```
(* lmap: ('a -> 'b) -> 'a leafy -> 'b leafy *)

val lstringify = lmap Int.toString

(* lfold: ('b*'b -> 'b) -> ('a->'b) -> 'a leafy -> 'b *)
val leafysum = lfold (op +) (fn x => x)
```

Examples for lmap and lfold

```
(* lmap: ('a -> 'b) -> 'a leafy -> 'b leafy *)
val lstringify = lmap Int.toString

(* lfold: ('b*'b -> 'b) -> ('a->'b) -> 'a leafy -> 'b *)
val leafysum = lfold (op +) (fn x => x)
```

What are the types of `ltringify` and `leafysum`?

Examples for lmap and lfold

```
(* lmap: ('a -> 'b) -> 'a leafy -> 'b leafy *)

val lstringify = lmap Int.toString

(* lfold: ('b*'b -> 'b) -> ('a->'b) -> 'a leafy -> 'b *)
val leafysum = lfold (op +) (fn x => x)
```

What are the types of `ltringify` and `leafysum`?

```
(* lstringify : int leafy -> string leafy *)

(* leafysum : int leafy -> int *)
```

Map and fold for non-recursive datatypes

```
datatype 'a option = NONE | SOME of 'a
```

Map and fold for non-recursive datatypes

```
datatype 'a option = NONE | SOME of 'a

(* opmap: ('a -> 'b) -> 'a option -> 'b option *)
```

Map and fold for non-recursive datatypes

```
datatype 'a option = NONE | SOME of 'a

(* opmap: ('a -> 'b) -> 'a option -> 'b option *)
fun opmap f NONE =
```

Map and fold for non-recursive datatypes

```
datatype 'a option = NONE | SOME of 'a

(* opmap: ('a -> 'b) -> 'a option -> 'b option *)
fun opmap f NONE = NONE
```

Map and fold for non-recursive datatypes

```
datatype 'a option = NONE | SOME of 'a

(* opmap: ('a -> 'b) -> 'a option -> 'b option *)
fun opmap f NONE = NONE
  | opmap f (SOME x) =
```

Map and fold for non-recursive datatypes

```
datatype 'a option = NONE | SOME of 'a

(* opmap: ('a -> 'b) -> 'a option -> 'b option *)
fun opmap f NONE = NONE
  | opmap f (SOME x) = SOME (f x)
```

Map and fold for non-recursive datatypes

```
datatype 'a option = NONE | SOME of 'a

(* opmap: ('a -> 'b) -> 'a option -> 'b option *)
fun opmap f NONE = NONE
  | opmap f (SOME x) = SOME (f x)

(* opfold: ('a -> 'b) -> 'b -> 'a option -> 'b *)
```

Map and fold for non-recursive datatypes

```
datatype 'a option = NONE | SOME of 'a

(* opmap: ('a -> 'b) -> 'a option -> 'b option *)
fun opmap f NONE = NONE
  | opmap f (SOME x) = SOME (f x)

(* opfold: ('a-> 'b) -> 'b -> 'a option -> 'b *)
fun opfold f z NONE
```

Map and fold for non-recursive datatypes

```
datatype 'a option = NONE | SOME of 'a

(* opmap: ('a -> 'b) -> 'a option -> 'b option *)
fun opmap f NONE = NONE
  | opmap f (SOME x) = SOME (f x)

(* opfold: ('a -> 'b) -> 'b -> 'a option -> 'b *)
fun opfold f z NONE = z
```

Map and fold for non-recursive datatypes

```
datatype 'a option = NONE | SOME of 'a

(* opmap: ('a -> 'b) -> 'a option -> 'b option *)
fun opmap f NONE = NONE
  | opmap f (SOME x) = SOME (f x)

(* opfold: ('a-> 'b) -> 'b -> 'a option -> 'b *)
fun opfold f z NONE = z
   | opfold f z (SOME x) =
```

Map and fold for non-recursive datatypes

```
datatype 'a option = NONE | SOME of 'a

(* opmap: ('a -> 'b) -> 'a option -> 'b option *)
fun opmap f NONE = NONE
  | opmap f (SOME x) = SOME (f x)

(* opfold: ('a-> 'b) -> 'b -> 'a option -> 'b *)
fun opfold f z NONE = z
  | opfold f z (SOME x) = f x
```

Examples for opmap and opfold

```
(* opmap: ('a -> 'b) -> 'a option -> 'b option *)

val ostringify = opmap Int.toString

(* opfold: ('a -> 'b) -> 'b -> 'a option -> 'b *)
val osum = opfold (fn x => x) 0
```

Examples for opmap and opfold

```
(* opmap: ('a -> 'b) -> 'a option -> 'b option *)

val ostringify = opmap Int.toString

(* opfold: ('a -> 'b) -> 'b -> 'a option -> 'b *)
val osum = opfold (fn x => x) 0
```

What are the types of `otringify` and `osum`?

Examples for opmap and opfold

```
(* opmap: ('a -> 'b) -> 'a option -> 'b option *)

val ostringify = opmap Int.toString

(* opfold: ('a -> 'b) -> 'b -> 'a option -> 'b *)
val osum = opfold (fn x => x) 0
```

What are the types of `otringify` and `osum`?

```
(* ostringify : int option -> string option *)

(* osum : int option -> int *)
```

Another use of HOF: Staging

Staging is a coding technique
that has a function perform useful work
prior to receiving all its arguments.

- Concern: efficiency (“cost”) of evaluation
- Employs partial application
  - to factor out expensive part
  - to specialize inexpensive part for specific argument.
- Improves efficiency when specialized function used many times.

Staging

Consider the following function:

```
fun f (x:int, y:int) : int =
    let
       val z : int = horriblecomputation(x)
    in
       z + y
    end
```

Suppose the horrible computation takes 10 months.
(And suppose that addition takes a picosecond.)

Then each of these expressions takes at least 10 months to evaluate:

```
f (5,2)
f (5,3)
```

without mutation

If only we could recall `horriblecomputation(5)`!

Staging

Consider the following function:

```
fun f (x:int, y:int) : int =
    let
       val z : int = horriblecomputation(x)
    in
       z + y
    end
```

What is the type of `f`?

```
(* f : int * int -> int *)
```

➡ Maybe currying can help?

➡ Let’s define a curried version of f!

Staging

Curried version of f:

```
fun g (x:int) (y:int) : int =
    let
       val z : int = horriblecomputation(x)
    in
       z + y
    end
```

Now the type of **g** is `(* g : int -> int -> int *)`,

so we can define `val g5 : int -> int = g(5)`

and then evaluate

```
g5 (2)  (* instead of f (5,2) *)
g5 (3)  (* instead of f (5,3) *)
```

➡ How long do the 3 lines above take?

Staging

➡ How long do the 3 lines above take?

Remember, the declaration of **g** created the following binding:

```
[(fn x => fn y => let val z = hc(x) in z+y end)/g]
```

In declaring `val g5 = g(5)`, one evaluates

```
[(fn x => fn y => let val z = hc(x) in z+y end)/g] g(5)
==> (fn x => fn y => let val z = hc(x) in z+y end) (5)
==> [5/x] fn y => let val z = hc(x) in z+y end
```

This is a lambda, and thus s a value!

No application, and thus no evaluation of body!

Staging

How long do the 3 lines above take?

Remember, the declaration of **g** created the following binding:

```
[(fn x => fn y => let val z = hc(x) in z+y end)/g]
```

In declaring `val g5 = g(5)`, one evaluates

```
[(fn x => fn y => let val z = hc(x) in z+y end)/g] g(5)
==> (fn x => fn y => let val z = hc(x) in z+y end) (5)
==> [5/x] fn y => let val z = hc(x) in z+y end
```

This is the closure returned by `g(5)`.

The horrible computation has not yet happened :-(

Staging

We now have the following binding:

```
[ env
  [5/x]
  fn y => let val z = hc(x) in z+y end  /g5]
```

Evaluating `g5(2)`

```
==> [5/x, 2/y] let val z = hc(x) in z+y end
==> [5/x, 2/y, n/z] z+y          (for some integer n)
==> n
```

10 months!

Similarly, `g5(3)` will take 10 months.

Defining `g` in place of `f` has not yet helped!

Staging

Recall the lambda expression for **g**:

```
fn x => fn y => let val z = hc(x) in z+y end
```

Let's move this computation here.

Horrible computation hidden underneath inner lambda.

- Move is valid because the computation does not depend on `y`.
- Such rearrangement of code — putting it in the "right spot" — we refer to as staging.

Staging

Let's stage properly:

```
fun h (x:int) : int -> int =
    let
       val z : int = horriblecomputation(x)
    in
       (fn y : int => z + y)
    end
```

Inner lambda free of `hc(x)`!

Now the type of h is `(* h : int -> int -> int *)`,

so we can define `val h5 : int -> int = h(5)`

and then evaluate `h5 (2)`

`h5 (3)`

How long do the 3 lines above take?

Staging

How long do the 3 lines above take?

Remember, the declaration of **h** created the following binding:

```
[(fn x => let val z = hc(x) in fn y => z+y end)/h]
```

In declaring `val h5 = h(5)`, one evaluates

```
[(fn x => let val z = hc(x) in fn y => z+y end)/h] h(5)
==> (fn x => let val z = hc(x) in fn y => z+y end) (5)
==> [5/x] let val z = hc(x) in fn y => z+y end
==> [5/x, n/z] fn y => z+y
```

(for some integer n)

10 months!

Staging

How long do the 3 lines above take?

Remember, the declaration of h created the following binding:

```
[(fn x => let val z = hc(x) in fn y => z+y end)/h]
```

In declaring `val h5 = h(5)`, one evaluates

```
[(fn x => let val z = hc(x) in fn y => z+y end)/h] h(5)
==> (fn x => let val z = hc(x) in fn y => z+y end) (5)
==> [5/x] let val z = hc(x) in fn y => z+y end
==> [5/x, n/z] fn y => z+y
```

(for some integer n)

10 months!

This is a lambda, and thus s a value!

Staging

How long do the 3 lines above take?

Remember, the declaration of h created the following binding:

```
[(fn x => let val z = hc(x) in fn y => z+y end)/h]
```

In declaring `val h5 = h(5)`, one evaluates

```
[(fn x => let val z = hc(x) in fn y => z+y end)/h] h(5)
==> (fn x => let val z = hc(x) in fn y => z+y end) (5)
==> [5/x] let val z = hc(x) in fn y => z+y end
==> [5/x, n/z] fn y => z+y
```

(for some integer n)

10 months!

This is the closure returned by `h(5)`.

Staging

We now have the following binding:

$$\left[\ \boxed{\begin{array}{l}\text{env}\\ \texttt{[5/x, n/z]}\\ \texttt{fn y => z+y}\end{array}}\ /\texttt{h5}\ \right]$$

Evaluating `h5(2)`

```
==> [5/x, n/z, 2/y] z+y
==> n'
```
(for some integer `n'`)

quick!

Similarly, `h5(3)` will be very quick.

→ Factoring `hc(x)` out of the inner lambda has improved efficiency!

Staging

Summary:

`f (5,2)`	> 10 months
`f (5,3)`	> 10 months
`val g5 = g(5)`	fast
`g5 (2)`	> 10 months
`g5 (3)`	> 10 months
`val h5 = h(5)`	> 10 months
`h5 (2)`	fast
`h5 (3)`	fast

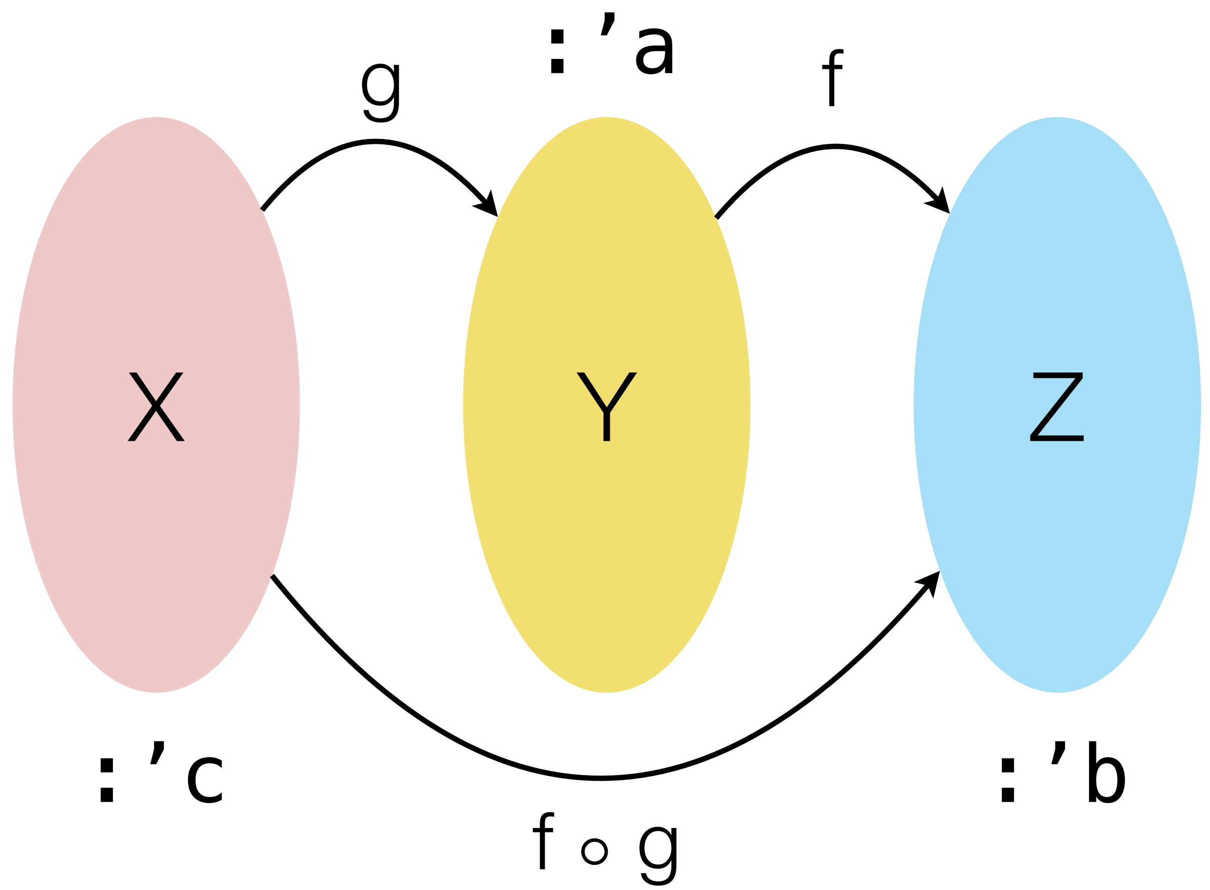
More combinators!

Recall function composition:

```
infix o
fun f o g = fn x => f(g(x))
```

Examples:

```
fun incr x = x + 1
fun double x = 2 * x
```



Combinators are functions that combine small pieces of code into larger pieces of code.

We will view combinators are higher-order functions that expect functions and return functions.

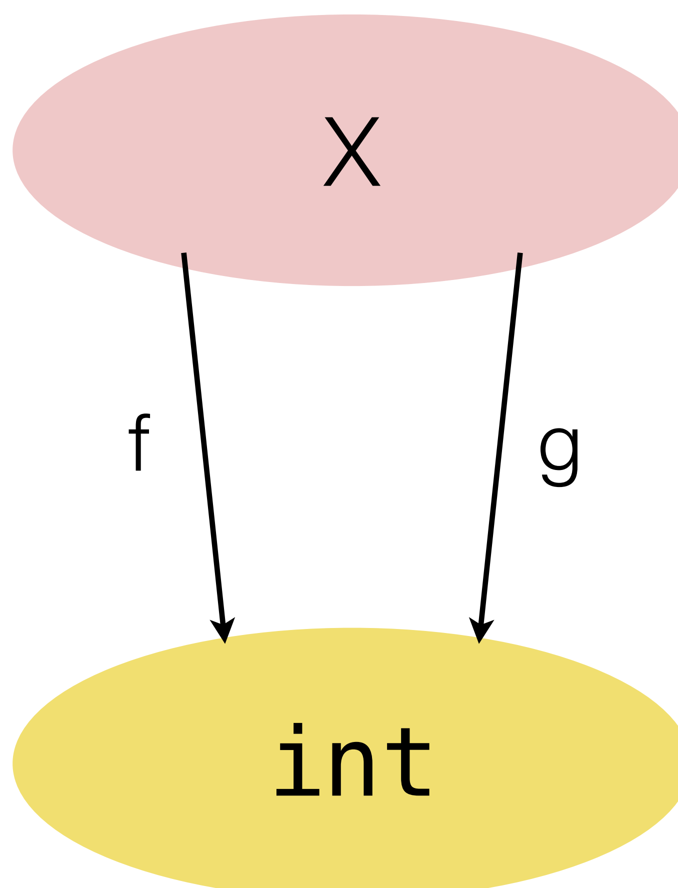
More combinators!

An abstract view of combinators:

Space (set):

Operations on elements:

Integer functions



Operations on functions:
`++`, `**`, `MIN`, …

combinators

Integers

Operations on integers:
`+`, `*`, `Int.min`, …

➡ **Combinators facilitate point-free programming.**

More combinators!

Combinators facilitate point-free programming.

In math, one may write the sum of two integer-valued functions in a **point-free** way:

$$f + g.$$

does not involve function arguments

More combinators!

Combinators facilitate point-free programming.

In math, one may write the sum of two integer-valued functions in a **point-free** way:

$$f + g.$$

If someone asks "what does that mean?", we would explain using a **point-specific** equation:

$$(f + g)(x) = f(x) + g(x).$$

combinator

integer addition

In SML, we define combinators using point-specific equations and use them for point-free programming.

Examples of combinators

Addition of functions:

```
infix ++
fun (f ++ g) x = f(x) + g(x)
```

Alternatively, we could first declare

`++(f,g) x = f(x) + g(x)` and subsequently write `infix ++`.

Other forms of declarations are possible, e.g.,

```
fun ++(f,g) = fn x => f(x) + g(x)
```

What is the type of `++`?

```
(* (op ++) : ('a -> int) * ('a -> int) -> 'a -> int *)
```

Examples of combinators

And more combinators:

```
fun square x = x * x
fun double x = 2 * x
```

We can combine these function values:

```
fun quadratic = square ++ double
```

Observe: `quadratic ≅ fn x => x * x + 2 * x`

I.e., `quadratic` represents the function $x^2 + 2x$.

quadratic (3) ↪ 15

See lecture notes for more examples!