PAUL G. ALLEN SCHOOL
OF COMPUTER SCIENCE & ENGINEERING

CSE341

Section 3

Standard-Library Docs, First-Class Functions, & More

Adapted from slides by Daniel Snitkovskiy, Nick Mooney, Nicholas Shahan, Patrick Larson, and Dan Grossman

Agenda

1. SML Docs
   - Standard Basis
2. Polymorphic Datatypes
3. First-Class Functions
   - Anonymous
   - Style Points
   - Higher-Order

Standard Basis Documentation

Online Documentation

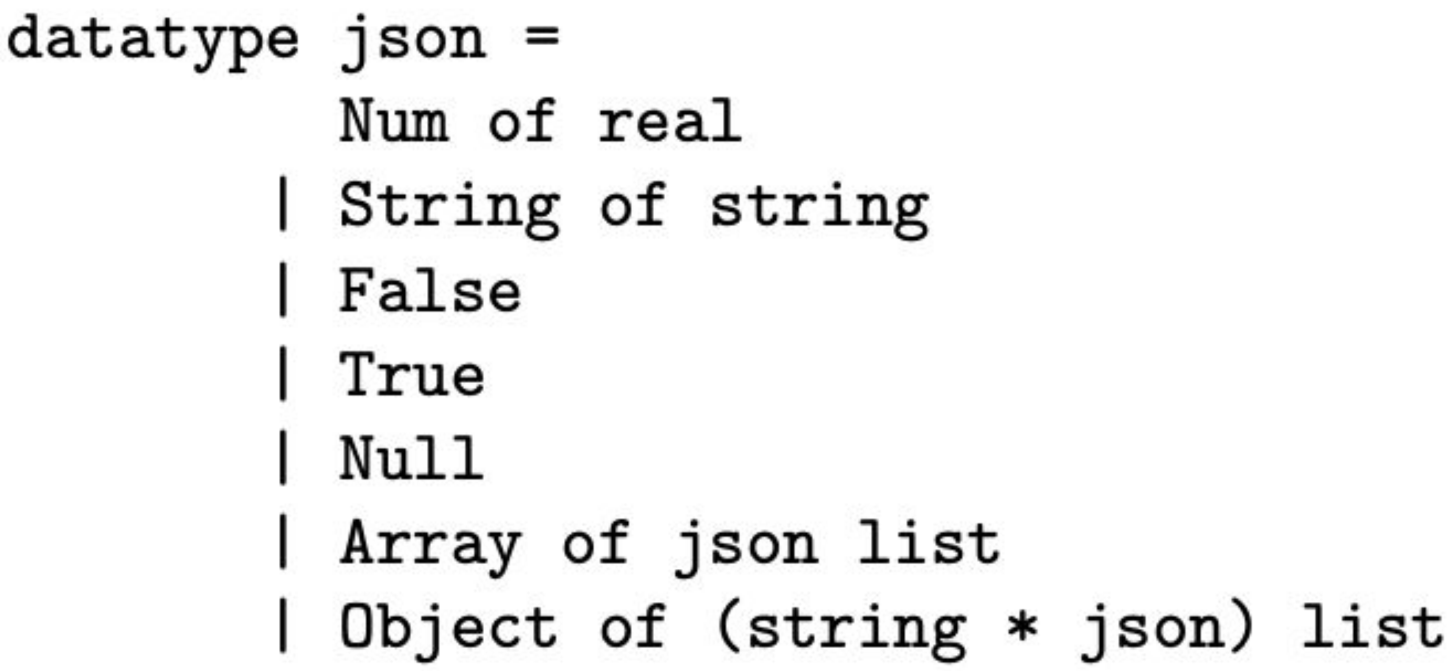
http://www.standardml.org/Basis/index.html

http://www.smlnj.org/doc/smlnj-lib/Manual/toc.html

Helpful Subset

Top-Level	http://www.standardml.org/Basis/top-level-chapter.html
List	http://www.standardml.org/Basis/list.html
ListPair	http://www.standardml.org/Basis/list-pair.html
Real	http://www.standardml.org/Basis/real.html
String	http://www.standardml.org/Basis/string.html

Is Json an equality type?



```
datatype json =
           Num of real
         | String of string
         | False
         | True
         | Null
         | Array of json list
         | Object of (string * json) list
```

Oh Shoot…. How to compare?

```
val x = String “abcd”; (* type json *)
val String y = x;

(* now y is equality type String *)
val test1 = y = “abcd”;
```

One more note

Real is not an equality type, you cannot compare them using “=”. Instead, you should….

```
val x = 3.14; (* real type *)
val epsilon = 0.00001;

val test = x - 3.14 < epsilon;
```

Polymorphic Datatypes

Suppose we want to create a Pair datatype

- A pair has two elements
- Both element must be of the same type

```
datatype ‘a pair = Pair of ‘a * ‘a
```

Now it’s your term

Suppose we want to create a tree datatype

- A node can be a leaf
- A node can be the root of a subtree
- Both leaf and non-leaf node contain some value, their value could be different

```
E.g.  Node 10
      Node (“abc”, Node 10, Node 20)
```

Now it’s your term

We solve this problem by having polymorphic datatypes:

```
datatype (‘a, ‘b) tree =
     Leaf of ‘a
   | Node of ‘b * (‘a, ‘b) tree * (‘a, ‘b) tree
```

Anonymous Functions

An Anonymous Function

```
fn pattern => expression
```

- An expression that creates a new function with no name.
- Usually used as an argument to a higher-order function.
- Almost equivalent to the following:

```
let fun name pattern = expression in name end
```

What’s the difference? What can you do with one that you can’t do with the other?

- The difference is that anonymous functions cannot be recursive!!!

Anonymous Functions

What's the difference between the following two bindings?

```
val name = fn pattern => expression;
fun name pattern = expression;
```

- Once again, the difference is recursion.
- However, excluding recursion, a `fun` binding could just be syntactic sugar for a `val` binding and an anonymous function.

Something is wrong….

What’s wrong with these expressions?

```
(if ex then true else false)
```

```
(fn xs => tl xs)
```

Unnecessary Function Wrapping

What's the difference between the following two expressions?

```
(fn xs => tl xs)          vs.     tl
```

STYLE POINTS!

- Other than style, these two expressions result in the exact same thing.
- However, one creates an unnecessary function to wrap `tl`.
- This is very similar to this style issue:

```
(if ex then true else false)       vs.     ex
```

Higher-Order Functions

Definition: A function that returns a function or takes a function as an argument.

- SML functions can be passed around like any other value.
- They can be passed as function arguments, returned, and even stored in data structures or variables.
- Generalized functions such as these are **very** pervasive in functional languages (and are starting to creep into more Object-Oriented ones too, e.g. Java)

Note: List.map, List.filter, and List.foldr/foldl are similarly defined in SML but use currying. We'll cover these later in the course.

Canonical Higher-Order Functions

map

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

What does the type tell is?

- What are the arguments?
- What is the return type?

map

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

  What does the type tell is?

  - What are the arguments?
  - What is the return type?

- `map` applies a function to every element of a list and return a list of the resulting values.
  - Example: `map (fn x => x*3, [1,2,3]) === [3,6,9]`

map

- `Sample: map (fn x => x*3, [1,2,3])`

```
[1, 2, 3]
```

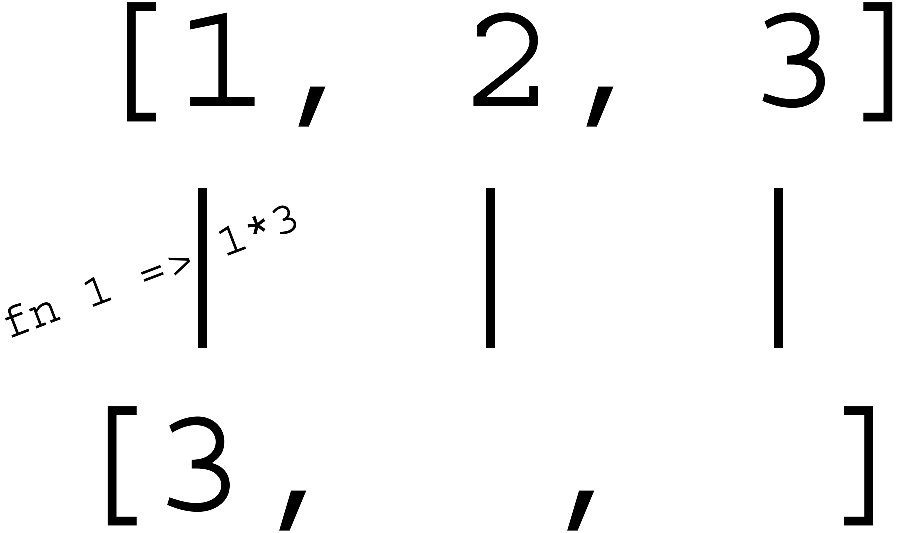
map

- `Sample: map (fn x => x*3, [1,2,3])`

```
[1,  2,  3]
 |   |   |
 |   |   |
[ ,   ,   ]
```

map

– Sample: `map (fn x => x*3, [1,2,3])`



[1, 2, 3]

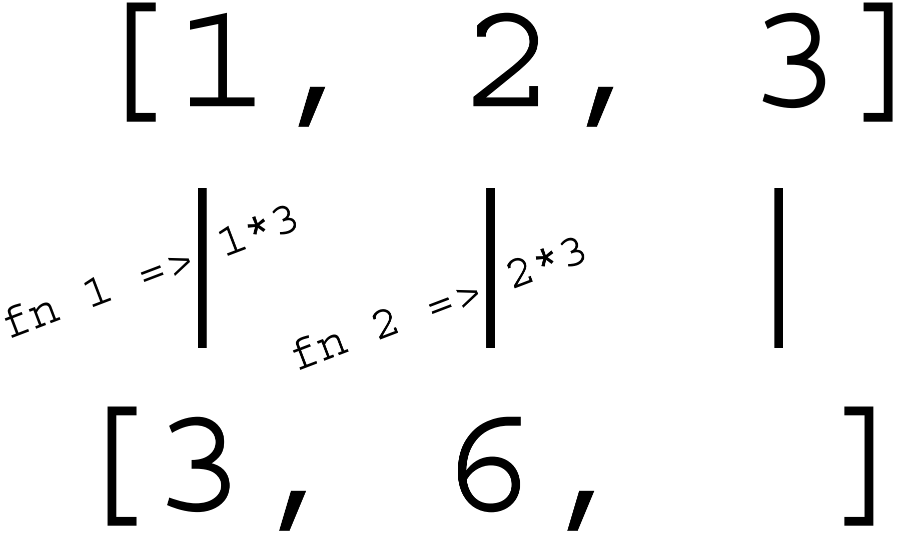
fn 1 =>

1*3

[3, ,]

map

- Sample: `map (fn x => x*3, [1,2,3])`



[1, 2, 3]

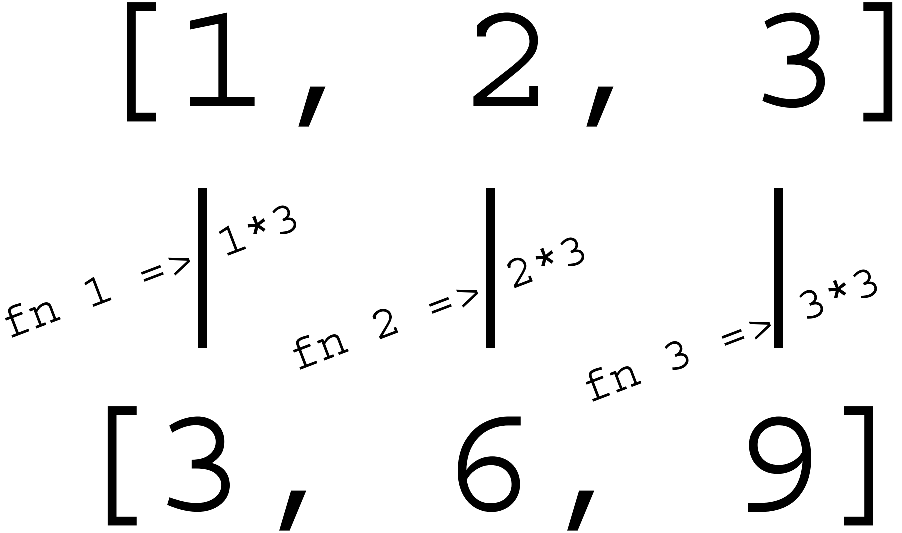
fn 1 => 1*3

fn 2 => 2*3

[3, 6,]

map

– Sample: `map (fn x => x*3, [1,2,3])`



[1, 2, 3]

fn 1 => 1*3

fn 2 => 2*3

fn 3 => 3*3

[3, 6, 9]

flat_map

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

Notice the difference?

flat_map

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

Notice the difference?

- `flat_map` applies a function which returns a list to every element of a list and return a concatenated list of the resulting lists.
  - Example:

```
flat_map (fn x => [x,~x], [1,2,3]) === [1,~1,2,~2,3,~3]
```

flat_map

- `Sample: flat_map (fn x => [x,~x], [1,2,3])`

```
[1, 2, 3]
```

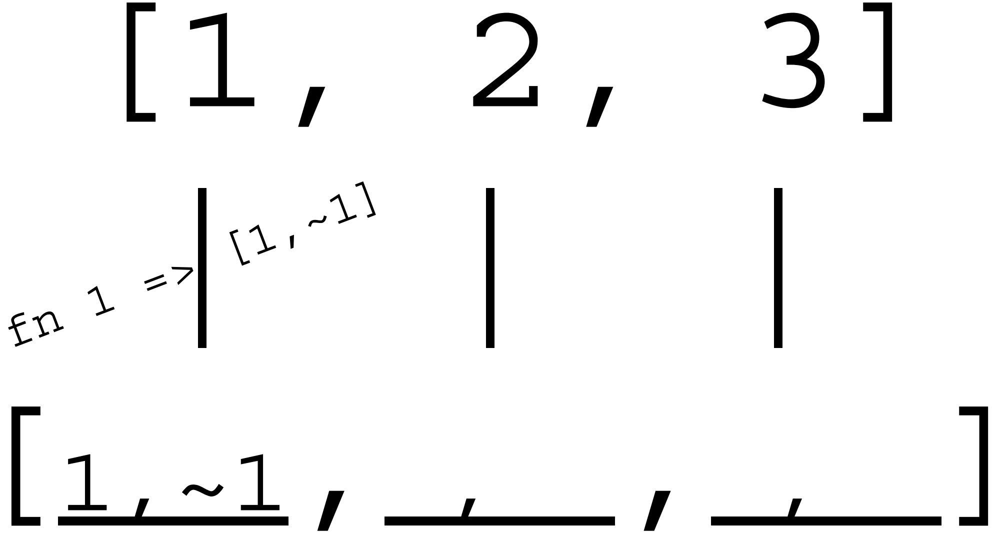
flat_map

- Sample: `flat_map (fn x => [x,~x], [1,2,3])`

```
[1,  2,  3]
 |   |   |
[_,__,_,__,_,__]
```

flat_map

- Sample: `flat_map (fn x => [x,~x], [1,2,3])`



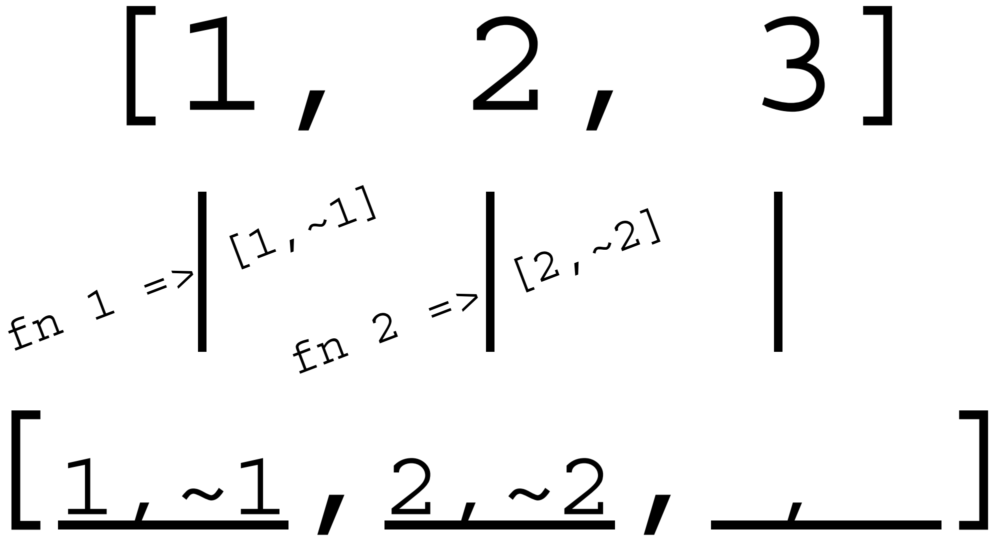
```
[1, 2, 3]
```

fn 1 => [1,~1]

```
[1,~1, ___,___, ___,___]
```

flat_map

– Sample: `flat_map (fn x => [x,~x], [1,2,3])`



[1, 2, 3]

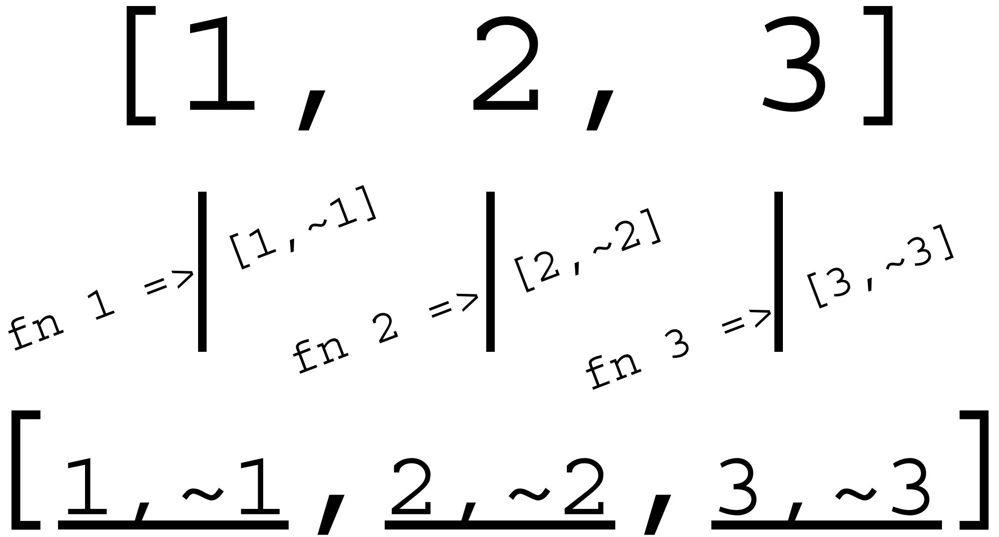
fn 1 => [1,~1]

fn 2 => [2,~2]

[1,~1,2,~2,__,___]

flat_map

- `Sample: flat_map (fn x => [x,~x], [1,2,3])`



[1, 2, 3]

fn 1 => [1,~1]

fn 2 => [2,~2]

fn 3 => [3,~3]

[1,~1,2,~2,3,~3]

filter

- `filter : ('a -> bool) * 'a list -> 'a list`

  What could be the type of this function?

  - What are the arguments?
  - What is the return type?

filter

- `filter : ('a -> bool) * 'a list -> 'a list`

  What could be the type of this function?
  - What are the arguments?
  - What is the return type?

- `filter` returns the list of elements from the original list that, when a predicate function is applied, result in true.
  - Example: `filter (fn x => x>2, [~5,3,2,5]) === [3,5]`

filter

- `Sample: filter (fn x => x > 1, [1,2,0,3])`

```
[1, 2, 0, 3]
```

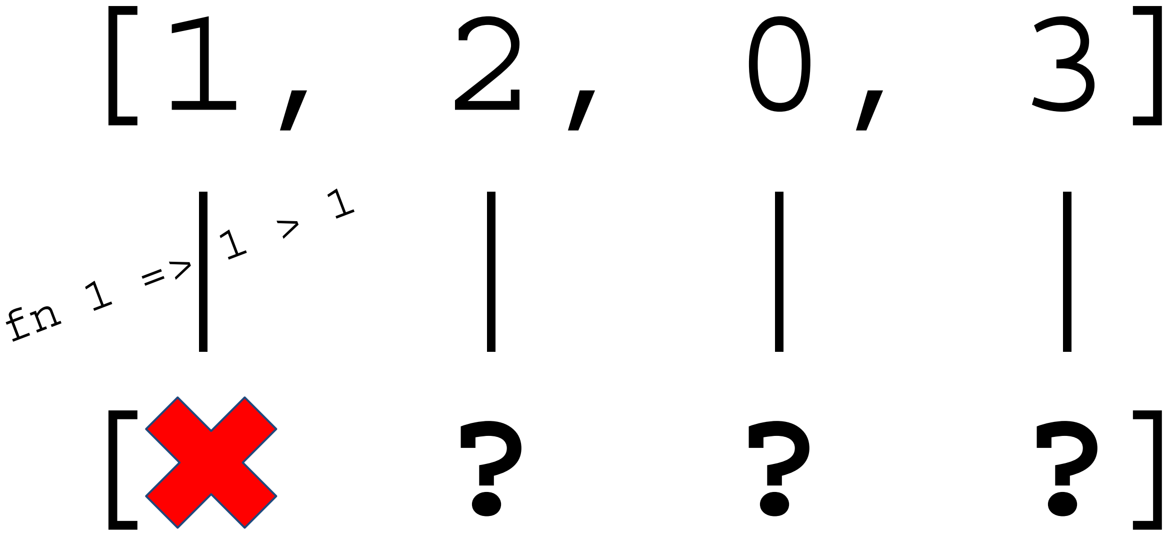
filter

- Sample: `filter (fn x => x > 1, [1,2,0,3])`

```
[1,  2,  0,  3]
 |   |   |   |
 |   |   |   |
[?   ?   ?   ?]
```

filter

- Sample: `filter (fn x => x > 1, [1,2,0,3])`



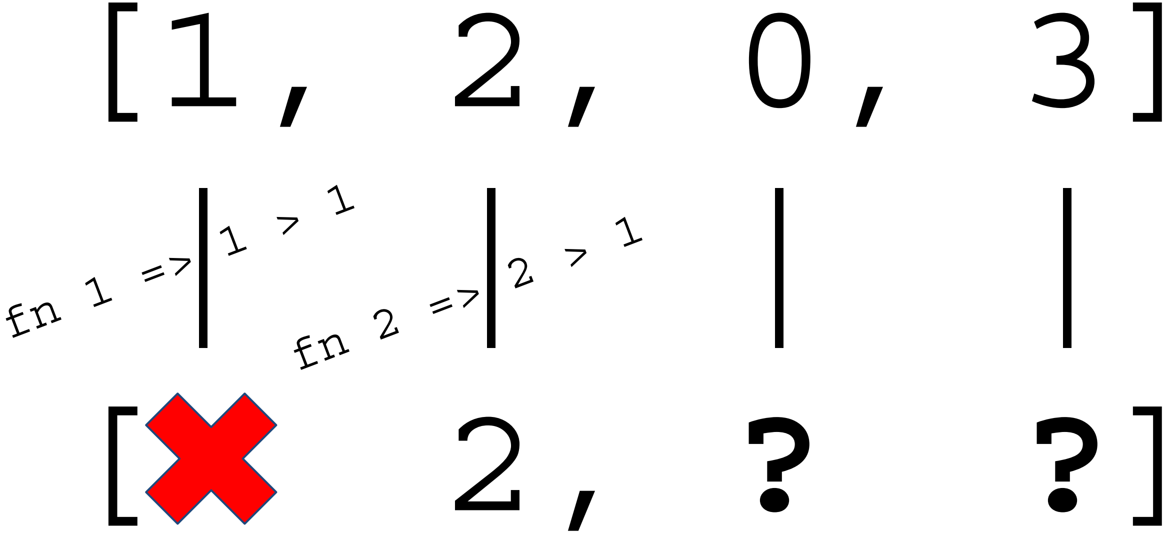
[1, 2, 0, 3]

fn 1 => 1 > 1

[✗ ? ? ?]

filter

- `Sample: filter (fn x => x > 1, [1,2,0,3])`



```
[1, 2, 0, 3]
```

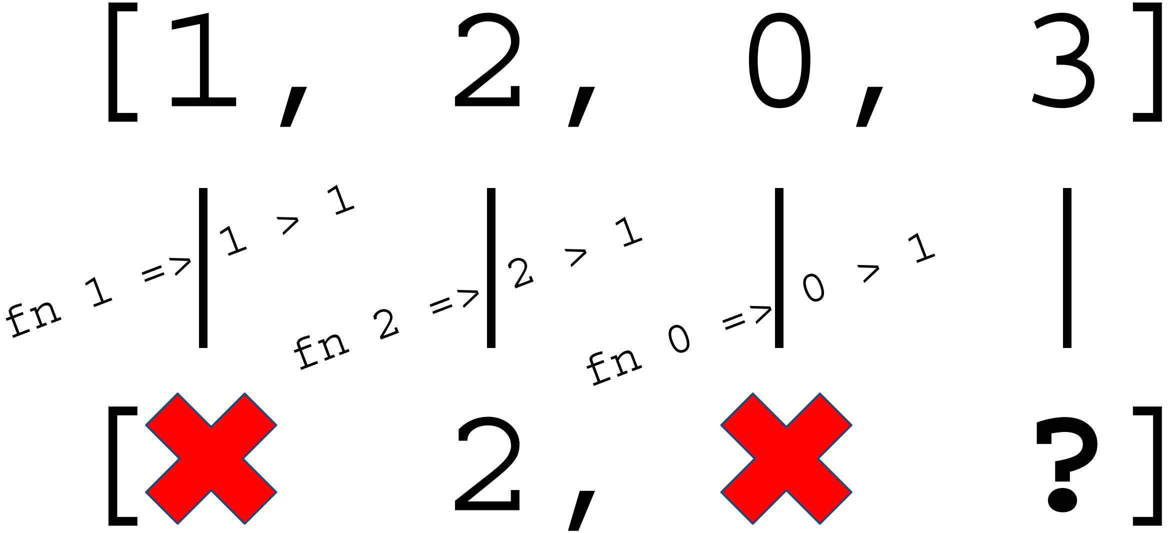
fn 1 => 1 > 1

fn 2 => 2 > 1

```
[✖ 2, ? ?]
```

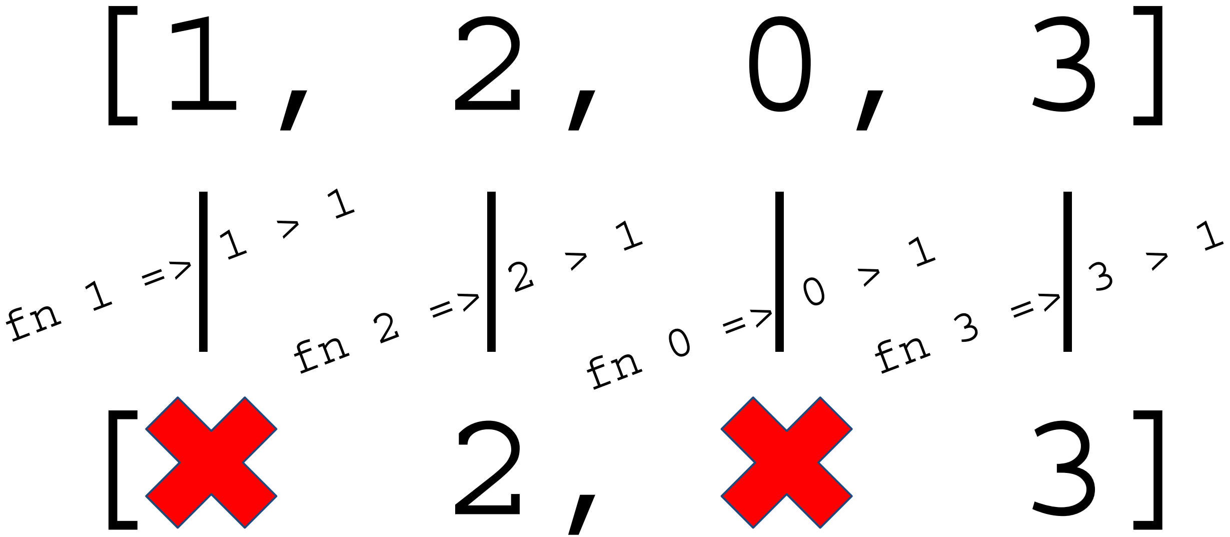
filter

- Sample: `filter (fn x => x > 1, [1,2,0,3])`



filter

– Sample: `filter (fn x => x > 1, [1,2,0,3])`



[1, 2, 0, 3]

fn 1 => 1 > 1

fn 2 => 2 > 1

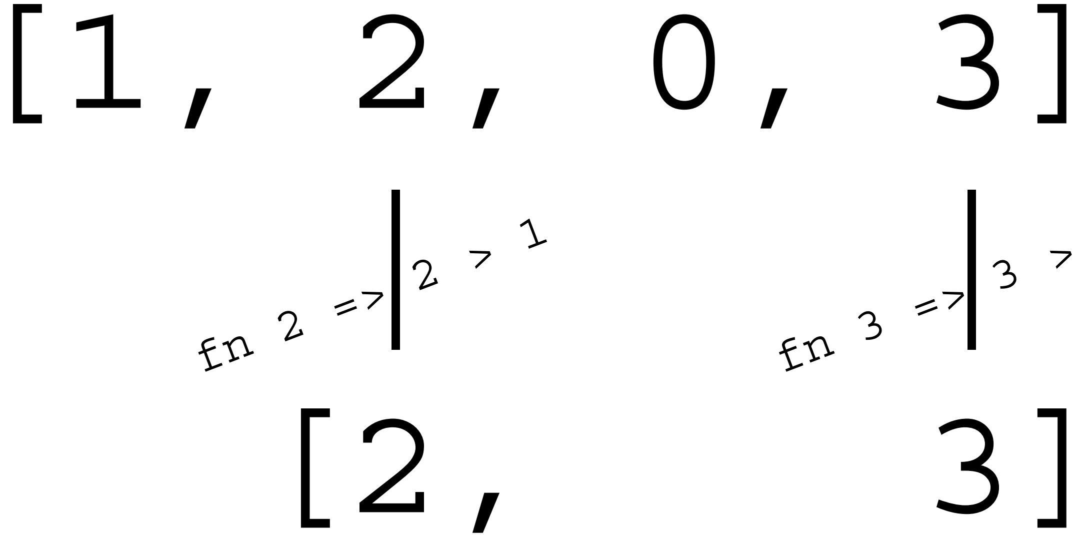
fn 0 => 0 > 1

fn 3 => 3 > 1

[2, 3]

filter

- `Sample: filter (fn x => x > 1, [1,2,0,3])`



[1, 2, 0, 3]

fn 2 => 2 > 1

fn 3 => 3 > 1

[2, 3]

fold

- `fold : ('a * 'b -> 'a) * 'a * 'b list -> 'a`
  - Returns a “thing” that is the accumulation of the first argument applied to the third arguments elements stored in the second argument.
  - Example: `fold((fn (a,b) => a + b), 0, [1,2,3]) === 6`

fold

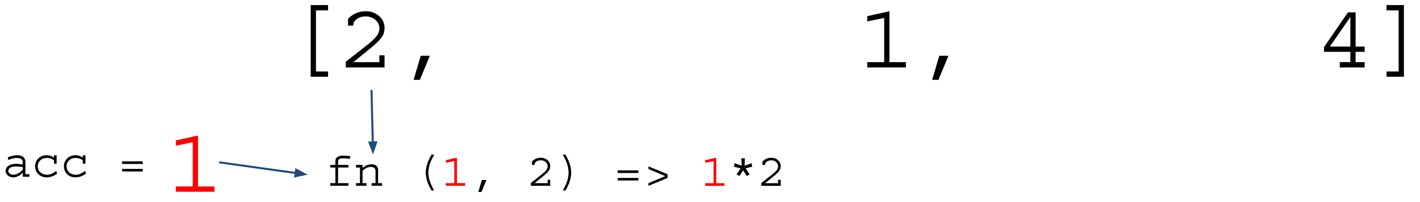
- Sample: `fold (fn (acc, x) => acc * x, 1, [2, 1, 4])`

```
[2,          1,          4]
```

```
acc = 1
```

fold

- Sample: `fold (fn (acc, x) => acc * x, 1, [2, 1, 4])`

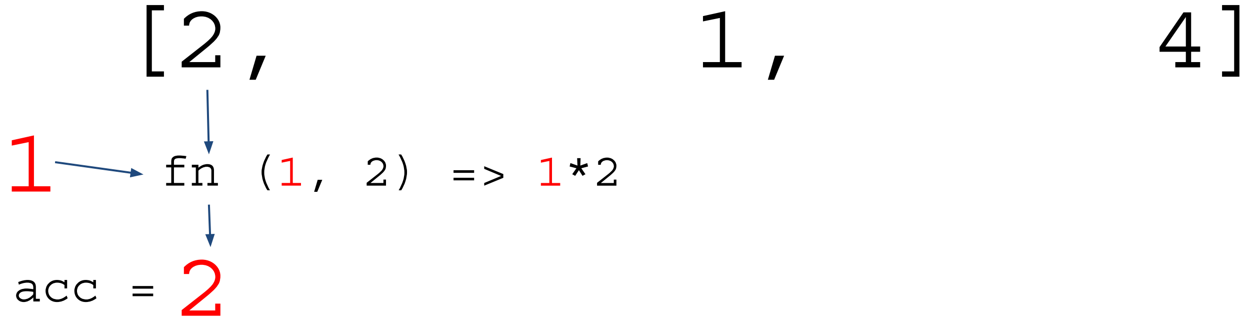


```
[2,                    1,                    4]

acc = 1    fn (1, 2) => 1*2
```

fold

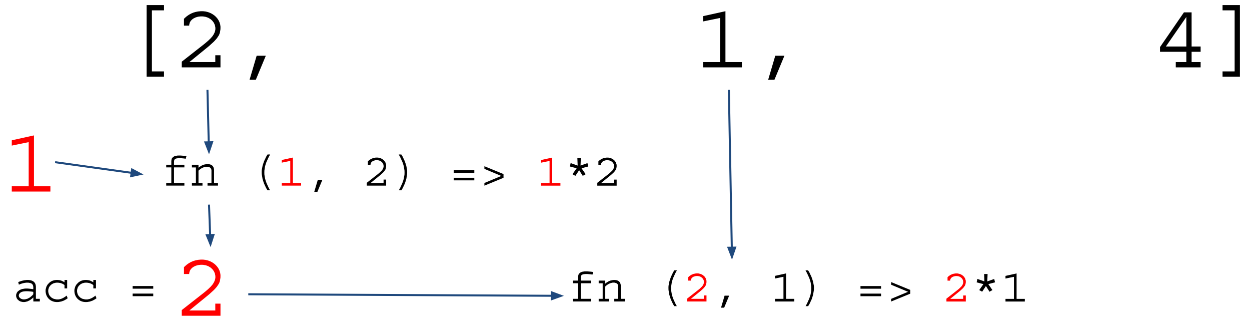
- Sample: `fold (fn (acc, x) => acc * x, 1, [2, 1, 4])`



```
[2,   1,   4]
1
fn (1, 2) => 1*2
acc = 2
```

fold

- Sample: fold (fn (acc, x) => acc * x, 1, [2, 1, 4])



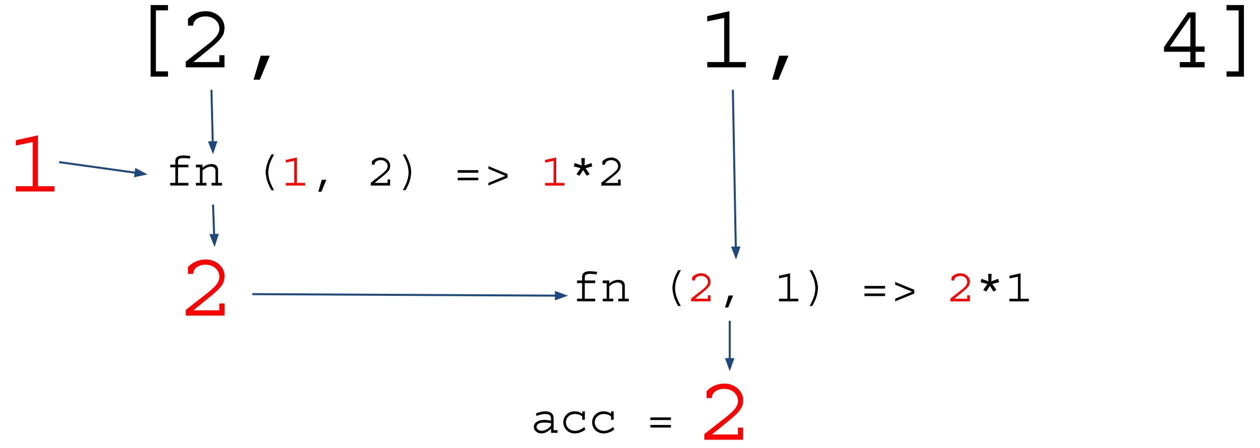
[2, 1, 4]

1 fn (1, 2) => 1*2

acc = 2 fn (2, 1) => 2*1

fold

- Sample: `fold (fn (acc, x) => acc * x, 1, [2, 1, 4])`



[2, 1, 4]

1

fn (1, 2) => 1*2

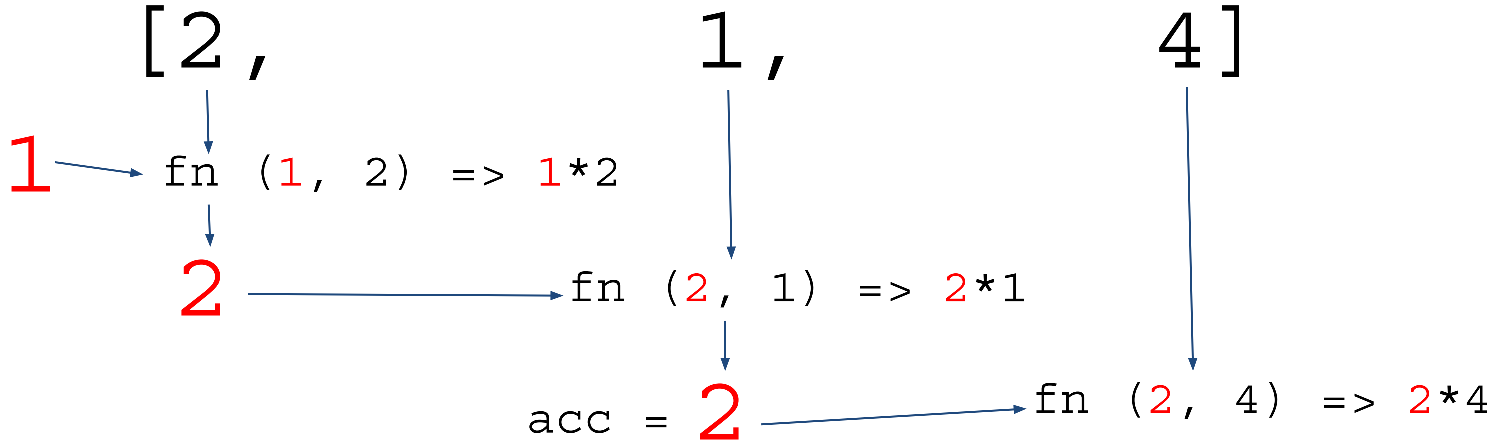
2

fn (2, 1) => 2*1

acc = 2

fold

- Sample: fold (fn (acc, x) => acc * x, 1, [2, 1, 4])



[2, 1, 4]

1

fn (1, 2) => 1*2

2

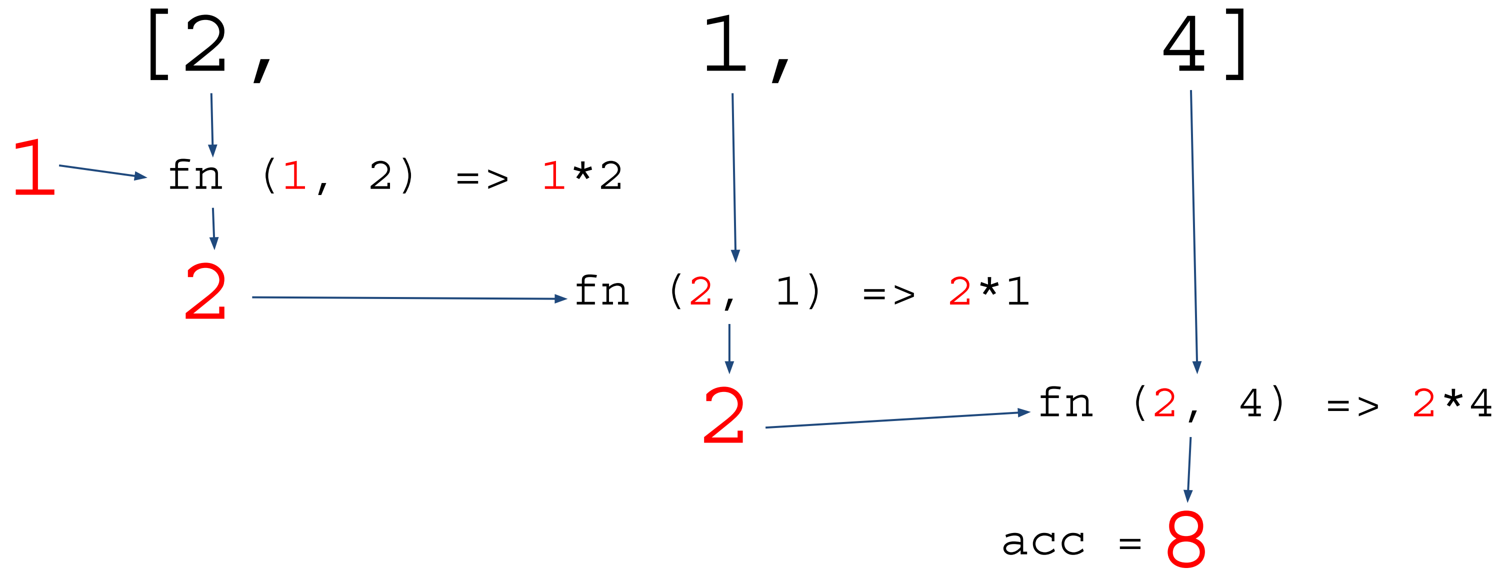
fn (2, 1) => 2*1

acc = 2

fn (2, 4) => 2*4

fold

- Sample: `fold (fn (acc, x) => acc * x, 1, [2, 1, 4])`



```
[2,                    1,                  4]
1 fn (1, 2) => 1*2
2 fn (2, 1) => 2*1
2 fn (2, 4) => 2*4
acc = 8
```