IT UNIVERSITY OF COPENHAGEN

Functional Programming

Algebraic datatypes and collections

Jesper Bengtson

Credit where credit is due

These slides are based on original slides by Michael R. Hansen at DTU. Thank you!

The original slides have been used for a functional programming course at DTU

Last week

We covered a substantial part of F#

- Higher-order functions
- Function composition
- Piping commands
- More lists

Folds (loops)

$$\texttt{fold f acc [}x_1\texttt{; }x_2\texttt{; …; }x_n\texttt{]}$$

returns

$$\texttt{f (… (f (f acc }x_1\texttt{) }x_2\texttt{) … }x_{n-1}\texttt{) }x_n$$

```
let acc = fold f init xs
```

vs.

```
acc = init;
for(i = 0; i < xs.Length; i++)
{
    acc = f (acc, xs[i]);
}
```

Folds (loops)

```
let rec fold f acc =
    function
    | []       -> acc
    | x :: xs -> fold f (f acc x) xs
```

```
fold : (‘b -> ‘a –> ‘b) -> ‘b -> ‘a list -> ‘b
```

Folds (loops)

```
let rec fold f acc =
    function
    | []        -> acc
    | x :: xs -> fold f (f acc x) xs
```

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

```
fold (fun acc x -> acc + x) 0 [1; 2; 3; 4] =
```

```
fold (+) 0 [1; 2; 3; 4] =
```

(+) ((+) ((+) (((+) 0 1) 2)) 3) 4 = 0 + 1 + 2 + 3 + 4 = 10

Folds (loops)

```
let rec fold f acc =
    function
    | []         -> acc
    | x :: xs -> fold f (f acc x) xs
```

```
fold : (‘b -> ‘a —> ‘b) -> ‘b -> ‘a list -> ‘b
```

```
fold (fun acc x -> acc * x] 1 [1; 2; 3; 4] =
```

```
fold (*) 1 [1; 2; 3; 4] =
```

(*) ((*) ((*) (((*) 1 1) 2)) 3) 4 = 1 * 1 * 2 * 3 * 4 = 24

Folds (loops)

```
let ??? lst =
    snd (fold (fun (y, acc) z -> (z, y <= z && acc))
              (List.head lst, true)
              (List.tail lst))
```

Folds (loops)

```
let ??? lst =
    snd (fold (fun (y, acc) z -> (z, y <= z && acc))
              (List.head lst, true)
              (List.tail lst))
let ??? lst =
    lst |>
    List.tail |>
    fold (fun (y, acc) z -> (z, y <= z && acc))
         (List.head lst, true) |>
    snd
```

Folds (loops)

```
let ??? lst =
    snd (fold (fun (y, acc) z -> (z, y <= z && acc))
              (List.head lst, true)
              (List.tail lst))
```

```
let ??? ls
    lst |>
    List.t
    fold (
        (List.head lst, true) |>
    snd
```

List.pairwise $[x_1; \ldots; x_n]$ =
$[(x_1, x_2); (x_2, x_3); \ldots; (x_{n-1}, x_n)]$

```
let ??? lst =
    lst
    List.pairwise |>
    List.forall (fun (x, y) -> x <= y)
```

Folds (loops)

```
let ??? lst =
    snd (fold (fun (y, acc) z -> (z, y <= z && acc))
              (List.head lst, true)
              (List.tail lst))
let ??? lst =
    lst |>
    List.tail |>
    fold (fun (y, acc) z -> (z, y <= z && acc))
         (List.head lst, true) |>
    snd
let ??? lst =
    lst
    List.pairwise |>
    List.forall (fun (x, y) -> x <= y)
```

This week

- Inductively defined datatypes
- Expression trees
- Collections (sets and maps)
- Data representation
- Live coding

Questions?

Inductively defined types

- … or algebraic datatypes
- … or disjoint unions

Allow us to concisely say how members of types are created

Inductively defined types

```
type month =
| January | February | March | April
| May | June | July | August
| September | October | November
| December
```

```
type weekDay =
| Monday | Tuesday | Wednesday
| Thursday | Friday |
| Saturday | Sunday
```

Inductively defined types

```
type month =
| January | February | March | April
| May
| Sep
| Dec
```

Observation 1

We use 'type', not 'let'

```
type weekDay =
| Monday | Tuesday | Wednesday
| Thursday | Friday |
| Saturday | Sunday
```

Inductively defined types

```
type month =
| January | February | March | April
| May
| Sep
| Dec
```

Observation 2

Behave similarly to enum-types in Java

```
type weekDay =
| Monday | Tuesday | Wednesday
| Thursday | Friday |
| Saturday | Sunday
```

Programming

```
type month =
| January | February | March | April
| May | June | July | August
| September | October | November
| December

 let numberOfDays =
 function
 | January | March | May | July
 | August | October | December -> 31
 | February -> 28
 | _ -> 30
```

Programming

```
type month =
|                                                     ..l
|
|
|
| December
```

We can pattern match on algebraic types

```
let numberOfDays =
function
| January | March | May | July
| August | October | December -> 31
| February -> 28
| _ -> 30
```

Programming

```
type weekDay =
| Monday | Tuesday | Wednesday
| Thursday | Friday |
| Saturday | Sunday

let nextWeekday =
  function
  | Monday -> Tuesday
  | Tuesday -> Wednesday
  | Wednesday -> Thursday
  | Thursday -> Friday
  | _ -> Monday
```

Arguments

Algebraic types can take arguments

```
type shape =
| Circ of float            (* radius *)
| Rect of float * float (* sides *)

let area =
  function
  | Circ r -> System.Math.PI * r * r
  | Rect (w, h) -> w * h
```

The option type

Options are used to encode
partial functions

```
type 'a option =
| None         (* No result *)
| Some of 'a (* result *)
```

You can think of options as terms that can be set to null (but nice, and type-safe, and does not cause as many bugs)

The option type

Some useful functions

```
Option.get : ‘a option -> ‘a

Option.map : (‘a -> ‘b) -> ‘a option -> ‘b option

Option.defaultValue :
        ‘a -> ‘a option -> ‘a
```

The option type

Some useful functions

```
Option.get : ‘a option -> ‘a

Option.m                                             ’b option

Option.c
        ‘a -> ‘a option -> ‘a
```

Option.get None throws an exception, similarly to taking the head of an empty list

Recursive

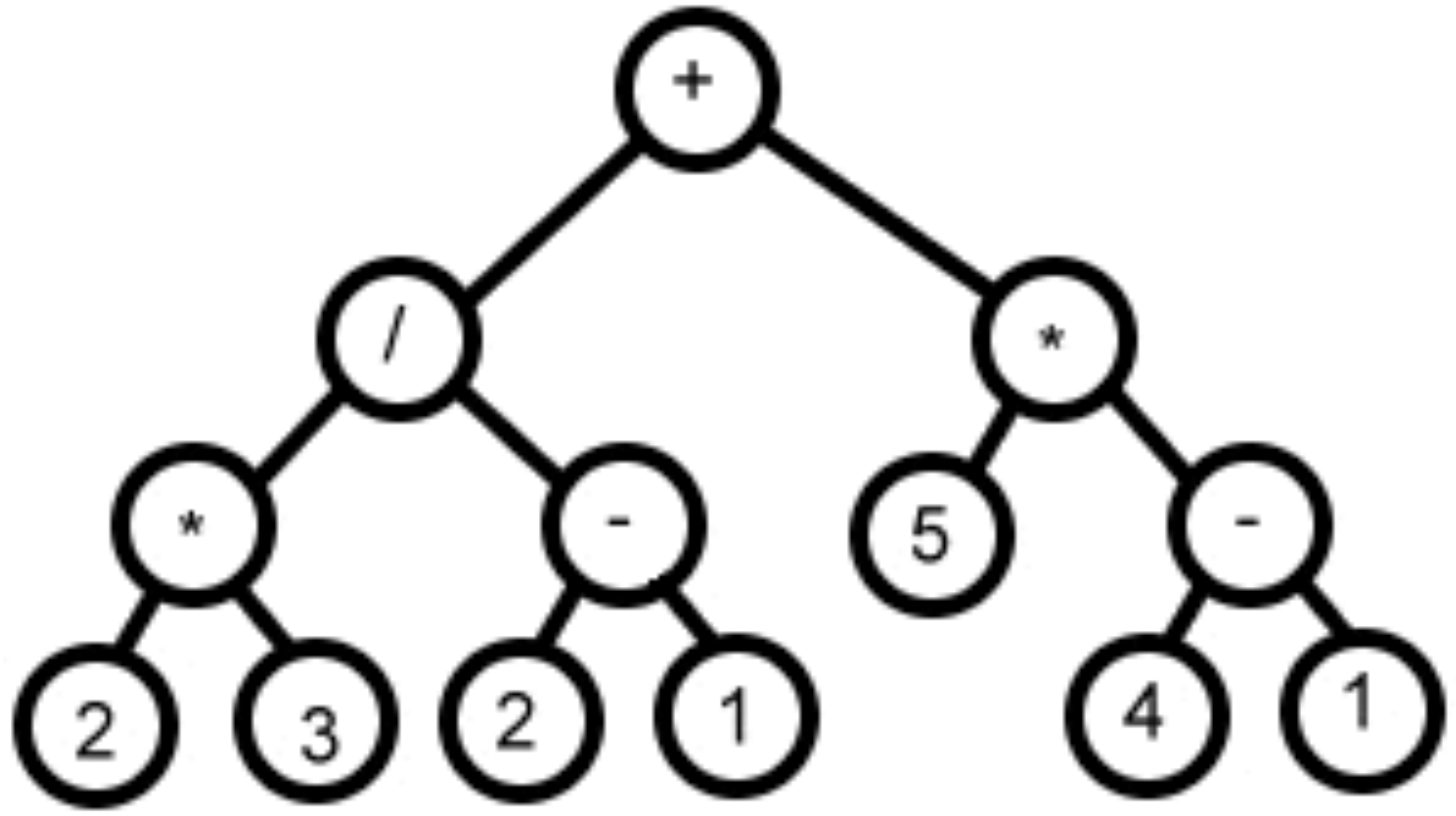
Algebraic types can be recursive and polymorphic

```
type ‘a myList =
| Nil                          (* empty list *)
| App of ‘a * ‘a myList (* cons *)

let rec length =
  function
  | Nil -> 0
  | App (_, lst) -> 1 + length lst
```

Expression trees

- Expression trees are heavily used by compilers
- Nodes contain operators
- Leaves contain values



Expression tree for 2*3/(2-1)+5*(4-1)

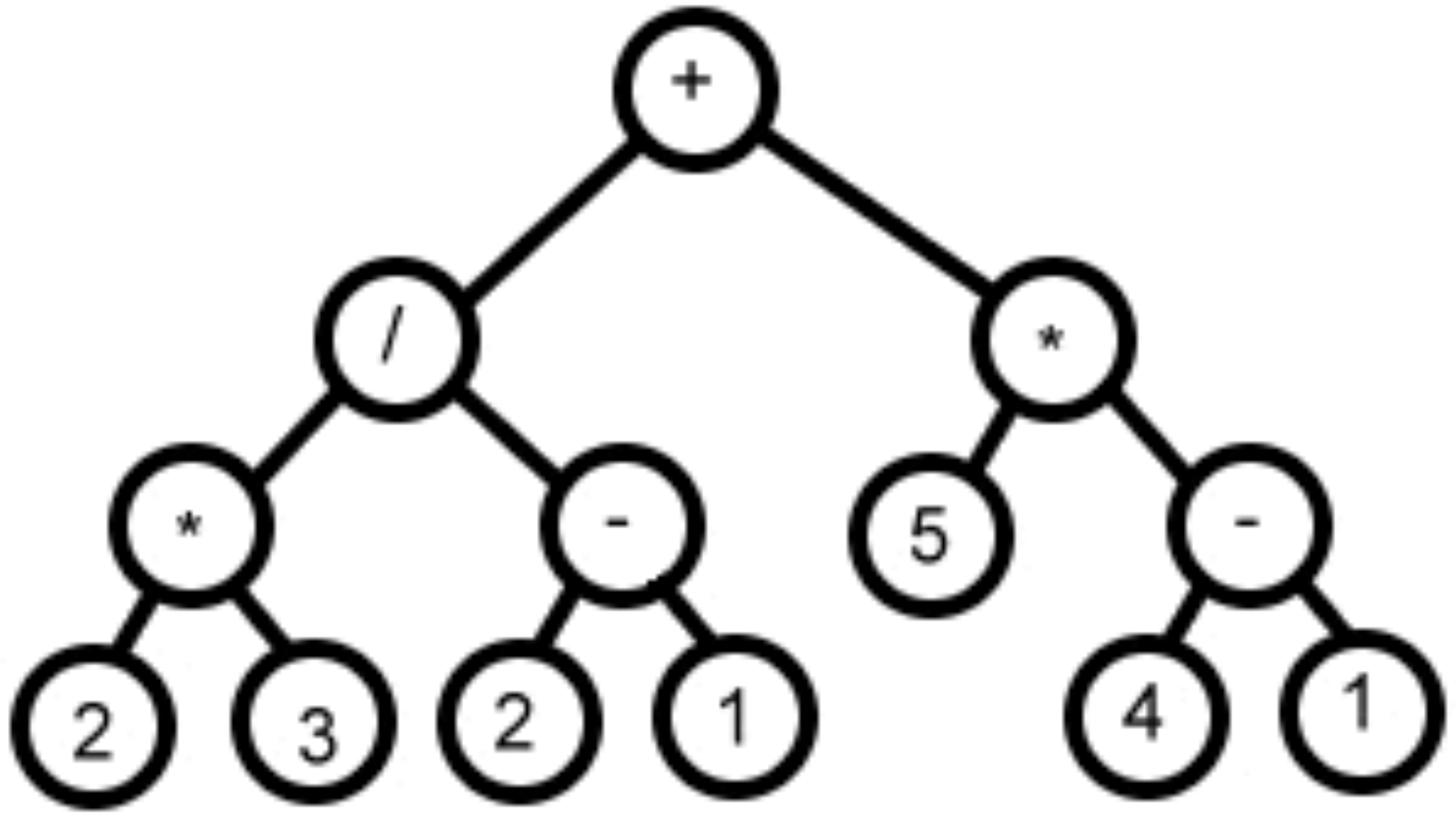
Expression trees

Expression trees are really easy to code

```
type term =
| Const of int
| Add of term * term
| Sub of term * term
| Mul of term * term
| Div of term * term
```

Expression trees

```
type term =
  | Const of int
  | Add of term * term
  | Sub of term * term
  | Mul of term * term
  | Div of term * term
```



Expression tree for 2*3/(2-1)+5*(4-1)

```
    Add
      (Div (Mul (Const 2, Const 3)
                ,Sub (Const 2, Const 1))
      ,Mul (Const 5
                ,Sub (Const 4, Const 1))
```

Exression trees

... and really easy to recurse over

```
let rec show : term -> string =
    function
    | Const f -> sprintf "%n" f
    | Add (t1, t2) ->
       "(" @ show t1 @ " + " @ show t2 @ ")"
    | Sub (t1, t2) ->
       "(" @ show t1 @ " - " @ show t2 @ ")"
    | Mul (t1, t2) ->
       "(" @ show t1 @ " * " @ show t2 @ ")"
    | Div (t1, t2) ->
       "(" @ show t1 @ " / " @ show t2 @ ")"
```

Expression Trees

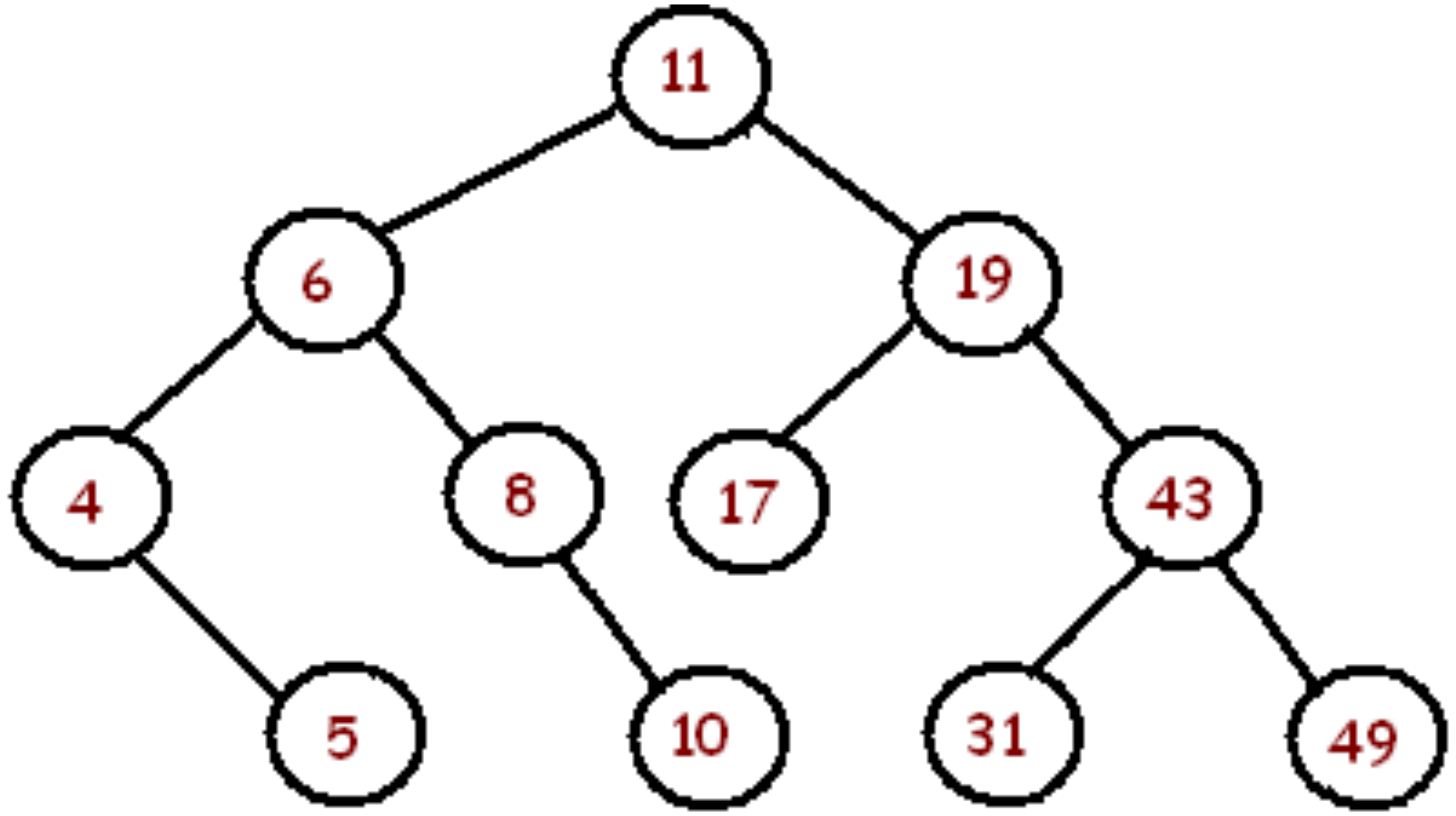
Recall the expression tree from before:

```
let t : term =
    Add
      (Div (Mul (Const 2, Const 3)
                ,Sub (Const 2, Const 1))
      ,Mul (Const 5
                ,Sub (Const 4, Const 1))
```

```
> show t;;
val it : string =
   "(((2 * 3) / (2 - 1)) + (5 * (4 - 1)))";;
```

Binary search trees

- All elements in the left subtree are smaller than or equal to the root
- All elements in the right subtree are greater than the root



Let’s code

Collections

F# has support for all collections from the .NET framework

- Lists
- Sets
- Maps
- Hash tables
- …

Sets

- Represents mathematical sets
- Intersection, union, …
- Not mutable
- Unordered
- Can only store values of comparison type
- Created inline by set [a1; a2; …; an] where all duplicates are removed

Sets (some functions)

```
Set.empty : Set<'a>
```

Sets (some functions)

```
Set.empty : Set<'a>
Set.singleton : 'a -> Set<'a>
```

Sets (some functions)

```
Set.empty : Set<'a>
Set.singleton : 'a -> Set<'a>
Set.union : Set<'a> -> Set<'a> ->
            Set<'a>
```

Sets (some functions)

```
Set.empty : Set<'a>
Set.singleton : 'a -> Set<'a>
Set.union : Set<'a> -> Set<'a> ->
                Set<'a>
Set.intersect :
      Set<'a> -> Set<'a> -> Set<'a>
```

Sets (some functions)

```
Set.empty : Set<'a>
Set.singleton : 'a -> Set<'a>
Set.union : Set<'a> -> Set<'a> ->
                Set<'a>
Set.intersect :
        Set<'a> -> Set<'a> -> Set<'a>
Set.map : ('a -> 'b) -> Set<'a> ->
              Set<'b>
```

Sets (some functions)

```
Set.empty : Set<'a>
Set.singleton : 'a -> Set<'a>
Set.union : Set<'a> -> Set<'a> ->
                Set<'a>
Set.intersect :
     Set<'a> -> Set<'a> -> Set<'a>
Set.map : ('a -> 'b) -> Set<'a> ->
          Set<'b>
Set.filter : ('a -> bool) ->
                Set<'a> -> Set<'a>
```

Sets (some functions)

```
Set.empty : Set<'a>
Set.singleton : 'a -> Set<'a>
Set.union : Set<'a> -> Set<'a> ->
                Set<'a>
Set.intersect :
     Set<'a> -> Set<'a> -> Set<'a>
Set.map : ('a -> 'b) -> Set<'a> ->
            Set<'b>
Set.filter : ('a -> bool) ->
                Set<'a> -> Set<'a>
Set.fold : ('b -> 'a -> 'b) ->
               'b -> Set<'a> -> 'b
```

Maps

- Represents mathematical maps
- add, lookup, …
- Not mutable
- Unordered
- Can only have keys of comparison type

Maps (some functions)

```
Map.empty : Map<‘a, ‘b>
```

Maps (some functions)

```
Map.empty : Map<'a, 'b>
Map.add : 'a -> 'b -> Map<'a, 'b> -> Map<'a, 'b>
```

Maps (some functions)

```
Map.empty : Map<'a, 'b>
Map.add : 'a -> 'b -> Map<'a, 'b> -> Map<'a, 'b>
Map.find : 'a -> Map<'a, 'b> -> 'b
```

Maps (some functions)

```
Map.empty : Map<'a, 'b>
Map.add : 'a -> 'b -> Map<'a, 'b> -> Map<'a, 'b>
Map.find : 'a -> Map<'a, 'b> -> 'b
Map.tryFind : 'a -> Map<'a, 'b> -> 'b option
```

Maps (some functions)

```
Map.empty : Map<‘a, ‘b>
Map.add : ‘a -> ‘b -> Map<‘a, ‘b> -> Map<‘a, ‘b>
Map.find : ‘a -> Map<‘a, ‘b> -> ‘b
Map.tryFind : ‘a -> Map<‘a, ‘b> -> ‘b option

Map.map : (‘a -> ‘b -> ‘c) ->
          Map<‘a, ‘b> -> Map<‘a, ‘c>
```

Maps (some functions)

```
Map.empty : Map<'a, 'b>
Map.add : 'a -> 'b -> Map<'a, 'b> -> Map<'a, 'b>
Map.find : 'a -> Map<'a, 'b> -> 'b
Map.tryFind : 'a -> Map<'a, 'b> -> 'b option

Map.map : ('a -> 'b -> 'c) ->
          Map<'a, 'b> -> Map<'a, 'c>
Map.filter : ('a -> 'b -> bool) ->
       Map<'a, 'b> -> Map<'a, 'b>
```

Maps (some functions)

```
Map.empty : Map<‘a, ‘b>
Map.add : ‘a -> ‘b -> Map<‘a, ‘b> -> Map<‘a, ‘b>
Map.find : ‘a -> Map<‘a, ‘b> -> ‘b
Map.tryFind : ‘a -> Map<‘a, ‘b> -> ‘b option

Map.map : (‘a -> ‘b -> ‘c) ->
            Map<‘a, ‘b> -> Map<‘a, ‘c>
Map.filter : (‘a -> ‘b -> bool) ->
         Map<‘a, ‘b> -> Map<‘a, ‘b>
Map.fold : (‘c -> ‘a -> ‘b -> ‘c) ->
         ‘c -> Map<‘a, ‘b > -> ‘c
```

Higher-order functions

For lists you can potentially get away without using higher-order functions by using recursion

For sets and maps you still can (translate them to lists, work on the lists and then translate them back) but this is a **really** bad idea.

Practice using higher-order functions :)

Program interpreters

Functional languages are great for working directly on abstract syntax trees

An imperative language

```
type aExp = …
type bExp = …
```

Assuming we have types for arithmetic and boolean expressions

The abstract syntax tree of our language is defined like this

```
type stm =
  | Skip
  | Ass of string * aExp
  | Seq of stm * stm
  | ITE of bExp * stm * stm
  | While of bExp * stm
```

State

In order to keep track of program state we need a mapping from program variables to values

State

In order to keep track of program state we need a mapping from program variables to values

```
type state = Map<string, int>
```

Arithmetic expressions

```
type aExp =
  | N of int
  | V of string
  | Add of (aExp * aExp)
  | Mul of (aExp * aExp)
  | Sub of (aExp * aExp)
```

Arithmetic expressions

Arithmetic expressions can be evaluated in the context of a state

```
evalA : aExp -> state -> int
```

```
type aExp =
| N of int
| V of string
| Add of (aExp * aExp)
| Mul of (aExp * aExp)
| Sub of (aExp * aExp)
```

```
let rec evalA arith st =
  match arith with
  | N n -> n
  | V v -> Map.find v st
  | Add(a, b) ->
    let va = evalA a st
    let vb = evalA b st
    va + vb
…
```

State (some functions)

```
type state = Map<string, int>


let binop : ('a -> 'b -> 'c)
            -> (state -> 'a)
            -> (state -> 'b)
            -> state -> 'c

  = fun f x y s -> f (x s) (y s)
```

Arithmetic expressions

Arithmetic expressions can be evaluated in the context of a state

```
evalA : aExp -> state -> int
```

```
let binop f x y s = f (x s) (y s)
let rec evalA =
  function
   | N n -> fun _ -> n
   | V v -> fun s -> Map.find v s
   | Add(a, b) -> binop (+) (evalA a) (evalA b)
   | Sub(a, b) -> binop (-) (evalA a) (evalA b)
   | Mul(a, b) -> binop (*) (evalA a) (evalA b)
```

Arithmetic expressions

Arithmetic expressions can be evaluated in the context of a state

```
evalA : aExp -> state -> int
```

```
let binop f x y s = f (x s) (y s)
let rec evalA =
 function
  | N n -> fun _ -> n
  | V v -> Map.find v
  | Add(a, b) -> binop (+) (evalA a) (evalA b)
  | Sub(a, b) -> binop (-) (evalA a) (evalA b)
  | Mul(a, b) -> binop (*) (evalA a) (evalA b)
```

Boolean expressions

```
type bExp =
| TT
| FF
| Eq of (aExp * aExp)
| Lt of (aExp * aExp)
| Neg of bExp
| Con of (bExp * bExp)
```

Boolean expressions

Boolean expressions can be evaluated in the context of a state

```
evalB : bExp -> state -> bool
```

```
let rec evalB =
   <Assignment for this week>
```

Evaluating a program

```
type stm =
  | Skip
  | Ass of string * aExp
  | Seq of stm * stm
  | ITE of bExp * stm * stm
  | While of bExp * stm
```

Programs evaluation is done by updating the state

```
evalS : stm -> state -> state
```

```
let rec evalS =
   <Assignment for this week>
```

Questions?