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)

```
fold f acc [x1; x2; …; xn]
```

returns

```
f (… (f (f acc x1) x2) … xn-1) xn
```

```
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
```

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

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)
```

Function composition

Functions can be composed
In mathematics we write

$$(f \circ g)(x) = f(g(x))$$

In F# we write

```
g >> f
```

Function composition

For example

Assume $f(x) = x * 3$ and $g(y) = y + y$

$$(f \circ g)(z) = (z + z) * 3$$

In F# we can write

```
let h = (fun y -> y+y) >>
        (fun x -> x*3)
```

```
h 0  = (0 + 0) * 3 = 0
h 4  = (4 + 4) * 3 = 24
h 20 = (20 + 20) * 3 = 120
```

Function composition

Functions can be composed
In mathematics we write

$$(f \circ g)(x) = f(g(x))$$

In F# we write

```
g >> f
```

Function composition

Functions can be composed
In mathematics we write

$$(f \circ g)(x) = f(g(x))$$

In F# we write

```
g >> f        or        f << g
```

Function composition

Functions can be composed
In mathematics we write

$$(f \circ g)(x) = f(g(x))$$

In F# we write

`g >> f` or `f << g`

The arrows point towards the outermost function

Function composition

Functions can be composed
In mathematics we write

$$(f \circ g)(x) = f(g(x))$$

In F# we write

```
g >> f        or        f << g
```

i.e. the function that will run last

Function composition

Functions can be composed
In mathematics we write

$$(f \circ g)(x) = f(g(x))$$

In F# we write

g >> f   or   f << g

```
let (>>) g f = fun x -> f (g x)
```

```
let (<<) f g = fun x -> f (g x)
```

Operators |> and <|

The operator |> means "send the value as argument to the function on the right"

`x |> f` is equivalent to `f x`

The operator <| means "send the value as argument to the function on the left"

`f <| x` is equivalent to `f x`

Operators |> and <|

The operator |> means "send the value as argument to the function on the right"

This just seems like wasted effort… `x |> f` is harder to read than `f x`

`f <| x` is equivalent to `f x`

Operators |> and <|

Both operators can be composed
(but we usually use |> for that)

```
4 |> (fun x -> x+x) |> (*) 3 =
```

Operators |> and <|

Both operators can be composed
(but we usually use |> for that)

```
4 |> (fun x -> x+x) |> (*) 3 =
(fun x -> x+x) 4 |> (*) 3 =
```

Operators |> and <|

Both operators can be composed
(but we usually use |> for that)

```
4 |> (fun x -> x+x) |> (*) 3 =
(fun x -> x+x) 4 |> (*) 3 =
4 + 4 |> (*) 3
```

Operators |> and <|

Both operators can be composed
(but we usually use |> for that)

```
4 |> (fun x -> x+x) |> (*) 3 =
(fun x -> x+x) 4 |> (*) 3 =
4 + 4 |> (*) 3 =
8 |> (*) 3
```

Operators |> and <|

Both operators can be composed
(but we usually use |> for that)

```
4 |> (fun x -> x+x) |> (*) 3 =
(fun x -> x+x) 4 |> (*) 3 =
4 + 4 |> (*) 3 =
8 |> (*) 3 =
(*) 3 8
```

Operators |> and <|

Both operators can be composed
(but we usually use |> for that)

```
4 |> (fun x -> x+x) |> (*) 3 =
(fun x -> x+x) 4 |> (*) 3 =
4 + 4 |> (*) 3 =
8 |> (*) 3 =
(*) 3 8 =
3 * 8
```

Remember that `(*) 3`
is the same as `fun x -> 3 * x`

Operators |> and <|

Both operators can be composed
(but we usually use |> for that)

```
4 |> (fun x -> x+x) |> (*) 3 =
(fun x -> x+x) 4 |> (*) 3 =
4 + 4 |> (*) 3 =
8 |> (*) 3 =
(*) 3 8 =
3 * 8 =
24
```

Remember that `(*) 3`
is the same as `fun x -> 3 * x`

Operators |> and <|

While

`x |> f`

is not easier to read than

`f x`,

`x |> f |> g |> h`

is easier to read than

`h (g (f x))`

(especially when the functions are large or partially applied with many arguments) and naturally shows how x is transformed by the functions

Piping and HoFs

We can do great things with piping, function composition and higher-order functions

Recall our equation solver

```
let solve (a, b, c) =
  let sqrtD =
    let d = b * b - 4.0 * a * c
    if d < 0.0 || a = 0.0 then
      raise SolveSDP
    else
       sqrt d
  ((-b + sqrtD) / (2.0 * a),
   (-b - sqrtD) / (2.0 * a))
```

Which solves the formula $ax^2 + bx + c = 0$

Piping and HoFs

Create a function that given a list of second degree polynomials (as defined before), return the smallest non-negative root smaller than 100 (assuming one exists).

Piping and HoFs

Create a function that given a list of second degre), return th aller than 100 (assuming one exists).

Start with the list of polynomials

```
sdps
```

Current result type: `(float * float * float) list`

Piping and HoFs

Create a function that given a list of second degre
),
return th
aller than 100 (assuming one exists).

Solve the polynomials

```
sdps |>
List.map solve
```

Current result type: `(float * float) list`

Piping and HoFs

Create a function that given a list of second degr[...]),
return [...]aller
than 100 (assuming one exists).

Flatten the list of solutions - rather than a list of pairs of roots return a list of roots

```
sdps |>
List.map solve |>
List.fold (fun acc (a, b) -> a::b::acc) []
```

Current result type: `float list`

IT UNIVERSITY OF COPENHAGEN

Piping and HoFs

Create a function that given a list of second
deg),
return aller
than 100 (assuming one exists).

Remove all roots smaller than zero

```
sdps |>
List.map solve |>
List.fold (fun acc (a, b) -> a::b::acc) [] |>
List.filter ((<) 0.0)
```

Current result type: `float list`

Piping and HoFs

Create a function that given a list of second
degr),
return aller
than 100 (assuming one exists).

Remove all roots smaller than zero

```
sdps |>
List.map solve |>
List.fold (fun acc (a, b) -> a::b::acc) [] |>
List.filter ((<) 0.0)
```

Same as `(fun x -> 0.0 < x)`

Current result type: `float list`

Piping and HoFs

Create a function that given a list of second
deg),
return aller
than 100 (assuming one exists).

Return the smallest root that is smaller than 100.0

```
sdps |>
List.map solve |>
List.fold (fun acc (a, b) -> a::b::acc) [] |>
List.filter ((<) 0.0) |>
List.fold min 100.0
```

Current result type: float

Piping and HoFs

```
sdps |>
List.map solve |>
List.fold (fun acc (a, b) -> a::b::acc) [] |>
List.filter ((<) 0.0) |>
List.fold min 100.0
```

is much easier to read than

```
List.fold min 100.0
  (List.filter ((<) 0.0)
    (List.fold (fun acc (a, b) -> a::b::acc) []
                (List.map solve sdps)))
```

Piping, composition, and HoFs

```
sdps |>
List.map solve |>
List.fold (fun acc (a, b) -> a::b::acc) [] |>
List.filter ((<) 0.0) |>
List.fold min 100.0
```

… and is identical to

```
(List.map solve >>
 List.fold (fun acc (a, b) -> a::b::acc) [] >>
 List.filter ((<) 0.0) >>
 List.fold min 100.0) sdps
```

IT UNIVERSITY OF COPENHAGEN

Piping, composition, and HoFs

```
sdps |>
List.map solve |>
List.fold (fun acc (a, b) -> a::b::acc) [] |>
List.filter ((<) 0.0) |>
List.fold min 100.0
```


```
sdps |>
List.map solve |>
List.fold (fun acc (a, b) -> a::b::acc) [] |>
List.filter ((<) 0.0) |>
List.fold min 100.0
```

Piping, composition, and HoFs

```
sdps |>
List.map solve |>
List.fold (fun acc (a, b) -> a::b::acc) [] |>
List.filter ((<) 0.0) |>
List.fold min 100.0
```

```
[(3., 2., 1); (6., 5., 4); (9., 8., 7)] |>
List.map solve |>
List.fold (fun acc (a, b) -> a::b::acc) [] |>
List.filter ((<) 0.0) |>
List.fold min 100.0
```

Piping, composition, and HoFs

```
sdps |>
List.map solve |>
List.fold (fun acc (a, b) -> a::b::acc) [] |>
List.filter ((<) 0.0) |>
List.fold min 100.0
```

```
[(0.14, -0.8); (0.28, -1.12); (0.32, -1.2)] |>
List.fold (fun acc (a, b) -> a::b::acc) [] |>
List.filter ((<) 0.0) |>
List.fold min 100.0
```

Piping, composition, and HoFs

```
sdps |>
List.map solve |>
List.fold (fun acc (a, b) -> a::b::acc) [] |>
List.filter ((<) 0.0) |>
List.fold min 100.0
```

```
[0.14; -0.8; 0.28; -1.12; 0.32; -1.2] |>
List.filter ((<) 0.0) |>
List.fold min 100.0
```

Piping, composition, and HoFs

```
sdps |>
List.map solve |>
List.fold (fun acc (a, b) -> a::b::acc) [] |>
List.filter ((<) 0.0) |>
List.fold min 100.0
```

```
[0.14; 0.28; 0.32] |>
List.fold min 100.0
```

Piping, composition, and HoFs

```
sdps |>
List.map solve |>
List.fold (fun acc (a, b) -> a::b::acc) [] |>
List.filter ((<) 0.0) |>
List.fold min 100.0
```

```
0.14
```

This week

- Inductively defined datatypes
- Expression trees
- Collections (sets and maps)
- Data representation
- Possibly 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
```

IT UNIVERSITY OF COPENHAGEN

Programming

```
type month =
  | 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)

Head and tails of a list

The functions List.head and List.tail will throw an exception if you try to get the head of an empty list (`List.head []`)

```
let safeHead =
   function
   | []     -> None
   | x :: _ -> Some x
```

```
let safeTail =
   function
   | []      -> None
   | _ :: xs -> Some xs
```

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                                                   ion

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

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

Examples

```
let safeHead =
   function
   | []       -> None
   | x :: _   -> Some x
```

```
let safeTail =
   function
   | []        -> None
   | _ :: xs   -> Some xs
```

```
Option.get : 'a option -> 'a
Option.map : ('a -> 'b) -> 'a option -> 'b option
Option.defaultValue :'a -> 'a option -> 'a
```

```
Option.get (safeHead [1;2;3]) = 1
```

which is the same as

```
[1;2;3] |> safeHead |> Option.get = 1
```

Examples

```
let safeHead =
  function
  | []       -> None
  | x :: _   -> Some x
```

```
let safeTail =
  function
  | []        -> None
  | _ :: xs   -> Some xs
```

```
Option.get : 'a option -> 'a
Option.map : ('a -> 'b) -> 'a option -> 'b option
Option.defaultValue :'a -> 'a option -> 'a
```

```
Option.get (safeHead [])
```

which is the same as

```
[] |> safeHead |> Option.get
```

Examples

```
let safeHead =
   function
   | []       -> None
   | x :: _   -> Some x
```

```
let safeTail =
   function
   | []        -> None
   | _ :: xs   -> Some xs
```

```
Option.get : 'a option -> 'a
Option.map : ('a -> 'b) -> 'a option -> 'b option
Option.defaultValue :'a -> 'a option -> 'a
```

```
Option.get (safeTail [1;2;3]) = [2; 3]
```

which is the same as

```
[1;2;3] |> safeHead |> Option.get = [2; 3]
```

Examples

```
let safeHead =
  function
  | []     -> None
  | x :: _ -> Some x
```

```
let safeTail =
  function
  | []      -> None
  | _ :: xs -> Some xs
```

```
Option.get : 'a option -> 'a
Option.map : ('a -> 'b) -> 'a option -> 'b option
Option.defaultValue :'a -> 'a option -> 'a
```

```
Option.map (fun x -> x + 10) (safeHead [1;2;3]) = Some 11
```

which is the same as

```
[1;2;3] |> safeHead |> Option.map (fun x -> x + 10) = Some 11
```

Examples

```
let safeHead =
  function
  | []     -> None
  | x :: _ -> Some x
```

```
let safeTail =
  function
  | []      -> None
  | _ :: xs -> Some xs
```

```
Option.get : 'a option -> 'a
Option.map : ('a -> 'b) -> 'a option -> 'b option
Option.defaultValue :'a -> 'a option -> 'a
```

```
Option.map (fun x -> x + 10) (safeHead []) = None
```

which is the same as

```
[] |> safeHead |> Option.map (fun x -> x + 10) = None
```

Examples

```
let safeHead =
   function
   | []       -> None
   | x :: _   -> Some x
```

```
let safeTail =
   function
   | []        -> None
   | _ :: xs   -> Some xs
```

```
Option.get : 'a option -> 'a
Option.map : ('a -> 'b) -> 'a option -> 'b option
Option.defaultValue :'a -> 'a option -> 'a
```

```
defaultValue 0 (safeHead [1;2;3]) = 1
```

which is the same as

```
[1;2;3] |> safeHead |> defaultValue 0 = 1
```

Examples

```
let safeHead =
   function
   | []       -> None
   | x :: _   -> Some x
```

```
let safeTail =
   function
   | []        -> None
   | _ :: xs   -> Some xs
```

```
Option.get : 'a option -> 'a
Option.map : ('a -> 'b) -> 'a option -> 'b option
Option.defaultValue :'a -> 'a option -> 'a
```

```
defaultValue 0 (safeHead []) = 0
```

which is the same as

```
[] |> safeHead |> defaultValue 0 = 0
```

Recursive

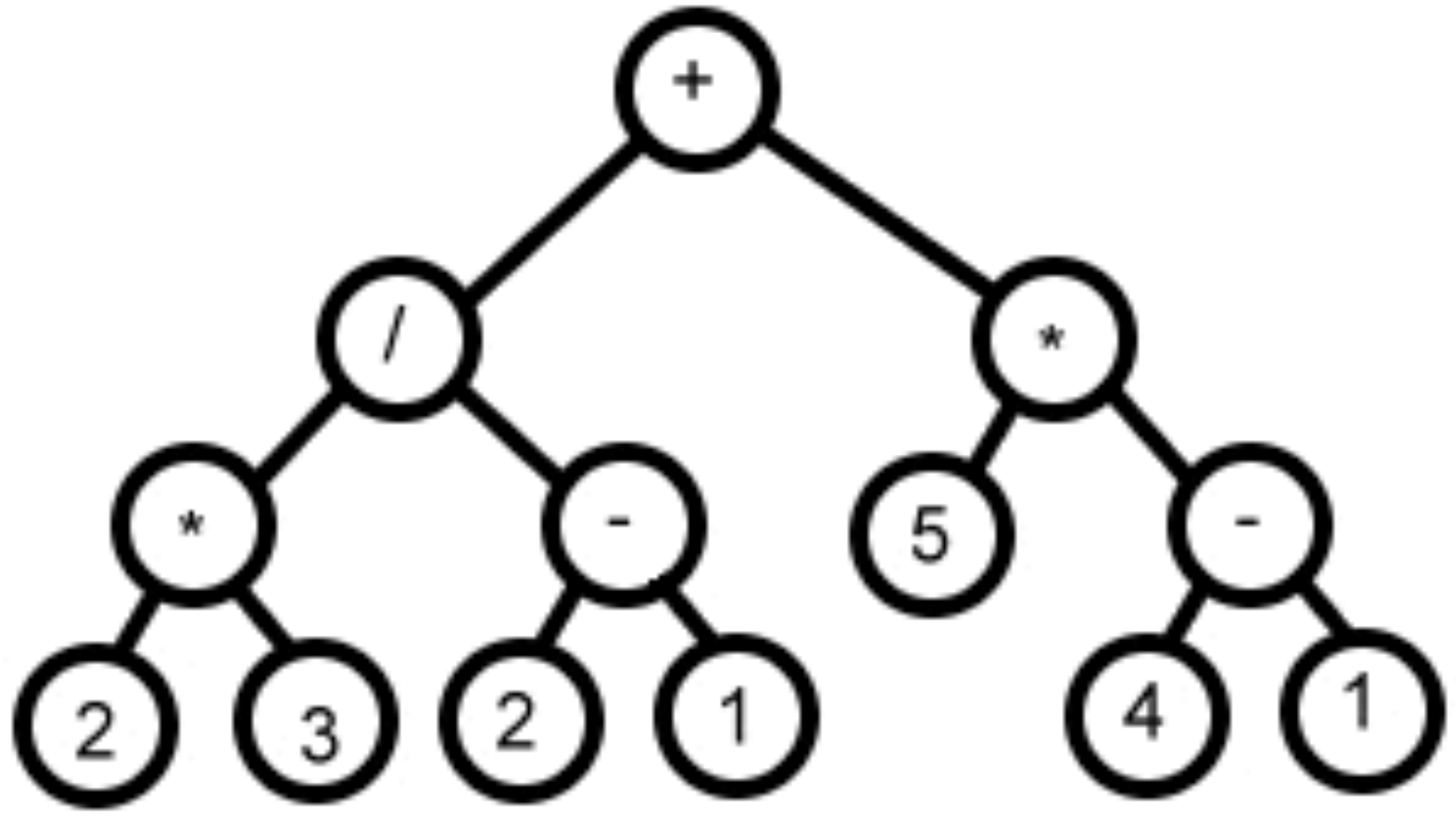
Algebraic types can be recursive and polymorphic

```
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)

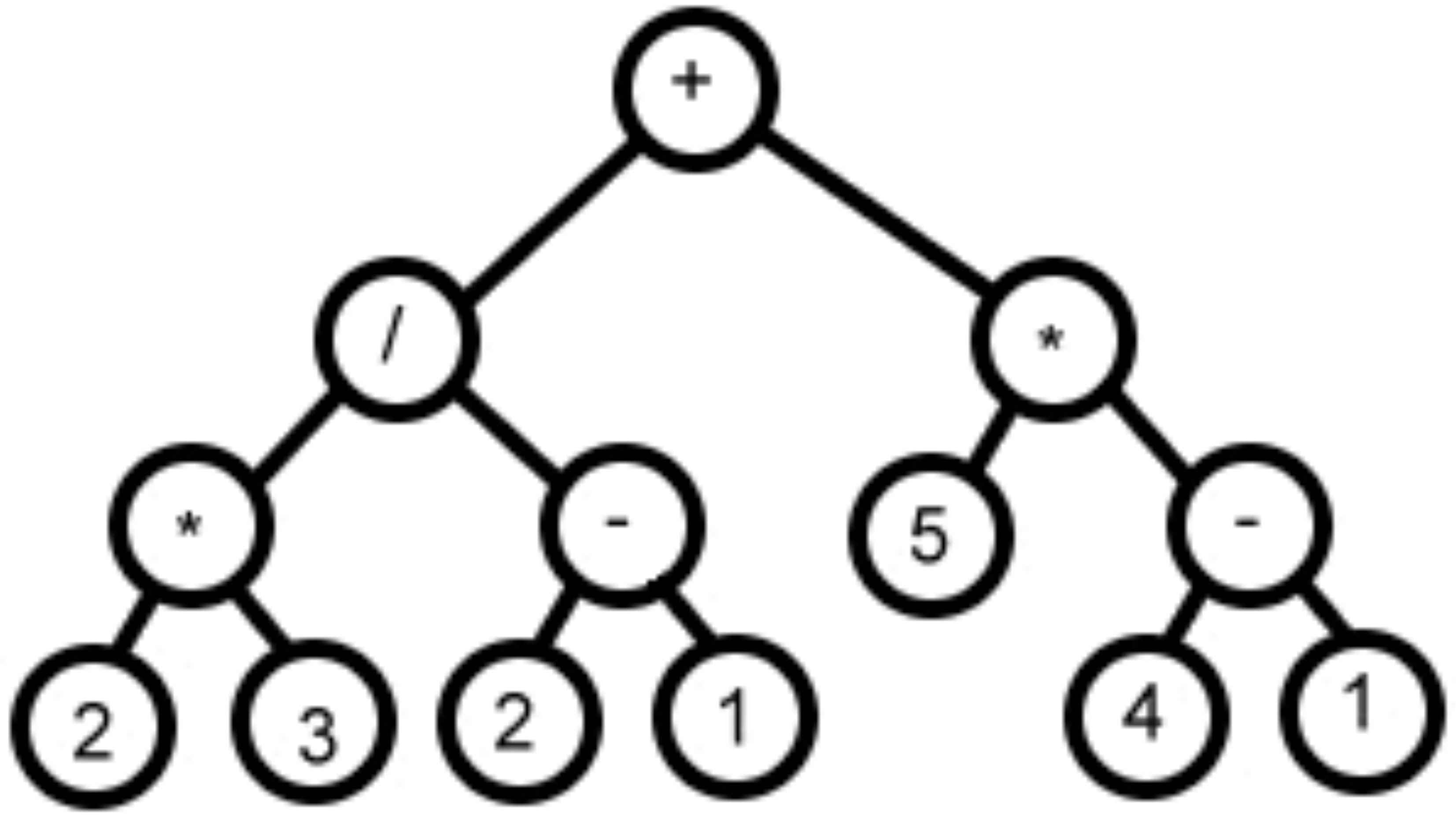
Expression trees

Expression trees are quite 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 also nice to recurse over

```
let rec show : term -> string =
    function
    | Const f -> sprintf "%d" 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)))";;
```

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>
```

```
Set.empty = set []
```

Sets (some functions)

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

```
Set.singleton 42 = set [42]
```

Sets (some functions)

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

```
Set.union (Set.singleton 42)
          (Set.singleton 123) = set [42; 123]
```

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.intersect (set [1;2;3]) (set [3;4;5]) = set [3]
```

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.map (fun x -> x + 2) (set [3;4;5]) = set [5;6;7]
```

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.map (fun x -> x + 2) (set [3;4;5]) = set [5;6;7]
Set.map ((+) 2) (set [3;4;5]) = set [5;6;7]
```

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.map (fun x -> x + 2) (set [3;4;5]) = set [5;6;7]

Set.map ((+) 2) (set [3;4;5]) = set [5;6;7]

Set.map (fun x -> x % 2 = 0) (set [3;4;5]) =
    set [false; true]
```

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.filter (fun x->x%2=0) (set[3;4;5]) = set[4]
```

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.filter (fun x->x%2=0) (set[3;4;5]) = set[4]

Set.filter (fun x->x%2=1) (set[3;4;5]) = set[3;5]
```

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
```

```
Set.fold (fun x y -> x + y) 0 (set[3;4;5]) = 12
```


IT UNIVERSITY OF COPENHAGEN

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
```

```
Set.fold (fun x y -> x + y) 0 (set[3;4;5]) = 12

Set.fold (+) 0 (set[3;4;5]) = 12
```

Sets (some functions)

```
Set.empty
```

Result: `set []`

Sets (some functions)

```
Set.empty |>
Set.add "hello"
```

Result: `set ["hello"]`

Sets (some functions)

```
Set.empty |>
Set.add "hello" |>
Set.add "there"
```

Result: `set ["hello"; "there"]`

Sets (some functions)

```
Set.empty |>
Set.add "hello" |>
Set.add "there" |>
Set.union (Set.ofList ["there";"you";"are"])
```

Result: `set ["hello"; "there"; "you"; "are"]`

Sets (some functions)

```
Set.empty |>
Set.add "hello" |>
Set.add "there" |>
Set.union (Set.ofList ["there";"you";"are"]) |>
Set.intersect (Set.ofList ["hello";"there";"you"])
```

Result: `set ["hello"; "there"; "you"]`

Sets (some functions)

```
Set.empty |>
Set.add "hello" |>
Set.add "there" |>
Set.union (Set.ofList ["there";"you";"are"]) |>
Set.intersect (Set.ofList ["hello";"there";"you"]) |>
Set.map String.length
```

Result: `set [5; 3]`

Sets (some functions)

```
Set.empty |>
Set.add "hello" |>
Set.add "there" |>
Set.union (Set.ofList ["there";"you";"are"]) |>
Set.intersect (Set.ofList ["hello";"there";"you"]) |>
Set.map String.length |>
Set.filter (fun x -> x % 2 = 1)
```

Result: `set [5; 3]`

Sets (some functions)

```
Set.empty |>
Set.add "hello" |>
Set.add "there" |>
Set.union (Set.ofList ["there";"you";"are"]) |>
Set.intersect (Set.ofList ["hello";"there";"you"]) |>
Set.map String.length |>
Set.filter (fun x -> x % 2 = 1) |>
Set.fold (+) 0
```

Result: 8

Maps

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

Maps (some functions)

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

```
Map.empty = map []
```

Maps (some functions)

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

```
Map.add "k" 4 Map.empty = map [("k", 4)]
```

Maps (some functions)

```
Map.empty    : Map<'a,'b>
Map.add      : 'a -> 'b -> Map<'a,'b> -> Map<'a,'b>
Map.ofList   : ('a*'b) list -> Map<'a,'b>
```

```
Map.ofList [("k",4); ("l",5)] =
    map [("k", 4); ("l",5)])
```

Maps (some functions)

```
Map.empty    : Map<'a,'b>
Map.add      : 'a -> 'b -> Map<'a,'b> -> Map<'a,'b>
Map.ofList   : ('a*'b) list -> Map<'a,'b>
Map.find     : 'a -> Map<'a,'b> -> 'b
```

```
Map.find "k" (Map.add "k" 4 Map.empty) = 4
```

which is the same as

```
Map.empty |> Map.add "k" 4 |> Map.find "k" = 4
```

Maps (some functions)

```
Map.empty    : Map<'a,'b>
Map.add      : 'a -> 'b -> Map<'a,'b> -> Map<'a,'b>
Map.ofList   : ('a*'b) list -> Map<'a,'b>
Map.find     : 'a -> Map<'a,'b> -> 'b
```

```
Map.find "l" (Map.add "k" 4 Map.empty)
```

Both are identical and throw exceptions

```
Map.empty |> Map.add "k" 4 |> Map.find "l"
```

Maps (some functions)

```
Map.empty   : Map<'a,'b>
Map.add     : 'a -> 'b -> Map<'a,'b> -> Map<'a,'b>
Map.ofList  : ('a*'b) list -> Map<'a,'b>
Map.find    : 'a -> Map<'a,'b> -> 'b
Map.tryFind : 'a -> Map<'a,'b> -> 'b option
```

```
Map.tryFind "k" (Map.add "k" 4 Map.empty) = Some 4
```

which is the same as

```
Map.empty |> Map.add "k" 4 |> Map.tryFind "k" =
    Some 4
```

Maps (some functions)

```
Map.empty    : Map<'a,'b>
Map.add      : 'a -> 'b -> Map<'a,'b> -> Map<'a,'b>
Map.ofList   : ('a*'b) list -> 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.map (fun k v->k+string v) (Map.ofList [("k",4);("l",5)]) =
   map [("k","k4");("l","l5")]
```

which is the same as

```
[("k",4);("l",5)]|>Map.ofList|>Map.map(fun k v->k+string v) =
map [("k","k4");("l","l5")]
```

IT UNIVERSITY OF COPENHAGEN

Maps (some functions)

```
Map.empty    : Map<'a,'b>
Map.add      : 'a -> 'b -> Map<'a,'b> -> Map<'a,'b>
Map.ofList   : ('a*'b) list -> 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.filter (fun _ v->v%2=0) (Map.ofList [("k",4);("l",5)]) =
   map [("k",4)]
```

Maps (some functions)

```
Map.empty    : Map<'a,'b>
Map.add      : 'a -> 'b -> Map<'a,'b> -> Map<'a,'b>
Map.ofList   : ('a*'b) list -> 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

Map.fold (fun acc k v->acc+k+string v)
         ""
         (Map.ofList [("k",4);("l",5)]) = "k4l5"
```

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
  …
```

Arithmetic expressions

Arithmetic expressions can be evaluated
in the context of a state

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

```
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
  …
```

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

Arithmetic expressions

Arithmetic expressions can be evaluated in the context of a state

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

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

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

Arithmetic expressions

Arithmetic expressions can be evaluated in the context of a state

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

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

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

Arithmetic expressions

Arithmetic expressions can be evaluated in the context of a state

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

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

```
let binop : ('a -> 'b -> 'c)
            -> (state -> 'a)
            -> (state -> 'b)
            -> state -> 'c =
  fun f x y s -> f (x s) (y s)
```

Arithmetic expressions

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

```
let binop : (‘a -> ‘b -> ‘c)
             -> (state -> ‘a)
             -> (state -> ‘b)
             -> state -> ‘c =
   fun f x y s -> f (x s) (y s)

let rec evalA arith =
  match arith with
  | N n       -> fun _ -> n
  | V v       -> Map.find v
  | Add(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
```

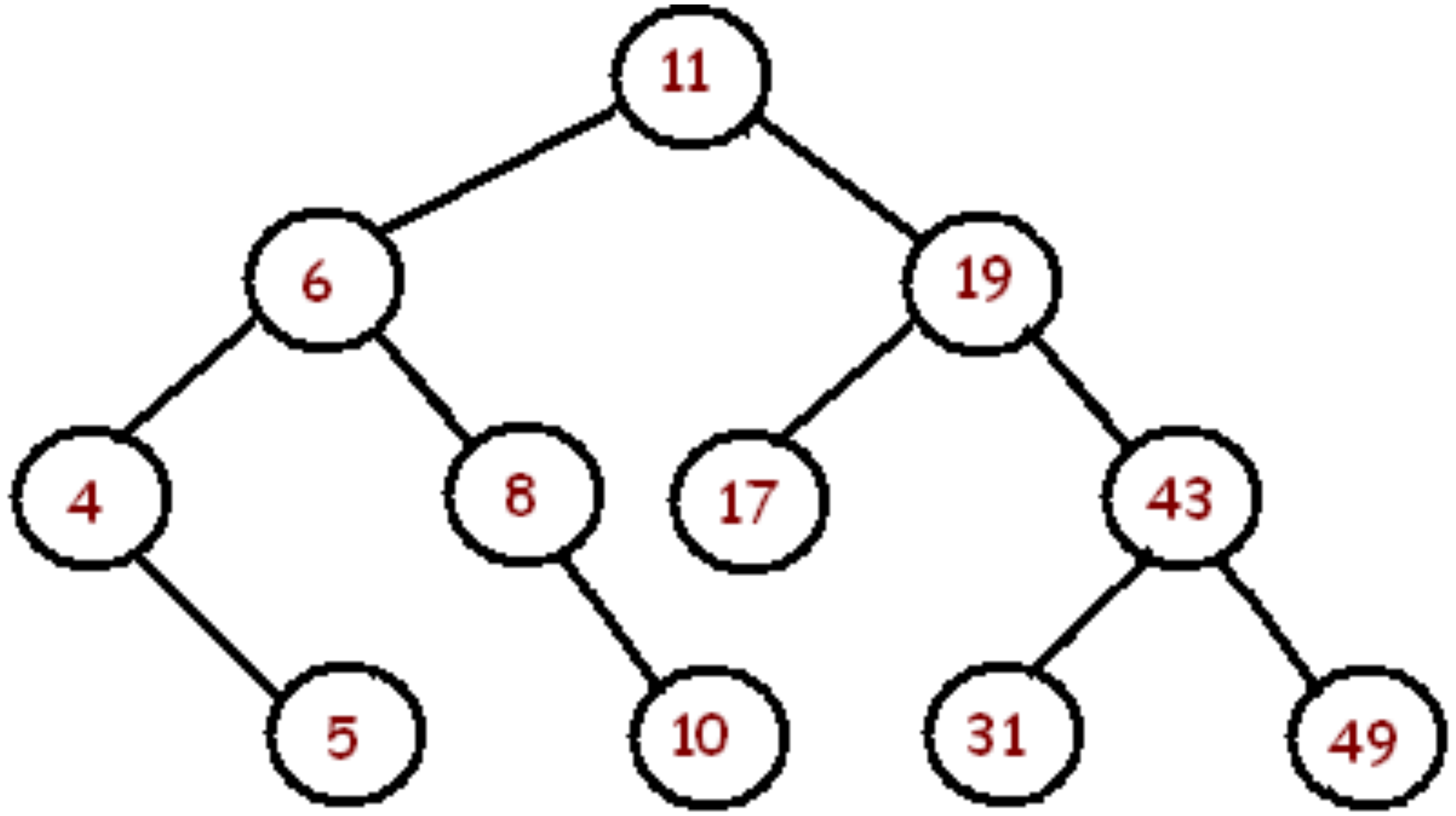
Programs evaluation is done by updating the state

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

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

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

Questions?