15-150

Principles of Functional Programming

Slides for Lecture 16

Modules

March 19, 2024

Michael Erdmann

Lessons:

- ML’s Module System:
  - Signatures and Structures
  - Encapsulate common idioms
  - Design large programs

Lessons:

- ML’s Module System:
  - Signatures and Structures
  - Encapsulate common idioms
  - Design large programs

Example: **`Int.toString`** is a function inside a *structure* called **`Int`**.

If you look in the SML Basis Library https://smlfamily.github.io/Basis/index.html, you will see that structure **`Int`** “ascribes” to a *signature* called **`INTEGER`**.

Example: **`Int.toString`** is a function inside a *structure* called **`Int`**.

If you look in the SML Basis Library http://sml-family.org/Basis/, you will see that structure **`Int`** "ascribes" to a *signature* called **`INTEGER`**.

We will learn what those words mean.

(Basically: the signature says the function has to exist and have type **`int -> string`**.)

Lessons:

- ML’s Module System:
  - Signatures and Structures
  - Encapsulate common idioms
  - Design large programs
- Abstraction (specified via a signature)
  - Abstract Data
  - Information Hiding
- Implementation (within a structure)
  - Abstraction Function (how does a specific implementation encode an abstraction)
  - Representation Invariants (what constraints must an implementation respect)

Signatures & Structures

A **signature** specifies an interface.

A **structure** provides an implementation.

Signatures & Structures

A **signature** specifies an interface.

A **structure** provides an implementation.

Example:

A **queue** is a first-in first-out datastructure.

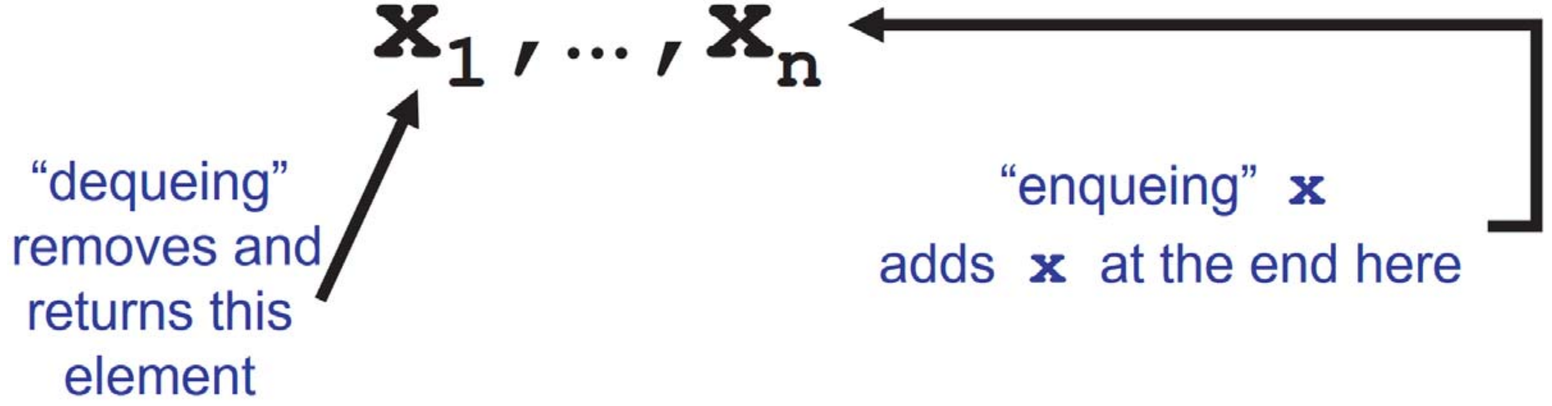
Signatures & Structures

A **signature** specifies an interface.

A **structure** provides an implementation.

Example:

A **queue** is a first-in first-out datastructure.



`x`$_1$`, …, x`$_n$

“dequeing” removes and returns this element

“enqueing” **`x`** adds **`x`** at the end here

Signatures & Structures

A **signature** specifies an interface.

A **structure** provides an implementation.

Example:

A **queue** is a first-in first-out datastructure.

We can **describe a queue abstractly** by specifying a (new) queue type, along with operations on that type.

That’s a signature.
Then we implement it in a structure.

Queue Signature

```
signature QUEUE =
sig

  type 'a q                   (* abstract *)

  val empty : 'a q

  val enq : 'a q * 'a -> 'a q

  val null : 'a q -> bool

  exception Empty

  (* will raise Empty if called on empty q *)
  val deq : 'a q -> 'a * 'a q

end
```

Representational Independence

The signature intentionally says *nothing* about how to represent the abstract datatype **`'a q`** for queues.

The responsibility of any queue implementation is to provide all the types and values specified in the signature, but details are unspecified.

That gives the implementation flexibility.

(We will see two different queue implementations.)

A user of queues in turn only needs to see the signature, not the details of any specific queue implementation. Indeed, the user should not see or rely on those details, in case the developer changes them.

First QUEUE implementation

Use a single list.

Need to say **how** the list represents the abstract queue:

(called “abstraction function”)

The list represents the queue elements in arrival order.

First QUEUE implementation

```
signature QUEUE =
sig
  type 'a q    (* abstract *)
  val empty : 'a q
  val enq : 'a q * 'a -> 'a q
  val null : 'a q -> bool
  exception Empty
  val deq : 'a q -> 'a * 'a q
end
```

```
structure Queue : QUEUE =
struct
```

Pronounced “ascribes” or “ascribes to” or “ascribes transparently”.

“ascribe” means:

The structure provides all the items specified in the signature. (The structure may contain additional items, e.g., helper functions, but those will not be visible outside the structure.)

“transparent” means:

The representation of the abstract queue type is visible outside the structure, e.g., to a client.

First QUEUE implementation

```
signature QUEUE =
sig
  type 'a q   (* abstract *)
  val empty : 'a q
  val enq : 'a q * 'a -> 'a q
  val null : 'a q -> bool
  exception Empty
  val deq : 'a q -> 'a * 'a q
end
```

```
structure Queue : QUEUE =
struct

  type 'a q  =  'a list

  val empty = []

  fun enq (q, x) = q @ [x]

  val null = List.null

  exception Empty

  fun deq [] = raise Empty
    | deq (x::q) = (x, q)

end
```

Extra Code is Hidden

We could put extra code constructs (such as helper functions) into the structure.

The code will be available within the structure.

Only what is specified in the signature will be accessible outside the structure.

Interacting with the Queue

```
val q2 = Queue.enq(Queue.enq(Queue.empty,1),2)
```

Q: What is the type of `q2` ?

(ignore that you know
it is **`int list`**)

Interacting with the Queue

```
val q2 = Queue.enq(Queue.enq(Queue.empty,1),2)
```

Q: What is the type of `q2` ?

A: `int Queue.q`

Why? Because:

First, the signature specifies that queues have type `'a q`, with `'a` representing the element type. That is `int` here.

Second, we have implemented queues using a structure called `Queue`. The type is defined inside the structure, so the type has the qualified name `'a Queue.q`, here with `'a` instantiated to `int`.

Interacting with the Queue

```
val q2 = Queue.enq(Queue.enq(Queue.empty,1),2)
```

Q: What is the type of `q2` ?

A: `int Queue.q`

Also:

ML will print the list `[1,2]`. We can see the list because of transparent ascription (more on how to hide that later).

Next, consider:

```
val (a, b) = Queue.deq q2
val (c, _) = Queue.deq q2
val (d, _) = Queue.deq b
```

Q: What are the bindings for `a, c, d` ?

Interacting with the Queue

```
val q2 = Queue.enq(Queue.enq(Queue.empty,1),2)
```

Q: What is the type of `q2` ?

A: `int Queue.q`

Also:

ML will print the list `[1,2]`. We can see the list because of transparent ascription (more on how to hide that later).

Next, consider:

```
val (a, b) = Queue.deq q2
val (c, _) = Queue.deq q2
val (d, _) = Queue.deq b
```

Q: What are the bindings for `a, c, d` ?

A: `[1/a, 1/c, 2/d]`

(We also have the binding `[[2]/b]`, but that is only because of transparent ascription. We will see how to hide queue internals.)

How long does enqueing take?

```
fun enq (q, x) = q @ [x]
```

O(n), with n the number of items in `q`.

We can improve that with a different representation of queues.

Second QUEUE implementation

Use a pair of lists:

`(front, back)`.

Abstraction Function:

`front @ (rev back)`

represents the queue
elements in arrival order.

Second QUEUE implementation

```
signature QUEUE =
sig
  type 'a q    (* abstract *)
  val empty : 'a q
  val enq : 'a q * 'a -> 'a q
  val null : 'a q -> bool
  exception Empty
  val deq : 'a q -> 'a * 'a q
end
```

```
structure Q :> QUEUE =
struct
```

"opaque ascription"

This means the representation details are hidden from any user external to the structure. Only items specified by the signature are visible.

With transparent ascription, a user can see and sometimes mess with a representation (earlier, ML would print out lists for queues).

With opaque ascription, ML will only print a dash. An external user cannot see or mess with the internal representation.

Second QUEUE implementation

```
signature QUEUE =
sig
  type 'a q    (* abstract *)
  val empty : 'a q
  val enq : 'a q * 'a -> 'a q
  val null : 'a q -> bool
  exception Empty
  val deq : 'a q -> 'a * 'a q
end
```

```
structure Q :> QUEUE =
struct

  type 'a q  = 'a list * 'a list

  val empty = ([],[])

  fun enq ((f,b), x) = (f, x::b)
```

Satisfies requirement that `f @ (rev(x::b))` constitute the queue elements in arrival order.

```
end
```

Second QUEUE implementation

```
signature QUEUE =
sig
  type 'a q   (* abstract *)
  val empty : 'a q
  val enq : 'a q * 'a -> 'a q
  val null : 'a q -> bool
  exception Empty
  val deq : 'a q -> 'a * 'a q
end
```

```
structure Q :> QUEUE =
struct

  type 'a q  = 'a list * 'a list

  val empty = ([],[])

  fun enq ((f,b), x) = (f, x::b)

  fun null ([],[]) = true
    | null    _    = false

  exception Empty

  fun deq ([],[]) = raise Empty
    | deq ([], b) = deq (rev b, [])
    | deq (x::f, b) = (x, (f, b))

end
```

Interacting with Q implementation

```
val q2’ = Q.enq(Q.enq(Q.empty,1),2)
```

Question: What is the type of `q2’` ?

Interacting with Q implementation

```
val q2’ = Q.enq(Q.enq(Q.empty,1),2)
```

Question: What is the type of `q2’` ?

Answer: `int Q.q`

```
signature QUEUE =
sig
  type 'a q    (* abstract *)
  val empty : 'a q
  val enq : 'a q * 'a -> 'a q
  val null : 'a q -> bool
  exception Empty
  val deq : 'a q -> 'a * 'a q
end
```

```
structure Q :> QUEUE =
struct
  type 'a q  = 'a list * 'a list

  val empty = ([],[])

  fun enq ((f,b), x) = (f, x::b)

  fun null ([],[]) = true
    | null     _     = false

  exception Empty

  fun deq ([],[]) = raise Empty
    | deq ([], b) = deq (rev b, [])
    | deq (x::f, b) = (x, (f, b))
end
```

Interacting with Q implementation

```
val q2' = Q.enq(Q.enq(Q.empty,1),2)
```

Question: What is the type of `q2'` ?

Answer: `int Q.q`

Also:

ML will now ***not*** print the internals, because of opaque ascription. ML will merely print a dash:

`val q2' = - : int Q.q` .

Consider again the following, now using the Q implementation:

```
val (a, b) = Q.deq q2'
val (c, _) = Q.deq q2'
val (d, _) = Q.deq b
```

Question: What are the bindings for `a, c, d` ?

Interacting with Q implementation

```
val q2’ = Q.enq(Q.enq(Q.empty,1),2)
```

Question: What is the type of `q2’` ?

Answer: `int Q.q`

Also:

ML will now ***not*** print the internals, because of opaque ascription. ML will merely print a dash:

```
val q2’ = - : int Q.q .
```

Consider again the following, now using the Q implementation:

```
val (a, b) = Q.deq q2’
val (c, _) = Q.deq q2’
val (d, _) = Q.deq b
```

Question: What are the bindings for `a, c, d` ?

Answer: As before: `[1/a, 1/c, 2/d]`

(We also have a binding of a queue to `b`, but the internals are now hidden.)

Now, how long goes enqueing take?

```
fun enq ((f, b), x) = (f, x::b)
```

O(1) !

dequeuing can now take O(n) time.

However, enqueing and dequeing n items will only take O(n) time total, so on average it is O(1).

One says the *amortized* cost is O(1).

The Two Implementations

```
structure Queue : QUEUE =
struct
  type 'a q  =  'a list

  val empty = []

  fun enq (q, x) = q @ [x]

  val null = List.null

  exception Empty

  fun deq [] = raise Empty
    | deq (x::q) = (x, q)
end
```

```
structure Q :> QUEUE =
struct
  type 'a q  = 'a list * 'a list

  val empty = ([],[])

  fun enq ((f,b), x) = (f, x::b)

  fun null ([],[]) = true
    | null    _    = false

  exception Empty

  fun deq ([],[]) = raise Empty
    | deq ([], b) = deq (rev b, [])
    | deq (x::f, b) = (x, (f, b))
end
```

Compare queue *internals*

operation	Queue	Q
`empty`	`[]`	`([],[])`
`enq 1`	`[1]`	`([],[1])`
`enq 2`	`[1,2]`	`([],[2,1])`

Compare queue *internals*

operation	Queue	Q
`empty`	`[]`	`([],[])`
`enq 1`	`[1]`	`([],[1])`
`enq 2`	`[1,2]`	`([],[2,1])`
`deq`		

(this returns `1` *and the new queue)*

Compare queue *internals*

operation	Queue	Q
`empty`	`[]`	`([],[])`
`enq 1`	`[1]`	`([],[1])`
`enq 2`	`[1,2]`	`([],[2,1])`
`deq`	`[2]`	

(this returns `1` *and the new queue)*

Compare queue *internals*

operation	Queue	Q
`empty`	`[]`	`([],[])`
`enq 1`	`[1]`	`([],[1])`
`enq 2`	`[1,2]`	`([],[2,1])`

`deq`	`[2]`	*briefly this:*
		`([1,2],[])`
		then this:
		`([2],[])`

(this returns `1` *and the new queue)*

Compare queue *internals*

operation	Queue	Q
`empty`	`[]`	`([],[])`
`enq 1`	`[1]`	`([],[1])`
`enq 2`	`[1,2]`	`([],[2,1])`

`deq`	`[2]`	*briefly this:*
		`([1,2],[])`
		then this:
		`([2],[])`

Note: With Q's opaque ascription, internals are hidden from client.

Compare queue *internals*

operation	Queue	Q
`empty`	`[]`	`([],[])`
`enq 1`	`[1]`	`([],[1])`
`enq 2`	`[1,2]`	`([],[2,1])`

`deq`	`[2]`	*briefly this:*
		`([1,2],[])`
		then this:
		`([2],[])`

`enq 3`	`[2,3]`	`([2],[3])`
`enq 4`	`[2,3,4]`	`([2],[4,3])`

Dictionary Signature

A dictionary is a collection of pairs of the form **`(key, value)`**.

We require all the keys to be unique in a given dictionary.

```
signature DICT =
sig









end
```

Dictionary Signature

A dictionary is a collection of pairs of the form **`(key, value)`**.

We require all the keys to be unique in a given dictionary.

(for the time being, we'll fix the key type)

```
signature DICT =
sig
  type key = string                (* concrete *)



end
```

Dictionary Signature

A dictionary is a collection of pairs of the form `(key, value)`.

We require all the keys to be unique in a given dictionary.

(for the time being, we'll fix the key type)

```
signature DICT =
sig
   type key = string               (* concrete *)
   type 'a entry = key * 'a        (* concrete *)




end
```

we'll allow the value type to be polymorphic

Dictionary Signature

A dictionary is a collection of pairs of the form **`(key, value)`**.

We require all the keys to be unique in a given dictionary.

```
signature DICT =
sig
  type key = string         (* concrete *)
  type 'a entry = key * 'a  (* concrete *)

  type 'a dict              (* abstract *)

  val empty : 'a dict

  val lookup : 'a dict -> key -> 'a option
  val insert : 'a dict * 'a entry -> 'a dict
end
```

(replace entry if key already appears in the dictionary)

Dictionary Implementation

We will use a tree implementation.

Abstraction Function: The `(key,value)` items in the tree constitute the dictionary.

We further impose a **Representation Invariant**:

The tree must be **sorted** on `key` (and all keys must be unique).

This means:

All functions within the structure may *assume* that any trees they receive are sorted

and

must ensure that any trees returned are sorted.

Dictionary Implementation

We will use a tree implementation.

Abstraction Function: The `(key,value)` items in the tree constitute the dictionary.

We further impose a **Representation Invariant**:

The tree must be **sorted** on `key` (and all keys must be unique).

This means:

All functions within the structure may *assume* that any trees they receive are sorted

and

must ensure that any trees returned are sorted.

(Similarly for key uniqueness.)

BST Implementation of Dictionaries

```
signature DICT =
sig
  type key = string             (* concrete *)
  type 'a entry = key * 'a      (* concrete *)

  type 'a dict                  (* abstract *)

  val empty : 'a dict

  val lookup : 'a dict -> key -> 'a option
  val insert : 'a dict * 'a entry -> 'a dict
end
```

```
structure BST : DICT =
struct
  type key = string
  type 'a entry = key * 'a

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

Observe: Because the datatype is *not* declared in the signature, a user external to the structure *cannot* pattern match on or otherwise use the constructors.

They *will* be visible because we will declare

type 'a dict = 'a tree

and because we are using transparent ascription.

So, a user can see the internals of our representation, but cannot mess with them.

BST Implementation of Dictionaries

```
signature DICT =
sig
  type key = string              (* concrete *)
  type 'a entry = key * 'a       (* concrete *)

  type 'a dict                   (* abstract *)

  val empty : 'a dict

  val lookup : 'a dict -> key -> 'a option
  val insert : 'a dict * 'a entry -> 'a dict
end
```

```
structure BST : DICT =
struct
  type key = string
  type 'a entry = key * 'a

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

  type 'a dict = 'a tree

  val empty = Empty

  fun lookup ...

  fun insert ...
end
```

BST Implementation of Dictionaries

```
(* insert : 'a dict * 'a entry -> 'a dict *)
```

BST Implementation of Dictionaries

```
(* insert : 'a dict * 'a entry -> 'a dict *)

fun insert (Empty, e) = Node(Empty, e, Empty)
  | insert (Node(lt, e’ as (k’,_), rt),...) =
```

Layered Pattern Matching

Here, this creates bindings of the full **`(key,value)`** entry to `e’`, of just the **`key`** part to `k’`, and the wildcard `_` matches the **`value`** part, without producing a binding.

BST Implementation of Dictionaries

```
(* insert : 'a dict * 'a entry -> 'a dict *)

fun insert (Empty, e) = Node(Empty, e, Empty)
  | insert (Node(lt, e’ as (k’,_), rt),
            e as (k,_)) =
     (case String.compare(k,k’) of
       EQUAL => Node(lt, e, rt)



                                            )
```

“replace” exisiting entry with new entry on same key

BST Implementation of Dictionaries

```
(* insert : 'a dict * 'a entry -> 'a dict *)

fun insert (Empty, e) = Node(Empty, e, Empty)
  | insert (Node(lt, e’ as (k’,_), rt),
            e as (k,_)) =
      (case String.compare(k,k’) of
        EQUAL => Node(lt, e, rt)
       |LESS => Node(insert(lt,e), e’, rt)
       |GREATER => Node(lt, e’, insert(rt,e)))
```

BST Implementation of Dictionaries

```
(* lookup : 'a dict -> key -> 'a option *)

fun lookup tree key =
 let
    fun lk (Empty) = NONE
      | lk (Node(left, (k,v), right)) =
         (case String.compare(key,k) of
             EQUAL => SOME(v)
           | LESS => lk left
           | GREATER => lk right)
 in
    lk tree
 end
```

Interacting with BST

```
val d = BST.insert(
         BST.insert(
          BST.insert(
           BST.insert(BST.empty,("a",1)),
                                ("b",2)),
                               ("c",3)),
                              ("d",4))
```

Question: What is the type of `d` ?

Interacting with BST

```
val d = BST.insert(
         BST.insert(
          BST.insert(
           BST.insert(BST.empty,("a",1)),
                                ("b",2)),
                               ("c",3)),
                              ("d",4))
```

Question: What is the type of `d` ?

Answer: `int BST.dict`

```
signature DICT =
sig
  type key = string            (* concrete *)
  type 'a entry = key * 'a     (* concrete *)

  type 'a dict                 (* abstract *)

  val empty : 'a dict

  val lookup : 'a dict -> key -> 'a option
  val insert : 'a dict * 'a entry -> 'a dict
end
```

```
structure BST : DICT =
struct
        ...
end
```

Interacting with BST

```
val d = BST.insert(
         BST.insert(
          BST.insert(
           BST.insert(BST.empty,("a",1)),
                                ("b",2)),
                               ("c",3)),
                              ("d",4))
```

Question: What is the type of `d` ?

Answer: `int BST.dict`

ML will print the internal tree representation of `d`.
We could have hidden that by using opaque ascription:

```
structure BST :> DICT = ...
```

(Reminder: Despite seeing the internals a client cannot pattern match on the constructors since they are not declared in the signature.)

Interacting with BST

```
val d = BST.insert(
         BST.insert(
          BST.insert(
           BST.insert(BST.empty,("a",1)),
                                ("b",2)),
                               ("c",3)),
                              ("d",4))
```

Question: What is the type of `d` ?

Answer: `int BST.dict`

ML will print the internal tree representation of `d`.
We could have hidden that by using opaque ascription:

```
structure BST :> DICT = ...
```

(Reminder: Despite seeing the internals a client cannot pattern match on the constructors since they are not declared in the signature.)

Now consider: `val look = BST.lookup d`

Question: What is the type of `look` ?

Interacting with BST

```
val d = BST.insert(
         BST.insert(
          BST.insert(
           BST.insert(BST.empty,("a",1)),
                                ("b",2)),
                               ("c",3)),
                              ("d",4))
```

Question: What is the type of `d` ?

Answer: `int BST.dict`

ML will print the internal tree representation of `d`.
We could have hidden that by using opaque ascription:

```
structure BST :> DICT = ...
```

(Reminder: Despite seeing the internals a client cannot pattern match on the constructors since they are not declared in the signature.)

Now consider: `val look = BST.lookup d`

Question: What is the type of `look` ?

Answer: `BST.key -> int option`

(same as `string -> int option`)

Interacting with BST

```
val d = BST.insert(
         BST.insert(
          BST.insert(
           BST.insert(BST.empty,("a",1)),
                               ("b",2)),
                              ("c",3)),
                             ("d",4))
```

Question: What is the type of `d` ?

Answer: `int BST.dict`

ML will print the internal tree representation of `d`.
We could have hidden that by using opaque ascription:

```
structure BST :> DICT = ...
```

(Reminder: Despite seeing the internals a client cannot pattern match on the constructors since they are not declared in the signature.)

Now consider:

```
val look = BST.lookup d
val x = look "e"
val y = look "a"
```

Question: What are the bindings for `x` and `y` ?

Interacting with BST

```
val d = BST.insert(
          BST.insert(
           BST.insert(
            BST.insert(BST.empty,("a",1)),
                                ("b",2)),
                               ("c",3)),
                              ("d",4))
```

Question: What is the type of `d` ?

Answer: `int BST.dict`

ML will print the internal tree representation of `d`.
We could have hidden that by using opaque ascription:

```
structure BST :> DICT = ...
```

(Reminder: Despite seeing the internals a client cannot pattern match on the constructors since they are not declared in the signature.)

Now consider:

```
val look = BST.lookup d
val x = look "e"
val y = look "a"
```

Question: What are the bindings for `x` and `y` ?

Answer: `[NONE/x, (SOME 1)/y]`

Two Comments

Here are two other ways to define the dictionary type within **BST**, producing an **`'a dict`** equivalent to what we wrote before:

```
datatype 'a tree = Empty | Node of 'a tree * 'a * 'a tree
type 'a dict = 'a entry tree

datatype 'a dict =
    Empty | Node of 'a dict * 'a entry * 'a dict
```

IF signature **`DICT`** had mentioned the constructors **`Empty`** and **`Node`**, then a client of **`BST`** could/would refer to them as **`BST.Empty`** and **`BST.Node`** (for instance in pattern-matching).

However, signature **`DICT`** does **not** mention these constructors, so a client of **`BST`** cannot refer to the constructors.
(Only inside the structure **`BST`** are the constructors accessible, and there directly as **`Empty`** and **`Node`**.)

That is all for today.

See you Thursday.

(We will discuss **functors**.)