A Fourth Look At F#

Type Definitions

- Predefined, but not primitive in F#:

```
type bool = true | false;;
```

- Type constructor for lists:

```
type 'element list = List.empty |
   :: of 'element * 'element list
```

Outline

- Enumerations
- Data constructors with parameters
- Type constructors with parameters
- Recursively defined type constructors

Programmer Defined Types

- New types can be defined using the keyword **`type`**
- These declarations define both:
  - *type constructors* for making new (possibly polymorphic) types
  - *data constructors* for making values of those new types

Example

```
type day = Mon | Tue | Wed | Thu | Fri | Sat | Sun;;
 type day = Fri | Mon | Sat | Sun | Thu | Tue | Wed

let isWeekDay x = not (x = Sat || x = Sun);;
  val isWeekDay = fn : day -> bool

isWeekDay Mon;;
  val it = true : bool

isWeekDay Sat;;
  val it = false : bool
```

- **`day`** is the new *type constructor* and **`Mon`**, **`Tue`**, etc. are the new *data constructors*
- Why "constructors"? In a moment we will see how both can have parameters…

No Parameters

```
type day = Mon | Tue | Wed | Thu | Fri | Sat | Sun;;
type day = Mon | Tue | Wed | Thu | Fri | Sat | Sun
```

- The *type constructor* **`day`** takes no parameters: it is not polymorphic, there is only one **`day`** type
- The *data constructors* **`Mon`**, **`Tue`**, etc. take no parameters: they are constant values of the **`day`** type
- Capitalize the names of data constructors

Strict Typing

```
type flip = Heads | Tails;;
  type flip = Heads | Tails
let isHeads x = (x = Heads);;
  val isHeads = fn : flip -> bool
isHeads Tails;;
  val it = false : bool
isHeads Mon;;
error FS0001: This expression was expected to have
type
    flip
but here has type
    day
```

```
enum flip {Heads, Tails};
```

- F# strictly enforces operations on new types
- Unlike C **`enum`**, no implementation details are exposed to the programmer. The following defines type *flip* with **`Heads=0, Tails=1.`**

Data Constructors In Patterns

```
let isWeekday d =
  match d with
     | Sat | Sun -> false
     | _ -> true;;
```

- *Data constructors* can be used in patterns
- In this simple case, similar to constants
- More general case where constructor contains a typed data item, considered next

Outline

- Enumerations
- Data constructors with parameters
- Type constructors with parameters
- Recursively defined type constructors
- Farewell to F#

Datatype Parameters - Wrappers

- *Parameterize* any type to a data constructor, using the keyword **`of`**:

```
type IR = I of int | R of double;;
```

- In effect, such a constructor is a *wrapper* that contains a data item of the given type


```
let a = R 38.5;;
```

```
let b = I 4;;
```

```
R 5;    error
```

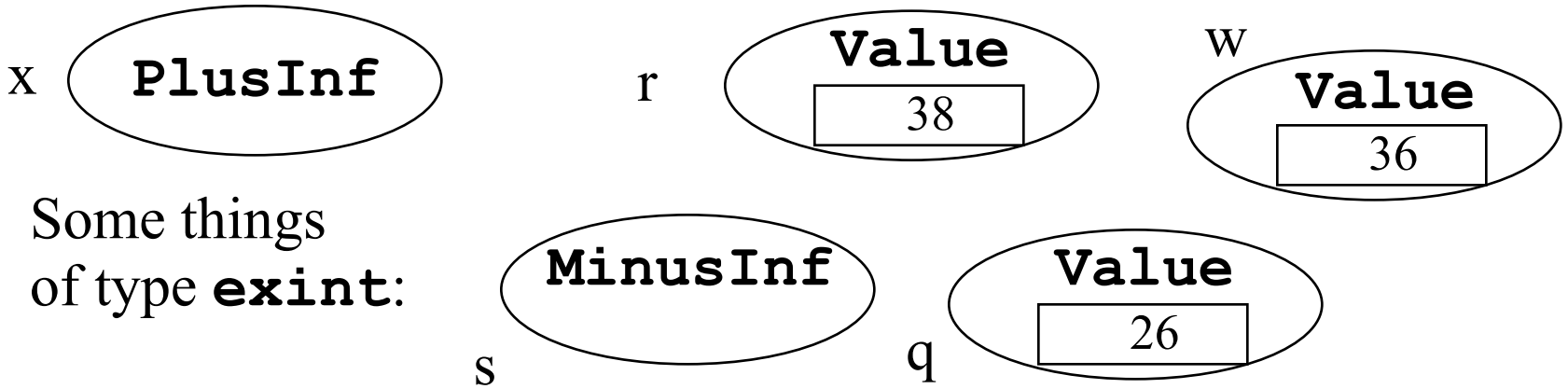
```
I 3.14; error
```

Wrappers

- You can add a parameter of any type to a data constructor, using the keyword **`of`**:

```
type exint = Value of int | PlusInf | MinusInf;;
```

- In effect, such a constructor is a wrapper that contains a data item of the given type



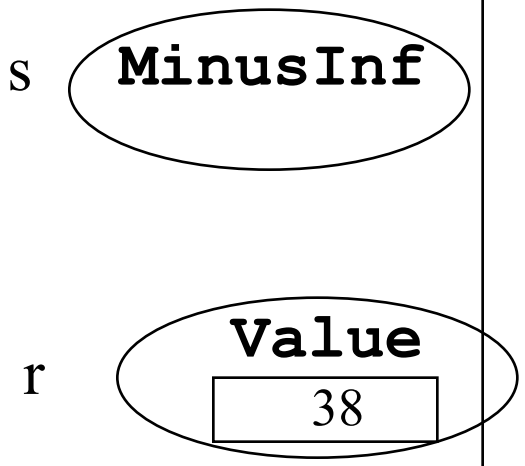
Wrapper Example

```
type exint = Value of int | PlusInf | MinusInf;;
  type exint = MinusInf | PlusInf | Value of int

let s = MinusInf;;
  val s : exint = MinusInf

Value;;
  val it : arg0:int -> exint

let r = Value 38;;
  val r : exint = Value 38
```



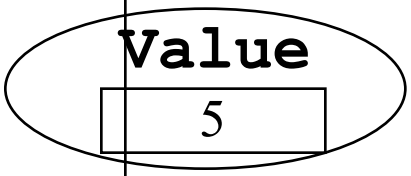
- **`Value`** is a data constructor that takes a parameter: the value of the **`int`** to store
- Signature is a function that takes an **`int`** and returns an **`exint`** containing that **`int`**

A **Value** Is **extint** Not **int**

```
let x = Value 5;;
  val x = Value 5 : exint

x+x;;
 error FS0001: The type 'exint' does
 not support the operator '+'

(Value 5) + (Value 5);;
 error FS0001: The type 'exint' does
 not support the operator '+'
```



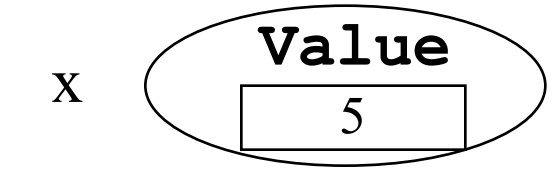
- **Value 5** is an **exint**
- It is not an **int**, though it contains one
- The **int** accessed by pattern matching

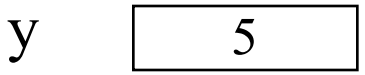
Patterns With Data Constructors

```
let x = Value 5;;
  val x : exint = Value 5

let (Value y) = x;;
  Warning: Incomplete pattern matches...

  val y : int = 5
```





- To recover a data constructor's parameters, use pattern matching
- So **`Value`** is no ordinary function: ordinary functions can't be pattern-matched this way

Exhaustive Patterns

```
let x = Value 5;
let s =
  match x with
    | PlusInf -> "infinity"
    | MinusInf -> "-infinity"
    | Value y -> y.ToString();;
val s : string = "5"
```

An **exint** can be a **PlusInf**, a **MinusInf**, or a **Value**

```
let x = Value 5;;
let s =
  match x with
    | PlusInf -> 9999999999999
    | MinusInf -> -9999999999999
    | Value y -> y;;
  val s : int = 5
```

Pattern-Matching Function

```
let square x =
  match x with
     | PlusInf -> PlusInf
     | MinusInf -> PlusInf
     | Value x -> Value (x*x);;
val square : x:exint -> exint

square MinusInf;;
  val it : exint = PlusInf

square (Value 3);;
  val it : exint = Value 9
```

Pattern-matching function definitions are especially important with programmer defined datatypes

Example - extint Factorial

```
let mult (Value x) (Value y) = Value (x*y);;
  val mult : exint -> exint -> exint

mult (Value 5) (Value 4);;
  val it : exint = Value 20

let rec fact x =
  match x with
    | PlusInf -> PlusInf
    | MinusInf -> MinusInf
    | Value 0 -> Value 1
    | Value n -> mult
          (fact (Value (n-1))) (Value n);;
  val fact : x:exint -> exint

fact (Value 5);;
  val it : exint = Value 120
```

Exercise 1

1. Give a datatype declaration for type constructor *color* with data constructor *Red, Green, Yellow, Blue, White*.
2. Define function *primary* that returns *true* when the single parameter is red, green or blue; else return *false*.
3. Give a parameterized *type* declaration for type *number* of integer (`AnInt`) or real (`AReal`).
4. **`type exint = Value of int | PlusInf | MinusInf;;`**

```
let x = AReal 5.0;;
  val x : number = aReal 5.0
let y = AnInt 4;;
  val y : number = anInt 4
```

Exercise 1 Continued

4. Define the function *plus* of type:

   `number->number->number`

   that adds two numbers, coercing *int* to *real* when mixed parameters. Recall *real(3)* converts *int* type to *real*.

```
let x = AReal 5.0;;
let y = AnInt 4;;

plus x y;;
     val it :number = AReal 9.0
plus y y;;
     val it : number = AnInt 8
```

Outline

- Enumerations
- Data constructors with parameters
- Type constructors with parameters
- Recursively defined type constructors
- Farewell to F#

Type Constructors With Parameters

- Type constructors without parameters:

```
type EXINT = NOINT | SOMEINT of int;;
```

- Type constructors with parameters:

```
type 'a option = NONE | SOME of 'a;;
```

- `'a` is the type parameter; `option` is datatype name
- Type constructor parameters are *type variables*
- Result: a new *polymorphic* type bound at execution time

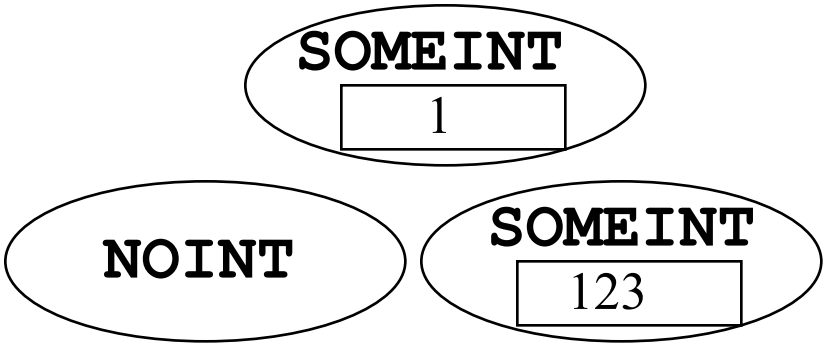
Type Constructors With Parameters

- Type constructors without parameters:

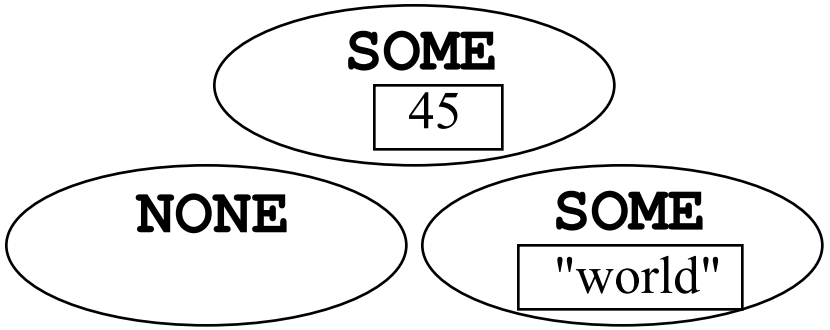
```
type EXINT = NOINT | SOMEINT of int;;
```

- Type constructors with parameters:

```
type 'a option = NONE | SOME of 'a;;
```



Values of type **EXINT**



Values of type
string option and
int option

Parameter Before Name

```
SOME;;
  val it : arg0:'a -> 'a option
SOME 4;;
  val it : int option = SOME 4
SOME 1.2;;
  val it : float option = SOME 1.2
SOME "pig";;
  val it : string option = SOME "pig"
SOME [1;2;3];;
  val it : int list option = SOME [1; 2; 3]
```

- Type constructor parameter **'a** before constructor name:

  type 'a option = NONE | SOME of 'a;;

- Types **'a option** and **int option**, just like **'a list** and **int list.**

Uses of **`option`** datatype

- Predefined type constructor in F#: None, Some
- Used by predefined functions (or your own) when the result is not always defined

```
let optdiv a b =
    if b = 0 then NONE else SOME (a / b);;
  val optdiv : a:int -> b:int -> int option

optdiv 7 2;;
  val it : int option = SOME 3

optdiv 7 0;;
  val it : int option = NONE
```

Uses of **`option`** datatype

```
type 'a option = NONE | SOME of 'a;;

type exint = Value of int
           | PlusInf
           | MinusInf;;

let x = SOME (Value 3);;
  val x : exint option = SOME (Value 3)
SOME (SOME 5);;
  val it : int option option = SOME (SOME 5)
SOME (SOME 5, 3.14);;
  val it : (int option * float) option =
                    SOME (SOME 5, 3.14)
```

Longer Example: **bunch**

```
type 'x bunch = One of 'x
      | Group of 'x list;;
```

- An **'x bunch** is either one of type **'x**, or a list of type **'x**
- As usual, F# infers types:

```
One 1.0;;
  val it : float bunch = One 1.0

Group [true,false];;
  val it : (bool * bool) bunch =
           Group [(true, false)]
```

Example: Polymorphism

```
type 'x bunch = One of 'x
      | Group of 'x list;;

let size x =
  match x with
     | One _ -> 1
     | Group x -> List.length x;;
       val size : x:'a bunch -> int

size (One 1.0);;
       val it : int = 1

size (Group [true; false]);;
       val it : int = 2
```

- F# can infer **`bunch`** types, but does not always have to resolve them, just as with **`list`** types

Example: No Polymorphism

```
type 'x bunch = One of 'x
      | Group of 'x list;;
let rec sum x =
  match x with
      | One x -> x
      | Group [] -> 0
      | Group (h::t) -> h + (sum (Group t));;
     val sum : x:int bunch -> int
sum (One 5);;
     val it : int = 5
sum (Group [1,2,3]);;
     val it : int = 6
```

- Applied the **+** operator to the list elements
- F# determines **`int bunch`** the parameter type

```
type bunch = One of int
     | Group of int list;;
```

a) **One 3;;**
b) **One "Hi";;**
c) **Group [1;2;3];;**
d) **Group ["Hi"; "OK"];;**

Exercise 2
Which are valid?

```
type 'x bunch = One of 'x
     | Group of 'x list;;
```

e) **One 3;;**
f) **One "Hi";;**
g) **Group [1;2;3];;**
h) **Group ["Hi"; 3];;**
i) **Group [One 1; One 2];;**
j) **Group [One "Hi"; One 3];;**
k) **Group [Group [1;2]; Group[3;4]];;**
l) **Group [Group ["Hi"; "OK"], Group[1;2]];;**

Exercise 2

What is wrong if anything?

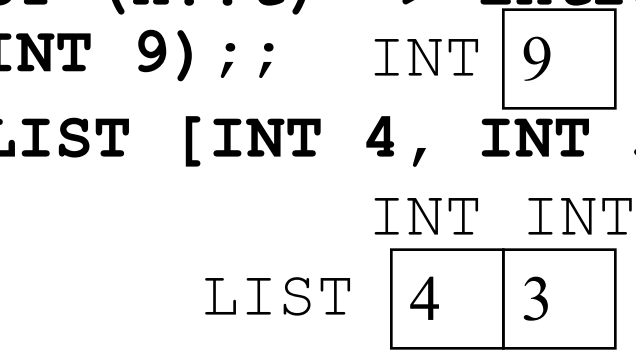
```
type intbunch = One of int
              | Group of int list;;
let rec sum i =
    match i with
      | One i -> i
      | Group [] -> 0
      | Group (h::t) -> (sum (One h))+(sum (Group t));;
```

Are these the same?

```
type intbunch = One of int
              | Group of int list;;
Group [1;2;3];;
type intbunch = One of int
              | Group of intbunch list;;
Group [One 1; One 2; One 3];;
```

Exercise 2

```
type intnest = INT of int
             | LIST of intnest list;;
let rec intret x =
  match x with
     | INT i -> i
     | LIST [a] -> intret a
     | LIST (h::t) -> intret (LIST t);;
intret (INT 9);;
intret (LIST [INT 4, INT 3]);;
```



Write function *sum* of type *intnest* that sums all integers in an *intnest*.

Hint: match (INT x) and (LIST L). For (LIST L), L is a list with *hd L* an INT and *tl L* a LIST. Return 0 when L is null, otherwise sum the INT and LIST parts.

Outline

- Enumerations
- Data constructors with parameters
- Type constructors with parameters
- Recursively defined type constructors
- Farewell to F#

Recursively Defined Type Constructors

The type constructor being defined may be used in its own data constructors:

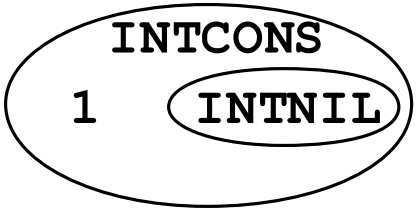
```
type intlist = INTNIL |
               INTCONS of int * intlist;;
```

let x = INTNIL;;

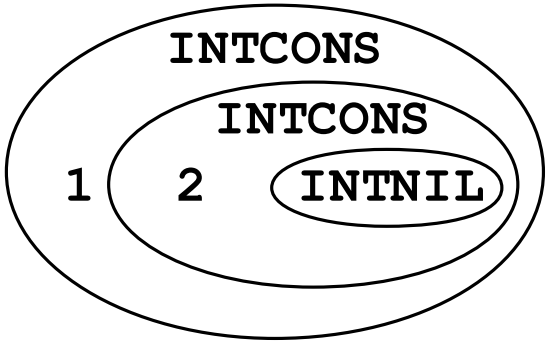
the empty list

Some values of type **`intlist`**:

INTCONS(1,INTNIL)



the list [1]



the list [1,2]

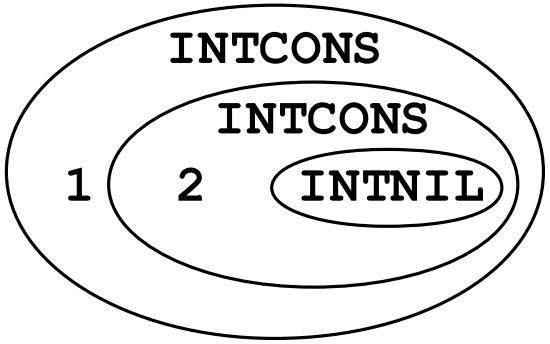
INTCONS(1, INTCONS(2,INTNIL))

Construction as a Tuple

```
INTNIL;;
val it : intlist = INTNIL
INTCONS (1,INTNIL);;
val it : intlist = INTCONS (1,INTNIL)
INTCONS (1, INTCONS (2, INTNIL));;
val it : intlist = INTCONS (1,INTCONS (2,INTNIL))
```

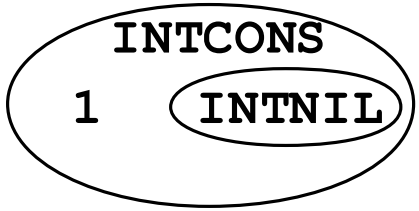
let x = INTNIL;

the empty list



the list [1,2]

INTCONS(1,INTNIL)



the list [1]

INTCONS(1, INTCONS(2,INTNIL))

An **intlist** Length Function

```
type intlist = INTNIL |
                    INTCONS of int * intlist;;
let rec intlistLength L =
  match L with
     | INTNIL -> 0
     | INTCONS (_, tail) -> 1 +
            (intlistLength tail);;

let x = INTCONS (3, INTCONS (4, INTNIL));

intListLength x;
     val it : int = 2
```

intlist similar to predefined lists except use of **INTNIL** or **int * intlist** tuples for each element

Parametric List Type

```
type 'element mylist = NIL
  | CONS of 'element * 'element mylist;;
```

- A parametric list type using nested tuples, similar to predefined **`list`**
- **`'element`** is polymorphic type parameter
- Allows a *mylist* datatype of polymorphic types

```
CONS(1.0, NIL);;
val it : float mylist = CONS (1.0,NIL)
CONS (1, CONS (2, NIL));;
val it : int mylist = CONS (1,CONS (2,NIL))
```

Some **mylist** Functions

```
type 'element mylist =
   | NIL
   | CONS of 'element * 'element mylist;;
```

```
let rec myLength x =
   match x with
      | NIL -> 0
      | CONS (_, t) -> 1 + (myLength t);;
let rec addup x =
   match x with
      | NIL -> 0
      | CONS (h, t) -> h + (addup t);;
addup (CONS(1, CONS(2, NIL)));;  returns 3
```

- Similar to predefined **list** type constructor
- Of course, to add up a list could use **foldBack**…

A **foldBack** For **mylist**

```
let rec myFoldBack f c x =
  match x with
     | NIL -> c
     | CONS (a, b) -> f a (myFoldBack f c b);;
```

- Definition of a function like **foldBack** that works on **'a mylist**
- Can now add up an **int mylist x** with:

```
myFoldBack (fun a b -> a+b) 0 x;;
```

- One remaining difference: **::** is an operator and **CONS** is not

Exercise 3

- Write *rev* function to reverse a *mylist*.
- Function *snoc* places an **'el** at end of a *mylist*.

```
type 'el mylist = NIL
     | CONS of 'el * 'el mylist;;
let c = CONS(1,CONS(2,CONS(3, NIL)));;
 val c : int mylist = CONS (1,CONS (2,CONS (3,NIL)))
let rec snoc a x =
  match x with
     | NIL -> CONS (a, NIL)
     | CONS (h, t) -> CONS (h, (snoc a t));;
snoc 4 (CONS(1,CONS(2,NIL)));;
 val it : int mylist = CONS (1,CONS (2,CONS
(4,NIL)))
rev (CONS(1,CONS(2,NIL)));
  val it : int mylist = CONS(2,CONS(1, NIL))
```

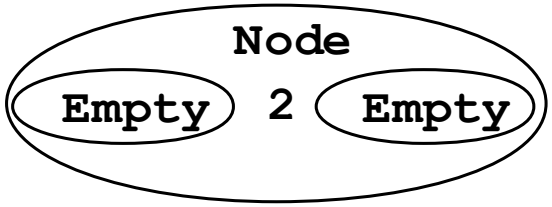
```
struct tree {
      tree *left, *right;
      char data;
};
void f(tree *t) {
      if(t != NULL) {
            cout << t->data;
            f(t->left);
            f(t->right);
      }
}
void main ( void ){
      tree  l1={NULL,NULL,'1'},
            l2={NULL,NULL,'2'},
            r2={NULL,NULL,'3'},
            r1={&l2,&r2,'*'},
            t={&l1,&r1,'-'};

            f(&t);
}
```

Polymorphic Binary Tree

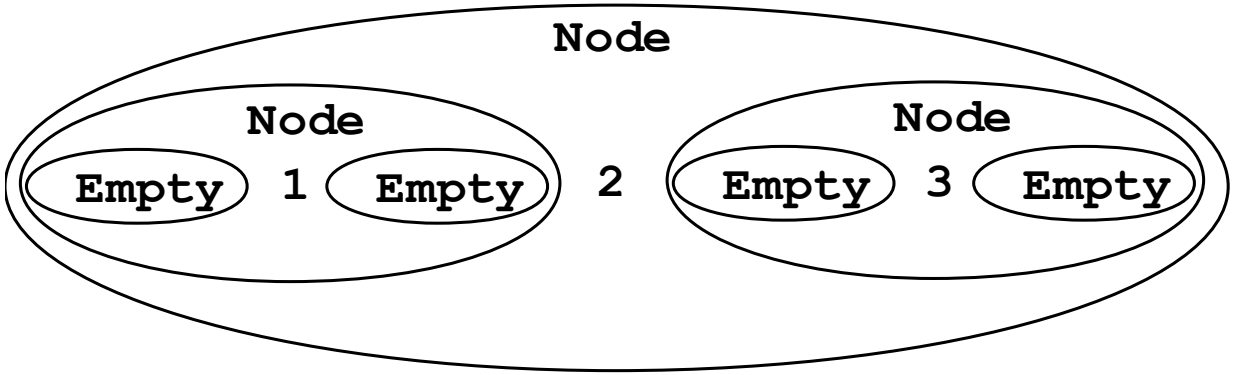
```
type 'data tree =
    | Empty
    | Node of 'data tree * 'data * 'data tree;;
```

the empty tree

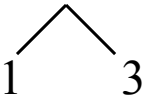


the tree 2

Some values of type **`int tree`**:

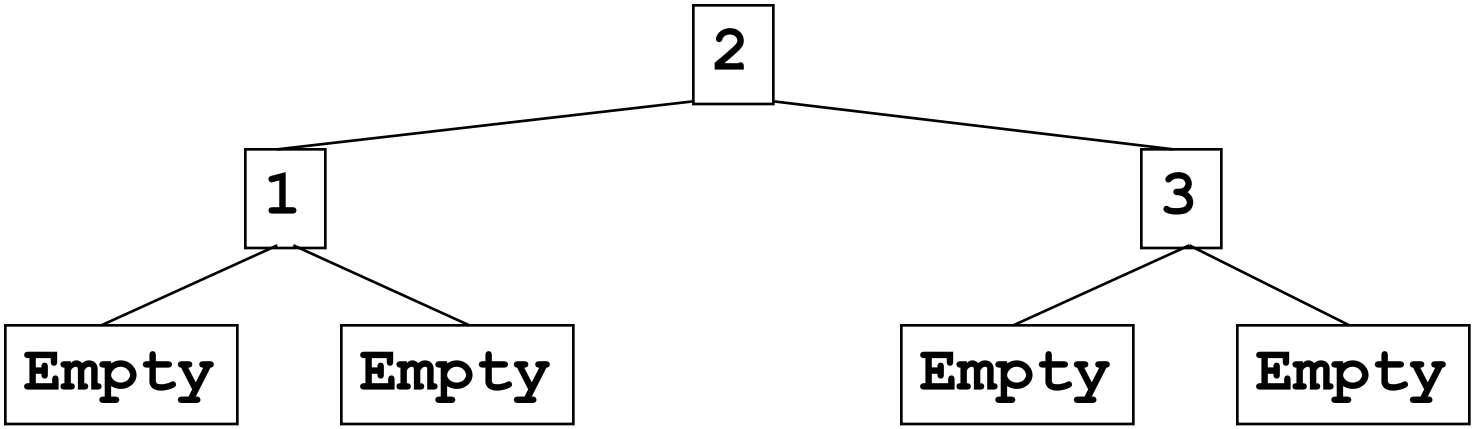


the tree 2



Constructing Those Values

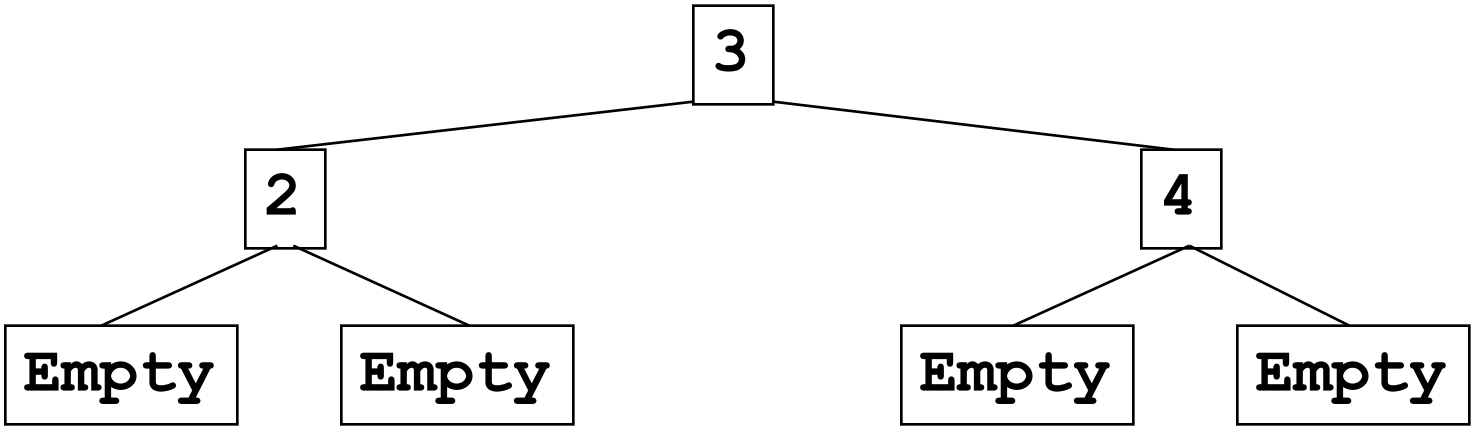
```
let treeEmpty = Empty;;
  val treeEmpty = Empty : 'a tree
let tree2 = Node (Empty,2,Empty);;
  val tree2 = Node (Empty,2,Empty) : int tree
let tree123 = Node (Node (Empty,1,Empty),
                    2,
                    Node (Empty,3,Empty));;
```



Increment All Elements

```
let rec incall x =
  match x with
     | Empty -> Empty
     | Node (x,y,z) ->
         Node (incall x, (y+1), (incall z));;
```

```
incall tree123;;
val it : int tree = Node (Node (Empty,2,Empty),
                          3,
                          Node (Empty,4,Empty))
```



Add Up The Elements

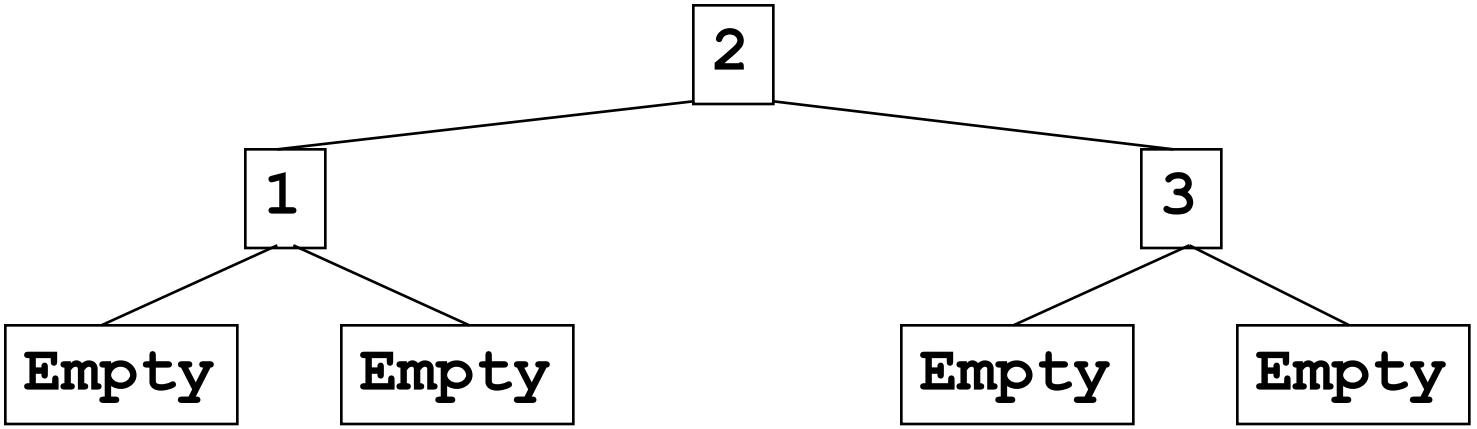
```
let rec sumall x =
  match x with
    | Empty -> 0
    | Node (x,y,z) ->
       sumall x + y + sumall z;;
```

```
sumall tree123;
val it : int = 6
```

Convert To List (Polymorphic)

```
let rec listall x =
  match x with
    | Empty -> []
    | Node (x,y,z) ->
        listall x @ y :: listall z;;
```

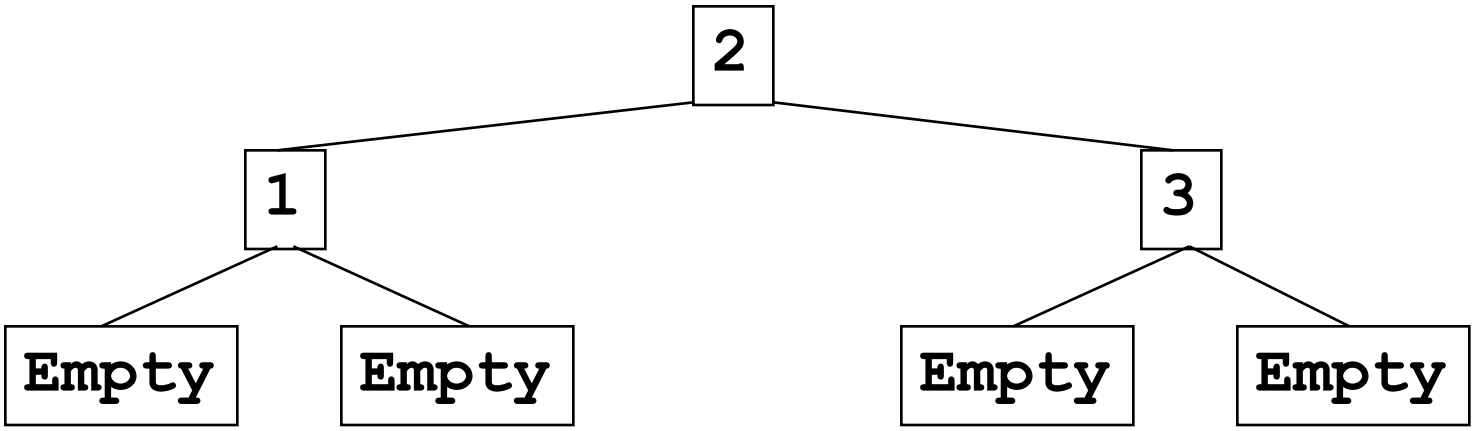
```
listall tree123;;
  val it : int list = [1; 2; 3]
```



Inorder Tree Search

```
let rec isintree x L =
  match L with
     | Empty -> false
     | Node (l,y,r) when x=y -> true
     | Node (l,y,r) when (isintree x l) -> true
     | Node (l,y,r) -> isintree x r;;
```

```
isintree 4 tree123;
val it : bool = false
isintree 3 tree123;
val it : bool = true
```



Insert in order

```
let rec insert a x =
  match x with
     | Empty -> Node (Empty, a, Empty)
     | Node (l,d,r) when a < d -> Node(insert a l,d,r)
     | Node (l,d,r) -> Node(l,d,insert a r);;

let t = insert 5 Empty;;
let t = insert 4 t;;
let t = insert 7 t;;
let t = insert 6 t;;
val t =  Node(
            Node(Empty,4,Empty),
            5,
            Node(
                  Node(Empty,6,Empty),
                  7,
                  Empty))
```

Exercise 4

- Write function *inorder* to return a string consisting of the concatenation of tree node data values when traversed in-order.
- Modify **`sumall`** function in minor way.
- Use *5.ToString()* to convert integer 5 to a string.
- "a" + "b" + "c" concatenates to "abc".

```
inorder Node(Node(Empty,4,Empty),5,Node(Node(Empty,6,Empty),7,Empty));;
val it : string = "4567"
```

```
let sumall x =
  match x with
    | Empty -> 0
    | Node (x,y,z) -> sumall x + y + (sumall z);;
```

Outline

- Enumerations
- Data constructors with parameters
- Type constructors with parameters
- Recursively defined type constructors
- Farewell to F#

That's All For Now

- That’s all the F# we will see for a while
- There is, of course, a lot more
- A few words about the parts we skipped:
  - records (like tuples with named fields)
  - arrays, with elements that can be altered
  - references, for values that can be altered
  - exception handling in depth

More Parts We Skipped

- support for encapsulation and data hiding:
  - structures: collections of datatypes, functions, etc.
  - signatures: interfaces for structures
  - functors: like functions that operate on structures, allowing type variables and other things to be instantiated across a whole structure

More Parts We Skipped

- API: the standard basis
  - predefined functions, types, etc.
  - Some at the top level but most in structures: **`String.replicate`**, **`cos`** `operator`, **`List.nth`**, etc.

More Parts We Skipped

- F# pipelines
- Modules for organizing code
- Object oriented F#

- Other dialects besides F#
  - OCaml
  - Standard ML (SML)
  - Concurrent ML (CML) extensions

Functional Languages

- F# supports a function-oriented style of programming
- If you like that style, there are many other languages to explore, like Lisp and Haskell