This topic covers some important higher-order functions: function composition and pipelining. It then introduces type definitions and discriminated unions.
Function pipelining

- The following infix polymorphic function simply applies it’s second argument to it’s first.
  
  ```
  let (|>) x f = f x
  val (|>) : 'a -> ('a -> 'b) -> 'b
  // parens ARE needed
  ```

- Is this useful? At first it seems not: 3 |> f is the same as f 3

- But, it is useful for replacing code like:

  ```
  let maxWordsOnLine str =
  List.max (map length (map words (lines str)))
  ```

  which has too many parens. Instead, using |

  ```
  let maxWordsOnLine str =
  lines str |> map words |> map length |> List.max
  ```

- This is called pipelining, and is a common style in F#
  
  - Values flow through the functions in the pipeline from left to right.
  - The functions in the pipeline are often formed by partial applications.
  - At the start of the pipeline is the value to begin with (lines str).
Function composition

- More complex pipelines can be written over many lines.
  
  ```
  let maxWordsOnLine str =
  lines str
  |> map words
  |> map length
  |> List.max
  ```

- Pipelining can make code clearer, but is just a syntactic change.
  - It also helps type inference sometimes with overloading.

- Sometimes it is useful to construct a “pipe” without giving the input.
  - This is exactly the mathematical concept of function composition, \((g \circ f)\).

    ```
    let (>>) f g = fun x -> g (f x) // \(f \gg g = g \circ f\) in mathematics.
    val (>>) : ('a -> 'b) -> ('b -> 'c) -> ('a -> 'c) // Last parens are unnecessary.
    ```

- Then we can rewrite our example, keeping str implicit:
  
  ```
  let maxWordsOnLine = lines >> map words >> map length >> List.max
  ```

- Composition is a powerful way to build functions from simpler ones
  - Particularly when passing the function to another function. E.g.,
    
    ```
    map (lines >> map words >> map length >> List.max) strs
    ```
Types in Functional Programming

- Languages can be classified as dynamically typed or statically typed.
  - Statically-typed languages type check prior to program execution.
  - Dynamically-typed languages do not.

- Modern functional languages (like F#, ML, Haskell) are often statically typed, with sophisticated type systems, however LISP and related languages are dynamically typed.

- Static typing is particularly powerful and useful in a functional language because pure functions only map from inputs to outputs, and types can easily describe these inputs and outputs.

- Static typing allows a majority errors to be quickly caught and fixed.

- It also allows much of the basic structure of the program to be made explicit by the programmer.

- Dynamic types appear more flexible, but in fact they can be emulated via “universal types”.
  - And, static types cannot be emulated in dynamically-typed languages.
Types: Perspective

- Types in functional languages have a long history: concepts like parametric polymorphism have been studied in typed \(\lambda\)-calculus since the 1930s.
- Type systems for other languages have been highly influenced by functional programming
  - Notable recently: the addition of parametric polymorphism/generics to Java and C#.
  - Also covariance and contravariance (in C# 4.0)
- Much current research in programming languages concerns even more sophisticated type systems.
- We will focus only on what is in F#, and we’ll come back later to details such as type inference.
Type definitions

- F# allows new types to be defined with `type` declarations.
- The simplest kind of type definition is a type abbreviation.
  ```fsharp
type lineNumber = int // Makes the kind of "int" clear
type indexEntry = wordstr * (lineNumber list) // abbreviates
  ```
- Abbreviations can also be parametric/generic.
  ```fsharp
type 'a table = 'a list list // Old syntax, avoid multi-params
type funAndInv<'a,'b> = ('a->'b) * ('b->'a) // New syntax
  ```
- Another kind of type definition is a discriminated union.
  - Lists are an example: the parametric list type constructor is defined by:
    ```fsharp
type 'a list =
| []
| [] of 'a * ('a list) // 2nd parens only for clarity.
```
Discriminated unions

- The general form for a union definition is:
  ```
  type unionName < 'a, 'b, ... > =  // Any no. of type parameters, optionally.
  | Cons1  // Some constructors have no argument type.
  | Cons2 of type2 // And some do - and are applied to this type.
  | ...    // There can be many constructors (also called discriminators)
  ```

- This introduces a new type called `unionName`.
  - If there are type parameters, `unionName` is actually a type constructor, which can be used to construct types like `unionName < type1, type2, ... >`

- It also introduces constructors `Cons1, Cons2, ...`
  - These can be used to construct values with the new type.
  - If the constructor has an argument type (via `of`) it must be applied to a value of this type.

- Constructors are also be used in patterns to “take apart” a value of the new type.
  ```
  match u with
  | Cons1 -> expr1
  | Cons2 argPat2 -> expr2 // argPat2 must match `type2`
  | ...
  ```
Examples

- Usually constructors start with a capital, while types do not.
- The simplest kind of union is just an enumeration, as found in languages like Java, C#, C.
  ```
  type direction = North | South | East | West
  let unitVec = function
      | North -> (0.0, 1.0)
      | South -> (0.0, -1.0)
      | East -> (1.0, 0.0)
      | West -> (-1.0, 0.0)
  ```

  /// Calculate the displacement of a sequence of unit steps.
  /// Note: fst, snd return the first and second parts of a pair
  let addSteps steps =
      let vecs = (List.map unitVec steps)
      (List.sum (List.map fst vecs), List.sum (List.map snd vecs))

  addSteps [North; East; North; North]  // yields (1.0, 3.0)
Value carrying constructors

- Having constructors carry values via “of” is extremely useful.

```ocaml
type Route = int
type Make = string
type Model = string
type Transport =
  | Car of Make * Model
  | Bicycle
  | Bus of Route

let averageSpeed = function
  | Car ("Ferrari", _) -> 55
  | Car _ -> 35
  | Bicycle -> 16
  | Bus _ -> 24

averageSpeed (Car ("Ferrari", "F40"))  // yields 55
averageSpeed (Car ("Mazda", "323"))  // yields 35
```
The option type

- The F# core library includes the following type:
  ```fsharp
type 'a option =
  | Some of 'a
  | None
  ```
- This is used to optionally pass/return values.
  - Many OO and imperative languages use null references instead.
  - Nulls are messy because any result or argument is potentially null.
  - Using option types makes it clear where there might be no value.
- Example - note `Option.map f` maps None to None, otherwise uses f
  ```fsharp
  let safeDiv x = function
    0 -> None
    | y -> Some(x/y)
  val safeDiv : int -> int -> int option
  let myFunc z = Option.map ((+)5) (safeDiv 3 z)
  val myFunc : int -> int option
  (myFunc 2, myFunc 0)
  val it : int option * int option = (Some 6, None)
  ```
Recursive types: binary trees

- Discriminated unions may be recursive (like lists).
  ```ocaml
type tree = Leaf | Node of tree * int * tree  (* left-child, key, right-child *)
```

- To use these as binary search trees for storing sets of integers:
  ```ocaml
  let rec insert newKey = function
    | Leaf -> Node (Leaf, newKey, Leaf)  (* Create a new Node *)
    | Node (left, nodeKey, right) ->
      if newKey < nodeKey
        then Node (insert newKey left, nodeKey, right)
      else Node (left, nodeKey, insert newKey right)
  
  let rec isElementOf searchKey = function
    | Leaf -> false  (* Not found *)
    | Node (left, nodeKey, right) ->
      if nodeKey = searchKey then true  (* Found *)
      elif nodeKey < searchKey then isElementOf searchKey left
      else isElementOf searchKey right
  ```
Generalized binary trees

- We can generalize the type for trees to take parameter, instead of always using int.
  
  ```plaintext
  type 'a tree = Leaf
               | Node of ('a tree) * 'a * ('a tree) //Parens unecc.
  ```

- Each node has a left child-tree, an 'a key and a right child-tree.

- The functions don’t need to be changed, just their types.
  ```plaintext
  val insert : 'a -> 'a tree -> 'a tree
  val isElementOf : 'a -> 'a tree -> bool
  ```

- However, we may not want the built in equality and less-than - instead we could use function abstraction, just like with qsort.

- This demonstrates some of the advantages of FP:
  - The code is very natural and elegant.
  - We have a very general and useful data structure with just a little code.
  - Insertion is functional: it does not modify the tree it is given - instead it creates a new tree, rebuilding the nodes that it passes through.
  - We can construct a number of trees to represent sets, and have them sharing sub-trees, and do insertions without other trees being affected.
Comparison with objects

- Discriminated unions are very much NOT object-oriented.
  - Some OO purists dislike them (sometimes quite strongly).
- With objects, instead you would create an interface and then have one implementing class for each constructor.
  
```java
class Node implements Tree {
    Node left, right;
    int key;
    public Node(Node l, int k, Node r) { left = l, key = k; right = r; }
    public Node insert(int k) { .... }
    public bool isElementOf(int k) { ... }
}
```

- The code for each function is distributed into the classes.
- OO works well when you have a relatively fixed set of operations, but you may have additional implementing classes/constructors.
- But, if you have a relatively fixed set of classes/constructors, but expect additional operations, then discriminated unions are better.