Advanced F#

Here we cover some of the defining features of the F# programming language.

Algebraic data types and pattern matching

Algebraic data types (ADTs) and pattern matching are two features first widely used in the Hope[34]) programming language, which was developed concurrently with ML. ADTs and pattern matching are like chocolate and peanut butter: better together[35] (this commercial is totally ridiculous and worth the 30 seconds of your time.)

Algebraic data types are a way of concisely express hierarchies of types *without inheritance*. Inheritance is a feature from *object-oriented programming*, which we will discuss later in the semester. If you already happen to know what inheritance is, you can think of ADTs as its functional equivalent. As you will see, they are in many ways much more elegant and easy to reason about, although there is a big tradeoff when it comes to large-scale software engineering projects.

Let's create a data type that represents a small set of animals. We'd like, at various points in our program, to be able to work generically with animals, and then at other points, to be able to work specifically with specific animals, like ducks and geese.

```
type Animal =
| Duck
| Goose
| Cow
| Human
```

The above is an algebraic data type. In F#, we call this kind of type definition a *discriminated union*. In other ML variants, these are sometimes called *union types* or *sum types*. These terms mean the same thing.

What is the meaning of `Duck` or `Cow`? They are, in fact, constructors. So if we want a `Duck`, we type `Duck`. Let's try it in `dotnet fsi`:

[34] https://en.wikipedia.org/wiki/Hope_(programming_language

[35] https://www.youtube.com/watch?v=hHPY5yoINMA

```
> let donald = Duck;;
val donald : Animal = Duck
```

Likewise, if we want a `Cow`, we type:

```
> let ephelia = Cow;;
val ephelia : Animal = Cow
```

Notice that the types of `donald` and `ephelia` are, quite specifically, `Animals`.

Let's now make a function that takes an `Animal` and does the "right thing" depending on the kind of animal.

```
let squeeze a =
    match a with
    | Duck -> "quack!"
    | Goose -> "honk!"
    | Cow -> "meeeeph!"
    | Human -> "WHY ARE YOU SQUEEZING MEEEEEE????"
```

The `match ... with` expression tells F# that we want to perform the appropriate thing using *pattern matching*. Pattern matching is a feature of many functional programming languages that let you concisely express conditional logic.

So if we `squeeze ephelia`, the function does the right thing:

```
> squeeze ephelia;;
val it : string = "meeeeph!"
```

Pattern matching is always type-safe. Suppose we forgot to put in the case for `Human`. The F# compiler will report that we missed a case.

```
    match a with
  --------^

warning FS0025: Incomplete pattern matches on this expression. For example,
the value 'Human' may indicate a case not covered by the pattern(s).
```

Pattern matching also lets us deal concisely with cases that don't matter to us. For example, imagine that all we really care about is squeezing `Cows`. We could write:

```
let squeeze a =
  match a with
  | Cow -> "meeeeph!"
  | _ -> "complaint"
```

The `_` indicates a *default case* that will only be exercised by an `Animal`

that is not a `Cow`.

ADTs that store data

The ADTs we've seen so for are limited in their usefulness because they don't store any data. We can extend them to store any data that we want.

```
type LinkedList<'a> =
| Node of 'a * LinkedList<'a>
| Empty
```

The above is a complete definition for a linked list. Expressions of the form `'a * 'b` are *tuples*. So a `'a * LinkedList` is a 2-tuple that stores data of type `'a` on the left and a reference to a `LinkedList<'a>` on the right. You can have tuples of any arity in F#.

Also, observe that, we can also write types recursively. A `Node` stores a reference to a `LinkedList`[36].

Let's write a `prettyprint` function that, when given a `LinkedList<'a>`, prints it out all pretty. Here is an imaginary example in `dotnet fsi`:

```
> let mylist = Node ("a", Node("b", Node("c", Empty)));;
val mylist: LinkedList<string> = Node ("a", Node ("b", Node ("c", Empty)))
> prettyprint mylist;;
val it: string = "[a, b, c]"
```

Below is a reasonable first attempt at making our fantasy come true:

```
let rec prettyprint ll =
  match ll with
  | Node(data, ll') -> data.ToString() + ", " + (prettyprint ll')
  | Empty -> ""
```

Each case in our `match` expression, known as a *pattern guard*, ensures that `ll` has the form specified on the left side before executing the right side. When a case matches, data is *bound* to the given variables, in this case, `data` and `ll'`. For example, the data stored in the given `Node` is bound to the variable `data`; the tail of the list is bound to the variable `ll'`.

Pattern guards are composed from *deconstructors*, which have the same syntax and are essentially the inverse of *constructors*. For example, you can construct a tuple and deconstruct it using the same syntax.

[36] If you feel comfortable with our notion of algebraic data types so far, great! See if you can modify our `LinkedList<'a>` type to represent a binary tree. After all, a binary tree really is just a linked list with an extra reference...

```
> let tup = (1, "hi");;
val tup : int * string = (1, "hi")

> let (a,b) = tup;;
val b : string = "hi"
val a : int = 1
```

Anyway, let's try out our `prettyprint` function:

```
> prettyprint mylist;;
val it: string = "a, b, c, "
```

Hmm. Not quite. But with a little massaging, we can get this to work. One approach is to define a helper function. Note that functional programming languages let use define functions inside of function definitions[37].

[37] If this sounds crazy to you ask yourself: why not? Eventually you'll see that disallowing this behavior is actually the crazy design choice. Nested function definitions are very useful.

```
let prettyprint ll =
  let rec pp ll =
    match ll with
    | Node(data, Empty) -> data.ToString()
    | Node(data, ll') -> data.ToString() + ", " + (pp ll')
    | Empty -> ""
  "[" + (pp ll) + "]"
```

Notice that we made a number of changes to our original function. First, we defined a helper function, `pp`, inside of our main `prettyprint` function. Second, we called `pp` at the very end of `prettyprint`, and we surround whatever it returns with square brackets. Finally, because we want to omit the trailing comma, we have a special case: we check to see that a node is the last node so that we can construct a special string for that case. We have to check that case first, because the second case in the above pattern match is more general. And finally, we kept the `Empty` case at the end. You might be wondering why we do that given that we check for emptiness in our first case. Well, consider the following kind of list.

```
> let mylist2 = Empty;;
val mylist2: LinkedList<'a>

> prettyprint mylist2;;
val it: string = "[]"
```

In short, the above definition works for empty lists too.

I personally find the use of ADTs and pattern matching an refreshing way to build software. Many concepts in computer science are simple

and elegant in theory but difficult to implement in code in practice. F# and other ML languages give us the tools to express simple concepts simply.

Limitations

There are two things to note about the above definitions. First, note that I am *not* able to write the following definition:

```
type Thing =
| A of char
| B of string
| C of A * B
```

This definition produces the following error:

```
  | C of A * B
  -------^

error FS0039: The type 'A' is not defined.
```

Although ADTs admit recursive definitions, that's not what we're looking at here. Importantly, cases in an ADT are *not data types*. Instead, `A`, `B`, and `C` are *constructors* for the one type called `Thing`. So we cannot refer to `A` or `B` or `C` in our definition of `Thing`. We could instead write the following:

```
type Thing =
| A of char
| B of string
| C of Thing * Thing
```

but that's not quite the same since that lets a `C` store a `C` and maybe that's not what you want. To get that, we'd have to write,

```
type A = char
type B = string
type C = A * B
type Thing =
  | A of A
  | B of B
  | C of C
```

and that all works although it's a tad inelegant.

A second limitation is that we had to use the `rec` keyword somewhere in our `pp` definition.

```
let rec pp ll =
```

Leaving out the `rec` produces the following error:

```
      | Node(data, ll') -> data.ToString() + ", " + (pp ll')
  ------------------------------------------------------^^

error FS0039: The value or constructor 'pp' is not defined.
```

This error occurs because F# strictly adheres to the rules of the lambda calculus[38]. We will talk more about the lambda calculus in the future.

[38] For the curious, the reason is that F# uses a "desugaring" approach to interpret `let` expressions. An expression of the form `let z = U in V` desugars to the expression `(λz.V)U`. Well, if `z` is a function name, it is not available to use in the expression `U` because `U` is outside the lambda abstraction. Adding the `rec` keyword tells F# that the name of the function should be available *inside* the function body. In effect, a recursive `let` is different than a regular `let`.

Equality and type checking

Type checking is the process of ensuring that the use of values is logically consistent. F# can sometimes check *at compile time* whether two values will ever be equal without having to inspect values.

F#'s type system is a little different than the kind you've seen before in Java and C. Java and C use a form of types called a *nominal type system*. In essence, a nominal type is simply a label. *Nominal type checking*, therefore, boils down to ensuring that these labels match. For example, the following C program fails to type check:

```
int i = 1;
char *c = i;
```

because `i` is an `int` and `c` is a `char *`, although because C is weakly typed, this is only a warning and not a fatal compilation error:

```
program.c:3:9: warning: incompatible integer to pointer conversion
initializing 'char *' with an expression of type 'int'
      [-Wint-conversion]
  char *c = i;
        ^   ~
```

Java adds extra expressiveness to C's nominal type system which allows it to express *subtypes* (i.e., inheritance), which is why we say that it has a *nominal type system with subtyping*. Subtyping essentially means that, under some circumstances, some labels can be substituted for others. Java also enforces types *strongly*, so the Java equivalent to the above C program is a fatal compilation error.

F# uses a system of *structural typing*. In this case, equality is more than just checking labels, although labels are at the "bottom" of the type system. Structural type checking means that the type checker must prove, inductively, that two expressions are equivalent because they yield the

same *structural type*.

An example helps to clarify. For example,

```
> let a = 1;;
val a : int = 1

> let b = 1;;
val b : int = 1

> a = b;;
val it : bool = true
```

In this case, not only are the types for `a` and `b` equal, so are the values (note that F# denotes equality using the = symbol, not the == symbol; "not equal" is denoted with <>).

Neither `a` nor `b` have any "structure" and so the base case for structural type checking is nominal: we simply check that the labels match (`int` equals `int`).

But how do we check the equality of something more complicated?

The following checks equality inductively:

```
> let c = (1,"hi");;
val c : int * string = (1, "hi")

> let d = (1,"hi");;
val d : int * string = (1, "hi")

> c = d;;
val it : bool = true
```

To see that `(1,"hi")` equals `(1,"hi")`, we need to know the type of each expression. Both are 2-tuples. Thus, we know for two 2-tuples to be equal, we must check that both left sides are equal and that both right sides are equal (inductive step). Both left sides are `int` and `1` equals `1` (base case). Both right sides are `string` and `"hi"` equals `"hi"` (base case). Therefore, both 2-tuples are equal. Therefore `(1,"hi")` equals `(1,"hi")`.

Structrual type systems make equality comparisons very easy, and they extend to what are normally "opaque" types in other languages, like lists and arrays.

Lists

Lists are frequently utilized in functional programming. The original functional programming language, LISP, demonstrated that lists can be

used as a fundamental unit of composition for many more complicated data structures. F# also allows you to use lists in this manner, and because they are so important and widely-used, they are even easier to manipulate than in LISP. Although we were able to define a `LinkedList<'a>` type above, lists are so important to F# that it has special, built-in syntax to support them.

For example, the following lets us define a list using *list literal* syntax in F#:

```
let xs = [1; 2; 3; 4;]
```

We can also perform a variety of operations on lists easily:

```
> let xs = [1; 2; 3; 4;];;
val xs : int list = [1; 2; 3; 4]

> List.head xs;;
val it : int = 1

> List.tail xs;;
val it : int list = [2; 3; 4]

> let xs' = 0 :: xs;;
val xs' : int list = [0; 1; 2; 3; 4]

> List.length xs';;
val it : int = 5
> List.append xs' xs';;
val it : int list = [0; 1; 2; 3; 4; 0; 1; 2; 3; 4]

> List.rev xs';;
val it : int list = [4; 3; 2; 1; 0]

> List.sum xs;;
val it : int = 10

> List.map (fun x -> x - x) ;;
val it : (int list -> int list) = <fun:it@60-6>

> List.fold (fun acc x -> acc + x) 0 xs;;  // this is fold left
val it : int = 10
```

```
> List.foldBack (fun x acc -> acc + x.ToString()) xs "";; // fold right
val it : string = "4321"

> xs' = xs';;
val it : bool = true

> xs = xs';;
val it : bool = false

> let ys = [0; 1; 2; 3; 4];;
val ys : int list = [0; 1; 2; 3; 4]

> xs' = ys;;
val it : bool = true
```

Notice that we were able to compare two lists simply, using =. This is possible because of F#'s structural type system. We can even pattern-match on lists, which is incredibly useful for functions that recurse on lists:

```
let rec list_length xs =
  match xs with
  | []     -> 0
  | x::xs' -> 1 + list_length xs'
```

where `[]` or `nil` represents the empty list and `::` means *cons*.

```
> list_length xs;;
val it : int = 4
```

The complete documentation on F# lists is available online[39].

[39] https://docs.microsoft.com/en-us/dotnet/fsharp/language-reference/lists

Nonetheless, F# is a *pragmatic* language, and as a matter of pragmatism, we should probably recognize that lists have undesirable performance properties for many applications, particularly numerical computing. For many applications, richer data types are desirable. F# has these too.

Arrays

Lists have O(n) performance for random-access. In other words, in the worst case, the cost of accessing an element is the cost of traversing the entire list. Arrays have much better performance for random-access: O(1). In other words, the worst case performance for arrays is a constant time, or the amount of time it takes to execute a single operation.

The following lets us define an array using *array literal* syntax in F#:

```
let arr = [|1; 2; 3; 4;|]
```

We can also perform a variety of operations on arrays easily:

```
> let arr = [|1; 2; 3; 4;|];;
val arr : int [] = [|1; 2; 3; 4|]

> let i = arr.[3];;
val i : int = 4

> Array.length arr;;
val it : int = 4

> Array.rev arr;;
val it : int [] = [|4; 3; 2; 1|]

> Array.filter (fun x -> x > 2) arr;;
val it : int [] = [|3; 4|]

> arr.[0..1];;
val it : int [] = [|1; 2|]

> arr.[2..];;
val it : int [] = [|3; 4|]

> arr.[..2];;
val it : int [] = [|1; 2; 3|]

> Array.init 10 (fun i -> i * i);;
val it : int [] = [|0; 1; 4; 9; 16; 25; 36; 49; 64; 81|]

> Array.sum (Array.init 10 (fun i -> i * i));;
val it : int = 285

> Array.map (fun x -> x + 1) arr;;
val it : int [] = [|2; 3; 4; 5|]

> Array.fold (fun acc x -> acc + x) 0 arr;;  // this is fold left
val it : int = 10

> Array.foldBack (fun x acc -> acc + x) arr 0;; // this is fold right
val it : int = 10

> let arr2 =  [|1; 2; 3; 4;|];;
val arr2 : int [] = [|1; 2; 3; 4|]

> arr = arr2;;
val it : bool = true
```

Notice that we were able to compare two arrays simply, using =. This

is possible because of F#'s structural type system.

The complete documentation on F# arrays is available online[40].

[40] https://docs.microsoft.com/en-us/dotnet/fsharp/language-reference/arrays

Other types

F# has many other types, including `Tuple`, `Sequence`, `Map`, `Option`, classes, interfaces, and abstract classes, and many others. The latter three types are object oriented features of F#. Please do not use classes, interfaces, or abstract classes unless I ask you to do so.

The complete F# language reference is available online[41]. I encourage you to refer to it often as it is thorough and quite easy to read.

[41] https://docs.microsoft.com/en-us/dotnet/fsharp/language-reference/

Conditionals

`if/else` expressions look a bit like their counterparts in Python:

```
if x > 0 then
  1
else
  2
```

Of course, since conditionals are *expressions* in F#, you can also do neat tricks like use them to conditionally assign values, much like how you might use the ternary operator (`a ? b : c`) in C:

```
let y = if x > 0 then
          1
        else
          2
```

Indentation is important for conditionals. Note that the body of the true and false clauses must be indented past the start of the `if` expression.

Loops

While F# has looping constructs, `for` and `while`, you should not use them in this class. Instead, you should use `map`, `fold`, and recursion instead.

The following table provides a handy guide for deciding which construct to use in F# when your brain tells you that you need to use a loop.

Recursion instead of `while`

Let's start with recursion to solve problems. One nice thing about a `while` loop is that you don't need to know how many times your pro-

Problem	Java	F#
Do something until a condition is satisfied	`while`	Write a recursive function and make the condition a base case.
Do something for a bounded number of times	`for`	Write a recursive function and make the bound a base case.
Convert every element of a collection into something else	`for` or `foreach`	`map`
Accumulate a value	`for`, `foreach`, or `while`	`fold`
Convert a recursive data structure into another data structure	Recursive function	Recursive function or `fold`

gram needs to repeat itself until it is done computing a value. For example, when doing a membership query in an unsorted linked list in C (e.g., "is 4 in the list?"), you can use a while loop to simply continue getting the next element until it is found.

```
bool contains(listnode *xs, int x) {
  while (xs != null) {
    if (xs->head == x) {
      return true;
    } else {
      xs = xs->tail;
    }
  }
  return false;
}
```

What characterizes problems like this is that there is no obvious bound on the loop before we find the element we're looking for. We solve this class of problems in functional programming with recursion.[42]

[42] Remember that recursive function definitions must use `let rec`.

```
let rec contains xs x =
  match xs with
  | []    -> false
  | y::ys ->
    if x = y then
      true
    else
      contains ys x
```

This particular solution pattern matches on the list to deconstruct it into a head (`y`) and a tail (`ys`), but you could also explicitly call `List.head` and `List.tail` if you wanted.

You might argue that this is a silly example because we know that there can be no more comparisons than the length of the list. That's true. But there are other problems where you actually don't know at all, and the same pattern applies. For example, what if we wanted to run a little experiment: how many times do we need to flip a coin before a heads comes up? The outcome is determined by the laws of probability. It *probably* won't take many coin flips for heads to come up—in fact, we

know that we have a 50% chance on the very first try—but it *could* take a very long time. There's a nonzero probability that it could take a trillion coin flips.

```
let rec num_tries_until_match(r: System.Random)(n: int)(i: int)(e: int) =
  let rn = r.Next(n)
  if rn = e then
    i
  else
    num_tries_until_match r n (i + 1) e
```

Where `n` is the number of "sides" of our coin (or die, or whatever), `i` is the count so far, and `e` is the value we're looking for. Let's say that, when `n = 2`, then when `e = 0` that's heads and when `e = 1` that's tails. If we call this a few times, you can see that the answer can vary quite a bit:

```
> let r = System.Random();;
val r : System.Random

> num_tries_until_match r 2 1 0;;
val it : int = 3

> num_tries_until_match r 2 1 0;;
val it : int = 1

> num_tries_until_match r 2 1 0;;
val it : int = 2

> num_tries_until_match r 2 1 0;;
val it : int = 4

> num_tries_until_match r 2 1 0;;
val it : int = 5
```

Map

On the other hand, many problems do exhibit bounded iteration. For example, if you want to convert every number in a list into a string, you know exactly how many you need to do: whatever the length of the list is. This is what `map` is for.

```
> List.map (fun x -> x.ToString()) [11;22;33;44];;
val it : string list = ["11"; "22"; "33"; "44"]
```

Note that the type of the output need not be the type of the input. Here, we convert `int` values into `string` values.

Let's look at the function definition for `List.map`:

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

We'll step through, bit by bit. The parens tell us that the first argument is a `'a -> 'b`. Right away, because it contains an `->`, we know that it is a function. What kind of function? A function from *some unknown type* `'a` to *some other unknown type* `'b`. Now, it is entirely plausible that `'a` and `'b` are the same type (e.g., `int`). But what this type definition tells us is that they don't have to be. We call this first parameter the *mapping function*. It "converts" or "maps" a given value to another value.

Next parameter, `'a list`. This parameter is the set of inputs, specifically a list in this case.

Finally, the last parameter, `'b list`. This last parameter is the type of the output. This should make intuitive sense, because if you call the mapping function on each element of a list of `'a`s, you will get a get a list of `'b`s.

Map has a few interesting properties. For starters, if the mapping function `'a -> 'b` always terminates, then `map` always terminates. For anybody who has ever accidentally written a loop that doesn't terminate, this is a nice feature! Second, notice that each call of the mapping function only depends on *one* of the values in the list. In fact, each call of the function is totally independent from every other call. Problems that have this structure are called *embarassingly parallel*, because it takes no work at all to actually do the computation in parallel. In fact, F# makes this almost absurdly easy to do, although we cannot do it with lists (for hopefully obvious reasons... think about this a bit if you don't know why):

```
> Array.map (fun x -> x.ToString()) [|11;22;33;44|];;
val it : string [] = [|"11"; "22"; "33"; "44"|]

> Array.Parallel.map (fun x -> x.ToString()) [|11;22;33;44|];;
val it : string [] = [|"11"; "22"; "33"; "44"|]
```

Note that it is not always faster to do things in parallel! In this case, because our list is small, the serial version is actually faster.

```
> let timeit (f: int[] -> string[])(input: int[]) =
-   let sw = System.Diagnostics.Stopwatch.StartNew()
-   f input |> ignore
-   sw.ElapsedTicks
- ;;
val timeit : f:(int [] -> string []) -> input:int [] -> int64

> let input = [|11;22;33;44|];;
val input : int [] = [|11; 22; 33; 44|]

> timeit (fun xs -> Array.map (fun x -> x.ToString()) xs) input;;
val it : int64 = 5657L

> timeit (fun xs -> Array.Parallel.map (fun x -> x.ToString()) xs) input;;
val it : int64 = 7296L
```

Fold

Folding is useful anytime you want to accumulate a value. This is, of course, useful in scenarios like the following. Here, we multiply the numbers 1 through 7 together.

```
> let product = List.fold (fun acc x -> acc * x) 1 [1;2;3;4;5;6;7];;
val product : int = 5040
```

Let's look at `fold`'s type signature:

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

The first parameter is a function of two values, `'a` and `'b`, which returns a `'a`. In our `fold` above, that function, which we call the *folding function*, multiplies those two values together. The thing to remember is that the first parameter to our folding function is the *accumulator*: it's where we store the result of the operation from one iteration of `fold` to the next. The second parameter to the folding function is the element, which is taken from the list (we'll get to that in a second).

The second parameter to `fold` is the *initial value of the accumulator*. Since we want to multiply, in this case, we set it to `1`. The last parameter to `fold` is the list. This is where the folding function's `'b` parameter comes from. Finally, the return value is a `'a`, because that's the type of our accumulator. When there are no more elements in the list, the accumulator is simply returned.

Here's another example, where we convert strings into numbers and sum them:

```
> List.fold (fun acc x -> acc + int x) 0 ["1";"2";"3";"4"];;
val it : int = 10
```

`fold` is an incredibly powerful function. It is not limited to lists. You can fold arrays, trees, graphs, etc. In fact, you can implement `map` using fold (try challenging yourself to figure it out). But a consequence of this power, which is the ability to make each step of a computation interdependent, is that it is no longer embarassingly parallel. A great deal of research in computer science has been devoted toward finding just enough structure in certain problems to parallelize special cases of `fold`. In general, it's impossible.

The particular `fold` we discuss here is technically "fold left." The term "left" refers to the side of the input sequence from which we take elements. Folding "right" takes elements from the end instead of the beginning.

By the way, in LISP, the original functional programming language, `fold` is called `reduce`. When you pair mapping and folding together, you get a form of computation called `map-reduce`. This is where the term came from that inspired Google's MapReduce framework, which is a platform for fault-tolerant, massively parallel computation.

Other important features

F# has essentially every feature that a modern language like Java has, so there are too many features to discuss in this reading. However, there are a few that are worth a mention.

Raising Exceptions

You can define an exception in F# like:

```
exception MyError of string
```

and you can throw it like:

```
raise (MyError("Error message"))
```

F# also has a "lightweight" syntax that I use frequently:

```
failwith "something bad happened!"
```

Runtime exceptions are more useful in F# than they are in ordinary

languages. Because F# is functional and strongly-typed, sometimes the type checker gets in the way. One very useful trick that you can use as your "stub out" a method is to use `failwith` to make the type checker temporarily go away.

```
let rec prettyprint e =
  match e with
  | Variable(c) -> c.ToString()
  | Abstraction(v,e') -> failwith "TODO1"
  | Application(e', e'') ->  failwith "TODO2"
```

I use this trick frequently, although you should be aware that doing so trades a compile-time error for a runtime one!

Catching Exceptions

You can catch exceptionsing using F#'s `try ... with` syntax,

```
try
  prettyprint (Abstraction('x', Variable('x')))
with
| MyError(msg) ->  "oops: " + msg
```

which returns the `string` value `"oops: TODO1"`.

Option types

Like Java and C#, you can store `null` in values.

```
let x = null
```

However, the use of `null` is now widely regarded as a mistake in computer science. Tony Hoare (winner of the Turing Award), and inventor of `null`, called it a "billion-dollar mistake":

> I call it my billion-dollar mistake. It was the invention of the null reference in 1965. At that time, I was designing the first comprehensive type system for references in an object oriented language (ALGOL W). My goal was to ensure that all use of references should be absolutely safe, with checking performed automatically by the compiler. But I couldn't resist the temptation to put in a null reference, simply because it was so easy to implement. This has led to innumerable errors, vulnerabilities, and system crashes, which have probably caused a billion dollars of pain and damage in the last forty years.

Instead, in F#, we prefer the type-safe `option` type. Let's write a function that uses `option`.

```
let onedivx x =
  if x = 0.0 then
    None
  else
    Some (1.0 / x)
```

Now when we use `onedivx`, we get back an `option` type:

```
> onedivx 0.0;;
val it : float option = None

> onedivx 1.1;;
val it : float option = Some 0.9090909091
```

`option` gives us a way to signal the failure of a computation in a type-safe manner. In fact, the type-checker *forces* us to deal with the error:

```
> (onedivx 0.0) + 2.2;;

  (onedivx 0.0) + 2.2;;
  ----------------^^^

error FS0001: The type 'float' does not match the type 'float option'
```

To use the return value, we must "unwrap" it first, using pattern matching.

```
> match onedivx 0.0 with
- | Some res -> printfn "%f" res
- | None     -> printfn "Oh, dear!"
- ;;
Oh, dear!
val it : unit = ()
```

Forward pipe

Forward pipe, `|>`, is my single favorite feature of F#. It allows you to build sophisticated data-processing pipelines easily. `|>` passes the result of the left side to the function on the right side. For example, instead of writing:

```
List.map (fun x -> x + 1) [1;2;3;4]
```

you can write:

```
[1;2;3;4] |> List.map (fun x -> x + 1)
```

I find the latter easier to read. But the benefit really becomes apparent when you need to do multiple operations.

```
> [1;2;3;4]
- |> List.map (fun x -> x + 1)
- |> List.zip [5;6;7;8]
- |> List.filter (fun (_,y) -> y > 3)
- |> List.fold (fun acc (x,y) -> acc + x * y) 0
- ;;
val it : int = 68
```

Cool, huh?