Monads & F# computation expressions

How to put people off a good simple idea!

- Wrap it in mathematical notation:
  - Monad
  - Monoid
  - Endofunctor
- What any pure mathematician who had studied category theory would use
  - Category theory – "like set theory but so general it has no use at all!"
  - Except it turns out that category theory is useful for computer scientists...
- Useful for what?

How not to learn about monads

"A monad in T is a monoid in the category of endofunctors of T with $\times$ defined as the composition of endofunctors and unit element defined by the identity endofunctor"

Programmers like patterns

- Identify a pattern in your program
- Turn it into
  - A template
  - A function
  - An abstract data type
  - An inheritance hierarchy
- This lecture is about patterns everyone uses
  - Especially useful in functional programming

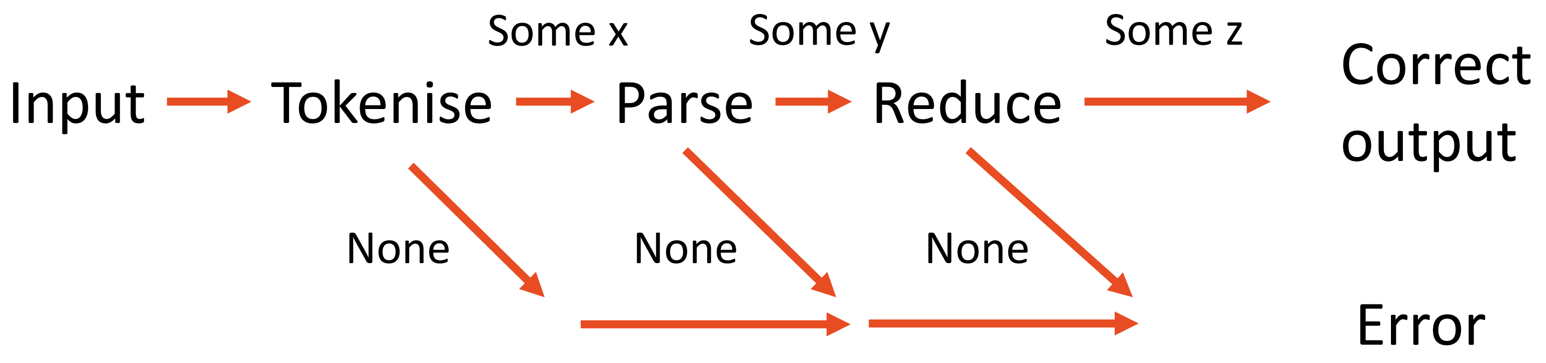
show how to control side effects in a functional way

- Key problem: how to control side effects!
- Understanding **monads** is understanding what side effects are, and how they combine!
- You don't need category theory to understand fully what *monads* are in the context of programming, so I'll give this as optional material at the end.

Option<T>

side effect problem

- "Pipeline with errors" problem
  - If any stage in pipeline fails we want execution to terminate with failure
  - Model this as Option<T_{out}>



Option.bind pattern

```
let Pipe =
    Some (Input())
    |> fun x -> if x = None then None else Tokenise x
    |> fun x -> if x = None then None else Parse x
    |> fun x -> if x = None then None else Reduce x
```

➢ `Option.bind f`

❖ *Lifts* **input** of function f to Option

```
let Option.bind (f: T -> S) (inp: Option<T>) : Option<S> =
    if inp = None then None else f inp
```

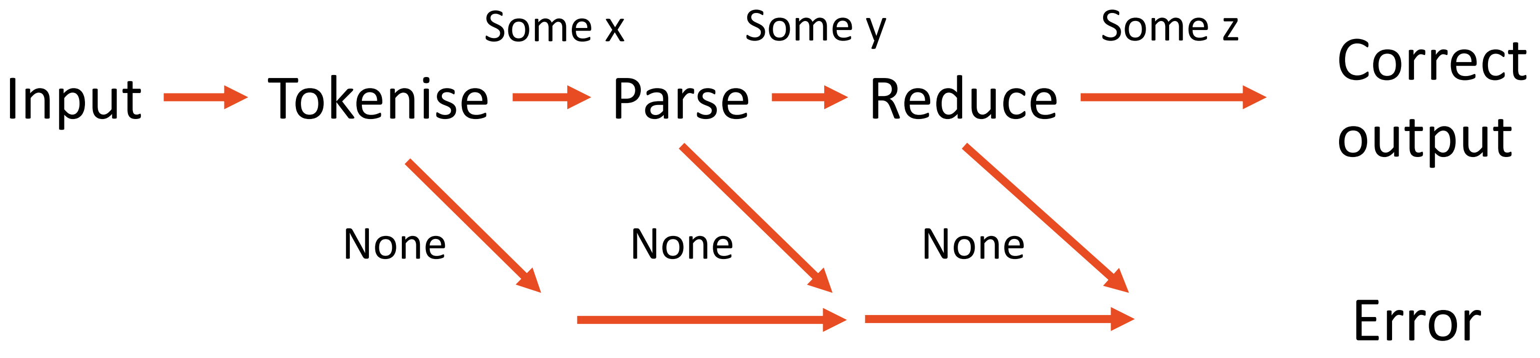
```
let Pipe1 =
    Option.Some (Input())
    |> Option.bind Tokenise
    |> Option.bind Parse
    |> Option.bind Reduce
```

This slide: so what is a better way to write a computer program that implements this diagram?

➢ `Option.map f`

❖ *Lifts* **input** and **output** of function f to Option

```
Option.map (f:T -> S) (inp: Option<T>) : Option<S> =
    if inp = None then None else Some (f inp)
```



Recap

- Consider a pipeline in the Option world where all stages have Option input and output.
  - This allows for normal and failure outputs to propagate through the pipeline
- This can be made using non-Option functions by transforming (lifting) each function with Option.bind or Option.map according to its type.

Normal world

f: T -> Option<S>

Option.bind

f': Option<T> -> Option<S>

g: T -> S

Option.map

g': Option<T> -> Option<S>

Option world

Printing without side-effects

- Use the same type of pattern as **Option.bind**
- The function f returns a tuple of 'real' output and string to be printed as a side-effect
- The string is appended to existing printout
- If `f1` returns `(_,"a")`, `f2` returns `(_,"b")`, `f3` returns `(_,"c")`, what is the value of `printed`?

Normal world

f: T -> S * string

`printBind`

f': T * string -> S * string

"Printing" world

```
let printBind (f: T -> S*string) (inp: T) =
    let t, p = inp
    let s', p' = f t
    (s', p + p' + "\n")
```

```
let (x, printed) =
    printBind f1 (init, "")
    |> printBind f2
    |> printBind f3
```

What is a side effect?

- x := 1 // has *side effect* of changing value of x to 1
  - Define a type `State` as a map with the names and the values of all variables
  - View assignment side-effect as a function that transforms input state to output state
  - All side effects can be modelled functionally as immutable functions: `State -> State`
- Sequence of assignments:
  - a1; a2; a3
- Equivalent to chained functions
  - `fun x -> a3' (a2' (a1'(x)))`
  - `a1' >> a2' >> a3' // composition`

`x := 1` ➡ `assign "x" 1`

```
type State = Map<string,int>

let assign (var:string) (value: int) =
    fun (s: State) -> Map.add var value s

let res =
   assign "x" 1      // x := 1
   >> assign "y" 2  // y := 2
```

Side effects have mathematical properties

- Model *any side effect* as a function *ss*: State->State where State includes the values of all things that can change due to side effects – such as mutable variables.
- *Set of all possible side effects S* (State->State) has binary operation >> (sequence or composition)
  - a >> b means "do a and then b" or s -> b(a(s)).
- Associativity of >>:
  - `(f >> g) >> h = f >> (g >> h)`
  - NB: `(a >> b) x = b (a (x))`
  - Proof:

    `((f >> g) >> h) x = h (g (f`
- Identity:

  id = fun x -> x

  f >> id = id >> f = f

(S, >>, id) forms a *monad*
Monads are structures made from functions which obey these (monoid) rules
You do not need to know this to use monads!

Computation expressions /?

- Adding extra stuff to functions, as with **option world** or the **printing world**, or a world in which assignment side-effects are modelled, is obviously useful.
- In these worlds other operations can be performed in parallel with the set of chained functions by adding extra "invisible" input and output to each function.
- **Computation expressions** provide a neat syntax to write code for such worlds.

```
// Define Option world C.E. Builder
type MaybeBuilder() =
    member this.Bind(m, f) = Option.bind f m
    member this.Return(x) = Some x

let maybe = new MaybeBuilder()

// C.E. syntax for Option world
maybe {
    let! tokens = tokenise input()
    let! ast = parse tokens
    let! result = reduce ast
    return result
    }
```

How let! works

- These steps show how the computation expression syntax is translated
- The "function called to continue" is called a *continuation*
- The continuation will normally include further expansion of `let!` lines
- There are other shorthand syntaxes in computation expressions in addition to `let!` see [MSDN]() for the full list with further explanation
- Other syntax (without !) works as normal.

we only want to evaluate cexp when the input is not None

Computation expression

```
maybe {
  let! a = aexp
  cexp
}
```

Equivalent

```
let! a = aexp in cexp
```

Definition of `let!`

```
maybe.Bind(aexp, (fun a -> cexpr))
```

Definition of `MaybeBuilder.Bind`

```
Option.Bind(fun a -> cexp, aexp)
```

Definition of `Option.Bind`

```
if aexp = None then None else (fun a -> cexp) aexp
```

Terminology you may read...

- Functions (T -> T) are called *endofunctors* of T (mappings from T *into* T)
  - Side effects are therefore by definition endofunctors of State
  - Every endofunctor of State is a (possible) side effect
- A *monad* is a special case of a *monoid* formed out of *endofunctors* and function composition
- The "side effect" *monoid* is therefore a *monad*.
- You may read it because:
  - Functional languages don't have side effects
  - BUT they can have functions *equivalent to* side effects
  - Think of the "side effect monad" as a way to encapsulate side effects inside a functional language
  - Ignore the mathematical language if you like

➢Why people talk about monads...

➢***Monads*** *are the most successful programming pattern arising in functional programming. Apart from their use to model a* ***generic notion of effect*** *they also serve as a convenient interface to* ***generalized notions of substitution****. Research in the area on the border between category theory and functional programming focusses on unveiling new programming and reasoning constructions similar to monads, such as comonads [1], arrows [2] and idioms (closed functors) [3]. Indeed, especially when working in an expressive and total language with dependent types, such as Agda [4],* ***we can exploit monads not only as a way to structure our programs but also their verification.***

1. *Uustalu, T., Vene, V.: Comonadic notions of computation. In Adam´ek, J., Kupke, C., eds.: Proc. of CMCS '08, ENTCS 203(5), Elsevier (2008) 263–284*
2. *2. Hughes, J.: Generalising monads to arrows. Sci. of Comput. Program. 37(1–3) (2000) 67–111*
3. *3. McBride, C., Paterson, R.: Applicative programming with effects. J. of Funct. Program. 18(1) (2008) 1–13*

Further Reading

- A long set of introductory tutorials on Computation expressions. These are well-written and have a lot of detail.
- For fun: in the spirit of Zen Koans:
  - Monolith