Explicit state is useful for modularity

- Before looking at data abstraction and object-oriented programming, let’s take a closer look at what explicit state is good for
- We say that a program (or system) is modular with respect to a given part if that part can be changed without changing the rest of the program
  - “part” = function, procedure, component, module, class, library, package, file, …
- We will show by means of an example that the use of explicit state allows us to make a program modular
  - This is not possible in the functional paradigm

A scenario (1)

- Once upon a time there were three developers, P, U1, and U2
- P has developed module M that implements two functions F and G
- U1 and U2 are both happy users of module M

```
fun {MF}  % Module definition
   fun {F ...}
      ⟨Definition of F⟩
   end
   fun {G ...}
      ⟨Definition of G⟩
   end
in 'export'(f:F g:G)
end
M = {MF} % Module instantiation
```

A scenario (2)

- One day, developer U2 writes an application that runs slowly because it does too much computation
- U2 would like to extend M to count the number of times F is called by the application
- U2 asks P to make this extension, but to keep it modular so that no programs have to be changed to use it

```
fun {MF}
   fun {F ...}
      ⟨Definition of F⟩
   end
   fun {G ...}
      ⟨Definition of G⟩
   end
in 'export'(f:F  g:G)
end
M = {MF}
```

Oops!

- This is impossible in the functional paradigm, because F does not remember what happened in previous calls: it cannot count its calls
  - The only solution is to change the interface of F by adding two arguments, F_{in} and F_{out}:
    fun {F … F_{in} F_{out}} F_{out}=F_{in}+1 … **end**
  - The rest of the program has to make sure that the F_{out} of each call to F is passed as F_{in} to the next call of F
- This means that M's interface has changed
- All M's users, even U1, have to change their programs
  - U1 is especially unhappy, since it makes a lot of extra work for nothing

Solution using a cell

- Create a cell when MF is called and increment it inside F
  - Because of static scope, the cell is hidden from the rest of the program: it is only visible inside M
- M's interface is extended without changing existing calls
  - M.f stays the same
  - A new function M.c appears that can safely be ignored
- P, U1, and U2 live happily ever after

```
fun {MF}
   X = {NewCell 0}
   fun {F ...}
      X:=@X+1
      ⟨Definition of F⟩
   end
   fun {G ...}
      ⟨Definition of G⟩
   end
   fun {Count} @X end
in 'export'(f:F g:G c:Count)
end
M = {MF}
```

Comparison

- Functional paradigm:
  - + A component never changes its behavior (if it is correct, it stays correct)
  - – Updating a component often means that its interface changes and therefore many other components must be updated
- Imperative paradigm:
  - + A component can be updated without changing its interface and so without changing the rest of the program (modularity)
  - – A component can change its behavior because of past calls (for example, it might break)

- Sometimes it is possible to combine both advantages
  - Use explicit state to manage updates, but make sure that the behavior of components does not change