Lec 12: Imperative Programming

CS220: Programming Principles

Sang Kil Cha

Facing the Wild World

Wild World

Wild World = Impure World

Attendance Check

Note:

1. This slide appears at random time during the class.
2. This link is only valid for a few minutes.
3. We don't accept late responses.

Side-Effects Everywhere!

1. Store some data to a file/DB.
2. Write characters to a monitor.
3. Send a network packet to another computer.
4. Modify memory contents.
5. ...

Printing

Hello World example revisited.

```
printfn "hello world" // returns what?
```

Every expression should result in a value, but what should be the value here?

The `unit` type

The `unit` type has only one value: `()`. It means “nothing”. It is indeed a dummy value. If a function returns `()`, that means the result of the function consists solely of side-effects.

Example: Monitor (Screen)

Let us try to emulate a computer monitor.

Monitor

```
type Monitor = {
  MaxColumn: int
  MaxRow: int
  Lines: string []
}
```

Functional Monitor

We should create a new monitor every time we output a line to it. An immutable monitor will be extremely costly.

Introducing States

We can view a monitor as an ***object***, which has a ***state*** that changes over time. In order to change a state, we need to introduce a new language expression: “assignment”.

Mutable Locations and Assignments

A mutable location is a part of the memory where we can assign a value. The keyword `mutable` in F# enables us to create such a location.

```
let mutable x = 1
x <- 2 // what is the return value of the assignment?
x // what's the value here?
x <- "xxx" // how about this?
```

Reference Cells

Variables can be defined as a reference cell, which is similar to mutable variables with different syntax.

```
let x = ref 1
x.Value <- 2 // assignment
x.Value // use .Value to dereference the value
x.Value <- 4 // assignment
```

Mutable Variables vs. Reference Cells

Reference cells can be aliased while mutable variables cannot.

```
let mutable x = 42
let y = x
x <- 1
y // what's the value here?
```

```
let x = ref 42
let y = x
x.Value <- 1
y.Value // what's the value here?
```

Mutable Location Should Always be Initialized

```
let mutable x
// Incomplete structured construct

let mutable x = 0 // Use some initial value
```

Unintialized Variables are Evil

C and C++ allow programmers to use uninitialized variables. This is a notorious source of bugs and security vulnerabilities.

F# does not allow uninitialized variables by its design.

Mutable Records

```
type MyRecord = {
  mutable Count: int
}

let r = { Count = 0 } // As usual
r.Count <- 1 // Update
```

Sequential Composition

We can now imperatively execute multiple side effects. That is, we can invoke multiple commands in a sequence: “do this, and do that”.

Note that, unit values can be naturally aligned in a sequential manner. To combine two sequential unit values, we use a semicolon (;), or simply put the next unit value in the ***next line***.

```
printfn "a"
printfn "b"
```

```
// In one line
printfn "a"; printfn "b"
```

Double Semicolons?

REPL uses dobule semicolons (`;;`) to indicate an end of input. It is specific to REPL, and should not be confused with sequential composition.

Sequential Composition (cont'd)

Can we sequentially lay out two non-unit expressions? How does the following evaluate?

```
1; 2
```

Ignoring Values

Q. Suppose there is a function f, that performs some actions with side effects, and returns either 0 or 1. And you want to ignore the return value of the function, and continue to call more functions in an ***imperative*** manner. How can we do so without encountering a warning message?

A. We ignore the return value ***explicitly***.

Ignore Function

The `ignore` function takes in anything and returns a unit.

```
ignore 1; 2 // returns 2

foo 123 |> ignore // A typical pattern.
```

Example: Counter Function

Let’s create a counter function, which takes in a unit, and returns how many times the function has been called.

```
val counter:  unit -> int
```

This function should return 1 when first called, 2 for the second call, and so on.

Example: Counter Function

Let's create a counter function, which takes in a unit, and returns how many times the function has been called.

```
val counter:  unit -> int
```

This function should return 1 when first called, 2 for the second call, and so on.

What if we want to create multiple counter functions?

In-Class Activity #12

Preparation

We are going to use the same git repository as before. Just in case you don't have it, clone the repository using the following command.

1. Clone the repository to your machine.
   ```
   - git clone https://github.com/KAIST-CS220/CS220-Main.git
   ```
2. Move in to the directory `CS220-Main/Activities`
   ```
   - cd CS220-Main
   - cd Activities
   ```

Problem 1

Modify the wrapper function `createCounter` for creating counter functions. And create two distinct counters. Give each counter the name `x` and `y`, respectively. Make an alias `z` of `y`, and observe why side effects can mess up the things.

Problem 2

Let's reimplement the counter function in a pure functional style. Write your `createPureCounter` function that takes in the following state type as input, and returns a modified state as output.

```
type CounterState = {
  Count: int
}

val count: CounterState -> CounterState
```

Imperative Data Structures

Higher-order Function: `iter`

Another frequently used higher-order function is `iter`. It takes in a function with side-effects, and applies the function to every element of the given data structure.

```
val iter:  ('a -> unit) -> 'a list -> unit
```

`List.iter`

`List.iter` function iterates every element of the given list by calling a function with side-effects. However, we can use this to fold values in the list with a mutable variable.

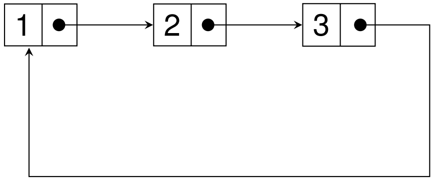
```
let mutable sum = 0
let f x = sum <- sum + x
List.iter f lst
```

String.iter

```
"hello world"
|> String.iter (printfn "%c")
```

Circular List

Circular list is a variation of list in which the first element points to the last element.



Circular List (Cont'd)

```
type MyList<'a> =
   | Nil
   | Cons of 'a * MyList<'a>

Cons (1, Cons (2, (Cons 3, ?)))
```

Arrays

Arrays look identical to Lists except for the syntax.

1. List: `[1; 2; 3]`
2. Array: `[|1; 2; 3|]`

Both Arrays and Lists have common higher-order functions: `fold`, `map`, etc.

Arrays are Mutable!

```
let arr = [| 1; 2; 3 |]
arr[0] <- 4 // what does this return?
arr[2] // random access is efficient
```

Why Random Accesses are Fast in Arrays?

Internally, arrays are totally different from Lists. Arrays are simply a sequence of n equally sized chunks. For example, `int[]` is a sequence of 4 byte chunks. Therefore, we should provide an initial size to create an array.

```
val Array.init:  int -> (int -> 'a) -> 'a[]
val Array.create:  int -> 'a -> 'a[]
```

More Imperative Data Structures

`System.Collections.Generic` namespace contains the followings:

1. `List<'a>`
2. `HashSet<'a>`
3. `Dictionary<'a, 'b>`
4. And many more.

Imperative List

```
let x = List<int> ()
x.Add 42 // returns unit
x.RemoveAt 0 // remove the first element
```

While and For Loops

```
while cond do body
```

While the `cond` expression is true, evaluate the `body` expression, which basically returns a unit (or a sequence of units).

While and For Loops

```
while cond do body
```

While the `cond` expression is true, evaluate the `body` expression, which basically returns a unit (or a sequence of units).

Loops are essentially a sequential composition of side-effects. And their conditions are naturally mutable, because otherwise, the loop will never end.

Writing a “`while`” function.

```
let rec w () =
  if cond then
    body
    w ()
  else ()
```

Is this tail recursive?

Conclusion

Summary

1. Programs written in imperative style are susceptible to bugs.
2. However, there are cases in practice where we need to deal with ***destructive updates***, which could be extremely useful when used properly.

Question?