Reflections on Worksheet 3

Lecture 4

F# Language Features: Coverage so far, or in last worksheet

Types

- `int` , `float` , `string` , `char`
- `bool`
  - `true` , `false`
- `unit`
  - `()`
- `T1 * T2` *(tuple or product type)*
  - `t1 , t2`
- `T1 -> T2` *(function type)*
- *polymorphic type variable*
  - `'a 'T1`
- *range (int or float)*
  - `3..10` , `0.5..2.5`
  - `1..2..9` , `1..-3..-5`
- *discriminated union* (sum type)
  - `| Age of int*int | Retired`
- **`Option`**
  - **`Some 22` , `None`**
- **`{Person:string ; Age:int}`** *(record type)*
  - **`{Person = "Me" ; Age = 10}`**
  - **`{rec1 with Age = 12}`**

Collections

- `List`
  - `[a ; b]` , `[]` , `::` , `[1..10]`
  - `lst.[10]` , `lst.[1..10]`
- `Array`
  - `[|a ; b|]` `x.[n]` `x.[a..b]`
- `Map`
  - `{"cat",1; "dog",10}`
  - `m.[k]`
- **`Set`**
- **`Seq`**

Language Constructs (Basic)

- *type definition*
  - **`type ...`**
  - **`type ... and ...`**
- let definition
  - `let`
  - `let rec ...`
  - `let rec ... and ...`
  - `let mutable`
- `fun ->`
- `if then else`
  - `&&` , `||` , `not`
  - `=` , `<>` , `>` , `<` , `>=` , `<=`
- `e1 ; e2`
  - `ignore`
- **`x <- 1`** *(assignment)*

Language Constructs

- *pipeline*: `|>`
  - Composition: `>>`
  - Backward pipeline: `<|`
- **`match ... with | p1 -> e1`**
  - **`when`** (guard expression)
  - List, tuple patterns
  - Active pattern
  - Partial active pattern
- *operators*
  - functionise: **`(op)`**
  - Custom definition
  - **`inline`**
- *exceptions*
  - **`failwithf`**
  - **`try with`**
  - **`try with finally`**
  - **`raise`**
  - **`reraise`**
- *computation expression*
- *seq comprehension*
- **`module`**
- **`interface`**
- **`class`**

Test-first Design

- Agile programming is a very successful paradigm in the OOP world
- Manage software projects intelligently
  - Program incrementally
  - Add tests incrementally for new features
  - Write tests as you go
- What Agile programming (can but need not) leave out
  - As systems become very large incremental techniques can't control complexity
  - Complex systems with distributed internal state become impossible to test
  - Difficult to identify problem structures and modularity
- Take home
  - Testing matters! Make sure you do it...
  - Testing alone is not enough

Design for testing

- Functional languages encourage smart programming
  - Think about problem structure and modularity (worksheet 4)
  - Write code in small composable blocks which can be independently tested
  - Put effort into making data structures follow problem domain
    - Reduces bugs
    - Makes code simpler
    - Makes testing easier
- Think hard about internal state "*moving parts*"
  - Is it really needed?
  - How does it interact with other bits of internal state?
  - Is this easily testable?

Key take-home for project work

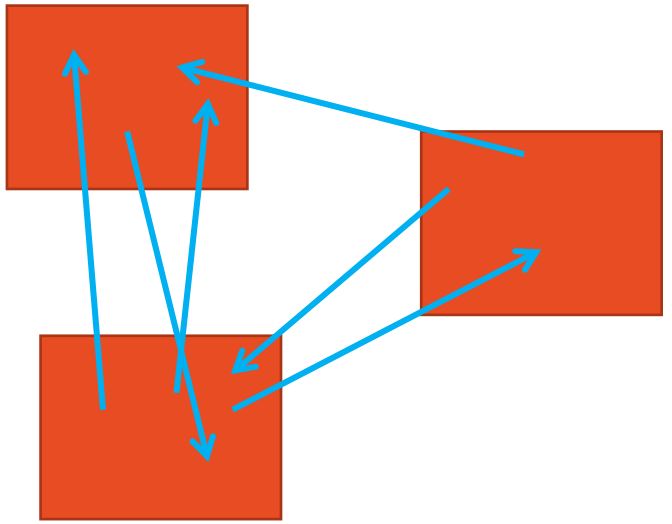
One paradigm to rule them all…

- **No!**
- Programming is an art as varied as the problems it solves
- Different problems map naturally into different paradigms
- Would OOP have become popular without windows-based GUIs?
- Simplicity (when it can be found) is a big winner
  - Use programming styles that encourage this
  - Understand what makes code complex and strategies to simplify it
  - Not all problems have simple solutions
- Use effort proportional to the problem
  - Code that must last 15 years, with maintenance and upgrade
  - Code thrown away after it is marked
  - Project code marked on modularity and testability

How to measure simplicity?

srand;for(0..5){$r[$_]=chr 65+rand 8}sub d{print$/x6;for(0..335){print$_<27&$_>13?'-':$_%14>12?"\n":$_<6?$_[0]?$r[$_]: 'O':$_%14==6?'|':(split//,$b[int$_/14])[$_%14]||$"}print"$/Enter m/[A-Ha-h]{6}/\n"}sub c{return if/[^A-H]/||length()-6;@c=split//, ${$f=\($b[24-++$w]=uc.$")};$w>21&&return 1;for(-6..35){($p[$h]=1)&($q[$h]=1)&($$f.="*")&$n++if$_<0&&$c[$h=$_+6]eq$r[$h];!$p[$b]&&!$q[$d]&&($p[$b]=1)&($q[$d]=1)&($$f.="+")if$c[$d=$_%6]eq$r[$b=$_/6]&&$_>-1}(d$])&die"Done$/"if$n>5;$n=@p=@q=()}while(!c){d|chop($_=uc<>)}d$/;print"$/Looser!$/"

- Lines of code?
- Ease with which good tests can be written?
- References between modules
  - Can each module be **understood** and **tested** independently?
  - "spaghetti-like" references with mutual recursion are bad?
  - Does F# "forward references only" policy encourage good coding?
- Do data structures map naturally and completely to problem domain data?



Use infrastructure

- Don't re-invent the wheel
- Use frameworks etc that are appropriate for the problem
- Use libraries as needed
  - Library ecosystems
    - .NET (F#, C#, C++)
    - JVM (Java, Scala)
  - Learning libraries can take longer than learning core languages
- See project

.NET =>
Hundreds of Thousands of man-years effort

Introduction to Worksheet 4 & project samples

- You are given a significant size program to explore
- 500 lines F#
- Very dense
  - Complete (capable) TINY lazy functional language compiler and run-time system
  - Written in F# but compiler could be written in TINY and compile itself!
- Compilers are one of the program areas where functional languages excel.
- Worksheet 4: cut-down program: explore modularity and active patterns
- project sample: working program: ported to client-side web code & GUI

Introduction to project

- Use FABLE transpiler from F# to Javascript
  - Write code in F#, run it from any browser
  - Use existing Javascript widgets & frameworks
- Javascript
  - Dynamically typed language
  - Functional and OOP
  - After many iterations is now capable but horrible to program
  - Good fit for F# transpilation
- Modern trend in software design – write code in Javascript/HTML to run under e.g. electron framework
  - Atom and Visual Studio Code good examples

Web technology

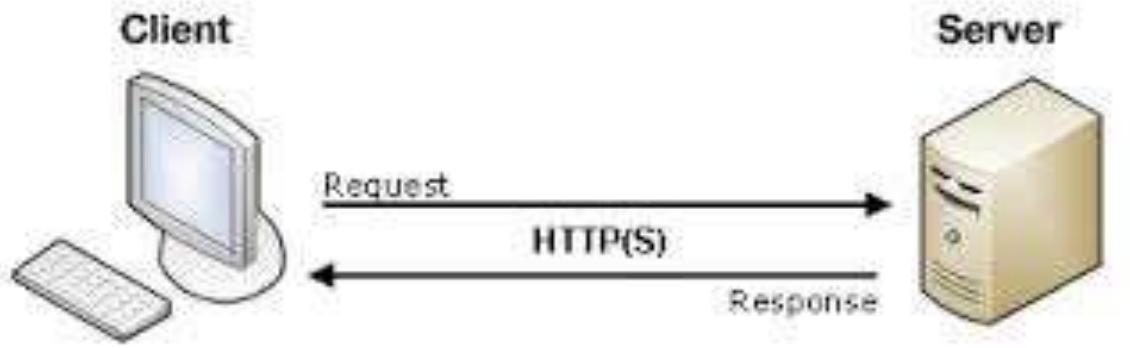
- A long time ago
  - HTML was boring and limited
  - Javascript was a poor scripting language
  - Web programming was writing HTML
  - Browsers were incompatible
- Now
  - HTML is a good standard GUI specification with excellent infrastructure
  - Javascript is highly standardised and capable
  - Javascript libraries are everywhere and create HTML widgets for everything
  - Web applications are written largely with client-side code as full-featured programs
  - Desktop applications can be written with HTML/Javascript using node framework and electron "browser with desktop access".

Why learn web programming?

- The obvious reason…
- Platform independence
  - Windows/Linux/OS-X/Android/I-OS
  - Javascript / HTML / CSS will run on any platform
- Good technology for desktop apps:
  - Node.js – Javascript framework for server-side asynchronous computation
  - Electron or NW.js – desktop frameworks based on node.js
- F#, Haskell, Java can easily be **transpiled** into Javascript

Introduction

- Protocols
  - HTTP - hypertext transfer protocol
    - Allows client browser to communicate with server on foreign computer
  - HTTPS – HTTP encrypted for security
- HTML
  - Hypertext Markup language designed to convey graphical information from server to client
- Basic paradigm
  - Client sends URL to server
  - Server sends web page coded in HTML to client



https://intranet.ee.ic.ac.uk/t.clarke/tom/index.html

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>title</title>
    <link rel="stylesheet" href="style.css">
    <script src="script.js"></script>
  </head>
  <body>
    <!-- page content -->
    <p class="h1"> My Web Page </p>
  </body>
</html>
```

Separate style information from content

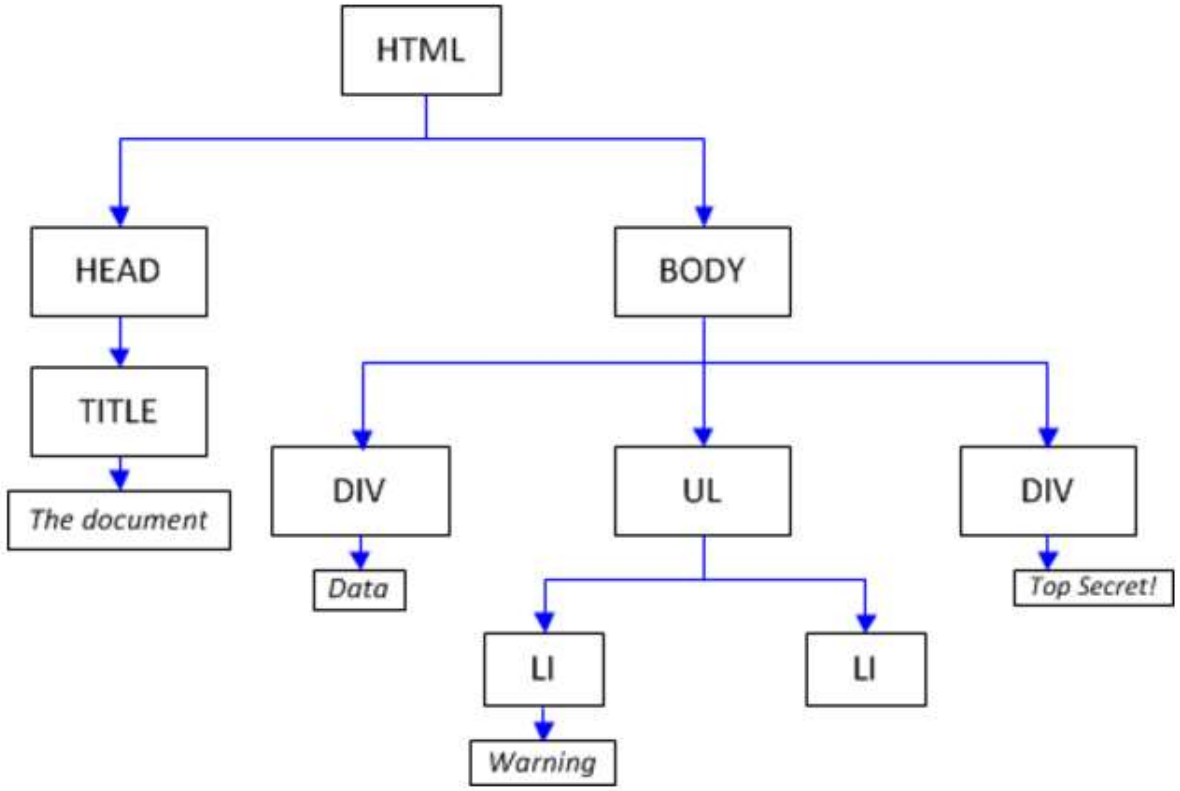
- HTML on server contains content of page
  - Text, buttons, arrays
- CSS file on server contains style information for whole classes of elements on page:
  - Fonts
  - Colors
  - Sizes
  - Margins
  - Borders
- CSS files get very complex
  - Use CSS file pre-processor with input in extended language
  - E.g. SASS or LESS

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>title</title>
    <link rel="stylesheet" href="style.css">
    <script src="script.js"></script>
  </head>
  <body>
    <!-- page content -->
    <p class="myid"> My Web Page </p>
  </body>
</html>
```

```
/*
Minimal CSS
*/
/* Layout */
p.myid { font: 20px "HelveticaNeue", Helvetica, Arial, sans-serif; }
```

DOM in web pages

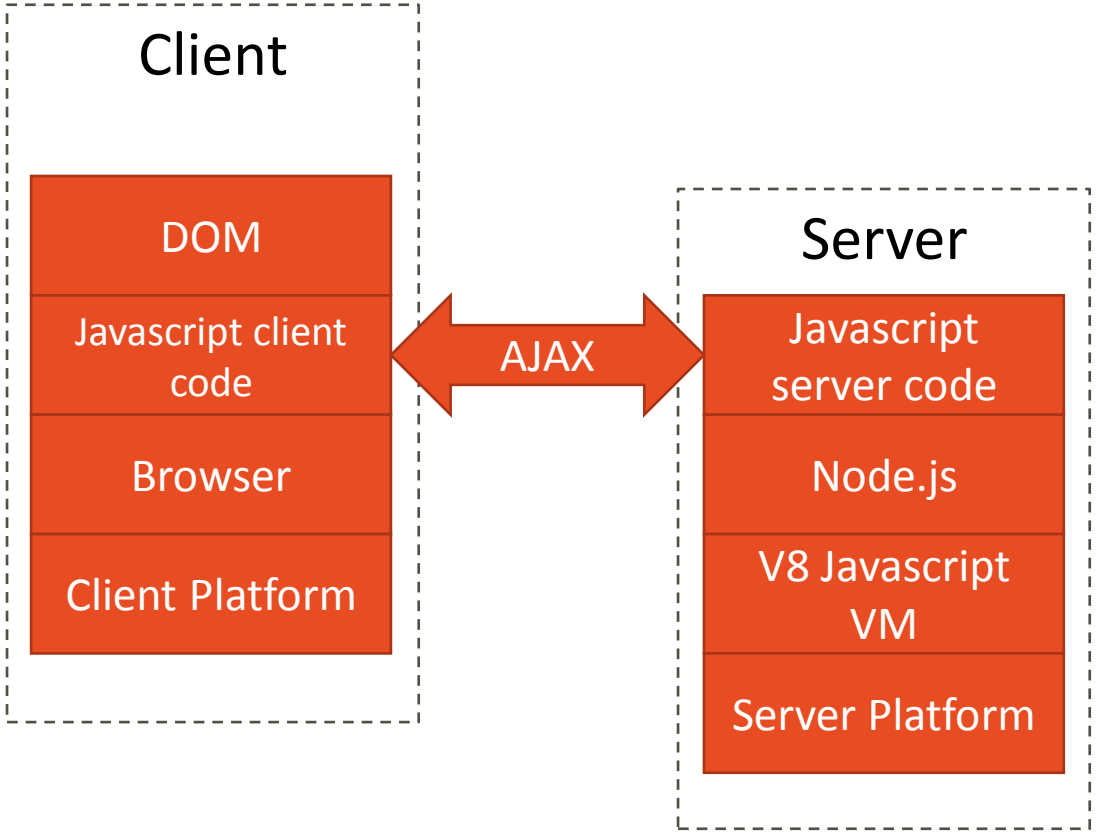
- HTML defines initial Document Object Model (DOM)
- DOM represents displayed page as a tree data structure of elements




```
<!DOCTYPE HTML>
<html>
    <head>
        <title>The document</title>
    </head>
    <body>
        <div>Data</div>
        <ul>
            <li>Warning</li>
            <li></li>
        </ul>
        <div>Top Secret!</div>
    </body>
</html>
```

Javascript as a programming language

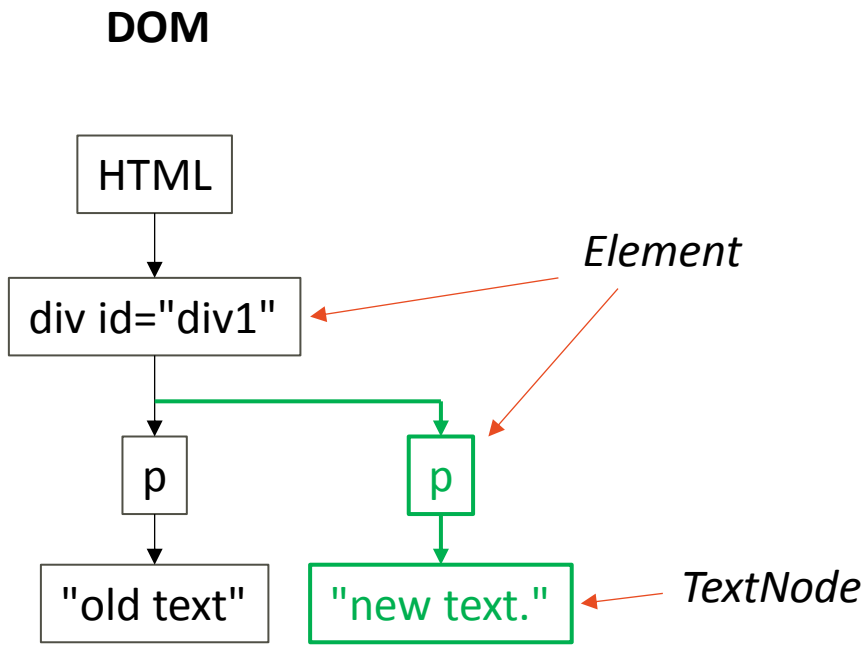
- On Client
  - Browser can run big Javascript programs downloaded as single web page
- On Server
  - Node.js library duplicates asynchronous event model used by browser
  - V8 environment is Virtual Machine (VM) for host platform to run Javascript
- AJAX
  - Client javascript program communicates with server via multiple asynchronous requests
  - XmlHttpRequest API on browser wraps client-server communication

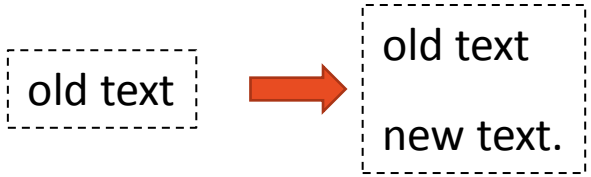


Client-side Javascript and SPA

- Javascript libraries on client can define function of complex DOM elements (widgets)
  - CodeMirror editor
- Javascript on client can mutate initial DOM as required by changes of state
  - AJAX allows bidirectional communication with server
  - No need for new page lookup to present new data
  - Single Page Application (SPA)

Text paragraph added






```
var para = document.createElement("p");
var node = document.createTextNode("new text.");
para.appendChild(node);
var element = document.getElementById("div1");
element.appendChild(para);
```

Javascript statements to mutate DOM