Worksheet 3 - Data and Types

Contents

Introduction

The focus in this worksheet will be using user-defined types to model complex data. Since functional programming sees programs as data + functions it is not surprising that working out types for data has a central role.

Types in functional programming should not, in theory, be any more important than types in any other statically typed language like Java or C#. In practice static types in a functional language are both more valuable and more important.

The reasons you will discover doing this sheet are:

1. It is easier to write and use complex type definitions due to compact syntax and type inference
2. Types are more flexible due to polymorphic typing and Hindley-Milner type inference
3. Data has a primary role in functional programming. In other paradigms code alters the meaning of data in ways that make things more complex.

Finally this worksheet explores how functional code can easily (and automatically, in many cases) be tested.

New F# syntax you need for this worksheet is:

- Records (as C struct, but immutable)
- Discriminated Union (or sum) type
- Option type

Throughout this module the focus is on **statically** typed languages where every identifier has a unique type determined by the compiler. Some languages (notably LISP, Python, Matlab, and Visual Basic) have **dynamic** typing, where types can change with data and are not checked by the compiler. Even though a dynamic type system may be every bit as rich and complex as a static type system this complexity is not so apparent. The programmer can just ignore types, and things work, but without the extra protection and documentation provided by a static type system. Dynamic type systems are (obviously) freer, and allow some types of meta-programming to be done with less complexity. Generally (but not universally) it is preferable to use static typing for large-scale programming and this module follows that. Functional and Object Oriented languages can be either static or dynamic typed. The arguments between advocates of static and dynamic typed languages are interesting. Understanding why they exist requires appreciation of the nuances of pragmatic language use.

Key to coloured boxes in these Worksheets

F# Language reference

Syntax	Name	Description
`type Person = {name: string; age: integer}`	Record type	Like tuple but with named components separated by `;`
`let rec1 = { name = "Blogs"; age = 32}`	record value	Every field must be specified. Values can be any expression
`{rec1 with age = 33}`	record update	New record based on `rec1` with specified fields changed
`rec1.age`	field selector	Value of given field
	`type Component = | Cons1 of type1 | Cons2 of type2` `type Component = Cons1 of type1 | Cons2 of type2` `type Component = Monday or Year of int| Discriminated Union (D.U.) type |` `Value can be any one of the alternates   single value (of Unit) alternates may   be written without type as`Monday`	

Syntax	Name	Description

`Cons2(value)`
`Cons2 value`
SvCons
	D.U. value	Use any constructor with `value` of correct type to construct D.U. type.
		Single values are written as `SvCons` without any type

`Option<'a>` or `'a option`	Option type	Elements are `Some` a, $a \in$ `'a` or `None`. `'a` is a polymorphic type variable.

`Some x`
`Some(x)`
`None`
	Option type constructors	Used to generate or match option values
		Brackets optional.

`function` *matches*	function expression	shorthand for `fun x -> match x with` *matches*
`[<ATTRNAME>]` `let square x = x * x`	Attribute	defines a .NET attribute on function square. Attributes carry metadata[1].

Figure 1. F# elements

F# Syntax: when are brackets needed?

Brackets in F# are used to control parsing but have *no significance* as language elements except in indicating the single `Unit` value `()`. Thus they are optional for function parameters and tuples:

```
1:  f (x)
2:  f x
3:  (a,b)
4:  a,b
```

They are needed where parsing would go wrong without them:

```
1:  f (a,b) // function with tuple parameter
2:  fun (a,b) -> // anonymous function with tuple parameter
3:  List.map (fun a -> a+a) // anonymous function should almost always be bracketed
```

F# Type Syntax: `int list` versus `list<int>`

Remember that these two versions of syntax are the same.

Other F# Language Resources

I don't expect the level of these worksheets to be optimal for everyone, although I try to make them self-contained and welcome feedback for improvement. There is obviously variation in previous experience programming and speed with which you master functional language concepts. Also, the worksheets are not a complete language reference. You are expected to do your own research when necessary as you would learning any language.

You should use additional material if either of two opposite reasons apply to you:

- You want a precise language reference more complete or more concise than here. I recommend reference pages mostly from the newly updated official MSDN F# language documentation. This is great material. For almost anything google search will get you to the MSDN appropriate page - so the example links below are not really needed.
- You don't fully understand what is going on doing these worksheets - in which case you want a more accessible *introduction*. Most of my recommendations here come from the fsharpforfunandprofit site which is designed for industry programmers wanting to learn F# but has very accessible material. Before looking at this remember the support chat channel - it is likely that I or a GTA can resolve problems. Also note the course recommended textbooks

Type	References
Easy Introduction	Introduction to F# pattern matching and unions

Type	References
	Introduction to F# discriminated unions
Definitive F# reference	Pattern matching
	Custom F# operators
	Disciminated unions
Quick F# reference	F# cheat PDF
	F# cheat HTML

Types and Set Theory

A natural way to think of a type is as the set of its possible values. **`Unit`** has a single element **`()`**, **`int`** has 2^{32} integer elements, etc.

There are naturally two ways to construct new sets from existing ones $T1$ and $T2$:

- *Sum*: the new set is the union of the elements of $T1$ and $T2$. We tag each value in $T1$ and $T2$ so that even if these types share elements (for example if they are the same) we get different values in the union for each of $T1$ and $T2$. Technically this is a *discriminated or disjoint union* and not the same as *set union* when the two sets have non-empty intersection. For example the sum of **`int`** and **`int`** would contain two distinct versions of the integer 11 labelled by the tags identifying $T1$ or $T2$.
- *Product*: as a Cartesian product, a pair $\{(t1, t2) : t1 \in T1, t2 \in T2\}$

The F# type equivalents are *discriminated union types* (for sum) and *tuple types* (or equivalently *record types*) for product. In F#:

```
1:  // example of F# product type
2:  type MyTuple = int * string // give name to a tuple type defined using the * "Tuple" type constructor
```

You have already used tuple types, and know that the *tuple* is a *type constructor* that creates a new (tuple) type from its constituent types.

Sometimes it is more helpful to have a type similar to a tuple but with named components. This is called a **record** and will be familiar to you from programming C++ as a **struct**. There is no semantic difference between a tuple and a record except that the record is accessed via **field names** and the fields of the tuple do not have names. This leads to different syntax:

```
1:  type Coord = {XCoord: int; YCoord: int} //record type definition
2:  let xRecord: Coord = {XCoord=10; YCoord=20} // record value
3:  let xTuple: int*int = (10,20) // 2-Tuple value
```

You can see from the above example there are pros and cons. Tuples are much simpler and easier to write, but they document data less well. In this case the definition of **`Coord`** tells you what the two coordinates are. In F# you choose to use tuples or records based on convenience. Tuples are lightweight: no type definition needed and easy to write, but less explicit.

Nevertheless anything that you can do with a tuple, you can equally do with a record type, and vice versa.

The F# syntax for a sum (called *discriminated union*) type is:

```
1:  type TypeName = | Tag1 of Type1 | Tag2 of Type2 | Tag3 of Type3 | Tag4
```

- The type definition defines a number of named alternates (or cases) each of which can have a different type. Note that the alternate types do not have to be different. The alternate names (**`Tag1`**..**`Tag4`** here) must start with a capital letter.
- The initial **|** is optional if there is more than one alternate
- Any number of alternates, including just one, are possible
- Note that the **`Tag4`** alternate has only one value. The type is omitted in this case.
- Single value alternate names like **`Tag4`** are used as constants in expressions, e.g **`Tag4`**.
- Multiple value alternate names are used as constructor functions that generate values of the D.U. type, e.g. **`Tag1: Type1 -> TypeName`** so that if **`t1: Type1`** then **`(Tag1 t1):TypeName`** .

  Q1. What is **`type A1 = | Monday1 of Unit | Tuesday1 of Unit`**? In FSI compare it with **`type A2 = Monday2 | Tuesday2`**.

  Answer

Examples of F# discriminated union (D.U.) types:

```
1:  type MyOrchardDU = | NumAppleTrees of int |  NumPairTrees of int // can have any number of apple or pair trees but not both
2:  type A = ...
3:  type B = ...
4:  type C = ...
5:  type MyDU = | Thing of A | Person of B * C | Weird
```

Q2. What is the set equivalent of the above D.U. types?

Answer

Q3. **`MyOrchardDU`** is not a perfect representation of orchards that can contain either apple or pair trees but not both. The problem is in how an orchard with *no* trees is represented. Why is there a problem? Think about this before you look at the answer.

Answer

Q4. Suppose **`A,B,C`** types have numbers of values a, b, c respectively. How many distinct values does **`MyDU`** have?

Answer

Q5. Suppose $A = \{a_1, a_2\}, B = \{b_1, b_2, b_3\}, C = \{c1, c2\}$. What are all the values of type **`MyDU`**?

Answer

Tuples, records and arrays in F#

Records and arrays will be familiar to you, so that you think of arrays as like records but with an integer *value* to access individual items instead of a hard-coded *field name*. Tuples, you have learnt, are like records without field names; instead an item is specified by its position (usually in a pattern match).

In addition, in F#, arrays have *mutable* elements whereas tuples and records do not. Arrays should be used where elements have some defined numeric order and records where elements are unique (and maybe of different type). Although F# arrays are mutable you can choose to use them (and I usually do) as immutable array types.

Tuples are a light-weight equivalent of records. The key difference is that you do not need to define or name tuple types, a tuple type is defined implicitly by the number and types of the items that are used to construct the tuple value. In F# there is no way to access the elements of a tuple by index, as in an array. Tuples seem weak and the reason they are used so much in functional programming is because grouping data together to return multiple results from functions is very common, and defining new records for each instance not worth the complexity and *syntactic noise*. Tuples are also very useful in conjunction with pattern matching (which tends not to exist in non-functional languages).

It is possible to define functions to access tuple elements. There are two such which are in the core language: **`fst`** and **`snd`**. These apply to 2-tuples only and select the first or second element. All others, if you want them, must be user defined.

Q6. what is the (polymorphic) type of **`snd`**?

Answer

Tuple selection can always be implemented through pattern matching:

```
1:   let x = 1,2,3 // 3-tuple
2:   let choose = match x with |_,_,a -> a // select third element of tuple
```

Q7. define **`snd3`**, a version of **`snd`** that will work on 3-tuples.

Answer

F# provides a convenient syntax for simple functions like this where a function body consisting of a **`match`** on the parameter can be written as a value omitting the parameter entirely. In this syntax **`= function`** substitutes **`x = match x with`**:

```
1:  let sel3 =  function | _,a,_ -> a
```

In practice, because tuple values are usually extracted in pattern matches, selector functions like snd3 are rarely used.

Q8. What are the differences between using a two element list **`[ a ; b ]`** and a tuple **`a,b`** to return two values **`a`**, **`b`** from a function, using pattern matching to unpack the result? What are the advantages of the tuple?

Answer

Discriminated Unions

Sum (D.U.) types are used in constructing linked lists and other recursive data structures. The built-in **`list`** datatype is in fact a D.U. although this fact is hidden by the syntax you use to create and match lists. A value of type **`string list`** has two options: **`Empty`** & **`ListNode of string * string list`**. We could define this as:

```
1:      type DUList = | StringListNode of string * DUList | Empty
2:
```

The immediate problem is the necessary recursion. You know **`DUList`** must be this type because the tail part of a list is another list of exactly the same type. But the type definition is now recursive. This is allowed. It does not create problems because in real lists the recursive alternate is not always taken. At some point we will meet **`Empty`** and the list stops.

Using this D.U. we can define our own equivalent of the built-in list **`[1;2]`**

```
1:      type DUList = | StringListNode of (string * DUList) | Empty
2:      let lis = StringListNode("1", StringListNode("2",Empty))
3:
```

The real list type is not restricted to **`string`**. We can make **`DUList`** similarly flexible by replacing **`string`** by a *wild card* (polymorphic) type variable in the type definition.

```
1:  type 'a DUPolyList = ListNode of 'a * 'a DUPolyList | Empty
```

This is actually quite significant. **`DUPolyList`** is not a type. It is a *type constructor* which when given a parameter (in place of **`'a`**) will return a type. You could think of it as a *type function*. In the examples below it creates different list types. In the last example the wildcard **`'a`** is replaced by **`int DUPolyList`** to make a list of lists! Note that **`ListNode`** is truly polymorphic, it will create a node of whatever type is required from the context:

```
1:  type 'a DUPolyList = ListNode of 'a * 'a DUPolyList | Empty
2:  let x: int DUPolyList = ListNode(1,Empty) // list of int
3:  let y: string DUPolyList = ListNode("Hello",Empty) // list of string
4:  let z: bool DUPolyList DUPolyList = ListNode(ListNode(true,Empty),Empty) // list of bool DUPolyList
```

Q9. Write a D.U. type for a binary tree of integers. Can you generalise this to a polymorphic binary tree **`'a BTree`**?

Answer

Q10. Why do cases such as **`Empty`** on D.U. types not create errors in another *billion-dollar mistake*?

Answer

Complex data can also be represented using *records* for nodes instead of *tuples*. There is a complication in a recursive type definition like a list or tree. The record and the D.U. types are *mutually recursive*. Neither can be written first because each depends on the other.

in fact mutually recursive types like this in F# are known to be a "code smell" that is something you are allowed to do but should try to (and nearly always can) avoid. the green box below, which can be skipped, shows how you use mutually recursive types in this case, and why they are a "code smell".

One solution is to use the `and` keyword to introduce the second type.

```
1:  type RListNodeBad = {Head: int ; Tail: RList} // warning mutually recursive types in F# are not good practice
2:  and RListBad = | RNode of RListNodeBad | RNil
```

Q11. What is the **`RListBad`** expression equivalent of **`[1;2;3]`**?

Answer

However using two D.U.s like this is very annoying: unpacking or creating a single node requires two alternates.

In this case the best way to get something like *records* inside a recursive D.U. is with a single D.U. type of tuples making use of *field names* on a tuple and avoiding the complexity of mutual recursive types:

```
1:  type RList = | RNode of Head: int * Tail: RList | RNil
```

`RNode(1 , RNil)` can then also be written as **`RNode(Head = 1 , Tail = RNil)`** in a way similar to a record. However there is no record so **`Head`** and **`Tail`** field selectors would need to be coded as functions **`let Head RNode(x,_) = x`** etc. Note the allowed single case pattern matching used in this function definition.

Notice that lists defined using D.U.s do not have the convenience of built-in lists. The green box question below shows how, if needed, you can get back most (but not all) of that convenience in custom types. You can skip it.

Q12. define an operator **`^*^`** which is the **`RList`** equivalent of **`cons`** by using **`( ^*^ )`** according to the method explained here and rewrite this list.

Answer

Another common use of D.U.s is as an *enumeration*, for data that can take only a fixed set of values:

```
type DayOfWeek = Monday | Tuesday | Wednesday | Thursday | Friday | Saturday | Sunday
```

D.U.s can be used to make *protected versions* of normal types. For example an integer field may be used as an identifier for a project record. These ids can be *wrapped* in a D.U. so that they are not mixed up with a normal integer by mistake:

```
type TPID = | projectID of int //single case D.U.
```

One of the most common mistakes in programming is calling a function with parameters swapped. In **`copyProject`** below one of those parameters is wrapped by **`Pid`** which makes this error impossible because it would correspond to a type error. This method is useful whenever the parameters represent different things, but are coded using the same F# type.

```
    type TPID = Pid of int // PIDs are integers wrapped by Pid
                           // representing project IDs
    let copyProject (pid: TPID) (count: int) =
        ...

```

Modelling data with types

Before tackling the next question it is helpful to introduce a D.U. type so useful that it is implemented in the core language. It is F#'s solution to Tony Hoare's self-professed *billion dollar mistake*.

```
type 'a option = | Some of 'a | None // defined in F# core language
```

(NB **`option`** is an alias for **`Option`** in F#, you will see both used for this type)

You can see that this type encapsulates a value (of polymorphic type **`'a`**) that can be either present or absent, and serves the purpose of Null in many other languages. The difference is that if a value could be absent, this is now encoded into the type system by wrapping the value type (**`'a`** in the above definition) in an option.

```
    let x: int = ... // value always an integer
    let x': int Option = ... // value could be either Some int or None

```

A classic example:

```
let betterSqrt x = if x < 0.0 then None else Some (sqrt x)
```

Q13. What is the type of **`betterSqrt`**?

Answer

Q14. The EEE departmental database has a table of *people* that lists, for each person, the data in Figure 2.

Field	type	Description
eeid	int	unique id
name	string	name & initials
role	string	"EEE" or "EIE" or "staff"
year	int	Year of UG course (1-4) undefined if staff
cid	int	College ID

Figure 2. Database records

Write a record type definition, with any associated discriminated union types you deem desirable, that represents this data.

Answer

Q15. Consider the function below

```
type TUGYear = One | Two | Three | Four
let incrementUGYear = function
    | One -> Some Two
    | Two -> Some Three
    | Three -> Some Four
    | Four -> None
```

What type does this function have? Why is it not possible to write a safe function for this operation of type **`int -> int`**?

Answer

Since adding one to a **`TUGYear`** may not be possible this is the only sensible type for this function. Otherwise you would need to use **`failwithf`** to throw an exception when used on the a 4th year student. This is not a good outcome and better to require the programmer to do something with the possibly **`None`** result.

This example shows how in F# (and typically in functional programming languages) the language constructs make it low cost to do what is good practice:

- Use data which precisely maps the problem domain, with all values representing valid inputs
- Write mathematically *total functions* that have valid results for all possible inputs

The programming equivalent of a mathematical partial function is a *possible run-time error*! Think of the type system as something that keeps things working by ensuring that every function can only be given its expected input. This only works if functions are written to correctly handle all values within their input type and therefore are total.

It should now be clear why making **`TUGYear`** an **`int`** - the lazy solution here - is not good practice.

The Tennis Game Problem

One nice example of how real-world data can be mapped into types in different ways occurs when modelling the scores of a *tennis game*. In a tennis game two players play a sequence of points until such time as the game is won by one or other player. Each point is always won by one player, and whenever a point is won the *score* changes to reflect this. However, the score of a tennis game changes in an unusually complex way, described below, which makes this a good example.

We model the progress in a Tennis game as a a value of type **`TennisGameScore`** which changes whenever a point is won and includes both the score of each player and any other information needed to determine who wins.

The state (represented as s **`TennisGameScore`** value) of a Tennis game at any one time is represented by the *scores* of the two players, and additionally (in some cases as below) by which player has *advantage*.

The tennis game consists of *points* which are played in sequence. As the game is played each point is won by one of other of the two players, changing the value of **`TennisGameScore`**, until the game is *won*.

1. Each player can have one of these numeric scores in one tennis game: Love (0), 15, 30, 40. For scores less than 40, winning a point increases the score of the winning player to the next in the order: Love, 15, 30, 40.
2. If player A has score of 40 and the other player score less than 40, and A wins a point, then A wins the game.
3.

There are special rules in the case that both players have score of 40.

-

  If both have score of 40, the players are initially in *deuce*, and neither player has *advantage*.
-

  If the game is in deuce, the next winner of a point will have *advantage*.
-

  If the player with *advantage* wins the point, (s)he wins the game.
-

  If the player without *advantage* wins, they are back in deuce and neither player has *advantage*.
- These rules codify the fact that you need to win two points in a row to win the game from a previous position of level scores.

These rules determine how the game state (as a value of type **`TennisGameScore`**) changes when a point is played. However the rules state that in some cases the game is won by one of other player and finishes. This is an additional terminal state outside **`TennisScoreValue`**.

Let us capture these rules as a function **`recordPoint`** with input the score before a point is played, and the player who wins the point, and output the result after the point. The **`recordPoint`** inputs clearly have types **`TennisGameScore`** and **`Player`**. The output however is either a **`TennisGameScore`** value or a win by one player or the other. By repeatedly transforming the game state using **`recordPoint`** we can model the way that the score changes as points are won, and when the game finished.

```
recordPoint: Player -> TennisGameScore -> MaybeWonTennisGameScore
```

Here we have invented a new type **`MaybeWonTennisGameScore`** has as elements possible game scores, and also the two possible states were a player has won the game as the result of winning the point.

The **`Player`** type can simply be a D.U.:

```
1:  type Player = PlayerOne | PlayerTwo
```

The problem is to work out a suitable type for **`TennisGameScore`**. Given this, it is relatively easy to work out a type for **`MaybeWonTennisGameScore`**.

To demonstrate the power of F# polymorphic types let us try to define **`MabeWonTennisGameScore`** before we have worked out the (more difficult) precise type **`TennisGameScore`**. This seems impossible! But we can note that **`MaybeWonTennisGameScore`** has two possibilities: a win by a player, or a value of **`TennisGameScore`**. That looks like a D.U. and we can parametrise a D.U. with an unknown type - in this case the type of **`TennisGameScore`**.

`'a OrWin` adds the *game won* alternate to type **`'a`** so that we can define:

```
1:  type MaybeWonTennisGameScore = TennisGameScore OrWin
```

This statement correctly defines how you can construct **`MabeWonTennisGameScore`** from **`TennisGameScore`** even though we do not yet know what **`TennisGameScore`** is.

Q16. Define a D.U. for **`'a OrWin`**.

Answer

Now we need to define **`TennisGameScore`**. The most obvious solution, a tuple or record of two **`int`**, does not work well because:

1. It includes many meaningless scores
2. It does not differentiate the extra game states where both players have 40 but neither player has yet won.

The type below is a possible solution, because **`bool`** fields have been added to deal with *advantage* and *deuce* states, but it is still a bad solution because many possible data values, for example where both players have advantage **`true`**, are meaningless. This deals with 2. above but not 1.

```
1:  type PlayerScore = {pts: int ; advantage: bool; deuce: bool}
2:  type TennisGameScoreBad = {playerOne: PlayerScore; playerTwo: PlayerScore}
```

Mathematically we would like the values in the **`TennisGameScore`** type, and the possible distinct states of the real tennis game score, to be in 1-1 correspondence. Figure 3 illustrates the function that maps *real game state* to **`TennisGameScore`** value. Cases 1. and 2. above correspond to this function being an *injection* (one-to-one into) or *surjection* (many-to-one onto) respectively. For complete and unique data mapping, the best case, we require a *bijection*.

Q17. What, mathematically, would the case where some real world states have no associated data value correspond to?

Answer

Q18. Suppose we allow two different data values to represent the same real world state. Is that possible? How is it represented mathematically?

Answer

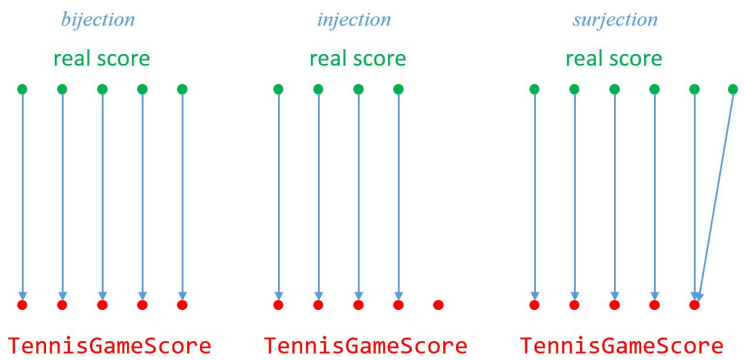

Figure 3

Q19. Work out a solution for **`TennisGameScore`** which makes it easy to write the **`recordPoint`** function and bring it to the next lecture together with your **`recordPoint`** definition. Note the function **`playGame`** defined below after the green box which makes it easy to test **`recordPoint`** with specific data.

Answer

Debugging Functions with Complex Types

This box shows you how to *debug type inference when it all gets too complex*! The principle is to rewrite your code to make it simpler, which is always possible.

The problem with type inference is that when there is a type error in a complex function the types inferred everywhere can all be wrong, making localisation of the error more difficult. There are however three simple solutions:

1. Add explicit types to named functions
2. Turn anonymous functions into named functions.
3. Make one-off local high order converter functions into normal functions (possible when there is only one parameter function to be converted). In fact a general rule is *never* define a local high order function which is only used once. A value will always be simpler. (If you don't understand what these words mean the example below will make it clearer).

Note that you *can* add explicit types to anonymous functions but the syntactic complexity makes this unhelpful unless the function is very simple. Also making an anonymous function named generally simplified complex code.

Here is an example. When writing the code for a function **`playGame`** it initially did not type check. Examining the inferred types here does not much help work out what is wrong.

```
1:  type TennisGameScore = | Score of int * int // dummy type definition, not yet correct
2:  type Player = PlayerOne | PlayerTwo
3:  type 'a OrWin = Win of Player | Game of 'a
4:  type MaybeWonTennisGameScore = TennisGameScore OrWin
5:
6:  /// example of badly written functional code which does not type check
7:  /// give score after sequence of points with results in string points is played
8:  /// initScore is the initial score (TennisGameScore)
9:  /// the result is the final game state (MaybewonTennisGameScore)
10: /// points is a sequence of '1' or '2' chars indicating point winner.
11: let playGame rcdPoint points (initScore: TennisGameScore) =
12:     /// converter function to make rcdPoint work here
13:     let rcdConv rcdP p mWTGS =
14:         match mWTGS with
15:         | Win p -> Win p // if game is won do nothing
16:         | Game tgs -> rcdP p tgs
17:     let getPlayer = function
18:         | '1' -> PlayerOne
19:         | '2' -> PlayerTwo
20:         | ch -> failwithf "Bad character %c in string, chs must be '1' or '2'" ch
21:     // Seq.fold works like List.fold but on the characters of a string
22:     Seq.fold (fun mtgs ch -> (rcdConv rcdPoint) (getPlayer ch)) (Game initScore) points
```

Applying these *simplification rules*, as in Figure 4 below, requires making the anonymous function in line 18 a named function **`playFolder`** and turning high order converter **`rcdConv`** into function **`rcdPoint`**'. The code rewritten like this type checked straight away because in rewriting the functions more explicitly the bug was corrected! But if it had persisted it would have been localised to one small subfunction. Note how the extra documentation on the subfunctions, and a parameter name change, also makes the intent of the programmer clearer. This would probably not be needed except in a tutorial.

Note also in the code below that the explicit types are not universal, types have been added to function parameters selectively.

In order to test **`recordPoint`** conveniently we need to be able to apply it repeatedly to an initial score with a sequence of points whose state (which player wins) is defined by, say, the characters in a string. Thus **`"121"`** indicates three games won in order by **`PlayerOne`**, **`PlayerTwo`**, **`PlayerOne`**.

```
1:      type TennisGameScore = | Score of int * int // dummy type definition, not yet correct
2:      type Player = PlayerOne | PlayerTwo
3:      type 'a OrWin = Win of Player | Game of 'a
4:      type MaybeWonTennisGameScore = TennisGameScore OrWin
5:
6:      /// Apply a sequence of played point results, represented as a string, to initScore
7:      /// Return the result which may be another score, or that a player has won
8:      let playGame recPoint (pointResults:string) (initScore: TennisGameScore) =
9:          /// Like recordPoint but with MaybeWonTennisGameScore input
10:         let recPoint' (p:Player) (mWTGS: MaybeWonTennisGameScore) =
11:             match mWTGS with
12:             | Win p -> Win p // if game is won do nothing
13:             | Game tgs -> recPoint p tgs
14:         /// in the string '1' and '2' represent ones from player one and two respectively
15:         let getPlayer = function
16:             | '1' -> PlayerOne
17:             | '2' -> PlayerTwo
18:             | ch -> failwithf "Bad character %c in string, chs must be '1' or '2'" ch
19:         /// folder function for fold that has MaybeTennisGameScore state varaiable
20:         let playFolder (mgs:MaybeWonTennisGameScore) (pt:char) = recPoint' (getPlayer pt) mgs
21:         // Seq.fold works like List.fold but on the characters of a string
22:         Seq.fold playFolder (Game initScore) pointResults
23:
```

Figure 4

Automated Testing

Programs need testing, not just to ensure basic functionality, but to make sure that future changes and enhancements to the code don't break previously tested features. Therefore automated test suites are essential. Running such a test suite for complex software can take days, with hundreds of thousands of individual tests.

EEE students might be interested to note that the test suite Salman Arif wrote for VisUAL took roughly 24 hours to run.

Testing (pretty obviously) is one of the things that must be taken more seriously as programs get larger, so it is sometimes omitted in introductory programming courses. When writing real programs the tests are as important as the program code.

In an object oriented language testing is inherently difficult. The program is represented by a set of objects each of which has internal state. Its operation comes from the interaction between all these objects, with changing state. It is difficult to isolate and test specific behaviour of one part.

In a functional language each (pure, with no side effects) function can be tested precisely by providing a set of inputs and checking the output for each input. There is no internal state to complicate things. A further advantage is that the required output need not be specified separately for each input. A tester must write, for given automated test software:

- A complete set of assertion checks
- A data generator that creates suitable input data (this process may be partially or completely automated)

FsCheck: Randomised Property-based testing

Reference: FsCheck documentation

In the final part of this worksheet you will (you've guessed it) write a set of *assertions* or *properties* to test your **`recordPoint`** function. You will then use FsCheck to do the testing.

The great advantage of automated tools like FsCheck is that testing a function is no more work than writing down the properties you know the function must satisfy.

For example, **`List.rev`** has a self-inverse property: **`List.rev (List.rev x) = x`**. Let us use FsCheck to see whether this holds (of course it should).

See the box below for methods to download and use libraries such as FsCheck.

Installing Third Party F# Libraries and Dependencies

F# libraries, for example **`FsCheck`**, are nearly all open source and published on github. There are two standard ways to access them. The "shrink-wrapped" Visual Studio method (1) is much simpler but only works when you have Visual Studio installed and is more opaque, so if there is some VS glitch it is difficult to debug.

1. Use Visual Studio. Create a project for your own code. Use NuGet to add FsCheck to your project.
2. Download the Fscheck source from github. Unpack the zip into a directory **`C:\github\FsCheck`**. Note that will need to rename the **`FsCheck-master`** root directory as **`FsCheck`** when unpacking for the paths given below to work. Use the path to the relevant dll in your F# script files to reference the **`FsCheck`** assembly. See examples, all of which assume that **`FsCheck`** has been downloaded to **`C:\github\FsCheck`**.
3. Variant of 2. Create a clone of the github project. This will also give you a directory with the same files. The only difference is that it will contain extra files managing your directory as a git repository and therefore track any changes you make, or any changes made to the master you have cloned. In the normal case that you are using and never changing library files this has little benefit over 2.

Compiling Third Party Projects

In cases 2. and 3. the source files you download will usually need to be compiled on your system to generate a (binary) *dynamic link library* assembly that you can reference (suffix **`.dll`**). The standard way to do this is to run from the package root directory - in this case **`C:\github\FsCheck`** - **`build.cmd`**. Note that on linux this will be **`build.sh`**. This command downloads the latest version of the **`PAKET`** F# dependency manager tool, which then ensures all necessary dependencies on your system for the **`FsCheck`** project are correctly installed, typically under directory **`.\packages`**. Finally **`.\build.fsx`** is run, which rebuilds (and maybe also tests) the **`FsCheck`** binary. At the end of this process the latest published binaries are available, either under the top level **`.\bin`** or as with **`FsCheck`** under **`.\src\bin\Release\`**. **`FsCheck`** has a number of variants each in a separate directory under **`.\src`**. The ones used here are **`.\src\FsCheck`** and **`.\src\FsCheck.NUnit`**. This shows how the (rather long) paths given below are constructed.

Checking dependencies

As always in F# the language intellisense will show errors immediately if modules referenced by **`open`** are not provided by correctly referenced assemblies.

Install FsCheck as above, and run this script:

```
1:      #r @"C:\github\FsCheck\src\FsCheck\bin\Release\FsCheck.dll"
2:
3:      open FsCheck
4:
5:      let revOfRevIsOrig (x: int list) = List.rev (List.rev x) = x
6:
7:      Check.Quick revOfRevIsOrig
8:
```

You get a message saying: OK - 100 tests.

Replace **`(x: int list)`** by **`(x: float list)`** and retest.

The `List.rev` function fails, surprisingly, and you get output something like (the checks are random so your example will be different):

```
Falsifiable, after 25 tests (13 shrinks) (StdGen (532566436,296164725)):
Original:
[-20.8; -18.25; -5.1875; -0.04545454545; nan; -19.57142857; -9.0; -41.0;
 -1.797693135e+308; 3.05; 4.940656458e-324; 1.797693135e+308; -1.797693135e+308;
 -24.4]
Shrunk:
[NaN]
```

FsCheck provides the failing example it found after 25 tests:

```
[-20.8; -18.25; -5.1875; -0.04545454545; nan; -19.57142857; -9.0; -41.0;
 -1.797693135e+308; 3.05; 4.940656458e-324; 1.797693135e+308; -1.797693135e+308;
 -24.4]
```

FsCheck tries to simplify (shrink) the size of the failing test by removing items from the list. It does a good job in this case and finds the simplest problem with a 1 element list: `[nan] <> [nan]`. Equality behaves as expected for nearly all float but in IEEE floating point `nan` is "not a number" and therefore does not even equal itself according to the IEEE standard! This is the type of unusual gap in specification that you would not find yourself but FsCheck finds automatically.

Replace `Check.Quick` by `Check.Verbose` to see details of *all* the checks conducted using FsCheck.

Some points:

- `revOfRevIsOrig` is a boolean assertion. `FsCheck` will check that it is true for a selection of random input values. Any number of input parameters to this are allowed, they are given random values by `FsCheck` according to a heuristic that biasses values towards lower numbers or shorter lists.
- A polymorphic input type `'a list` will not do, you must specify precise input types.

Try an assertion with three inputs:

```
open FsCheck
let plusIsAssociative a b c =
   (a + b) + c = a + (b + c)

Check.Quick plusIsAssociative

```

Use `Check.Verbose` to print out the input values actually used.

Try testing a Boolean equation. There are two differences illustrated in this example:

- For convenience the name of the test function is made highly descriptive with spaces, this is allowed because it is enclosed in double back-ticks.
- The tuple of the input parameters `a`, `b` to test is not required since this can safely be inferred.

```
let ``de Morgan's Theorem Fails`` a b =
    a && b = not ((not a) || (not b))

Check.Quick ``de Morgan's Theorem Fails``

```

This test fails immediately!

> **Q20**. What is the problem?
>
> Answer

Correct the equation for de Morgan's Theorem and check it again. It will pass.

FsCheck can even generate random functions for testing higher order functions!

```
let associativity (x:int) (f:int->float,g:float->char,h:char->int) =
    ((f >> g) >> h) x = (f >> (g >> h)) x

Check.Quick associativity

```

Here `f`, `g`, `h`, will be randomly generated and tested with a random parameter x.

> **Q22**. Look up the definition of `>>` if you don't remember it and determine what property this checks. Is it true for all functions `f`, `g`, `h`?

If your input data has n possible distinct combinations (for de Morgan's theorem $n = 4$) and you conduct r random tests what are the chances that you cover all combinations?

Answer

FsCheck is designed for cases where random inputs are useful and so can test pretty well anything. For complex functions FsCheck allows control how inputs are generated and combined to form custom distribution random tests that cover the most important cases quickly.

Using FsCheck with different numbers of random tests:

The chance that any one combination is not covered is: $(1 - \frac{1}{n})^r$

This is true, independently, for all n combinations, so the chance of complete coverage is: $(1 - (1 - \frac{1}{n})^r)^n$

Using the approximation $(1 - \epsilon)^n \approx e^{-n\epsilon}$ twice, this simplifies to: $e^{-n(e^{-\frac{r}{n}})} \approx 1 - ne^{-\frac{r}{n}}$

So for $r = Kn + \log_e n$ this becomes very close to $1 - e^{-K}$. The penalty in required extra tests using random testing instead of exhaustive testing is thus not very large and $10\times$ the number of tests ($K = 10$) will suffice to make random testing almost certainly as good as exhaustive testing. Normally, for complex problems, exhaustive testing is impossible so random testing is as good as it gets.

FsCheck will automatically determine random input data for all F# types including D.U. and therefore its default data generators often suffice.

Q21. What value of K will lead to 6 σ (6 standard deviation) confidence that all combinations have been checked?

(Q21 Answer not supplied)

```
1:      // like Check.Quick testF but with adjustable number of tests
2:      // see other fields of type Config for more complex configuration options
3:      Check.One({ Config.Quick with MaxTest = 1000}, TestF)
4:
```

Using FsCheck with multiple properties:

```
1:  type ListProperties = // define set of properties as members of a user-defined type
2:    static member ``reverse of reverse is original`` (xs: int list) =
3:           List.rev(List.rev xs) = xs
4:    static member ``reverse is original`` (xs: int list) =
5:           List.rev xs = xs
6:
7:  Check.QuickAll<ListProperties>()
8:      // use Check.All( config, propType) to alter number of tests for each property
```

NUnit - Unit Testing

In the world of imperative languages mutable state means that properties as assertions about inputs and outputs cannot generally be written. Testing is a lot more difficult, and the tools available are less sophisticated. Testing is commonly called *unit testing* where a unit is some part of a program to be tested which could be as small as one class or as large as a whole GUI.

Unit testing typically runs by specifying manually a set of data inputs and the corresponding output. The test, once coded, will be run automatically. Unit tests are hard work to write but they have a place even in functional testing where specific ad hoc examples can be added to a suite of random tests based on properties.

A standard .Net testing framework is **`NUnit`** which is integrated into FsCheck, and provides convenient syntax for both property and unit-based tests. In the samples below **`[<Property>]`** indicates an FsCheck property-based test. **`[<Test>]`** indicates an NUnit set of unit tests. The **`Assert.*`** lines indicate a condition that must be satisfied for the test to pass using classic NUnit syntax.

```
1:      #r @"C:\github\FsCheck\src\FsCheck.NUnit\bin\Release\nunit.framework.dll"
2:      #r @"C:\github\FsCheck\src\FsCheck.NUnit\bin\Release\FsCheck.NUnit.dll"
3:
4:
5:      open NUnit.Framework
6:      open FsCheck
7:      open FsCheck.NUnit
8:
9:      [<Test>]
10:     let VariousStringTests() = // groups together unit test cases
11:         let a = "1234"
12:         let b = "12"
13:         let c = "34"
14:         Assert.IsEmpty("") // unit test case 1
15:         Assert.IsNotEmpty("Hello!") // unit test case 2
16:         Assert.AreEqual( a, b+c) // unit test case 3
17:
18:     [<Property>]
19:     let revUnit (x:char) =
20:         List.rev [x] = [x]
21:
22:     [<Property>]
23:     let revApp (x:string) xs =
```

```
24:         List.rev (x::xs) = List.rev xs @ [x]
25:
26:
```

Q23. Can you think of properties that must hold for **`recordPoint`** which you could incorporate into tests? It does not matter if your properties are incomplete. To get you started, for example, if a player has advantage, and they win a point, then they win the game. How you code this will depend on your data-structure for **`tennisGame Score`**. But it will be something like:

```
1:      [<Property>]
2:      let playerWithAdvantageWinsPointMustWinGame (p: Player) (x:TennisGamescore) =
3:          let hasAdvantage p = ... // true if p has advantage in x
4:          if hasAdvantage p x then match recordPoint p x with
5:                                   | Game p' when p'=p -> true
6:                                   | -> false
7:          else true
8:
```

Write a set of property-based or NUnit-based tests for your function **`recordPoint`**. Bring this to the next lecture. Note that sometimes all you can do is write specific properties: given fixed input, specify the output. The function **`playGame`** from Figure 4 in the last section can be used (modifying **`TennisGameScore`** to be your own type definition). For convenience it is given below as the incomplete answer to this question.

Answer

Uses of FsCheck in this module

You should now have FsCheck working correctly on your own laptops. There are two reasons why this will be important later in the module:

- In your project work each person's contribution will need to be properly tested with FsCheck or some other framework.
- During lecture slots there will be some quick-fire assessed programming problems for credit. These can be done without testing, but an FsCheck set of tests will be released with the problem in the lecture - it may be helpful for you to use this.

F# and Object Oriented Programming

F# uses Microsoft's .NET ecosystem of libraries, with which it interoprates seamlessly. You don't have to use .NET since "core" F# is a complete language but practically you will. The .NET ecosystem is object oriented, so it is often necessary to understand how to use objects when interfacing to it. F# is a true hybrid language and you can create classes and objects in F# (slightly more easily than in C#). However classes and objects should be used with care since functional solutions, when possible, are usually more robust and re-usable.

A good introduction to this topic comes from Scott Wlashcin. Scott is writing mostly for those who have already used object oriented programming, so if you have not here is an introduction (one of many such that can be found on the web) using C++ with which you are familiar and which cohabits .NET with F#.

In this worksheet we will look at things you need to know about OOP when programming F#.

Everything is an object

In F# every data item is also an object and therefore you can use methods to process data instead of functions. For example, try:

```
1:  let x = [1;2;3]
2:  let n1 = List.length x
3:  let n2 = x.Length
4:  let s = x.ToString()
```

This code shows you how to access properties (**`Length`**) and methods (**`ToString`**) of the list x, which happens also to be an object of class List, although normally you do not think of it as this. Use intellisense on **`x.`** to find definitions for these, and note the distinction between properties and methods. You can think of properties and being value fields of an object and methods as being functions applied to the object (and possibly other additional parameters.

Note that .NET properties and methods (**`Length`**) always start with capital letters, whereas F# functions (**`List.length`**) always start with lowercase letters.

It is tempting to use methods instead of functions where both are available: **`x.Length`** is so much easier than **`List.length x`**. It turns out that this is not wise. If you have read Scott's note linked above you will perhaps recognise the problem.

Methods (and properties) are typically highly *overloaded*. Thus **`ToString`** and **`Length`** both apply to many different data types. In fact **`ToString`** applies to all objects! Therefore type inference breaks when methods are used.

Q24. Rewrite the function below using suitable methods (which you can find from intellisense) instead of functions.

```
let getEndsOf lst =
    match List.length lst with
    | n when n < 2 -> ""
    | _ -> lst.[0] + lst.[List.length lst - 1]
```

Answer

Q25. Mixing method calls with functions leads to surprising syntax. Consider:

```
let x = [1;2;3]
printfn "%s" x.ToString()
```

This does not parse, and the error message tells you why. Correct it.

Answer

Using Interfaces

In F# data objects that appear to be records can be written instead as classes and made to implement interfaces. Thus a record can also (surprisingly) be made to behave as a sequence by adding to the record definition an implementation of the **`Ienumerable`** interface which defines .NET sequences of data. A good example is found in the **`FAKE`** library, with type **`FileIncludes`**. This is a record that represents a set of file paths with wildcards, and therefore matches a sequence of filenames. When this record is used in a context that expects a **`string seq`** its value appears to be the matching sequence of filenames. In this case the correct implementation of **`Ienumerable`** is used to generate the filename sequence from the record.

Here is the **`FileIncludes`** API reference. You need to access the source code to find the **`FileIncludes`** type definition and its **`IEnumerable`** interface implementation, which defines how the sequence of filenames is generated from the record definition. That then makes sense of this code which would appear to magically turn a **`FileIncludes`** record into a **`string seq`**.

```
#I @"C:\github\FSharp.Formatting\packages\FAKE\tools"
#r "FakeLib.dll"

open Fake  // for the FAKE functions

let test =
    !!("./*.pptx") // Define a FileIncludes record
    |> fun fx -> {fx with BaseDirectory = "lecturesRoot"} // update a field in the record
    |> Seq.toList // convert record (interpreted as sequence) to list
    |> List.map (fun fn -> printfn "%s->" fn; (fn, FileHelper.changeExt ".pdf" fn))

```

Q26. Find the interface definition and see how this works.

(Q26 Answer not supplied)

Ticked Exercise

Write a type definition for a **marking** defined as follows: A set of students, each identified by a unique username string, submit assessed work. Each assessment has a name (given by a string) and results in a mark in the range 0-100. Assessments are divided into two types: *Tick* Test

Tick assessments may have an integer mark which is *normal*, *exceptional*, or *resit*. *Test* assessments simply have an integer mark. A **marking** gives the mark of a given student for a given assessment. Markings are used by processing the list of all markings to determine total students marks. This depends on the mark type (if this exists) and the assessment type.

Submit your type definition without test code as an **`fsc`** file to the Worksheet 3 Tick link.

Reflection

- *Testing* is a key aspect of programming that is now accepted in industry to be central to successful projects.
- Agile development practices (in various forms now almost universal) make the writing of tests at the same time as code central in the development process.
- Debugging (making a program pass its tests - in some cases working out what new test is needed to cover a newly discovered bug) is a very large part of programming effort. Functional languages like F# claim that they allow programs to be written with much less time debugging than imperative languages. Is this true? If so what are the reasons for this?
- Why are pure functions so much easier to debug than OOP objects which encapsulate mutatable data? Does this mean there is no role for objects (and therefore OOP) in comprehensible well-written programs