HLP Worksheet 1: types, values and lists

Contents

How to read these worksheets

These worksheets will lead you through F# language via examples and questions, and you will only need to refer to the language reference when these are unclear. Normally this is **after** you have done the relevant sections of the worksheet. To help you do this language reference is color-coded light yellow (for language elements used in the worksheet) or yellow (for general comments on language). Skip these on first reading.

Material relevant to the worksheet but not directly used is put in boxes:

Relevant material which should be read

Language elements new in worksheet

Language notes not directly relevant

Supplementary or advanced material not directly relevant

Prerequisites

You should have Visual Studio & tools, or Visual Studio Code with the F# standalone compiler, set up to do this Tutorial. F# can also be installed and used under OS-X or linux.

This introduction assumes you have learnt some C++. It does not assume that you are familiar with .NET (the microsoft run-time system and libraries) or OOP in C++.

F# code can be run in *FSharp Interactive* (`FSI`) the F# interpreter, (see here), or compiled to executable code.

Note on F# source files. Code can be written in source files (`.fs`) or script files (`.fsx`). F# code runs the same in both: but scripts can also be run from desktop by double-clicking if you associate the `FSI` interpreter with `.fsx` extension. More information here.

F# Language Reference

You will find the **F# cheat sheet** useful throughout this course:

- F# cheat PDF

- F# cheat HTML

F# is very well documented by the newly revised official documentation and also tutorials on many aspects can be found here. These worksheets mostly provide a self-contained introduction to the language. Where the language syntax or its usage are not completely described in the worksheets the web documentation can be read.

Figures 1,2 and 3 detail the F# syntax used in this first worksheet. **Remember, as noted above, to skip these on first reading**. As you go through the sheet they will summarise for you what you are learning.

Syntax	Name	Description
let name **:** typename **=** exp **let** name **=** exp	let binding	Defines identifier name to be equal to *exp* for all subsequent expressions in current block *typename* is optional because types will be inferred by compiler.
let fname p **=** exp	function let binding	Defines function *fname* with parameter *p* to be equal to expression *exp* which may contain *p*.
let fname **(** p **:** typename **)** **=** exp`	as above	Types can be added to any parameters as shown
(e1 **,** e2 **,** ... **,** en **)**	n-tuple value	*ei* can be arbitrary expressions. Like a record but with unnamed elements and no type definition
[e1 **;** ... **;** en **]**	list	All the *ei* must be the same type. List elements are separated by **;** not **,**.
[e1..e2]	range list	List of all integers from *e1* to *e2*. *e1*,*e2* must be `int`. If given `float` will generate list of `float` separated by 1.0. If *e2* < *e1* the list will be empty.
[e1..step..e2]	range list with step	As above but numbers vary by *step* which may be negative, in which case *e2* < *e1*
fun p1 **->** e1	anonymous function	Identical to *name* after the binding `let name p1 = e1`
(op **)**	functionised operator	*op* must be a binary operator. **(** op **)** **a** **b** is equivalent to **a** op **b**. Mostly used to feed operators to list processing functions

Figure 1. F# elements

Type	Description
'a	*polymorphic* type variable can fit any specific type
int **float** **string** **bool** **unit**	Basic types. **unit** has a single value of **()** and is used when return value is not relevant. Like **void** but returns a value.

Type and identifier naming
In F# **'** is a valid character in identifiers. Identifiers starting with **'** must be types and the **'** indicates a wildcard (polymorphic) type variable as in **'a list** where **'a** stands for any type.

Identical wildcards within a type specification indicate matching types so:
'a list -> 'b list -> ('a * 'b)

Type	Description
T1 -> T2	Type of function with parameter type T1 and result type T2
T1 * T2	Type of 2-tuple with first element type T1, second type T2

Figure 2. F# Types

`list`
is different from:
`'a list -> 'b list -> ('a * 'a) list`

Op	Function	Type	Meaning
+ - * /		`float -> float -> float` or `int -> int -> int`	*overloaded* arithmetic operators use `int` or `float` but cannot mix types
**		`float -> float -> float`	power operator
	`float`	`int -> float`	convert `int` or other numeric type to `float`
	`List.map` `List.map f lst`	`('a -> 'b) -> 'a list -> 'b list`	map `f` over `lst`
	`List.collect` `List.collect f lst`	`('a -> 'b list) -> 'a list -> 'b list`	map `f` over `lst` and concatenate the results
	`List.reduce` `List.reduce f lst`	`('a -> 'a -> 'a) -> 'a list -> 'a`	reduce list to value using `f`
	`List.filter` `list.filter f lst`	`('a -> bool) -> 'a list -> 'a list`	filter `lst` to contain only elements `x` for which `f x = true`
|>		`'a -> ('a -> 'b) -> 'b`	Pipeline operator: `x |> f` = `f x`

Figure 3. F# Functions and operators

Introduction

In a functional language, such as F#, every program is a sequence of constant or function definitions, followed by a single expression.

```
let adj = 3
let opp = 4
let hyp = 5
let square x = x*x
square adj + square opp - square hyp
```

Figure 4. F# program

Figure 4 calculates Pythagoras's Theorem for a right-angled triangle. The 4 values defined using **`let`** are all used in the final line which calculates any error in the expected hypotenuse length (and will always, in a Euclidean geometry world, be 0). In lines 1-3 the values defined look (and behave) like variables in C with one big difference. They are *immutable* and so the values defined can't change.

Figure 5 shows the same program written in C++ for comparison:

```
const int adj = 3;
const int opp = 4;
const int hyp = 5;

float square( float x)
{
   return x*x;
}

float main()
{
   return square(adj) + square(opp) - square(hyp);
}
```

Figure 5. C++ program

Differences between C++ and F# in this example:

- Line 4 of Figure 4, defining the **`square`** function in F#, uses the same syntax as the first three lines because functions are just values like any other. In Figure 5 the difference is obvious.
- Function application in F# does not require brackets **`square(adj)`** around the parameter. In a functional languages two expressions placed side by side **`f g`** will be evaluated as a function **`f`** applied to parameter **`g`** as in **`square adj`**.
- The functional program has no explicit type information! This is not needed because types can be *inferred* accurately by the compiler. See these by hovering your mouse over them in Figure 4[1].
- Let bindings, and expressions, are normally terminated by a newline, **`;`** is not needed.

This is a valid, though simple, F# program. It is not however very well written. The problem is that the real calculation, the final line, has data which is baked in via the previously defined parameters **`adj`**, **`opp`**, **`hyp`**. We need to rewrite this as a function with input a triple of values **`adj, opp, hyp`** and output the error. In F# any sequence of values separated by **`,`** forms a tuple data type and is treated as a single value. This will allow us to rewrite the final line as a function of one 3-tuple parameter using the same syntax as **`square`**.

```
let pythagorasError (adj,opp,hyp) =
   square adj + square opp - square hyp
```

The brackets are needed around the tuple to prevent a parse error because **`,`** binds less tightly than almost anything else.

Note that the function definition has been split onto two lines for convenience. The indentation of the second line tells the compiler it is a continuation of the first.

Let us now put this together, noting that the square function could be a sub-function of **`pythagorasError`** if it is not used anywhere else:

```
1:  let adj = 3
2:  let opp = 4
3:  let hyp = 5
4:
5:  let pythagorasError (adj,opp,hyp) =
6:      let square x = x*x
7:      square adj + square opp - square hyp
8:  pythagorasError(adj, opp, hyp)
```

Figure 6

Note that standard form of the Figure 6 program: again a sequence of let bindings followed by an expression. The body of `pythagorasError` is indicated by the indentation and it now consists of the `square` let binding followed by the expression that determines the return value of the function.

- Note that a function body (lines 6-7) has the same form as a program.
- In line 8 `pythagorasError` looks like a 3 parameter function in C. In F# it has a *single* 3-tuple parameter. We will see later on how to write functions with multiple parameters.
- Whether functions should be written at top level separately, or included as subfunctions, is a matter of style. Normally single-line functions not used elsewhere would always be written as subfunctions.
- Identifiers in F# are case-sensitive. CamelCasing is used to represent multiple words in identifiers: use capitals to mark words instead of `_`.

Types in F#

We have seen in Figure 4 that the types of each identifier do not need to be written by the programmer, thus making the code shorter and easier to read. F# is a strong static typed language, where every identifier has a definite type. Hover your mouse over elements in Figure 6 to discover the types. Note that `->` indicates a function type.

- Discover what is the *type signature* of the 3-tuple parameter to `pythagorasError`.

F# tooltips
The code snippets in this worksheet have been processed by the F# compiler to generate type information that appears on hovering the mouse. Visual Studio does this, with type or syntax errors underlined as you type.

In F# as in C++ the normal arithmetic operators `+`,`-`,`*`,`/`are overloaded and can apply to both `int` and `float` numbers (and some other less common types). The functions `square` and `pythagorasError` have `int` type signatures.

- Use F# interactive to check what happens when you use `pythagorasError` with `float` parameters. returns an error
- Any identifier `id` can be marked with a type as `(id : int)`. Rewrite the Figure 6 code replacing `adj` in `pythagorasError` by `(adj: float)`. How does this affect the compiler-generated type signature for `pythagorasError`?

This example shows that in the special case of arithmetic operators F# is more strictly typed than C++. There is no automatic conversion of `int` to `float` and so the `pythagorasError` function can operate on three `int` or (differently annotated) three `float` but not a combination. The extra strictness here is in fact helpful, it prevents the incorrect use of integers when floating point numbers[2] are needed or vice versa. Complete type specification is not needed, in this case just one of the input types (or the output) is enough for the compiler to work out what they all must be. Subfunctions will correctly determine type from their usage, so normally no types need be specified.

Exercise 1.1

A Pythagorean triple is a set of three integers: $p, q, r : p^2 + q^2 = r^2$. The identity in Equation 1 can be proved algebraically always to hold

Equation (1) $(a^2 - b^2)^2 + (2ab)^2 = (a^2 + b^2)^2$.

For any integer a, b this equation will therefore generate a Pythagorean triple:

$p = a^2 - b^2, q = 2ab, r = a^2 + b^2$

In order to generate a large number of such triples we can apply this for different values of a, b.

In pure functional languages it is not possible to use loops and although F# (having procedural elements) does allow this it is almost never the best way to solve problems. In this case we could start with a *declarative* solution by introducing the **list** data structure that represents (linked) lists. Using this, function parameters or return values can be arbitrarily large data structures. The pseudo-code would be like this:

1. Input a list of 2-tuples (a, b)
2. For each item in the list apply Equation 1 to generate the Pythagorean triple as a 3-tuple.

Note that lists can contain anything, here 2-tuples and 3-tuples, but that all the elements of a single list must have the same type.

We need one new F# operation to code this. A function which will *map* a function **f** over a list **lst** to perform step 2.

```
List.map f lst // return a list of f applied in turn
               // to each element of lst
```

This is a standard **List** module function in F#. Note that F# is case sensitive (**List** not **list** for the module name). You can now write the code:

```
let pairs = [ (1,2) ; (1,3) ; (2,3); (3,4) ; (4,7) ]
let makeTriple (a,b) = ....
let pythagTriples = List.map makeTriple pairs
```

Q1. Complete the **makeTriple** function definition using Equation 1 and previously defined **square**.

(Q1 Answer not supplied)

Q2. What is the type of: **pairs**, **makeTriple**, **pythagTriples**? In each case see if you can guess the answer.

(Q2 Answer not supplied)

Test your code in the F# interactive REPL (`FSI`), checking whether your guessed types are correct.

Visual Studio is an IDE that allows an edit/run/debug cycle when writing code. More importantly it has *intellisense* and *autocomplete*. These two features are invaluable writing code and can best be understood by trying them out.

- Set up a F# VS project as in the install guide with a *console application* `*.fs` F# source file.
- Type the `pythagTriples` code into the source file line by line observing the *intellisense* indications of type and other errors as you type. Hover the mouse over any identifier to get information on its (compiler-inferred) type.
- Go to `List.map` and delete the `.map` function. Typing `.` at the end of `List` will show an autocomplete box of possible `List` library module functions. Highlighting any one of these will provide further data.
- Finally you can get copious data on these functions from MSDN - the fastest way to find this is google search e.g. *fsharp List.map*. Try this.

Q3. Change your working code to make `makeTriple` a subfunction of `pythagTriples`.

(Q3 Answer not supplied)

Running as a console application the program requires some extra code to work; a `print` expression and something to prevent the program from terminating prematurely before you have viewed the output.

```
printfn "Triples are: %A" pythagTriples
System.Console.ReadLine() // prevent the program from terminating
```

- `printfn` works like C `printf` with a newline (`printf` is also available). Note that `%A` will print out *any* type in a sensible printable form. You can also use type-specific print specifiers: `%s`,`%d`, `%f` etc.
- Printing multiple items requires each expression to be printed to be placed after the `printfn` specifier string. Compound expressions need to be bracketed.
- Multiple expressions in F# are processed sequentially as here `printfn` before `ReadLine()`. All expressions except the last must return the `unit` type which has a single value `()` and is used when no return value is expected. Printfn returns `()` since its only purpose is the side-effect of printing.

```
printfn "Print the first 3 powers of 2: %d, %d, %d." 2 (2*2) (2*2*2)
printfn "Look at this Pythagorean triple: %A" (makeTriple(3,4))
printfn "Pairs are: %A\n, triples are: %A." pairs pythagTriples
```

Check that your program works as expected using suitable print statements. Note that the `makeTriple` function application needs a bracket to force the `makeTriple(3,4)` function call - unlike in C++. Without the bracket it will be parsed as two separate parameters to `printfn`: `makeTriple` and `(3,4)`.

Exercise 1.2

What if we want to generate the list of pairs automatically - say as all pairs $(a, b) : 1 \leq a \leq 5, 1 \leq b \leq 5$? In the spirit of declarative coding - writing what you want not how to get it - we could write pseudocode:

1. Let `intList` = the list of all integers from 1 to 5.

2. Let **pairList** = the list of all 2-tuples (a, b) for $a, b \in$ **intList**
3. Let **tripleList**= the list of all 3-tuples got from applying Equation 1 to **pairList**.

Let us work through how we would write these steps in F#.

Step 1

```
let intList = [1..5]
```

- **[a..b]** denotes the list of numbers from **a** to **b** inclusive. In this notation **a** and **b** can be any valid expression: **[x + 1..x + square a]**.

Step 2

The pairs we want consist of 2-tuples such that for each $a_0 \in \texttt{intList}$ we have, for each $b \in \texttt{intList}$, a distinct 2-tuple (a_0, b).

Let us write that as subfunctions concentrating on the *data* and how it is *transformed*.

```
let tmp1 b = (a0,b) // turn b into the pair (a0,b)
```

Transforms b into the pair (a_0, b) for given a_0. Think of this as a picture: $b \rightarrow (a_0, b)$

```
let tmp2 lst = List.map tmp1 lst
```

This transforms the input list **lst** into the set of pairs $(a_0, x)\forall x \in \texttt{lst}$

$$\begin{aligned}1 &\rightarrow (a_0, 1)\\ 2 &\rightarrow (a_0, 2)\\ 3 &\rightarrow (a_0, 3)\\ 4 &\rightarrow (a_0, 4)\\ 5 &\rightarrow (a_0, 5)\end{aligned}$$

Notice how **tmp1** is applied once for each element of the input (**lst = [1..5]**) to form the corresponding element of the output list.

Putting this together we get the long-winded decomposition of **makePairs**:

```
let makePairs a0 =
    let tmp1 b = (a0,b)
    let tmp2 lst = List.map tmp1 lst
    tmp2 intList
```

To emphasise what **tmp1** does we will write it directly using anonymous function syntax. Compare these two functions:

```
let tmp1 b = (a0,b) // normal function with name
fun b -> (a0,b) // anonymous function: can be used directly in an expression.
```

We can now rewrite **tmp2** directly without needing the name **tmp1**, in a way that emphasises the transformation of data:

```
let tmp2 lst = List.map (fun b -> (a0,b)) lst
```

Notice the brackets around the anonymous function are **nearly always** needed in F# to make clear where the function body stops.

```
let makePairs a0 =
    let tmp2 lst = List.map (fun b -> (a0,b)) lst
    tmp2 intList
```

> **Q4**. We have key F# syntax. Functions are so-called **first class objects** and we can define a function like **tmp1** in a familiar way with name, parameters, body, or as a value where the function itself is written without a name: **fun b -> (a0,b)**. In **tmp1** does it matter if **a0** is changed to another variable name? If **b** is changed to another variable? Is there any difference in meaning between the two definitions of **tmp1** (as a value or a function)?
>
> Answer

Finally we could repeat the same trick with **tmp2**. In this case we do not need another anonymous function because F# allows functions to be partially applied, so we can use **List.map** with its first parameter given and its second parameter missing:

```
let tmp2 lst = List.map (fun b -> (a0,b)) lst
let tmp2 = List.map (fun b -> (a0,b))
```

These give **tmp2** exactly the same value. This is more than a syntactic trick. In F# the function **List.map** uses two parameters **f** and **lst**. We can however also use it with only its first parameter when defining **tmp2** as above, where it is set equal to the "incomplete" function call . For example when, later, the second parameter is supplied:

```
let tmp2 = List.map (fun b -> (a0,b))
tmp2 [1..5]
```

the **List.map** function will execute correctly with first parameter **(fun b -> (a0,b))** and second parameter **[1..5]**. We will learn the details of how this process of *partial application* works in Workshop 2.

```
let makePairs a0 =
    let tmp2 = List.map (fun b -> (a0,b))
    tmp2 intList
```

Rewriting **tmp2** as a partial application of **List.map** as above we can see that the **tmp2** name is unnecessary. Removing this name and substituting the value of **tmp2** directly we have turned an initial 3 lines into one which (in this case) is clearer because the operations are simple enough:

```
let makePairs a0 = List.map (fun b -> (a0,b)) intList
```

or

```
let makePairs = fun a0 -> List.map (fun b -> (a0,b)) intList
```

We now completed first part of step 2, which generates a 5 element list of pairs (a_0, x) from a number a_0.

Step 2 is completed by *concatenating* the results got from applying **makePairs** to each element $x \in \texttt{intList}$ in turn:

$$\left.\begin{array}{l} 1 \rightarrow \texttt{makePairs}\ 1 \\ 2 \rightarrow \texttt{makePairs}\ 2 \\ 3 \rightarrow \texttt{makePairs}\ 3 \\ 4 \rightarrow \texttt{makePairs}\ 4 \\ 5 \rightarrow \texttt{makePairs}\ 5 \end{array}\right\} \quad \text{concatenate result lists}$$

This operation is similar to **map** and called **collect**. To see the difference compare in **FSI**:

```
List.map (fun a -> [a ; a+1 ; a+2]) [10 ; 20]
List.collect (fun a -> [a ; a+1 ; a+2]) [10 ; 20]
```

Check the types of **List.map** and **List.collect**. The F# type system will prevent typical errors, for example a function to return a new list containing only the positive elements of **lst** might be incorrectly written as:

[a]

```
    let retainPositive lst = List.collect (fun a -> if a > 0 then a else []) lst
    retainPositive [ 1; -2; -4; 0; 11] // should return [1; 11]

```

Q5. Enter **retainPositive** and check the type error. Did you predict this from the code? Correct the function.

Answer

We have not quite finished this exploration of functions. Consider:

```
let intList = [1..5]
let makePairs a0 = List.map (fun b -> (a0,b)) intList
let pairList = List.collect makePairs intList
```

Both **makePairs** and **pairList** have **intList** *baked in*. This works fine but it would be more flexible to generalise by turning the baked in value into a parameter, e.g. **lst**:

good example to generalise the code

```
let makePairs a0 = fun lst -> List.map (fun b -> (a0,b)) lst
let pairList = fun lst -> List.collect (fun a0 -> makePairs a0 lst) lst
```

Q6. In this worksheet we are avoiding user-defined functions that have two parameters (like `List.map`). One parameter functions have enough complexity to start! However, if you wrote the above `makePairs` definition as a function of two parameters:

```
let makePairs lst a0 = ...
```

it simplifies this code. Do this.

Answer

Step 3

```
let makePairs lst = fun a -> List.map (fun x -> (a,x)) lst
let pairList = List.collect (makePairs intList) intList
let makeTriple (a,b) = ... //as before
let tripleList = List.map makeTriple pairList
```

Q7. What is the difference in type between `tripleList` and `makeTriple` as defined above?

Answer

This code now generates the answer as a set of transformations. Each line uses the result in the previous line and when testing these intermediate results can be printed out with `printfn` to check.

Q8. Suppose you had made a mistake and used `List.map` in `pairList`. Would this lead to a type error, and if so where?

Answer

Q9. Test the final solution, as a working program printing out the answer. Check that you understand the type and value of each named intermediate computation.

Answer

Q10. Rewrite `let makePairs a b = (a,b)` without changing its meaning as `let makePairs1 =...` by using an anonymous function. Check that `makePairs1` works as expected.

(Q10 Answer not supplied)

How could we rewrite the code to print a set of lines each containing a pair and the corresponding triple? Thinking declaratively we want:

- Write a new function `multiList`, using `makeTriple` to return a list of 2-tuples each containing a pair and its corresponding triple, when given the list of pairs.
- Make a function `printMultiList x` that will print out one element `x` of the new `multiList`
- Map `printMultiList` over `multiList` using `List.map`

Capitalisation in F#.

All identifiers are `camelCased` to indicate words and underscore `not_camel_case` is discouraged. The case of the *first letter* of an identifier is confusing because F# conventions and .NET (Microsoft C,C# etc) conventions are different. In F# *user-defined types* should always capitalised. Functions and values are not normally capitalised. However common types shared with C and C# `System.Int32`, `System.Double` etc have abbreviations `int`, `float` which are always lower-cased. Similarly the common F# types: `list`,`set`,`array`,`map` ,`string` have lowercase abbreviations which are usually used. Note that *module* `List` contains functions (e.g. `List.map`) on *lists* with *type*

Q11. See if you can do this, looking at the answer if you need inspiration.

Answer

constructor abbreviation **list**. For now you can follow the rule: functions and standard types are lower-cased, module names are upper-cased.

Standard module functions which are part of .NET and not in the F# core are capitalised, as are OOP methods which can be used in F#. All of these are used very rarely in this course. Thus **System.Console** is a .NET module of I/O functions and these (e.g. **ReadLine**) are capitalised.

The standard Fsharp modules for types: **List**,**Map**,**Array**,**Set**,**String**contain functions (**String.concat**, **List.map**) which are never capitalised. User functions (**makeTriple**) should normally **not** be capitalised to distinguish them from user types (used in later Tutorials).

Use autocompletion on the module name to check standard function capitalisation if not sure.

The triples produced so far are redundant since (a, b) and (b, a) generate the same triple and (a, a) is not interesting. Let us correct this by writing a function that returns only the pairs in **pairList** for which $a < b$. There is a library function under the List module that will perform the necessary *filtering* of a list. Find it using *autocompletion* on **List.**. List.filter

Q12. Work out the (single line) function **deleteBadPairs x** which filters a list of pairs according to the function you have found. You need to know that the operators **>**,**<** work as expected on integers to return a **boolean** value.

Answer

The code so far constructs **pairList** and performs three operations on it in sequence:

1. **deleteBadPairs**: filter elements where $a \geq b$
2. **multiList**: turn pairs into Pythagorean triples (in a 2-tuple with the original pair)
3. **printTriple**: print each pair with its triple

Each step is a function and what is needed is just:

```
List.map printTriple (multiList (deleteBadPairs pairList)))
```

the above code can be written with the pipeline operator as:

```
pairList |> deleteBadPairs |> multiList |> List.map printTriple
```

You can see how the pipeline equivalent is easier to read because:

- there are no brackets
- the functions are written left to right in the order they are used.

Q13. Rewrite this code:

```
let pairList = List.collect (fun a -> List.map (fun b -> (a,b)) [1..5]) [1..5]
let makeTriple (a,b) =
    let sq x = x*x
    (sq a - sq b, 2*a*b, sq a + sq b)
let tripleList = List.map makeTriple pairList
printfn "Triples are:%A" tripleList
```

using a pipeline. (There is more than one way to do this, see the answer for one option).

Answer

F# has "standard formatting guidelines" which help to keep code tidy and readable. Visual Studio comes with FSharp Power Tools (see install instructions). If you have these installed, type `Ctrl-K Ctrl-D` in Visual Studio type to re-indent and format the file being edited, or `Ctrl-K Ctrl-F` to do this with a highlighted part of the code.

Exercise 1.3

Write a function `expo x n` to calculate the first `n+1` terms of the Taylor series for exponential $(n \geq 0)$:

$$\text{expo}\ x\ n = 1 + \sum_{i=1}^{n} \frac{x^i}{i!}$$

Use the new `List` function: `List.reduce: ('a->'a->'a) -> 'a list -> 'a`:

```
List.reduce f lst // use f to combine elements of lst
                  // List.reduce f [1 ; 2 ; 3] = f (f 1 2) 3
                  // f = plus => sum the list
                  // f = times => product of the list
```

You will also need:

- `** : float -> float -> float` defined as `a ** b` $= a^b$
- `float : int -> float` which converts `int` type to `float`. `float a` is the floating point equivalent of integer `a`.
- `let times a b = a * b` convert `*` (also `+` etc) into a function to pass to `List.reduce` .

It is possible to solve the problem without `**`. Can you see how to do this?

```
let expo x n =
   let sum a b = a + b
   let times a b = a * b
   let term i = x ** (float i) /
                List.reduce times ([1.0..float i])
   1.0 + ([1..n] |> List.map term |> List.reduce sum)
```

You could implement `x ** i` as:

```
[1..i] |> List.map (fun j -> x) |> List.reduce times
```

or more nicely using a *don't care* for the parameter that is never used:

```
[1..i] |> List.map (fun _ -> x) |> List.reduce times
```

Shortcuts

- Any binary operator `+` can be turned into a function using brackets around the operator `(+)`.
- The `(+)` and `(*)` functions can be written directly into the code without names:

```
let expo x n =
   let fRange i = [1..i] |> List.map float
```

```
    let term i = x ** (float i) / List.reduce (*) (fRange i)
    1.0 + ([1..n] |> List.map term |> List.reduce (+))
```

- This function is complicated because of the necessary conversion of **`int`** values to **`float`**.
- The **`1 +`** is necessary because **`List.reduce`** will not work on zero length lists. Next tutorial you will learn **`List.fold`** which will work on zero length lists and allow a neater solution.

Exercise 1.4

Q14. Write a function **`raggedList n`** which when given an integer **`n`** returns a *ragged array* represented as a list of lists. The list has **`n`** elements representing array rows the **`i`**th element of which is a list **`[1.0..(float i)]`** (counting elements from 1 upwards).

```
    raggedList 3 =
    [ [1.0] [1.0 ; 2.0] ; [1.0 ; 2.0 ; 3.0] ]

```

Answer

Q15. write a function **`stats: float list -> float * float * float list`** which returns a 3-tuple **`(average of x, variance of x, x)`** when given a list **`x`**. Test this by applying it to every row of **`raggedList 5`**.

Answer

Q16. Use auto-complete on **`List.`** to find **`List.sumby`**. Simplify **`stats`** using **`List.sumBy`**.

Answer

Ticked Exercise

Write an F# function named **`sineSeries`** of type **`(x:float) * (n:int) -> float`** which implements the equation below for $n > 0$:

$$\sin x \approx \sum_{i=0}^{n} (-1)^i \frac{x^{2i+1}}{(2i+1)!}$$

Your solution need not be execution-time efficient. Submit your function definition without test code as an **`fsx`** file to the Worksheet 1 Tick link.

Reflection

- The core constructs of F# form a declarative language which is simple compared with procedural languages. There are no assignments, no statements, no need even for explicit types, although static type checking is strong using *inferred* types.
- Recasting programs with declarative programming generates results without loops or even, usually, recursion[3]. What is surprising is that so much programming can be done without any of these techniques!
- We can see from the examples here some of the reasons why functional programming is a useful technique:
  - It is easy to understand programs which read like maths

- The start and endpoint of loops are always implicitly given as the length of the list being processed.
- Type errors catch nearly all programming mistakes, without the extra programmer burden of specifying types.
- This creates very robust and error-free programming.
- The rest of this course will be looking at the other concepts and language constructs[4] that relate to programming in F# specifically, and more generally other high level languages.

Further Reading

Finally, some future matters. This Worksheet is an *operational* introduction to basic F# programming. It hints at many issues, both language and conceptual, which you meet here but do not fully explore. If you want clarification of any specific detail of the **F# language** start with the MSDN language reference. Relevant pages in this are most easily found from google search.

Key concepts and features of F# touched on here will be covered in detail later:

- *Functions with more than one parameter:* `f x y` *as distinct from* `f (x,y)`. See the next tutorial for the concept of *currying* and high order functions.
- *Polymorphic types*. This tutorial uses them (in list functions: `('a -> 'b) -> 'a list -> 'b list` before they have been precisely defined.
- *Numeric types*. There are (complex) ways to make F# functions that will work with *both* `int` and `float`. Many languages have a *Number* type which contains both *int* and *float*. More later.
- *List functions*. `map` and `reduce` (and `fold`) will be explored in Workshop 2.
- *Impure Code*. This introduction teaches pure functional code (no assignment, no loops, functions are defined only by their value) but even so, in order to print results, we have had to depart from that regime with `printfn`. How best to deal with *side effects* such as printing, or assignment, in a functional language is an interesting topic we return to later.

Extensions

- E1. Modify the code for exercise 1.2 so that is prints out only *co-prime* Pythagorean triples. For example, if (p, q, r) is a triple, and n is an integer, then (np, nq, nr) is also a Pythagorean triple but not very interesting. You can determine whether a triple has any common factors using the following function which computes the greatest common divisor of two integers:

```
let rec gcd a b =
  if b = 0
  then abs a
  else gcd b (a % b)
```

- E2. Find Pythagorean triples directly by filtering a list of pairs (a, b) according to whether $a^2 + b^2$ is a perfect square. Determine whether there is a perfect square by filtering a list of possible integer square roots. It turns out this "brute force" method properly implemented is very compact. Can you find any Pythagorean triples not of the form used in this exercise?
- E3. The implementation of the sine function Fourier series here is very inefficient because each term can be calculated from the previous term more easily than from scratch as here. What is the time order (where the size of the problem is the number of series terms) of the function suggested here? What is the time order of of an efficient iterative implementation? Can you see how to do the iteration efficiently using a single recursive function? Hint - use one parameter for each loop variable in the iteration.