# Functional Programming with F#

Yusuf M Motara

2021 Edition

# Contents

# Notation and conventions

We will use the following notation in this textbook:

- `keyword` is text that must be written exactly as it is on the page.
- `pattern` is replaced in actual code by a pattern (see page 11).
- `ident` is replaced in actual code by an identifier.
- `type` is replaced in actual code by a type.
- `expression` is replaced in actual code by an expression.
- ↓ indicates where a line can, optionally, be broken. To indicate that code on subsequent lines is part of this line, it is indented.
- `optional` indicates a part of the syntax that is optional.

Syntax[1] is defined in a box like this:

**Syntax: construct**

Examples appear in a box like this:

```
```

Common misunderstandings that first-time functional programmers have will appear like this:

**Description**

Important terms which should be remembered with their meaning are written ✪**like this**. You can look up brief definitions of such terms in the Glossary on page 77.

## Whitespace conventions

Whitespace is significant in F#; in other words, the indentation and spacing of code can affect its meaning. Indentation usually begins a new scope.

Expressions within a new scope are placed on a new line and indented, usually by 2-4 spaces[2].

```
fun x ->
  let k = 21
  k + x
```

The only exception to this is single-line lambda functions.

```
fun y -> y - 5
```

Multiple-step single-line expressions are typically broken up over multiple lines, with an operator being used to start each line.

```
[3..200]
|> List.map (fun x -> x * 3)
|> List.filter (fun x -> x % 5 = 0)
|> List.length
```

More complex code tends to use slight variations on the above conventions.

```
List.mapi (fun i (v,p) ->
  let w =
    (v+p)*i
  w, v*w
) |> List.filter (fun (_,n) -> n < 0)
|> List.fold (fun s (a,b) -> s+a+b) 4
```

Here we have an example that shows single-line expressions and multiple-line expressions in a multiple-step single-line expression. Notice the fifth line, where a small variation to the above conventions can be seen. The code could just as easily have been written as follows:

```
List.mapi (fun i (v,p) ->
  let w =
    (v+p)*i
  w, v*w
)
|> List.filter (fun (_,n) -> n < 0)
|> List.fold (fun s (a,b) -> s+a+b) 4
```

Always double-check to make sure that your code is correctly indented.

[1] Syntax in this textbook is opinionated. Try to follow it exactly while you're learning; it tries to split up concepts visually and give you a good foundation. When you start to write your own functional programs, after the textbook is done, you can find a style that suits you.

[2] Technically speaking, single-line expressions *can* occur on the same line. But we will follow the stricter coding standard of using newlines while you are learning, to make scopes easier to visually identify.

# Setup

You can experiment with F# by using anything from the command-line to an online editor; however, for the purposes of this module, we'll be using Visual Studio Code and the .Net Core Software Development Kit. Both of these are cross-platform. You can download and install them from:

- https://dotnet.microsoft.com/download (choose .NET 5.0)
- https://code.visualstudio.com/download

Installation of the necessary components may take a while, depending on machine speed and internet connection speed.

## F# files

You can create a new F# project using the command:

- **dotnet new console -lang F# -n** *ProjName*

Once you are done, you can open it by going to **File** → **Open Folder...** in Visual Studio Code. In the **Program.fs** file, you should see code like:

```
// Learn more about F# at http://docs.microsoft.com/dotnet/fsharp

open System
⊙ ⟵ Your Code Goes Here
// Define a function to construct a message to print
let from whom =
  "from %s" whom

[<EntryPoint>]
let main argv =
  let message = from "F#" // Call the function
  printfn "Hello world %s" message
  0 // return an integer exit code
```

The `main` function is what gets evaluated when the program is compiled and run. Compilation, testing, and running can be done using the **dotnet** task runner. For now, don't worry about this: we will not be running code by compiling programs at first. Instead, we will be using F# Interactive.

## F# interactive

You will find yourself writing many small, independent functions and pieces of code in F#. One of the best ways to test these out and to explore the language is via F# Interactive, which is a Read-Execute-Print Loop (REPL) environment that comes with F#[1]. The majority of the examples in this textbook can be typed into F# Interactive so that you can play around with them, so getting familiar with the environment will make the content of this module much easier to learn.

If the F# Interactive session gets messed up in any way, you can reset it by clicking on the "bin" icon at the top-right of the F# Interactive panel.

You can use F# Interactive in two ways, and you can combine these ways as well.

### Send to interactive

Highlight a section of code in your file and press Alt + Enter to send that code to F# Interactive for evaluation. Any identifiers that are bound will be available for use in the interactive session. This will probably be the most common way that you use F# Interactive.

### Interactive evaluation

F# Interactive can be used as an interactive language evaluator. Type in the *full code* that you want to evaluate at the prompt, then indicate that the code entry is complete by typing `;;` and pressing Enter. The code that you have typed in will be evaluated, and a result will be printed.

```
> let simba = "Lion"
simba + " king"
;;
val simba : string = "Lion"
val it : string = "Lion king"
> simba;;
val it : string = "Lion"
```

This example shows you the interactive way to use F# Interactive. Terminal output is displayed in **bold**. "it", in the output "val it : ...", is what F# Interactive calls the final expression that it evaluates.

[1] If you'd like to use the command prompt, the **dotnet fsi** command is what you're looking for.

# Introduction

This textbook tries to teach you functional programming, using F# as the language of instruction. F# is a multi-paradigm functional-first language: this means that it supports functional programming natively as well as the "usual" constructs such as objects and classes and interfaces and mutable variables and so on, which you may be familiar with from imperative programming. Since we are learning functional programming, we will *only* be using the functional parts of F#.

There are many different functional languages such as Swift, Ocaml, Haskell, Elm, Purescript, Racket, and Erlang. Some of these have strong typing, and others don't. Some of them have algebraic data types, and others don't. But all of them follow certain principles, and their strengths and weaknesses are directly related to these principles.

## Expressions

An ✪**expression** is anything that, when evaluated, generates a ✪**value**. In fact, we can think of a value as an expression that does not change its form when it is evaluated.

You may encounter *non-executable* code which defines a data type, or links a value to an identifier, or organizes code into different namespaces. However, all *executable* constructs are expressions. There is no functional equivalent of an "if-**statement**" or similar construct which evaluates something but *doesn't* generate a value.

## Immutability

A ✪**value**, once created, can *never* be altered.

## Functions

A function is anything which accepts exactly one input value and, when this is supplied, evaluates an expression to generate exactly one output value. The ✪**definition** and ✪**application** of a function are separate. Code within a function is only executed when the function is applied.

All functions are ✪**first-class**: they can be used in any context that a value can be used. It can be created, it can be passed to another function, it can be given a name, it can be returned as the result of an expression, it can be part of a data structure, and so on.

## Referential transparency

An identifier, once it has been linked to a value, will always refer to that value. An identifier can therefore always be replaced by its value without changing the meaning of the program.

**Immutability ≠ referential transparency**

**Immutability** means that a value can never be changed.

**Referential transparency** means that the link between an identifier and a value can never be changed[a].

These two concepts are often confused by beginners. Make sure that you understand the difference!

[a] However, a *different* link between the identifier and a value *can* be set up. This is called *shadowing* (see p. 11).

## Purity

A ✪**pure function** must fulfill two conditions:

1. its output depends *only* on its input; and
2. it has no ✪**side-effect**s

There is a continuum of purity: a language can be completely pure (such as Haskell or PureScript), or mostly pure (such as Ocaml or F#). Completely pure languages do not allow *any* impure functions to be written, and mostly pure languages allow them but discourage them.

### Depending only on input

A function, when given an input, will execute some instructions and generate an output value. Neither the execution of those instructions nor the generation of the output value can depend on anything other than the input to the function.

A function is still pure if it *doesn't* use the input. However, in this case, the function must necessarily generate a predetermined value, since it cannot possibly depend on anything other than its input.

A function is still pure if it uses constant values within it (such as `true` or `0.8`). Similarly, a function remains pure if it uses identifiers[1] that exist in the envi-

[1] We assume, of course, that the identifiers are referentially transparent, and that the linked values are immutable!

ronment where it is defined. Since nothing can change about these identifiers or the values that they refer to, the situation is equivalent to using constant values.

As a guideline, ask yourself whether the function can ever possibly return a different result when given a particular input. If the answer is "yes", then it must depend on something other than the input, and it cannot be pure. Common examples of such dependencies are mutable global variables, random number generators, user input, countdown timers, files, and date/-time lookups.

## No side-effects

A side-effect is an observable interaction of the function with anything outside of the function. The substitution of a generated value at a call site occurs *after* the application of a function has been completed, and is therefore not an interaction of the function with anything outside of the function. The only way in which the function should be able to affect anything at all is by returning a value; *all* other interactions that affect the external environment are considered to be side-effects.

Common side-effects include: altering a mutable externally-visible variable, writing data to a file, printing or displaying something on the screen, and sending data across a network.

#### Why would you program like this?

Programming would be so much easier if we could change values, and write functions however we wanted to! So why would we want to *not* do all of that? There are two reasons.

Firstly, if we program functionally, then complicated code becomes incredibly easy to reason about. When you are reading a line of imperative code and you see a variable, you have to always think to yourself: "What will the value of this be now? When and where was it last changed?". If you program functionally, identifiers are referentially transparent and values are immutable, so you *never* need to think about this. In fact, students who struggle with functional programming often ask "What will the value be now?" without realizing that the question doesn't make any sense. An identifier will *always* refer to the value it was initially bound to!

Secondly, think of a complicated imperative program. Perhaps some code in it looks like this:

```
SpecialObject a = N();
Whatever b = Mirror(a, N());
SpecialObject c = Combine(b, a);
```

Wouldn't it be great if we could take this code and reduce it to a single line?

```
Combine(Mirror(N(), N()), N());
```

However, we can't do that. What if `N()` changes each time? What if `Mirror` changes something in `a`, or affects the next call to `N`? Replacing variables with initially-assigned values, or executing a function more than once – even with the same arguments – might change the meaning of the program!

In functional programming, none of these problems occurs. We can always replace identifiers with their values, just as we would in an algebraic equation, so we can make our programs smaller and more efficient very easily. In fact, we can go a step further: we could calculate `N()` *once*, and we can use it three times — and we are guaranteed that none of these optimizations will ever change the meaning of the program!

# Exercises

1. A friend shows you their code and says that a particular value is *immutable*. What does this mean?
2. What is the difference between an *expression* and a *statement*?
3. What is a *function*?
4. What are the defining characteristics of a *pure function*?
5. For each of the the following C# methods, determine whether the method is *pure*. Give a reason for your answer.

   (a)
   ```
   int Register(string t) {
     return t.Length - 5;
   }
   ```

   (b)
   ```
   int Tristan(string t) {
     Console.WriteLine("X: {0}", t);
     return t.Length - 5;
   }
   ```

   (c)
   ```
   int Ark(int[] q) {
     if (q==null || q.Length==0) {
       return -1;
     } else {
       q[0]++;
       return 1;
     }
   }
   ```

   (d)
   ```
   int YesNo(int g, int v) {
     while (g > 3) {
       v += 4;
       g--;
     }
     return g;
   }
   ```

   (e)
   ```
   bool Decide(DateTime d) {
     if (d.Year > 944) return true;
     return false;
   }
   ```

6. If variables in a language are *referentially transparent*, what does that mean?
7. What is a *first-class* function?
8. What is the difference between *referential transparency* and *immutability*?
9. Write down the word or phrase that means "something that a function does *in addition to* generating an output value".

# Interlude: Going Functional

Since the start of your Computer Science education, you've probably been told that *variables* and *assignment* are important, and that *iteration* is done with a loop, and that methods don't have to have a return value, and so on. Perhaps you've even done some low-level work where you programmed with registers and opcodes and memory locations, so you *know* that these things are incredibly important. And now, here is this textbook to tell you that you could be computing without all of these things. You might be asking yourself: "What madness is this?"

It's not madness at all. Functional code is compiled down to registers and opcodes and memory locations and so on, just like imperative code—but that doesn't mean that those things need to be exposed within the higher-level programming language. Many problems can be solved in a better way if you aren't worrying about variables which can change their values. This module will teach you to program, design, and *think* in a different way that will help you in your career as you encounter different problems.

But what does functional-style F# code look like, and how is it different from imperative code? To find out, we'll look at the problem of calculating the average of a list of numbers[1]. First, let's look at some imperative Python code that does this:

```
def averageOf(numbers):
  sum = 0
  count = 0
  for n in numbers:
    sum = sum + n
    count = count + 1
  return sum / count
```

I'm sure that you can follow the logic of this already. Now let's take a look at what this might look like in F#:

```
let averageOf numbers =
  let rec avg count sum numbers =
    match numbers with
    | n::rest ->
      avg (count+1) (n+sum) rest
    | [] ->
      sum / float count
  avg 0 0.0 numbers
```

We would write `averageOf [9.5; 33.0; -8.2; 14.0]` to calculate the average of a list of numbers. We'll cover all of the syntax later on, so don't worry about it too much. What's important now is for you to see some real, working code, and to try to understand *how* (or even *why*) it works.

[1]There's a much easier way to do this in F#, but (for now) we'll show you the way to do it without using any built-in functions.

Let's go through it, line-by-line.

`let averageOf numbers` defines a function called `averageOf` that takes an input called `numbers`. So far, that's just like the first line of the Python code. You should also be able to see that, just like Python, whitespace is significant.

`let rec avg count sum numbers =` is a bit more involved. It defines another function, called `avg`, and the `rec` keyword means that we're expecting to call this function again from within itself. In Python, we use a `for`-loop to repeat our actions; in F# (and other functional languages) we call a function **rec**ursively[2] to repeat our actions.

Although `avg` looks like it takes 3 inputs, it doesn't. I'll explain more about this later (pages 17-24) but in this particular case it doesn't matter, so we can treat it as if it is a 3-input function.

`match numbers with` begins a *match expression* (pages 27-29 & 11-15). This is used instead of an `if`-construct in this module. Each "branch" of the match expression begins with a `|`, and if the pattern for that branch matches, then the code after the `->` gets executed in a nested scope.

`n::rest` sets up an list pattern (pages 11-15 & 55-58) that causes the first element in the list to be bound to the name `n`, and the remainder of the list to be bound to the name `rest`.

Then the `avg (count+1) (n+sum) rest` part is executed. This calls `avg` again, but with different inputs:

- `count+1`, which represents the number of elements seen so far in the list;
- `n+sum`, which represents the total sum of the elements seen so far;
- `rest`, which represents all the elements after the first element of the list.

Eventually, because we keep taking the first element and calling `avg` with the rest of the elements, there will be no more elements in the list! At this point, the pattern `n::rest` will fail to match the list, because that pattern can only match a list with at least one element. If the pattern fails to match, the next pattern will be tried.

`[]` is a list pattern (pages 11-15 & 55-58) that only matches an empty list. If the list we are given is empty, then we must have reached the end of the numbers, so we use the `sum` and `count` to calculate the average. However, `sum` is a floating-point number and the `count` is an integer, and we can't divide values that have different types (pages 49-52). We therefore use the `float` function to create a floating-point value from the integer value, and then divide. The result is what is "returned" by the match expression. The

[2]i.e. from within itself

`sum / float count` code, unlike the line above it, is *not* recursive. Once it is executed, the call stack can unwind and the final value returned by `avg` is therefore the result of that last division.

Of course, the `avg` function has to actually be called from outside of itself, not just defined! `avg 0 0.0 numbers`, the very last line of the F# code, does this. So, at a high level, you can look at the `averageOf` function as doing two things:

1. Defining a function, `avg`, which implements our averaging algorithm.
2. Calling this function with some initial values to start the computation.

Can you see how we have achieved the same goal as the imperative code, while obeying all the principles stated in the Introduction?

There have been a lot of unfamiliar ideas introduced in this section: don't worry if you don't understand all of them right now. After reading through other parts of this textbook, come back and read it again. Each time that you go through it, aim for greater understanding.

# Basics

Each value in the F# language has a type associated with it. The basic types are `int`, `float`, `bool`, `string`, `char`, and `unit`.[1] F# does not implicitly convert between types, even when it is "safe" to do so. This means that an expression such as `3 + 4.0` is illegal.

The `unit` type may be unfamiliar to you. Unit has only one possible value: `()`. Unit values are equal to other unit values. Unit values are very useful in conjunction with functions, so we will talk about them in the chapter on functions (pages 17-24).

A `true` value is considered to be greater than a `false` value.

Types are ✪**inferred** from their context. This means that F# will attempt to prove—not guess!—what the type of an identifier *must* be. For now, you can understand this to mean that you often don't need to write down the types of identifiers; F# will do that for you. Types and typing are covered in more detail on page 49.

A single-line comment is started with the `//` characters. Multi-line comments are started with `(*`, ended with `*)`, and may be nested.

Identifiers should start with a lowercase character or underscore, and can include single-quotes (`'`). An identifier can also be any text enclosed in double-backticks (e.g. ``` ``this is an identifier!`` ```).

## Operators

Arithmetic operators are the usual familiar ones: `+`, `-`, `*`, `/`, and `%`. The `+` operator can also be used to concatenate strings.

Logical operators are `&&` and `||`. Instead of a "not" operator, there is a built-in function called `not` which inverts any boolean value that it is given.

Some comparison operators are familiar: `<`, `>`, `<=`, and `>=`. The "is equal to" operator is `=`; note the *single* equal sign! The "is not equal to" operator is `<>`.

F# also includes some operators that are useful for joining functions together, and for manipulating lists. These will be introduced in later chapters.

Brackets in F#, just as in algebra, are only used to separate logical groups or clarify the order in which operations are performed. They have no special meaning other than this.

## Patterns

A ✪**pattern** expresses a possible shape for a value. ✪**Pattern-matching** is the process of looking into a value and seeing if it matches a specified pattern. If pattern-matching is successful, then parts of the value will be ✪**bound** to any identifiers which were specified in the pattern. Binding is the only way in which values can be linked to names, and it is therefore impossible for a name to exist without a value. The process of looking into a value is called ✪**structural decomposition**. If any part of the value does not match the pattern, then pattern-matching fails—and, consequently, no binding occurs.

Pay careful attention to syntax in this text: wherever you see code formatted like `this`, that is a place where pattern-matching and binding can be used. There are four important differences between "assignment" and pattern-matching & binding:

1. Assignment cannot fail. Pattern-matching can.
2. Assignment must assign a value to at least one variable. Pattern-matching does not necessarily result in any identifiers being bound.
3. Assignment always makes a copy: for a "value type", a value is copied, and for a "reference type"/"pointer type", a reference/pointer is copied. Binding never causes any copies to be made: a bound identifier is just a name for a value.
4. Re-assignment to an existing variable changes the value that is linked to that variable name. Re-binding a name does not affect any existing binding. Instead, the existing name is ✪**shadow**ed by the new name. If the new name is ever removed from consideration—for example, if it passes out of scope—then the older name is no longer shadowed, and it becomes accessible again.

Different patterns will be introduced throughout this textbook, and patterns can be nested arbitrarily. Pattern-matching is a much more powerful and general mechanism than assignment, and is used very extensively throughout F#; you must develop a good understanding of it. One way of using patterns to bind values in our expressions is by using a ✪**let binding**.

**Syntax: `let` binding**

```
let x = ↓expr₀ in
  expr₁
```

This syntax evaluates $expr_0$ and pattern-matches the resulting value against the pattern `x`. Any identifiers that are bound to values during the pattern-match can be used in $expr_1$. Identifiers must be defined before they are used in expressions. Bound identifiers only

[1] Other basic types such as `byte` exist, but won't be used here.

begin to exist after the `in` part of the `let` binding[2]. `let` bindings can be nested to any depth.

```
let alex = "Lion" in
  alex + " king"
```

The value of this expression is "Lion king". The bound value is limited in scope to the expression that it is used in.

We typically use a shorter syntax to mean almost the same thing as shown in the previous syntax box:

**SYNTAX: SIMPLER `let` BINDING**

```
let x =
  expr0
expr1
```

Now that we have a way to see some code that can be used for pattern-matching, it's time to turn our attention to patterns themselves. For now, there are four basic patterns that you should know about.

**SYNTAX: IDENTIFIER PATTERN**

```
x
```

An ✪**identifier pattern**[3] matches any value and binds it to the supplied identifier. It always succeeds. The identifier must be a valid F# identifier.

```
let test =
  "Hi there!"
let ``this is fun...`` =
  732
```

We typically use an identifier pattern to extract a particular part of a value — such as a single element of a list — from a larger value, so that we can use that part in a future calculation.

```
let w = 5 in
  let x = 7 in
    let y = w - x in
      let w = x in
        w + y
```

What happens when this is evaluated?

1. The value `5` matches the pattern and is bound to the identifier `w`.
2. `7` matches the pattern and is bound to `x`.
3. The expression `w - x` is evaluated, resulting in the value `-2`.
4. `-2` matches the pattern and is bound to `y`.
5. The expression `x` is evaluated, resulting in the value `7`.
6. `7` matches the pattern and is bound to `w`. *The binding in step (1)* ***continues to exist*** *and is unchanged!*
7. The expression `w + y` is evaluated, resulting in the value `5`.

**A DETAIL OF SHADOWING**

In F#, an identifier at the top level (i.e. the outermost scope) of a file or module cannot be shadowed at that level of scoping. You will probably never run into this exception to the rule in a real-world functional program.

Note that, in some functional languages, there is no obligation to execute an expression if the language can prove that the result of an expression will not affect the semantics of the program.

```
let w =
  5
let x =
  7
let y =
  w - x
let w =
  x
w + y
```

This example is semantically equivalent to the previous example, but written using a shorter syntax.

**F# INTERACTIVE**

The above example doesn't work if you type it into F# Interactive in one go. However, it does work if you type it into F# Interactive and end each expression with `;;`. This occurs because F# Interactive evaluates "blocks" of code in a single top-level environment, where duplicate values are not allowed. The code itself is valid and correct.

**SYNTAX: WILDCARD PATTERN**

```
_
```

The ✪**wildcard pattern** is a single underscore. It matches any value; it does not bind any identifier; and it always succeeds.

[2] Later on, you will learn about the `rec` keyword which lifts this restriction.

[3] Sometimes called a ✪**variable pattern**

```
let _ =
  43.2
```
This pattern-match will succeed. No identifiers will be bound.

We typically use a wildcard pattern to match and ignore parts of a value that we don't want to use later on. For example, we can use an identifier pattern together with a wildcard pattern to bind the first element of a list to a name, and ignore the rest of the list.

**SYNTAX: CONSTANT PATTERN**

Any literal value

A constant pattern is a literal value such as a numeric, string, boolean, or character literal. `5`, `"Kind sir"`, `false`, `45.109`, and `'p'` are all constant patterns. A constant pattern will only succeed if the value exactly matches the specified literal. The type of the value must be the same as that specified by the constant pattern.

```
let 'k' =
  'k'
let 10 =
  1 + 2 + 3 + 4
let "batman" =
  "joker"
```

The first two of these pattern-matches will succeed. The last pattern-match will compile, but will fail at runtime if it ever needs to be executed. Execution would result in a `MatchFailureException`, which is what F# uses to indicate that a pattern-match has failed.

Since the type of the value must be the same as the type specified by the constant pattern, code such as `let 7 = 'p'` is invalid.

We typically use a constant pattern when we want to answer very specific questions and make decisions[4]. For example,

- We can answer "is the list empty?" by checking whether the list matches a constant pattern for an empty list
- We can answer "is the return value -1?" by trying to pattern-match against the `-1` pattern
- We can answer "did the operation succeed?" by matching against the constants `true` and `false`

[4]You'll have to wait a while to see how we use pattern-matching to make decisions, and a discussion of lists will occur even later.

**PATTERNS ≠ EXPRESSIONS**

A pattern is *not* an expression, and an expression is *not* a pattern. They can look the same, but they mean very different things. For example:

```
let x =
  "ndi"
let y =
  x + "funda"
```

On the first line of this code, `x` is used as a pattern. This means that it will be pattern-matched and, if the pattern-match succeeds, it will be bound to a value. On the second line of this code, `x` is used as an expression. This means that it will be evaluated and will generate a value. If you do not understand the difference, you will end up writing incorrect and nonsensical code like `let x + "funda" = "ndifunda"`, and you won't understand why it doesn't compile!

Read the Syntax sections very carefully and note where a *pattern* is allowed and where an *expression* is allowed. This will allow you to interpret and write F# code correctly.

**SYNTAX: OR PATTERN**

$\underline{\texttt{pat}_0}$ | $\underline{\texttt{pat}_1}$ | $\cdots$ | $\underline{\texttt{pat}_n}$

An or-pattern matches if *any* of the patterns, separated by `|`, match. Or-patterns are sometimes enclosed by brackets to separate them from surrounding syntax. All of the patterns separated by `|` must bind to the same set of variables.

```
let 3 | 4 = 3
let 5 | 6 | 7 = 1 + 2 + 3
let m = "luthor"
let "joker" | "riddler" = m
let p | r = 23
let a | 9 = 9
```
The first two of these pattern-matches will succeed. The third pattern-match is an identifier pattern and will succeed. The fourth pattern-match will compile, but will fail at runtime if it ever needs to be executed. The fifth and sixth patterns will not compile because the patterns separated by `|` do not contain the same set of identifiers.

We typically use an or-pattern when different values should be treated in the same way.

# Exercises

1. List three differences between *pattern-matching / binding* and *assignment*.

2. In C#, I would write `a==b` to test for equality between `a` and `b`. What would I write in F#?

3. Rewrite the following code so that it does not use the `in` keyword. For example,

   ```
   let test = 11 in
     test-2
   ```

   could be rewritten as

   ```
   let test =
     11
   test-2
   ```

   Preserve all the bindings in your answers, even if they are not used.

   (a)
   ```
   let a = "hi" in
     "zoot"
   ```

   (b)
   ```
   let b = "zoo" in
     "zooty" + b
   ```

   (c)
   ```
   let c = 7.5 in
     let d = 34 in
       c + 0.2
   ```

   (d)
   ```
   let e = 8 in
     let f = e in
       let e = 20 in
         let g = 3 in
           e+f+g
   ```

4. What is the difference between *pattern-matching* and *binding*?

5. Which of the following are valid F# identifiers? Give a reason for each answer.

   (a) `_`
   (b) `a`
   (c) `a'ight`
   (d) `"amp and such"`
   (e) ``` ``amp and such`` ```
   (f) `@called`
   (g) `let`
   (h) ``` `` _`` ```

6. Explain the difference between *shadowing* and *updating* (or *mutating*) a symbol.

7. A student writes this code:

   ```
   let r =
     System.Random ()
   let v1 =
     r.Next ()
   let v2 =
     v1
   ```

   The student is surprised to see that `v1` is always the same as `v2`. Shouldn't they be different values from the random number generator? Explain why the values are the same.

8. Examine the following patterns and write down C, W, I, or O if the pattern is a constant, wildcard, identifier/variable, or or-pattern respectively.

   (a) `213`
   (b) `q`
   (c) `q|q`
   (d) `'q'`
   (e) `""`
   (f) `_`
   (g) `false`
   (h) `_ | 5`
   (i) `practically`
   (j) `not_really`
   (k) `"Not | really"`
   (l) `_true`
   (m) `3|2`
   (n) `43.50`

9. Which of the patterns below are valid patterns? If a pattern is valid, write a ✓ and identify the kind of pattern that it is. If a pattern is invalid, write a ✗ and explain what is wrong with it.

   (a) `'c'|'k'`
   (b) `1.0|2.0`
   (c) `1.0|2.0|3`
   (d) `m|p`
   (e) `kappa-delta`
   (f) `O'neill`
   (g) `let`
   (h) `tweet'`
   (i) `_|v`
   (j) ``` ``aye 'b' |C|_`` ```
   (k) `__`

# Functions

A function is anything which accepts exactly one input value and, when this is supplied, evaluates an expression to generate exactly one output value. The ✪**definition** and ✪**application** of a function are separate. Code within a function is only executed when the function is applied.[1]

**Syntax: function definition**

```
fun x -> ↓expr
```

It is extremely important to understand that a function is just a value, and function definition is how we create a function value.

The `x` is a pattern. The `expr` is the function body; it necessarily generates a value when an input is provided, which is the result (or "return value") of the function. There is no "return statement" in F#; the last value to be generated by the function *is* the returned value.

A definition should always be read as `fun x -> (expr)`: imagine that there are implicit brackets that group the `expr` part together and separate it from the `x`. This is always the case, even if the actual expression is many lines long, or a single continuous long line of logic.

When a function is not bound to a name, we call it a ✪**lambda function**.

```
let qwan =
  fun k -> k * 4
```

In this example we have bound a function to a name.

**Why name anything?**

When we name something in our code, it is because we want to refer to it later on, or because we want to give that "block" of functionality a label that makes it easier for humans to think about. If a function is long and complex, or if it is used often, then it makes sense to give it a name. But if the function is small and easy—such as a function which just multiplies things by 4, or a function which joins two strings—then it doesn't make sense to give it a name. You might name a recipe, but you don't name every particle of flour that you use in it! As a result, you will often find small, simple lambda functions scattered throughout functional code.

We say that we ✪**invoke**, ✪**call**, ✪**evaluate**, or ✪**apply** a function; these terms are largely synonymous and all of them mean "provide an input to the function, thereby obtaining an output". A function's body is not executed until the function is applied.

A function's body is evaluated in the context of the code that the function is *defined* in (i.e. in its ✪**lexical scope**), **not** the scope that it is *applied* in. This fact is very important for understanding some of the techniques, such as closures, which are discussed later on.

**Syntax: function application**

```
function input
```

We apply a function by giving it its input, separated by a space. Function application causes the input to be bound to the input-pattern of the function before the body of the function is executed and a value is consequently generated.

```
(fun p -> p + 3) 7
```

Here we define a function and then immediately apply it; note that the brackets exist only to show where the function begins and ends. During evaluation of this function, `p` is bound to `7`. Due to referential transparency, `p` can immediately be replaced at all places by `7`. This simplifies the expression to `7+3`, which is evaluated and generates the value 10.

It is critical to understand that the function we have defined is *unnamed*. The `p` identifies the name of the function *input*, and not the name of the function. There is no way to refer to this function by a name, because it has not been bound to any name.

The technique of substituting a function input with a value, resulting in a simpler expression, is called $\beta$-reduction (pronounced "beta-reduction"). In fact, function application in a functional language is just repeated $\beta$-reduction!

**The Value of Functions**

Always remember that identifiers are bound to *values*, not *expressions*. However, you have also been told that functions, which evaluate an expression to generate a value, *are* values. How does this all make sense?

Keep this in mind: when we *define* a function, we are creating a value. This created value can be bound to an identifier. The code within the function is not executed during this process.

It is only later on, when we *apply* that function, that the code within the function will be executed.

[1] This is exactly the same definition of a function that you saw on page 5.

This "trick" lets us use functions as values *and* apply them to generate values. The difference between definition and application is very important, so be sure that you understand it!

Function application is always *greedy*[2]. A function will always grab the first input that it's given, use it, and replace itself with the resulting $\beta$-reduced value. The `.` operator, which you will read about on page 65, is one of the very few parts of the language that is "greedier" (i.e. has higher precedence) than function application.

```
let rex =
  fun ny -> ny + 3
rex 7
```

This is exactly equivalent to the previous example; if you use referential transparency to replace `rex` with its value, you will obtain the code of the previous example. Having bound the function to a name, we apply it and the value 10 is generated. It is unambiguously clear where the function begins and ends, and brackets are therefore not needed here.

**KEEP IT SIMPLE!**

Can you write a F# function which takes an input and subtracts 10 from it?

YES — you can! Following the instructions and examples that you've read so far, you will not only be able to create the function, you will also be able to apply it. Think about what that function will look like. After you have it in your head, read the next line.

```
fun v -> v - 10
```

Did your function look like that? Did you actually try to write it? If so, congratulations: many students don't. After years of programming in an imperative language, most students hesitate to try writing their own functions. This is because all of the languages that they've used up to this point are complicated: they require you to keep state in your head, they have different "kinds" of methods/functions, the values are split into value-types and reference-types and addresses, some things are aliased, and so on. In a language like C#, similar code would be longer and more complicated:

```
int SomeName(int k) {
  return k - 10;
}
```

[2]Another way of phrasing it is that function application has a very high precedence. For a full precedence table, go to https://goo.gl/pFdKNs.

Programmers coming from an imperative background usually think "there must be more to write!", because things are complicated when you program imperatively. But functional programming is not like that. There is only one kind of function; there are no "aliases" or different kinds of values; there's no necessity to put in semicolons or braces or a `return` statement or explicit type-names or names for functions; and identifiers are referentially transparent. Even shadowing, which seems like an alien and weird concept, is actually easier than the imperative way: just look upwards in the program code until you find the name, and you're done. When programming in a functional language, *keep it simple*, and you'll be surprised at how often it just works!

Binding a name to a function is such a common thing that there is a convenient syntax for it.

**SYNTAX: FUNCTION DEFINITION & BINDING**

```
let ident pat =
  expr
```

```
let increase n =
  n+16
let t =
  increase 8 / 4
```

The `increase` function is applied before the division, so the identifier `t` is bound to the value 6. To evaluate the `8 / 4` before applying the function, one would have to write `increase (8 / 4)`.

## USING *unit*

Remember that a function *must* accept a single input, and when it is given that input, it *must* generate a single output. If it doesn't do that, it isn't a function ... and if it *does* do that, then it is a function, even if it might not look like a function! There's no such thing as a "void method" in functional programming: *all functions return a value*.

However, when a function is called only for its side-effect, we want to return a value that cannot possibly have an effect on the program's working. If we return an `int` or a `bool` or any other type, an evil programmer may decide to use that value to affect the output of their function, and we would thus be making it easy to introduce even more side-effects into our program! We want to avoid that, so we return `()`: a `unit` value which gives no indication of success or failure, and which cannot be used to change control flow.

A second reason to use `unit` is as a function input. Sometimes we want to define a function, but delay

its application. The function does not need any additional input to generate its result, but we must nevertheless give it an input to apply it. The only sensible input value to give is `()`, since this is the only type of input which the function can depend on without causing the control flow of the function to be affected in any way.

## Curried functions

A function cannot accept more than one input. However, functions are first-class, and function values can therefore be returned from a function. We can use these facts to simulate $n$-input functions, using a technique called ✪**currying**[3].

**Syntax: currying**

```
fun i₀ ->
  fun i₁ ->
    ··· ->
      fun iₙ ->
        expr
```

A curried function simulates an $n$-input function by returning additional functions.

```
(fun z ->
  fun e ->
    e*z+1) 9 5
```

Here we have a "2-input" function which is actually two 1-input functions. We seem to be providing our "2-input" function with two inputs, though what we are really doing is providing one input, in turn, to each of the functions. Remember that this should be read as:

```
(fun z ->
  (fun e ->
    (e*z+1))) 9 5
```

and that function application is greedy – so the `9` will be grabbed immediately. A single $\beta$-reduction results in `(fun e -> (e*9+1)) 5`. The next $\beta$-reduction results in `5*9+1`, which evaluates to 46.

Currying is such a common technique that there is a convenient syntax for it.

[3]The technique is named after the mathematician Haskell Curry, not after the delicious food. Interestingly, Curry rediscovered the technique; it was initially discovered by Moses Schönfinkel, and possibly earlier by Friedrich Frege. We will nevertheless continue to call it "currying", not "schönfinkeling" or "fregeing".

**Syntax: convenient currying**

```
fun i₀ i₁ ··· iₙ -> ↓expr
```

```
(fun z e -> e*z+1) 9 5
```

This is exactly equivalent to the previous example; it's just written in a shorter way.

In fact, binding a name to a curried function is so common that there is a convenient syntax for that, too!

**Syntax: binding & currying**

```
let ident i₁ ··· iₙ =
  expr
```

```
let em i nem =
  i/nem
```

The above line of code is exactly equivalent to these lines:

- ```
  let em =
    fun i nem -> i/nem
  ```
- ```
  let em =
    fun i ->
      fun nem ->
        i/nem
  ```

In all of these, the identifier `em` is bound to a curried "2-input" function.

**Syntactic sugar**

The "binding & currying" and "convenient currying" syntaxes are ✪**syntactic sugar**: a way of writing exactly the same thing using a more convenient (and usually shorter) syntax. You may have already encountered other examples of syntactic sugar in your Computer Science journey; for example, many languages allow you to say `count += 7`, which is syntactic sugar for `count = count + 7`. They both mean exactly the same thing, but one way of writing it is shorter. If you don't want to use the syntactic sugar, or if you want to spend more time writing out the full form so that you understand currying a bit better, *don't use the sugared syntax*! Stick to the basic forms. They mean exactly the same thing.

If you are struggling to see what some sugared code does, *de-sugar* it by getting some paper out and translating it into a more expanded form. It will only take 5-10 seconds, and it will often help you to understand exactly what's going on. Staring at the screen or page for 40 seconds and trying to do the

same thing mentally will just waste your time! As you get more familiar with the syntax, you'll need to do this less and less often, and with enough practice you'll be able to switch between the syntaxes effortlessly.

## Function types

All functions are values, and all values have types. Therefore, all functions have a type. As you know, a function is anything that takes an input and, when this input is provided, evaluates code to generate an output. The type of a function can therefore be specified entirely in terms of the type of the input and the type of the output. We use the following syntax to write a function type:

**Syntax: function type**

$t_{in}$ `->` $t_{out}$

$t_{in}$ is the type of the input, and $t_{out}$ is the type of the output. Remember that a function can only take one input and generate one output. If it appears to take more than one input, then it is actually a *curried* function: a function which returns a function value. This means that the type of a curried function such as $t_0$ `->` $t_1$ `->` $t_2$ should be understood as $t_0$ `->` `(`$t_1$ `->` $t_2$`)`. For simplicity and convenience, we often write a curried function type without the brackets.

## Higher-order functions

A ✪**higher-order function** accepts or returns a function. Higher-order functions are not "special"; they exist as a consequence of the decision to make functions first-class. The flexibility and expressive power of higher-order functions is remarkable: they allow us to build functions which express a general algorithm, and pass in the specific functionality on a case-by-case basis.

```
let twice f v =
  f (f v)
let add4 x =
  x + 4
twice add4 10
```

In this toy example, the function `add4` is applied twice to the value 10. The `twice` function is a higher-order function.

## Closures

Recall from page 17 that a function is evaluated in the context of the code that it is *defined* in, not the scope that it is *applied* in. If we have a way to delay function execution until later, then we can have the function "carry" its defining environment with it as it goes.

The usual way to delay function execution is by pattern-matching on a `unit` input.

```
let subtractor a b =
  fun () -> a - b
⋮
let to_execute = subtractor 17 5
⋮
let v = to_execute ()
```

The lambda function is defined, bound to a name (`to_execute`), and evaluated at different places in the program. The lambda function "carries" the context of its definition with it.

A function which uses identifiers in its defining scope *in addition to* its input is called a ✪**closure**; we say that it "closes over" or "captures" such identifiers.

**What kind of function is that?**

There is a lot of function-related terminology that has been introduced so far. Let's summarise:

- If a function is unnamed, it is a ✪**lambda** function.
- If a function appears to take multiple inputs, but really returns functions that accept subsequent inputs, it is ✪**curried**.
- If a function accepts or returns a function, it is ✪**higher-order**.
- If a function refers to a value from its defining scope during its application, then it is a ✪**closure**.

This can be very confusing if you try to think about what, exactly, a particular function is. Is it a lambda function? Or a closure? Or . . . ?

The confusion goes away when you realize that all of these things are *descriptions*, and not distinct categorizations. So if you have an unnamed function that refers to a value from its defining scope, and which takes a function as input, then it is a "higher-order lambda function closure"—and there is nothing wrong with that. All functions in a functional language are first-class, so we usually don't even bother to use the description "first-class". But if we wanted to, we could say that our example function is a "first-class higher-order lambda function closure", and still be correct.

It also helps to realize that some of these descriptions are more specialized varieties of other descriptions. For example, curried functions must return other functions, so all curried functions are higher-order functions.

# Operators

It often happens that we wish to apply one function, then use another function on the result, then use another function on that result, and so on. If we have initial data `d` and functions `f`, `g`, and `h` to be applied in order, we might achieve this as follows:

```
h (g (f d))
```

Not only is this annoying to type, but it makes the order of operations more difficult to see. The first function to be applied is `f`, but this is the most deeply-nested function — and, therefore, the last to be read during a left-to-right scan of the code. Good use of built-in operators, most of which are available in most functional languages, can make code much more readable and understandable.

## Pipe operator

The ✪**pipe** operator `|>`, sometimes called ✪**pipe-to**, accepts a value on the left and a function on the right, and applies the function to the value. Instead of `h (g (f d))`, this allows us to write `d |> f |> g |> h`—and, written like this, we can clearly see the execution of each function in-order from left to right. The pipe operator allows us to build a "pipeline" of functions, each one building on the previous one, that takes in data on the left and spits out a result on the right. When reading the code, we vocalise this as "`d` pipe-to `f` pipe-to `g` ...".

## Compose operator

The ✪**composition** operator `>>`, sometimes called ✪**compose**, "glues together" two functions to form a new function[4]. Instead of `h (g (f d))`, this allows us to write

```
let z =
1
f >> g >> h z d
```

For any two functions `a` and `b` which you want to compose as `a >> b`, the output type of `a` must be usable as the input type of `b`. The composed function's input type will be `a`'s input type, and its output type will be `b`'s output type.

[4] If you know a bit of mathematics, you might recognize this as the *composition* of two functions—hence the name of the operator.

### "Composition sounds weird!"

While you might not use composition as much in your earlier functional programming attempts, it becomes critically important for understanding and using the vast majority of advanced functional programming techniques—including "monadic" techniques that you may discover online. Composition allows a programmer to split functionality into small and simple sections, while having the ability to join these sections together to form a much more powerful and complex entity. It can also help you to reuse functions and develop exceptionally elegant and concise code, so search for opportunities to use composition.

## Other operators

Other interesting operators are:

- Backwards-pipe `<|`:

  ```
  let (<|) f x = f x
  ```

  This is mostly used to avoid using brackets. For example, `g (f d)` could be written as `g <| f d` instead.

- Backwards-compose `<<`:

  ```
  let (<<) g f = fun x -> g (f x)
  ```

### Operator madness

Consider the expression `a / b`, and think about what the operator, `/`, does. It takes two inputs—one on each side—and when it is evaluated, it uses them to generate a result. Taking inputs and using them to generate a result is not so different from what a function does... and, in fact, operators in functional languages are often simply functions in disguise! For convenience, the language allows you to use the operator *after* its first argument; this is called ✪**infix notation**. If you wanted to use it in the ✪**prefix notation**, i.e. *before* its arguments, you could do that by enclosing it in round brackets. For example, `23 * 45` can be equivalently written as `(*) 23 45`!

You might want to use prefix notation for an operator if you want to only specify one of the operands. If you use infix notation, you have to specify both.

# Exercises

1. Briefly define *β-reduction*.
2. *β*-reduce the following expressions:

   (a)
   ```
   (fun a -> a + " killer") "hunter2"
   ```

   (b)
   ```
   (fun k -> fun r -> r > k) 16
   ```

   (c)
   ```
   (fun k ->
     let p =
       k-5
     let r =
       fun t -> p/t
     r
   ) 25 10
   ```

   (d)
   ```
   let n k p =
     (k * 3) + p
   let t s =
     n 2 s
   let r v =
     fun () -> v + (t 3 - t 1)
   r 10
   ```

   (e)
   ```
   let f x =
     x 3.0
   let g k =
     k + f (fun q -> q - 1.5)
   let t =
     f (fun p -> p + g p)
   ```

   (f)
   ```
   let k =
     15
   let z =
     fun p -> p - 10
   let b =
     fun r -> z (r + 2)
   let t =
     b k
   ```

   (g)
   ```
   let k p z =
     z p
   let t =
     k 10 (fun r -> r+9)
   ```

   (h)
   ```
   let f cat b =
     b ("see " + cat)
   let t h =
     f h (fun k -> "I " + k)
   ```

   (i)
   ```
   let f cat b =
     b ("see " + cat)
   let t h =
     f "dog" h
   ```

3. Write down the type of the following functions:

   (a) `fun p -> "quiz" + p`

   (b) `fun p -> p + 600`

   (c) `fun p -> fun r -> r-p`

   (d) `fun p r -> p-r`

   (e) `fun 8 -> 25.13`

   (f) `fun status -> fun () -> status/2`

   (g) `fun i s -> s () - 10 * i`

4. `let zoq fot pik = fot/pik` can be rewritten in two alternative and semantically-equivalent forms, *without* altering the body of the function at all:

   - ```
     let zoq =
       fun fot pik -> fot/pik
     ```
   - ```
     let zoq =
       fun fot ->
         fun pik ->
           fot/pik
     ```

   Rewrite `let k = fun m -> fun t -> t*3` in two different forms.

5. Define the following terms:

   (a) Prefix notation

   (b) Currying

   (c) Lambda function

6. Write down the word or phrase that means...

   (a) "A function which can reference variables that exist within the lexical scope of its definition"

   (b) "A function that takes a function as input"

   (c) "Single-input, single-output functions which are chained to emulate the multiple-input functions which exist in other languages"

7. (a) Write a function

   ```
   log: string -> (unit -> 'a) -> 'a
   ```

   which prints the input string before the function is executed. To print a string with identifier `myStr` to the output, you can use the built-in function `printfn` as follows:

   ```
   printfn $"{myStr}"
   ```

   (b) Here is an existing line of code:

   ```
   fly 300 12.4
   ```

   Modify this line of code so that it uses your function to log the message "Flying now" before executing the code.

   (c) Explain why the modification that you made cannot possibly change the change the meaning of the program.

8. Describe the function using one or more of the following terms, in any order: "lambda", "curried", "higher-order", "closure".

   (a) `let a b c = b + c`

(b) `let a = fun b -> fun c -> b + c`

(c) `let a = fun b -> b`

(d) `fun b -> b true`

(e) `fun _ -> 0`

(f) `fun _ -> fun k -> k/2`

(g) `fun r -> fun k -> k/9`

(h) `fun r -> fun k -> k/r`

(i) `let z () _ = ()`

(j) `let arf m = m () 0.5`

(k) `let a = 3`
1let v x = a * x

9. Write down the name of the following functions.
If a function has no name, write "unnamed".

(a) `let a b c = b + c`

(b) `fun r a -> a + 5`

(c) `fun w -> let a = 5 in w + a`

(d) `let v = fun r -> ()`

(e) `fun h -> fun r -> ()`

# TUTORIAL

In this tutorial, we will use functions to create a simple unit converter. Most of the world uses metric (or "SI") units of measurement such as kilograms, meters, and degrees Celsius. However, there are still a few places that use "US customary" units such as pounds, feet, and degrees Fahrenheit. Our converter should be able to convert between customary units and SI units.

You might be unfamiliar with US customary units, so let's go over what you'll need to know:

- 1 *inch* is 25.4 millimeters. There are 12 inches in a *foot*. There are 3 feet in a *yard*. There are 1760 yards in a *mile*.
- 1 *pound* is 453.59237 grams. 1 *ounce* is $\frac{1}{16}$ of a pound. There are 2000 pounds in a *short ton*.

Amusingly, one of the reasons that people in the United States refuse to use the SI system is because they find the customary units to be "easier"!

1. Create a new F# console application called **UnitConverter**, and open it up in Visual Studio Code.

2. In the code file (which ends with `.fs`), above the `main` method, type:

   ```
   let inchToMillimeter v =
     v * 25.4
   ```

   Select these lines and run them through F# Interactive (using Alt + Enter), to make sure that F# Interactive integration is working. You should see text saying `val inchToMillimeter : v:float -> float`.

   In the F# Interactive window, you should be able to write the expression `inchToMillimeter 7.5`—remember to end the expression with a `;;`—and get the result of `190.5`. Make sure that you can do this.

3. Below the `inchToMillimeter` line, let's write a function to convert from feet to inches. There are 12 inches in a foot, so

   ```
   let footToInch x =
     x * 12.0
   ```

   Select *all* the lines you have typed and send them down to F# Interactive. Test out `footToInch`.

4. Let's put in some other length conversions.

   ```
   let yardToFoot v =
     v * 3.0
   let mileToYard mile =
     mile * 1760.0
   ```

   Send them down to F# Interactive. We can now convert from feet to millimeters by evaluating this expression in F# Interactive:

   ```
   footToInch 6.0 |> inchToMillimeter
   ```

   Note what the `|>` does: it takes the *output* of the expression on the left (i.e. `footToInch 6`), and uses it as the *input* of the function on the right `inchToMillimeter`. That gives us a figure of `1828.8` – so we now know that someone who is "six feet tall" is about 1.8 meters tall.

5. We might want the output to be meters rather than millimeters, though. How do we convert millimeters to meters? We divide by 1000, of course — or, equivalently, we can multiply by 0.001. Let's make a function to multiply an input by 0.001.

   ```
   let divBy1000 =
     (*) 0.001
   ```

   Remember that the multiplication operator `(*)` uses *currying* to accept two inputs, and (when used with `float` values) has the type ***`float -> float -> float`***. Here, we have given it a single input. It accepts that input and gives us the output, which is a ***`float -> float`***.

   Send this one down to F# Interactive and see what it looks like. You should see it as a `float -> float` function.

6. Let's use that `divBy1000` function to convert millimeters to meters, and meters to kilometers. Write the following functions after `divBy1000`:

   ```
   let millimeterToMeter n =
     divBy1000 n
   let meterToKilometer =
     millimeterToMeter
   let millimeterToKilometer =
     divBy1000 >> divBy1000
   ```

   The first function here is quite easy to understand. The second and third might be a bit more confusing, so let's talk about them.

   To convert a meter to a kilometer, you just divide by 1000. That's exactly what happens in `millimeterToMeter` — and since a name is exactly equivalent to the value bound to it, we can simply bind `meterToKilometer` directly to a function which performs the needed task.

   In the third function, remember that the `>>` operator will *compose* two functions: it "glues" the functions together to make a more complicated function which takes an input acceptable to the first function, and returns an output specified by the second function. Our `millimeterToKilometer` will therefore take in a `float` input, as desired by the first function `divBy1000`, and return a `float` output, as generated by the second function `divBy1000`.

Send these down to F# Interactive and play around with them. Do they work as expected?

7. We can convert miles to kilometers as well! Just write

```
1024.0 |>  mileToYard |> yardToFoot
|> footToInch |> inchToMillimeter
|> millimeterToKilometer
```

This requires a lot of typing, though. So let's some convenient functions to do some useful conversions for us. Make sure you put these functions below the other functions you've written.

```
let inchToMeter =
  inchToMillimeter >> millimeterToMeter
let footToMeter =
  footToInch >> inchToMeter
let yardToMeter =
  yardToFoot >> footToMeter
let yardToKilometer =
  yardToMeter >> meterToKilometer
let mileToKilometer =
  mileToYard >> yardToKilometer
```

Send it all down to F# Interactive and you'll be able to evaluate `mileToKilometer 1024.0` there and discover that it's `1647.968256` kilometers.

It's important to notice something at this point: for the last few steps, we haven't actually done anything other than compose together existing functions. We last did any "real" conversion work in step 4. Once we were able to do the conversions between basic customary units, everything that we've done has just been gluing together functions to make larger functions—and it all works! In fact, if you want to change the system to use UK imperial units instead of US customary units, all you'd need to do is change those basic functions, and functions like `mileToKilometer` would require no changes whatsoever. Now *that's* a great way to reuse code!

## Extensions

1. How many ways can you find to write each of the functions, such that the function type remains the same?

2. You have enough information to do mass conversions using pound, ounce, and short ton. Add in functions to do these conversions.

3. What about converting the other way, from metric to customary units? Can you write functions to do those?

# Match expressions

A ✪**match expression** is used to select between alternatives[1].

**Syntax: match expression**

```
match expr with
| pat_0 ->
  expr_0
| pat_1 ->
  expr_1
⋮
| pat_n ->
  expr_n
```

The value generated by `expr` is pattern-matched against each of the patterns $pat_0$ through $pat_1$, starting from the topmost pattern and continuing on towards the last pattern. If no pattern matches the value, then a `MatchFailureException` is thrown. Each pattern must be consistent with the other patterns: for example, one pattern cannot assume that the value is a string, while another assumes that it is an integer or a unit.

Let us assume that the first pattern to match is $pat_m$. The expression $expr_m$ is then evaluated, and the result of this expression is the result of the match expression. Any identifiers which were bound in $pat_m$ may be used within the scope of $expr_m$.

Each branch of the match expression must result in the same type of value. For example, if $expr_0$ evaluates to an integer, then $expr_{1..n}$ must also evaluate to integers.

```
match 5+8 with
| 4 ->
  "hi"
| v ->
  "yes"
| 13 ->
  "zip"
| _ ->
  "roy"
```

The expression `5+8` is evaluated, resulting in the value 13. Then the constant pattern `4` is checked against this value. The match fails. The identifier pattern `v` is checked. It succeeds and binds `v` to the value `13`. The corresponding expression is evaluated and the result of the match expression is `"yes"`. The remaining patterns in the match expression are skipped. Note that every branch of the match expression will evaluate to a string, and every pattern is consistent with the assumption that the value to be matched is an integer.

[1] F# supports an `if` expression and `when` guards, but we won't be using these because new functional programmers tend to rely on them instead of learning about patterns, matching, and binding. This cripples their understanding of functional code, so to avoid that, we will avoid using them in the first place.

```
let maxValue a b =
  match a > b with
  | true ->
    a
  | false ->
    b
```

This code finds the maximum of two values, and uses testing similar to what an if-then statement might use. The trick that it uses is simple: evaluate the boolean expression and match the result using the constant patterns `true` and `false`. You can use similar tricks throughout your code to do if-then tests, but be warned: some things in F# *cannot* easily be phrased as if-then tests, so you *must* learn how to use pattern-matching *in general*!

**Match expressions ≠ switch statements!**

A match expression checks whether the value of an evaluated expression *matches a pattern*. A switch statement checks whether a value *is equal to another value*. These are not the same thing (and, in fact, the F# language has no "switch statement"). First-time functional programmers tend to write code like this:

```
let a =
  25
let b =
  34
let c =
  match a with
  | b ->
    "Same!"
  | _ ->
    "Different!"
```

The programmer is trying to compare the values of `a` and `b`, but that's not what this code does. The bold line above attempts to match the value 25 to the identifier pattern `b`. This pattern-match succeeds, `b` is bound to 25 (which shadows the earlier binding of `b`), the expression `"Same!"` is evaluated, and the resulting value is the value of the match expression.

# Exercises

1. A friend runs the following code and is convinced that F# is executing code incorrectly.

```
let x =
  5
let y =
  8
let z =
  match x with
  | y ->
    y+1
  | x ->
    12
```

   (a) The value of `z` is `6`. Explain why.

   (b) In the match expression, `x` could be replaced by `_`. Explain why a good programmer would do this.

   (c) Rewrite the match expression to return `y+1` if `x` is equal to `y`, and to return `12` otherwise.

2. Consider the following code:

```
let "hazmat" =
  match 901 with
  | 201 ->
    "wut"
  | kray ->
    "lol"
  | 901 ->
    "hazmat"
```

   (a) Circle every pattern-match that fails.

   (b) Underline every pattern-match that is not executed.

   (c) Write down the result of the match expression.

3. Consider the following code:

```
let t =
  0.8
let r =
  match t/2.0 with
  | 0.0 ->
    6.5
  | t ->
    t
let v =
  t + r
```

   (a) What is the value of `v`?

   (b) What is the value of `r`?

   (c) Will this code compile if you wrote `t/2` instead of `t/2.0`? Why or why not?

   (d) Will this code compile if you wrote `0` as your constant pattern instead of `0.0`? Why or why not?

4. Consider the following code:

```
let roll =
  4
match roll with
| 2 | 4 | 6 ->
  "evens"
| 1 | 3 | 5 ->
  "odds"
```

   (a) Write down the result of the match expression.

   (b) Which numbers could you bind `roll` to if you wanted to obtain the other result?

   (c) What happens if someone binds `roll` to `24`?

5. Consider the following function.

```
(fun b ->
  match b%2 with
  | 0 ->
    "Steven"
  | 1 ->
    "Codd"
) 61
```

   (a) $\beta$-reduce the function.

   (b) What is the type of the function?

   (c) What is the name of the function?

# Tutorial

In this tutorial, we will create a program that takes in a number and prints out that number in words. For example, if given the number 7716, it should print out "seven thousand seven hundred and sixteen". The input number must be in the range [-9999, 9999]. We will deal with each "place" in the number – thousands, hundreds, tens, and units – separately.

Some "trivial" code has been left out in this tutorial, and you are expected to fill it in. Each step ends with code that is complete and executable. As you go through this tutorial, you are encouraged to send code down to F# Interactive to play with it. You will also be introduced to a small number of other functions in this tutorial; when an unfamiliar function occurs, we will briefly describe it. All of the functions used here are given fuller descriptions on page 59.

1. Just as you have done previously, start a console application. Once again, your code will be written above the `main` function.

2. We will begin with a small function to parse numbers in the range [1,9].

```
let parseUnits = function
  | 0 ->
    ""
  | 1 ->
    "one"
  | 2 ->
    "two"
  ⋮
  | 9 ->
    "nine"
  | _ ->
    failwith "I expected a single digit."
```

   Notice that we are using the shorthand syntax for a function that accepts an input and immediately pattern-matches it using a match expression. The `failwith` function just throws an exception, typically crashing the program as it does so.

3. Now let's write a function to parse numbers in the range [1,99]. In English, this annoying because all the numbers in the range [10..19] have special names. However, we can use the already-defined `parseUnits` function to help us. We cannot use any identifier that still needs to be defined, so we will need to place our `parseTens` function below the `parseUnits` function.

```
let parseTens n =
  let tens =
    (n % 100) / 10
  let units =
    n % 10
  match tens with
  | 0 ->
    parseUnits units
  | 1 ->
    match units with
    | 0 ->
      "ten"
    ⋮
    | 9 ->
      "nineteen"
    | _ ->
      failwith "Units can't be >9"
  | n ->
    $"{tensName n} {parseUnits units}"
```

   Notice that we've used some let-bindings at the top of this function so that we can break up the number into "tens" and "units" places. We didn't have to use let-bindings, but we did because it makes the code easier to read. We can't use same shorthand form that we used for `parseUnits`, because we need an explicit identifier to use for the let-bindings at the top of the function.

   An *interpolated string* is introduced here, too. This is a string in which the parts in `{}` braces are replaced by the values that they contain. The `$` at the start of the string is what tells F# that it is an interpolated string.

   If you're confused about why the last case in the match expression works, re-read it *carefully*! We are re-binding the `n` identifier, so within the scope of that branch of the match expression, `n` is *not* the same as the `n` that is bound in the topmost line.

   You will need to create a `tensName` function, with the signature `int -> string`, to get this to work. Such a function is very similar to what you've already done in step 2, so you should have no problem with it.

   *Hint*: If you have written the `tensName` function and are getting told that it is "not defined", check if you have put it at the correct place in the code. The `tensName` identifier must be defined *before* you can use it.

4. Now let's make a function that can handle numbers in the range [1,999]. Once again, we'll use previously-defined functions to help us out.

```
let parseHundreds n =
  match (n % 1000) / 100 with
  | 0 ->
    parseTens (n % 100)
  | h ->
    match n % 100 with
    | 0 ->
      $"{parseUnits h} hundred"
    | t ->
      $"{parseUnits h} hundred and {parseTens t}"
```

5. We'll now make a function to parse numbers in the range [1,9999]. We can do this in a similar way to the `parseHundreds` function, with a few changes. The following cases need to be correctly handled:

   - 299 ↦ two hundred and ninety nine
   - 2999 ↦ two thousand nine hundred and ninety nine
   - 2099 ↦ two thousand and ninety nine
   - 2009 ↦ two thousand and nine
   - 2000 ↦ two thousand

```
let parseThousands n =
  match (n % 10000) / 1000 with
  | 0 ->
    parseHundreds (n % 1000)
  | t ->
    let prefix =
      $"{parseUnits t} thousand"
    match n % 1000 with
    | 0 ->
      prefix
    | h ->
      match (n % 1000) / 100 with
      | 0 ->
        $"{prefix} and {parseHundreds h}"
      | _ ->
        $"{parseHundreds h}"
```

6. We'll tie everything together into a single function that can parse numbers in the range [-9999, 9999].

```
let parse = function
  | 0 -> "zero"
  | n ->
    match n < -9999 || n > 9999 with
    | true ->
      failwith "Out of range!"
    | _ ->
      match n > 0 with
      | true ->
        parseThousands n
      | _ ->
        $"minus {parseThousands n}"
```

   By now, you will have noticed how more complicated functions build on simpler functions. Functions are the workhorses of functional programming, so this is the kind of design that you will find again and again!

7. Now we'll make this into a "real" program that you can run from the console, rather than running through F# Interactive. Leave the `[<EntryPoint>]` annotation intact, and replace the `main` method with this:

```
let main _ =
  printf "Gimme a number: "
  System.Console.ReadLine ()
  |> int
  |> parse
  |> System.Console.WriteLine
  0
```

   You can now build your program and run it from the command line! You will probably have to actually go to the command line console and run this. If you don't, the answer will flash up for a few milliseconds and then the program will end ... and the console window will probably close before you are able to see it.

## Extensions

1. Can you extend this to work for numbers in a larger range — up to 99999 or 999999, for example? Try it out!

2. In step (3), you created a `tensName` function. If you created it outside of `parseTens`, move it inside `parseTens`. If you created it inside `parseTens`, move it outside. How difficult is it to make this kind of code change?

# Iteration

F# supports various looping constructs, including `while` and `for`, but we won't be using them. The functional way to loop is by using ✪**recursion**.

Recursion is the calling of a function from within itself, usually with a different input. This cannot be achieved with the syntax that you already know because a `let` binding is only accessible after its definition, not within its definition. However, we can use the special keyword `rec` to allow the use of a bound name *within* its definition. This allows us to create recursive functions.

```
let sumBetween m n =
  let rec sumTo v =
    match n=v with
    | true ->
      v
    | _ ->
      v + sumTo (v+1)
  match m>n with
  | true ->
    0
  | _ ->
    sumTo m
```

This function adds up the numbers in the range [m,n]. It works because the `rec` keyword allows the identifier `sumTo` to be referred to *within* the function definition. Trace through it and understand how it works.

A recursive function will always have one or more ✪**base case**s, which describe the conditions under which the recursion stops, and one or more ✪**recursive case**s, which describe the conditions under which the recursion continues. It is extremely common to see the following structure in many looping functions:

```
let rec funcname input =
  match expr with
  | basecase_pattern ->
    basecase_expression
  | recursive_pattern ->
    recursive_expression
```

Verify for yourself that our earlier example does follow this pattern.

## More efficient recursion

When a function call is made, a new stack frame typically needs to be allocated, and this takes up a bit of space and time. Since recursion involves calling a function for each iteration, it is typically slower than looping in a non-functional language.

However, it is only necessary to set up a stack frame because some work may remain to be done after a recursive call is made. For example, examine line 7 in the example. After the call to `sumTo (v+1)` is completed, the resulting value must still be added to `v`. It is only after this is done that *all* the work in the function is complete. If a new stack frame was not set up, we would lose the ability to continue with the additional work after the recursive call completes.

To make recursion just as efficient as iteration, most functional languages (including F#) support a technique called ✪**tail recursion**. Using this technique, the compiler tries to see if any work remains to be done after a particular recursive call is made. If *no work remains to be done*, then the recursive call is said to be in ✪**tail position**, and the *same* stack frame can safely be used to do the recursive call. If no new stack frame needs to be allocated, then recursion becomes just as efficient as iteration!

Converting some (or all) of your recursive calls into tail recursive ones can improve the efficiency of your program. If you convert all of your function's recursive calls into tail recursive calls, then we say that your function is tail recursive.

## Tail recursion, practically

To convert a non-tail recursive call into a tail recursive one, you must pass all the relevant data that you still need to use to the recursive function. This is often done by means of one or more ✪**accumulator parameters**. However, if no data needs to be passed onward, then no additional parameters are necessary.

```
let sumBetween m n =
  let rec sumTo v acc =
    match n=v with
    | true ->
      v + acc
    | _ ->
      sumTo (v+1) (v+acc)
  match m>n with
  | true ->
    0
  | _ ->
    sumTo m 0
```

This function adds up the numbers in the range [m,n], using tail recursion. The accumulator parameter is called `acc`. Notice that after the recursive call is made, no work whatsoever remains to be done. Trace through the code and understand how it works.

# Exercises

1. Explain the problem that is solved by an *accumulator* in a tail recursive function.
2. What would happen if a recursive function does not have a base case, or never executes its base case?
3. What characterises a recursive call in *tail position*?
4. Convert the following iterative C# methods into recursive F# functions:

   (a)
   ```
   int NumOdd(int start, int end) {
     int total = 0;
     for (int i = start; i < end; i++)
       if (i%2 == 1) {
         total++;
       }
     }
     return total;
   }
   ```

   (b)
   ```
   int NextMul(int start, int n) {
     while (start % n != 0) {
       start++;
     }
     return start;
   }
   ```

   (c)
   ```
   bool IsPrime(int n) {
     int test = n-1;
     while (test > 1)
       if (n % test == 0) {
         return false;
       }
       test = test-1;
     }
     return true;
   }
   ```

# Tutorial

In this tutorial, we will practice creating recursive functions.

## Square root

The square root of a number $n$ is the value which, multiplied by itself, will result in $n$. Square roots are defined for positive numbers only. The Newton-Raphson method finds a square root by guessing a number and then improving the guess iteratively until it is "good enough". Given a guess $x_i$ for a number $n$, we can say that the next approximation $x_{i+1}$ can be found using the equation

$$x_{i+1} = \frac{x_i + \frac{n}{x_i}}{2}$$

Let's make our own implementation of a square root function.

1. We will start by checking the input for validity.

   ```
   let sqrt n =
     match n <= 0.0 with
     | true ->
       failwith "Impossibru!"
     | _ ->
       calculate (n/2.0) 0
   ```

   Of course, we still need to write the `calculate` function. For now, notice that we pass along a `0` parameter, which will keep track of the number of times that we're iterating through.

2. Let's write `calculate`. We will place it between the `let sqrt n =` and `match n <= 0.0 with` lines, indented appropriately. Ten approximations should get us quite close to the "real" answer.

   ```
   let rec calculate guess i =
     match i with
     | 10 ->
       guess
     | _ ->
       let g = (guess + n/guess) / 2.0
       calculate g (i+1)
   ```

   Once you're done, try it out in F# Interactive! You should have some very good approximations: `sqrt 2.0` should give you `1.414213562`, for example.

### Extensions

1. Is `calculate` tail recursive? How can you demonstrate that it is or is not?
2. Why did we place `calculate` within `sqrt`? What benefit do we get?
3. Instead of stopping arbitrarily after 10 iterations, you could choose to stop when two successive guesses are "close enough". Can you modify the code to do this?

## Chocolate wrappers

A shop sells chocolates for $r$ rands each, and will exchange $e$ chocolate wrappers for a new chocolate. Assuming that you start off with $c$ rands, how many chocolates will you be able to obtain from the shop?

Work out the answers yourself for the following values of $c$, $r$, and $e$, using a pencil and paper.

- $c = 15, r = 1, e = 3$ (you should get 22)
- $c = 16, r = 2, e = 2$ (you should get 15)
- $c = 12, r = 4, e = 4$ (you should get 3)
- $c = 6, r = 2, e = 2$ (you should get 5)

1. We need to keep track of the two things that we can use to buy chocolates: cash and wrappers. If we can't buy chocolates with either, then we're done! If we can, then we should buy them, unwrap them, and see if we can make another trip to the store. Create this function:

   ```
   let rec buy c e cash wraps =
     match cash/r with
     | 0 ->
       match wraps/e with
       | 0 ->
         0
       | n ->
         n + buy c e cash (wraps % e + n)
       | n ->
         n + buy c e (cash % r) n
   ```

   The `cash` tracks how much cash we have on hand, and the `wraps` tracks how many wrappers we have on hand. Initially, we start off with $r$ cash and 0 wraps. Use the code to work through the problems above, and verify that you obtain the correct solutions.

2. It's very awkward to call this function: we have to expect our callers to always pass a 0-valued `wraps` input, for example. We can do much better. Change your code as follows:

   ```
   let chocs c r e =
     ⊙ ⟵ buy function goes here!
     buy c e r 0
   ```

   Now we can call `chocs 15 1 3` for the first sample problem — much nicer!

3. We're not done yet. Now that we have the `c` and `e` in scope for `buy`, we don't need to keep passing them through. Simplify the `buy` function by removing the `c` and `e` inputs entirely. The last line of the `chocs` function should then read: `buy c 0` .

4. This all seems quite nice, but it could be more efficient. Let's make `buy` into a tail recursive function. Change it to read as follows:

```
let rec buy cash wraps t =
  match cash/r with
  | 0 ->
    match wraps/e with
    | 0 ->
      t
    | n ->
      buy cash (wraps % e + n) (n + t)
    | n ->
      n + buy (cash % r) n (n + t)
```

# Basic data structures

A data structure or ✪**composite type** helps you to model a problem in a way that makes it easier to encode or solve using a programming language. There are many different kinds of data structure, but we will only be discussing three of the more common kinds of structure here.

1. A ✪**tuple** or ✪**record** allows you to squash together a bunch of data into a single structure. Example use case: representing a person with a name *and* a birthdate *and* a weight *and* a height, and so on. It makes sense to keep all of these things together in one data structure.

2. A ✪**discriminated union** allows you to represent discrete alternatives. Example use case: representing the states that water can be in by using the different cases "Solid", "Liquid", and "Gas"[1]. These states are discrete alternatives because water cannot be in more than one state at a time.

3. A ✪**list** allows you to represent zero or more values, all of which must have the same type. Example use case: storing the titles of favourite books.

F# supports more data structures than this natively, but these will be enough to allow us to start representing a wide variety of problems. This chapter will cover tuples and discriminated unions; we will leave discussions of lists and records until later.

Throughout the descriptions of data structures, take notice of two things:

1. Each kind of data structure is created in a different way.

2. Pattern syntax for a data structure is often very similar to creation syntax. If you learn the one, then the other is typically very easy to remember. Pattern-matching is used throughout for ✪**structural decomposition**: the extraction and binding of specific data from within a data structure.

**Know your Syntax!**

The similarity of pattern syntax and creation syntax makes it much easier to learn both of them, but it also makes it much easier to *confuse* them! Go through your syntax sections and take very careful note of where patterns and expressions occur in the syntax. Remember that a `pattern looks like this` and an `expression looks like this` in the syntax sections.

[1] Yes, I know that there are five states of matter. No, I'm not going to include the other two.

## Tuples

**Syntax: tuple creation**

$expr_0$ , $expr_1$ , ··· , $expr_n$

**Syntax: tuple pattern**

$p_0$ , $p_1$ , ··· , $p_n$

A tuple is created by using the comma operator. Each tuple has a certain number of items in it; this is called its ✪**arity**. A tuple must have at least 2 elements.

Each pattern in the tuple pattern matches an element of the tuple. The arity of the pattern must match the arity of the tuple; otherwise, the code will not compile.

The type of a tuple is the type of each of its components, separated by a `*` symbol.

```
"F#", 4.0, 2015, (), 'K'
```

This is a tuple with arity 5. Its type is `string * float * int * unit * char`. The following patterns will all match the above tuple:

- `"F#",_,_,_,_`
- `_,4.0,k,t,_`
- `t,_,y,_,'K'`
- `_`
- `_,_,_,(),c`

## Discriminated unions

Each alternative in a discriminated union is called a ✪**case**. Each case can have a value associated with it.

**Syntax: discriminated union definition**

**type** Name =
| $Case_0$ **of** $field\text{-}name_0$ : ***$type_0$***
⋮
| $Case_n$ **of** $field\text{-}name_n$ : ***$type_n$***

The type of a discriminated union value is the name of the discriminated union. Each case identifier must start with an uppercase letter. Case identifiers are

sometimes called ✪**label**s, ✪**tag**s, or ✪**constructor**s. A case may optionally be linked to data using the `of type` syntax. In this syntax, the specified `type` may be any type, including the type of the discriminated union itself (as referred to by its `Name`); this feature makes recursive data structures, such as trees, very easy to implement.

Field-names are optional and, if included, must start with a lowercase letter. Field-names can make pattern-matching much easier. When a field has a name, we will say that it is a ✪**named field**; when it does not, we say that it is an ✪**anonymous field**. Although you can have a discriminated union case that has both named and anonymous fields, there are very strict syntax rules that can make creation and pattern-matching difficult for such cases; therefore, you are strongly advised to either name *all* the fields of a case or name none of them.

```
type WinVersion =
| Seven
| Ten
type OS =
| Windows of WinVersion
| OSX
| BSD of string * int * int * int
| Linux of name : string * version : float
```

The `OS` discriminated union represents different operating systems.

**SYNTAX: DISCRIMINATED UNION CREATION**

Case `expr`

A tag *without* linked data will construct the discriminated union simply by writing the tag name. A tag *with* linked data is a function which, when supplied with its input, will construct the discriminated union. As a convenience, if you have a named field, you can specify that named field (in brackets) and give it a value[2].

```
let t =
  Linux ("NixOS", 20.09)
let v =
  OSX
let z =
  Linux (version = 20.04, name = "Ubuntu")
let x =
  "Dragonfly", 5, 8, 3
let b =
  BSD x
let r =
  Windows Ten
```

[2]However, if you have some named and some anonymous fields, then this may not work unless you are quite careful about your syntax. My advice: don't mix named and anonymous fields.

Each of the bindings above will create a discriminated union of type `OS`, and bind it to an identifier. Note than when brackets are used, they are used only to *group* the elements of a tuple so that the language can know exactly where the tuple begins and ends. When no misinterpretation is possible—for example, in the `BSD x` case—then no brackets are necessary.

**SYNTAX: DISCRIMINATED UNION PATTERN**

Case `pat`

If a case is defined as having linked data, then a pattern to match that data is mandatory; if a case is defined without linked data, then no such pattern is necessary. A discriminated union pattern will only match when both the tag and data (if there is linked data!) match. If you have one or more named fields, then pattern-matching can be used to extract only the fields that you are interested in. If multiple fields are specified in the pattern, then they must be separated by semicolons (i.e. `;`).

The following patterns will all match one or more of the values bound above.

- `Linux _`
- `OSX`
- `Linux (a,b)`
- `_`
- `Linux (version = v)`
- `Linux (version = v; name = n)`
- `blobby`
- `Linux ("NixOS",ver)`
- `Windows _`
- `BSD _`
- `Windows Ten`
- `BSD (_, major, minor, patch)`

Unfortunately, if the linked data of a case is a tuple, F# will not allow you to get the entire tuple value through an identifier pattern; you will need to get the tuple components instead. This means that a pattern such as `Linux k` will *not* succeed, but `Linux (k,m)` will.

**THE ; VS , INCONSISTENCY**

When you create a discriminated union using named fields, you will write something like:

```
Linux (name = "Debian", version = 10.3)
```

Notice that you use a comma to separate the field names. If you want to pattern-match against this without using fields, you would write:

```
Linux (x, y)
```

However, when you pattern-match against this *using fields*, you must use a semicolon:

```
Linux (name = x; version = y)
```

Make sure that you get this right[a], or you will have some very strange errors to deal with!

[a] I don't know why this inconsistency exists. If I had to guess, however, I'd say that the language designers just didn't think it through properly. We're all human, and sometimes we all make mistakes...

## SINGLE-CASE DISCRIMINATED UNIONS

To store grouped data, it is often useful to create a single-case discriminated union which makes the *intention* of the data obvious. The case of the discriminated union is usually given the same name as the type itself, to reinforce the fact that there is only one possible case.

When we have simple data that is stored in a simple discriminated union[3] which serves only to add a layer of meaning to the stored data, we often say that the discriminated union ✪**wrap**s the data. When we get the data from the discriminated union, we say that we are ✪**unwrap**ping it.

A person has a name and age. The corresponding data structure might be:

```
type Person =
| Person of name : string * age : int
```

We could create and bind a `Person` value like this:

```
let p = Person ("Aragorn", 87)
```

We might then unwrap this data through pattern-matching, often with a let-binding or in a function definition:

```
let (Person (_, n)) = p
```

or

```
let isOld (Person (age = x)) = x > 65
```

Note that in a let-binding or as the input to a function, a discriminated union pattern is enclosed by brackets to make it clear where the pattern begins and ends.

[3] A "simple" discriminated union typically has only one or two cases.

## COMPARISON

You can only compare the same data type with the same data type, so you can't compare lists with tuples, or tuples with discriminated unions.

**Tuples** If all the types in a tuple are comparable, then the tuple is comparable to another tuple of the same type. Tuples are compared item-by-item, starting from the first item. As soon as one item is greater or less than the corresponding item in the other tuple, a result is generated.

The following statements are true.

- `8,7,6` > `6,7,8`
- `8,7,6` > `8,6,7`
- `9,2` = `9,2`
- `6,7,8` cannot be compared to `8,5`. The types (***int * int * int*** vs ***int * int***) are different.
- `9,6` cannot be compared to `7,true`. The types (***int * int*** vs ***int * bool***) are different.
- `(fun _ -> 5),4` cannot be compared to `(fun _ -> 2),2`. The types are the same (i.e. ***('a -> int) * int***); however, function types are not comparable.

**Discriminated unions** A tag that occurs earlier in the data definition is considered to be less than a tag which occurs later. A discriminated union is considered to be comparable if all the data linked to all tags of the discriminated union is comparable — or if no data is linked to any tag. If two values are equivalent in terms of their tags, and there is linked data, then the linked data is compared.

# Exercises

1. Consider the following code.

```
let a =
  2, 3
let b =
  9.0, true, ((5, 83), a)
```

   (a) Write down a pattern to extract the value `3` from `a` and bind it to the identifier `t`

   (b) Write down a pattern to extract the values `true` and `3` and bind them to the identifiers `a` and `b` respectively

   (c) Is the pattern that you wrote for (b) the *shortest* pattern that you could use to answer the question? If not, write down a shorter pattern that does the same thing.

   (d) Circle patterns which will match `a`:

   - `_, 3`
   - `3, _`
   - `_`

   (e) Circle patterns which will match `b`:

   - `_, _`
   - `_`
   - `_, true, (a, b, c)`
   - `_, true, _, _`
   - `9.0, _, _`
   - `9.0, true, _, _, _`
   - `b, _, (a, _)`
   - `_, c, ((_, a))`

2. Evaluate `(fun (a,b) -> b-2, a+8) (5,5)`

3. Consider the following function.

```
let so (la, ti) doh =
  match ti + "km", la with
  | _, true ->
    doh + 5
  | _ ->
    0
```

   (a) What is the type of this function?

   (b) If the first line was written as `let so la (ti, doh) =`, what would the type of the function be?

4. What is *structural decomposition*?

5. A circle with a radius, a rectangle with a width and length, and a blob are different shapes represented by the following discriminated union:

```
type Shape =
| Circle of int
| Rectangle of int * int
| Blob
```

   (a) Create a `Circle` with radius 100.

   (b) Revise the definition so that `Rectangle` uses named fields instead. The first field should be named `length` and the second should be named `width`. For the remaining questions, you will be using your revised definition.

   (c) Create a `Rectangle` with length 5 and width 8, *without* naming the fields.

   (d) Create a `Rectangle` with length 5 and width 8, naming the fields during the creation.

   (e) Create a `Blob`.

   (f) Assume that your `Rectangle` from (c) has been created. Now look at the following patterns. If a pattern will match your `Rectangle`, write down **Y**; if it won't match, write down **N**; and if it is an invalid pattern that won't compile, write down **WTF**. If a match is successful, also write down any identifiers and the values that they are bound to.

   i. `Rectangle`
   ii. `Rectangle _`
   iii. `Rectangle (_,_)`
   iv. `Rectangle _, _`
   v. `Rectangle (height, size)`
   vi. `Rectangle (width=k)`
   vii. `Rectangle (height=h; width=w)`
   viii. `Rectangle (width=w; length=h)`
   ix. `Rectangle (width=w, length=h)`
   x. `Shape`
   xi. `Shape Rectangle`
   xii. `Shape.Rectangle`
   xiii. `Shape.Rectangle _`
   xiv. `Shape Rectangle _`
   xv. `Shape _`
   xvi. `Rectangle (5, 8)`
   xvii. `Rectangle (length = 5; width = b)`
   xviii. `Circle _`

6. Would your answers for (f) be the same if you were using your `Rectangle` from (d)?

7. You are making a booking system for a theatre. Each seat in the theatre may be sold, unsold, or occupied by a Very Important Person (VIP). Sold seats are linked to the name of the person they're sold to, and VIP seats are given a numeric rating that indicates how important the VIP is. Define a data type to represent a seat.

8. Consider the following types.

```
type Glass =
| Full of percent:int
```

```
| Empty
type Customer =
| Customer of name:string * drink:(string * Glass)
```

(a) Create a `Glass` that is half-full (or, if you prefer, half-empty).

(b) Circle all the patterns which will match the `Glass` that you created.

i. `Glass`
ii. `Glass _`
iii. `Empty`
iv. `Full`
v. `Full _`
vi. `Full 50.0`
vii. `_`
viii. `Full empty`

(c) Create a `Customer` named Severus Snape. Snape drinks Pumpkin Juice, but his glass is empty.

(d) Assume that your `Customer` from (c) has been created. Now look at the following patterns. If a pattern will match your `Customer`, write down **Y**; if it won't match, write down **N**; and if it is an invalid pattern that won't compile, write down **WTF**. If a match is successful, also write down any identifiers and the values that they are bound to.

i. `Customer _`
ii. `Customer`
iii. `(name = "Severus Snape")`
iv. `Customer (name = "Severus Snape")`
v. `Customer (name=_)`
vi. `Customer (name=n; drink=d)`
vii. `Customer (name=n; drink=(_, glass))`
viii. `Customer (drink=(_; glass))`
ix. `Customer (drink=me)`
x. `Customer (name=n, drink=d)`
xi. `Customer (Drink=(s, Empty))`

9. The technology of the future allows a user to order a drink (tea, coffee, or fruit juice). The tea may be Earl Grey, Rooibos, or Ceylon. Coffee has a strength (weak, OK, or strong) and you can add any number of spoons of sugar to it. Tea and coffee may be iced or hot; juice is always cold. Write down data types that describe a drink order and, as far as possible, make it impossible to describe invalid combinations. Your data types should allow me to order `Tea (EarlGrey, Hot)`.

# Tutorial

In this tutorial you will use the composite types that you've just learned about.

## Commission of Sales

Instead of paying a salary, some businesses give a monthly *commission* to their workers. Each worker will buy some of the product from the business. Then workers will sell the product to customers at a higher price, and pay the money to the business. The business then pays the worker a percentage of the profit.

In this problem, the business sells cosmetics. You can buy branded lipstick (Levron or Maybepoint brands) in either red, green, blue, or black colors. You can also buy nailpolish, in all of the aforementioned colors. Lastly, you can buy mascara. The cost prices for these are as follows:

| | |
|---|---|
| Mascara | R13.99 |
| Blue Levron lipstick | R79.49 |
| Red lipstick | R10.99 |
| Any other lipstick | R12.99 |
| Green nailpolish | R10.34 |
| Red or blue nailpolish | R17.49 |
| Any other nailpolish | R19.99 |

Selling prices add a mark-up of 35%.

We will create a very small system that lets the business (or a worker) figure out how much commission someone has earned, based on what they've bought and sold.

1. Let's start by defining a few data types that help us to represent the problem. First, we'll define the products.

   ```
   type Color =
   | Red
   | Green
   | Blue
   | Black
   type Brand =
   | Levron
   | Maybepoint
   type Product =
   | Lipstick of Color * Brand
   | NailPolish of Color
   | Mascara
   ```

   Here we have defined all of the different products that are offered. We've done this quite differently from how you might have done it in other languages, so let's see why we've laid it out like this.

   - We realize that a product's color can't be red *and* blue, and it can't be black *and* green, at the same time. More generally, we realize that the colors are distinct: an item can't have more than one color. Once we realize that, it's easy to see that the colors form a discrete set of alternatives, and whenever we have that, we should be using a discriminated union to represent it.
   - We realize the same thing about brands. An item is made by one manufacturer, not two. So we represent that as discriminated union as well.
   - Our lipsticks have a color and a brand, so we link a tuple of that data to the `Lipstick` case.
   - Our nailpolish has only a color, so we just link a color to it.
   - Mascara is always black, so there is nothing to link to it.

   Note that the tags of a discriminated union *must* start with a capital letter!

2. Now we should represent a worker who buys and sells products. For working out the commission, the only things that are important are the amounts bought and sold, and the name of the worker.

   ```
   type Worker =
   | Worker of name:string * bought:float *
               sold:float
   ```

   We've made this information into a discriminated union with named fields. This allows us to pick out only the fields that we want to pick out when we are pattern-matching, and it also makes it easy to see what each value means in the record.

3. We will need to see the cost price of cosmetics, so let's make a function that takes in a product and gives us the cost price of that cosmetic.

   ```
   let costPrice = function
     | Lipstick (Red, _) ->
       10.99
     | Lipstick (Blue, Levron) ->
       79.49
     | Lipstick _ ->
       12.99
     | Mascara ->
       13.49
     | NailPolish Green ->
       10.34
     | NailPolish (Red | Blue) ->
       17.49
     | NailPolish _ ->
       19.99
   ```

   Here we use the short syntax of a mapping function to reduce the amount of code we have to

write. We use discriminated union patterns to match the data that is sent in to this function. If a pattern matches, then we send back the corresponding price. Note that the order of the cases can be very important; for example, if the `Lipstick _` pattern is placed at the top, then none of the other `Lipstick` patterns can possibly match.

The second-last pattern is particularly interesting because it shows how you can use an or-pattern to reduce the size of your patterns. Instead of writing the pattern

```
NailPolish Red | NailPolish Green
```

we were able to write `NailPolish (Red | Green)`. Seek opportunities to get better at pattern-matching, because it is very important in controlling your execution path!

Try removing the third, fourth, or seventh cases in the match expression. What warning does F# throw up? If you remove different cases, there is no warning; can you see why?

4. The mark-up on a product is 35%. We can write a very simple function that takes in a product and gives back a selling price.

```
let sellingPrice product =
  costPrice product * 1.35
```

That was easy! We use the already-defined `costPrice` function and just multiply. If we really feel like it, we can even reduce the amount of code that we need to use to define exactly the same function. Consider this rewritten code:

```
let sellingPrice =
  costPrice >> (*) 1.35
```

Can you see why this function is exactly the same as the original `sellingPrice` function? In fact, some would argue that the shorter function is also clearer.

5. How about a function to create a new worker?

```
let newWorker name =
  Worker (name, 0.0, 0.0)
```

This is a trivial function to create a worker with the specified name, who has not bought or sold anything. Don't be afraid to create "trivial" functions: in functional programming, we often create complex functions by manipulating simple functions.

6. We want to express the idea of a worker buying $n$ units of a product from the business. Once again, this is a trivial function:

```
let bought n product (Worker (name, v, sold)) =
  let amt =
    float n * costPrice product
  Worker (name, v + amt, sold)
```

Notice that we have used a pattern-match to structurally decompose our `Worker` value and extract the fields.

7. Now that we can express the buying operation, how about the selling operation?

```
let sold n product (Worker (name, bought, v)) =
  let amt =
    float n * sellingPrice product
  Worker (name, bought, v+amt)
```

Unsurprisingly, this looks very similar to the previous function!

8. All of the pieces are now in place for writing a `commission` function which calculates a person's commission. We'll want this function to give back both the name of the salesperson and their commission.

```
let commission (Worker (name, bought, sold)) =
  let profit =
    sold - bought
  let fraction =
    0.35
  match profit > 0.0 with
  | true ->
    let amt =
      round (profit * fraction * 100.0)
    name, amt / 100.0
  | false ->
    name, 0.0
```

There is no commission if there's a negative profit, of course. We want a two-digit rounded-off amount, so we use an old trick: we multiply by 100, round the result to get rid of any fractional parts, and then divide by 100 to push the last two digits to the other side of the decimal point.

We give back a tuple of name and commission; it's not worth our time to make a separate data structure to represent this information.

9. Go ahead and use what we've made! Write the following, for example, and then pass it down to F# Interactive.

```
newWorker "Alice"
|> bought 5 (Lipstick (Red, Levron))
|> sold 3 (Lipstick (Red, Levron))
|> bought 18 Mascara
|> bought 3 (NailPolish Green)
|> sold 3 (NailPolish Green)
|> sold 15 Mascara
|> commission
```

# Can you be helped?

Introducing functional programming to the dysfunctional mind can result in a number of peculiar mental conditions. These conditions can have serious consequences, including *module failure* and *examination dysfunction*. If you suspect that you may be afflicted by any of these conditions, please get yourself checked out by your nearest functional-programming lecturer, and follow the condition-specific advice given below. If you suspect that one of your classmates is developing one or more of these conditions, encourage them to obtain help and, whatever else you do, do not take any action that might aggravate the condition.

## Scepticalepsy

This mental condition is characterised by only half-believing what the lecturer, language documentation, or online resources say. It is often found during discussions of fundamental concepts such as immutability, first-class functions, or referential transparency. Students afflicted by scepticalepsy often find themselves asking a question, receiving an answer, and then asking exactly the same question again.

> “Values can’t *really* be immutable. Nobody could program like that! There must be a way to change them, somehow, I’m sure... maybe I can trick the language somehow. There’s got to be a way!”

To achieve understanding, the sufferer must learn to have some faith in their lecturer, or at least in the language documentation. If we say that this is how things are, then this is how things are. Simply accept it.

## Analogiculosis

This is characterised by a deep-rooted conviction that a particular concept or feature of a functional language is actually a similar feature in the non-functional language of your choice—despite all evidence to the contrary. Sufferers can often be found staring directly at problematic code and insisting that it works in the way that it clearly does not.

> “So these *match* expression things ... they’re actually *switch* statements in disguise, right? Right? Right!”

Analogiculosis can only be cured by obtaining an understanding of how the constructs in the language actually work. Some therapists report that extensive practical work may be helpful in this regard.

## Functionalophobia

This mental illness is characterized by the belief that functional programming can only be done by the insane[1]. The sufferer’s mind begins to race and whirl whenever they encounter functional code and, seeking to avoid becoming insane, they seek to avoid functional programming itself. A student afflicted by functionalophobia is calm and stable while others are struggling with difficult concepts, and often pretends that functional programming does not exist or, if they accept that it does exist, that it is not something that they would ever have to learn. Extensive and determined procrastination is a symptom shared by all functionalophobia sufferers.

> “What are you doing? Oh ... that stuff. Yeah, I’ll get to it later. I’ll do it on my own. Hey, do you want to go out tonight?”

Functionalophobia is often fatal; the cure is simple, but must be enacted before too much time has passed. Sufferers must study very simple code examples, understand them fully, and then write, run, and modify the code themselves. Simple tutorials or examples are highly recommended. An afflicted student must incrementally move on to more and more complex examples as time passes, but be warned: attempting to move too quickly to a too-complex example may result in a relapse.

## Syntactic dyslexia

This is usually characterised by an inability to distinguish between *patterns* and *expressions*. In severe cases, the sufferer is unable to see larger syntactic differences: function application starts to look like “two variables next to each other”, the unit value starts to look like “a function call without a function”, and so on. Interpretation and understanding of code quickly becomes impossible. Syntactic dyslexia can lead to analogiculosis or even, in extreme cases, functionalophobia.

> “So we bind `x` here ... no, wait, we evaluate `x` ... or do we apply `x`? Or pass it as an input? Is this shadowing or something? I don’t know! Well, I’ll just guess and move on. I’m sure it’s not that important.”

Syntactic dyslexia can be remedied by studying the Syntax sections of these notes carefully. Lines of code

[1] This belief is incorrect. Functional programming can be done by the sane *and* the insane.

that are found in the wild should be written out and compared to the relevant Syntax section, and each part of the line should be characterized as pattern or expression. In extreme cases, the kind of expression—for example, "function application" or "addition"—should also be written down. As familiarity with the syntax increases, the syntactic dyslexia should naturally fade away.

## Hypopractical psychosis

This is characterized by the strong belief that functional programming is purely theoretical, and that no "real" systems are ever written in a functional language. To achieve such a strong belief, a sufferer must ignore significant software such as WhatsApp and Facebook Messenger (both written in Erlang), Facebook spam detection (written in Haskell), Pinterest's web API (written in Elixir), the XBox Achievements system (written in F#), and so on; in addition, they often need to fixate on a particular imperative programming style as being the "one true way" of programming. The disconnect from reality is maintained by constantly shifting the goalposts: whenever an example of real-world functional programming is provided, the sufferer will shift their definition of what a "real" system is to compensate, until only a small subset of software (such as "AAA games") is considered to be a "real" system.

> "But what can you actually *do* with functional programming? There's no way that I'll ever use it! We shouldn't be learning such useless stuff!"

There is no definitive cure for hypopractical psychosis, though some therapists have found that repeated exposure to practical functional programming in a workplace environment can be very helpful. Other therapists suggest introducing functional programming within the "safety net" of non-functional programming (see, for example, C#'s LINQ or the popular `lodash` JavaScript library).

## Space dysmorphopsia

A person suffering from space dysmorphopsia is unable to correctly perceive whitespace in a programming language where whitespace is significant, such as Python or F#. This leads to an inability to understand why their code isn't working. A secondary issue is that sufferers find it difficult to modify their code, simply because they laid it out improperly in the first place. In extreme cases, a sufferer's code will look as though it has been indented randomly, making it very difficult to read—and making it much more likely to have whitespace-related bugs.

| Dysmorphopsic | Correct |
|---|---|
| `let make a = let b = 7`<br>`a + b` | `let make a =`<br>`  let b =`<br>`  7`<br>`  a + b` |

If the name of the `make` function ever needs to change, the dysmorphopsic sufferer will have to re-indent the entire function ... manually.

> "Every time I make a small change, I have to re-indent everything! And then my code just stops working properly, and I haven't changed anything! How can anyone program like this?"

Current therapeutic guidelines recommend that space dysmorphopsic individuals should examine well-indented code and try to emulate it wherever they can. A minority of therapists believe that space dysmorphopsia is merely a symptom of a failure to learn what the Tab key is for, and if this is the case, then the sufferer should immediately stop manually indenting using Space, and start using their code editor's automatic Tab and Shift+Tab identation.

# Type theory

Though not an intrinsic part of functional programming, F# and many other modern languages—functional and non-functional—use ✪**type systems** to make programming more reliable. Typed functional programming has become so pervasive that a discussion of modern functional programming cannot be complete without it.

A more ✪**type-safe** system can make programs more reliable by catching more kinds of errors before the code ever executes; however, more type-safe systems are also more restrictive than less type-safe ones, which makes certain things difficult to express. In a type system, every value is considered to have a type, and the type of a value restricts what operations can be performed with that value.

## Type-safety

Let's consider three possible type systems, and see how they would deal with the three-byte string `"moo"`.

- An ✪**untyped** language, such as assembly language, thinks that this is just 3 bytes that happen to be next to each other in memory. You could take any of the values of the bytes and divide it by 2, if you wanted to, and the language wouldn't complain at all. Allowing programmers to do this kind of craziness has led to some very difficult-to-solve bugs! Aside from assembly language, there are almost no untyped languages that are in common use today.

- A less type-safe language, such as PHP, knows about types and won't allow certain operations.

  ```
  $moo = true;
  $moo();
  ```

  Here we've tried to use a boolean value as if it was a function, and PHP won't allow it. However, PHP is much more type-unsafe than a language such as C#; for example, it will allow us to write code such as `"hello"/2` and will evaluate it successfully to have the result `0`. Here the division operation has been applied to a string, and the language has no problem with that.

- A more type-safe language, such as F#, insists that certain operations can only be performed on certain types. Not only is `"hello"/2` considered to be invalid, code such as `8.0/2` is also considered to be invalid since the types used for division don't match up exactly. It restricts the operations that can be used with a value by looking at the type of the value, and it will not implicitly convert types without the programmer's explicit authorisation.

Some languages are untyped and some are completely type-safe, but most (including F#) fall somewhere in-between these two. Type-safety leads directly to reliability: when you apply an operation, you can be absolutely certain that the operation has valid semantics for the value that is used with it, and "nonsense results" such as `!("abc"/2)[3]-'4k' = -5`[1] are impossible. Because "nonsense results" are impossible, semantic errors cannot occur because of such results.

## Parametric polymorphism

Being type-safe is a good thing, but it reduces flexibility. Consider the following function:

```
let whenZero a b c ->
  match a with
  | 0 -> b
  | _ -> c
```

This function will return `b` when `a` is 0, and return `c` otherwise. Clearly, `a` must be an integer; otherwise, it could not be matched against the constant pattern `0`. What should the type of `b` and `c` be? Clearly, they should have the same type[2]. It would be nice to be able to define this function for integers, floating-point values, strings, and anything else. In a basic type-safe system such as the ones described above, this is not possible; in a type-unsafe system, it is possible, but we must then give up the reliability of a type-safe language. Unfortunately, if we give up a type-safe language, then code such as `(whenZero n 5 "moo") + 3` becomes plausible: everything will work fine as long as `n` is `0`, and bad things will happen whenever it is not `0`.

Is there a way to keep type-safety and also be flexible? Yes, there is: a type system can be ✪**parametrically polymorphic** (or ✪**generic**; the two terms can be used interchangeably). In a parametrically polymorphic type system, a type may be ✪**concrete** or parametrically polymorphic. A composite type may include a parametrically polymorphic component, and therefore be partially parametrically polymorphic.

- A concrete type is something like `int` or `unit`; the operations that can be used with this type are known.

[1] This is valid PHP 7.0.3 code.

[2] Remember: each branch of a match expression must result in the same type of value.

- A parametrically polymorphic type is defined in relation to other types in an expression, but has no particular operations that are specifically valid for it. It is indicated by an apostrophe and some letters (e.g. `'a` or `'moo`). At runtime, any concrete type can be substituted in for a particular generic type.

Let us return to the example of `whenZero`. We know that `b` and `c` must have the same type, and we also want them to be *any* type. F# will automatically ✪**generalise** the type of `whenZero` appropriately so that these constraints are met; the type of the function will therefore be `int -> 'a -> 'a -> 'a`. When type-checking occurs, F# will ensure that each application of `whenZero` has its `'a` replaced by an appropriate concrete type, and will refuse to compile code which is type-unsafe.

## Partially parametric polymorphism

The basic concept of partially parametric polymorphism is simple: imagine that you want to create a function that checks whether the length of a list is odd. Of course, the list will hold elements of a particular type, perhaps `int`s. In F#, we might say that it is an `int list` if it holds `int` values. Should our length-checking function have the signature `int list -> bool`, then?

If it does have that signature, that wouldn't be as useful to us as it could be. After all, there is nothing in the length-checking function itself that makes it only applicable to lists of integers. We might want to check whether the length of a list of strings is odd, or whether the length of a list of functions is odd. However, if we make the function fully generic, then the signature would look like `'a -> bool`; and you would be able to pass in something like a floating-point number instead of a list, and our type-safety would break down!

To keep both flexibility and type-safety, F# allows its data structures to be partially generic. This means that we are able to specify that the input must be a list, and also specify that the elements of the list can be of any type. The signature of the function could therefore be specified as `'a list -> bool`—notice that the input is not fully generic, but partially generic!—and all problems fall away.

# Type inference

F# is able to identify the type of an identifier by looking at the way in which it is used. Code is read top-to-bottom, left-to-right, until the language has identified a particular type that an identifier must be[3]. It is important to understand that the language does not *guess* the type of an identifier; it *proves* what it must be by constraining the type based on the available information. This process of identifying the type by looking at how it is used is called ✪**type inference**.

Type inference can be very easy to do. For example, if an identifier is bound to a string value, then it must have the ***`string`*** type. An important limitation to be aware of is that when an arithmetic operator such as `+`, `-`, or `*` is used, F# will assume that any identifiers it is used with must be `int`s if no additional information exists to clarify the type.

### Piping for improved inference

F# does type inference from left to right, and the pipe operator allows us to reorder an expression such as `f (g x)` to be `x |> g |> f`. These expressions are semantically equivalent, but in the first expression, F# encounters `f` first and tries to infer its type; in the second, F# encounters the input value `x` first. Examining the input value first sometimes allows it to make inferences that it would otherwise be unable to make, so if you write code that uses the pipe operator more frequently, F# might be able to do better type inference for you!

If there is no information that constrains an value to be a particular type, then F# understands that value to be generic.

```
let t r = fun e -> e + 2.5
```

What is the type of this function? To find it, we must examine the function itself and add constraints wherever they are necessary. Try and work from left to right, top to bottom.

1. Let's begin with a completely generic function, `'a -> 'b`.
2. The output of `t`, when given its input `r`, is the function `fun e -> e + 2.5`. This is a function type, we can therefore constrain `'b` to be a function type. Our updated type for `t` looks like `'a -> ('b -> 'c)`.
3. The expression `e + 2.5` must result in a float. Therefore, we can update our type to be `'a -> ('b -> float)`.
4. The only thing that can be added to a float is another float, so `e` must be a float. Therefore, we can update our type to be `'a -> (float -> float)`.
5. There is nothing that constrains `r`, so it can remain as type `'a`.

That concludes the type inference. The brackets are unnecessary because a curried function `'a -> 'b -> 'c -> ...` is always read as `'a -> ('b -> ('c -> (...)))`, so we can remove

[3] If you are interested in the exact details of this process, look up *Hindley-Milner type inference*.

the brackets. The type of the function is `'a -> float -> float`.

## Type annotations

**Syntax: type annotation**

`(x : type)`

Any pattern can be supplemented by a ✪**type annotation** which overrides the type inference logic and forces the language to only match that pattern to values of a particular type. The round brackets in annotation syntax are not strictly necessary, but are useful for avoiding ambiguity in larger patterns.

You can also use a generic type annotation (such as `'a`) to force F# to regard a pattern as being generic. If you do so when the pattern cannot be generic, then F# will issue warnings specifically point out any lines of code which force the pattern to be of a more specific type.

Another use of type annotation is to specify what a named function's output should be. When a function is bound to a name, a ✪**type annotation** can be used to indicate the type of the function's generated value, using the following syntax:

**Syntax: function output type annotation**

`let` ident $i_1 \cdots i_n$ `: outType =` expr

Don't go too crazy with type annotations, though. F# is pretty good at figuring out what things are. Allow the compiler to figure it out and you'll typically end up with code that is constrained enough to be correct, while also being as flexible as possible.

### "Some" or "Any"?

When a type is parametrically polymorphic, it must be substitutable by *any* concrete type, not just *some* concrete types. The difference is subtle, but important! For example, consider the following function:

```
let choice (x : 'a) (y : 'a) : 'a = x + y
```

Send this down to F# Interactive and you'll see the following warning:

```
  This construct causes code to be less generic
than indicated by the type annotation
```

F# infers that its type is actually `int -> int -> int`. This occurs because the `+` operator works for `float`s; it works for `int`s; it even works for `string`s. So we know that it works for *some* types, but it doesn't work for *any* type, such as `bool`. Accordingly, F# is unable to accept that it is parametrically polymorphic, and defaults to what it considers to be sensible[a].

[a] Other typed functional languages, such as Haskell, apply more sophisticated reasoning and can handle such cases.

## Type errors

By now, you've probably seen the dreaded error that says:

> *This expression was expected to have type [...]*
> *but here has type [...]*

You get this error when F# thinks that you've used an expression in a way that forces it to have *more than one* type — and since this is impossible, one of the ways that you've used it must contradict the other way. The F# compiler is quite good at finding these issues, so don't immediately assume that it's wrong! Instead, try to fix the error.

To fix a type error, you need to do your own type inference and verify that there's no contradiction between your types. This is what functional programmers are mean when they say that the types in a system must "line up". Try following this process:

1. Start with a fresh F# Interactive session: reset it.
2. Add type annotations to every identifier in your function. These will tell F# what you expect the type to be. Send what you have down to F# Interactive, and note any errors or warnings. These usually indicate where you can focus your attention.
3. Begin at the top of your function and continue in a left-to-right fashion, paying special attention to the places where F#'s type-checker has found problems. As soon as you reach a contradiction, you're done! Fix the problem by removing the contradiction.
4. If you can't find the problem after a bit of thought, then your function might be too large to keep in your head. This sort of "debugging" works best when you have small functions, so try to break your functions into small pieces so that you make your own life easier.

The above process is also useful whenever F# seems to be inferring the type of a function or identifier incorrectly.

# Generic composite types

Generic data structures — in other words, data structures which contain one or more fields/-cases which are parametrically polymorphic — can be defined by specifying the existence of generic ✪**type parameter**(s) in angle-brackets after the type-name in the type definition. These type parameters should be present in the type definition.

**Syntax: generic discriminated union definition**

**type** Name<***'a,'b,···,'z***> =
| Case$_0$ **of** field-name$_0$ : ***type***$_0$
⋮
| Case$_n$ **of** field-name$_n$ : ***type***$_n$

The type of such a generic data structure is the name of the type, as parameterised by the concrete types that replace the parametrically polymorphic variables. Two generic types are not considered to be the same type, unless both types are parameterised by comparable type parameters, and are therefore not comparable. A type parameter $x$ is comparable with a type parameter $y$ if one of the following conditions is true:

- $x$ and $y$ are the same concrete type.
- $x$ is generic and $y$ is a concrete type (or *vice versa*).
- $x$ and $y$ share the same generic type parameter.[4]

```
type Tree<'a> =
| Empty
| Node of 'a * Tree<'a> * Tree<'a>
let q =
  Empty
let r =
  Tree("moo", Empty, Empty)
let s =
  Tree(7, Empty, Empty)
```

Here we define a binary tree. It contains two cases: an `Empty` case for when there's nothing in a tree, and a `Node` case that has a tuple consisting of the data to be stored in the node and the "left" and "right" sub-trees.

We then bind three identifiers: `q` is a `Tree<'a>`, `r` is a `Tree<string>`, and `s` is a `Tree<int>`. We will be unable to compare `r` and `s`, because the types are different. We will, however, be able to compare `q` to `r` and `s`. It will compare as being less-than either of those because the `Empty` case is defined above the `Node` case.

[4] If you are comparing generic types, F# will restrict your function to be usable only in cases when the concrete type — supplied by calling code on a case-by-case basis — is comparable.

# Options and Results

One of most important built-in types is the `Option<'a>` type, often referred to informally as the `Option` type:

```
type Option<'a> =
| None
| Some of 'a
```

You will find such a type in almost all functional languages, although its name might differ. In a type annotation, an `Option<'a>` can be written as `'a option`. An `Option` type is often used to explicitly indicate that no value exists. In other languages, a `null` is used to indicate that no value exists. An `Option` type is a better choice than a `null` for the following reasons:

1. To extract a value from an `Option` type, a programmer must explicitly pattern-match *both* cases. If this is not done, then F# will display a warning. The `Option` type therefore forces the programmer to deal with the successful and unsuccessful case, whereas it is easy to forget to deal with the possibility that a value could be `null`. Forgetting to deal with a `null` can lead to an error when a `null` value is used.
2. The `Option` type explicitly indicates to the programmer that a particular value may not be available. In all other cases, the programmer can assume that a value will be available.

A variation of the `Option` type is the `Result<'a,'b>` type, informally referred to as the `Result` type, defined as

```
type Result<'a,'b> =
| Ok of 'a
| Error of 'b
```

Whereas the `Option` type allows you to explicitly express a value that may or may not exist — and, related to that, whether a value-generating operation has succeeded or not — the `Result` type allows you to also express the kind of error which has occurred if a value could not be created.

Since both `Result` and `Option` types are simple discriminated unions that are used to add another layer of meaning to existing values, they are considered to be built-in ✪**wrapper types**. You may hear a functional programmer way that they've "wrapped" a value in an `Option`, or "unwrapped" a `Result`.

# Exercises

1. Briefly define *type inference*.
2. Write down the type of the following functions:
   (a) `fun p -> "quiz" + p`
   (b) `fun p -> p + 600`
   (c) `fun p -> fun r -> r-p`
   (d) `fun p -> "quiz"`
   (e) `fun p r -> p-r`
   (f) `fun 8 -> 25.13`
   (g) `fun status -> fun () -> status/2`
   (h) `fun act -> act "boldly"`
   (i) `fun i s -> s () - 10 * i`
3. Write down the type of each of the following functions.
   (a) `fun (d,a,e) -> d,e,a,d`
   (b)
   ```
   fun a p ->
     match p with
     | (_,3) ->
       a (p, p)
     | a ->
       p
   ```
4. Study the following type.
   ```
   type Circe<'a,'b> =
   | Transform of ('a -> 'b) * string
   | Die
   ```
   Write down a pattern that matches the value `Transform (string, "int")`, but does not match `Transform (int, "string")`.
5. Write down the definition of the built-in `Option<'a>` type.
6. The built-in function `defaultArg` has the type `'a option -> 'a -> 'a`. Using only the type, figure out what it does.
7. The `bind` function has the type
   ```
   ('a -> 'b option) -> 'a option -> 'b option
   ```
   Write the function.
8. Study the following type which represents a node in a linked list.
   ```
   type Node<'a> =
   | Nothing
   | Next of 'a * Node<'a>
   ```
   (a) Create an empty linked list.
   (b) Create a linked list that holds the values 4.22, 1.9, and 0.45.
   (c) Write a `map` function with signature
   ```
   ('a -> 'b) -> Node<'a> -> Node<'b>
   ```
9. Why could the `Result` type be considered superior to the `Option` type?
10. Before you attempt this exercise, look through the "Strings" sub-section on page 59. Write down the type of the following functions:
    (a) `fun toi -> fun tli kea -> "toiger"`
    (b)
    ```
    fun apple (bake : float) ->
      apple-bake
    ```
    (c)
    ```
    fun sugar cake ->
      $"Sugar: {sugar = true}"
    ```
    (d) `let super duper = duper`
    (e)
    ```
    fun titan fall ->
      (fall+8) |> titan |> (+) 0.03
    ```
    (f)
    ```
    let koan yin yang : bool =
      (1.0 / yang) |> yin
    ```
    (g) `fun x y z -> (y >> x) z`
    (h) `fun x y -> y x`
    (i) `fun k p -> p k "5"`
    (j)
    ```
    fun t ->
      match t with
      | 'R' ->
        fun a -> a
      | _ ->
        fun _ -> 35
    ```
    (k) `fun k -> (k >> k) ()`
    (l)
    ```
    let twilight a v =
      let s =
        a
      let t =
        v 4
      let issue name =
        $"{s || t > 0} @ {name}"
      issue
    ```
    (m) `fun p -> "quiz"`
    (n) `fun act -> act "boldly"`

# Lists

A list is a singly-linked-list of elements, all of which have the same type[1]. A list has a ✪**head**, which is the first element of the list, and a ✪**tail**, which are the remaining elements of the list. The head is a value and the tail is a list. A list may have zero or more elements.

**Syntax: list creation**

`[` $\texttt{expr}_0$ `;` $\cdots$ `;` $\texttt{expr}_n$ `]`

To create a new list with an additional item at the start, we can use the ✪**cons** operator (`::`) with an expression and a list. The expression must generate a value of the same type as the existing values in the list.

**Syntax: cons**

```
expr :: list
```

The type of a list with elements of type `t` can be expressed either as `t list` or as `List<t>`. The former syntax is preferred. If the type of the elements is not constrained, then it can be generic; this means that a list with a type like `'a list` is common.

```
let a =
  [7;6;5]
let b =
  83::a
```

Here we create a list containing the elements `7`, `6`, and `5`, and bind that list to the name `a`. We then cons an element (`83`) to the list, thus creating a new list, and bind the new list to `b`. Remember that lists, like all data structures discussed here, are *immutable*; therefore, `a` is still bound to `[7;6;5]` after `b` is bound to `[83;7;6;5]`.

The type of both of these lists is `int list`.

As a convenience, a list of integers between $n$ and $m$, where $n < m$, can be created using a special syntax.

**Syntax: int list creation**

```
[n .. m]
```

[1] If you recall the differences between linked lists and arrays, you might think that this makes functional lists "slow" or that it makes them space-inefficient. In fact, functional lists can be surprisingly performant and space-efficient; as with many things in Computer Science, a lot of the magic is in the implementation! See, for example, the following paper: Bagwell, P., 2002, September. Fast functional lists. In *Symposium on Implementation and Application of Functional Languages* (pp.34-50). Springer, Berlin, Heidelberg.

**Syntax: list pattern**

`[` $\texttt{p}_0$ `;` $\cdots$ `;` $\texttt{p}_n$ `]`

Each pattern in the list pattern matches an element of the list. The pattern will not match unless the number of elements in the list is the same as the number of patterns in the list pattern.

**Syntax: cons pattern**

`head :: tail`

The cons pattern matches the head of the list and the tail of the list separately; the tail pattern can be another cons pattern, or a list pattern.

`[9;8;7;6;5]`
The following patterns will all match the above list:

- `[9;8;7;6;5]`
- `[p;8;_;q;_]`
- `h::[r;7;_;5]`
- `x::y::_`
- `_`
- `x::8::y::_::_::[]`

Can you see why each pattern will match?

A new list can be created from two existing lists using the "concat" operator `@`. However, this operator is much more inefficient than the cons operator and should therefore be used less often.

**Syntax: concat**

$\texttt{list}_0$ `@` $\texttt{list}_1$

**@ or ::?**

Adding an element to the start of the list (i.e. the cons operation) is a very quick $\mathcal{O}(1)$ operation. However, concatenating lists using `@` requires traversal of the first list, making it a $\mathcal{O}(n)$ operation where $n$ is the length of the first list. It is therefore preferable to use `::` instead of `@` whenever possible.

## Comparison

If the elements in a list are comparable, then the list is comparable to another list of the same type. Lists are compared element-by-element, beginning from the first element. As soon as one element is greater or less than the corresponding element in the other list, a result is generated. If all items are the same, but one list is longer, then the longer list is considered to be "greater than" the shorter list.

# Exercises

1. Consider the following code:

```
let crow =
  2
let beef =
  [1; crow; crow; 8]
let x =
  crow :: (beef @ beef)
```

   (a) What is the value of `beef`?

   (b) What is the value of `x`?

   (c) Rewrite the `beef` binding so that it only uses an empty list and the cons operator.

2. The `clevva` function is defined as follows:

```
let clevva smoke =
  match smoke with
  | [] ->
    0
  | [e;3] ->
    e
  | _::_::e::_ ->
    e
  | [p] ->
    p
  | _::k::[] ->
    k
  | _ ->
    4
```

   Evaluate the following expressions:

   (a) `clevva [5;3;7;1]`

   (b) `clevva [9;2;5]`

   (c) `clevva [8;6]`

   (d) `clevva [0]`

3. Your code contains the binding `let k::_::m = [0;4]`. If the pattern match will succeed, write down the identifiers and the value associated with each bound identifier. If the binding will *not* succeed, write down **Pattern match fails**.

4. Assume that the following function is defined.

```
let tz (a::b::c::r) =
  match a <= b, c with
  | true, _ ->
    [c]
  | _ ->
    r
```

   (a) Evaluate: `tz [10;8;6;4;2]`

   (b) Evaluate: `tz [20;18;16;14;12]`

   (c) Evaluate: `tz (tz [20;18;16;14;12])`

5. Assume that the following function is defined.

```
let dino mite =
  let rec dino saur bambi =
    match saur with
    | [] ->
      4::bambi
    | [_] ->
      8::bambi
    | x::t ->
      dino t (x+1::t)
  dino mite [25]
```

   (a) Evaluate: `dino [1;3]`

   (b) Evaluate: `dino []`

6. Write down a list that will match the pattern `_::_::[6.0]::_`.

7. Write down the type of each of the following functions.

   (a)
```
fun dayS ->
  dayS |> List.filter ((=) 'n')
```

   (b)
```
fun a p ->
  match p with
  | [] ->
    a (p, p)
  | a ->
    p
```

   (c)
```
fun art -> art::("sm"+art)::["heart"]
```

   (d)
```
let rec sweet ness acc =
  match ness, ness > 0 with
  | 0, _ ->
    acc ness + 0.5
  | _, false ->
    sweet -ness acc
  | loch, _ ->
    sweet (loch-1) acc
```

8. If a pattern would match the value `5.3, "hi", (9, [9])`, write the bound symbols and their values. If the pattern match succeeds but no symbols are bound, write **Nothing bound**. If the pattern would not succeed, or if it is invalid, write **No match**.

   (a) `5.3, "hi", _`

   (b) `hi, _, (_, _)`

   (c) `this, is, a, test`

   (d) `this, is, (a, bad::test)`

   (e) `_, _, (w<10, x::_)`

   (f) `so, "hi", (said, [i])`

   (g) `_, (tess, ting)`

9. Consider the following code.

```
let prefix =
  2,1
let six =
  (prefix, 6), [prefix; 1,2], false, prefix, 3
```

Write a pattern to...

(a) extract the first element from the list in `six` and bind it to the identifier `k`

(b) extract the values `6` and `3` from `six` and bind them to the identifiers `six` and `e` respectively

(c) extract all the `2` values from `six`, binding them to identifiers of your choosing

10. Consider the following just-for-practice code.

```
let prep t xs =
  t::(t+1)::xs
let strip (a,_,t) =
  a*t, t/3.0
let flow p =
  match p with
  | [] ->
    None
  | (b,a)::_ ->
    Some (a + "," + a)
let rap x =
  fun () -> strip x
let flip q =
  let t, next =
    q
  (next, t, t) |> strip
let swatch (b,a) =
  a,b
```

Write down the type of:

(a) `prep`

(b) `strip`

(c) `flow`

(d) `rap`

(e) `flip`

(f) `swatch`

(g) `swatch >> swatch`

11. Circle all patterns which would match the value `Some 5, 7::[11], 9.0`.

(a) `Option _, _, v`

(b) `Some n, [_;_], _`

(c) `_, _::_::_, k`

(d) `_`

(e) `Option<int>, _::_, r`

# Useful functions

**System.Console.ReadLine** $\underset{unit}{()} \rightarrow \underset{string}{\text{userinput}}$

Reads in a string from the user.

**int** $\underset{'a}{\text{input}} \rightarrow \underset{int}{\text{output}}$

Creates an integer value based on the `input`. Throws an exception if the `input` can't be used to create such a value.

**float** $\underset{'a}{\text{input}} \rightarrow \underset{float}{\text{output}}$

Creates a floating point value based on the `input`. Throws an exception if the `input` can't be used to create such a value.

**string** $\underset{'a}{\text{input}} \rightarrow \underset{string}{\text{output}}$

Creates a string representation of the input.

## Strings

To print data to the console, use `printf`. To print data to the console with a trailing newline, use `printfn`. To create a formatted string from data, it is most convenient to use an interpolated string:

```
let should =
  2
let good =
  "or not"
let question =
  2.419
let a =
  'B'
$"{should} {a} {good} {round question} be?"
```

An interpolated string has a `$` before its opening quotation-mark. All of the parts between braces (`{}`) are evaluated, and their string-form values are put into the resulting string. Can you figure out what the interpolated string above will evaluate to? Fire up F# Interactive and see if you're correct!

To print out literal { or } characters in an interpolated string, double them in the string (i.e. write `{{` or `}}`).

You can also use *format strings* (%s, %b, %d, %f, %A, etc) and functions such as `sprintf` to generate strings; you might be familiar with this way of creating strings if you have used C or similar languages.

**String.length** $\underset{string}{\text{input}} \rightarrow \underset{int}{\text{length}}$

Returns the number of characters in the `input`.

**String.concat** $\underset{string}{\text{separator}} \rightarrow \underset{seq<string>}{\text{inputs}} \rightarrow \underset{string}{\text{output}}$

Joins all the strings in `inputs` (which may be, for example, a list of strings) into a single string, inserting `separator` between the joined elements in the `output`.

## Catastrophic errors

If a logic error occurs in the program, it is often best to crash the entire program rather than return an invalid result. The `failwith` function is very useful for this purpose.

**failwith** $\underset{string}{\text{error string}} \rightarrow \underset{'a}{\text{ignored}}$

Raises an exception, using the `error string` as the exception message. If uncaught, this will crash the program.

## Options and Results

**Option.map** $\underset{'a \rightarrow 'b}{\text{function}} \rightarrow \underset{'a\ option}{\text{value}} \rightarrow \underset{'b\ option}{\text{result}}$

If `value` is the `None` case, ignores `function` and returns `None`. If it's the `Some` case, then the value held within it is unwrapped, passed to `function`, and re-wrapped.

**Option.bind** $\underset{'a \rightarrow 'b\ option}{\text{function}} \rightarrow \underset{'a\ option}{\text{value}} \rightarrow \underset{'b\ option}{\text{result}}$

Like `Option.map`, but the re-wrapping is done by the higher-order function.

**Option.iter** $\underset{'a \rightarrow unit}{\text{function}}\ \underset{'a\ option}{\text{value}} \rightarrow \underset{unit}{()}$

If the `value` is `Some`, `function` is executed.

**Option.defaultValue** $\underset{'a}{\text{default}} \rightarrow \underset{'a\ option}{\text{option}} \rightarrow \underset{'a}{\text{result}}$

If the `option` is `None`, return the `default` value; otherwise, unwrap and return the wrapped value.

**Result.map** $\underset{'a \rightarrow 'b}{\text{function}} \rightarrow \underset{Result<'a,'c>}{\text{value}} \rightarrow \underset{Result<'b,'c>}{\text{result}}$

If `value` is the `Error` case, ignores `function` and returns the `Error` case. If it's the `Ok` case, then the value held within it is unwrapped, passed to `function`, and re-wrapped.

A `Result.bind` function, analogous to the `Option.bind` function but applicable to `Results`, also exists.

## List operations

Several important built-in functions make it easy to express very powerful operations in a tiny amount of code. Many of these functions are built into the vast majority of functional languages, and their utility is

so widely acknowledged that they have been steadily making their way into non-functional languages as well. Knowing how to use them in F# will give you transferable knowledge that you can use in non-functional languages such as C#, Java, C++, Ruby, and JavaScript, as well as functional languages such as Haskell, Elm, Phoenix, and Erlang, among others.

In order of importance, you should know *at least* the following operations:

**List.map** $\overset{\text{func}}{(\,'a \to\, 'b)} \to \overset{\text{input}}{\,'a\ list} \to \overset{\text{output}}{\,'b\ list}$

Applies `func` to each element of `input`, returning the resulting list.

**List.filter** $\overset{\text{tester}}{(\,'a \to\, bool)} \to \overset{\text{input}}{\,'a\ list} \to \overset{\text{output}}{\,'a\ list}$

Returns a list containing only the elements for which `tester` returns `true`.

**List.fold** $\overset{\text{combiner}}{(\overset{\text{state}}{\,'a} \to \overset{\text{item}}{\,'b} \to \overset{\text{combined}}{\,'a})} \to \overset{\text{initial}}{\,'a} \to \overset{\text{input}}{\,'b\ list} \to \overset{\text{output}}{\,'a}$

Uses the `combiner` on each element in turn to combine all of the elements of `input` together into a single value. The `combiner` starts by taking `initial` and the first `item` of `input`, and using them to generate an updated `combined` value. This value is then used as the `state` for the next call to `combiner`, until all the elements of `input` are exhausted. The last-generated value becomes the `output`.

This function will throw an exception if `input` is empty.

**List.exists** $\overset{\text{tester}}{(\,'a \to\, bool)} \to \overset{\text{input}}{\,'a\ list} \to \overset{\text{output}}{\,bool}$

Returns `true` if `tester` is `true` for any element of the `input`.

**List.mapi** $(\overset{\text{index}}{\,int} \to \overset{\text{item}}{\,'a} \to \overset{\text{result}}{\,'b}) \to \overset{\text{input}}{\,'a\ list} \to \overset{\text{output}}{\,'b\ list}$

Like **List.map**, but passes the index of the element as well as the value to `func`.

**List.rev** $\overset{\text{input}}{\,'a\ list} \to \overset{\text{output}}{\,'a\ list}$

Returns a list with the elements in the reverse order.

**List.find** $\overset{\text{tester}}{(\,'a \to\, bool)} \to \overset{\text{input}}{\,'a\ list} \to \overset{\text{output}}{\,'a}$

Returns the first element for which `tester` returns `true`. This function will throw an exception if `input` is empty.

**List.init** $\overset{\text{length}}{\,int} \to \overset{\text{generator}}{\,int \to\, 'a} \to \overset{\text{output}}{\,'a\ list}$

Creates a list with with the desired `length` by calling `generator`, passing it the index of each element to be generated.

**List.length** $\overset{\text{list}}{\,'a\ list} \to \overset{\text{count}}{\,int}$

Returns the number of items in the list.

**List.iter** $\overset{\text{function}}{(\,'a \to\, unit)} \to \overset{\text{input}}{\,'a\ list} \to \overset{()}{\,unit}$

Evaluates `function` for every element of the `input`.

**List.iteri** $\overset{\text{function}}{(\,int \to\, 'a \to\, unit)} \to \overset{\text{input}}{\,'a\ list} \to \overset{()}{\,unit}$

Like **List.iter**, but passes the index of the element as well as the value to `function`.

**List.partition** $\overset{\text{tester}}{(\,'a \to\, bool)} \to \overset{\text{input}}{\,'a\ list} \to \overset{\text{passed} * \text{failed}}{\,'a\ list * 'a\ list}$

Splits the `input` elements into two lists, `passed` and `failed`, based on whether `tester` returns `true` or `false` for a particular element.

**List.zip** $\overset{\text{one}}{\,'a\ list} \to \overset{\text{two}}{\,'b\ list} \to \overset{\text{combined}}{(\,'a * 'b)\ list}$

Returns a list consisting of corresponding elements from `one` and `two`.

**List.distinct** $\overset{\text{original}}{\,'a\ list} \to \overset{\text{duplicates removed}}{\,'a\ list}$

Returns a list that does not contain any duplicate elements.

**List.ofSeq** $\overset{\text{sequence}}{\,'a\ seq} \to \overset{\text{list}}{\,'a\ list}$

Returns a list that contains the same elements as the `sequence`. Can be used to conveniently obtain a list of `char`s from a `string`.

**List.head** $\overset{\text{input}}{\,'a\ list} \to \overset{\text{element}}{\,'a}$

Returns the first element in the `input` list. This function will throw an exception if the list has no elements.

**List.tail** $\overset{\text{input}}{\,'a\ list} \to \overset{\text{remaining}}{\,'a}$

Returns the `input` list *without* the first element. This function will throw an exception if the list has no elements.

**List.tryFind** $\overset{\text{tester}}{(\,'a \to\, bool)} \to \overset{\text{input}}{\,'a\ list} \to \overset{\text{output}}{\,'a\ option}$

Returns the first element for which `tester` returns `true`, or `None` if no such element exists.

**List.choose** $\overset{\text{func}}{(\,'a \to\, 'b\ option)} \to \overset{\text{input}}{\,'a\ list} \to \overset{\text{output}}{\,'b\ list}$

Applies `func` to each element of `input`, including the output value in the resulting list only if it is not `None`. In other words, `choose` can filter and map in a single pass through the list.

**List.pick** $\overset{\text{func}}{(\,'a \to\, 'b\ option)} \to \overset{\text{input}}{\,'a\ list} \to \overset{\text{output}}{\,'b}$

Like `List.choose`, but only returns the first result. This function will throw an exception if an appropriate element does not exist.

**List.tryPick** $\overset{\text{func}}{(\,'a \to\, 'b\ option)} \to \overset{\text{input}}{\,'a\ list} \to \overset{\text{output}}{\,'b\ option}$

List `List.pick`, but uses an `Option` rather than throwing an exception.

A complete list of list functions can be found at https://goo.gl/72ZZub.

## Sequence functions

You can treat both strings and lists, among other things, as sequences. The `Seq` module, in addition to the functions mentioned below, contains many functions that perform the same tasks as `List` functions and have the same names as those functions.

**Seq.head** $\overset{\text{sequence}}{seq<'a>} \rightarrow \overset{\text{result}}{'a}$

Returns the first item in the sequence. If no such item exists, an exception is thrown.

**Seq.toList** $\overset{\text{sequence}}{seq<'a>} \rightarrow \overset{\text{result}}{'a\ list}$

Creates a list from all the items in the sequence.

## Mathematical functions

**round** $\overset{\text{input}}{float} \rightarrow \overset{\text{output}}{float}$

Rounds a fractional number to the nearest whole number, preferring rounding towards even numbers instead of odd numbers whenever the fractional component is exactly halfway between two numbers.

**floor** $\overset{\text{input}}{float} \rightarrow \overset{\text{output}}{float}$

Rounds a fractional number to the nearest whole number which is smaller.

**ceiling** $\overset{\text{input}}{float} \rightarrow \overset{\text{output}}{float}$

Rounds a fractional number to the nearest whole number which is larger.

**truncate** $\overset{\text{input}}{float} \rightarrow \overset{\text{output}}{float}$

Removes the fractional component from a number.

**min** $\overset{\text{x}}{'a} \rightarrow \overset{\text{y}}{'b} \rightarrow \overset{\text{result}}{'a}$

Returns the smaller value, choosing between `x` and `y`. This function works with any comparable data types, including strings, ints, and floats.

**max** $\overset{\text{x}}{'a} \rightarrow \overset{\text{y}}{'b} \rightarrow \overset{\text{result}}{'a}$

Returns the larger value, choosing between `x` and `y`. This function works with any comparable data types, including strings, ints, and floats.

# Exercises

1. The `Result` type would not be used by an operation such as `choose`. Why not?

2. Which is the best operation to use to achieve each of the following results? (You may assume that any data types which are named are defined suitably; we'll get to custom data types in the next chapters)

   (a) Finding the largest item in a list

   (b) Finding all the negative numbers in a list

   (c) Getting the length of each word in a list

   (d) Checking whether any of the lists in a list is empty

   (e) Creating space-filled strings with increasing lengths from 1 to 50 characters.

   (f) Dividing a class of `Students` into those who have more then 50%, and everyone else

   (g) Finding a `Car` with the lowest price in a list of `Cars`

   (h) Numbering each line of code in a list of code-lines

   (i) Writing all the lines of text in a list into a file

   (j) Determining whether any `Student` has achieved more than 75%

   (k) Concatenating all the strings in a list together into one string

   (l) Convert a list of strings into a list of `BumperStickers`

3. The `scan` variant of `fold` has the type

   ```
   ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a list
   ```

   Use this information, and your knowledge of what `fold` does, to work out what `scan` does.

4. The `reduce` variant of `fold` has the type `('a -> 'a -> 'a) -> 'a list -> 'a`. Use this information to work out the condition under which it will fail.

# Records

A record is a predefined group of named values. Each value can have a different type.

**Syntax: record definition**

```
type Name =
  { field0 : type0
  ; field1 : type1
  :
  ; fieldn : typen
  }
```

It is conventional to start type names and field names with a capital letter. The type of a record is the name of the record.

```
type Person =
  { Name : string
  ; Birth : int * int * int
  ; Weight : float
  }
```

Generic records—in other words, records which contain one or more fields which are parametrically polymorphic—can be created by specifying the generic types as part of the record definition. The type of a generic record is the name of the record, parameterised by the concrete types that replace the parametrically polymorphic variables.

**Syntax: generic record definition**

```
type Name<'a,'b,···,'z> =
  { field0 : type0
  ; field1 : 'a
  :
  ; fieldn : 'z
  }
```

```
type Datum<'a> =
  { Key : int
  ; Value : 'a
  }
```

You will rarely have to define generic records in your own code, but it is useful to know what the syntax looks like so that you have the option of doing so. After a record is defined, it can be created.

**Syntax: record creation**

```
{ Name.field0 = expr0
; field1 = expr1
:
; fieldn = exprn
}
```

All fields must be specified during record creation. However, the fields may be named in any order. Once again, a `;` character can be used to separate fields instead of a newline. One can, of course, also define generic records.

If there are different records that have the same field name(s), then F# is sometimes unable to figure out which type you are creating. To make it explicit, you can choose to prefix the first field binding with the record name, as shown in the syntax above.

```
{ Birth = 1981,8,1
; Name = "Y M M"
; Weight = 67.2
}
```

The above code will create a record of type `Person`, as defined in a previous example.

```
let v =
  { Key = 2
  ; Value = "moo"
  }
let r =
  { Key = 6
  ; Value = true
  }
```

`v` is of type `Datum<string>`, and `r` is of type `Datum<bool>`.

The value of a field within a record can be extracted by using dot-notation.

**Syntax: dot-notation**

```
value.fieldname
```

Records are immutable, so it is impossible to change them after they have been created. However, it is possible to create a new record with most of the values taken from an existing record and some of the values being different.

```
let k =
  { Birth = 1981,8,1
  ; Name = "Y M M"
  ; Weight = 67.2
  }
{ Birth = k.Birth
; Name = "X"
; Weight = k.Weight
}
```

Here we create a new `Person` record with most values taken from the existing record `k` via dot-notation, and some of the values being different.

F# provides ✪**copy-and-update** syntax to make this easier.

**SYNTAX: COPY-AND-UPDATE**

**{** v **with**
    $f_0$ = $e_0$
    $f_1$ = $e_1$
    ⋮
    $f_n$ = $e_n$
**}**

This syntax makes it easy to create a record based on an existing record: you only need to supply the fields and values that differ. The fields which are *not* specified have their values copied from the existing record.

```
let k =
  { Birth = 1981,8,1
  ; Name = "Y M M"
  ; Weight = 67.2
  }
{ k with
    Name = "X"
}
```

This example does the same thing that the previous example does, but using copy-and-update syntax.

**BEWARE THE -**

F# allows custom operators to be defined, and interprets `=-` as if it were a custom operator. This means that syntax such as `{Blah=-4}` will be misinterpreted! Make sure that you write it as `{Blah = -4}` instead.

**SYNTAX: RECORD PATTERN**

**{** $field_i$ = $pat_i$ ; ⋯ ; $field_m$ = $pat_m$ **}**

The record pattern matches each named field individually. Only the fields that you are interested in need to be named.

The following patterns will all match the record bound to `k` in the previous example.

- `{ Birth = y,m,d ; Name = s }`
- `{ Birth = 1981,v,1 }`
- `{ Weight = v ; Birth = _,8,_ }`
- `{ Birth = k ; Name = "Y M M" }`

## "AS" PATTERNS

Sometimes we want to structurally decompose a value *and* keep a reference to the whole. To do this, we can use an ✪**as-pattern**.

**SYNTAX: AS PATTERN**

pattern **as** identifier

The binds the entire value that is matched to an identifier, allowing you to structurally decompose and extract identifiers while keeping a reference to the whole. It is particularly useful when dealing with records and discriminated unions. If `pattern` succeeds, then the binding to `identifier` also succeeds.

```
let addWeight v ({ Weight = w } as r) =
  { r with
      Weight = w+v
  }
```

This function uses a record pattern to extract only the `weight` field from a record, while at the same time binding the entire record to the identifier `r` using an as-pattern. We have both the entire record and the field that we're interested in, so we can easily use a copy-and-update expression to give back a record with an altered `Weight` value.

## RECORDS, TUPLES, OR DISCRIMINATED UNIONS?

A record can be used to group data; a single-case discriminated union can also be used to group data; a tuple can also be used to group data. When should you use a particular data structure?

- Records make it much easier to use "as" patterns for updating parts of them, and are never dependent on the order of the fields inside them for pattern-matching. If you plan to use copy-and-update a lot with your data structure, or if your

grouped data has many fields, then you might want to use a record.

- If you have many data fields that have the same type, then you want to use either a record or a discriminated union with named types — you definitely don't want to use a tuple!
- Otherwise, use a discriminated union.

In general, you don't need to use records much[1]: that's why they've been kept until last. You'll note, in fact, that none of the functions on pages 59-61 use records at all.

## Comparison

If all the types in a record are comparable, then the record is comparable to another record of the same type. Records are compared item-by-item, starting from the first field in the record definition and ending with the last field in the record definition. As soon as one field's value is greater or less than the corresponding field's value in the other record, a result is generated.

[1] Contrast this with most imperative languages, where the primary function of data structures is to *group data*!

# Exercises

1. The following record describes a mythical sea-beast known as a Kraken.

```
type Kraken =
  { Victims : string list
  ; Age : int
  ; Where : float * float
  }
```

   (a) Create a Kraken that has killed two people (**Jason** and **Freddy**, with Jason being placed before Freddy), is **16384** years old, and can be found at latitude **-33.69** and longitude **26.68**.

   (b) Assume that your Kraken from (a) has been created. Now look at the following patterns; if a pattern will match your Kraken, write down **Y**; if it won't match, write down **N**; and if it is an invalid pattern, write down **WTF**. If a match is successful, also write down any identifiers and the values that they are bound to.

      i. `["Jason"; "Freddy"]`
      ii. `{ Age=a }`
      iii. `{Where=g}`
      iv. `{ Where=g,h }`
      v. `{Victims=k}`
      vi. `{Victims=j::f::k}`
      vii. `{Victims=_::[]}`
      viii. `{ Victims=f::_ }`
      ix. `{ Victims=[_; _] }`
      x. `{ Age=16384; Victims=[j; f] }`
      xi. `{ Victims=[freddy, "Freddy"] }`
      xii. `{ Age=_, Where=(p,q) }`
      xiii. `{Where=[-33.69; 26.68]}`
      xiv. `["Jason"; "Freddy"]; 16384; (lat,long)`
      xv. `Age=a, Victims=v`
      xvi. `{Age = 16384}`
      xvii. `{Age = a>10}`
      xviii. `{Victims=["Freddy"; "Jason"]}`

   (c) A record pattern in which a field is matched by the wildcard pattern, such as `{Age=_}`, is valid. However, a good programmer will not write such a pattern. Why?

   (d) The `Where` field should be interpreted as (latitude,longitude), but could also be interpreted as (longitude,latitude). Define a Coordinate record to make this unambiguous, and write down a revised Kraken record definition that makes use of it.

   (e) Create a Kraken that contains the same information as in (a), using the definitions you've created in (d).

   (f) Some of the valid patterns from (b) must be changed so that they continue to match the new definition of Kraken. Identify only the patterns that need to change and, for each identified pattern, write down a changed pattern that matches and binds the same identifiers.

2. (a) Define a record type that represents an XYZ coordinate in 3D space.

   (b) Write down a pattern which extracts *only* the Y-coordinate from a record value of the type you defined, and binds it to `guyver`.

3. Consider the following types.

```
type Glass =
| Full of percent:int
| Empty
type Customer =
  { Name : string
  ; Height : float // in meters
  ; Drink : string * Glass
  }
```

   (a) Create a `Customer` named Severus Snape. Snape is 189cm tall, and drinks Pumpkin Juice ... but his glass is empty.

   (b) Circle all the patterns which will match the `Customer` that you created.

      i. `Customer _`
      ii. `Customer`
      iii. `{Name="Severus Snape"}`
      iv. `Name=_`
      v. `{}`
      vi. `{Name=n; Height=h; Drink=d}`
      vii. `{Name=n; Height=h; Drink=(_, glass)}`
      viii. `{Height=h; Name=me}`
      ix. `{Drink='a}`
      x. `{Drink=_}`
      xi. `{Drink=(s, Empty)}`

4. A hotel sells accommodation. Every room has a room number and a status: it can be unavailable, booked, available, or occupied. If a room is unavailable, there must be a reason (such as *undergoing maintenance* or *needs to be fumigated* or ... well, anything else). A booked room is associated with the name of the person who booked it. An occupied room is occupied by a customer, and from the hotel's perspective, the only important things about a customer are their name and their credit card number.

   (a) Write data structures which describe this scenario.

(b) Create a room 11 that's been booked by Frank Talk.

(c) Create room 16 that's occupied by Banksy, who has credit card number 5555-4325-1836-0019.

(d) Create an available room 909.

(e) Write a function which, given a list of rooms, will book all the available or booked rooms on the list in the name of Lord Voldemort.

5. You must make a system to track particular animals, which have been tagged for research purposes, on a game farm. An animal has a species (which is a string), a tag number (which is an integer), and a last-seen location (expressed as a latitude and longitude, both of which are floating-point values). An animal *may* have a research note attached to it, or it may not.

   (a) Define a record type, containing **four** fields, which represents this.

   (b) Create a lion with the tag 9177, last seen at latitude 40.77, longitude -73.97, with no research note attached to it.

   (c) Write a function which will take an animal and return a new animal with a research note attached that says "Very Dancerous".

   (d) Write a pattern (*just the pattern!*) which will match an antelope and bind its tag number to the identifier `tagNo`.

6. Assume that you have defined types as follows:

```
type Op<'a> =
| Success of 'a
| Failure
type Parseable<'a> =
  {
  ; Input : string
  ; Parsed : Op<'a>
  }
```

   (a) The `Parseable` type requires a generic parameter. Why?

   (b) What are the types of the following functions? (Be careful — this is a bit trickier than it might look!)

   i.
```
let success x v =
    { x with
        Parsed = Success v
    }
```

   ii.
```
let success x v =
    { Input = x.Input
    ; Parsed = Success v
    }
```

   (c) Explain the difference in function types.

# TUTORIAL

In this tutorial, we'll be using recursion, generic types, and lists. We'll also be using almost every feature that we've covered so far, so this is a great tutorial for understanding how features in the language work together.

Recursive functions can often be used effectively with recursive types. We'll create a recursive generic binary search tree data structure and throw in a few tree operations to give it a workout. This part of the tutorial is slightly more difficult to grasp, so you might want to have a paper & pen ready to sketch out some binary trees and check for yourself that everything works. You might also want to have a quick look online, or consult some past study notes, so that you can remember how binary search trees work.

1. We can start with this definition of a binary tree:

   ```
   type Tree<'a> =
   | Empty
   | Node of 'a * Tree<'a> * Tree<'a>
   ```

   Each child of a binary tree is, of course, a binary tree itself! A tree might be empty, or it might store some data and two children.

2. We want to insert things into the tree, so let's create an `insert` function. Remember that this is going to be a search tree, so we want things to be inserted in the correct order.

   ```
   let rec insert v tree =
     match tree with
     | Empty ->
       Node (v, Empty, Empty)
     | Node (x, l, r) ->
       match v = x with
       | true ->
         tree
       | false ->
         match v < x with
         | true ->
           Node (x, insert v l, r)
         | false ->
           Node (x, l, insert v r)
   ```

   If the `tree` is empty, we just add in a new node with some empty sub-trees. When the `tree` isn't empty, we need to look at the value of the node (which extract by binding it to `x`). If we've already got this value in the tree, we don't need to store it again, so we check for equivalent values as we go. If the value we're inserting is less than `x`, then we create a new node that contains this value, and we push the old value down through the left sub-tree via a recursive call. However, if the value we're inserting is greater than `x`, then we create a new node that has this node's value, and pushes the new value down though the right sub-tree via a recursive call.

   Remember that we are *not* "updating" or "mutating" the tree in any way here! It is impossible to update or mutate when you're working with immutable values, as is the case here. Instead, we are creating new nodes as we go[1]. Note, also, the type of this function as we pass it down to F# Interactive: it can handle any kind of values, as long as they are comparable.

3. We expect the nodes to be inserted in the correct places in the tree. It would be convenient to be able to take a walk through the tree, pulling out all the values, and seeing whether a walk through the tree actually does give us the expected order. To do this, we can create a function to take a tree and give us its string representation after in-order traversal (i.e. Left-Node-Right / LNR traversal).

   Creating a string form for the tree is going to be slightly tricky because `Node` cases will have their data represented in the string, and `Empty` cases won't be in the string at all. Each piece of data should be separated by a space. We'll start from the tree root and we'll integrate strings from subtrees into our tree. However, we won't know in time whether a sub-tree is `Empty` or `Node`, so we won't be able to tell whether we should insert it, followed by a space, before we visit it. To see this problem, try playing with this function:

   ```
   let rec toStringA tree =
     match tree with
     | Empty ->
       ""
     | Node (v, l, r) ->
       $"{toStringA l} {v} {toStringA r}"
   ```

   Notice what it prints out for the tree `insert 4 Empty`. Can you see why the problem occurs?

   We'll implement three different approaches to solving the issue. Each approach will teach you some different tricks, so implement both. Our first approach is to observe that the problem occurs because we can't see which sub-node values are empty, and which aren't. But we can explicitly indicate which ones are empty by using `Option` values...

   ```
   let toStringB tree =
     let rec toStringB tree =
       match tree with
       | Empty ->
   ```

[1] This not as expensive as it sounds because modern functional languages find ways to share common data between structures.

```
      None
    | Node (v, l, r) ->
      match toStringB l, toStringB r with
      | None, None ->
        Some <| $"{v}"
      | None, Some x ->
        Some <| $"{v} {x}"
      | Some x, None ->
        Some <| $"{x} {v}"
      | Some x, Some y ->
        Some <| $"{x} {v} {y}"
  match toStringB tree with
  | None ->
    ""
  | Some s ->
    s
```

Here we explicitly represent the difference between a value that can be placed within the string and a value which doesn't have a viable string representation. As a result, we can pass back viable or non-viable values from our recursive calls, check for them, and choose to execute the appropriate code.

However, all of this presents us with a problem: we want the type of `toStringB` to be Tree -> string, but it's actually `Tree -> string option`. We'll use a little trick to fix this: create an inner function (also named `toStringB` here) which does the actual work, and then "unwrap" the results of that function and pass it out.

Push this down to F# Interactive and see play around with it using a simple tree such as

```
insert 4 Empty |> insert 5 |> insert 1
|> insert 3
```

4. Our second approach also starts by observing that the problem occurs because we can't see which sub-node values are empty, and which aren't ... or can we? After all, we only ever need to look one level deeper than we are. Patterns allow us to deconstruct values of arbitrary depth. So let's try a pattern-matching solution.

```
let rec toStringC tree =
  match tree with
  | Empty -> ""
  | Node (v, Node _ as a), (Node _ as b)) ->
    $"{toStringC a} {v} {toStringC b}"
  | Node (v, (Node _ as a), Empty) ->
    $"{toStringC a} {v}"
  | Node (v, Empty, (Node _ as a)) ->
    $"{v} {toStringC a}"
  | Node (v, Empty, Empty) ->
    $"{v}"
```

Note that this solution is shorter and arguably clearer than the `toStringB` solution. You will find that solutions that leverage pattern-matching are often shorter and clearer, so learning to think in terms of patterns and the "shape" of data is a valuable skill to learn.

5. Our third approach is to break the problem down. We want to get the nodes; we want to convert them to strings; and we want to merge them into a final string such that they have spaces between them. Let's see if we can make functions to do each of those things.

```
let rec toList = function
  | Empty ->
    []
  | Node (v, l, r) ->
    toList l @ (v :: toList r)
```

The `toList` function simply traverses the tree in LNR order, creating a list of values as it goes. Send it down to F# Interactive and play around with it, perhaps using the same simple tree that you created in (3). Trace through the code and understand exactly why it gives you the output that it does.

Then we'd like to convert each of the node values into a string. This is exceptionally easy.

```
let stringifyList =
  List.map (fun v -> $"{v}") x
```

Lastly, we want to put a space between each of the string values. There's a built-in function, described in these notes, that can help us with that: `String.concat`. So let's tie it all together with this function:

```
let toStringD x =
  toList x |> stringifyList |> String.concat " "
```

A one-line function—wow! And as a bonus, we now have a generally useful function, `toList`, that we can use to convert any tree to an ordered list. This an even better result than pattern-matching was able to give us. All of the functions are so small that errors in them should be very easy to spot. We often get this kind of excellent result when we break the problem down into steps, because each step (or a few steps) can often be encoded as a function, and functions can then be composed together.

**Type inference ☹**

The careful reader will have noted that we could have made the `stringifyList` and `toStringD` functions even shorter. For example, `stringifyList` could have been written as

```
let stringifyList = List.map (fun v -> "{v}")
```

and `toStringD` could have been written as

```
let toStringD =
  toList >> stringifyList >> String.concat " "
```

However, F# Interactive's type inference isn't good enough to figure out what these types should be, and a more complex explicit type would have to be written to make it all work. A different functional

language with better type inference — or an untyped language — would have no trouble with the shorter code.

6. For debugging purposes, we might want to print the tree's values in LNR order, while not affecting the tree itself in any way. Fortunately, this is now exceptionally easy to do.

```
let debug tree =
  printfn $"{toStringD tree}"
  tree
```

7. We can insert things. How about finding whether a particular value is in a tree?

```
let rec query v tree =
  match tree with
  | Empty ->
    false
  | Node (x, l, r) ->
    let queryMore () =
      match v < x with
      | true ->
        query v l
      | false ->
        query v r
    x = v || queryMore ()
```

   Here we use shortcut boolean evaluation to either continue querying down the tree, or stop querying and return true. The difference between function *definition* and function *application* is particularly important here: we've defined the `queryMore` function, but the code within it won't be executed until—and unless!—we apply it. This ensures that the `query` function does the least amount of work that is necessary to find a value.

   (Incidentally, we can't use the boolean evaluation trick in `queryMore`, e.g. by replacing the body with

```
(v < x && query v l) || query v r
```

   Can you see why?)

8. How about a function that allows us to merge binary trees together? This is easier than it looks: we can just take the first tree, get its values, and then push those into the second tree. However, before we begin to write code, let's think about what we already know about functional programs: they typically build larger functions out of smaller functions. Do we have a function to "take a tree and get its values"? Yes, we do: it's `toList`. Do we have a function to "push a value into a tree"? Yes, we do: it's `insert`. So we already have the building blocks for making a new function. We just have to figure out how to tie them together.

   We'll want to take the first value and insert it into the second tree, and take the result of that insertion and use it to insert the second value, and take the result and use it to insert the third value, and so on. If you know your operations well, you'll immediately recognize that this is a `fold` operation!

```
let merge a b =
  toList a |> List.fold (fun t v -> insert v t) b
```

   Once again, a simple one-line function gives us the complex operation that we want.

9. We can add things to the tree, and we can check if an item is in the tree; but we can't remove things yet. Let's make a function to do that. In fact, let's extend ourselves a little bit here: we won't make a simple `remove` function, we'll make a `filter` operation for our trees. Just like a List's `filter`, we will pass through a function that determines whether a node should be kept or not.

```
let rec filter f = function
  | Empty ->
    Empty
  | Node (x, l, r) ->
    match f x with
    | true ->
      Node (x, filter f l, filter f r)
    | false ->
      merge (filter f l) (filter f r)
```

   If the function `f` determines that we should keep this node, then we keep it and run the filter on each sub-tree. If we're not keeping the node, then we ignore the node's value and merge the left and right sub-trees of the node instead.

# Extensions

1. Can you implement a `remove` operation for a tree, which removes a particular value? There are many ways to do this. Can you find the way which is...
   (a) ...the shortest, in terms of lines of code?
   (b) ...the most efficient?
2. If we try to turn a sorted list into a binary tree, we'll end up with a very deep tree — essentially, our tree will be a glorified linked list. This is because our tree isn't *self-balancing*. How difficult would it be to convert our binary tree into a self-balancing 2-3 tree[2]?

[2] https://en.wikipedia.org/wiki/2-3_tree

# Lambda calculus

Let's take a break from the practical stuff, and delve into the theory behind functional programming. It should now be clear to you that, in F#, it is possible to do any computation that you could do in any other language. You have already learned other languages, such as C# and C and SQL, but F# can be a little bit more tricky to pick up than other languages that you've learned. Why? What makes it so *weird*?

One answer to this is found in the roots of the language. You already know that imperative languages such as C are, in a very real sense, just implementations of a Turing machine. A Turing machine manipulates symbols on a tape, following a predefined set of rules; an imperative language manipulates variables in memory, following a programmer-defined algorithm. Functional languages don't use the Turing machine model of computation; instead, they all share a common ancestor in the equally powerful, and often more expressive, ✪**lambda calculus**. For a student who has only ever encountered one way of looking at the world of programming—as if it is all about slightly different Turing machines!—this can be a mind-blowing experience.

### "How" vs "What"

In the Turing machine model, we often end up saying *how* we want to do things: declare an output array, loop through an input array, set the variable at each index of the output array, return the output array. In the lambda calculus model, we often end up saying *what* we want done: map these values using this transformation. The result is the same, but you need to change your way of looking at the problem to be able to solve it functionally!

## Basics

There are only two things in the lambda calculus, both of which are expressions:

- **Variables**. These are typically written as single letters, such as $a$ or $b$ or $z$.
- **Functions**. These are written as an input variable, a dot, and an output expression; for example, $\lambda x.x$. We say that the input variable is *bound* in the output expression.

The only operation that is permitted in the lambda calculus is left-associative function application, which is shown by separating the function from its input using a space. Function application, of course, is nothing more than $\beta$-reduction! Different functions can bind different variables which happen to have the same name. When a bound variable has the same name as a free variable, we tend to rename the bound variable to make it clear that they are different. For example,

$$\lambda x.\lambda y.(x\ y)\ a = \lambda y.(a\ y)$$

Using only variables, functions, and $\beta$-reduction, we can express any computation.

## True and false

Computer scientists have a tendency to think in binary ways: true/false, on/off, yes/no, 1/0. What good can a programming abstraction be, if we cannot define the constants "true" and "false"? It turns out that we *can* define them — as functions.

- **true** is $\lambda x.\lambda y.x$
- **false** is $\lambda x.\lambda y.y$

We can go on to define all of the usual boolean operations using functions, too.

- **and(x,y)** is $\lambda x.\lambda y.(x\ y\ (\lambda a.\lambda b.b))$
- **or(x,y)** is $\lambda x.\lambda y.(x\ (\lambda a.\lambda b.a)\ y)$
- **not(x)** is $\lambda x.(x\ (\lambda a.\lambda b.b)\ (\lambda c.\lambda d.c))$

If all of the above really does work out, then we should be able to evaluate simple boolean expressions! Let's see if we can do that. We'll evaluate the expression **and(not(false),true)**, and see if we get the desired output **true**. We begin by phrasing the expression using the lambda calculus definitions that we've got above, and renaming bound variables so that they don't conflict.

$$\begin{aligned}
&\lambda x.\lambda y.(x\ y\ (\lambda a.\lambda b.b))\\
&((\lambda t.(t\ (\lambda h.\lambda i.i)\ (\lambda c.\lambda d.c)))\ (\lambda u.\lambda v.v))\ (\lambda j.\lambda k.j)\\
=&\lambda y.(((\lambda t.(t\ (\lambda h.\lambda i.i)\ (\lambda c.\lambda d.c)))\ (\lambda u.\lambda v.v))\ y\\
&(\lambda a.\lambda b.b))\ (\lambda j.\lambda k.j)\\
=&((\lambda t.(t\ (\lambda h.\lambda i.i)\ (\lambda c.\lambda d.c)))\ (\lambda u.\lambda v.v))\ (\lambda j.\lambda k.j)\\
&(\lambda a.\lambda b.b)\\
=&(\lambda u.\lambda v.v\ (\lambda h.\lambda i.i)\ (\lambda c.\lambda d.c))\ (\lambda j.\lambda k.j)\ (\lambda a.\lambda b.b)\\
=&(\lambda v.v\ (\lambda c.\lambda d.c))\ (\lambda j.\lambda k.j)\ (\lambda a.\lambda b.b)\\
=&\lambda c.\lambda d.c\ (\lambda j.\lambda k.j)\ (\lambda a.\lambda b.b)\\
=&\lambda d.(\lambda j.\lambda k.j)\ (\lambda a.\lambda b.b)\\
=&\lambda j.\lambda k.j
\end{aligned}$$

## Practical impact

Nobody programs in lambda calculus; then again, nobody programs on a pure Turing machine, either. So, why should we learn about it? We study the basis of computation for at least two reasons:

1. By learning the underlying theory, we can understand the commonalities between different languages which implement the same paradigm. You may never have programmed in Haskell, Ocaml, or Phoenix — but because you know that they are based on the lambda calculus, you now also know that they must have first-class functions which are $\beta$-reducible. There are certain principles that are described in the introduction. By understanding lambda calculus, you also understand why these principles are a necessary part of functional programming; and you are in a much better position to understand certain features of other functional languages, such as lazy evaluation, without as much struggle.

2. Understanding the theory on a deep enough level will broaden your mind and make you capable enough to tackle larger and larger problems, and able to apply existing tools more creatively. In other words, a bit of reflection will go a long way towards making you a better programmer.

# Glossary

**anonymous field** unnamed data linked to a ✪**tag**.

**as-pattern** a ✪**pattern** that ✪**binds** to the whole of a structure which is being ✪**structurally decomposed**.

**arity** the number of data items in a tuple.

**accumulator parameter** a ✪**function** input used to store the partial result of a ✪**recursive** computation.

**application** supplying an input to a ✪**function**, thus causing the function to be evaluated.

**base case** a case that terminates a recursive function.

**binding** , *n.* the link between an ✪**identifier** and a ✪**value**.

**binding** , *v.* the action of linking an ✪**identifier** and a ✪**value**.

**call** see ✪**application**.

**case** Either:

1. a branch of a ✪**match expression**.
2. a discrete alternative within a ✪**discriminated union**.

**closure** a ✪**function** that uses identifiers from the ✪**lexical scope** that it is defined within.

**compose** given functions `'a -> 'b` and `'b -> 'c`, generate a new function `'a -> 'c`.

**composite type** a data type built up out of other data types.

**composition** see ✪**compose**.

**concrete type** see ✪**type**.

**cons** an operation which takes a ✪**value** on the left and a ✪**list** on the right, and returns a new list with the specified value as the ✪**head**.

**constructor** see ✪**tag**.

**copy-and-update expression** ✪**syntactic sugar** for creating a new ✪**record** based largely on the fields of an existing record.

**curry** to simulate a multiple-input function by consuming an input and returning a function that can consume the next input.

**definition** a part of code which sets out the structure of a construct.

**discriminated union** a data structure that can represent discrete cases.

**evaluate** Either:

1. to compute.
2. ✪**application** (if used in relation to a ✪**function**).

**expression** anything that, when ✪**evaluated**, generates a ✪**value**.

**first-class function** a ✪**function** that can be used in any context in which a value can be used.

**function** A function is anything which accepts exactly one input value and, when this is supplied, evaluates an expression to generate exactly one output value. The ✪**definition** and ✪**application** of a function are separate. Code within a function is only executed when the function is applied.

**generalise** to only restrict the ✪**type** of a ✪**function** as much as is strictly necessary.

**generic** see ✪**parametric polymorphism**.

**head** the first element of a ✪**list**.

**higher-order function** a ✪**function** which accepts or returns a function.

**identifier pattern** a pattern which matches any ✪**value** and binds it to the supplied identifier.

**inference** see ✪**type inference**.

**infix notation** to place the symbol for the operation between the operands. See also ✪**prefix notation**.

**invoke** see ✪**application**.

**label** see ✪**tag**.

**lambda calculus** Both:

1. a mathematical answer to the question "what does it mean to *compute*?"
2. the foundation of all functional programming

**lambda function** a ✪**function** without a name.

**let binding** a ✪**binding** that uses the `let` keyword.

**lexical scope** the bindings present at the point at which an entity is defined in the source code.

**list** a sequence of zero or more data items, all of which have the same type.

**match expression** a language construct that uses pattern-matching to select between alternatives.

**named field** named data linked to a ✪**tag**.

**operator** a named ✪**function** where the name is made up of non-alphanumeric characters (in F#, valid operator characters are: ! , % , & , * , + , - , . , / , < , = , > , ? , @ , ^, and | ). Curried operators use ✪**infix notation**.

**parametric polymorphism** a feature in some type systems that allows a type to be defined in relation to other types.

**pattern** a possible shape for a value.

**pattern-matching** examining a value and seeing if it matches a specified ✪**pattern**

**pipe** an operator which accepts a ✪**value** on the left and a ✪**function** on the right, and ✪**applies** the function to the value.

**pipe-to** see ✪**pipe**.

**prefix notation** to place the symbol for the operation before the operand(s). See also ✪**infix notation**.

**pure function** a ✪**function** that always generates the same result when given the same input, and has no ✪**side-effect**s.

**record** a collection of named data items.

**recursion** calling a function from within itself. See also: ✪**recursion**.

**recursive case** a case which begins or continues a recursive call.

**shadow** to re-bind an existing name within a nested scope.

**structural decomposition** examining the individual parts of a composite value.

**side-effect** anything that a function does in addition to generating a value.

**syntactic sugar** a shorter or more convenient way of writing down a longer ✪**expression**, which is exactly (or largely) equivalent to the longer expression.

**tag** the identifier that constructs a case.

**tail** all elements of a ✪**list** except for the ✪**head**.

**tail position** describes a recursive call that, once it completes, terminates the function with no further work to be done.

**tail recursion** a technique whereby the same stack frame is reused during a recursive call, thus improving the efficiency of the call.

**tuple** a collection of data items.

**type** a name that denotes a specified set of valid values.

**type annotation** a way of explicitly specifying the type of an identifier.

**type inference** proving what a type must be by looking at the context(s) of its use.

**type parameter** a way of specifying the parametrically polymorphic variable(s) of a type definition.

**type-safe** a system which restricts computation to those computations which can be performed on identified types.

**untyped** data which is not associated with a ✪**type**.

**unwrap** to retrieve ✪**wrap**ped data.

**value** an ✪**expression** that has a ✪**type** and, when evaluated, remains the same.

**variable pattern** see ✪**identifier pattern**.

**wrap** to store data in a discriminated union which serves only to add a layer of meaning to the stored data.

**wrapper type** a type which ✪**wrap**s.

**wildcard pattern** a pattern which matches any value and does not bind it to any identifier.

# Answers

## Page 7

1. It means that the value that the variable is linked to cannot be modified/updated/changed.

2. Expressions generate values. Statements do not.

3. A function is a value which accepts only one input and, when this is supplied, evaluates an expression to generate exactly one output.

4. A pure function has no side-effects, and will always generate the same output when given the same input.

5. (a) Yes. It will always generate the same result when given the same input, and it has no side-effects.

   (b) No. It has the side-effect of writing something to the screen.

   (c) No. It has the side-effect of modifying its parameter. The fact that modification of the parameter *might not* occur is irrelevant; the potential for it to occur is present.

   (d) Yes. It will always generate the same result when given the same input. Although it modifies `g` and `v` during execution, this does not count as a side-effect because it is not externally observable, nor does it modify anything outside the function. Remember that `int`s are value-types, and changes made are therefore local to the method.

   (e) Yes. It will always generate the same result when given the same input, and it has no side-effects.

6. A variable can be replaced by the value it holds at any point, without changing the meaning of the program. (or: a name, once linked to a value, will always have that value.)

7. A first-class function can be used in any context that a value can be used.

8. Referential transparency is about the unchangeability of the link between a name and a value; immutability is about the unchangeability of values.

9. Side effect.

## Page 15

1. Any 3 of:

   - Assignment can't fail, pattern-matching can.
   - Pattern-matching doesn't necessarily bind any identifiers.
   - Assignment makes a copy; binding doesn't.
   - Re-bound identifiers are shadowed, not updated.

2. `a = b`

3. (a)
```
let a =
  "hi"
"zoot"
```
   (b)
```
let b =
  "zoo"
"zooty" + b
```
   (c)
```
let c =
  7.5
let d =
  34
c + 0.2
```
   (d)
```
let e =
  8
let f =
  e
let e =
  20
let g =
  3
e+f+g
```

4. Pattern-matching determines whether a particular pattern and a particular value are compatible or not. Binding links an identifier with a value.

5. (a) No. The only way to bind to an identifier is via a pattern, and `_` is the wildcard pattern which does not bind any identifier. Therefore, `_` cannot be a valid identifier name.

   (b) Yes.

   (c) Yes. A `'` in a name is legitimate.

   (d) No. This is a string.

   (e) Yes. Any sequence of characters within double-backticks is a valid identifier.

   (f) No. An identifier cannot start with an `@` character — unlike in C#.

   (g) No. An identifier cannot be a reserved word in the language.

   (h) Yes. Any sequence of characters within double-backticks is a valid identifier.

6. A shadowed identifier continues to exist and remains linked to the same value that it was initially bound to. It cannot be accessed using the shadowed name within the scope of the shadowing. The value linked to an updated or mutated symbol is overwritten and the old value is lost.

7. Identifiers are bound to *values*, not to expressions. This means that `r.Next()` is evaluated once and the resulting value is bound to `v1`. On the last line, `v1` is evaluated, and the resulting value is bound to `v2`. This why the values are always the same.

8. (a) C
   (b) I
   (c) O
   (d) C
   (e) C
   (f) W
   (g) C
   (h) O
   (i) I
   (j) I
   (k) C
   (l) I
   (m) O
   (n) C

9. (a) ✓. Or pattern.
   (b) ✓. Or pattern.
   (c) ✗. The type expected by one of the alternatives in the or-pattern is not the same as the type expected by the other two alternatives.
   (d) ✗. Both sides do not bind the same set of identifiers.
   (e) ✗. A "-" sign in the middle of an identifier is not allowed.
   (f) ✓. Identifier pattern.
   (g) ✗. A reserved word cannot be used as an identifier.
   (h) ✗. An identifier cannot start with an apostrophe.
   (i) ✗. Both sides do not bind the same set of variables.
   (j) ✓. Identifier pattern.
   (k) ✓. Identifier pattern. (look closely – it's *two* underscores, not one!)

# Page 23

1. $\beta$-reduction is the process of substituting a function input with a value, resulting in a simpler expression.

2. (a) `"hunter2 killer"`
   (b) `(fun r -> r > 16)`
   (c) `2`
   (d) `fun () -> 12`
   (e) `7.5`
   (f) `7`
   (g) `19`
   (h) `fun h -> "I see " + h`
   (i) `fun h -> h "see dog"`

3. (a) `string -> string`
   (b) `int -> int`
   (c) `int -> int -> int`
   (d) `int -> int -> int`
   (e) `int -> float`
   (f) `int -> unit -> int`
   (g) `int -> (unit -> int) -> int`

4. - `let k = fun m t -> t*3`
   - `let k m t = t*3`

5. (a) Using a two-input operator before the operands instead of between the operands, by enclosing it in brackets.
   (b) Currying is the technique of simulating a multi-parameter function by sequentially returning different single-parameter functions.
   (c) A lambda function is a function that has not been bound to a name.

6. (a) Closure
   (b) Higher-order function
   (c) Curried functions

7. (a)
```
let log s f =
  printfn $"{s}"
  f ()
```
   (b)
```
log "Flying now"
  (fun () -> fly 300 12.4)
```
   (c) The only way in which the `log` function can affect the computation is by altering the output of the function that it is passed. However, the only place that `log` can obtain an `'a` value is via the function; and `log` returns an `'a` value as well. There is no modification operation which can operate on a parametrically polymorphic type, and therefore the `log` function has no choice but to pass the output of the passed function onwards unchanged.

8. (a) curried, higher-order, closure
   (b) curried, higher-order, closure
   (c) (intentionally blank)
   (d) lambda, higher-order
   (e) lambda
   (f) curried, higher-order, lambda
   (g) curried, higher-order, lambda

(h) curried, higher-order, lambda, closure

(i) curried, higher-order

(j) higher-order

(k) closure

9. (a) `a`

(b) unnamed

(c) unnamed

(d) `v`

(e) unnamed

## Page 29

1. (a) A match expression never compares values; it can only match patterns. The `x` is evaluated and results in the value `5`. We attempt to match this value to the pattern `y`; this succeeds and `5` is bound to `y`, shadowing the earlier binding of `y`. The corresponding expression `y+1` is executed, resulting in the value `6`. This value is the result of the match expression, and is bound to `z`.

(b) Code which binds an identifier, but doesn't use it, is more difficult to understand. It forces a programmer to use up some mental space to remember the binding. A wildcard pattern succeeds but doesn't bind anything, so there is no additional binding for the programmer to remember.

(c)
```
match x = y with
| true ->
  y+1
| _ ->
  12
```

2. (a)-(b)
```
let "hazmat" =
  match 901 with
  | 201 ->
    "wut"
  | kray ->
    "lol"
  | 901 ->
    "hazmat"
```

(c) `"lol"`

3. (a) `1.2`

(b) `0.4`

(c) No. F# does not allow the mixing of different types, nor does it implicitly cast one type into another.

(d) No. An `int` constant pattern cannot be used to match against a `float` value.

4. (a) `"evens"`

(b) `1`, `3`, or `5`

(c) A `MatchFailureException` would occur.

5. (a) `"Codd"`

(b) `int -> string`

(c) The function has no name: it is a lambda function.

## Page 35

1. An accumulator solves the problem of having any relevant information, which is still useful for computing with, remaining in the previous stack frame and being lost in a tail recursive call.

2. The recursive function would continue to recurse forever, or (if it is not tail recursive) until it runs out of stack space or reaches the maximum function call depth.

3. This is a recursive call after which there is no work that remains to be done in the calling function.

4. (a)
```
let numOdd start ``end`` =
    let rec count n =
      match n >= ``end`` with
      | true ->
        0
      | _ ->
        match n % 2 with
        | 1 ->
          1 + count (n+1)
        | _ ->
          count (n+1)
    count start
```

(b)
```
let nextMul start n =
    let rec count v =
      match v % n with
      | 0 ->
        v
      | _ ->
        count (v+1)
    count start
```

(c)
```
let isPrime n =
    let rec check v =
      match v > 1 with
      | false ->
        true
      | _ ->
        match n % v with
        | 0 ->
          false
        | _ ->
          check (v-1)
    check (n-1)
```

# Page 43

1. (a) `_,t`
   (b) `_,a,(_,(_,b))`
   (c) Yes.
   (d) `_,3` and `_`
   (e) `_`, `9.0,_,_` and `b,_,(a,_)`

2. `3,13`

3. (a) `bool * string -> int -> int`
   (b) `bool -> string * int -> int`

4. The unpacking of components of a data structure via pattern-matching of the data structure's structure, with the option to bind to selected values after a successful pattern-match.

5. (a) `Circle 100`
   (b)
   ```
   type Shape =
   | Circle of int
   | Rectangle of length:int * width:int
   | Blob
   ```
   (c) `Rectangle (5, 8)`
   (d) `Rectangle (length = 5, width = 8)` or `Rectangle (width = 8, length = 5)`
   (e) `Blob`
   (f) i. N
   ii. Y
   iii. Y
   iv. N
   v. Y. `height` is `5`, and `size` is `8`.
   vi. Y. `k` is `8`.
   vii. WTF
   viii. Y. `w` is 8, and `h` is `5`.
   ix. WTF
   x. WTF
   xi. WTF
   xii. WTF
   xiii. Y
   xiv. WTF
   xv. WTF
   xvi. Y
   xvii. Y. `b` is `8`.
   xviii. N

6. Yes. Either way would result in exactly the same value being created.

7.
   ```
   type Seat =
   | Unsold
   | Sold of string
   | VIP of int
   ```

8. (a) `Full 50`
   (b) `Full _`, `_` and `Full empty`
   (c) `Customer ("Severus Snape", ("Pumpkin Juice", Empty))` (you could also do this using named fields)
   (d) i. Y
   ii. WTF
   iii. WTF
   iv. Y
   v. Y
   vi. Y. `n` is bound to `"Severus Snape"`, and `d` is bound to `"Pumpkin Juice", Empty`.
   vii. Y. `n` is bound to `"Pumpkin Juice"`, and `glass` is bound to `Empty`.
   viii. WTF
   ix. Y. `me` is bound to `"Pumpkin Juice", Empty`.
   x. WTF
   xi. WTF

9.
   ```
   type Variety =
   | Ceylon
   | Rooibos
   | EarlGrey
   type Strength =
   | Weak
   | OK
   | Strong
   type Heat =
   | Hot
   | Iced
   type Spoons = Sugar of int
   type Drink =
   | Tea of Variety * Heat
   | Coffee of Strength * Spoons * Heat
   | Juice
   ```

# Page 53

1. Type inference is the process of determining the type of an identifier based on the context and content of surrounding code.

2. (a) `string -> string`
   (b) `int -> int`
   (c) `int -> int -> int`
   (d) `'a -> string`
   (e) `int -> int -> int`
   (f) `int -> float`
   (g) `int -> unit -> int`
   (h) `(string -> 'a) -> 'a`
   (i) `int -> (unit -> int) -> int`

3. (a) `'a * 'b * 'c -> 'a * 'c * 'b * 'a`
   (b) `(('a * int) * ('a * int) -> 'a * int) -> 'a * int -> 'a * int`

4. `Transform (_, "int")`

5. 
```
type Option<'a> =
| Some of 'a
| None
```

6. `defaultArg` takes an `'a option` and a `'a` value. If the `'a option` is `None`, then the provided `'a` value is returned; otherwise, the value wrapped in the `'a option` is returned.

7. 
```
let bind a b =
  match b with
  | Some x ->
    a x
  | None ->
    None
```

8. (a) Nothing

   (b) 
```
Next(4.22,
     Next(1.9,
       Next(0.45,Nothing)))
```

   (c) 
```
let rec map f x =
    match x with
    | Nothing ->
      Nothing
    | Next(v,next) ->
      Next (f v, map f next)
```

9. It allows the programmer to pass back some data that can help to diagnose or explain the error that has occurred.

10. (a) `'a -> 'b -> 'c -> string`

    (b) `float -> float -> float`

    (c) `bool -> 'a -> string`

    (d) `'a -> 'a`

    (e) `(int -> float) -> int -> string`

    (f) `(string -> bool) -> float -> bool`

    (g) `('a -> 'b) -> ('c -> 'a) -> 'c -> 'b`

    (h) `'a -> ('a -> 'b) -> 'b`

    (i) `'a -> ('a -> string -> 'b) -> 'b`

    (j) `char -> (int -> int)`

    (k) `(unit -> unit) -> unit`

    (l) `bool -> (int -> int) -> ('a -> string)`

    (m) `'a -> string`

    (n) `(string -> 'a) -> 'a`

# Page 57

1. (a) `[1;2;2;8]`

   (b) `[2;1;2;2;8;1;2;2;8]`

   (c) 
```
let beef =
    1::crow::crow::8::[]
```

2. (a) 7

   (b) 5

   (c) 6

   (d) 0

3. `k` is `0` and `m` is `[]`.

4. (a) `[4; 2]`

   (b) `[14; 12]`

   (c) A `MatchFailureException` is thrown; the pattern `a::b::c::r` does not match the input `[14; 12]`.

5. (a) `[8; 2; 3]`

   (b) `[4; 25]`

6. `[[1.0];[];[6.0];[8.0;3.5]]`

7. (a) `char list -> char list`

   (b) `(('a list * 'a list) -> 'a list) -> 'a list -> 'a list`

   (c) `string -> string list`

   (d) `int -> (int -> float) -> float`

8. (a) Nothing bound

   (b) `hi` is `5.3`

   (c) No match

   (d) `this` is `5.3`; `is` is `"hi"`; `a` is `9`; `bad` is `9`; `test` is `[]`.

   (e) No match. (the pattern is invalid!)

   (f) `so` is `5.3`; `said` is `9`; `i` is `9`.

   (g) No match

9. (a) `_,k::_,_,_,_`

   (b) `(_,six),_,_,_,e`

   (c) `((q,_),_),[w,_;_,e],_,(r,_),_`

10. (a) `int -> int list -> int list`

    (b) `float * 'a * float -> float * float`

    (c) `('a * string) list -> string option`

    (d) `float * 'a * float -> unit -> float * float`

    (e) `float * float -> float * float`

    (f) `'a * 'b -> 'b * 'a`

    (g) `'a * 'b -> 'a * 'b`

11. `Some n, [_;_], _` and `_, _::_::_, k` and `_`

# Page 63

1. `choose` is only interested in the successful case (`OK`, in the case of the `Either<'a,'b>` type), so it has no need for any diagnostic information that would be passed back with the diagnostic case. Therefore, the `Option` type would be a semantically superior choice for this operation.

2. (a) fold

   (b) filter

(c) map
(d) exists
(e) init
(f) partition
(g) fold
(h) mapi
(i) iter
(j) exists
(k) fold
(l) map

3. Like `scan`, but returns a list of the intermediate outputs as well.

4. It will fail when the list is empty. In this case, there is no possible way that the function can generate a `'a` value.

# Page 69

1. (a)
```
{ Victims = ["Jason";"Freddy"]
; Age = 16384
; Where = -33.69, 26.68
}
```
(b)
i. N
ii. Y. `a` is `16384`.
iii. Y. `g` is `-33.69,26.68`
iv. Y. `g` is `-33.69`, and `h` is `26.68`.
v. Y. `k` is `["Jason";"Freddy"]`.
vi. Y. `j` is `"Jason"`, and `f` is `"Freddy"`, and `k` is `[]`.
vii. N
viii. Y. `f` is `"Jason"`.
ix. Y
x. Y. `j` is `"Jason"`, and `f` is `"Freddy"`.
xi. Y. `freddy` is `"Jason"`.
xii. N
xiii. N
xiv. WTF
xv. WTF
xvi. Y
xvii. WTF
xviii. N

(c) A record pattern only needs to specify the fields which must match and/or the fields to which one wishes to bind identifiers; the other fields are not considered during the matching process. A wildcard pattern always succeeds, but never binds an identifier. Therefore, specifying a wildcard pattern for a field is always unnecessary; it has the same effect as not specifying that field at all.

(d)
```
type Coordinate =
  { Lat : float
  ; Long : float
  }
type Kraken =
  { Victims : string list
  ; Age : int
  ; Where : Coordinate
  }
```
(e)
```
{ Victims = ["Jason";"Freddy"]
; Age = 16384
; Where =
  { Lat = -33.69
  ; Long = 26.68
  }
}
```
(f) iv. `{ Where = { Lat = g; Long = h } }`

2. (a)
```
type XYZ =
  { X : int
  ; Y : int
  ; Z : int
  }
```
(b) `{ Y = guyver }`

3. (a)
```
{ Name = "Severus Snape"
; Height = 1.89
; Drink = "Pumpkin juice", Empty
}
```
(b)
`{Name="Severus Snape"}`
`{Name=_}`
`{Name=n; Height=h; Drink=d}`
`{Name=n; Height=h; Drink=(_, glass)}`
`{Height=h; Name=me}`
`{Drink=_}`
`{Drink=(s,Empty)}`

4. (a)
```
type Customer =
  { Name : string
  ; CCN : string
  }
type Status =
| Unavailable of string
| Booked of string
| Available
| Occupied of Customer
type Room =
  { Number : int
  ; Status : Status
  }
```
(b)
```
{ Number = 11
; Status = Booked "Frank Talk"
}
```
(c)
```
{ Number = 16
; Status =
  Occupied
    { Name = "Banksy"
    ; CCN = "5555-4325-1836-0019"
    }
```

```
}
```

(d)
```
{ Number = 909
; Status = Available
}
```

(e)
```
List.map (fun x ->
  match x.Status with
  | Available | Booked _ ->
    { x with
        Status = Booked "Lord Voldemort"
    }
  | _ -> x
)
```

5. (a)
```
type Animal =
  { Species : string
  ; TagNo : int
  ; LastSeen : float * float
  ; Note : string option
  }
```

(b)
```
{ Species = "Lion"
; TagNo = 9177
; LastSeen = 40.77, -73.97
; Note=None
}
```

(c)
```
let f a =
  { a with
      Note = Some "Very Dancerous"
  }
```

(d)
```
{ Species = "Antelope"
; TagNo = tagNo
}
```

6. (a) The `Op` type requires a type parameter, and cannot be included without one. Consequently, any type that embeds an `Op` type must either be defined with such a type parameter that it then passes to `Op`, or it must pass a concrete type to `Op` as its type parameter.

(b) i. `Parseable<'a> -> Parseable<'a>`

ii. `Parseable<'a> -> Parseable<'b>`

(c) The first function uses a copy-and-update expression. Such an expression will *always* return the same type that it is given. Newly generated records that do not use copy-and-update are not subject to the same restriction.

# Practicals

Read through *all* the practicals before your first practical session. You *do not* need to wait for the "right" practical to do any of these practicals: feel free to finish all your practical work as quickly as you can! You do, however, need to finish particular practicals by the time that they must be handed in. For this reason, it is suggested that you complete the practicals in-order.

## Practical 1

The goal of this practical is to give you some experience with the basic constructs of F#. Work on Practical 1 and Practical 2 together: when you feel that you have learned enough from Practical 1 to do a question in Practical 2, then put your skills to use there. You should be able to learn the material much more quickly and efficiently this way.

### Setup

1. If you are using Visual Studio Code, install the Visual Studio Code **.Net Core Test Explorer** extension:
   (a) Click on the "Extensions" icon on the left of the IDE, or press Ctrl + Shift + X.
   (b) Search for **.Net Core Test Explorer** using the search-box at the top of the left panel. Install it.
2. Download and unzip the FSharpKoans zipfile from RUConnected.
3. Open up Visual Studio Code and use **File** → **Open Folder...** to open up the folder where you unzipped FSharpKoans.

### Instructions

Complete all the tests, starting from the `AboutTesting.fs` file and continuing on from there. You do not need to complete the tests in `AboutYou.fs`, though it would be nice.

There are two ways to run the tests:

1. Via the command-line. From the command-line, navigate to the folder that contains the `FSharpKoans.fsproj` file. Type `dotnet test` to run all the tests. If you only want to run only the tests in a particular module, use the `--filter` syntax, e.g.:

   ```
   dotnet test --filter "FullyQualifiedName~06:"
   ```

   ...will run only the tests in the module named "06: Tuples".
2. Via the integrated test runner. You can use the "Test" panel on the left side of VS Code to run all of the tests. Unfortunately, that panel won't allow you to double-click on a test and go directly to it, as of the time of this writing, and it will not allow you to see the exact error that causes a test to fail. This can make for a very frustrating experience, so the command-line is often a better choice.

### Assessment

A partial hand-in will occur at the start of your second practical session. This hand-in will count for 40% of your mark. Upload it to RUConnected.

The second hand-in will occur at the start of your fourth practical session. This hand-in will count for 60% of your mark. Upload it to RUConnected.

## Practical 2

This practical will prepare you for Part 2 of the examination, where you will be asked to solve relatively trivial problems.

### Setup

Download the Synthesis zipfile from RUConnected.

You will work in the Library.fs file. Unit tests are found in the UnitTests.fs file.

### Instructions

Solve the numbered problems in this section. You may not use any built-in functions (other than `failwith`) or any data structures (other than tuples) for this practical.

- Use the error messages of the tests to figure out which test is failing. Pay attention to the **Source** line and to the last line of the stack trace.
- Use F# Interactive to test your functions.
- Types may look a bit "strange" and types such as `'a` may appear. Dont worry about this, we'll cover it soon enough (pages 49-52).
- Dont try to skip ahead the earlier problems are easier than the later problems.
- You may discuss problems and solutions with others, or with your lecturer, and you may post your solution(s) on Slack or WhatsApp for discussion and comment. Your solutions, however,

must be your own — and it is sometimes *very* difficult to think of your own solution *after* you've seen someone else's solution. Be responsible!

- **In later problems, you can use functions that you've created in earlier problems**. Use this fact to save yourself some time and effort, if you can see how to do it.

Now, on to the problems!

1. Create a function `abelar` to return true if the input is greater than 12, and less than 3097, and is a multiple of 12. [target: 1 line]
2. Create a function `area` which, given a base and height, finds the area of a triangle. If either the base or the height is negative, throw an exception. [target: 5 lines]
3. Create a function `zollo` which returns a positive number if given a negative number, or doubles the number if the input is positive. [target: 5 lines]
4. Create a function `min` which chooses the smaller of two values. [target: 5 lines]
5. Create a function `max` which chooses the larger of two values. [target: 5 lines]
6. Create a function `ofTime` to convert hours, minutes, and seconds to a number of seconds. [target: 1 line]
7. Create a function `toTime` to convert a number of seconds to hours, minutes, and seconds. [target: 12 lines]
8. Create a function `digits` to count the number of digits in a number. The input may be positive or negative. [target: 9 lines]
9. Create a function `minmax` which finds the largest and smallest values out of four values that are provided. [target: 5 lines]
10. Create a function `isLeap` which returns true if the given year is a leap year. Every year that is divisible by 4 is a leap year, unless it is also divisible by 100. However, if it is also divisible by 400, then it is still a leap year. The function should throw an exception of the input year is less than 1582. [target: 9 lines]
11. Create a function `month` which accepts an integer between 1 and 12 inclusive, and returns the corresponding month and the number of days in that month, assuming that it is not a leap year. If an integer less than 1 or greater than 12 is supplied, an exception should be thrown. [target: 25 lines]
12. Create a function `toBinary` which converts a positive integer to a binary string. Throw an exception if a negative integer is supplied. [target: 19 lines]
13. Create a function `bizFuzz` to accept an integer `n` and return the number of times a number between 1 and `n` inclusive is divisible by 3, divisible by 5, and divisible by both 3 and 5. [target: 15 lines]
14. Create a function `monthDay` which accepts an integer `d` and a year `y`, and returns a string for the month that the day `d` falls within. The function must accept a range of `d` from 1 to 365 if `y` isn't a leap year, and must accept `d` between 1 and 366 if `y` is a leap year. If `d` is out of range, or if `y` is less than 1582, then an exception must be thrown. Remember that:

    - April, June, September, and November have 30 days.
    - January, March, May, July, August, October, and December have 31 days.
    - February has 29 days in a leap year, and 28 days otherwise.

    [target: 28 lines]
15. Create a function `coord` which is given a Cartesian coordinate and returns functions to calculate:

    - the straight-line distance to another Cartesian coordinate, as calculated by

      $$dist = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

      *Hint*: you developed a square-root function in a tutorial on page 37 of your textbook.
    - whether a rectangle (described by top-left coordinate, width, and height, in that order) will contain the initial coordinate.

    [target: 9 lines, excluding `sqrt` function]

## Assessment

Each question is worth 7 marks. Five of those marks are for correctness. One mark is for a correct solution that is formatted correctly (e.g. no run-on lines, no using semicolons to artificially break lines, correct whitespacing) according to the style laid out in the textbook. One mark is for a correct solution with a function body that is $\leq$ the target number of lines which is indicated on the problem.

This practical is due by the start of your third practical session. Upload it to RUConnected.

# Practical 3

For this practical you will work in your assigned groups and implement an ancient two-player strategy game, still popular today, called Oware. It will teach you about how to design and architect a functional solution for a medium-sized, relatively straightforward

problem. You should work on this practical and Practical 4 together, since they will test different parts of your skill-set.

## Setup

- There are many variations of Oware; use the rules that you find at http://bit.ly/OwareRules, but note that the usual name for the places is "house" and not "hole". To understand what it means to sow your seeds counterclockwise, see how the board is set up in the online game at http://bit.ly/OwareOnline.
- For standardisation between implementations, we will say that south's leftmost house is house 1; north's leftmost house is house 7; and north's rightmost house is house 12.
- Using the GitHub account that would come first in the dictionary, fork and clone https://github.com/cynic/oware.git
- Keep a remote pointed to the original repository (as usual), and invite the other group members as collaborators.

## Instructions

Your game should make use of discriminated unions, records, and tuples — in fact, it will be quite difficult to build without these, so youll probably find them slipping in naturally. Your game must pass certain tests, provided with the skeleton, which will use certain functions. The mandatory functions are:

**start** which accepts a `StartingPosition` and returns an initialized game where the person in the `StartingPosition` starts the game.

**getSeeds** which accepts a house number and a *board*, and returns the number of seeds in the specified house.

**score** which accepts a *board* and gives back a tuple of *(southScore , northScore)*

**gameState** which accepts a *board* and gives back a string that tells us about the state of the game. Valid strings are: "South's turn", "North's turn", "Game ended in a draw", "South won", and "North won".

**useHouse** which accepts a house number and a *board*, and makes a move using that house.

The *board* referred to above is a data structure, designed by you, to represent the progress of a game. The unit tests included in the skeleton solution will help you to understand all the requirements.

You may include only two impure functions in your code, one for handling input and one for printing output. The maximum length of each of these functions is 15 lines. Any impure functionality in these functions which is not related to handling input or printing output will result in a mark of zero for the entire practical.

## Assessment

This practical is due by the start of your fifth practical session. Your mark will depend on the number of tests that pass. Upload it to RUConnected.

# Practical 4

The problems in this practical are slightly modified from the finals of the International Collegiate Programming Competition (ICPC). These problems will prepare you for the difficulty of Part 3 of your examination. They will also test your ability to design and implement functional-style algorithms.

## Setup

- You may discuss the problems with your classmates, but your solutions must be your own.
- Download and unzip the ICPC zipfile from RUConnected.

## Instructions

*Hint*: the Useful functions chapter could be really useful here. Pay particular attention to the usefulness of List operations...

### Problem B, 2018: Comma Sprinkler

As practice will tell you, the English rules for comma placement are complex, frustrating, and often ambiguous. Many people, even the English, will, in practice, ignore them, and, apply custom rules, or, no rules, at all.

Doctor Comma Sprinkler solved this issue by developing a set of rules that sprinkles commas in a sentence with no ambiguity and little simplicity. In this problem you will help Dr. Sprinkler by producing an algorithm to automatically apply her rules.

Dr. Sprinklers rules for adding commas to an existing piece of text are as follows:

1. If a word anywhere in the text is preceded by a comma, find all occurrences of that word in the text, and put a comma before each of those occurrences, except in the case where such an occurrence is the first word of a sentence or already preceded by a comma.
2. If a word anywhere in the text is succeeded by a comma, find all occurrences of that word in the

text, and put a comma after each of those occurrences, except in the case where such an occurrence is the last word of a sentence or already succeeded by a comma.

3. Apply rules repeatedly until no new commas can be added using either of them.

As an example, consider the text:

> please sit spot. sit spot, sit. spot here now here.

How would we handle this?

- Because there is a comma after "spot" in the second sentence, a comma should be added after "spot" in the third sentence as well (but not the first sentence, since it is the last word of that sentence). The sentence is now:

  > please sit spot. sit spot, sit. spot**,** here now here.

- Also, because there is a comma before the word "sit" in the second sentence, one should be added before that word in the first sentence (but no comma is added before the "sit" beginning the second sentence because it is the first word of that sentence). The sentence is now:

  > please**,** sit spot. sit spot, sit. spot, here now here.

- Finally, notice that once a comma is added after "spot" in the third sentence, there exists a comma before the first occurrence of "here". Therefore, a comma is also added before the other occurrence of "here". There are no more commas to be added so the final result is:

  > please, sit spot. sit spot, sit. spot, here now**,** here.

A word is a maximal sequence of letters within the text. The input string should meet all of the following criteria:

- It should contain at least 2 characters.
- Each character must be either a lowercase letter, a comma, a period, or a space.
- The string must begin with a word.
- Between every two words in the text, there is either a single space, a comma followed by a space, or a period followed by a space (denoting the end of a sentence and the beginning of a new one).
- The last word of the text is followed by a period with no trailing space.

Return the modified text. If text does not conform to the above, then a `None` value should be returned.

## Problem F, 2018: Go with the Flow

In typesetting, a "river" is a string of spaces formed by gaps between words that extends down several lines of text. For instance, Figure 1 shows several "rivers" which have been highlighted in pen.

Celebrated river authority Flo Ng wants her new book on rivers of the world to include the longest typographic rivers possible. She plans to set the text in a monospaced font (all letters and spaces have equal width) in a left-aligned column of some fixed width, with exactly one space separating words on each line (the text is not aligned on the right). For Flo, a river is defined as a sequence of spaces lying in consecutive lines in which the position of each space in the sequence (except the first) differs by at most 1 from the position of the space in the line above it. Trailing white space cannot appear in a river.

Words must be packed as tightly as possible on lines; no words may be split across lines. The line width used must be at least as long as the longest word in the text. For instance, Figure 2 shows the same text set with two different line widths.

Given a string, you have been tasked with determining the line width that produces the longest river of spaces for that text. Valid strings meet all of the following criteria:

- Each word consists only of lowercase and uppercase letters.
- Words are separated by a single space.
- No word exceeds 80 characters.
- There are at least two words in a string.

Return both the line width for which the input string contains the longest possible river, and the length of this longest river. If more than one line width yields this maximum, return the shortest such line width.

If a string is invalid, return a `None` value.

## Assessment

This practical is due by the end of the sixth practical session. Upload it to RUConnected.

Each problem is worth 6 marks. Five of those marks are for correctness: you must pass all the tests for a problem before it is considered to be correct. One mark is for a correct solution that is formatted well

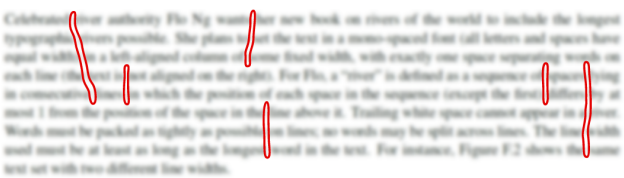

Figure 1: Example of "rivers" in typesetting

(e.g. no run-on lines, no using semicolons to artificially break lines, correct whitespacing) according to the conventions seen in the textbook.

| Line width 14: River of length 4 | Line width 15: River of length 5 |
|---|---|
| The Yangtze is\| | The Yangtze is \| |
| the third \| | the third \| |
| longest river \| | longest*river \| |
| in*Asia and \| | in Asia*and the\| |
| the*longest in\| | longest*in the \| |
| the*world to \| | world to*flow \| |
| flow*entirely \| | entirely*in one\| |
| in one country\| | country \| |

Figure 2: Different rivers are created in the same text when you use different line widths