* 1. 

Hands-On Lab

Introduction to F#

* 1. Lab version: 1.0.0
  2. Last updated: 1/4/2011
  3. 

Contents

[Overview 3](#_Toc281930824)

[Exercise 1: Types in F# 4](#_Toc281930825)

[Task 1 – Observing Type Inference in F# 4](#_Toc281930826)

[Task 2 – Working with Tuples 6](#_Toc281930827)

[Task 3 – Functions as Values 7](#_Toc281930828)

[Exercise 2: Using the “Let” Keyword in F# 10](#_Toc281930829)

[Task 1 – Using the Let Keyword to Bind a Value to an Identifier 10](#_Toc281930830)

[Task 2 – Working with tuples and identifiers 12](#_Toc281930831)

[Exercise 3: Functions 16](#_Toc281930832)

[Task 1 – Binding Functions to Identifiers 16](#_Toc281930833)

[Task 2 – Binding Functions with Multiple Parameters 17](#_Toc281930834)

[Task 3 – Working with partially-applied functions 19](#_Toc281930835)

[Exercise 4: Lists 21](#_Toc281930836)

[Task 1 – Creating Lists 21](#_Toc281930837)

[Exercise 5: Pattern Matching and Recursion 24](#_Toc281930838)

[Task 1 – Using Simple Pattern Matching to Output Information 25](#_Toc281930839)

[Task 2 – Using Simple Pattern Matching and Recursion to Implement the Fibonacci Sequence 26](#_Toc281930840)

[Task 3 – Using Recursion with F# Lists 27](#_Toc281930841)

[Exercise 6: Types and Discriminated Unions 28](#_Toc281930842)

[Task 1 – Creating a Simple Type in F# 29](#_Toc281930843)

[Task 2 – Creating a Discriminated Union 30](#_Toc281930844)

[Task 3 – Using Pattern Matching in Discriminated Unions 31](#_Toc281930845)

[Summary 32](#_Toc281930846)

Overview

* 1. F# is a new functional programming language from Microsoft. While it is primarily a functional programming language, it is known as a multi-paradigm language because it also supports object-oriented programming. F#’s tight integration with the entire .NET system of libraries and development environment enables developers to bring the power of Functional Programming to the .NET platform.

# Objectives

* 1. In this Hands-On Lab, you will learn how to:
  + Use F# Types
  + Leverage the **let** keyword to bind values to identifiers
  + Use functions in F#
  + Use lists in F#
  + Leverage Pattern Matching and Recursion
  + Create your own types and consume discriminated unions

# System Requirements

* 1. You must have the following items to complete this lab:
  + Microsoft Visual Studio 2010

# Setup

* 1. All the requisites for this lab are verified using the **Configuration Wizard**. To make sure that everything is correctly configured, follow these steps.
  2. Run the **Configuration Wizard** for the Training Kit if you have not done it previously. To do this, browse to **Source\Setup** folder of this lab, and double-click the **Dependencies.dep** file. Install any pre-requisites that are missing (rescanning if necessary) and complete the wizard.
     1. **Note:** The Configuration Wizard is used for checking dependencies and setting up the environment. If the Configuration Wizard is not installed on your machine, you must install it running the DependencyChecker.msi file located on the %VS2010TKInstallationFolder%\Assets\DependencyChecker folder (e.g. by default the Training Kit is installed under C:\VS2010TrainingKit).
     2. For convenience, much of the code you will be managing along this lab is available as Visual Studio code snippets. The **Dependencies.dep** file launches the Visual Studio installer file that installs the code snippets.

# Exercises

* 1. This Hands-On Lab is comprised of the following exercises:
  + Examine the basic F# types including tuples and functions
  + Discover how the **let** keyword allows values to be bound to identifiers
  + See that in F# functions are the same as any other value and are handled the same way. Demonstrate how this allows advanced language features such as partially-applied functions (also known as “curried” functions).
  + Discover how F# lists are built and explore the power that can be achieved by F#’s “Head + Tail” approach
  + Demonstrate the powerful pattern matching and recursion capabilities of F#
  + Demonstrate the usefulness of discriminated unions in F#
  1. Estimated time to complete this lab: **60 minutes**.

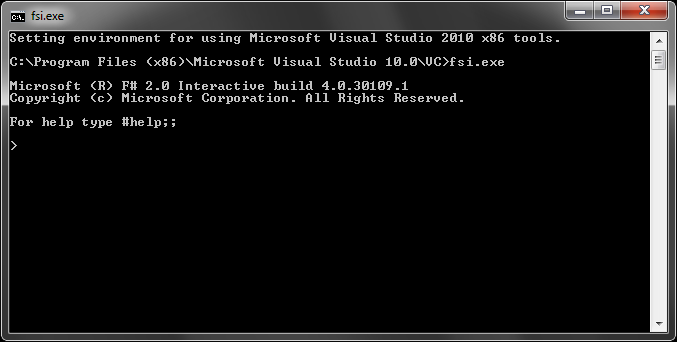
# Next Steps

Exercise 1: Types in F#

Exercise 1: Types in F#

* 1. In this exercise, you will be introduced to the F# interactive console built into Visual Studio 2010. You will also learn how you can leverage F#’s type inference and will also be introduced to the concept of “tuples”.

Task 1 – Observing Type Inference in F#

* 1. In this task you will use the F# interactive console to see that F# is capable of inferring data type via the context in which is used.
  2. F# is a statically typed language that makes heavy use of type inference. When provided with a primitive type (int, float, string, etc.) it’s able to infer the type the identifier refers to.
  3. Open the Visual Studio 2010 Command Prompt from **Start | All Programs | Microsoft Visual Studio 2010 | Visual Studio Tools | Visual Studio 2010 Command Prompt**
  4. Launch F# Interface by launching **fsi.exe**:
     1. 
     2. Figure 1
     3. The F# interactive console
  5. Type the following command into the F# interactive console:
     1. F#
     2. **42;;**
     3. Response
     4. **val it : int = 42**
     5. **Note:** The F# console will evaluate the value by determining its type and binding it to an identifier called “it”.
  6. Type the following command in the F# interactive console and press enter:
     1. F#
     2. **it;;**
     3. Response
     4. **val it : int = 42**
     5. **Note:** In absence of an identifier, the F# environment has bound the value to an identifier called “it”. This identifier is available to other code in the same interactive session.
  7. At the command prompt in the F# interactive session enter the following command and press enter:
     1. F#
     2. **42.0;;**
     3. Response
     4. **val it : float = 42.0**
     5. **Note:** By adding a decimal to your value F# has inferred that it is float.
  8. At the command prompt in the F# interactive session type the following command and press enter:
     1. F#
     2. **"42";;**
     3. Response
     4. **val it : string = "42"**
     5. **Note:** By enclosing your value in double quotes F# is able to infer that this value is a string.

Task 2 – Working with Tuples

* 1. In this task, you will see how to work with **tuples** in F#. **Tuples** are structures in F# that allow developers to ensure that two or more pieces of information are always presented together. This is useful in situations where it may not be sensible to separate data (i.e. a transaction amount, transaction date and transaction ID number).
  2. **Note:** In F#, a **tuple** is simply several values grouped together. **Tuples** are a fundamental type in F# and can be used anywhere any other value would be used.
  3. At the F# interactive command prompt type the following command and press enter:
     1. F#
     2. **(42, "Hello F#");;**
     3. Response
     4. **val it : int \* string = (42, "Hello F#")**
     5. **Note:** F# has inferred this value as a **tuple**. In this case, the tuple is composed of an integer value and a string value. The type description shown in the interpreter (“int \* string”) indicates that there are two values in this **tuple** and that the first is an **integer** and the second is a **string**.
  4. **Tuples** are not limited to pairs of values; they can contain as many values as are needed. Type the following command into the F# interactive console and press enter:
     1. F#
     2. **(42, "Hello F#", 42.0);;**
     3. Response
     4. **val it : int \* string \* float = (42, "Hello F#", 42.0)**
     5. **Note:** In this case, the **tuple** that was created is of type “int \* string \* float” or a structure that takes an **integer**, a **string** and a **float** respectively.

Task 3 – Functions as Values

* 1. In this task, you will see that functions in F# are values like any other and can be used in the exact same way.
  2. **Note:** In F#, there is no distinction between values and functions because functions themselves are values. They are first-class citizens. As such, a function can be bound to an identifier just like any other value.
  3. Type the command below at the F# interactive command prompt and press enter:
     1. F#
     2. **fun x -> x + 1;;**
     3. Response
     4. **val it : int -> int = <fun:clo@0>**
  4. Since the function that was created in the last step was bound to the “it” identifier, it can be accessed via that identifier. Enter the following command at the F# interactive command prompt:
     1. F#
     2. **it 41;;**
     3. Response
     4. **val it : int = 42**
     5. **Note:** F# accessed the function that was bound to the “it” identifier and executed the function. But how did F# know this was a function call? Unlike traditional Object-Oriented languages that use parentheses and commas to denote function calls (i.e. “MyFunction (firstParam, secondParam)”), F# knows that the first identifier is a function type (since functions are first-class citizens in F#) and expects all parameters of a function call to be separated by spaces (i.e. “MyFunction firstParam secondParam”).
     6. Hence, the F# code “it 41” is telling F# to call the function bound to the identifier **it** and passing it **41** as the first parameter.
  5. As a side effect, the value returned from the function was itself bound to “it”. At the command prompt, type the following command and press enter:
     1. F#
     2. **it;;**
     3. Response
     4. **val it : int = 42**
     5. **Note:** In this case you can see that the return value of 42 has been bound to the “it” identifier. This means that your function is no longer bound the identifier “it” and therefore cannot be used any longer. Well, that’s just a bit unfortunate. So, in the next section you will learn how to bind values to identifiers that will allow you to access those values as needed.

# Next Step

* 1. Exercise 2: Using the “Let” Keyword in F#

Exercise 2: Using the “Let” Keyword in F#

* 1. Strictly speaking, functional languages like F# do not have variables. Instead, values are said to be bound to identifiers so that they may be referenced elsewhere in the program. Accordingly, the F# “let” keyword allows you to assign any expression to an identifier.
  2. In this exercise you learn how to use the let keyword to bind values to identifiers.

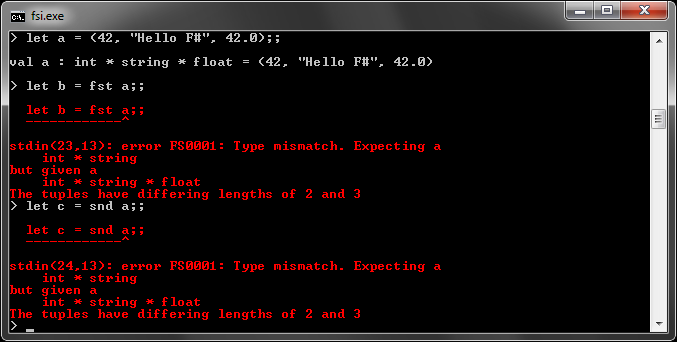
Task 1 – Using the Let Keyword to Bind a Value to an Identifier

* 1. If it is not already open, open the F# interactive console (see the beginning of Exercise 1 for instructions)
  2. Enter the following command at the F# interactive command prompt:
     1. F#
     2. **let a = 42;;**
     3. Response
     4. **val a : int = 42**
  3. The value 42 is now accessible by the identifier **a** that it is bound to. To verify this, enter the following command at the F# interactive console prompt:
     1. F#
     2. **a;;**
     3. Response
     4. **val it : int = 42**
     5. **Note:** In functional languages like F#, instead of assigning a value to a variable, a value is said to be bound to an identifier. This is because in pure functional languages once a value is assigned to an identifier it never changes. If a developer binds a value to an identifier (i.e. “let x = 42;;”) and then later in the same scope binds a new value to an identifier of the same name (i.e. “let x = ‘forty two’”) F# creates a new identifier, and gives it the same name. The previous identifier (in this case, the one bound to 42) still exists, but is no longer reachable.

F# is not, however, a pure functional language. By using the mutable keyword and the left ASCII arrow ( <- ) developers can create an identifier whose value can change. The mutable keyword is used to define the initial value, while the left arrow is used to change the value. It is possible to change the value; however, the type cannot be changed.

* 1. Identifiers can be used in computations. Enter the following command at the F# interactive command prompt;
     1. F#
     2. **a + 1;;**
     3. Response
     4. **val it : int = 43**
     5. **Note:** F# used the bound identifier in the computation and returned the result.
  2. Remember that unless you are using the **mutable** keyword, values in F# cannot be changed. Enter the following commands at the F# interactive command prompt:
     1. F#
     2. **let a = 0;;**
     3. **a;;**
     4. Response
     5. **val it : int = 0**
     6. **Note:** F# has bound the value 0 to a new identifier also called **a** (as mentioned above, the previous **a** identifier still exists, but is no longer reachable).
  3. Because binding a new value to **a** causes F# to create a new identifier called **a**, the new value that is being bound can be a completely different type than the previously bound value. Enter the following command at the F# interactive command prompt:
     1. F#
     2. **let a = "42";;**
     3. Response
     4. **val a : string = "42"**
     5. **Note:** You can see that even though the previous **a** contained an integer value, this new **a** contains a string.

Task 2 – Working with tuples and identifiers

* 1. In this task you will learn how tuples can be composed of identifiers that have been bound to values and separated into individual values which can also be in turn bound to identifiers.
  2. For this example you will need a tuple defined in F#. Enter the following command at the F# interactive command prompt:
     1. F#
     2. **let a = (42, "Hello F#");;**
     3. Response
     4. **val a : int \* string = (42, "Hello F#")**
     5. **Note:** This creates a **tuple** of type **int** \* **string** (a structure with an integer and a string);
  3. F# provides the functions **fst** and **snd** to break apart **tuples** made up of two elements. Enter the following commands at the F# interactive command prompt:
     1. F#
     2. **let b = fst a;;**
     3. Response
     4. **val b : int = 42**
     5. F#
     6. **let c = snd a;;**
     7. Response
     8. **val c : string = "Hello F#"**
     9. F# binds the first element (the integer) of the **tuple** to **b** and the second element (the string ) to **c**. Enter the following into the interactive console to verify the values of your identifiers:
     10. F#
     11. **b;;**
     12. Response
     13. **val it : int = 42**
     14. F#
     15. **c;;**
     16. Response
     17. **val it : string = "Hello F#"**
     18. **Note:** As you can see, **b** and **c** have now been bound to the individual values of the **tuple** **a**.
  4. An alternative method for binding the individual values of a **tuple** to identifiers is to use a let statement to bind the values within the **tuple** to a pattern of identifiers. Enter the following commands into the F# interactive command prompt:
     1. F#
     2. **let (b,c) = a;;**
     3. Response
     4. **val c : string = "Hello F#"**
     5. **val b : int = 42**
     6. F#
     7. **b;;**
     8. Response
     9. **val it : int = 42**
     10. F#
     11. **c;;**
     12. Response
     13. **val it : string = "Hello F#"**
     14. **Note:** F# is able to use its pattern matching abilities to bind each individual value in the **tuple** to the **b** and **c** identifiers respectively.
  5. The **fst** and **snd** functions are fine if your tuple only has two values. For tuples that have more than two values, F# allows us to use the **let** statement to bind the elements of a tuple to a pattern. Enter the following command into the F# interactive console:
     1. F#
     2. **let a = (42, "Hello F#", 42.0);;**
     3. Response
     4. **val a : int \* string \* float = (42, "Hello F#", 42.0)**
     5. Because this **tuple** is made up of more than one element, **fst** and **snd** will not work. In fact, if you try to use these methods F# Interactive will complain.
     6. 
     7. Figure 2
     8. Failure of fst and snd
  6. Even though **fst** and **snd** will only work on **tuples** containing two values, pattern matching can be used no matter how many values are contained in a **tuple**. Enter the following command at the F# interactive command prompt:
     1. F#
     2. **let (b, c, d) = a;;**
     3. Response
     4. **val d : float = 42.0**
     5. **val c : string = "Hello F#"**
     6. **val b : int = 42**
     7. You can verify this by examining the values bound to each identifier. Enter the following into the F# interactive console:
     8. F#
     9. **b;;**
     10. **c;;**
     11. **d;;**
  7. When using patterns of identifiers to get individual values from a **tuple**, there may be occasions where you only want a subset of the available values. F# provides a mechanism with its pattern matching syntax to ignore values that are not needed. By simply replacing the identifier in the pattern with an underscore character (**\_**), you can instruct F# to ignore that value when it performs the pattern match. In the example below, you are only concerned with getting the string value (the second value) from your **tuple**, you do not care about the first or third values. Enter the following command at the F# interactive command prompt:
     1. F#
     2. let (\_, e, \_) = a;;
     3. Response
     4. **val e : string = "Hello F#"**
     5. **Note:** F# has only bound the identifier that you provided in out pattern; in this case **e** is bound to the string “Hello F#”.

# Next Step

Exercise 3: Functions

Exercise 3: Functions

* 1. Because F# views functions as being simply another type of data, the language provides a great deal of flexibility in how you work with functions. In this exercise you’ll continue to work with the F# interactive console to become familiar with the way F# handles functions.

Task 1 – Binding Functions to Identifiers

* 1. In this task you will see how functions are values by binding them to identifiers.
  2. Create a function and bind it to the identifier **addTenToNumber**. Enter the following command at the F# interactive command prompt;
     1. F#
     2. **let addTenToNumber = (fun x -> x + 10);;**
     3. Response
     4. **val addTenToNumber : int -> int**
     5. This binds a function to the identifier **addTenToNumber**. The function is declared with the **fun** keyword, followed by an argument (or list of arguments), the **->** operator and finally the body of the function.
     6. **Note:** F# has bound the identifier **addTenToNumber** to your type, which is defined as **int -> int**. This indicates that the value bound to **addTenToNumber** is a function which takes an argument that is an integer, and returns an integer.
  3. You can invoke the function by referencing it via its identifier. Enter the following command at the F# interactive command prompt;
     1. F#
     2. **addTenToNumber 32;;**
     3. Response
     4. **val it : int = 42**

Task 2 – Binding Functions with Multiple Parameters

* 1. In this task you will examine how F# handles functions with multiple parameters.
  2. First you need to create a function with multiple parameters. To build on your last example of **addTenToNumber**, you will create a more general function called **addTwoNumbers**. Enter the following command at the F# interactive command prompt:
     1. F#
     2. **let addTwoNumbers = (fun x -> (fun y -> x + y));;**
     3. Response
     4. **val addTwoNumbers : int -> int -> int**
     5. **Note:** The syntax used here to declare a function with two arguments goes to further demonstrate that in F# function are values. Read that code carefully. The first thing to notice is that **addTwoNumbers** isn’t really one function, it’s actually two functions: one function that accepts the first parameter and returns yet another function that accepts the second parameter.
     6. This is completely different to how functions behave in traditional .NET languages that exist today.
     7. Your first thought might be “wow, that syntax is very ugly and hard to understand.” Luckily, there is a much easier shortcut syntax that you will see shortly. But once you understand how functions are defined under the hood like this, it becomes much easier to understand other advanced concepts you will cover like “partially-applied functions.”
  3. You can call the function and see that it behaves the way you would expect it to by adding two numbers together. Enter the following code at the F# interactive command prompt:
     1. F#
     2. **addTwoNumbers 21 21;;**
     3. Response
     4. **val it : int = 42**
  4. While the syntax used in the previous example works (and is more in line with the way F# handles functions), it can be quite difficult to work with, especially as the number of parameters continues to grow. Even in your scenario with just two arguments, the code is more difficult to read and understand than it needs to be. Luckily, F# provides a simpler syntax for declaring functions. Enter the following command at the F# interactive command prompt;
     1. F#
     2. **let addTwoNumbers x y = x + y;;**
     3. Response
     4. **val addTwoNumbers : int -> int -> int**
     5. **Note:** F# has bound your function, which takes two integers, to the identifier **addTwoNumbers**. Notice that F# still says the signature is “int -> int -> int”, meaning a “function that takes an int that returns a function that takes an int that returns an int.” And yes, that sentence may take a couple of reads to soak in!
     6. It is important to remember that in F#, arguments for functions are not typically enclosed in parentheses like they are in C#. To declare a function with two arguments, the syntax is:
     7. let functionName x y = …;;
     8. This is NOT the same as:
     9. let functionName (x, y) = …
     10. In the second case you have told F# that your function expects a single parameter, a **Tuple**, for the argument to your function.
  5. Just like any other value, values returned from functions in F# can be bound to identifiers. Enter the following command in the F# interactive console;
     1. F#
     2. **let a = addTwoNumbers 21 21;;**
     3. Response
     4. **val a : int = 42**
     5. F#
     6. **a;;**
     7. Response
     8. **val it : int = 42**

Task 3 – Working with partially-applied functions

Because of the way functions are defined in F#, F# lends itself naturally to the usage of “partially-applied” functions (also known as “curried” functions).

* 1. Partially-applied functions are functions that allow them to be called without having all of their input arguments applied. What is returned from a partially-applied function is a function whose parameter is the next expected parameter in the function chain. First, you need to recreate your function that takes two integer values as arguments. Enter the following code into the F# interactive command prompt:
     1. F#
     2. **let addNumbers x y = x + y;;**
     3. Response
     4. **val addNumbers : int -> int -> int**
  2. You can call this function by passing it two integers and see the results. Enter the following command at the F# interactive command prompt:
     1. F#
     2. **addNumbers 20 22;;**
     3. Response
     4. **val it : int = 42**
  3. To show the power of partially-applied functions, rewrite your **addTenToNumber** function to partially apply the **addNumbers** function. Enter the following command into the F# interactive command prompt:
     1. F#
     2. **let addTenToNumber = addNumbers 10;;**
     3. Response
     4. **val addTenToNumber : (int -> int)**
     5. **Note:** Notice that even though the **addNumbers** function expects two parameters, you are only calling it with a single parameter. Remember how functions are defined in F#. Your **addNumbers** function is really defined as “let addNumbers = (fun x -> (fun y -> x + y))”. So, when you call **addNumbers** with a single parameter, a **function** is returned that is expecting the second parameter. In this vein, the **addNumbers** **function** has been partially applied. This returned **function** is then bound to the identifier **addTenToNumber**.
  4. In this case, the identifier **addTenToNumber** is bound to a function that expects an integer argument and returns an integer. This happens to be the output of the **addNumbers** function when it is passed one argument. You can call the function bound to **addTenToNumber** by entering the following code at the F# interactive command prompt:
     1. F#
     2. **addTenToNumber 32;;**
     3. Response
     4. **val it : int = 42**

# Next Step

Exercise 4: Lists

Exercise 4: Lists

* 1. This exercise will demonstrate how F# works with lists. It will cover how F#’s implementation of a linked list is very different from its counterpart in imperative programming languages, explain the Head + Tail approach and demonstrate the abilities this approach gives your code through the use of recursive functions and pattern matching.

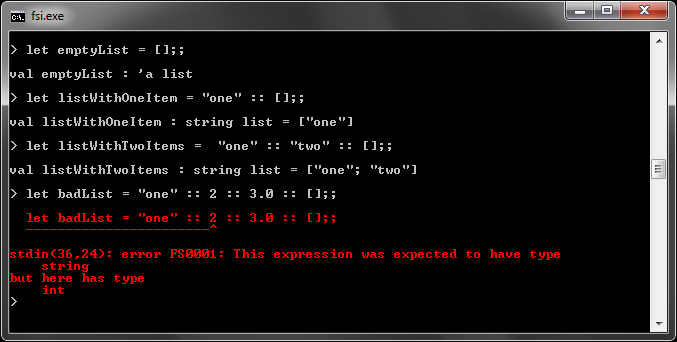
Task 1 – Creating Lists

* 1. In this task you will learn the syntax to create lists in F#.
  2. **Note:** F#, like all functional languages, implements its lists as linked lists. However, the implementation of linked lists in F# is a little different from that of imperative languages. Most imperative languages implement linked lists by creating nodes which contain a value and a pointer to the next node in the list. This makes it very easy to add items in the middle of the list. F# implements lists as **Heads** and **Tails**. Essentially, a list item node in F# consists of a value and a **tail**, which itself is another list.
  3. For more information on how F# deals with lists, please refer to Dustin Campbell’s blog at <http://diditwith.net/2008/03/03/WhyILoveFListsTheBasics.aspx>
  4. One way to create lists in F# is to specify a list of item separated by double-colons (**::**) with the empty list (**[]**) at the end of the list. Technically, F# views this as concatenating lists: each element in the list is viewed as being concatenated to either another element or an empty list. The end result is the same; F# views the concatenated values as a list. The sample code below creates three lists; and empty list, a list with one item and a list with two items. Enter the following code into the F# interactive console:
     1. F#

**let emptyList = [];;**

* + 1. Response
    2. **val emptyList : ‘a list**
    3. F#
    4. **let listWithOneItem = "one" :: [];;**
    5. Response
    6. **val listWithOneItem : string list = ["one"]**
    7. F#
    8. **let listWithTwoItems = "one" :: "two" :: [];;**
    9. Response

**val listWithTwoItems : string list = ["one"; "two"]**

* + 1. **Note:** Notice that in all cases, F# was able to correctly infer the type of the list automatically. What is especially interesting is the case of the first list you created. What does **‘a** mean? That simply means it is a generic list where the type will be inferred at usage time.
    2. Also notice that through this entire lab so far **you have not actually needed to specify a type a single time**. Because of this, a lot of developers mistakenly think F# is a dynamic language. It is not. It is very much a statically-typed language. But F#’s aggressive use of type inference removes a lot of the necessary of specifying types that exist with other languages.
  1. F# does not allow lists to contain different types. For example, the sample code below will not compile because it is trying to put strings and integers in the same list:
     1. F#
     2. **let badList = "one" :: 2 :: 3.0 :: [];;**
     3. **Note:** Because these values are not of the same type, F# cannot put them in the same list.
     4. 
     5. Figure 3

A bad list

* 1. The syntax above is very helpful in concatenating lists, but it’s a little unwieldy for creating lists from scratch. F# offers a short-hand for creating lists. Placing semi-colon delimited values in between square brackets will define a list in F#. Enter the following code into the F# interactive console :
     1. F#
     2. **let easyList = ["A"; "B"; "C"];;**
     3. Response
     4. **val easyList : string list** **= ["A"; "B"; "C"]**
  2. Because of the nature of how F# implements lists, it is easy to combine two lists into a single list. The example below creates two lists and then combines them into a third list. Enter the following code into the F# interactive console:
     1. F#
     2. **let firstList = ["A"; "B"; "C"];;**
     3. Response
     4. **val firstList : string list = ["A"; "B"; "C"]**
     5. F#
     6. **let secondList = ["1"; "2"; "3"];;**
     7. Response
     8. **val secondList : string list = ["1"; "2"; "3"]**
     9. F# will create two lists with the appropriate values. Now, to combine these lists, F# uses the **@** operator. Enter the following code into the F# interactive console:
     10. F#
     11. **let combinedList = firstList @ secondList;;**
     12. Response
     13. **val combinedList : string list = ["A"; "B"; "C"; "1"; "2"; "3"]**
     14. F# has created new list. To verify that the new list is the values of your combines list, you can examine its contents by entering the following into the interactive console:
     15. F#
     16. **combinedList;;**
     17. Response
     18. **val it : string list = ["A"; "B"; "C"; "1"; "2"; "3"]**

# Next Step

Exercise 5: Pattern Matching and Recursion

Exercise 5: Pattern Matching and Recursion

* 1. Pattern matching in F# is similar to switch statements in C#; however they provide abilities and functionality beyond the C# construct, particularly when used in concert with F#’s implementation of lists and recursion. In this Exercise you will be introduced to F#’s pattern matching syntax and see some examples which highlight the power of pattern matching.

Task 1 – Using Simple Pattern Matching to Output Information

* 1. In this task you will learn the basic pattern matching syntax in F#.
  2. The pattern matching syntax in F# is very straight forward. Enter the following code into the F# interactive console command prompt:
     1. **Note:** Whitespace is significant in the F# console. You’ll need to have at least one whitespace character to indent on subsequent lines after the first. The convention is to use four spaces, not a tab.
     2. F#
     3. **let patMatch x =**
     4. **match x with**
     5. **| 0 -> printfn "Value is 0"**
     6. **| 1 -> printfn "Value is 1"**
     7. **| \_ -> printfn "Value is not 0 or 1";;**
     8. Response
     9. **val patMatch : int -> unit**
  3. By stacking patterns with the **|** operator, it is possible to compare input values to various patterns. Enter the following code into the F# interactive command prompt:
     1. F#
     2. **let patMatch x =**
     3. **match x with**
     4. **| 0 | 1 -> printfn "Value is 0 or 1"**
     5. **| \_ -> printfn "Value is not 0 or 1";;**
     6. Response
     7. **val patMatch : int -> unit**
  4. F# also provides syntax to easily perform pattern matching over **tuples**. The next example will demonstrate how an **if** control structure for “anding” two values can be implemented with pattern matching. Enter the following code into the F# interactive command prompt:
     1. F#
     2. **let andTrue x =**
     3. **match x with**
     4. **| (true, true) -> true**
     5. **| \_ -> false;;**
     6. Response
     7. **val andTrue : bool \* bool -> bool**

This function takes a **tuple** made up of two **boolean** values and returns a **boolean**.

* 1. You can verify this code by calling it with the following tuples:
     1. F#
     2. **andTrue (false, true);;**
     3. Response
     4. **val it : bool = false**
     5. F#

**andTrue (true, true);;**

* + 1. Response
    2. **val it : bool = true**

Task 2 – Using Simple Pattern Matching and Recursion to Implement the Fibonacci Sequence

* 1. In this task you will learn to supplement pattern matching in F# with recursive functions. Recursion is defining a function in terms of itself. In the following example if **x** is not 0 or 1, the function calls itself passing values derived from **x**. This approach is often used in functional programming in place of loops as it can make algorithms easier to understand.
  2. To create a recursive function in F#, you must use the **rec** keyword. The example below is an F# implementation of a Fibonacci function which uses recursion to call itself to determine the result. Enter the following code into the F# interactive console command prompt:
     1. F#
     2. **let rec fib x =**
     3. **match x with**
     4. **| 0 -> 0**
     5. **| 1 -> 1**
     6. **| \_ -> fib(x - 1) + fib(x - 2);;**
     7. Response
     8. **val fib : int -> int**
     9. **Note:** The **Fibonacci Sequence** is a sequence of numbers where the next number in the sequence is defined by adding the value of the two numbers previous two it (starting with the first two number 0 and 1). For instance, here are the first 10 numbers in the Fibonacci Sequence starting with 0:
     10. **0, 1, 1, 2, 3, 5, 8, 13, 21, 34**
  3. Calling the function with the value of 42:
     1. F#
     2. **fib 42;;**
     3. Response
     4. **val it : int = 267914296**
     5. **Note:** The **rec** keyword is required in F# to indicate that a function is recursive.

Task 3 – Using Recursion with F# Lists

* 1. F#’s recursion ability also makes it ideal for processing items in a list. This task will simulate performing some sort of processing on items in a list in a recursive manner.
  2. The function below takes a list as an argument. Here you leverage the power of pattern matching. Notice that pattern matching works with the **::** operator you used earlier to build lists. So, if the list passed in is empty, you will simply print “done”. Otherwise, you are going to print the value of the **head** and then recursively call the function again to process the **tail**. Enter the following code into the F# interactive console command prompt:
     1. F#

**let rec processList x =**

* + 1. **match x with**
    2. **| head::tail -> printfn "processing %i" head; processList tail**
    3. **| [] -> printfn "done";;**
    4. Response
    5. **val processList : int list -> unit**
    6. **Note:** In F# the single semi-colon can be used to separate different statements on the same line.
    7. Also notice that F# was able to infer that the function accepts a list of integers merely because the string you printed used a formatting of “%i” (for int).
  1. You will create a **list** to send to the function for processing. Enter the following code in the F# interactive console command prompt:
     1. F#
     2. **processList [1; 2; 3; 4; 5];;**
     3. Response
     4. **processing 1**
     5. **processing 2**
     6. **processing 3**
     7. **processing 4**
     8. **processing 5**
     9. **done**
     10. **val it : unit = ()**

# Next Step

Exercise 6: Types and Discriminated Unions

Exercise 6: Types and Discriminated Unions

* 1. Like most modern languages, F# provides facilities to create user defined types. F# also allows developers to create “discriminated unions”, which represent data or structures and convey different meanings depending on how they are used.
  2. The following exercise will demonstrate how to create simple types in F#, create discriminated unions and then use those unions in pattern matching algorithms.

Task 1 – Creating a Simple Type in F#

* 1. In this task you will create a simple type in F#.
  2. You will create a type to hold an item name and a unit price. You will define the item name and unit cost as type aliases first. In the F# interactive console, enter the following command:
     1. F#
     2. **type ItemName = string;;**
     3. Response
     4. **type ItemName = string**
     5. F#
     6. **type UnitPrice = float;;**
     7. Response
     8. **type UnitPrice = float**
  3. *Record Types* in F# are akin to classes or structures in that they are types that are composites of different values. You can use the standard primitive F# types in your record types, or your user defined types. In the F# interactive console, enter the following command:
     1. F#
     2. **type OrderItem = {ItemOrdered : ItemName; Quantity : int; PricePer : UnitPrice};;**
     3. Response
     4. **type OrderItem =**
     5. **{ItemOrdered: ItemName;**
     6. **Quantity: int;**
     7. **PricePer: UnitPrice;}**
     8. **Note:** The syntax for creating the record type is pretty simple. The fields are enclosed in curly braces with each field being delineated by a semi-colon. The fields themselves are composed of a name and a type separated by a colon. You can see that the ItemOrdered and PricePer fields are made up of type aliases you defined, but you are also able to define the Quantity field as a normal F# type.

Task 2 – Creating a Discriminated Union

* 1. In this task you will see how discriminated unions can be used to hold temperature information. While Celsius, Fahrenheit and Kelvin temperatures are quite different, a discriminated union is able to represent the correct value. Discriminated unions are made up of two pieces of information, the type of the value, known in F# as the **constructor**, and the **value** itself.
  2. Create a type which is a discriminated union to hold your temperature data. In the F# interactive console, enter the following command:
     1. F#
     2. **type temperature =**
     3. **| Celsius of float**
     4. **| Fahrenheit of float**
     5. **| Kelvin of float;;**
     6. Response
     7. **type temperature =**
     8. **| Celsius of float**
     9. **| Fahrenheit of float**
     10. **| Kelvin of float;;**
     11. **Note:** In this case, the constructors are Celsius, Fahrenheit and Kelvin which all take a value of type float.
  3. To use the type you simply declare an identifier and tell F# which type of value you are providing. In the F# interactive console, enter the following command:
     1. F#
     2. **let temp1 = Celsius 32.0;;**
     3. Response
     4. **val temp1 : temperature = Celsius 32.0**
     5. F#
     6. **let temp2 = Fahrenheit 98.6;;**
     7. Response
     8. **val temp2 : temperature = Fahrenheit 98.6**
     9. F#
     10. **let temp3 = Kelvin 5.0;;**
     11. Response
     12. **val temp3 : temperature = Kelvin 5.0**
     13. **Note:** Although these are all different type of values, they are all of the type temperature.
  4. If you look at the values, you can see that in addition to the data, the type of the data was stored as well. Enter the following into the F# interactive console:
     1. F#
     2. **temp1;;**
     3. Response
     4. **val it : temperature = Celsius 32.0**

Task 3 – Using Pattern Matching in Discriminated Unions

* 1. In this task, you will see how to use pattern matching to deconstruct discriminated unions.
  2. Building on your temperature example, you will create functions to convert temperatures. Create these functions by entering these commands into the F# interactive console:
     1. F#
     2. **let convertToFahrenheit x =**
     3. **match x with**
     4. **| Celsius x -> Fahrenheit (x \* (9.0 / 5.0) + 32.0)**
     5. **| Fahrenheit x -> Fahrenheit x**
     6. **| Kelvin x -> Fahrenheit (x \* (9.0 / 5.0) - 459.67);;**
     7. Response
     8. **val convertToFahrenheit : temperature -> temperature**
  3. By calling these functions and passing in the temperature values you created earlier, you can see that they are able to perform pattern matching against the constructor of each union to determine which algorithm to apply.
     1. F#
     2. **convertToFahrenheit temp1;;**
     3. Response
     4. **val it : temperature = Fahrenheit 89.6**

# Next Step

* 1. Summary

Summary

* 1. In this lab, you have gotten an introduction to F#, the new functional programming language from Microsoft. F#’s use of functions as first-class citizens, and It’s support for tuples, pattern matching, and its succinctness make it a very powerful multiparadigm language.