Practical 1

Functional Programming, 2024

Deadlines:

- Practical section submitted by **5pm** on **Thursday 22 February**.
- Homework section submitted by **2pm** on **Tuesday 27 February**.

In this practical, you will practice creating some simple functions, do a little recursion, and get familiar with the language as you type it out.

Please **do not** copy and paste code from this document—instead, type it out. Research has shown that the act of *writing something out yourself* will improve learning and retention.

If you complete the Warm-up section, you will easily complete the Practical section and be well-prepared for the Homework section.

Setup

1. Install Visual Studio Code.
2. Install the .Net 8 SDK.
3. Install the "Polyglot Notebooks" and "Ionide for F#" VS Code extensions.
4. Download the "Practical 1" notebook from RUConnected, and open it up in Visual Studio Code.
    a. You may need to "trust the workspace" if you see a warning message in Visual Studio Code.

Warm-up

We'll start with a very simple function that you've already seen in class: calculating VAT.

1. In the first code block, write the code:

```
fun x → x * 1.15
```

2. Now click on the "Run" icon that's just on the left of the code block. This icon will appear when your mouse hovers over the code block.
3. You'll see that a green check-mark appears, indicating that it is recognized as valid F# code. Call a tutor if you're not seeing this.
4. However, there is no output: the software we're using doesn't know how to display a function. But if we give it an input, then we can see the value displayed as an output. Make a new code block and **apply** the VAT function. Write and run this code:

```
(fun x → 1.15 * x) 16.51
```

5. Try these simple experiments:
    a. Change the VAT rate to be 14% and re-run the code block. What does the answer change to?
    b. Change the `x` to be a different symbol—perhaps `y` , or `foo` . Identifiers in this functional language (F#) always begin with a lowercase letter or underscore, and may contain any combination of letters or numbers[1]. *Make sure that your code still runs after this*!
    c. Try to use a negative floating-point number as input.
    d. Change the input to be `7.0` .
6. Now try to change the input in the code block to be 7:

```
(fun x → 1.15 * x) 7
```

   Now run this. What do you notice?
7. An error has occurred. This is because F# will refuse to multiply a floating-point number (1.15) by an integer (7). Other languages (such as Python) will just convert the integer 7 into the floating-point number 7.0 behind your back[2], but F# will not do that.
8. Return the code block to its original, correct state:

```
(fun x → 1.15 * x) 16.51
```

9. Make a new code-block and create a lambda function to evaluate the following function:

   $2x^3 + \frac{1}{4}x^2 - 5x + 10$

   function. Remembering what we've said about types, can you do it by yourself? Try it out—and *remember* that integer and floating-point types don't mix!

   If you *can*'t figure it out by yourself, this is one way of doing it:

```
fun x → 2.0 * x * x * x + (1.0 / 4.0) * x * x - 5.0 * x + 10.0
```

   Notice how I've put spaces between the items. That's useful for readability, and it's useful to avoid certain errors such as accidentally typing "`-5.0`" (negative 5) instead of "`- 5.0`" .
10. Now run your function with the value `7.0` . You should get the answer 673.25.

---

[1] There's also an odd feature that lets you write *any* character sequence and have it accepted as an identifier. by including it in double-backticks. So you can have ``` ``My very long name`` ``` as an identifier, if you really want to…

[2] The technical name for this is **type coercion**: where a programming language sees that the two types can't actually work together in an expression, but might work together if one (or both) were converted into different types, so it "coerces" one or both of their types and then proceeds to evaluate the expression.

11. I have some exercises for you to try out, each one in a new code block. I've included answers below if you can't see how it's done after trying for 5 minutes. But it will help you to try them out yourself first, before you look at any answers. So, why not try to make functions that...
    a. Add the two numbers
    b. Only return the first number
    c. Return the inputs, but swapped around
    d. Return the value of **a** multiplied by 100
    e. Return **true** if **b** is less than **a**
    f. Return the remainder[3] after dividing **a** by **b**
    g. Return the sum of **a** and **b**, the product of **a** and **b**, and the difference between **a** and **b**, in that order

```
fun (a, b) → a + b
or
fun a b → a + b
or
fun a → fun b → a + b

fun (a, b) → a
or
fun a b → a
or
fun a → fun b → a

fun (a, b) → (b, a)

fun a → a * 100

fun (a, b) → b < a
or
fun a b → b < a
or
fun a → fun b → b < a

fun (a, b) → a % b
or
fun a b → a % b
or
fun a → fun b → a % b

fun (a, b) → (a + b, a * b, a - b)
or
fun a b → (a + b, a * b, a - b)
```

[3] The "mod" operator is % , just as it is in many other languages.

```
or
fun a → fun b → (a + b, a * b, a - b)
```

Notice that when there are "multiple inputs", we have sometimes written them as a *group of inputs* (i.e. a tuple) and sometimes we have written them as a function returning a function. The second style is much, much more common in functional programming, and much more flexible as well; you will discover why later on. We will prefer the second style, going forward.

12. In a new code block, see if you can create a "maximum" function which accepts two inputs and returns the larger one. You will need to use an **if**-expression for this.

    if-expressions must have an **else** branch, because an **if**-expression generates a result[4]. If you are having trouble creating it yourself, see the code below.

```
let maximum a b =
  if a > b then
    a
  else
    b
```

13. Lastly, let's try some recursion in a new code block:

```
let rec addUp x y =
  if x < y then
    x + addUp (x+1) y
  else
    y
addUp 2 5
```

    We have placed brackets around the "x+1" to indicate where that input begins and ends. Can you see how this function works? It adds up the numbers between the specified values. For example, the function application above will perform the calculation 2+3+4+5 = 14 and show that answer. Play around with it until you are comfortable with how it works.

14. Now let's try a bigger example. We will begin by creating a function to get the *last* digit from a number. All numbers—including zero—have at least one digit in them, so there's always a "last digit" that we can work with.
15. You'll see a code block just after the "last : int -> int" description. Write this code in that code block:

```
let last n =
```

[4] There is one exception to this rule: when the result of **if**-expression is a **unit** value. That's because a unit value conveys no information whatsoever, so it doesn't matter if there's no **else** branch.

```
  n % 10
```

This is quite simple: just use the "modulo" operation to get the remainder after dividing by 10. That's the last digit, of course.

The **int -> int** part just specifies the domain and codomain of the function: in this case, both are integers (i.e. "ℤ" in mathematics).

16. Run the code block and verify that it compiles and runs. (you will see a green ✔ mark if it does).
17. Now run the tests for **last**. You will notice that the last two tests fail, and you can see what it expected and the actual value received. When a negative number is given to **last**, the remainder is also negative—that's how the "modulo" operation works. We need to fix **last** so that it works properly.

    Change the **last** code so that it reads:

```
let rec last n =
  if n ⩾ 0 then
    n % 10
  else
    last (-1 * n)
```

    (that's "n >= 0" written above; my code font displays it as the mathematical operator "≥"). The change we've made is this: if the value is positive, give back the last digit; if it is negative, feed a positive version to the same function—and we already know that the function works properly for positive values. Now run the tests again and verify that they are all passing.

    Notice that this is a bit different from how you might do it in imperative code. I've included an imperative version of **last** under the tests. When writing imperative code, you often "get it wrong" first and then "fix it up" afterwards—often by mutating something. Sometimes, you might end up using the "get it wrong" ➡ "fix it up" approach in functional code; but much more often, you'll focus on just getting it right in the first place. That's what our functional code above does.
18. We can already get the last digit. Let's make a function to get the first one now.

19. Insert this into a new code-block:

```
let rec first n =
  if n < 0 then
    first (-1 * n)
  elif n < 10 then
    n
  else
```

```
    first (n / 10)
```

You can already see why the first branch is there. Can you follow the rest of the code? Can you see the logic behind this approach? See if you can follow it through, and then check if your explanation is the same as the one in the next paragraph.

Every number must, of course, have a "first" digit—for the same reason that every number must have a "last" digit. If a number is in the range 0..9, then that's the first *and* the last number: we can just give back the same number. But if it's a larger number than that, then its first number must be somewhere on the left. So we get rid of the rightmost digit (by dividing by 10), and we see if our smaller number is the first digit. Because we keep on removing digits, we will eventually end up with the first number.

20. Execute the code, then execute the tests. Verify that all the tests pass.
21. Now let's have some code that finds out how many digits there are in a number. Enter this into a new code-block:

```
let rec length v =
  if v < 0 then
    length (-1 * v)
  elif v < 10 then
    1
  else
    1 + length (v / 10)
```

The logic of this is quite similar to what you've seen before. Ensure that we have a positive number; if it's a single-digit number, then it must be 1 digit long; and otherwise, it must be 1 digit ***plus*** all the digits that remain.

Notice this: just like in the other code we've written, we don't care about updating some memory location or variable! We're giving a *definition* of the length of a number, in a mathematical sense. Each branch in the program is just another case of the definition:

$$\text{Let } length(n) = \begin{cases} length(-1 \cdot n) & n < 0 \\ 1 & 0 \geq n < 10 \\ length(\lfloor \frac{n}{10} \rfloor) & \text{otherwise} \end{cases}$$

22. Execute the code and tests, and verify that the tests pass.
23. Now let's try something a bit more interesting. What do we have to do to ***reverse*** the digits in a number? Let's think it through together.
    a. Every digit in a number is a *positional value*. A number like 718 and a number like 178 are not the same because the positions of the digits are different:
        i. 718 = 700 + 10 + 8

ii. 178 = 100 + 70 + 8

b. If we start on the right side of a number, and move towards the left, every position that we move is 10x more than the last position. So, for example, 5718 = 5·10·10·10 + 7·10·10 + 1·10 + 8.

c. If we wanted to *reverse* a number like 5718, the digits would stay the same, but their positions would change to become: 8175 = 8·10·10·10 + 1·10·10 + 7·10 + 5.

That gives us enough to work with. However, one function that would be very convenient is a function to give us the multiplier for a particular positional value. Let's make that.

24. Start by inserting a new code block. In this code block, create a **multiplier** function to provide the correct multiplier for each position.

```
let rec multiplier n =
  if n ⩽ 1 then
     1
  else
     10 * multiplier (n - 1)
```

25. Insert another code cell, and insert some tests. **multiplier 0** should be **1 ; multiplier 1** should be **1** ; **multiplier 2** should be **10** ; **multiplier 3** should be **100** ; and so on. Compile the code, run the tests, and satisfy yourself that it works.
26. Now create a code block for **reverse**. Place this code into it:

```
let rec reverse n =
  if n < 0 then
    -1 * reverse (-1 * n)
  elif n = 0 then
    0
  else
    last n * (length >> multiplier) n + reverse (n / 10)
```

The first branch is trivial: if it's negative, change the sign so that we're only dealing with positive numbers; and multiply by -1 when we are returning, so that we can restore the correct sign. The second branch, which is a base case, is also trivial. It's the last branch which is interesting. Using the functions we've already defined, we

a. grab the last digit of the number;
b. multiply it by the correct positional multiplier, which we get by looking at the length of the number;
c. add it to the remaining numbers.

27. Unfortunately, there's a "bug" in **reverse**. If you give it a number such as 100, it will gladly reverse it ... but the result will be **1**. In our positional number system, 1 = 001, so this is correct. However, if you reverse the same number twice, you are *not* guaranteed to get back to your original number. We don't (yet) have a way of solving this problem, but perhaps we will have some method of doing so in the

coming weeks—stay tuned! For now, though, just remember that **reverse** is a dangerous function that can result in information being lost.

28. We might want to index into a number, to find the n^{th} digit from the left. How do we do that? Well, let's consider what we have. We have a **length** function which will tell us what the valid range for an index is. We have a way of going through the digits, by dividing by 10 each time. Let's put these together and see what we come up with:

```
let rec digitAtIndex x index =
  if index = 0 then
    last x
  else
    digitAtIndex (x/10) (index - 1)

let digitAt x index =
  let len = length x
  if index < 0 || index ≥ len then
    -1 // this is out of bounds.
  else
    digitAtIndex x (len - index - 1)
```

Can you see how the code works? It's similar to other code that you've seen above, in terms of algorithm and how it goes through the number. However, there are a number of features that make this code distinct and interesting:

   a. We see a binding inside a function, and it's a binding of the **len** symbol to a value: **let len = length x**. We do this because we're going to be reusing the value that we get from "length x", and when we reuse values, we attach names to them.
   b. We have *two* functions: **digitAtIndex** and **digitAt**. The **digitAt** function is actually the one that we're going to be calling. It does some basic sanity checks and sets up the values that we're going to use. The **digitAtIndex**, on the other hand, is what does the actual work. When **digitAtIndex** is called, we're sure that the values that it's using are going to be sensible values, because **digitAt** has already taken care of that for us.

29. Compile and run the code and tests, and verify for yourself that it all works as expected.
30. We've said that **digitAtIndex** won't be called unless it's called with sane values, but in fact, someone might call it directly—completely bypassing **digitAt**! We're going to make a small change that will stop that from happening. We're going to move the **digitAtIndex** binding *into the scope* of **digitAt**. That means that if you are *outside* of that scope, the binding won't exist at all, and the function can't be called directly.

We do this by just copying the **digitAtIndex** code, pasting it into **digitAt**, at the top, and indenting it[5]. Your code should then look like this:

```
let digitAt x index =

  let rec digitAtIndex x index =
    if index = 0 then
      last x
    else
     digitAtIndex (x/10) (index - 1)

  let len = length x
  if index < 0 || index ⩾ len then
    -1 // this is out of bounds.
  else
    digitAtIndex x (len - index - 1)
```

For readability, we've left a blank line before and after the pasted-and-indented code, and I've made it bold in the text above. None of this makes any difference.

31. Again, compile the code and run the tests. Verify for yourself that it all works as expected.
32. We can make a small change for readability. Notice that **x** is bound by **digitAt**, and it is also bound by **digitAtIndex**. This is legal, but it can get confusing. Rename the **x** symbol of **digitAtIndex** so that it is named **n**. In the end, your code should look something like this:

```
... code elided ...
  let rec digitAtIndex num index =
    if index = 0 then
      last num
    else
     digitAtIndex (num/10) (index - 1)
... code elided ...
```

Does this all work so far? Have you understood what you've done?

If so, then it's time to move on to...

Practical [22]

In this section:

- you may use functions that you have already defined (including those from the warm-up), but no built-in functions;

[5] **Indenting the code** is critically important here. Just like Python, F# figures out where scope begins and ends by using *indentation*. "Messy" code that's indented in weird ways is much more likely to be buggy code.

- you may create as many functions as you want, to achieve your goals.

There are tests that your functions are expected to pass. There's some syntax in the tests that you might not understand yet; we'll get to that later. For now, it's the tests themselves that are important.

1. [1] Create a function named **seconds** which accepts a group of 3 inputs representing (hours, minutes, seconds) and returns the total number of seconds.
2. [5] Create a recursive function **betterAddUp** which will accept either `2 5` or `5 2` and give the same answer. (In other words, it should count up from the *smaller* number to the *larger* one, no matter which order they are supplied in).
3. [5] Now create **evenBetterAddUp** which is like **betterAddUp**, but also converts negative numbers to positive numbers before it begins adding them up.
4. [5] Create a function named **clockHours** which accepts a number of hours on a 24-hour clock and returns the hour that a 12-hour clock would display. *Example*: On a 24-hour clock, "18" hours means "6" o'clock in the evening; "13" hours means "1" o'clock in the afternoon; and "0" hours means midnight. Of course, "8" hours means "8" o'clock in the morning.
5. [1] Create a function named **addHours** that takes a 24-hour clock value and adds a certain number of hours to it. It then tells you what the hand on a 12-hour clock will point to afterwards. For example, if it is 10 o'clock now, and we want to know what the time will be in 16 hours, then we should be able to write `addHours 10 16` and learn that it will be 2 o'clock after 16 hours have passed.
6. [5] Create a **peek** function which accepts an integer and an index, and returns the n^{th} digit from both sides of the integer. See the tests to understand the functionality of **peek**.

Submit via RUConnected.

Homework [21]

Continue using the same file you used for your practical.

1. [5] Create a recursive function that implements this mathematical function (you may assume that n ≥ 1):

$$\text{Let } f(n) = \begin{cases} 0 & n = 1 \\ 1 + f(\frac{n}{2}) & \frac{n}{2} \in \mathbb{N}, \\ 1 + f(3n+1) & \text{otherwise.} \end{cases}$$

2. [5] Create a recursive function that returns a string which is the hexadecimal representation of the input number. Do not use built-in functions. If the input number is negative, return an empty string.
    a. Example: input 12345678 should result in output string “BC614E”.
3. [5] Create a recursive function that returns a string which is the binary representation of the input number. Do not use built-in functions. If the input number is negative, return an empty string.
    a. Example: input 12345678 should result in output string “101111000110000101001110”.
4. [1] Create a function that accepts an integer and returns two strings: the hexadecimal representation of the number and the binary representation of the number. If the input number is negative, return two empty strings.
5. [5] Create a **palindrome** function which checks if a given number is a palindrome.

Submit via RUConnected.