Assignment description

Assignment 1

Solve the following exercises. Any non-code part of the assignment must be handed in as comments in the source code.

You write multiline comments between `(*` and `*)` and single-line comments using `//`.

Some of the exercises (which are marked) are copied more or less verbatim from the course book, for your convenience, possibly with minor reformulations.

Use the template Assignment1.fs and upload that file. There are a few important things to note.

- The module title must be Assignment1 or the project will not compile
- All functions have a placeholder with a `failwith "not completed"`. This placeholder must be kept for all of the assignments that you have not solved or the project will not compile.

Green Exercises

Exercise 1.1

Write a function `sqr : int -> int` that given an integer `x` returns `x` squared

Exercise 1.2

Write a function `pow : float -> float -> float` that given two floating point numbers `x` and `n` returns `x` to the power of `n`

Hint: Use the library function: `System.Math.Pow`

Exercise 1.3 (based on 1.4 from HR)

Write a recursive function `sum : int -> int` such that given an integer `n` such that `n` $\geq$ `0` returns the sum of all integers from `0` to `n` inclusive.

```
sum n = 0 + 1 + 2 + 3 + ... + n
```

Hint: Use two clauses with `0` and `n` as patterns.

Exercise 1.4 (based on 1.5 from HR)

The sequence $F_0, F_1, F_2, \ldots$ of Fibonacci numbers is defined by:

$$\begin{aligned} F_0 &= 0 \\ F_1 &= 1 \\ F_n &= F_{n-1} + F_{n-2} \end{aligned}$$

Thus, the first members of the sequence are $0, 1, 1, 2, 3, 5, 8, 13, \ldots$.

Write a function `fib : int -> int` that given an integer `n` such that `n` $\geq$ `0` computes F_n. Use a declaration with three clauses, where the patterns correspond to the three cases of the above definition.

Exercise 1.5

Write a function `dup : string -> string` that given a string `s` concatenates `s` with itself. You can either use `+` to concatenate strings.

Example: `dup "Hi " = "Hi Hi "`

Exercise 1.6

Write a function `dupn : string -> int -> string` that given a string `s` and an integer `n` concatenates `s` with itself `n` times.

Example: `dupn "Hi " 3 = "Hi Hi Hi "`

Exercise 1.7 (based on 2.8 from HR)

The following figure gives the first part of Pascal's triangle

$$1$$
$$1\ 1$$
$$1\ 2\ 1$$
$$1\ 3\ 3\ 1$$
$$1\ 4\ 6\ 4\ 1$$

The entries of the triangle are called binomial coefficients. The k'th binomial coefficient of the n'th row is denoted $\binom{n}{k}$, for $n \geq 0$ and $k \leq n$. For example, $\binom{2}{1} = 2$ and $\binom{4}{2} = 6$ (note that we start counting at 0). The first and last binomial coefficients, that is, $\binom{n}{0}$ and $\binom{n}{n}$, of row n are both 1. Binomial coefficient inside a row is the sum of the two binomial coefficients immediately above it. These properties can be expressed as follows:

$$\binom{n}{0} = \binom{n}{n} = 1$$

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k} \text{ if } n \neq 0, k \neq 0, \text{and } n > k$$

Declare a function `bin : int * int -> int` that given a pair (n, k) computes $\binom{n}{k}$.

Extra-curicular activities

There is an interesting video on fractal-like properties that are found when looking at even and odd numbers of Pascal's triangle.

YellowExercises

Exercise 1.8

Assume the time of day is represented as a pair `(hh, mm) : int * int` where `hh` represents the hour (a number betwen `0` and `23`) and `mm` represents the minutes (a number between `0` and `59`).

Write a function `timediff : int * int -> int * int->int` so that `timediff t1 t2` computes the difference in minutes between `t1` and `t2`, i.e., `t2-t1`.

Examples:

```
> timediff (12,34) (11, 35);;
val it : int = -59

> timediff (12,34) (13, 35);;
val it : int = 61
```

Exercise 1.9

Write a function `minutes : int * int -> int` that computes the number of minutes since midnight.

Hint: This is easily done using the function timediff.

Examples:

```
> minutes (14,24);;
val it : int = 864

> minutes (23,1);;
val it : int = 1381
```

Exercise 1.10 (based on 2.13 from HR)

Write declarations for the functions `curry` and `uncurry`, which have the types

```
curry : ('a * 'b -> 'c) -> 'a -> 'b -> 'c

uncurry : ('a -> 'b -> 'c) -> 'a * 'b -> 'c
```

and are defined as follows:

- The `curry` function takes a function `f` of type `'a * 'b -> 'c` as its first argument. This function `f` takes a single tuple `(a, b)`, where `a` and `b` have the types `'a` and `'b` respectively, and returns a value of type `'c`. The `curry` function then returns a function of type `'a -> 'b -> 'c` that given two arguments `a` and `b`, applies them to `f` and returns the result.

- The `uncurry` does the reverse. It takes a function `f` of type `'a -> 'b -> 'c` as its first argument. This function `f` takes two arguments `a` and `b`, that have the types `'a` and `'b` respectively, and returns a value of type `'c`. The `uncurry` function then returns a function of type `'a * 'b -> 'c` that given a tuple `(a, b)` takes `a` and `b`, applies them to `f`, and returns the result.

More formally,

- `uncury (curry f) (x, y) = f (x, y)`
- `curry (uncury f) x y = f x y`

Write definitions of `curry` and `uncurry`.

Examples:

```
> curry (fun (x, y) -> x + y) 5 3;;
val it : int = 8

> uncurry (fun x y -> x + y) (5, 3);;
val it : int = 8
```

Scrabble assignments

If you have not done so already, make sure that you have read and understood the rules of Scrabble. It is not strictly necessary to solve this assignment, but it helps to have a mental image of what we are implementing.

We are now going to get started with the Scrabble project as well as getting familiar with the *first-class* nature of functions in functional programming languages. Before we start, we will use the following terminology for the duration of the project:

- A *tile* is a piece that you place on the board. It contains a character and a point value. The pair `('H', 4)` or `('A', 1)` are, for instance, two examples of tiles where the first element denotes a charater and the second value its point value. We currently do not consider wild card tiles but they will be covered later.
- A *square* is one element of the grid of the board that helps calculate the points of words that are placed over it. Two examples of squares on a standard Scrabble board are *double letter score* or *tripple word score* but we will have the oportunity to make our own. All empty squares on the board can be thought of as *single letter score*.

This week we focus on two things:

1. representing words as a sequence of tiles
2. representing squares as functions that compute point values

As we progress through the course this design will change slightly, but the principles that we cover here will remain the same.

Sequences of tiles

From your previous courses you are familiar with several ways to handle collections of data (Linked lists, arrays, maps, etc). F# has all of these. Today we will focus on representing words as functions of type `int -> char * int` that given an integer position `p` returns a character and its point value at that position. This solution is not entirely optimal.

- All words have infinite length (up to 32-bit integers anyway) and a default value needs to be used outside the intended length of the word.
- The length of the word is not made clear from the representation while structures like lists and arrays have their lengths readily available.
- Lookup times can be absolutely anything depending on the complexity of the function.

We will optimise this in future exercises.

Assignment 1.11

Create a function `empty : char * int -> (int -> char * int)` (note that the parentheses are strictly not necessary for the type but help with readibilty in this case) that given a default value `def` returns a function that given any integer returns `def`.

Hint: Here are four different correct ways that you can start the `empty` function

```
let empty (letter, pointValue) = fun pos -> ...
let empty (letter, pointValue) pos = ...

let empty def = fun pos -> ...
let empty def pos = ...
```

Examples:

```
// Explicit typing information will, depending on your definition of empty
// most likely be necessary here.
> let theLetterA : int -> char * int = empty ('A', 1);;
- val theLetterA : (int -> char * int)

> theLetterA 0;;
- val it : char * int = ('A', 1)

> theLetterA 42;;
- val it : char * int = ('A', 1)

> theLetterA -762;;
- val it : char * int = ('A', 1)
```

Red Exercises

Assignment 1.12

Create a function `add : int -> (char * int) -> (int -> char * int) -> (int -> char * int)` that given a position `newPos`, a pair containing a character and a point value `cv`, and a word represented by the function `word` returns another function that behaves exactly like `word` except that it returns `cv` when its position argument is equal to `newPos`. Phrased another way the returned function should, given a position `pos`, return `cv` if `pos` = `newpos` and `word pos` otherwise.

Hint: Here are two different correct ways that you can start the `add` function

```
let add newPos cv word = fun pos -> ...
let add newPos cv word pos = ...
```

Examples:

```
> let theLettersAB = add 1 ('B', 3) theLetterA;;
- val theLettersAB : (int -> char * int)

> theLettersAB 0;;
- val it : char * int = ('A', 1)

> theLettersAB 1;;
- val it : char * int = ('B', 3)

> theLettersAB 42;;
- val it : char * int = ('A', 1)
```

Assignment 1.13

Use the functions `empty` and `add` to create the function `hello : int -> char * int` that spells HELLO starting at index 0 and ending at index 4. Use `(char 0, 0)` as the default value. The letter H is worth four points and all other letters are worth one point.

Hint: By far the easiest way to do this is by using the piping operator `|>` as described in HR 2.11.

Creating squares

The squares on the board help calculate the number of points players get when placing words over them. Modelling them in a good way is non-trivial and we start here with a few simple cases. For this exercise we focus on squares that give points for single letters of words only (double letter score for instance) and next week we move on to squares that give points for the entire word (like double word score). We also keep our representation reasonably general in order to be able to expand to more esoteric types of squares down the line (squares that give negative points for vowels for instance).

Assignment 1.14

For this exercise, squares are functions and have the type `(int -> char * int) -> int -> int`. They take a word function `word`, the position in the word of the tile that is placed on the square `pos`, and returns the number of points you get for that square.

Create the square functions `singleLetterScore`, `doubleLetterScore` and `trippleLetterScore` that all have the type `(int -> char * int) -> int -> int` such that:

- `singleLetterScore` returns the point value of the tile placed on the square
- `doubleLetterScore` returns twice the point value of the tile placed on the square
- `trippleLetterScore` returns thrice the point value of the tile placed on the square

Examples:

```
> singleLetterScore hello 4;;
- val it : int = 1

> doubleLetterScore hello 4;;
- val it : int = 2

> trippleLetterScore hello 4;;
- val it : int = 3
```

Next week we will cover more types of squares and alternative ways to represent our words. The overall structure and the general intuition