Lab 3
Due: June 2, 2023 at 11:59PM

1 Lab 3 Specification

1.1 Overview

Relational database operations common to the SQL language supported by Access, MySQL, Oracle, and many others are based upon the standard mathematical operations on sets of *union*, *intersection*, *difference* and *Cartesian product*. Other operations specifically for manipulating relational databases are *select*, *project* and *join*. For example, the standard `SELECT` operation serves to filter elements that meet some specified criteria. In this lab and the upcoming assignment, we will implement a simple relational database using these set-theoretic operations.

Below is a definition of the user-defined `SET` data type to support a simple database. The sets are defined as lists where `I` is an `int list`, `S` is a `string list`, etc..

The `SET` type and the types defined in it are custom types, so the value expression `I [1;2;3]` *constructs* the custom type, based on its definition as an `int list`. Similarly, you can *deconstruct* a custom type with a pattern match like `I myVar` – this would bind `myVar` to `[1;2;3]`.

You can read the `SET` type definition as similar to the `|` usage in `match` statements: the type `SET` has one of the possible subtypes listed on the `|` lines.

In this lab, you will implement the Cartesian product: given two SETs, it creates all combinations of pairs that the inputs can create. For example, the product of I×S, produces an (int, string) tuple list. The naming of the data types hints at tuples definition: SI is a list of (String, Int) tuples, the result of S×I. SIIS is a list of ((String, Int),(Int,String)), the result of (S×I)×(I×S), etc. Below are all required basic set definitions in F#:

```
type SET =
    | I of int list                                   // I [1;2;3]
    | S of string list                                // S ["a";"b";"c"]
    | IS of (int * string) list                       // IS [(1, "a");(2, "b")]
    | II of (int * int) list                          // II [(1,2); (3,4); (5,6)]
    | SS of (string * string) list                    // SS [("a","b"); ("c","d")]
    | SI of (string * int) list                       // SI [("a", 1); ("b", 2); ("c", 3)]
    | SISI of ((string * int) * (string * int)) list   // SISI [(("a", 1), ("b", 2)); (("c", 3), "d", 4))]
    | SIIS of ((string * int) * (int * string)) list;; // SIIS [(("a", 1), (2, "b")); (("c", 3), (4, "d"))]
```

1.2 Implementation

The skeleton code at `https://classroom.github.com/a/8XA9-j9P` is the start of this week's lab. Write and test the following functions, along with any necessary helper functions.

1.2.1 pairs

Define a `pairs` function that generates all possible pairs of elements from two input lists, as tuples. You can assume that both of the lists are non-empty. The signature should be:

```
> pairs;;
val it : ('a list -> 'b list -> ('a * 'b) list)
```

Example results, showing all possible combinations of pairs:

```
> pairs [1000;1050] ["CHEM";"MATH"];;
val it : (int * string) list =
  [(1000, "CHEM"); (1000, "MATH"); (1050, "CHEM"); (1050, "MATH")]
```

Hint: Write a helper function to distribute one element across every element of a list: `dist 3 ["a"; "b"; "c"];;` returns `[(3, "a"); (3, "b"); (3, "c")]`

1.2.2 product

The Cartesian product produces a set as a tuple list of two sets. Some example patterns are below.

```
let product s1 s2 =
  match (s1, s2) with
    | (I s1, I s2) -> II (pairs s1 s2)
    | (S s1, S s2) -> SS (pairs s1 s2)
    | (I s1, S s2) -> IS (pairs s1 s2);;
```

Some example `product` runs:

```
> let i2 = I [1000; 1050];;
> let s2 = S ["CHEM"; "MATH"];;
> product i2 s2;; // IxS
val it : SET = IS [(1000, "CHEM"); (1000, "MATH"); (1050, "CHEM"); (1050, "MATH"
    )]

> product s2 i2;; // SxI
val it : SET = SI [("CHEM", 1000); ("CHEM", 1050); ("MATH", 1000); ("MATH",
    1050)]
```

Starting with the definition above, **complete defining the `product` for the three remaining `SET` cases: `SI`, `SIIS`, and `SISI`**. (I and S are not possible product results. You will get a warning about incomplete matches that you can ignore!)

You can test `product` using the following small database, or build your own. Some products can get quite long.

```
let i1 = I [1111;2222;3333];;
let i2 = I [5555; 6666];;
let s1 = S ["COMP"; "HIST"; "PHYS"];;
let s2 = S ["CHEM"; "MATH"];;

let is = product i1 s1;;
let si1 = product s1 i1;;
let si2 = product s2 i2;;
let sisi = product si1 si2;;
let siis1 = product si1 is;;
let siis2 = product si2 is;;
```

2 Grading

Be professional. Make results easy to understand and grade. Include only those parts to be graded. Leave comments where necessary, especially if it aids in grading.

Each of the 2 functions is worth $\frac{1}{2}$ of the lab grade.