COMP 212 Spring 2025
Lab 10

In this lab, we will look at some ways to read data from user input, either things the user types in or from files. First, we need a couple of new datatypes.

1 Unit

The type `unit` represents an "empty tuple", and has value `()`. It is useful for functions that do their work imperatively (by updating things) rather than functionally (by creating new values).

2 Input and output

In this lab, you will use functions from the `TextIO` library; see https://www.cs.princeton.edu/~appel/smlnj/basis/text-io.html.

The most basic functions are

- `TextIO.print : string -> unit` prints text to the terminal
- `TextIO.inputLineIn : unit -> string` reads a line of input from the terminal, and returns the input text without the final newline character (this is a wrapper around some of the official library functions that you can find at the top of `lab10.sml`).

Task 2.1 Run the following in SMLNJ:

```
- TextIO.print ("hello world");
- TextIO.print ("hello world\n");
- TextIO.inputLineIn();
[then type something and press enter]
```

Task 2.2 Write a function that prints out "Please enter a word", then reads a word as a line of input that the user types, and then prints out the same word with an 's' at the end. For example,

```
Please enter a word:
apple
The plural version of apple is apples
```

2.1 Text Input/Output from the Terminal

Next we will look at the more general versions of printing and reading input.

The types `TextIO.instream` and `TextIO.outstream` represent "something you can read from" and "something you can write to", respectively.

Here are some input and output streams for reading from/writing to the terminal:

- `TextIO.stdIn : TextIO.instream` ("standard input") reads input you type in the terminal
- `TextIO.stdOut : TextIO.outstream` ("standard output") writes output to the terminal

Here are some functions for reading and writing:

- `TextIO.inputLine : TextIO.instream -> string option` read a line of input, returning `NONE` if no further input is available, or `SOME(input)` if a line of input was available. This was used in the controller code for the shopping cart problem, for example.
- `TextIO.output : TextIO.outstream * string -> unit` write a string to the given output stream.

Unlike all of the functions we have seen so far, `inputLine` and `output` *change* the provided input stream and output stream — by requesting data from the user, by making text appear on the screen, or (using the streams we'll use later in the lab) reading/writing files.

Task 2.3 In `smlnj`, try out these functions, using them to read and write from the terminal: what do the following do?

- ```
  TextIO.output (TextIO.stdOut, "hello world")
  ```

  One place where you have seen `output` before is the function `print s` (used in the tester functions all semester), which is defined to be `TextIO.output(TextIO.stdOut, s)`.

- ```
  let val () = TextIO.output (TextIO.stdOut, "hello")
      val () = TextIO.output (TextIO.stdOut, "world")
  in () end
  ```

- ```
  TextIO.inputLine TextIO.stdIn
  ```

  Note: you have to type some text and then press enter for the `inputLine` to proceed.

- ```
  val a = TextIO.inputLine TextIO.stdIn;
  val b = TextIO.inputLine TextIO.stdIn;
  ```

  Explain what is unusual about this.

Task 2.4 Write a function

```
val copy : TextIO.instream * TextIO.outstream -> unit
```

that copies the entire input stream to the output stream. Try it out interactively:

```
- copy (TextIO.stdIn, TextIO.stdOut);
hi there      [you type this and press enter]
hi there      [it prints this]
how are you   [you type this and press enter]
how are you   [it prints this]
[waiting for more input]
```

You can use `Control-c` to stop the loop from running.

Have us check your work before proceeding!

2.2 Text Input/Output from Files

The following functions create input and output streams from files; the argument is the file name:

- `TextIO.openIn : string -> TextIO.instream`
- `TextIO.openOut : string -> TextIO.outstream`
  WARNING: overwrites the file specified by the file name

Task 2.5 Write a function

```
val copy_files : string * string -> unit
```

that takes two filenames and copies the contents of the first to the second.

Task 2.6 Try this out on some file. **Make sure your file has more than one line, and that they are all copied.**

Have us check your work before proceeding!

3 Mapreduce on a file

The signature

```
signature MAP_REDUCE =
sig
    type 'a mapreducable
    val mapreduce :
           ('a -> 'b)
        * 'b
        * ('b * 'b -> 'b)
        * 'a mapreducable -> 'b
end
```

represents a data source that we can do a map-reduce on.

We can implement this signature using a `TextIO.instream` (which can stand for a file or for the terminal). However, to think of a file or the terminal as an `'a mapreducable` for some specific type `'a`, we need to have a way to convert lines of the file into `'a`'s. Thus, we say that the type

```
TextIO.instream * (string -> 'a)
```

is map-reducable, where the second component of the pair is used to parse each **line** of the file into a piece of data.

Task 3.1 Implement the `mapreduce` function in `FileMR`. **Your implementation should not use any space beyond what is necessary to store the `'b` values that are produced—in particular, it should not first read the file as a sequence.**

Have us check your work before proceeding!

Task 3.2 Make a value

```
val numbersFromStdIn : int FileMR.mapreducable
```

that reads numbers from `TextIO.stdIn`; you can use the provided `intFromString'` function.

Task 3.3 Write a function

```
val add : int FileMR.mapreducable -> int
```

that adds the numbers in an `int mapreducable`.

Task 3.4 Test this on `numbersFromStdIn`. Note that you will need to type `Control-d` to signal the end of input, which will also (unfortunately) quit SMLNJ.

Task 3.5 Make an `int FileMR.mapreducable` that reads from a file and test your function on a file too.

Have us check your work!