COMP 212 Fall 2024
Lab 6

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 Options

The type `'a option` is defined as follows:

```
datatype 'a option =
    NONE
  | SOME of 'a
```

For example, a value of type `int option` is either `NONE`, or `SOME 0`, or `SOME 1`, or `SOME(17)`, etc. Options are often used to signal that a function may or may not successfully return a value. For example, the function `Int.fromString : string -> int option` attempts to read the first characters from a string as an integer, and if successful returns `SOME(n)` with the integer it found, or if not returns `NONE`.

Task 1.1 Try the following examples in SMLNJ:

- `Int.fromString("123")`
- `Int.fromString("a123")`
- `Int.fromString("123,456")`
- `Int.fromString("123.22")`
- `Real.fromString("123.22")`
- `Real.fromString("the number 123.22")`

In code that calls a function that returns an option, you will typically use case analysis to distinguish `NONE` and `SOME`

```
case (v : 'a option) of
   NONE => e0
|  SOME x => e1 [ x has type 'a here ]
```

2 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).

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

3.1 Text Input/Output from the Terminal

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 3.1 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 3.2 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!

3.2 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 3.3 Write a function

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

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

Task 3.4 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!

4 Reading a list

Task 4.1 Write a function `int_file_to_list : string -> int list` that takes the name of a file (as a string) and produces a list of all of the integers in that file, assuming that each integer is at the start of a separate line of the file. For examples, if the file `nums.txt` contains

```
1
12
33
a
4
```

then `int_file_to_list("nums.txt")` should return `[1,12,33,4]`.

Task 4.2 Write a function `float_file_to_list : string -> int list` that takes the name of a file (as a string) and produces a list of all of the floats in that file, assuming that each floating point number is at the start of a separate line of the file. For examples, if the file `nums2.txt` contains

```
1
12.22
33.12
a
4.444
```

then `int_file_to_list("nums2.txt")` should return `[1.0,12.22,33.12,4.444]`. (What does `int_file_to_list` return for `nums2.txt`?)

Task 4.3 Write a higher-order polymorphic function that generalizes the previous two tasks, avoiding repeated code.

Have us check your work!