F28PL - Programming Languages

Question F (Strings and chars)

Tommy Lamb

H00217505

The plain text file which accompanies this PDF contains a listing of all functions alongside test function calls, with some formatting to aid readability. It can be opened and executed in Poly/ML from the Terminal by navigating to the save location of the file, opening Poly/ML, and using the command : ' use "F28PL - Tommy Lamb - Question F - Code.txt"; '. This method will cause Poly/ML to ignore the document formatting and only display system output (not input). To maintain formatting, all text can be copied and pasted into a running instance of Poly/ML. For best results it should be copied/pasted in small blocks, around 1 question in size.

Note:

Certain functions utilise other functions defined in earlier questions. Any such prerequisites are noted in blue for each applicable question.

1. Write a function to remove all the non-letters off the front of a list of characters:

   `strip [#".",#" ",#"t#,#"h",#"e"] ==> [#"t",#"h",#"e"].`

   - If the list is empty, return the empty list.
   - If the list starts with a letter, return the list.
   - Otherwise remove all the non-letters from the start of the tail of the list.

```
fun strip [] = []
  | strip (head::tail) =
      if Char.isAlpha head then
            (head::tail)
      else
            strip tail;
```

This function simply checks if the head of a list is a letter (upper or lower case), before either returning the list if true, or checking recursively the rest of the list whilst discarding the current head element.

2. Write a function to take the next word off a list of characters. The function returns a tuple of the word as a list of characters and the rest of the list.

```
next [#"t",#"h",#"e",#" ",#"c",#"a",#"t",#" "…] ==>
([#"t",#"h",#"e"],[#" ",#"c",#"a",#"t",#" " ...]) : char list * char list.
```

- If the list is empty, return a tuple of two empty lists.
- If the list starts with a non-letter, return a tuple of the empty list and the list.
- Otherwise, take the rest of the next word from the tail of the list returning a tuple of the rest of the next word and the rest of the tail of the list, put the head of the original list, which is the head of the word, onto the front of the rest of the word, and return the whole word list and the rest of the tail of the list.
  Hint: use a let with a tuple pattern to match the tuple for the rest of word and rest of tail of list.

```
fun next [] = ([],[])
  | next (head::tail) =
      if Char.isAlpha head then
            let
                  val (word,list) = next tail
            in
                  (head::word,list)
            end
      else
            ([],(head::tail));
```

Acting recursively this function checks if the head of a list is a letter, and if so, pulls it from the list and concatenates it with the result of successive calls to return in a tuple. The returned tuple is a `char list` representing the first sequence of concurrent letters, alongside the rest of the initial list. In either base case - the list is empty, or the head is not a letter - an empty list is returned in the place of the head element. It is on to this empty list that the heads of each preceding call are concatenated as control moves up the stack.

A local declaration is used to manipulate the returned tuple of a successive call into a more useable format, allowing the concatenation to be achieved easily and the return tuple values properly formatted. Unfortunately this methodology runs counter to the idea of tail recursion, as the result of each recursive call is needed before evaluating the final return values.

3. Write a function to turn a list of characters into a list of individual words as strings.

`words [#"t",#"h",#"e",#" ",…] ==> ["the","cat","sat","on","the","mat"] : string list`

- If the list is empty, return an empty list
- Otherwise, get the first word from the stripped list as a list of characters and the rest of the list, get the list of words from the rest of list and put the imploded first word on the front of the word list. Hint: use a local definition with a tuple pattern to match the first word and rest of list

```
fun words [] = []
  | words list =
      let
            val (word,tail) = next(strip(list))
      in
            [implode word]@(words tail)
      end;
```

Questions 1 & 2 are prerequisites for this function.

Using a local declaration, this function uses the previously defined functions `strip` (Q1) and `next` (Q2) to get two `char list` variables representing the first word of the argument list and the remainder of that list. Using the inbuilt `implode` method to convert `char list` to `string`, this first word is concatenated to result of a recursive call with the remainder of the original argument list as the argument.

Put simply, this function:

- Takes a list, `list`.
- Strips any leading punctuation from it.
- Separates the first sequence of concurrent letters from the list.
- Converts this sequence to a string.
- And repeats recursively until the list is exhausted (empty).

Finally returning a list containing all of the strings created in each recursive call.

This implementation also introduces an empty string `""` at the end of the list. This is thought to be because at some point the function `next` is called on the empty list, and returns a tuple of two empty lists. This empty list is then passed to `implode word`, which returns `""`. This being said, the base case pattern matching should handle it, but seemingly doesn't.

4. Write a function to increment the count for a word in a list of words and counts.

```
incCount "sat" [("cat",1),("the",1)]
==> [("cat",1),("sat",1),("the",1)] : (string * int) list

incCount "the" [("cat",1),("mat",1),("on",1),("sat",1),("the",1)]
==> [("cat",1),("mat",1),("on",1),("sat",1),("the",2)] : (string * int) list
```

- To increment the count for a word in an empty list, create a list with a tuple for the word and a count of 1.
- To increment the count for a word in a list, if the word is the word in the first tuple, return the list with the first tuple's count incremented.
- Otherwise if the word comes before the word in the first tuple in the list, add a new tuple for the word and a count of 1 on the front of the list.
- Otherwise, put the head of the list on the front of the result of incrementing the count for the word in the tail of the list.

```
fun incCount (word:string) [] = [(word,1)]
  | incCount word (head::tail) =
      if #1 head = word then
            (word,(#2 head)+1)::tail
      else if word < (#1 head) then
            (word,1)::head::tail
      else
            head::(incCount word tail);
```

This function implements a linear search and modification/insertion algorithm over a rudimentary Map data structure (in the form of `(string*int) list`). The three base cases in this function are where:

- The Map list is empty - `(word:string) [] = [(word,1)]`
  Either the recursive search didn't find the key, or the initial map argument was empty.
  Regardless, return the new entry in a new list for addition to the map (via concatenation in preceding calls).
- The first map entry has the required key - `#1 head = word`
  The key of the first entry of the map matches that being searched for, thus the value of this entry is incremented by 1.
- The search key does not feature later in the map - `word < (#1 head)`
  If the `string` key appears alphabetically before that of the first map element, then the key will not be found later in the map. Thus the new entry is inserted in this position in the map using concatenation.

The step case is taken when no base cases hold true, and makes a recursive call to apply the base case checking to each successive entry in the map. Eventually a base case is met, whereby the necessary action is taken and concatenation here allows the full map to be returned, as is also true of the base cases themselves.

5. Write a function which given a list of words and an empty list, constructs the frequency count list.

```
counts ["the","cat","sat","on","the","mat"] [] ==>
[("cat",1),(mat",1),("on",1),("sat",1),("the",2)];

fun counts [] list = list
  | counts (head::tail) list = counts (tail) (incCount head list);
```

Question 4 is a prerequisite for this function.

Using an accumulator variable `list` and tail recursion, this function iterates through a `string list` and makes a call on the previously defined function `incCount` (Q4). The use of `list` allows the efficiency of tail recursion whilst avoiding messy concatenation of each call to `incCount`. In the base case it is this accumulation variable which is returned, which is simply the accumulated return list of each call to `incCount`.

6. Write a function which given a file name, opens the file, inputs the whole file as a string, explodes the string, makes the word count list, closes the file and returns the word count list.

```
fun Q6 file =
	let
	val filestream = (TextIO.openIn(file))
	in
		(counts(words(explode(TextIO.inputAll(filestream)))) [])
		before
		(TextIO.closeIn(filestream))
	end;
```

Questions 1, 2, 3, 4, & 5 are all prerequisite.
Questions 3 & 5 are prerequisites for this function.
Questions 1 & 2 are prerequisites for Question 3.
Question 4 is a prerequisite for Question 5.

The stages of execution of this function is best described using a list.

1. `val filestream = (TextIO.openIn(file))`
   Opens an IO input stream of type `instream` and sets a variable `filestream` to it.
   The type of `TextIO.openIn` is `string -> instream`, where `string` is a file name (or path?) referring to a file.

2. `TextIO.inputAll(filestream)`
   Reads in everything from the file (input stream) as a single string. The exact format of the returned string is not known, with regard to control characters and the like.

3. `explode(TextIO.inputAll(filestream))`
   Takes the returned `string`, and explodes it into a `char list`.

4. `words(explode(TextIO.inputAll(filestream)))`
   Using `words` from Q3, converts the `char list` to `string list` of words in the file.

5. `(counts(words(explode(TextIO.inputAll(filestream)))) [])`
   Uses `counts` from Q5 to count the occurrences of each word in the `string list`. The empty list is passed as the initial value for the accumulator variable used by `counts`.

6. `before`
   See below

7. `(TextIO.closeIn(filestream))`
   Closes the input stream `filestream`, releasing system resources and cleaning up after itself.

The inbuilt infix function `before` has type `'a * 'b -> 'a` and is a concise way of first evaluating `'a`, then `'b`, discarding the result of evaluating `'b`, and returning the result from `'a`. It is simply used here to ensure order of evaluation and provide a neat way of executing the stream closing without caring about the `unit` it returns. The local declaration was utilised to ensure that the correct stream was closed. There was some concern that another call to `TextIO.openIn(file)` would create another input stream, which would be closed instead and leave the original behind.

As this function depends on Q3, it too will have an errant `""` included in its result.

This function was developed with close reference to the SML Basis library, in particular:

- http://sml-family.org/Basis/text-io.html#SIG:TEXT_IO.openIn:VAL
  For `TextIO.openIn`
- http://sml-family.org/Basis/imperative-io.html#SIG:IMPERATIVE_IO.inputAll:VAL
  For `TextIO.inputAll`
- http://sml-family.org/Basis/general.html
  For `before`

While the function `inputAll` cannot be found the `TextIO` structure, it is defined as part of the `ImperativeIO` structure, which is 'inherited' by `TextIO`. Interestingly some comments at the top of the `TextIO` signature document suggest that the structure is not technically valid SML, due to some issues around this inheritance - though the terminology is unfamiliar.

Regardless it was possible to smash these particular rocks together thanks to SML's dictatorial type system and some trial-and-error: if the types match, you're probably doing something right. Curiously, while the `Imperative_IO` signature dictates that `inputAll` has type `instream -> vector`, `TextIO.inputAll` has type `TextIO.instream -> string`. This is believed to arise as `Imperative_IO` exists only as a *signature* but not a *structure*. The context and usage of these terms suggests that *signatures* are roughly equivalent to Java Interfaces and *structures* to Java Classes, presenting the unexpected result that SML may have the concept of classes, inheritance, and so to a limited extent, Object-Orientated principles. Since `TextIO` 'implements' `Imperative_IO`, this apparent discrepancy of type could be explained by `string` being a sub-class of `vector`; sub-type may prove more in fitting with SML terminology.

Continuing the Java parallels, SML hides much of the I/O work in a series of inbuilt functions, which are wrapped around one another to achieve an easy-to-use I/O interface for the programmer. This also contributed to making constructing the function surprisingly easy, despite the wafer thin explanation of the functions' exact operations.

As a note on usage, it appears that *structures* are named in Pascal case, like `TextIO` and `Char`, whilst *signatures* are presented in block capitals: `TEXT_IO` and `CHAR`.

7. Write a function which given a word count list displays it on standard output as a table of words and counts.

```
fun Q7 [] = ()
  | Q7 ((head:string*int)::tail) =
      TextIO.output(TextIO.stdOut,((#1 head) ^ "\t" ^ Int.toString(#2 head) ^ "\r\n"))
      before
      (Q7 tail);
```

Reusing the list format:

1. `((head:string*int)::tail)`
   Pattern matches on the list to access the head element of the list, with some necessary type casting (see below).

2. `((#1 head) ^ "\t" ^ Int.toString(#2 head) ^ "\r\n")`
   Concatenates the following together, in order:
   a. The first element of the head tuple, the "word" value: `(#1 head)`
   b. A tab: `"\t"`
   c. The second element of the head tuple, the "word count" value, parsed to `string`: `Int.toString(#2 head)`
   d. A Windows new line character sequence (as it works for both Linux and Windows): `"\r\n"`

3. `TextIO.output(TextIO.stdOut,((#1 head) ^ "\t" ^ Int.toString(#2 head) ^ "\r\n"))`
   Outputs the string `result` of the concatenation to standard output, represented by inbuilt value: `TextIO.stdOut`

4. `before`

5. `(Q7 tail)`
   Recursively calls this function to iterate through the entire word count list.

As previously, the `before` function is used to discard the result of the recursive call and ease enforcement of the order of execution; since the only result of the function is to print to standard output, the end return value is irrelevant.

The type declaration for `head` is required for the function to pass muster with the compiler. Without it, the compiler complains about `head` not being a "fixed record type", or in other words, a tuple. This declares it to be a tuple, which keeps the baying hounds of SML's typing docile.

The reason the output stream is not closed is because attempting to close the standard output may raise an exception, or have undesirable consequences. All I/O streams are closed by Poly/ML upon exit anyway - or should be according to documentation.

Much like the previous question the SML Basis Library was closely consulted here, in particular:

- http://sml-family.org/Basis/text-io.html#SIG:TEXT_IO.stdOut:VAL
  For `TextIO.stdOut`
- http://sml-family.org/Basis/imperative-io.html#SIG:IMPERATIVE_IO.output:VAL
  For `TextIO.output`

Nothing of particular note was found during this research, being much like the previous question.