Name: ____

Solutions

1. (1 pt.)

   - **Read all material carefully.**
   - *If in doubt whether something is allowed, ask, don't assume.*
   - You may refer to your **books, papers, and notes** during this test.
   - **E-books** may be used *subject to the restrictions* noted in class.
   - **Computers** (including smart phones, tablets, etc.) **are not permitted**, except when used strictly as e-books or for viewing ones own notes.
   - **Network access** of any kind (cell, voice, text, data, ...) is **not permitted**.
   - Write, and draw, carefully. **Ambiguous or cryptic answers receive zero credit.**
   - Use **class and textbook conventions** for notation, algorithmic options, etc.
   - **Do not attach or remove any pages.**

   Write your name in the space provided above.

2. (10 pts.) For each of the following *Standard ML* expressions, provide the response when that expression is evaluated by the `sml` REPL (read-eval-print loop). Assume that the expressions are evaluated in the order listed. In your response, *draw a **box** around the **type** and `oval` around the **value**.* (If there is an error then clearly explain the error.)

   (a) (2 pts.) `42.42 / 2.0;` Ⓐ *`val it = 21.21 : real`*

   (b) (2 pts.) `"My name is nil.";` Ⓐ *`val it = "My name is nil." : string`*

   (c) (2 pts.) `42 / 2;` Ⓐ *Error because operator `/` requires `real` operands but the ones in the expression are `int`.*

   (d) (2 pts.) `fun f101(x) = x + 101;` Ⓐ *`val f101 = fn : int -> int`*

   (e) (2 pts.)

   ```
   fun f201 (nil) = nil
     | f201 (h::t) = h :: f201(t);
   ```

   Ⓐ *`val f201 = fn : 'a list -> 'a list`*

3. (4 pts.) Provide *Standard ML* expressions for each of the following.

   (a) (2 pts.) Bind the identifier `score` to the integer 42. Ⓐ *`val x = 42;`*

(b) (2 pts.) Multiply the integer bound to `score` by 2. Ⓐ *`x * 2;`*

4. (10 pts.)

   (a) Define a recursive function (of your choice) that is **not** *tail recursive*, using Standard ML.

   (b) Define another recursive function (also of your choice) that is *tail recursive*, using Standard ML.

   (c) **Explain your answers**.

Ⓐ *Here are two functions that compute the length of a list. The first version uses non-tail recursion that directly expresses a recursive definition of the length of a list: The length of an empty list is 0 (base case) while the length of a nonempty list is one plus the length of its tail (i.e., the list obtained by removing the head, or first element). This version is clearly recursive as the second case (non-base) invokes the function again. It is also not tail-recursive because when the recursive call returns, it's value is used in another operation (to add one to it) instead of being returned directly. The second version uses a helper function called* `lp` *which may be thought of as a function that walks down a list. The invariant is that when this function is invoked as* `lp`(L, p), *L is a suffix of* `lst` *and p is the length of the corresponding prefix of* `lst`. *Equivalently, p is the length of* `lst` *minus the length of* `L`. *In the definition, the outer function (*`lstlentr`*) is non-recursive. The inner function,* `lp` *is recursive in its second case (when the first argument is not* `nil`*). It is tail-recursive because the result of the recursive call is simply returned without any further operations on it.*

Ⓐ

```
    fun  lstlen (nil) = 0
    |    lstlen (hd :: tl) = 1 + lstlen (tl);
```

Ⓐ

```
     fun lstlentr (lst) =
         let
              fun  lp (nil, pfxlen) = pfxlen
                 |  lp (hd :: tl, pfxlen) = lp (tl, pfxlen + 1)
         in
               lp (lst, 0)
         end;
```

5. (10 pts.) Provide the *Standard ML* definition of a recursive function `f301` that takes a list of integers as argument and returns a similar list with each element incremented by 100. For instance, when invoked on the list `[3, 1, 4]`, the list `[103, 101, 104]` should be returned. Explain why your answer is correct. Trace the operation of your function on the list `[3, 1, 4]`.

Ⓐ

```
fun f301 (nil) = nil
  | f301 (hd :: tl) = (100 + hd) :: f301 (tl);
```

Ⓐ *The function `f301` above is a direct expression of a recursive definition of the list obtained by adding 100 to each element of a given list: If the given list is empty then the result is also empty; otherwise, the result is 100 plus the first item of the given list followed by (recursively) the same function applied to the rest of the list. Using f to denote `f301` for brevity, the call-stack grows as*
$f([3, 1, 4])$ *calls*
$f([1, 4])$ *which in turn calls*
$f([4])$ *which further calls*
$f($`nil`$)$ *which being the base case returns*
`nil` *to which is consed (added as first element)* $100 + 4$ *returning (call-stack shrinking)*
`[104]` *to which is consed* $100 + 1$ *returning*
`[101, 104]` *to which is consed* $100 + 3$ *returning*
`[103, 101, 104]` *at which point the initial call returns.*

6. (15 pts.) Provide a **complete JCoCo assembly language program** that

   (a) Reads two newline-terminated strings from *standard input*.

   (b) Writes the sum of the lengths of those two strings to *standard output*.

   (c) **Explain why your program is correct.**

Ⓐ

```
Function: main/0
Constants: ""
Globals: input, len, print
BEGIN
  LOAD_GLOBAL      2
  LOAD_GLOBAL      1
  LOAD_GLOBAL      0
  LOAD_CONST       0
  CALL_FUNCTION    1
  CALL_FUNCTION    1
  LOAD_GLOBAL      1
  LOAD_GLOBAL      0
  LOAD_CONST       0
  CALL_FUNCTION    1
  CALL_FUNCTION    1
  BINARY_ADD
  CALL_FUNCTION    1
  RETURN_VALUE
END
```

Ⓐ *The* op. stack *column depicts the state of the operand stack (with the top of stack on the left) immediately following the operation on the line listed in the previous column.*

line#	*op. stack*	*comment*
5	*(print)*	*Loads* `print` *function object onto stack.*
6	*(input print)*	*Ditto for* `len`*.*
7	*(input print)*	*Ditto for* `input`*.*
8	*(`""` input print)*	*Loads constant empty-string as prompt for input.*
9	*(`"Hello,"` print)*	*Invokes* input*; its result now TOS.*
10	*(6 print)*	*Invokes* len *on TOS; its result now TOS.*
11–15	*(7 6 print)*	*A repeat of lines 6–10, resulting in the length of the second input string (say, 7) now pushed onto stack.*
16	*(13 print)*	*result of TOS + TOS1 is now TOS.*
17	*(*None*)*	*Invokes* print*; its result now TOS.*
18	*(*None*)*	*Returns None as result of* main.

7. (15 pts.) Provide a **complete JCoCo assembly language program** that

   (a) Reads a newline-terminated string from *standard input*.

   (b) Writes this string to *standard output* but with all characters converted to upper case. (For instance, if the input string is `Hello, World!` then the output should be `HELLO, WORLD!`.) [Hint: A Python string object has a method `upper` that returns an upper-case version of that string.]

   (c) **Explain why your program is correct.**

Ⓐ

```
Function: main/0
Constants: ""
Globals: input, print, upper
BEGIN
  LOAD_GLOBAL     1
  LOAD_GLOBAL     0
  LOAD_CONST      0
  CALL_FUNCTION   1
  LOAD_ATTR       2
  CALL_FUNCTION   0
  CALL_FUNCTION   1
  RETURN_VALUE
END
```

Ⓐ *The following uses the conventions from the previous answer.*

line#	*op. stack*	*comment*
5	*(print)*	*Loads* `print` *function object onto stack.*
6	*(input print)*	*Ditto for* `input`*.*
7	*(*`""` *input print)*	*Loads constant empty-string as prompt for input.*
8	*(*`"saMPle"` *print)*	*Invokes* input*; its result now TOS.*
9	*(upper* `"saMPle"` *print)*	*Loads* upper *attribute (function) of TOS (string* `saMPle`*).*
10	*(*`"SAMPLE"` *print)*	*Invokes* upper*; its result now TOS.*
11	*(*None*)*	*Invokes* print*; its result now TOS.*
12	*(*None*)*	*Returns None as result of* main.

[JCoCo note: The current implementation does not seem to export the `upper` *method used above and so this code will not work with the* coco *interpreter.]*