```
(*
                              CS51 Lab 13
                   Procedural Programming and Loops
 *)

(*
Objective:

This lab introduces and provides practice with:
  - loops and procedural programming
  - tail recursion as an alternate form of iteration.
 *)

(*======================================================================
Part 1: Revisiting list operations

In Lab 2, you created recursive functions to find the lengths and sums
of lists. Below are examples of both:

    let rec length (lst : int list) : int =
      match lst with
      | [] -> 0
      | _hd :: tl -> 1 + length tl ;;

    let rec sum (lst : int list) : int =
      match lst with
      | [] -> 0
      | hd :: tl -> hd + sum tl ;;

As noted in the textbook, these functional recursive implementations
may run into stack overflow errors when called on exceptionally long
lists.

    # sum (List.init 1_000_000 Fun.id) ;;
    Stack overflow during evaluation (looping recursion?).

As computation proceeds, each recursive call of `length tl` or `sum
tl` is suspended, with the suspended computation stored on a stack,
until we reach the end of the list. At that point, the stack finally
unwinds and the expression is evaluated. If the number of calls grows
too large, we run out of room allocated to the stack of suspended
computations, and the computation fails with a `Stack_overflow`
exception.

Two ways of addressing this problem are (i) using a tail recursive
function or (ii) using an explicit loop.

In a tail recursive function the recursive invocation *is* the final
result of the invoking call. The value of the recursive function is
immediately returned; no further computation must be done to it, so no
suspended computation needs to be stored on the call stack, thus
avoiding the problem of stack overflow.

Below, the length function above has been rewritten to use a
tail-recursive auxiliary function `length_tr` (the "tr" stands for
"tail-recursive") to demonstrate this:

    let length lst =
      let rec length_tr lst accum =
        match lst with
        | [] -> accum
        | _hd :: tl -> length_tr tl (1 + accum) in
      length_tr lst 0 ;;

The technique used here, using a tail-recursive auxiliary function
```

```
that makes use of an added argument that acts as an accumulator for
the partial results, is a common one for converting functions to
tail-recursive form.

......................................................................
Exercise 1: Tail-recursive sum

Rewrite the `sum` function to be tail recursive. (As usual, for this
and succeeding exercises, you shouldn't feel beholden to how the
definition is introduced in the skeleton code below. For instance, you
might want to add a `rec`, or use a different argument list, or no
argument list at all but binding to an anonymous function instead.)

Verify your implementation is tail-recursive by summing up the
elements of an extremely long list, like this 1,000,000 element list:

    # sum (List.init 1_000_000 Fun.id) ;;
    - : int = 499999500000

Gauss would be proud!
....................................................................*)

let sum _ =
  failwith "sum not implemented" ;;

(*....................................................................
Exercise 2: Write a tail-recursive function `prods` that finds the
product of the corresponding elements of two integer lists. Your
function should raise a `Failure` exception when called on lists of
differing length. You may remember implementing a similar function in
Lab 2. It should work like this:

   # prods [1; 2; 3] [1; 2; 3] ;;
   -: int list = [1; 4; 9]

Your initial try at a tail-recursive function may output a list that
is in reverse order of your expected output. This is a common outcome
in tail-recursive list iteration functions. (In general, you'd want to
consider whether or not this has negative outcomes on your intended
use case. It may be that the output order is not significant.)

In this case, for testing purposes, please preserve the initial
ordering of the lists. One method to do so is simply to reverse the
list at the end, using a tail-recursive reversal function of
course. Fortunately, the built-in `List.rev` is tail-recursive (even
though the documentation doesn't mention that fact).
....................................................................*)

let prods _ =
  failwith "prods not implemented" ;;

(*....................................................................
Exercise 3: Modify your `prods` function to use option types to deal
with lists of different lengths. Call it `prods_opt`. The `prods_opt`
function should return `None` if its two list arguments are of
different lengths, and if the lists are of the same length, `Some lst`
where `lst` is the products of the corresponding elements of its
arguments.
....................................................................*)

let prods_opt _ =
  failwith "prods not implemented" ;;

(*....................................................................
Exercise 4: Finally, combine your `sum` and `prods` functions to
```

```
create a tail-recursive dot product function (that is, the sum of the
products of corresponding elements of the lists). (For reference, you
implemented dot product in lab 2.)
......................................................................*)

let dotprod _ =
  failwith "dotprod not implemented" ;;

(*======================================================================
Part 2: Loops

Another method of solving the problem of stack overflow when dealing
with large lists is to use loops. Unlike tail recursion, loops rely on
state change in order to function, and therefore go beyond the now
familiar purely functional paradigm, bringing us into the domain of
procedural programming.

Let's get some practice with simple loops.

.......................................................................
Exercise 5: Write two non-recursive functions, `odds_while` and
`odds_for`, that use `while` and `for` loops, respectively, to return
a list of all positive odd numbers less than or equal to a given
`int`. (Don't worry about dealing with negative arguments.)

For example, we expect the following behavior:

  # odds_while 10
  - : int list = [1; 3; 5; 7; 9]
  # odds_for 7
  - : int list = [1; 3; 5; 7]
......................................................................*)

let odds_while (limit : int) : int list =
  failwith "odds_while not implemented" ;;

let odds_for (limit : int) : int list =
  failwith "odds_for not implemented" ;;

(*......................................................................
Exercise 6: Rewrite the functional recursive `sum` function from above
using a `while` loop.

For reference, here is the `length` function implemented using a
`while` loop, as in the reading:

    let length_iter (lst : 'a list) : int =
      let counter = ref 0 in        (* initialize the counter *)
      let lstr = ref lst in         (* initialize the list *)
      while !lstr <> [] do          (* while list not empty... *)
        counter := succ !counter;   (*   increment the counter *)
        lstr := List.tl !lstr       (*   drop element from list *)
      done;
      !counter ;;                   (* return the counter value *)

Note that both the counter for the loop and the list need to be
references. Otherwise, their values can't be changed and the loop
will never terminate.
......................................................................*)

let sum_iter (lst : int list) : int =
  failwith "sum_iter not implemented" ;;

(*......................................................................
Exercise 7: Rewrite the recursive `prods` function from above using a
```

```
`while` loop. Don't forget to handle the case where the two lists
have different lengths, by raising an appropriate exception.
....................................................................*)

let prods_iter (xs : int list) (ys : int list) : int list =
  failwith "prods_iter not implemented" ;;

(* You've now implemented `prods` a few times, so think about which of
them you think is the most efficient, and which of them required the
most work to write. Remember the famous quotation from computer
scientist Donald Knuth: "We should forget about small efficiencies,
say about 97% of the time: premature optimization is the root of all
evil."

Tail recursion is a type of optimization, and it may not always be
worth the sacrifice in development time and code readability.
Iterative solutions in a functional programming language like OCaml
may also not be worth the time. It is critical to assess the impact
that your use cases will have on your design. *)

(*....................................................................
Exercise 8: You'll now reimplement one last familiar function:
reversing a list. Write a non-recursive function `reverse` to reverse
a list. (This function is implemented tail-recursively in the `List`
module as `List.rev` (see
https://github.com/ocaml/ocaml/blob/trunk/stdlib/list.ml), and you've
likely used it in previous exercises.)
....................................................................*)

let reverse (lst : 'a list) : 'a list =
  failwith "reverse not implemented" ;;

(* As you've observed in this lab, procedural programming can be
useful, but most problems can and typically should be solved with
functional techniques. However, there is one famous problem that you
may be familiar with a procedural implementation of: CS50's
Mario. (For those unfamiliar, the task is to print a half-pyramid of
lines of a given height as shown in the example below. See
https://docs.cs50.net/2017/ap/problems/mario/less/mario.html.) *)

(*....................................................................
Exercise 9: Implement a function in the procedural paradigm that
prints out a half-pyramid of a specified height, per the below. Use
hashes (#) for blocks. The function should raise an `Invalid_argument`
exception for heights greater than 23. (Why? I don't know. That's just
how CS50 did it.)

    # mario 5 ;;
        ##
       ###
      ####
     #####
    ######
    - : unit = ()

    # mario 24
    Exception: Invalid_argument "This pyramid is way too high for Mario".

....................................................................*)

let mario (height : int) : unit =
  failwith "mario not implemented" ;;
```