CSE 341, Fall 2004, Assignment 3 (version 2)
Due: Friday 29 October, 9:00AM

This homework has to do with TADPoLE (The Absolute Dumbest Programming Language Ever). TADPoLE programs are written using these ML datatypes:

```
datatype exp = I of int               (* integer constant *)
             | V of string            (* variable *)
             | Succ of exp            (* add one *)
             | Pred of exp            (* subtract one *)
             | If of exp * exp * exp  (* if 1st is not 0, then 2nd else 3rd *)
             | Not of exp             (* if 0 then 1 else 0 *)

datatype stmt = Assign of string * exp   (* mutate variable *)
              | Seq    of stmt * stmt    (* do s1 and then s2 *)
              | While  of exp * stmt     (* while e is not 0, do s *)

type environment = (string * int) list
```

- We use ML strings for TADPoLE variables, which are mutable
- All TADPoLE variables hold integers (the environment maps TADPoLE variables to integers)
- *Every variable is always in scope.* If a variable is not explicitly in the environment, then it maps to 0.
- A TADPoLE expression evaluates to an integer given an environment.
- A TADPoLE statement produces a "new" environment given an environment.

Here is a silly TADPoLE statement. If evaluated in an environment where x maps to i, it produces an environment where x maps to $i-2$ and y maps to $i-1$:

```
Seq(Assign("x",Pred(Pred(V "x"))),
    Assign("y",If(Not(I 3), I 7, Succ(V "x"))))
```

Warning: The sample solution is less than 75 lines, not including the `datatype` bindings above. However, this assignment is probably more difficult than earlier ones.

1. (Multiplication in TADPoLE)
   Write a TADPoLE statement that for the environment `[("x",`i`),("y",`j`),("z",0)]` produces an environment where `"z"` maps to i times j. Assume i and j are not negative. The result environment can have any other mappings (temporary variables are fine and it's fine to change what `"x"` and `"y"` map to). Bind your TADPoLE statement to the ML variable `multiply`.
   Hints: Use nested loops where the inner loop adds i to `"z"`. Use another variable in addition to `"x"`, `"y"`, and `"z"`. Sample solution is 6 lines. (Do not test `multiply` on large numbers; see problem EC1.)

2. (Expression evaluation with a lookup function)
   Write an ML function `eval_exp` for evaluating TADPoLE expressions to integers. It should *not* take an expression and an environment. Instead it takes an expression and an ML function of type `string->int`. `eval_exp` should use the function to evaluate variables. (Problem 3 implements the necessary function.)
   Hints: Evaluating expressions requires a recursive ML function. Evaluating a TADPoLE if-expression will evaluate 2 of the 3 subexpressions. Sample solution is 11 lines.

3. (The lookup function)
   Write an ML function `lookup` that given a TADPoLE environment (i.e., a `string*int list`) returns an ML function suitable for passing to `eval_exp`. The returned function takes a string s and returns the `int` that is paired with the occurrence of s *closest to the list's beginning*, or 0 if s is not in the list.
   Hints: You can use `=` to see if two strings are equal. Do not worry if `lookup` has the more general type `(''a * int) list -> ''a -> int`. The sample solution uses currying and is 4 lines.

4. (Statement evaluation)
   Write an ML function `eval_stmt` that given a TADPoLE statement and environment evaluates the statement under the environment and returns a "new" environment.
   Hints: To produce a new environment, you can just add a new pair to the beginning of another environment because pairs earlier in the list shadow later ones. Use `eval_exp` and `lookup`. For sequences, the environment the first statement produces is the environment for evaluating the second statement. For loops, if the expression is not 0 under the environment, we (re)evaluate the loop under the environment that the loop's body produces. Sample solution is 7 lines.

   Test: `lookup (eval_stmt (multiply,[("x",7),("y",9)])) "z";` should evaluate to 63.

5. (If-expression simplification)
   Write an ML function `simplify_tests` of type `exp->exp`. The result expression must be equivalent to the argument (i.e., evaluate to the same `int` for every environment) and *not* have any subexpressions of any of the following forms (where e and e' stand for any TADPoLE expression and i and j stand for any `int`):

   - `If(I` i`,` e`,` e'`)`
   - `If(`e`, I 0, I 1)`
   - `If(`e`, I 1, I 0)`
   - `If(`e`, I` i`, I` j`)` (if $i = j$)
   - `Not(I` i`)`

   Hints: For each "illegal" form, there is a straightforward simplification (but the third one is tricky because e might evaluate to something that is neither 0 nor 1). Recursively process *all* subexpressions of *every* expression form; do this "before" checking the "outer" expression. Sample solution is 18 lines.

6. (Changing toplevel expressions)
   Write an ML function `change_all_toplevel_exps` that takes a function f (of type `exp->exp`) and evaluates to a function that takes a TADPoLE statement s. This function should return a statement just like s except each *toplevel expression* e is replaced with the result of applying f to e. A toplevel expression is just the expression part of an assignment or loop (and not a subexpression of such an expression).

7. (Statement simplification)
   Using `change_all_toplevel_exps`, write an ML function `simplify_stmt` that takes a statement and produces a statement where each expression is simplified with `simplify_tests`.
   Hint: Sample solution is 1 line.

8. (Negating top-level expressions)
   Using `change_all_toplevel_exps`, write an ML function `negate_all_toplevel` that takes a statement and produces a statement where each toplevel expression e is replaced with `Not` e.
   Hint: Sample solution is 1 line.

9. (Replacing top-level variables)
   Using `change_all_toplevel_exps`, write an ML function `replace_all_toplevel_var` that takes a TADPoLE variable x, a TADPoLE expression e, and a TADPoLE statement s and replaces any *toplevel* expression `V` x in s with e. For example, for x=`"y"`, e=`I 0`, and s=
   `Seq(Assign("y",V "x"),Seq(Assign("z",V "y"),Assign("z",Succ (V "y"))))`,
   the result would be
   `Seq(Assign("y",V "x"),Seq(Assign("z",I 0),Assign("z",Succ (V "y")))).`
   Hint: Sample solution is 7 lines.

(See the next page for extra credit and turn-in instructions.)

Extra Credit: The following three questions are all extra credit. You can receive up to 1 point for each. You can do any subset of them.

EC1 (More efficient evaluation) The evaluator you wrote for problems 2–4 is inefficient because it keeps building a bigger environment every time a TADPoLE variable is mutated. Instead, write an ML function `update_env` of type `env*string*int->env` that changes the environment to map the `string` to the `int` and does *not* have the `string` in the list more than once. Do *not* use mutation in ML; it isn't necessary. Write `eval_stmt2` that is like `eval_stmt` except it uses `update_env`. (You should be able to, for example, `multiply` larger numbers in TADPoLE.)

EC2 (No negated tests) Write an ML function `no_test_negates` of type `exp->exp`. The result expression must be equivalent to the argument and *not* contain any expressions of the form `If(Not` e_1`,` e_2`,` e_3`)`.

EC3 (Tetris meets TADPoLE) Write an ML function `on_horizontal_line` that takes two integers, i and n, and produces a TADPoLE program that does the following: Essentially, it decides if a "Tetris piece" with n squares has all its square on horizontal line i (as in homework 1, problem 2). It assumes that in the environment, the x and y coordinates for the squares are in `"x1"`, ..., `"xn"` and `"y1"`, ..., `"yn"`. It "returns" by assigning to `"ans"` (1 for true and 0 for false).
Hint: The `^` operator concatenates ML strings and `Int.toString` converts an `int` to a `string`.

Type Summary: Evaluating a correct solution should generate these bindings: (You may have different type synonyms or more general types. In particular, it is fine if the REP loop gives `lookup` the type `(''a * int) list -> ''a -> int`.) Also, note the value for `multiply` has been replaced by `_` below.

```
val multiply = _ : stmt
val eval_exp = fn : exp * (string -> int) -> int
val lookup = fn : (string * int) list -> string -> int
val eval_stmt = fn : stmt * (string * int) list -> (string * int) list
val simplify_tests = fn : exp -> exp
val change_all_toplevel_exps = fn : (exp -> exp) -> stmt -> stmt
val simplify_stmt = fn : stmt -> stmt
val negate_all_toplevel = fn : stmt -> stmt
val replace_all_toplevel_var = fn : string * exp * stmt -> stmt
```

Of course, generating these bindings does not guarantee that your solutions are correct: *Test your functions.*

The bindings for the extra-credit problems are:

```
val update_env = fn : (string * int) list * string * int -> (string * int) list
val eval_stmt2 = fn : stmt * (string * int) list -> (string * int) list
val no_test_negates = fn : exp -> exp
val on_horizontal_line = fn : int * int -> stmt
```

Turn-in Instructions

- Put all your solutions in one file, `lastname_hw3.sml`, where `lastname` is replaced with your last name.
- The first line of your `.sml` file should be an ML comment with your name and the phrase `homework 3`.
- Email your solution to `daverich@cs.washington.edu`.
- The subject of your email should be *exactly* `[cse341-hw3]`.
- Your `.sml` file should be an *attachment*.