CS 4: Fundamentals Of Computer Programming, Winter 2017

Assignment 4: Lists, recursion, datatypes, and abstraction

Due: *Friday, February 10, 02:00:00*

Coverage

This assignment covers the material up to lecture 10, corresponding to section 2.4.2 of SICP.

What to hand in

All of your code should be saved to a file named `lab4.ml`. This files should be submitted to csman as usual.

Part @: OCaml notes

Testing

For this assignment, we are supplying you with these support files:

1. a `.mli` OCaml interface file (for this assignment: `lab4.mli`)
2. a test script: `tests_lab4.ml`)
3. a `Makefile`.)

You should download all of these files into the directory in which you are writing your `lab4.ml` code. You should not change them, and you should not submit them as part of your assignment submission to csman.

Once your assignment is done, you should compile it and check that it conforms to the interface file by entering this command:

```
$ make
```

Of course, you can also compile your code from inside the OCaml interpreter using the `#use` directive, as we've previously described. To run the test script, type this:

```
$ make test
```

This will compile the `tests_lab4.ml` file (which contains the unit tests) and output an executable program called `tests_lab4`. Then it will run that program and report all test failures.

If you want to compile the code and immediately run the test script (useful during development/debugging), type:

```
$ make all
```

Running the tests generates some log files; to get rid of them (as well as all compiled OCaml files), type:

```
$ make clean
```

Note that almost all the tests/examples described below are also in the test script.

Names and the single quote character

Unlike most programming languages you have used, OCaml allows identifier names to contain the single quote character (`'`) in any position except the first position. Typically, we use the single quote character at the end of names, and sometimes we use more than one single quote character. These are often used to refer to a name which is related in some way to another name but distinct. The single quote character used this way is often called a "prime" character, which comes from its use in mathematics. So the name `foo'` would be pronounced "foo-prime".

Polymorphic variants

Problem A.1 uses polymorphic variants, which are similar to algebraic datatypes except that you don't have to declare the type ahead of time. Polymorphic variants were discussed in lecture 10, and are discussed in the ocaml manual and in Real World OCaml. Polymorphic variants are a truly wonderful feature of OCaml, and it's the only language that has them. We will be using them quite a lot in future assignments.

Part A: Data structures and abstraction

1. [**60**] SICP, exercise 2.29

   A binary mobile consists of two branches, a left branch and a right branch. Each branch is a rod of a certain length, from which hangs either a weight or another binary mobile. We can represent a binary mobile in OCaml using algebraic datatypes:

   ```
   type mobile = Mobile of branch * branch  (* left and right branches *)
   and branch =
     | Weight    of int * int     (* length and weight *)
     | Structure of int * mobile  (* length and sub-mobile *)
   ```

   Note that these types are mutually recursive, which is why we define them with `type...and...`. Note that this `type` declaration defines *two* types; `mobile` and `branch` are each distinct types.

   Here is an incomplete abstraction layer around these types:

   ```
   let make_mobile l r = Mobile (l, r)
   let make_weight l w = Weight (l, w)
   let make_structure l m = Structure (l, m)
   ```

   a. Complete the abstraction layer by defining accessors (which SICP calls "selectors"). Define `left_branch` and `right_branch` to return the branches of a mobile, `branch_length` to return a branch's length, and `branch_structure` to return a branch's structure. `branch_structure` should return a polymorphic variant which will use the tag `` `Weight `` along with an integer weight if the structure is a weight, or `` `Structure `` along with a mobile if the structure is a (sub-)mobile. Neither the `` `Weight `` nor the `` `Structure `` variant should contain the length of the branch (*i.e.* they are not just copies of the `Weight` and `Structure` constructors of the `branch` type).

b. Write the functions `branch_weight1` and `total_weight1` which return the weight of a branch and the total weight of a mobile, respectively. These functions should directly use the representation of the data types `mobile` and `branch`. These functions will need to be mutually recursive, so use the `let rec ... and ...` syntactic form when defining them.

   Then write new versions of these functions called `branch_weight2` and `total_weight2` which only use the abstraction layer functions. These will also be mutually recursive. Make sure you don't inadvertently call `branch_weight1` or `total_weight1` in either function!

   Clearly, `branch_weight1` and `branch_weight2` must be different functions (and similarly for the `total_weight` functions) but they should return the same results given the same inputs.

c. A mobile is said to be *balanced* if the torque applied by its top-left branch is equal to that applied by its top-right branch (that is, if the length of the left rod multiplied by the weight hanging from that rod is equal to the corresponding product for the right side) and if each of the submobiles hanging off its branches is balanced. Design a predicate `is_balanced` that returns `true` if a mobile is balanced according to this criterion. This function should only use the abstraction layer functions described above. In other words, it will be independent of the internal structure of the mobile. We will be taking advantage of this fact below.

d. Suppose we change the representation of mobiles so that the constructors are

```
type mobile'  = { left: branch'; right: branch' }
and  branch'  = Branch' of int * contents
and  contents = Weight' of int | Structure' of mobile'
```

   Write a new abstraction layer around the new representation. Call the new functions `make_mobile'`, `make_weight'`, `make_structure'`, `left_branch'`, `right_branch'`, `branch_length'` and `branch_structure'`. They should return the same kinds of values as their "unprimed" counterparts, taking into account the new representation of mobiles and branches.

   Modify your functions `branch_weight2` and `total_weight2` so that they work with the new abstraction layer. Call the new versions `branch_weight'` and `total_weight'`. If you wrote `branch_weight2` and `total_weight2` correctly (only using the abstraction layer functions), this will be trivial (you'll just have to change the names of the abstraction layer functions in the corresponding function definitions).

   Finally, write a version of `is_balanced` which works with the new representation; call it `is_balanced'`. Again, if you have done everything right so far, this will be trivial (just involving name changes from "unprimed" to "primed").

One major advantage of defining an abstraction layer is so you can rewrite the internal structure of a datatype without having to change much code. Here, for the sake of testing, we are preserving all versions of the abstraction layer functions and functions which depend on them. In a real world scenario, after changing the internal representation of mobiles we would simple change *e.g.* the `make_mobile`, `make_weight`, `make_structure`, `left_branch` and `right_branch` functions to use the new representation. After that, all other code could remain the same.

Here are some test cases you should use to test both representations. (You'll have to modify the `make_xxx` names to work with the second representation.)

```
let m0 =
  make_mobile
    (make_weight 1 1)
    (make_weight 1 1)

let m1 =
```

```
  make_mobile
    (make_weight 3 4)
    (make_structure
      4
      (make_mobile
        (make_weight 1 2)
        (make_weight 2 1)))

let m2 =
  make_mobile
    (make_weight 1 400)
    (make_structure
      10
      (make_mobile
        (make_weight 100 1)
        (make_weight 1 200)))

let m3 =
  make_mobile
    (make_weight 1 (total_weight1 m2))
    (make_structure 1 m2)
```

For example:

```
# total_weight m0;;
- : int = 2
# is_balanced m0;;
- : bool = true
# total_weight m1;;
- : int = 7
# is_balanced m1;;
- : bool = true
# total_weight m2;;
- : int = 601
# is_balanced m2;;
- : bool = false
# total_weight m3;;
- : int = 1202
# is_balanced m3;;
- : bool = false
```

2. [**30**] SICP, exercise 2.30

   A *tree* is a kind of data structure. There are many different kinds of tree-like data structures used in computer programming. One example is the following:

```
type tree = Tree of elem list
and elem =
  | Num of int
  | Sub of tree
```

   This defines a tree where each component of the tree can be an integer or a subtree, and where a tree can have arbitrarily many components. Define a function called `square_tree` which will make a copy of a tree, except that all numbers in the tree will be squared. That is, `square_tree` should behave as follows:

```
# square_tree (Tree [Num 1;
                     Sub (Tree [Num 2; Sub (Tree [Num 3; Num 4]); Num 5]);
                     Sub (Tree [Num 6; Num 7])])
- : tree = Tree [Num 1;
                 Sub (Tree [Num 4; Sub (Tree [Num 9; Num 16]); Num 25]);
                 Sub (Tree [Num 36; Num 49])]
```

Define `square_tree` both directly (*i.e.*, without using any higher-order functions) and also by using map and recursion. Call the first function `square_tree` and the second one `square_tree'`.

Note that in OCaml, the built-in `map` function is `List.map` *i.e.* it exists in the `List` module.

Use these tests:

```
let tree1 = Tree
  [Num 10;
   Sub (Tree [Num 20;
              Sub (Tree [Num 42; Sub (Tree []); Num 12]);
              Sub (Tree []);
              Sub (Tree [Num 13; Sub (Tree [])])]);
   Sub (Tree []);
   Sub (Tree [Num 1; Num 2; Num 3])]

let tree2 = Tree
  [Num 100;
   Sub (Tree [Num 400;
              Sub (Tree [Num 1764; Sub (Tree []); Num 144]);
              Sub (Tree []);
              Sub (Tree [Num 169; Sub (Tree [])])]);
   Sub (Tree []);
   Sub (Tree [Num 1; Num 4; Num 9])]

(* Check that these are equal: *)
(*
square_tree  (Tree []) = (Tree [])
square_tree' (Tree []) = (Tree [])
square_tree  tree1 = tree2
square_tree' tree1 = tree2
*)
```

3. [**15**] SICP, exercise 2.31

   Abstract your answer to the previous problem to produce a function `tree_map` with the property that a version of `square_tree` could be defined as:

   ```
   let square_tree'' tree = tree_map (fun n -> n * n) tree
   ```

   Use the same tests as in the previous problem.

4. [**15**] SICP, exercise 2.32

   We can represent a *set* as a list of distinct elements, and we can represent the set of all subsets of the set as a list of lists. For example, if the set is `[1; 2; 3]`, then the set of all subsets is `[[]; [3]; [2]; [2; 3]; [1]; [1; 3]; [1; 2]; [1; 2; 3]]` . Complete the following definition of a function that generates the set of subsets of a set and give a clear explanation of why it works:

   ```
   let rec subsets = function
     | [] -> [[]]
     | h :: t -> let rest = subsets t in
         rest @ (List.map <??> rest)
   ```

   This problem is a classic. Note that the order of elements in a set is unimportant, so the list results can come in any order (but don't duplicate anything!).

   Don't forget the "clear explanation of how it works"! Put this in an OCaml comment.

   Here are some examples.

```
# subsets [];;
- : 'a list list = [[]]
# subsets [1];;
- : int list list = [[]; [1]]
# subsets [1;2;3];;
- : int list list = [[]; [3]; [2]; [2; 3]; [1]; [1; 3]; [1; 2]; [1; 2; 3]]
```

5. [**25**] SICP, exercise 2.33

   This is another classic problem (SICP is full of them!).

   Fill in the missing expressions to complete the following definitions of some basic list-manipulation operations as accumulations:

```
let rec accumulate op initial sequence =
  match sequence with
    | [] -> initial
    | h :: t -> op h (accumulate op initial t)

let map p sequence =
  accumulate (fun x r -> <??>) [] sequence

let append seq1 seq2 =
  accumulate (fun x r -> x :: r) <??> <??>

let length sequence =
  accumulate <??> 0 sequence
```

   Hints: The `op` part of the problem is the most important part. Each `op` function must be a function of two arguments, the first being the current list value being looked at, and the second the rest of the list after being recursively processed by the `accumulate` function. Also, don't assume that the initial value is always *e.g.* the empty list. The missing parts are very short.

6. [**15**] SICP, exercise 2.36

   The function `accumulate_n` is similar to `accumulate` except that it takes as its third argument a list of lists, which are all assumed to have the same number of elements. It applies the designated accumulation function to combine all the first elements of the lists, all the second elements of the lists, and so on, and returns a list of the results. For instance, if `s` is a list containing four lists, `[[1;2;3];[4;5;6];[7;8;9];[10;11;12]]`, then the value of `accumulate_n (+) 0 s` should be the list `[22;26;30]`. Fill in the missing expressions in the following definition of `accumulate_n`:

```
let rec accumulate_n op init seqs =
  match seqs with
    | [] -> failwith "empty list"
    | [] :: _ -> []
    | h :: t -> accumulate op init <??> :: accumulate_n op init <??>
```

   Use these tests:

```
# accumulate_n (+) 0 [[];[];[]];;
- : 'a list = []
# accumulate_n (+) 0 [[1;2;3];[4;5;6];[7;8;9];[10;11;12]];;
- : int list = [22; 26; 30]
# accumulate_n ( * ) 1 [[2;3];[4;5]];;
- : int list = [8; 15]
```

   Hint: `map` (or `List.map` if you prefer) is your friend. You may also find the `List.hd` (head) and `List.tl` (tail) functions to be useful.

7. [**30**] SICP, exercise 2.37

Suppose we represent vectors $v = (v_i)$ as lists of numbers, and matrices $m = (m_{ij})$ as lists of vectors (the rows of the matrix). For example, the 3x4 matrix:

```
+---------+
| 1 2 3 4 |
| 4 5 6 6 |
| 6 7 8 9 |
+---------+
```

is represented as the list of lists `[[1;2;3;4];[4;5;6;6];[6;7;8;9]]`. With this representation, we can use list operations to concisely express the basic matrix and vector operations. These operations (which are described in any book on matrix algebra) are the following:

```
dot_product v w
  (* returns: the number d, where d = sum_i (v_i * w_i) *)

matrix_times_vector m v
  (* returns: the vector t, where t_i = sum_j (m_ij * v_j) *)

matrix_times_matrix m n
  (* returns: the matrix p, where p_ij = sum_k (m_ik * n_kj) *)

transpose m
  (* returns: the matrix n, where n_ij = m_ji *)
```

We can define the dot product as:

```
let dot_product v w = accumulate (+) 0 (map2 ( * ) v w)
```

where `map2` is a version of `map` which maps a two-argument function (or operator) over *two* lists of equal lengths.

Fill in the missing expressions in the following functions for computing `map2` and the other matrix operations. (The function `accumulate_n` is defined in the previous problem).

```
let rec map2 f x y =
  match (x, y) with
    | ([], []) -> []
    | ([], _) -> failwith "unequal lists"
    | (_, []) -> failwith "unequal lists"
    | ...

let matrix_times_vector m v = map <??> m

let transpose mat = accumulate_n <??> <??> mat

let matrix_times_matrix m n =
  let cols = transpose n in
     map <??> m
```

Examples:

```
# dot_product [] [];;
- : int = 0
# dot_product [1;2;3] [4;5;6];;
- : int = 32
# matrix_times_vector [[1;0];[0;1]] [10;20];;
- : int list = [10; 20]
# matrix_times_vector [[1;2];[3;4]] [-2;3];;
- : int list = [4; 6]
```

```
# transpose [[1;2];[3;4]];;
- : int list list = [[1; 3]; [2; 4]]
# transpose [[1;2;3];[4;5;6]];;
- : int list list = [[1; 4]; [2; 5]; [3; 6]]
# matrix_times_matrix [[1;0];[0;1]] [[1;2];[3;4]];;
- : int list list = [[1; 2]; [3; 4]]
# matrix_times_matrix [[1;2];[3;4]] [[1;2];[3;4]];;
- : int list list = [[7; 10]; [15; 22]]
# matrix_times_matrix [[1;2;3];[4;5;6]] [[1;2];[3;4];[5;6]];;
- : int list list = [[22; 28]; [49; 64]]
```

Hints: You can use the solutions of some of the functions in later functions. The missing parts are quite short, so don't go doing something complicated! Finally, realize that multiplying a matrix by a matrix can be decomposed into multiplying each row of the first matrix by the entire second matrix.

Part B: Structural and generative recursion

In class, we talked about the difference between structural and generative recursion. In this section we'll give you a chance to learn about generative recursion first-hand, by implementing a version of a new sorting algorithm called "quicksort". The version we'll implement may be quite different from ones you may have seen before; this version of quicksort will work on lists (not arrays) and will not change the input list. This section also includes some other problems involving structural and generative recursion.

In SICP, the authors introduce the `filter` higher-order function. Translated into OCaml, that definition would be:

```
let rec filter predicate sequence =
  match sequence with
    | [] -> []
    | h :: t when predicate h -> h :: filter predicate t
    | _ :: t -> filter predicate t
```

where `predicate` is a function of one argument returning a `bool`, and `sequence` is a list. We will use this in this section. (Actually, it is available in OCaml as the `List.filter` function).

1. [**30**] Implement a `quicksort` function that sorts a list of integers in ascending order, returning the new (sorted) list. The `quicksort` function works like this:

    1. If the list is empty, return the empty list.

    2. Otherwise, the first element in the list is called the *pivot*. Use it to create a list of all the elements in the original list which are smaller than the pivot (using the `filter` function), and another list of elements in the original list which are equal to or larger than the pivot (not including the pivot itself). Then recursively quicksort those two lists and assemble the complete list using the OCaml list append operator (`@`).

    3. To make this function extra-general, instead of using the `<` operator to define whether an element is smaller than another, abstract it around a comparison function `cmp` which takes two values and returns a `bool`. We saw examples of this in lecture 9.

    Use these tests:

    ```
    quicksort [] (<) --> []
    quicksort [1] (<) --> [1]
    quicksort [1;2;3;4;5] (<) --> [1;2;3;4;5]
    quicksort [5;4;3;2;1;1;2;3;4;5] (<) --> [1;1;2;2;3;3;4;4;5;5]
    quicksort [5;4;3;2;1;1;2;3;4;5] (>) --> [5;5;4;4;3;3;2;2;1;1]
    ```

2. [**10**] Explain (in an OCaml comment) why the `quicksort` function is an instance of generative recursion and not structural recursion.

3. [**10**] Ben Bitfiddle doesn't understand why the `merge_sort` function in lecture 9 has to have two base cases. He writes a version which only checks for the empty list, not for lists of length 1. Recall that the `merge_sort` function and its helper functions were defined as:

```
let rec odd_half a_list =
  match a_list with
    | [] -> []
    | [x] -> [x]  (* copy 1-element list *)
    | h :: _ :: t -> h :: odd_half t (* skip second element in list *)

let even_half a_list =
  match a_list with
    | [] -> []
    | _ :: t -> odd_half t

let rec merge_in_order list1 list2 cmp =
  match (list1, list2) with
   | ([], _) -> list2
   | (_, []) -> list1
   | (h1 :: t1, h2 :: _) when cmp h1 h2 ->
       h1 :: merge_in_order t1 list2 cmp
   | (_, h2 :: t2) ->
       h2 :: merge_in_order list1 t2 cmp

let rec merge_sort a_list cmp =
  match a_list with
    | []
    | [_] -> a_list
    | _ ->
      let eh = even_half a_list in
      let oh = odd_half a_list in
        merge_in_order
          (merge_sort eh cmp)
          (merge_sort oh cmp) cmp
```

   Ben's version is identical, except for the `merge_sort` function, which looks like this:

```
let rec merge_sort a_list cmp =
  match a_list with
    | [] -> []
    | _ ->
      let eh = even_half a_list in
      let oh = odd_half a_list in
        merge_in_order
          (merge_sort eh cmp)
          (merge_sort oh cmp) cmp
```

   Explain in an OCaml comment why this won't work. *Hint:* Try it on some very simple test cases.

4. [**5**] The insertion sort function defined in lecture 9 generated a linear iterative process when run. A much shorter insertion sort can be written as a linear recursive process. Fill in the `<???>` section in the following code to write the linear recursive insertion sort. Is this an example of structural recursion or generative recursion? Write a comment explaining which kind of recursion this represents.

```
let rec insert_in_order new_result a_list cmp =
  match a_list with
    | [] -> [new_result]
    | h :: t when cmp new_result h -> new_result :: a_list
    | h :: t ->  h :: insert_in_order new_result t cmp
```

```
let rec insertion_sort a_list cmp =
  match a_list with
    | [] -> []
    | h :: t -> <???>
```

You only need to add a single line in the indicated position.

Part C: Miniproject: algebraic expressions

[**90**] Loosely "derived" from SICP, exercise 2.56.

One of the best things about OCaml is how easy it is to express complicated data structures using algebraic datatypes. We will use this facility to represent algebraic expressions (real algebra, like what you learned in high school) and perform interesting tasks on them.

We will use the type `expr` to represent algebraic expressions. Its definition is:

```
type expr =
  | Int of int           (* constant *)
  | Var of string        (* variable *)
  | Add of expr * expr   (* expr1 + expr2 *)
  | Mul of expr * expr   (* expr1 * expr2 *)
  | Pow of expr * int    (* expr^n *)
```

There is one non-structural constraint that we will impose on this type: the exponent in a `Pow` expression must be a non-negative integer.

Clearly, expressions in this type can't represent all possible algebraic expressions, but they can represent a lot of cases of interest (*e.g.* polynomials with arbitrary numbers of variables), and it would be easy to extend this type to deal with more complex kinds of algebraic expressions.

It's possible to write a parser from algebraic expressions written in the usual mathematical notation to this datatype, but we won't do that here. Instead, we will directly use the datatype constructors to build our expressions (we're not worrying about abstraction here either). Some example expressions written in this form would be:

```
42 --> Int 42

x --> Var "x"

x * y --> Mul (Var "x", Var "y")

(x - 1) * (y + x) --> Mul (Add (Var "x", Int (-1)), Add (Var "y", Var "x"))

x**3 + 6*x*y - 1 -->
  Add (Add (Pow (Var "x", 3), Mul (Int 6, Mul (Var "x", Var "y"))), Int (-1))
```

Once we have an algebraic expression in this form, we can do a number of interesting manipulations on it. We will do two of these in this section.

1. Many algebraic expressions can be converted into equivalent but simpler algebraic expressions. This process is called "simplifying" the algebraic expression. Simplification is a difficult problem in general, because there can be many mutually-exclusive definitions of what "simpler" means. However, some simplifications are universally agreed upon. For instance:

- An expression which adds two integers can be replaced by a single integer (the sum of the two integers).
- An expression which multiplies two integers can be replaced by a single integer (the product of the two integers).
- An expression which takes an integer to the power of another integer can be replaced by a single integer (the power of the two integers).
- An expression which adds zero to another expression E is just the expression E.
- An expression which multiplies an expression E by 0 is just 0.
- An expression which multiplies an expression E by 1 is just E.
- An expression which raises an expression E to the power of 0 is just 1 (we are ignoring the tricky case of 0 to the power of 0 here).
- An expression which raises an expression E to the power of 1 is just E.

Write a function called `simplify1` which does all of these simplifications on an algebraic expression represented as a value of the type `expr`. Make sure that you consider the cases where *e.g.* 0 is added to an expression from the left or from the right, and similarly for other simplifications. Also, if an expression can't itself be simplified, try to recursively simplify its subexpressions.

One thing that makes simplification difficult is that simplifying an expression can yield another expression which can itself be simplified. The best way to handle this is to repeat the simplification process until no more simplification can be performed. We are providing you with the following function `simplify` which will call your `simplify1` function to simplify an expression until no more of the simplifications described above are possible:

```
let rec simplify expr =
  let e = simplify1 expr in
    if expr = e
      then expr
      else simplify e
```

Technically, we say that `simplify` computes the *fixpoint* of `simplify1`. This makes the definition of `simplify1` simpler. For instance, if you simplify `E + 0` to just `E`, you don't then have to recurse on `E` in the `simplify1` function, because the expression `E` will be simplified again by the `simplify` function until it can't be simplified any more. On the other hand, if you have an expression `E + F` which can't be simplified by any of the rules above, you have to recursively simplify the parts, because if you don't, calling `simplify1` on this expression again won't do anything.

Here are some expressions to test your simplifier. These represent the results after `simplify` has been called (not just `simplify1`).

```
Int 42 --> Int 42
Var "x" --> Var "x"
Add (Int 32, Int 41) --> Int 73
Add (Add (Int 1, Int 2), Add (Int 3, Int 4)) --> Int 10
Add (Mul (Int 1, Int 2), Mul (Int 3, Int 4)) --> Int 14
Mul (Mul (Int 1, Int 2), Mul (Int 3, Int 4)) --> Int 24
Mul (Add (Int 1, Int 2), Mul (Int 3, Int 4)) --> Int 36
Pow (Int 0, 0) --> Int 1
Pow (Int 10, 2) --> Int 100
Pow (Add (Int 1, Int 2), 2) --> Int 9
```

```
Add (Var "x", Int 0) --> Var "x"
Add (Int 0, Var "x") --> Var "x"
Mul (Int 0, Var "y") --> Int 0
Mul (Var "y", Int 0) --> Int 0
Mul (Int 1, Var "z") --> Var "z"
Mul (Var "z", Int 1) --> Var "z"
Pow (Var "x", 0) --> Int 1
Pow (Var "x", 1) --> Var "x"

Pow (Add (Var "x", Int 0), 1) --> Var "x"
Add (Add (Var "x", Int 0), Mul (Var "y", Int 0)) --> Var "x"
```

2. Another fun thing you can do with algebraic expressions represented programmatically is to symbolically differentiate them (*i.e.* compute their derivatives with respect to some variable). Specifically, you need to implement the following differentiation rules:

   - Differentiating any constant gives zero.
   - Differentiating a variable by the same variable gives 1; for a different variable it gives zero.
   - Differentiating a sum by a variable is done by differentiating the components of the sum separately and then adding them up.
   - Differentiating a product by a variable uses the product rule: the derivative of `E1 * E2` by `x` is `d(E1)/dx * E2 + E1 * d(E2)/dx`.
   - Differentiating an expression raised to a (positive) integer power uses the power rule: the derivative of `E^n` by `x` is `n * E^(n-1) * dE/dx`.

   Write a function called `deriv` which implements all of these rules. This function only needs to be a few lines long. It's beneficial if the expression to be differentiated is simplified before and after computing the derivative, so use this wrapper function when testing your function:

   ```
   let derivative expr var =
     let e = simplify expr in
     let d = deriv e var in
       simplify d
   ```

   Use (at least) these tests:

   ```
   derivative (Int 10) "x" --> Int 0
   derivative (Var "x") "x" --> Int 1
   derivative (Var "y") "x" --> Int 0
   derivative (Add (Var "x", Var "x")) "x" --> Int 2
   derivative (Add (Add (Var "x", Var "x"), Var "x")) "x" --> Int 3
   derivative (Mul (Var "x", Int 42)) "x" --> Int 42
   derivative (Mul (Var "x", Var "y")) "x" --> Var "y"
   derivative (Mul (Var "x", Var "y")) "z" --> Int 0
   derivative (Mul (Pow (Var "x", 2), Mul (Int 3, Var "x"))) "x" -->
              Add (Mul (Pow (Var "x", 2), Int 3),
                   Mul (Mul (Int 2, Var "x"), Mul (Int 3, Var "x")))
   derivative (Pow (Var "y", 1)) "x" --> Int 0
   derivative (Pow (Var "x", 1)) "x" --> Int 1
   derivative (Pow (Var "x", 2)) "x" --> Mul ((Int 2), (Var "x"))
   derivative (Pow (Mul (Int 3, Var "x"), 3)) "x" -->
              Mul
               (Mul
                 (Int 3,
                   Pow (Mul (Int 3, Var "x"), 2)),
                Int 3)
   derivative (Add (Mul (Int 4, Pow (Var "x", 3)),
   ```

```
     Mul (Int 6, Pow (Var "x", 2)))) "x" -->
Add (Mul (Int 4, Mul (Int 3, Pow (Var "x", 2))),
     Mul (Int 6, Mul (Int 2, Var "x")))
```

If your code produces these results, but with some of the terms rearranged (*e.g.* `Mul (Var "x", Int 2)` instead of `Mul (Int 2, Var "x")`, we'll accept it. However, the results should be equivalent. *Note*: the test script may fail on correct answers in this case; don't assume that there is a bug! Instead, try the failing examples and see if your result is reasonable.
