CSE P505, Autumn 2016, Homework 1
Due: Wednesday October 12, 11:00PM

- This assignment emphasizes OCaml (or F#) programing, pattern matching, and higher-order functions.
- Understand the course policies on academic integrity (see the syllabus) and challenge problems.
- Modify `hw1.ml`, available on the course website, to produce your solution.
- Do not use mutation nor features not used in class.
- Do not modify the code provided to you.
- To turn in your solution, follow the "Turn-in" link on the course website and complete a simple file upload. Turn in just one file, named `hw1.ml` or `hw1.fs`.
- This assignment assumes you are using OCaml. Where necessary, differences between OCaml and F# will be noted in parentheses. For example, "...consider using function `foo` (F#: `bar`)".

1. A type `inttree` for representing trees of ints is provided to you, as well as functions `insert`, `contains`, and `fold`. The sample solution includes less than 40 additional lines of code.
   (a) Define `fromList` of type `int list -> inttree` to make a sorted tree containing exactly the ints in the list without repeats. Use `insert` (provided).
   (b) Define three functions, `sum1`, `prod1`, and `avg1`, to compute the sum, product, and average of the ints in a tree. Each has type `inttree -> int`. The sum and product of an empty tree are 0 and 1, respectively. For average, the empty tree should cause a `Division_by_zero` exception (F#: `System.DivideByZeroException`) to be raised. For `sum1` and `prod1`, do not use helper functions. For `avg1`, do not traverse the tree more than once. (Hint: Have a helper function do the traversal and return a pair.)
   (c) Define `map` of type `(int -> int) -> inttree -> inttree` to produce a tree with the same shape as its second argument with the int at each position the result of applying the first argument to the int at the same position in the second argument.
   (d) Define `negateAll` of type `inttree -> inttree` using `map`. It produces a tree of the same shape where each int is replaced with its negation. Note the result is therefore *not* properly sorted, but do not do anything about that.
   (e) In a *short English paragraph* (in an ML comment), explain how a client of `fold` (provided) would use it to compute something about the ints in a tree. Do *not* explain how `fold` is implemented, though obviously you will have to understand its implementation to document it.
   (f) Define `sum2`, `prod2`, and `avg2` to compute the sum, product, and average of a tree (see above), but using `fold`. You should not need more than 1 line (possibly 2 for `avg2`). Use the same pair technique for average.

2. In this problem, we will "play around" with short functions that use function arguments, the list library, and strings. This has the advantage of giving you experience using functional programming for "little script" kind of things. It has the disadvantage of feeling a little more like "little coding puzzles" than the other problems.
   (a) A function `flatten_map` has been given to you. Write a function `flatten_map2` that is equivalent but instead of using library functions it uses only the append operator (`@`) and recursion.
   (b) Write a function `stutter` to take a list of strings and return a list of strings where each string from the input is repeated twice in order. (See `test1` for an example.)

(c) Change the provided definition of `firsts` to be shorter but do the same thing.

(d) In at most one English sentence in your program, describe what happens if an argument to `firsts` contains empty strings.

(e) Write a function `firsts2` that is like `firsts` except for an empty string it "uses" the empty string itself "rather than" `String.sub 0 1`.

(f) Write a function `filter` that is equivalent to `List.filter` but which is implemented in terms of a call to `flatten_map2`. (This is not a particularly efficient way to implement filtering but "it works.")

(g) Write a function `remove_empties` that takes a `string list` and returns a list containing only the non-empty strings from its argument. Use the `filter` you defined in the previous problem.

Cute little facts: `stutter`, `firsts2`, and `remove_empties` all *commute* with each other. And two of the three are *idempotent*.

3. In this problem, you will write two higher-order functions for lists. Both will be useful in the next problem. The sample solution is less than 15 lines.

   (a) Write a function `first_answer` of type `('a -> 'b option) -> 'a list -> 'b option`. The first argument should be applied to elements of the second argument until the first time it returns `Some v` for some `v` and then `Some v` is the result of the call to `first_answer`. If the first argument returns `None` for all list elements, then `first_answer` should return `None`.

   (b) Write a function `all_answers` of type `('a -> 'b list option) -> 'a list -> 'b list option`. The first argument should be applied to elements of the second argument. If it returns `None` for any element, then the result for `all_answers` is `None`. Else the calls to the first argument will have produced `Some lst1`, `Some lst2`, ... `Some lstn` and the result of `all_answers` is `Some lst` where `lst` is `lst1`, `lst2`, ..., `lstn` appended together (order doesn't matter). Note `all_answers f []` should evaluate to `Some []`. Hints: The sample solution uses a helper function with an accumulator and uses the `@` operator.

4. This problem uses these type definitions, which are similar to ones an ML implementation might use to implement pattern matching:[1]

```
type pattern = Wildcard | Variable of string | UnitP | ConstP of int
             | TupleP of pattern list | ConstructorP of string * pattern
type valu = Const of int | Unit | Tuple of valu list | Constructor of string * valu
```

Given `valu` v and `pattern` p, either p *matches* v or not. If it does, the match produces a list of `string * valu` pairs representing what variables would be bound to what values. Order in the list does not matter. The rules for matching should be unsurprising:

- `Wildcard` matches everything and produces the empty list.
- `Variable s` matches any value `v` and produces the one-element list holding `(s,v)`.
- `UnitP` matches only `Unit` and produces the empty list.
- `ConstP 17` matches only `Const 17` (and similarly for other integers) and produces the empty list.
- `TupleP ps` matches a value of the form `Tuple vs` if `ps` and `vs` have the same length and for all i, the i^{th} element of `ps` matches the i^{th} element of `vs`. The list produced is all the lists from the nested pattern matches appended together.
- `ConstructorP(s1,p)` matches `Constructor(s2,v)` if `s1` and `s2` are the same (strings can be compared with `=`) and `p` matches `v`. The list produced is the list from the nested pattern match.

[1]Don't be confused that an ML implementation might itself be implemented in ML.

- Nothing else matches.

Problems (a)–(d) use the `pattern` type definition, but aren't about implementing pattern matching. Problems (e)–(g) are about implementing pattern matching. The sample solution is less than 35 additional lines of code.

(a) A function `g` has been provided to you. In an ML comment, describe in a few English sentences the arguments that `g` takes and what `g` computes (not how `g` computes it, though you will have to understand that to determine what `g` computes).

(b) Use `g` to define a function `count_wildcards` that takes a pattern and returns how many `Wildcard` patterns it contains.

(c) Use `g` to define a function `count_wild_and_variable_lengths` that takes a pattern and returns the sum of the number of `Wildcard` patterns it contains and the string lengths of all the variables in the variable patterns it contains. (Use `String.length`.)

(d) Use `g` to define a function `count_some_var` that takes a string and a pattern (as a pair) and returns the number of times the string appears as a variable in the pattern.

(e) Write a function `check_pat` that takes a pattern and returns true if and only if all the variables appearing in the pattern are distinct from each other (i.e., use different strings). Note the choice of strings for constructors does not matter. Hints: The sample solution uses two helper functions. The first takes a pattern and returns a list of all the strings it uses for variables. Using `List.fold_left` (F#: `List.fold`) with a function that uses append is useful in one case. The second takes a list of strings and decides if it has repeats. It uses `List.exists`.

(f) Write a function `get_match` that takes a `valu * pattern` (notice this is a pair) and returns a `(string * valu) list option`, namely `None` if the pattern does not match and `Some lst` where `lst` is the list of bindings if it does. Hints: Sample solution has one match expression with 7 branches. The branch for tuples uses `List.length`, `all_answers`, and `List.combine` (F#: `List.zip`).

(g) Write a function `first_match` that takes a value and a list of patterns and returns a `(string * valu) list option`, namely `None` if no pattern in the list matches or `Some lst` where `lst` is the list of bindings for the first pattern in the list that matches. Hint: Sample solution is one line, using two functions previously defined.

*See the next page for **challenge problems**.*

5. **Challenge Problem** This problem continues problem 1 with a function `iter` (provided).

   (a) In a *short English paragraph*, explain how a client of the `iter` function (provided) would use it to process all the ints in a tree. In a second *short English paragraph*, explain how `iter` is implemented (e.g., "when" and "how" it traverses the tree).[2]

   (b) Define `sum3`, `prod3`, and `avg3` to compute the sum, product, and average of a tree (see above), but using `iter` (provided). For product, the code must "stop as soon as it sees a 0" (this is easier than when using `fold`). Hint: You should need about 5 lines for each function. For each, use a local helper function as a "loop" that takes the iterator and the answer-so-far.

6. **Challenge Problem** This problem continues problem 4. Write a function `typecheck_patterns` that "type-checks" a `pattern list`. Types for our made-up pattern language are defined by:

```
type typ = Anything (* any type of value is okay *)
         | UnitT (* type for Unit *)
         | IntT (* type for integers *)
         | TupleT of typ list (* tuple types *)
         | Datatype of string (* some named datatype *)
```

`typecheck_patterns` should have type `((string * string * typ) list) -> (pattern list) -> typ option`. The first argument contains elements that look like `("foo","bar",IntT)`, which means constructor `foo` makes a value of type `Datatype "bar"` given a value of type `IntT`. You may assume list elements all have different first fields (the constructor name), but there are probably elements with the same second field (the datatype name). Under the assumptions this list provides, you "type-check" the `pattern list` to see if there exists some `typ` (call it t) that *all* the patterns in the list can have. If so, return `Some t`, else return `None`.

You must return the "most lenient" type that all the patterns can have. For example, if the patterns are `TupleP[Variable("x");Variable("y")]` and `TupleP[Wildcard;Wildcard]`, you must return `TupleT[Anything;Anything]` even though they could both have type `TupleT[IntT;IntT]`. As another example, if the only patterns are `TupleP[Wildcard;Wildcard]` and `TupleP[Wildcard;TupleP[Wildcard;Wildcard]]`, you must return `TupleT[Anything;TupleT[Anything;Anything]]`.

[2] You might also try implementing `iter` using mutation instead of higher-order functions. It is not very pleasant.