Instructions. Your task is to answer 2 written problems, and to write 16 SML functions (excluding local helper functions) as well as test cases for those functions. See the end of this document for instructions on how to submit your solution. In case you’re curious, our reference solution to the (non-karma) programming problems contains about 75 lines of code, excluding comments, blank lines, and the provided code described next.

Provided code. Download hw3template.sml from the course website. The provided code defines several datatypes, as well as a function and a type synonym, for you; you may not change those.

Prohibitions. You may not use mutable references or arrays. You must use good style. Recall that using pattern matching is typically better style than using functions like hd and selectors like #1.

Library functions. Several of the problems on this homework point you to functions in the Standard ML Basis Library. Documentation of that library is available at http://www.standardml.org/Basis/manpages.html.

Part I: Written Problems

Problem 1. Consider the following curried, higher-order functions:

fun K x y = x
fun S x y z = x z (y z)

These two functions are remarkably powerful.

A. What is the value that results from evaluating S K K 17?

B. Define a KS expression to be an ML expression that uses only K, S, and spaces. Give a KS expression I such that the result of evaluating I is a function whose behavior is identical to the function fn x => x.

C. Define an KSI expression to be an ML expression that uses only K, S, I, and spaces. Consider the following two functions:

fun t x y = x
fun f x y = y

Give a KSI expression, call it T, whose behavior is equivalent to t. And give a KSI expression, call it F, whose behavior is equivalent to f.

D. Give a KSI expression, call it OR, such that OR interacts with T and F like logical “or”. That is, your expression should have the following behavior:

T OR T = T
T OR F = T
F OR T = T
F OR F = F

Hint: the answers to parts B, C, and D are each three characters or fewer, excluding spaces.
Problem 2. This is the last problem from Midterm 1. Since very few people solved it on the exam, you now have a second chance to solve it as a homework problem.

The type-checking rules for if expressions require the guard of the expression to have type bool. (In expression if e0 then e1 else e2, the guard is e0—i.e., the expression between if and then.) In C, however, guards are required to have type int: the then branch is executed if the guard is nonzero, otherwise the else branch is executed. In C, for example,

```c
if (1) {
    printf("1\n");
} else {
    printf("0\n");
}
```

will print 1.

Let’s modify ML if expressions such that guards have type int rather than bool.

A. Give a new type-checking rule for ML if expressions that requires guards to have type int. Your rule should otherwise be the same as the standard ML rule. Give the entire rule, not just a fragment of it.

B. To use the “Boolean” comparison operators with our new if expressions, they now need to return integers. Let’s consider the less-than operator <. It should now be a function that takes two integers and returns an integer. Write down its new type (and just its type—nothing else).

C. Also, let’s consider the equality operator =. It should now be a function that takes two values that can be compared for equality and returns an integer. Write down its new type (and just its type—nothing else).

D. Give a new evaluation rule for ML if expressions. Your rule should result in the same semantics as that given for C above. Give the entire rule.

E. ML if expressions are actually syntactic sugar for case expressions. Show how to desugar our new kind of integer-guarded if expressions. That is, how should the compiler rewrite if e0 then e1 else e2 as a case expression, given your new evaluation rule from the previous part? Your answer may not use if expressions.

Part II: Higher-order Functions

All the solutions to these problems require only a single line of code (or perhaps two lines, because of the usual line break in a function binding after =). Beware: despite—or perhaps because of—their concise solutions, these problems are probably more difficult than the programming problems on homework 2. Do not write any functions or helper functions that are themselves recursive.

1. Write a function only_capitals that takes a string list and returns a string list that has only the strings in the argument that start with an uppercase letter. Assume all strings have at least 1 character. Use library functions List.filter, Char.isUpper, and String.sub.

2. Write a function longest_string1 that takes a string list and returns the longest string in the list. If the list is empty, return "". In the case of a tie, return the string closest to the beginning of the list. Use List.foldl and String.size.

3. Write a function longest_string2 that is exactly like longest_string1 except in the case of ties it returns the string closest to the end of the list. Your solution should be almost an exact copy of longest_string1.

4. Write functions longest_string_helper, longest_string3, and longest_string4 such that:

   • longest_string3 has the same behavior as longest_string1 and longest_string4 has the same behavior as longest_string2.
• The longest_string3 and longest_string4 are defined with val-bindings and partial applications of longest_string_helper (described next). In both functions, the first argument passed to longest_string_helper should be a built-in SML operator.
• longest_string_helper has type (int * int -> bool) -> string list -> string (notice the currying). This function will look a lot like longest_string1 and longest_string2 but is more general because it takes a function as an argument. We deliberately leave the behavior of longest_string_helper unspecified; you need to figure it out.

5. Write a function longest_capitalized that takes a string list and returns the longest string in the list that begins with an uppercase letter (or "" if there are no such strings). Use a val-binding and the ML library's o operator for composing functions. Resolve ties like in problem 2.

6. Write a function rev_string that takes a string and returns the string that has the same characters in reverse order. Use ML's o operator, the library function rev for reversing lists, and two library functions in the STRING signature. (Browse the documentation to find the most useful functions.)

Part III: Implementing Pattern Matching

The remaining problems use these (data)type bindings:

datatype pattern = WildcardP | VariableP of string | UnitP | ConstP of int
  | TupleP of pattern list | ConstructorP of string * pattern

datatype value = ConstV of int | UnitV | TupleV of value list
  | ConstructorV of string * value

type bindings = (string * value) list option

Pattern matching is the problem of determining, given a value v and pattern p, whether p matches v. If so, the match produces a list of string * value pairs; the order of that list does not matter. The rules for matching are essentially a subset of those we studied in class for SML:

• WildcardP matches everything and produces the empty list.
• VariableP s matches any value v and produces the one-element list containing (s,v).
• UnitP matches only UnitV and produces the empty list.
• ConstP 42 matches only ConstV 42 (and similarly for other integers), and produces the empty list.
• TupleP ps matches a value of the form TupleV 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 ConstructorV(s2,v) if s1 and s2 are the same string (you can compare them with =) and p matches v. The list produced is the list from the nested pattern match.
• Nothing else matches.

In this part of the homework, you will implement these pattern matching rules.

1. (This problem uses the pattern datatype but is not really about pattern-matching.)
   (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. Note: you will write no code for this subproblem.
   (b) Use g to define a function count_wildcards that takes a pattern and returns how many WildcardP patterns it contains.
(c) Use \texttt{g} to define a function \texttt{count\_wild\_and\_variable\_lengths} that takes a pattern and returns the sum of the number of WildcardP patterns it contains and the string lengths of all the variables in the VariableP patterns it contains. Use \texttt{String.size}.

(d) Use \texttt{g} to define a function \texttt{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.

2. Write a function \texttt{check\_pat} that takes a pattern and returns \texttt{true} if and only if all the variables appearing in the pattern are distinct from one another. (That is, they are different strings. Note that differences in the strings directly carried by ConstructorP patterns are not relevant.) 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; \texttt{List.fold1} and \texttt{@} are useful. The second takes a list of strings and decides whether it has any duplicates; \texttt{List.exists} is useful. Our reference solution is about 15 lines. Note: you will not need to use this function in any of the problems below.

3. Write a function \texttt{all\_answers} of type \texttt{('a \to 'b list option) \to 'a list \to 'b list option} (notice the 2 arguments are curried). The first argument should be applied to elements of the second argument. If it returns \texttt{NONE} for any element, then the result for \texttt{all\_answers} is \texttt{NONE}. Else the calls to the first argument will have produced \texttt{SOME lst1, SOME lst2, \ldots, SOME lstn} and the result of \texttt{all\_answers} is \texttt{SOME lst} where \texttt{lst} is \texttt{lst1, lst2, \ldots, lstn} appended together (order doesn’t matter). Calling \texttt{all\_answers} \texttt{f []} should return \texttt{SOME \[]}. Hints: Our reference solution is about 10 lines. It uses a helper function with an accumulator and uses \texttt{@}.

4. Write a function \texttt{match} that takes a \texttt{value * pattern} and returns a \texttt{bindings}. Return \texttt{NONE} if the pattern does not match, and return \texttt{SOME lst} where \texttt{lst} is the list of \texttt{string*value} pairs if it does match. Hints: Our reference solution has one \texttt{case} expression with 7 branches. The branch for tuples uses \texttt{all\_answers} and \texttt{ListPair.zip}. Our reference solution is about 10 lines.

5. Write a function \texttt{first\_answer} of type \texttt{('a \to 'b option) \to 'a list \to 'b} (notice the 2 arguments are curried). The first argument should be applied to elements of the second argument until the first time it returns \texttt{SOME v} for some \texttt{v}. Then \texttt{v} is the result of the call to \texttt{first\_answer}. If the first argument returns \texttt{NONE} for all list elements, then \texttt{first\_answer} should raise the exception \texttt{NoAnswer}. Hint: Our reference solution is about 5 lines and doesn’t do anything fancy with higher-order functions.

6. Write a function \texttt{first\_match} that takes a \texttt{value * pattern list} and returns a \texttt{bindings}. Return \texttt{NONE} if no pattern in the list matches or \texttt{SOME lst} where \texttt{lst} is the list of bindings for the first pattern in the list that matches. Hints: Our reference solution is 3 lines and uses \texttt{first\_answer} and a \texttt{handle} expression. The \texttt{curry} function is useful.

Part IV: Karma Problem

Write a function \texttt{typecheck\_patterns} that \textit{type checks} a \texttt{pattern list}. Types for our pattern language are defined as follows:

\[
\begin{align*}
\text{datatype typ} &= \text{AnythingT} (* \text{can be used as a type for any value} *) \\
&\quad | \text{UnitT} (* \text{type for Unit} *) \\
&\quad | \text{IntT} (* \text{type for integers} *) \\
&\quad | \text{TupleT of typ list} (* \text{type for tuples} *) \\
&\quad | \text{DatatypeT of string} (* \text{type for named datatype} *)
\end{align*}
\]

Function \texttt{typecheck\_patterns} should have type

\[
((\text{string} \times \text{string} \times \text{typ}) \text{ list}) \times (\text{pattern list}) \to \text{typ option}
\]

The first argument contains elements that look like \texttt{("foo","bar",IntT)}, which means constructor \texttt{foo} makes a value of type \texttt{DatatypeT "bar"} when given a value of type \texttt{IntT}. (In ML, the closest equivalent would be \texttt{foo : int -> bar}.) Assume list elements all have different first fields (the constructor name), but
that there may be many elements with the same second field (the datatype name). Under the assumptions
this list provides, your function should 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; otherwise, return NONE.

You must return the “most lenient” type that all the patterns can have. For example, given patterns
\texttt{TupleP[VariableP("x"),VariableP("y")]} and \texttt{TupleP[WildcardP,WildcardP]}, the most lenient type is
\texttt{TupleT[AnythingT, AnythingT]}. (Even though both patterns could both have (e.g.) type \texttt{TupleT[IntT,IntT]},
that is not what your function should return.) As another example, given \texttt{TupleP[WildcardP,WildcardP]}
and \texttt{TupleP[WildcardP,TupleP[WildcardP,WildcardP]]}, return \texttt{TupleT[AnythingT,TupleT[AnythingT,
AnythingT]]}.

Further Instructions

\textbf{Type Summary.} Evaluating a correct homework solution must generate at least the following bindings,
in addition to the bindings from the code provided to you:

\begin{verbatim}
val only_capitals = fn : string list -> string list
val longest_string1 = fn : string list -> string
val longest_string2 = fn : string list -> string
val longest_string_helper = fn : (int * int -> bool) -> string list -> string
val longest_string3 = fn : string list -> string
val longest_string4 = fn : string list -> string
val longest_capitalized = fn : string list -> string
val rev_string = fn : string -> string
val count_wildcards = fn : pattern -> int
val count_wild_and_variable_lengths = fn : pattern -> int
val count_some_var = fn : string * pattern -> int
val check_pat = fn : pattern -> bool
val all_answers = fn : ('a -> 'b list option) -> 'a list -> 'b list option
val match = fn : value * pattern -> bindings
val first_answer = fn : ('a -> 'b list option) -> 'a list -> 'b
val first_match = fn : value * pattern list -> bindings
\end{verbatim}

Of course, generating those bindings does not guarantee that your solutions are correct...

\textbf{Testing.} You are required to test your functions. Put your testing code in a separate file. We will not
directly grade it, but you must turn it in. Good test cases might help you get some partial credit if your
solution is erroneous.

\textbf{Submission Instructions}

Submissions that do not adhere to these criteria will lose points:

- Put all your written solutions to part I in one file, \texttt{netid\_hw3written.txt}, where \texttt{netid} is your GW
  NetId. This file must be plain text. We recommend using Emacs to create it.
- Put all your solution code to parts II and III (and IV, if you do it) in one file, \texttt{netid\_hw3.sml}
- Put all the tests you wrote for part II and III (and IV, if you do it) in another file, \texttt{netid\_hw3\_test.sml}.
- The first line of all three files should be an ML comment with your name, GW NetId, and the phrase
  Homework 3.
- Upload all three files to the Homework 3 assignment on BlackBoard.
Evaluation Criteria

Solutions will be evaluated on correctness with respect to the specifications in this assignment; style, including indentation and line breaks, with respect to the style guide on the course website; elegance, which is an ineffable quality that includes beauty, effectiveness, and simplicity; and adherence to using only those SML features permitted in this homework.