Contributed by Charilaos Skiadas

Notes on the material

- *Functions are values*, they can be stored in pairs, in lists, passed as arguments to functions, returned as values from functions, etc.
- Anonymous-function syntax is very useful for passing functions or returning functions.
- In the absense of recursion issues, `fun f x = e` is the same as `val f = fn x => e` (but the former is preferred style).
- Avoid *"unnecessary function wrapping":* prefer `val g = f` over `fun g x = f x` and `fun f x = g` rather than `fun f x y = g y`.
- Clearly understand the type signature of the functions `map` and `filter`.
- Study and understand the mechanics of *lexical scope*. It is important.
- Understand the difference between *functions* and *closures*.
- Think about why lexical scope is so important when dealing with "higher-order-functions".
- Always look at the type signatures of your functions and think about how they make sense.
- A type signature like `'a -> 'b -> 'c` should be read as `'a -> ('b -> 'c)`.
- You can use closures to avoid recomputation of expressions that don't change with each call.
- Study and understand the type of `fold`.
- `fold` allows separating recursive traversal from the data processing.
- Currying is another idiom for achieving the concept of multi-argument functions, and it allows callers to provide only some of the arguments, creating a function that takes the remaining ones.
- Study the `curry` and `uncurry` methods.
- SML's `foldl` function has a slightly different form than the `fold` presented in class.
- Sometimes `val f1 = f2 x` leads to "value-restriction" errors. If you encounter this, doing `fun f1 y = f2 x y`, which is normally "unnecessary" function wrapping, is a fine workaround.

Notes on the assignment

- It might be a good idea, while they are fresh in your mind, to rewrite the first homework's problems using pattern-matching.
- It would be a great idea, while they are fresh in your mind, to review Assignment 1 and Assignment 2 to find ways to use higher order functions to simplify them. `map`, `filter`, and `fold` can turn a lot of the previous problems into 1-liners:

1. `number_in_month` can be done via `fold` or via `filter` together with `length`.
2. `number_in_months` can be done via `fold` on the months list.
3. `dates_in_month` can be done via `filter`.
4. `dates_in_months` can be done via `fold`.
5. `month_range` can be done by creating a list of the days, followed by using `map` with `what_month`.
6. `oldest` can be done via `fold`.
7. `get_substitutions` can be done via `fold`.
8. `similar_names` can be done via `map`.
9. `all_same_color` can be done via `fold`, but takes a bit of thinking, and requires treating the empty list separately.
10. `sum_cards` can be done via `fold`.

11. **officiate** can probably be modeled around **fold**, where the list is the moves and the "accumulator" is a tuple describing the state of the system.

- You are required to use certain library functions. Do *NOT* create your own instead.
- Some questions ask that you use certain functions. Make sure you do!
- Use pattern matching. Avoid **#** and **hd**.
- Start with the provided file, you need what is in there. Add your code to it.
- Almost all answers in the first part are very short, many are a simple **val** binding.
- Study the relevant library documentation.
- Don't forget about the function composition operator **o**: If you find yourself doing `fun f x = h (g x)`, this can be written as `val f = h o g`.
- It can be easier to start by writing a function as `fun f x y =` ... and then seeing if some of the arguments can be removed (see "unnecessary function wrapping").
- Take the time to write down the types of the functions you want to combine on a piece of paper, before trying to code. You can avoid a lot of typechecking errors by thinking about what makes sense.
- For problem 9, make sure to understand what the provided function **g** does. Its goal is to traverse a pattern structure, applying the functions **f1** and **f2** that you give it at key parts. Make sure to understand the types of those two functions. The solutions to this problem will all be very short, mostly partial applications of **g**.
- Typing `f;` in the REPL is a great way to find out the type of **f**.
- For **longest_string1** note that if multiple strings have the same length, you need to return the earlier one -- and the opposite for **longest_string2**.
- Infix operators like >, <>, etc. cannot be used just like that as functions wherever you would have used say an **f**. However, you can obtain the "function" corresponding to an infix operator via **op**, like so: `(op <>)` is the function `fn (x,y) => x <> y`.
- **longest_string3** and **longest_string4** are both extremely short and defined via **val** bindings. They should be just a partial application of **longest_string_helper**.
- For **first_answer** and **all_answers**, do some cases manually to understand what they should do.
- Nested patterns in the function **match** are probably a good idea.
- Note that the arguments to **first_match** are curried.
- To help you think about the pattern matching problems, here are some patterns in "normal SML speak" and their equivalents in the assignment's setting. `(_, 5)` is like `TupleP [Wildcard, ConstP 5]`, `SOME (x, 3)` is like `ConstructorP ("SOME", TupleP [Variable "x", ConstP 3])`, and `(s, (t, _))` is like `TupleP [Variable "s", TupleP [Variable "t", Wildcard]]`.
- For the purposes of the assignment, an empty **Tuple** is different from **Unit**, and a **Tuple** with one element is different from that element by itself. So for example, the pattern **UnitP** should not match the value `Tuple []`.

Notes on the Challenge Problem

Contributed by Edwin Dalorzo

Let's imagine that we have a case expression with different patterns, like

```
case x of p1 | p2 | ... | pn
```

Where **x** is of certain type **t** that we could infer out of the patterns `p1, p2, ..., pn`.

In summary, the objective of the challenge exercise is to create an algorithm that (like the SML compiler), is capable of inferring the type **t** of **x** based on the patterns `p1, p2, ..., pn`.

These patterns are provided as the second argument of the challenge exercise function and they represent every one of the branches in a case expression.

If all the patterns in the case expression are compatible with some type **t** then the answer is `SOME t`, otherwise **NONE**.

We would not need the first argument of the challenge exercise except constructor patterns do not "tell us" what type they are. For instance, consider a case expression like:

```
case c of Red => ... | Green => ... | _ => ...
```

We cannot tell what is the type of `Red` or `Green` here. Likewise, in the challenge exercise if we found a constructor like:

```
Constructor("Red",UnitP)
```

How could we possibly infer the type of this constructor unless we had some additional information? And so this explains why we need a first argument of the challenge function containing a type list. It is nothing but our definition of datatypes.

```
datatype color = Red | Green | Blue
```

Would become somewhat like:

```
 [ ("Red", "color", UnitT),
   ("Green", "color", UnitT),
   ("Blue", "color", UnitT)
]
```

Now let's consider several examples:

Example 1

Suppose we had this function:

```
fun b(x) =
   case x of
       (10) => 1
      | a => 3
```

The compiler would determine that `x` has type `int`. How? Easy: one of the patterns is a integer constant. Thus, the other pattern named a must be an integer as well. And there you have it, we just inferred the type of `x`.

In our challenge exercise, this pattern would be expressed as

```
[ConstP 10, Variable "a"]
```

And our algorithm should say that the answer is `SOME IntT` which corresponds with the type the compiler would infer.

Example 2:

A piece of code like the following would not even compile, because we cannot infer a common type for all patterns. The types in the different patterns are conflicting. We cannot tell if `x` is an int or an option.

```
fun b(x) =
   case x of
      (10) => 1
      | SOME x => 3
      | a => 3
```

Thus, consider the following pattern, corresponding with the code above:

```
[ConstP 10, Variable "a", ConstructorP("SOME",Variable “x”)]
```

This cannot produce a common type and the answer our algorithm yields should be `NONE`, equivalent with the compiler throwing an error due to incapacity to determine a common type.

Example 3:

Let's do a more complicated example now:

```
fun c(x) =
    case x of
        (a,10,_) => 1
      | (b,_,11) => 2
      | _ => 3
```

What is the type of **x**?

Well, we can easily infer it's a tuple of three elements. Based on the patterns, we know the second and third elements of this tuple are integers. The first one, on the other hand, can be "anything".

This would correspond with:

```
[TupleP[Variable "a", ConstP 10, Wildcard], TupleP[Variable "b", Wildcard,
ConstP 11], Wildcard]
```

And the answer given by our algorithm should be: `SOME TupleT[Anything,IntT,IntT]`.

Example 4:

Let's use a datatype now.

```
datatype color = Red | Green | Blue
```

Then we need to define the first argument of our challenge function as:

```
[("Red","color",UnitT),("Green","color",UnitT),("Blue","color",UnitT)]
```

Let's say now that we have a function like this:

```
fun f(x) =
   case x of
     Red => 0
    | _ => 1
```

Corresponding with something like:

```
[ConstructorP("Red", UnitP), Wildcard]
```

Our algorithm should go over the patterns and say this is of type:

```
SOME (Datatype "color")
```

Example 5:

Let's use now a more complex datatype

```
datatype auto =  Sedan of color
               | Truck of int * color
               | SUV
```

This would correspond to a first argument as follows:

```
[("Sedan","auto", Datatype "color"),("Truck","auto",TupleT[IntT, Datatype
"color"]),("SUV","auto",UnitT)]
```

Let's say now that we had a function like this:

```
fun g(x) =
   case x of
        Sedan(a) => 1
      | Truck(b,_) => 2
      | _ => 3
```

What is the type of **x**? Well, we can easily infer they are all of type auto.

So, the following argument:

```
[ConstructorP("Sedan", Variable "a"), ConstructorP("Truck", TupleP[Variable "b",
Wildcard]), Wildcard]
```

Should yield `SOME (Datatype "auto")`.

Example 6:

Let's now define a polymorphic type to make this interesting

```
datatype 'a list = Empty | List of 'a * 'a list
```

So, we must first define our first argument:

```
[("Empty","list",UnitT),("List","list",TupleT[Anything, Datatype "list"])]
```

The trick is to notice that the polymorphic type `'a` corresponds to anything here, and so the type inference becomes a bit trickier later on.

Now if we had this function

```
fun j(x) =
   case x of
       Empty => 0
     | List(10,Empty) => 1
     | _ => 3
```

Evidently the patterns are of type list, but not just that, but a list of integers.

So, the following argument corresponding to the patterns in the function:

```
[ConstructorP("Empty",UnitP),ConstructorP("List",TupleP[ConstP 10,
ConstructorP("Empty",UnitP)]), Wildcard]
```

Should yield: `SOME (Datatype "list")`.

This case is tricky, because ConstP(10) needs to correspond with Anything in the constructors metadata as you can see above.

Example 7:

Let's consider this variation of the previous case:

```
fun h(x) =
   case x of
      Empty => 0
    | List(k,_) => 1
```

In this case `k` could be anything. So, we represent these branches as:

```
[ConstructorP("Empty",UnitP),ConstructorP("List",TupleP[Variable "k",
Wildcard])]
```

And the answer should be `Datatype "List"`.

And once more, notice how `Variable "k"` needs to correspond with `Anything` in the datatype definition.

So, in the previous example `ConstP(10)` and now `Variable "x"` can be considered "compatible with" `Anything`.

Example 8:

Just another example

```
fun g(x) =
   case x of
      Empty => 0
    | List(Sedan(c),_) => 1
```

Corresponding with:

```
[ConstructorP("Empty",UnitP),ConstructorP("List",TupleP[ConstructorP("Sedan",
Variable "c"), Wildcard])]
```

Should evidently yield `SOME (Datatype "list")`.

Example 9:

Now for the "most lenient" pattern. In the assignment we get two examples.

The first one suggest that we have two patterns of the form:

```
TupleP[Variable “x”, Variable “y”]
TupleP[Wildcard, Wildcard].
```

This would correspond to something like

```
fun m(w) =
    case w of
          (x,y) => 0
        | (_,_) => 1
```

Interestingly this would not compile, since the patterns are redundant, namely, we would alway go out throught the first branch. But this was simply used with illustration purposes.

We can infer that `w` is a tuple with two elements that can be of anything. So the answer to this type of patterns should be:

```
TupleT[Anything, Anything].
```

What is meant by "most lenient" is that the type `TupleT[IntT, IntT]` (for example) is also a fine type for all the patterns, but it is not as "lenient" (does not match as many values as) `TupleT[Anything,Anything]` so `TupleT[IntT, IntT]` is not correct.

Example 10:

The second example of the "most lenient" requirement is similar but a little more interesting.

The second example suggest a list of patterns like this:

```
TupleP[Wildcard, Wildcard]
TupleP[Wildcard, TupleP[Wildcard,Wildcard]]
```

Which would correspond with

```
fun m(w) =
    case w of
      (_,(_,_)) => 0
    | (_,_) => 1
```

We can infer that `w` is a tuple of two elements, the first one can be anything, the second one is evidently a tuple of other two elements, which in turn can be anything.

So, if we had to infer this we had to say the type of this is

```
TupleT[Anything, TupleT[Anything, Anything]]
```

Which is the expected answer by the challenge exercise. But yet again, the compiler would not handle this type of expression without errors.

Mark as completed