15-150 fall 2020
Homework 08

Out: Sunday 1st November, 2020

Due: Monday 9th November, 2020 at 23:59

LaTeX

The written part of your assignments must be written in LaTeX. To make your job easier, a LaTeX template is provided for each assignment where you simply need to fill in your solutions.

If you want to know what is the LaTeX name for a symbol, you can use Detexify. Also, Mathcha can help you type math. For constructs particular to this course (inductive definitions, proofs, etc), we provide a LaTeX guide with some templates.

Finally, if you are just lost on how to type something or why your file is not compiling, reach out to the course staff.

Code Structure

Your programs will be graded on more than just their input-output behavior. It's not enough to have programs that happen to work: they need to clearly state what they do, have some empirical evidence that they work as advertised, and be easy for other people to read and reason about.

You must use the following five step methodology for writing functions, for *every* function you write the assignments:

1. In the first line of comments, write the name and type of the function.
2. In the second line of comments, specify via a `REQUIRES` clause any assumptions about the arguments passed to the function.
3. In the third line of comments, specify via an `ENSURES` clause what the function computes (what it returns).
4. Implement the function (include type annotations for the arguments and result of the function). Your code should compile without errors (and preferably without warnings). Compilation errors in a file will result in every task carried out in this file getting no credit.
5. Provide test cases, generally in the format
   `val <return value> = <function> <argument value>`.
   Test cases are individual. Sharing them is not allowed.

For example, for the factorial function presented in lecture:

```
1  (*  factorial: int -> int
2   *  REQUIRES:  n >= 0
3   *  ENSURES: res is  n!
4   *)
5  fun factorial (0: int): int = 1
6    | factorial n = n * factorial (n-1)
7
8
9  (* Tests: *)
10
11 val 1 = factorial 0
12 val 720 = factorial 6
```

Style

Programs are written for people to read — it's convenient that they can be executed by machines, but their high level text is a way for one person to explain an idea to another person. Your code should reflect this, and we will grade your code on how easy it is to understand.

The published style guide is your primary resource here. Strive to write clear, concise, and elegant code. If you have any questions about style, or just have a feeling that a piece of code could be written more simply, don't hesitate to ask!

Testing Modules

Because modules encourage information hiding, the way to test SML structures and functors is a bit different from what you did in the past. In fact, outside of a module, you may have no way to view the values of an abstract type. This means you can't compare the result of an operation with the expected value because you have no way to construct this expected value.

So, how to test modular code? There are essentially two ways to proceed.

Inside-the-box testing: You can't build values outside your module, but you can do so inside (typically). Then, what you would do is to put your normal tests inside the structure you are working on. As usual, if a test fails, a `binding non exhaustive` exception will be raised.

This is a bit trickier to do with functors, because you may not have a way to build values that depend on the functor's parameters. In this case, outside-the-box testing is your only option.

Outside-the-box testing: Many modules export a printing function and an equality function (conventionally called `toString` and `eq`, respectively). You can then use the equality function to test that the value returned by a function is the value you expect. You can use the printing function to visualize returned values of hidden type.

When a module does not provide such functions, it exports operations that interact with each other, somehow. You can leverage these interactions for testing purposes. For example, a dictionary exports `insert` and `lookup` operations. You may test a module implementing

dictionaries by populating a dictionary using `insert` and then use `lookup` to check that the expected entries are in it (and that unexpected entries are not).

Best of all, you want to use a combination of inside- and outside-the-box testing. Notice that inside-the-box testing is implementation-dependent, but outside-the-box is not.

The Compilation Manager

We will be using several SML files in this assignment. In order to avoid tedious and error-prone sequences of `use` commands, we will use SML's *compilation manager*. The compilation manager (CM) is a system that keeps track of what files have been modified and runs just them (and the files that depend on them) through SML. If you have used `make` on a Unix system, the idea is very similar.

Using CM is simple. In fact, there are two ways to do so:

Go to the directory containing your work and run at the terminal prompt (written `#`):

```
# sml -m sources.cm
```

Launch SML from the directory containing your work, and then run at the SML prompt (written `-`):

```
- CM.make "sources.cm";
```

Both wil load all the files listed in `sources.cm`[1] and take you to the SML prompt. Do so whenever you change your code. No need to call `use` — in fact you may confuse CM. For large programs, CM offers a better interface to the command line. There is less typing and less of an issue with name shadowing between iterations of your code.

In short, on this assignment, the development cycle will be:

1. Edit your source files.
2. Type either `sml -m sources.cm` at the terminal prompt or `CM.make "sources.cm";` at the SML prompt.
3. Fix errors and repeat.

`CM.make` creates a directory called `.cm` in the current working directory. It gets populated with metadata needed to work out compilation dependencies. The `.cm` directory can safely be deleted at the completion of this assignment (in fact, it can become quite large)

It's sometimes happens that the metadata in the `.cm` directory gets into an inconsistent state — if you run `CM.make` with different versions of SML in the same directory, for example. This often results in bizarre error messages. When that happens, it is safe to delete the `.cm` directory and compile again from scratch.

[1] The file `sources.cm` contains a list of the files tracked by CM. Feel free to take a peek if you are curious!

1 Countdown

Countdown is a British game show based on a French TV program from the mid 1960's called *Des Chiffres et des Lettres*. In this exercise, we will automate one of the tasks the contestants were charged to solve.

In this task, *le compte est bon* in French, the game show host drew six "small numbers" and one larger target number. The contestants had to write an arithmetic expression that used the small numbers and whose value was the large number. They could apply the standard four operations $+$, $-$, $\times$ and / (integer division) any number of times, but use each small number at most once. The first contestant to find such an expression won.

Here's an example:

Small numbers						Target number
4	6	4	8	25	9	306

This instance is solvable since $((8 \times 4) + (6 - 4)) \times 9 = 306$. Note that not all small numbers need to be used (here 25 plays no role). Note also that not all instances have a solution. For example, there doesn't seem to be a combination of the above small numbers that can evaluate to 1007.

We will now automate the solution of the *Countdown* game. Actually, we will solve it for an arbitrarily long drawing of small numbers, any operations we fancy, and in fact these operations won't even need to operate on numbers at all.

In all the tasks in this homework, you are welcome to use any functions in the SML standard libraries. In particular, higher-order functions will lead to very succinct solutions, much more so than writing everything from scratch. Write your solutions in file `countdown.sml` and test them in file `test-all.sml`.

1.1 List Operations

Our first strategy to implement *Countdown* will be to generate all possible arithmetic expressions based on the given input (the small numbers) and the available operations, and check them in turn until one evaluates to the target number (that's not very smart, but it will be a good warmup to do better in the next section). We will carry out this program in stages: we will convert our input, represented as a list, into a tree, then splice operations into the nodes to obtain expressions, and finally evaluate them. But first, we need a few operations on lists themselves.

Coding Task 1.1 (5 points) Define the SML function

```
1 | partitions: 'a list -> ('a list * 'a list) list
```

such that the call `partitions l` returns the list of all the pairs `(l1,l2)` such that `l` $\cong$ `l1@l2` and neither `l1` nor `l2` are empty. Said differently, this function computes all the possible ways to partition `l` into two non-empty lists while maintaining the original order.

Coding Task 1.2 (5 points) Write the function

```
cartesian: 'a list * 'b list -> ('a * 'b) list
```

such that the call `cartesian (l1,l2)` returns the list of all the pairs `(x1,x2)` where `x1` occurs in `l1` and `x2` occurs in `l2`. That's the Cartesian product of `l1` and `l2`.

The above functions will be enough to solve most but not all *Countdown* problems. For the real thing, you will need the following functions.

Coding Task 1.3 (5 points) Write the function

```
permutations: 'a list -> 'a list list
```

such that the call `permutations l` returns the list of all the permutations of `l`.

Coding Task 1.4 (2 points) Define the SML function

```
permoPartitions: 'a list -> ('a list * 'a list) list
```

such that the call `permoPartitions l` returns the list of all the pairs `(l1,l2)` such that `l1` is a non-empty sublist of `l` and `l2` is the rest of `l`. The list `l2` shall not be empty either.

1.2 Lists to Trees

Do you remember the `inorder` function, the one that returns the list of the elements encountered during an in-order traversal of a tree? Easy enough. What about doing the opposite? Given a list, build all the trees whose in-order traversal are that particular list.

We will be working with trees that hold data in their leaves. They are defined as follows:

```
datatype 'a tree = Leaf of 'a
                 | Node of 'a tree * 'a tree
```

This declaration and some useful operations on trees can be found in the structure `Tree`, provided as part of the starter code for this assignment.

Coding Task 1.5 (8 points) Write the function

```
trees: 'a list -> 'a tree list
```

so that the call `trees l` returns the list of all the trees `t` such that `inorder t` $\cong$ `l`.

Coding Task 1.6 (2 points) Write the function

```
permoTrees: 'a list -> 'a tree list
```

which operates like `trees` but that return all trees whose inorder traversal is any permutation of the input list.

1.3 Trees to Expressions

Next, we will transform trees into expressions. An expression is a tree that holds values in its leaves and carries an operation in each inner node. Expressions and a few useful functions on them are defined in the structure `Expression` provided as starter code. The type of expressions is declared as follows:

```
type 'a operation = string * ('a * 'a -> 'a)
datatype 'a exp = Val of 'a
                | Op of 'a operation * 'a exp * 'a exp
```

For example, addition on integers (or reals) is defined as `("+", op+): int operation`, where the first component of the pair is used for printing. The function `eval` that evaluates an expression to a final value is pre-implemented for you.

Coding Task 1.7 (1 points) As warmup, write the function

```
treeToExp: 'a operation -> 'a tree -> 'a exp
```

such that the call `treeToExp opr t` returns the expression whose values are exactly the leaves of the tree `t`, and whose inner nodes all contain the operation `opr`. The resulting expression should have the same shape as the input tree, with identical values in corresponding leaves.

Coding Task 1.8 (6 points) Now the real thing: write the function

```
treeToExps: 'a operation list -> 'a tree -> 'a exp list
```

such that the call `treeToExps ops t` returns the list of all expressions obtained by instrumenting each inner node of `t` with one of the operations in `ops`. Again, the leaves shall contain the values in the input tree in the given order, and the overall shape shall remain the same.

1.4 Solving Countdown

At this point, we have all the ingredients to implement *Countdown*.

Coding Task 1.9 (5 points) Define the function

```
mko_countdown: ''a operation list -> ''a list * ''a -> bool
```

such that `mko_countdown ops (inp,tgt)` returns `true` iff there is a way to write an expression `e` that uses *all* the values in the input list `inp` and any of the operations in `ops` so that `e` evaluates to the target value `tgt`. Each value in `inp` shall be used exactly once. Values are to be used in the order they occur in `inp`. Recall that `''a` is type variable that can be instantiated only to types that admit equality.

Coding Task 1.10 (2 points) Define the function

```
ordered_countdown: int list * int -> bool
```

that solves a variant of the *Countdown* problem that requires that the small numbers be combined in the given order, although not all of them need to be used. The resulting expression can use the four basic arithmetic operations we learned in school.

Coding Task 1.11 (2 points) Let's solve the real thing. Define the functions

```
mk_countdown: ''a operation list -> ''a list * ''a -> bool
countdown: int list * int -> bool
```

that operate like the above counterparts, but allows using the numbers in the input sequence whenever needed, thereby lifting the restriction on using them in the given order. You still cannot use them more than once.

2 Kountdown

The implementation of *Countdown* in the last section has two defects. The first is that it generates all the possible expressions for the given input and available operations, and only then checks if any of them evaluates to the target. That's clearly a waste. Have you tried your implementation on the instance at the beginning of the last section? Did you notice how long it takes? Did it even terminate for you on the example inputs? The second issue is that, whenever it reports that an instance is indeed solvable, there is no way to verify this claim. In this part of the assignment, we will address both concerns.

To tackle the first issue, we will make use of a data structure that we call an *iterator*. Iterators center around the following type

```
datatype 'a iterator = NoMore
                     | Next of 'a * (unit -> 'a iterator)
```

and are used to efficiently report the results of a costly computation that may return multiple solutions. Say that `e` is an expression that computes to an iterator. The evaluation of `e` returns `NoMore` if there are no solutions. If it instead returns `Next (v,h)`, then `v` is the first solution and `h` is a function that, when called, returns the next solution (together with a function that returns the one after that, and so on). The structure `Iterator` in the starter code defines many useful operations on iterators. Iterators are closely related to continuations.

Because an iterator pauses at the first solution, we can avoid computing the remaining alternatives if all we are interested in is the first solution — like in *Countdown*. This can result in substantial savings. In this part of the assignment, we will use iterators to implement *Countdown*. We will proceed pretty much in the same way as in the last section, but this time rely on iterators. Write your solutions in file `kountdown.sml` and test it in `test-all.sml`. You are not allowed to make calls to the functions you defined in the previous section. Furthermore the list-conversion functions should be used only for testing.

2.1 Lists Operations

Let's write a version of the list partition function that returns an iterator rather than the list of all results. It is convenient to rely on proper continuations to do so.

Coding Task 2.1 (5 points) Write the SML function

```
val part : 'a list -> (unit -> ('a list * 'a list) iterator)
         -> ('a list * 'a list) iterator
```

such that the call `part l k` returns an iterator containing all the partitions of the list `l` into a pair of non-empty lists, or calls the continuation `k` if there are no such pairs.

Coding Task 2.2 (1 points) Define the function

```
val partitions: 'a list -> ('a list * 'a list) iterator
```

that returns an iterator containing all the partitions of the input list into pairs of non-empty lists.

We can similarly define the permutation-based function to return iterators.

Coding Task 2.3 (4 points) Write the functions

```
val permutations: 'a list -> 'a list iterator
val permoPartitions: 'a list -> ('a list * 'a list) iterator
```

that behave as in the previous section, but return their results in an iterator rather than a list.

The analogous of function `cartesian` has been implemented for you in structure `Iterator`, were you to need it.

2.2 List to Trees

We can similarly use iterators to return the trees whose in-order traversal is a given list.

Coding Task 2.4 (6 points) Write the function

```
val trees: 'a list -> 'a tree iterator
```

that behaves as the homonymous function from the previous section.

Coding Task 2.5 (1 points) Write the function

```
val permoTrees: 'a list -> 'a tree iterator
```

that returns an iterator containing the trees whose in-order traversal is a permutation of the input list.

2.3 Trees to Expressions

We can do the same for the function `treeToExps`.

Coding Task 2.6 (4 points) Define the SML function

```
val treeToExps: 'a operation list -> 'a tree -> 'a exp iterator
```

such that `treeToExps ops t` returns an iterator containing all the expressions obtained from tree `t` by placing one of the operations in `ops` in each inner node.

2.4 Solving Countdown

As we implement *Countdown* itself, we address the second issue mentioned above: rather than producing a yes/no answer, the functions you will be implementing in this section output an iterator that returns the expressions that evaluate to the given target, if any.

Coding Task 2.7 (5 points) Implement the functions

```
1 val mko_countdown: ''a operation list -> ''a list * ''a -> ''a exp
    iterator
2 val ordered_countdown: int list * int -> int exp iterator
```

such that `mko_countdown ops (inp,tgt)` produces an iterator that contains the expressions that use all the values in `inp` in the given order and evaluate to `tgt`. The function `ordered_countdown` applies it to the four arithmetic operations in *Countdown*, and allows some input values not to be used.

Coding Task 2.8 (1 points) Write functions

```
1 val mk_countdown: ''a operation list -> ''a list * ''a -> ''a exp
    iterator
2 val countdown: int list * int -> int exp iterator
```

that fully implement *Countdown*, allowing in particular input values to be used in any order.