15-150 Fall 2015
Homework 08

Out: Wednesday, 28 October 2015
Due: Tuesday, 3 November 2015 at 23:59 EDT

1 Introduction

In this homework, you will get practice with exceptions and with SML's module system. You will implement a structure to match a particular signature and perform backtracking with continuations.

1.1 Getting The Homework Assignment

The starter files for the homework assignment have been distributed through our `git` repository, as usual.

1.2 Submitting The Homework Assignment

Submissions will be handled through Autolab, at `https://autolab.cs.cmu.edu`

In preparation for submission, your `hw/08` directory should contain a file named exactly `hw08.pdf` containing your written solutions to the homework.

To submit your solutions, run `make` from the `hw/08` directory (that contains a `code` folder and a file `hw08.pdf`). This should produce a file `hw08.tar`, containing the files that should be handed in for this homework assignment. Open the Autolab web site, find the page for this assignment, and submit your `hw08.tar` file via the "Handin your work" link.

The Autolab handin script does some basic checks on your submission: making sure that the file names are correct; making sure that no files are missing; making sure that your code compiles cleanly. Note that the handin script is *not* a grading script—a timely submission that passes the handin script will be graded, but will not necessarily receive full credit. You can view the results of the handin script by clicking the number corresponding to the "check" section of your latest handin on the "Handin History" page. **If this number is** 0.0**, your submission failed the check script; if it is** 1.0**, it passed.**

Remember that your written solutions must be submitted in PDF format—we do not accept MS Word files or other formats.

All the code that you want to have graded for this assignment should be contained in `exceptions.sml` and `sudoku.sml`, and must compile cleanly. If you have a function that happens to be named the same as one of the required functions but does not have the required type, it will not be graded.

1.3 Due Date

This assignment is due on Tuesday, 3 November 2015 at 23:59 EDT. Remember that you may use a maximum of one late day per assignment, and that you are allowed a total of three late days for the semester.

1.4 Methodology

You must use the five step methodology discussed in class for writing functions, for **every** function you write in this assignment. Recall the five step methodology:

1. In the first line of comments, specify the type of the function.
2. In the second line of comments, specify via a `REQUIRES` clause any assumptions about the arguments to be passed to the function.
3. In the third line of comments, specify via an `ENSURES` clause what the function computes (what it returns when applied to an argument that satisfies the assumptions in `REQUIRES`).
4. Implement the function.
5. Provide testcases, generally in the format
   `val <return value> = <function> <argument value>`.

For example, for the factorial function presented in lecture:

```
(*  fact : int -> int
 *  REQUIRES:  n >= 0
 *  ENSURES: fact(n) ==> n! *)
fun fact (0 : int) : int = 1
  | fact (n : int) : int = n * fact(n-1)

(* Tests: *)
val 1 = fact 0
val 6 = fact 3
val 720 = fact 6
```

1.4.1 Emphatic Warning

CM will not return cleanly if any of the files listed in the sources have no code in them. Because we want you to learn how to write modules from scratch, we have handed out a few files that are empty except for a few place holder comments. That means that there are a few files in the `sources.cm` we handed out that are commented out, so that when you first get your tarball `CM.make "sources.cm"` will work cleanly.

You must uncomment these lines as you progress through the assignment! If you forget, it will look like your code compiles cleanly even though it almost certainly doesn't.

2 Types

Task 2.1 (3 pts). Determine the most general type of the following expression:

```
local
  exception Factorial
in
  fun fact_checked n =
    case (Int.compare(n,0)) of
      LESS    => raise Factorial
    | EQUAL   => 1
    | GREATER => n * fact_checked (n-1)
end
```

Task 2.2 (3 pts). Determine the most general type of the following expression:

```
let
  exception NOPE
  fun p "42" = true
    | p _ = raise NOPE
in
   List.filter p
end
```

Task 2.3 (3 pts). Determine the most general type of the following expression:

```
let
  exception OOPS
  fun f (SOME x, s) = (Int.toString x) ^ " " ^ s
    | f (NONE, _) = raise OOPS
in
  foldr f ""
end
```

3 Exceptions

On hw07, you wrote the function `findOne` with continuations. You'll now modify that function to use exceptions instead.

Task 3.1 (10 pts). Write a recursive function

```
findOneExn : ('a -> bool) -> 'a shrub -> 'a
```

such that for all types `t` and all total functions `p : t -> bool`, all values `T : t shrub`, `findOneExn p T` evaluates to the leftmost value `v` in `T` such that `p v = true`, if there is one and raises the exception `NotFound` otherwise. As in hw07, by leftmost, we mean that your function should give priority to the left subtree over the right subtree. For example:

```
val T = Branch(Leaf 1, Leaf 2)
findOneExn (fn x => x > 0) T = 1
```

A function `f : t -> bool` is called weakly total if, for all values `x` of type `t`, `f(x)` either evaluates to a value or raises an exception. (So `f(x)` doesn't loop forever.)

Task 3.2 (10 pts). Write a function

```
findOneExnWeak : ('a -> bool) -> 'a shrub -> 'a
```

such that for all types `t` and all weakly total functions `p : t -> bool`, all values `T : t shrub`, `findOneExnWeak p T` evaluates to the leftmost value `v` in `T` such that `p v = true`, if there is one and raises the exception `NotFound` otherwise. The note at the bottom of this section may be helpful. It's okay to use `findOneExn`, but you don't have to.

In homework 3, we wrote `subset_sum_cert`, which used brute force to evaluate to a tuple: `bool * int list`.
In lab 5, we wrote `subset_sum_option`, which used brute force to evaluate to an option: `int list option`.
In lab 7, we wrote `subset_sum_cont`, which used continuations to solve the subset sum problem.

Well, buddy, guess what! It's your lucky day. We'll do it all over again with exceptions!

Task 3.3 (15 pts). Write a recursive function

```
subset_sum_exn : int list * int -> int list
```

such that `subset_sum_exn (L,s)` evaluates to a subset of `L` whose elements sum to `s`, assuming such a subset exists. If no such subset exists, raises the exception `NotFound` instead. Make sure you do this recursively, and read the note below!

Note: You may find it helpful to use expressions of the form

```
e handle E1 => e1 | E2 => e2 | . . . | En => en | _ => e'
```

where `e`, `e1`, `e2`, ..., `en`, `e'` are expressions with the same type and `E1`, `E2`, ..., `En` are exception names. The wildcard exception handler is used when `e` raises an exception different from any of the named exceptions.

4 Sudoku

The objective of a sudoku puzzle is to fill an n^2 by n^2 grid with n^2 distinct 'entries' such that each row, each column, and each 'block' contains each entry exactly once. The grid has n^2 rows, n^2 columns, and n^2 blocks. A block is an n by n square of cells, and no two blocks overlap. Entries are most commonly the integers 1 through n^2, but we will make our solution more general to allow for any set of n^2 entries. (Most commonly $n = 3$, with 81 cells arranged in 9 rows of 9 cells each, and the entries are 1 through 9.)

It is easier to draw pictures of puzzles with $n = 2$. For example, here is a solution to one of these, in which we draw double lines to indicate block boundaries and the entries are the digits 1 through $n^2 = 4$.

1	2	3	4
3	4	1	2
2	1	4	3
4	3	2	1

Note that each row, column, and block has exactly one of each entry, where the entries are $\{1, 2, 3, 4\}$.
The case $n = 1$ also makes sense, in a degenerate way: a 1 by 1 grid with a single cell, whose only possible entry is the digit 1.

For this section, we will guide you through writing a sudoku solver that, given a size n, a list of n^2 entries, and a partially-full board of size n^2 by n^2, returns SOME of a full (solved) board if there is one and NONE otherwise.

Representation

The following is the signature we will use for sudoku problems. Descriptions are in `SUDOKU.sig`.

```
signature SUDOKU =
sig
   type entry
   type range = entry list

   val entEq : entry * entry -> bool
   val entToString : entry -> string

   val N : int
   val Range : range

   type cell
   type board
   type choices
```

```
   type solution

   val displayBoard : board -> string
   val displayPart : choices * solution -> string
   val displaySol : solution -> string

   val init : board -> choices * solution
   val solver : choices * solution -> (unit -> solution option)
        -> solution option
   val sudoku : board -> solution option
end
```

For our implementation...
A `cell` is represented as a `(row, col)` pair where $0 \leq$ `row, col` $\leq N^2 - 1$.
A `board` is represented as a list of pairs, where every cell is present exactly once and if `(c, SOME e)` is present, the entry in cell `c` is `e`, and if `(c, NONE)` is present, cell `c` does not yet have an entry, and the board isn't solved.
`choices` is a list pairing cells with a list of the entries we have yet to try for that cell. No cell should be repeated, and no entry should be repeated for any cell.
A `solution` is a list pairing cells with entries. If we don't yet know what's in a cell, that cell won't be in `solution`. No cell is repeated.
Further descriptions of these types are in `sudoku.sml`.

For example, the `solution` above would be represented as:

```
[((0,0), 1),((0,1), 2),((0,2), 3),((0,3), 4),
 ((1,0), 3),((1,1), 4),((1,2), 1),((1,3), 2),
 ((0,0), 2),((0,1), 1),((2,2), 4),((2,3), 3),
 ((0,0), 4),((0,1), 3),((3,2), 2),((3,3), 1)]
```

If we had a `board` passed in as follows:

```
[((0,0),SOME 1),((0,1),NONE  ),((0,2),SOME 3),((0,3),NONE  ),
 ((1,0),NONE  ),((1,1),NONE  ),((1,2),SOME 1),((1,3),SOME 2),
 ((2,0),NONE  ),((2,1),NONE  ),((2,2),NONE  ),((2,3),SOME 3),
 ((3,0),SOME 4),((3,1),NONE  ),((3,2),NONE  ),((3,3),NONE  )]
```

Then the corresponding partial `solution` would be:

```
[((0,0), 1),           ((0,2), 3),
                       ((1,2), 1),((1,3), 2),
                                  ((2,3), 3),
 ((0,0), 4),                                ]
```

or

1		3	
		1	2
			3
4			

And the corresponding `choices` would be:

```
[                  ((0,1),[1,2,3,4]),                  ((0,3),[1,2,3,4]),
 ((1,0),[1,2,3,4]),((1,1),[1,2,3,4]),
 ((0,0),[1,2,3,4]),((0,1),[1,2,3,4]),((2,2),[1,2,3,4]),
                   ((0,1),[1,2,3,4]),((3,2),[1,2,3,4]),((3,3),[1,2,3,4])]
```

or

	[1,2,3,4]		[1,2,3,4]
[1,2,3,4]	[1,2,3,4]		
[1,2,3,4]	[1,2,3,4]	[1,2,3,4]	
	[1,2,3,4]	[1,2,3,4]	[1,2,3,4]

(Note: constructing the `solution` and `choices` from a `board` is exactly what the function `init` will do.)

Implementing a solver

The following tasks will guide you through an implementation of the signature given above. All code for this section should go in `sudoku.sml`.

Task 4.1 (3 pts). Write a function

```
filter : ('a -> bool) -> 'a list -> 'a list
```

such that for all total functions `p`, `filter p L` evaluates to a list `L'` containing all elements `x` in `L` such that `p x` is true.

Task 4.2 (3 pts). Write a function

```
exists : ('a -> bool) -> 'a list -> bool
```

such that for all total functions `p`, `exists p L` evaluates to true if there is an element `x` in `L` such that `p x` is true and false otherwise.

Task 4.3 (10 pts). Write a function

`relates : cell -> cell -> bool`

such that `relates c1 c2` evaluates to true if `c1` and `c2` are in the same row, column, or block in a board of size N^2 by N^2. (Remember that `N` is a global variable within this structure).

Task 4.4 (3 pts). Write a function

`clashes : (cell * entry) -> (cell * entry) -> bool`

such that `clashes (c1,e1) (c2,e2)` evaluates to true if `c1` and `c2` are in the same row, column, or block AND `e1 = e2`.

The following functions will use one or more of the functions you already wrote as helpers. Below, a `board` is 'valid' if it is N^2 by N^2 and has exactly one list element for every cell and no extra elements.
A `solution` is 'valid' if it has at most one element for each `cell` and has no additional `cells` that are not within the size of the board. (It is complete if it has exactly one element for each `cell`.)
A `choices` and `solution` pair `(C,S)` is 'valid' if `S` is valid and for every cell `c` on a grid of size N^2 by N^2, `c` does not appear in `C` and appears with `(c,e)` in `S` for some `entry e` OR `(c, L)` appears in `C` for some `entry list L` and `c` does not appear in `S`.

Task 4.5 (12 pts). Write a function

`init : board -> (choices * solution)`

such that for all valid boards `B`, `init B` evaluates to `(C,S)` such that if the cell `(c, SOME e)` appears in `B`, the cell does not appear in `C` and appears with `(c,e)` in `S` and if `(c, NONE)` appears in `B`, `(c, Range)` appears in `C` and `c` does not appear in `S`. (Remember that `Range` is a global variable within this structure).

Task 4.6 (15 pts). Write a function

`solver : (choices * solution) -> (unit -> solution option) -> solution option`

such that for all valid `choices` and `solution` pairs `(C,S)`, `solver (C,S) k` evaluates to `SOME S'` where `S'` is a complete solution for `S` if a solution exists and `k()` otherwise.
In order to satisfy the above spec, `solver (C,S) k` should
1. evaluate to `SOME S` if `C` is empty
2. otherwise, pick an entry `(c, L)` from `C`. If `L` is empty, there is no way to extend `S` to a complete solution because cell `c` has no options, so call `k()`. Otherwise, extend `S` with some value in `L` and update `C` to match. Call `solver` recursively with these updated values and a continuation that, if called, will try to solve the original problem but with the entry already attempted removed from the list of possible entries for `c`.
Note that our solution is fewer than 10 lines of code. Use the functions you've written as helpers to make this code simple.

Task 4.7 (10 pts). Finally, using `init` and `solver`, write a function

`sudoku : board -> solution option`

such that for all valid boards `B`, `sudoku B` evaluates to `SOME S` where `S` is a complete solution for `B` if a solution exists and `NONE` otherwise.

Testing

You are encouraged to use the provided functions `displayBoard`, `displayPart`, and `displaySol` to test your code. They evaluate to printable strings. We have also provided you with some test cases so that you can reasonably test with `n=3`, but you are still expected to write your own test cases (preferably with `n=2`) for all functions except `solver`. If you choose to test `solver` only by testing `sudoku`, we will accept that.
All tests should go in the `SudokuTests` structure.

For a sanity check, our (unoptimized) solution runs in a reasonable amount of time for `n=3` and `n=4`, but takes quite a while if `n>=5`.
Remember to use `CM.make "sources.cm"` to compile your code!