CSCI 2041: Advanced Programming Principles
Fall 2017, University of Minnesota
Homework 6

Posted: Nov 13, 2017
Due: Nov 27, 2017 by 23:59

Note: This homework has three parts, the first pertaining to mutable lists, the second pertaining formatted input and the third pertaining to coloring graphs. Before you get started, you should copy the three files mutlists.ml, readreal.ml and graph_color.ml. Each of these files contains some code that you will have to augment to solve the problems that are described in detail below. Once you have done this, you will have to push these files to your personal git repository for the course for grading.

To make it a bit simpler for you to download the needed files, I have made a tarball of them here. In have also put these files in the public-repo for the course on github, under homeworks and then hw6.

Submission Protocol

Some requirements that you must adhere to in submitting your work for grade for this homework:

- All of your work must be submitted in a files indicated above: *mutlists.ml*, *readreal.ml* and *graph_color*. These files must be **pushed** to your personal git repository in the organization at the UMN github for this course, i.e. in a repository that has the name *umn-csci-2041F17d/repo-abcde001* where *abcde001* is your username. The three files should be placed at the top level in this directory.

- The three files correspond to three categories of problems described below. Each category has multiple problems. I have included a comment of the form (* Solution for Problem X *), where X is the number of the problem in the relevant category, just before the place where you are expected to write out a solution. Make sure to retain this comment to help the grader quickly identify the code that has to be looked at.

- The problems are specific about the names of the types, functions and the types of the functions you must define. As usual, you **must** stick to the requirements spelled out in each case. To help you with this, I have already included the header for the functions that you are to define.

Note that general requirements for homework submissions are described on the homeworks page and have also been discussed in class. In addition, use all the requirements for good programming that you would have learnt in preceding programming courses, doing so can have an impact on the grade.

Before submitting the final version of your homework, you should ensure it is well-formed and type-correct, i.e., contain no syntax or type errors. To check you could use the following commands on a cselabs machine in a directory containing the file:

```
ocamlbuild mutlists.byte || echo "\n\n---INCORRECT SUBMISSION---\n\n"

ocamlbuild readreal.byte || echo "\n\n---INCORRECT SUBMISSION---\n\n"

ocamlbuild graph_color.byte || echo "\n\n---INCORRECT SUBMISSION---\n\n"
```

If you see:

```
---INCORRECT SUBMISSION---
```

for any of these cases, it means that the corresponding file does not compile and hence is not yet ready to be submitted.

Part I: Mutable lists (3 + 3 points)

We looked at a type for mutable lists in Lab 11 that allowed us to mimic the C style of treating lists, where we can mutate the part of a list cell that points to the tail. To realize this ability, the tail part had to be a reference to a reference to a list cell rather than just a reference to a list cell. In this problem, we will simplify this structure a little to avoid two references. We can still realize operations on lists via mutations but, as I hope you will see, not quite in the way we would do this in C: now we will mutate the entire list cell, not just the part of it that "points to" the tail.

Here is the type for mutable lists that is to be used in this problem:

```
type 'a mylist = 'a listcell ref
  and  'a listcell = Nil | Cons of 'a * ('a mylist)
```

You will find this definition also in the file mutlists.ml that provides the skeleton code which you have complete as described below:

1. Define the function `append : 'a mylist -> 'a mylist -> 'a mylist` that appends two lists by making a destructive change to the first list but using (and leaving unchanged) the second list. Here is an example interaction that shows the behavior of this function:

```
# let l1 = (ref (Cons (1, ref (Cons (2, ref Nil)))));;
val l1 : int listcell ref =
  {contents = Cons (1, {contents = Cons (2, {contents = Nil})})}
# let l2 = (ref (Cons (3, ref (Cons (4, ref Nil)))));;
val l2 : int listcell ref =
  {contents = Cons (3, {contents = Cons (4, {contents = Nil})})}
# append l1 l2;;
- : int mylist =
{contents =
  Cons (1,
   {contents =
     Cons (2, {contents = Cons (3, {contents = Cons (4, {contents = Nil})})})})}
# l1;;
- : int listcell ref =
{contents =
  Cons (1,
   {contents =
     Cons (2, {contents = Cons (3, {contents = Cons (4, {contents = Nil})})})})}
# l2;;
- : int listcell ref =
```

```
{contents = Cons (3, {contents = Cons (4, {contents = Nil})})}
#
```

2. Define the function `rev_app : 'a mylist -> 'a mylist -> 'a mylist` that reverses the list that is its first argument and appends it to the list that is the second argument. The function should destructively change the first list thereby reversing it "in place," and it should use (but leave unchanged) the second list. Some interactions that illustrate the behavior of these functions follow:

```
let l1 = (ref (Cons (1, ref (Cons (2, ref Nil)))));;
val l1 : int listcell ref =
  {contents = Cons (1, {contents = Cons (2, {contents = Nil})})}
# let l2 = (ref (Cons (3, ref (Cons (4, ref Nil)))));;
val l2 : int listcell ref =
  {contents = Cons (3, {contents = Cons (4, {contents = Nil})})}
# rev_app l1 l2;;
- : int mylist =
{contents =
  Cons (2,
   {contents =
     Cons (1, {contents = Cons (3, {contents = Cons (4, {contents = Nil})})})})}
# l1;;
- : int listcell ref =
{contents =
  Cons (1, {contents = Cons (3, {contents = Cons (4, {contents = Nil})})})}
# l2;;
- : int listcell ref =
{contents = Cons (3, {contents = Cons (4, {contents = Nil})})}
#
```

   Note that the identifier `l1` is bound to a very funny looking list at the end. In reality, it is the list that corresponds to the first cell of the first list added on the end of the second list, something that you would expect to happen in an in-place reverse.

Part II: Formatted Input (4 + 4 + 4 points)

In Lab 11, you would have tried your hand at defining a function in OCaml that is capable of reading the next token from an input channel as an integer. This function returned an object of the `option` type and used the value `None` to signal that this token was in fact not an integer. In this problem you will define a function that behaves similarly except that it reads a floating point number presented as a sequence of digits followed by a decimal point, i.e. the character '.', followed by another sequence of digits. In this presentation, the sequence of digits to the left of the decimal point, the whole number part of the floating point number, can be empty as can also the fractional part, the sequence of digits to the right of the decimal point. However, either the whole number part or the fractional part must be non-empty. Further, the presentation of the number *must* contain a decimal point. Thus, the following are "good" floating point numbers

```
10.1   10.  .1
```

whereas

```
10
```

is not.

In implementing this function, we will use the idea of a *lookahead* character. Here is how this will work. First, we will use the identifier `ch` as a reference to a cell that stores the "next" character in the input channel. Any time we want to check what this character is, we will use the function

```
lookupChar : unit -> char
```

Sometimes the lookup will consume the character. In this case, we will use the function

```
getChar : in_channel -> unit
```

to read the next character on the input channel and to save its value in the cell that `ch` is a reference to.

The first few lines in the skeleton code in the file readreal.ml define these functions and also the reference identifier `ch`.

Your task in this problem is to do the following:

1. Define the function `getWhole: in_channel -> int` that is invoked when the next character in the input channel, i.e. the one in the cell referenced by `ch` is a digit and that reads the whole number corresponding to the ensuing sequence of digits in the input channel. This function can be tested by the function `test_getWhole`contained in the file `readreal.ml` and should result in the following kinds of interactions:

   ```
   # test_getWhole ()
   Enter a number: 10
   Your input: 10
   - : unit = ()
   # test_getWhole ();;
   Enter a number: 10.
   Your input: 10
   - : unit = ()
   # test_getWhole ();;
   Enter a number: 11101
   Your input: 11101
   - : unit = ()
   #
   ```

   Note that in the second example, the "cursor" should be left over the decimal point that terminates the whole number, i.e. this should be the lookahead character.

2. Define the function `getFrac: in_channel -> float` that is invoked when the next character in the input channel is a digit that it interprets as being just to the right of a period and that returns the ensuing sequence of digits as the fractional part of a floating point number. This function can be tested using the function `test_getFrac` and should result in the following kinds of interactions

   ```
   # test_getFrac ();;
   Enter a number: 101
   Your input: 0.101000
   - : unit = ()
   # test_getFrac ();;
   Enter a number: 1101
   Your input: 0.110100
   - : unit = ()
   #
   ```

3. Define the function `getFloat : in_channel -> float option` that expects the next token in the input to be a floating point number in the format described and returns it using an option type; the value `None` should be used if the token is not a floating point number. This function can be tested using `test_getFloat` and should result in the following kinds of interactions:

   ```
    # test_getFloat();;
   Enter a number: 10.01
   Your input: 10.010000
   ```

```
- : unit = ()
# test_getFloat();;
Enter a number: 10.
Your input: 10.000000
- : unit = ()
# test_getFloat();;
Enter a number: .1
Your input: 0.100000
- : unit = ()
# test_getFloat();;
Enter a number: 10
Bad input
- : unit = ()
# test_getFloat();;
Enter a number: .
Bad input
- : unit = ()
 #
```

You should find the previously defined functions useful in defining `getFloat`.

In doing this problem you will need a function that converts characters to the number corresponding to their ASCII code. You can use the OCaml library function `int_of_char : char -> int` that you would have also encountered in Lab 11 for this. You will also need a way to convert an integer to a floating point number. Use the OCaml library function `float : int -> float` for this purpose.

Part III: The Graph Colouring Problem (6 + 6 points)

The problem of graph colouring is the following: we are given a collection of nodes with (undirected) edges between them and a palate of colours and we are asked to find a way to assign colours to the nodes in such a way that two nodes that are adjacent do not get the same colour.

This is a problem that turns out to be of significant practical importance. One place where it is useful is in colouring maps: here the nodes are countries or states or counties, depending on what kind of map you are looking at, and two nodes are adjacent if they represent distinct regions that share a border. Another place where graph colouring shows up is in assigning space (such as registers) to the variables in a program. Think here of the space or register as a colour (the palate is especially limited if we are thinking of the registers in a real machine as the colours), the variables are nodes and two variables share an edge if their values are "live" at the same time somewhere in the program; clearly, they can be assigned the same space/register only if they *do not* share an edge. Looked at differently, if we are able to colour the variables satisfactorily, then the colouring gives us a way to allocate space for them.

Graph colouring is a problem for which we do not know a good algorithm currently and for which it is conjectured we will never know one; to put this more precisely, it is what is known as an *NP-complete* problem. Problems of this kind that are practically important are often solved using search. Typically, the search uses heuristics that help in cutting down the work in practice. In this homework, we will take a simplistic approach, using search to solve graph colouring but without building in any heuristics. If the problem excites you, I can suggest some heuristics for you to try later.

The two problems below for which you have to write code differ in the way they want you to implement the search. Before getting to them, we describe some things that are common to their setup. We will represent graphs using two components: a list of nodes and, for each node a list of nodes adjacent to it. As an example, consider the complete graph with 4 nodes named 1, 2, 3 and 4; for simplicity we will use numbers for nodes. (Recall that a complete graph is one in which every node is connected to every other node.) This graph would be represented by the nodes list `[1;2;3;4]` and the adjacency list

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

The functions you will define will also be parameterized by a collection of colours. These will be provided in the form of a list of strings such as the following:

```
["R"; "B"; "G"; "Y"]
```

Problem 1

The goal in this problem is to define a function called `color_graph` that takes a graph and a set of colours presented in the fashion described above and determines a colouring for the graph. There are, however, two special requirements:

- Like in the problems in Lab 12, the function will display the colouring to user and interact with her/him to decide if the colouring is satisfactory. Thus, the type of the function should be the following:

  ```
  color_graph : int list -> (int * int list) list -> string list -> unit
  ```

  Here the kind of interaction expected is the following:

  ```
  # color_graph [1;2;3;4] [(1,[2;3;4]); (2,[1;3;4]); (3,[1;2;4]); (4,[1;2;3])]
                ["R"; "B"; "G"; "Y"];;

  Coloring for the graph: [(4,Y), (3,G), (2,B), (1,R)]
  More solutions (y/n)? y

  Coloring for the graph: [(4,G), (3,Y), (2,B), (1,R)]
  More solutions (y/n)? n
  - : unit = ()
  # color_graph [1;2;3;4;5]
                [(1,[2;3;4;5]); (2,[1;3;4;5]); (3,[1;2;4;5]);
                 (4,[1;2;3;5]); (5,[1;2;3;4])]
                ["R"; "B"; "G"; "Y"];;

  No (more) colourings possible
  - : unit = ()
  #
  ```

  I am giving you the functions necessary for arranging the interactions; see below. Your task will be mainly to organize the search in the problem.

- **You must organize the search in this problem using exceptions**; the next problem will get you to do the same thing using success and failure continuations and the whole point of this part of the homework is to make sure you have understood these two different techniques.

How should we go about solving the graph colouring problem? Here is the basic structure:

1. We pick the first node that has still not been coloured and try to colour it; if we are successful, i.e. if it can be given a colour that is different from the colours given to its neighbours, we add the colouring and move on to the next node.

2. The colour that we have picked for the node in the previous step may turn out to be "bad" in that the rest of the graph cannot be coloured with that choice. In this case, we would backtrack to considering another colour for the node. In the model to be used in this problem, the failure **must be** treated by raising an exception, specifically, the exception `Search_Failure`, and backtracking **must be** treated by catching and handling the exception.

3. On the other hand, we may succeed in colouring all the nodes satisfactorily. In this case, we should interact with the user using the `ask_user` function provided to you and whose behaviour is described below.

To let you focus on the core task, that of organizing the search using exceptions, I have provided you the code for displaying a colouring once one has been found and for checking if the user is satisfied. Read this code and try to understand it so that you can write such functions yourself in the future. However, here is an example that shows you how you would use the code when you are defining `color_graph`:

```
# ask_user show_coloring [(1,"R"); (2,"B"); (3,"G"); (4,"Y")];;

Coloring for the graph: [(1,R), (2,B), (3,G), (4,Y)]
More solutions (y/n)?
```

Note that colourings are represented by lists of node and colour pairs. After displaying the colouring, the function awaits an input from the user. If the input is "y", it raises the exception `Search_Failure`, otherwise it finishes normally.

I have included the header for `color_graph` and comments with it to help you get started on this problem. Your task is mainly to understand the structure and to fill in the code for this function.

Problem 2

In this problem, we are going to solve the same graph colouring task but this time using success and failure continuations to handle the situations were search ends in success and where it ends in failure. **Note that to get credit, you must organize your solution using the idea of continuations to realize search, that is the main point of this problem.** The task is to complete the definition of the function

```
color_graph_cps : int list -> (int * int list) list -> string list -> unit
```

to realize the following computation structure:

1. We pick the first node that has still not been coloured and try to colour it; if we are successful, i.e. if it can be given a colour that is different from the colours given to its neighbours, we add the colouring to the ones we have already determined, and move on to colouring the next node.

2. The colour that we have picked for the node in the previous step may turn out to be "bad" in that the rest of the graph cannot be coloured with that choice. To prepare for this contingency, we should pass on a suitable failure continuation to the function that we are calling. Similarly, it should get a suitable success continuation that would enable it to do the right thing in case of success.

3. We may also fail to find a suitable colour for the node in the first step. In this model, we deal with this situation by calling the failure continuation.

4. Finally, we may have succeeded in colouring all the nodes in the graph. Now we need to interact with the user. To do this, we use the function `ask_user_cps` as described below. Of course, we have to pass it the right success and failure continuations to prepare for the situations where the user is happy with the colouring or rejects it.

The only remaining thing to be explained is how to use the code I have provided you for displaying a solution and getting a response from the user. If `coloring` is bound to a good colouring, then you would invoke

```
(ask_user_cps show_coloring colouring succ fail)
```

where `succ` and `fail` are suitable continuations. Some examples that illustrate the use of this function are the following:

```
#  ask_user_cps show_coloring [(1,"R"); (2,"B"); (3,"G"); (4,"Y")]
                          (fun () -> ())
                          (fun () -> Printf.printf "\nNo (more) colorings\n");;

Coloring for the graph: [(1,R), (2,B), (3,G), (4,Y)]
More solutions (y/n)? n
- : unit = ()
# ask_user_cps show_coloring [(1,"R"); (2,"B"); (3,"G"); (4,"Y")]
                          (fun () -> ())
                          (fun () -> Printf.printf "\nNo (more) colorings\n");;

Coloring for the graph: [(1,R), (2,B), (3,G), (4,Y)]
More solutions (y/n)? y

No (more) colorings
- : unit = ()
#
```

Check out how the two continuations impact on behaviour after the user's response.

For completeness, here is an interaction I got when I ran my code:

```
# color_graph_cps [1;2;3;4] [(1,[2;3;4]); (2,[1;3;4]); (3,[1;2;4]); (4,[1;2;3])]
                  ["R"; "B"; "G"; "Y"];;

Coloring for the graph: [(4,Y), (3,G), (2,B), (1,R)]
More solutions (y/n)? y

Coloring for the graph: [(4,G), (3,Y), (2,B), (1,R)]
More solutions (y/n)? n
- : unit = ()
# color_graph [1;2;3;4;5]
            [(1,[2;3;4;5]); (2,[1;3;4;5]); (3,[1;2;4;5]);
             (4,[1;2;3;5]); (5,[1;2;3;4])]
            ["R"; "B"; "G"; "Y"];;

No (more) colourings possible
- : unit = ()
#
```

Last modified: Nov 20, 2017. Created by gopalan atsign cs dot umn dot edu. Copyright (c) Gopalan Nadathur.

The views and opinions expressed in this page are strictly those of the page author(s). The contents of this page have not been reviewed or approved by the University of Minnesota.