Homework 10: ADTs
Due: 11:59 PM, Nov 13, 2018

Contents

1 Lists (Practice) 2

2 Monopoly Board with OCaml Records! 3

3 Dictionaries 4

3.1 Dictionary Signature .......................................................... 5
3.2 List Dictionaries .............................................................. 5
3.3 Tree Dictionaries ............................................................... 6
  3.3.1 Binary Search Trees (BSTs) ............................................ 6
  3.3.2 Task ............................................................................... 6
3.4 Timing Experiments ............................................................... 6
3.5 A Note on Modules ............................................................... 7

Objectives

By the end of this homework you will be able to:

1. construct records and write produces that work with records
2. implement dictionaries as an abstract data type in OCaml
3. represent dictionaries in more than one concrete way

How to Hand In

For this (and all) homework assignments, you should hand in answers for all the non-practice questions.
For this homework specifically, this entails answering the Records and Dictionaries questions.

In order to hand in your solutions to these problems, they must be stored in appropriately-named files. In particular, each should be named for the corresponding problem, as follows (e.g.,
sig_dictionary.ml corresponds to Dictionary Signature):

- README.txt
- CS17setup.ml
• monopoly.ml
• sig_dictionary.ml
• list_dictionary.ml
• tree_dictionary.ml
• analyze_dict.txt

For this assignment, all files you turn in that contain code must be OCaml files, so they must end with extension .ml. For all coding problems in this homework, you should follow the OCaml design recipe outlined in the previous homework.

For this and every future assignment, you should include the CS17setup.ml file with your hand-in. You should also have a README.txt file whose first line contains only your Banner ID, and optionally with a message to the person grading explaining anything peculiar about the hand-in. For example:

README.txt:
klein@brown.edu
There’s nothing to say except that I’m turning in these files plus this README the way the instructions say that I should.

To hand in your solutions to these problems, you must upload them to Gradescope. Do not zip or compress them. If you re-submit your homework, you must re-submit all files. If you choose to also store these files on department machines, all your solution files should reside in your ~/course/cs0170/homeworks/hw10 directory.

Set-Up

We have provided you with all of the files you will need for this homework assignment. The complete CS17setup file as well as outlines for the other files you will fill out for this assignment are in the hw10 src directory.

Before starting this assignment you’ll want to copy the contents of the hw10 src directory into your personal hw10 directory. On the department system, this would look something like this:

cp /course/cs0170/src/hw10/* ~/course/cs0170/homeworks/hw10

After you’ve transferred the files, begin filling in your solutions to the tasks on this homework. Make sure you do not modify any of the code we have written for you, especially not the #use statements. If you modify the template, there will be deductions.

Practice

1 Lists (Practice)

Here is a signature for lists.
module type LIST =
sig
  type 'a list

  (* Constructs an empty list *)
  val empty : 'a list

  (* Determines whether a list is empty *)
  val is_empty : 'a list -> bool

  (* Constructs a new list from a datum and an old list *)
  val cons : 'a -> 'a list -> 'a list

  (* Determines whether a list is non-empty *)
  val is_cons : 'a list -> bool

  (* Deconstructs a list, and produces its head *)
  val head : 'a list -> 'a

  (* Deconstructs a list, and produces its tail *)
  val tail : 'a list -> 'a list

  (* Produces the nth element of a list *)
  val nth : 'a list -> int -> 'a

  (* Appends two lists together *)
  val append : 'a list -> 'a list -> 'a list

  (* Reverses a list *)
  val reverse : 'a list -> 'a list

  ...
end

One possible representation of the type 'a list follows:

type 'a list =
  | Empty
  | Cons of 'a * 'a list

Task: Using this representation, implement the module type LIST.

Note: You can find this signature in /course/cs0170/src/hw10/sig_list.ml.

Problems

2 Monopoly Board with OCaml Records!

In this problem, we’ll create a basic representation of a Monopoly board using records, and implement a few different procedures to find information about the game. You’re familiar with tuples which
are compound types that contain multiple values. Records, also known as structs, also hold multiple values. However, records are unordered collections of values, with each value having a name, instead of a position.

Consider a tuple for a person. We could represent it as something like this:

("Philip", "Klein", 25, "RI")

It’s easy to forget which position in the tuple a piece of data represents. Instead, we could represent this as a record.

First, we declare a new type `person`:

```plaintext
type person = {
    first_name: string;
    last_name: string;
    age: int;
    state: string;
}
```

Then, we can construct a record:

```plaintext
{first_name="Philip"; last_name="Klein"; age=25; state="RI"}
```

**Task:** Define a record type, `place` that stores two fields: `location`, represented a string, and `cost`, represented as an integer.

**Task:** Define a record type, `player` that stores three fields: `piece`, a string representing the game piece a player is using, `cash`, an integer representing how much money the player has, and `places`, a list of places the player owns (so type `places list`).

Try constructing a couple of records of each type.

You should also know how to pattern match with records. Here is a procedure that takes a list of people, or in our case a list of `person`, and produces the cumulative age:

```plaintext
let rec sum_ages = function
  [[] -> 0 |
  {age} :: tl -> age + sum_ages tl;;
```

Let’s consider a slightly more complex procedure which converts a list of `person` records to tuples of name and age:

```plaintext
let rec name_age_list = function
  [[] -> [] |
  {first_name; last_name; age} :: tl ->
    (first_name ^ " " ^ last_name, age) :: name_age_list tl;;
```

With this understanding of records, we’re going to implement a few procedures to tell us about the state of a Monopoly game.
Task: Write a procedure, `most_expensive_place` that takes in a player record and outputs the option of a record of the most expensive place they own. If the player owns no places, the procedure should evaluate to None.

Task: Write a procedure, `greatest_net_value` that takes in a list of player records, and outputs an option of the player record with the greatest net value (the sum of all their places’ values and their cash). If the list is empty, the procedure should evaluate to None.

3 Dictionaries

As you begin Rackette, you’re going to become more familiar with the concept of environments. At a high level, environments keep track of the bindings between symbols and their respective values. Often, we will need to insert things into environments, look things up in environments, etc. One way to implement an environment is by using a dictionary.

A dictionary is a collection of key-value pairs. You will be implementing dictionaries in two ways: first with lists, and next using trees. For this homework, a dictionary is composed of \((\text{string} \times \text{int})\) tuples. That means the keys in the dictionary will be strings, and the values will be integers. In general, dictionaries can work on any types, but in this homework, we restrict it to just \((\text{string} \times \text{int})\) tuples.

3.1 Dictionary Signature

To start, we need to determine what functionality to expect from the dictionary.

Task: Write a module type that includes the following:

- A type `dict` for the dictionary.
- `new_dict`, which constructs an empty dictionary.
- `lookup`, which takes in a tuple of a dictionary and a key (of type string) and produces an int option. Specifically, it produces Some of the corresponding value, if the key is in the dictionary, and None otherwise.
- `keys`, which takes in a dictionary and returns a list of all the keys in the dictionary. Note that the list of all keys need not be in any particular order.
- `insert`, which takes in a tuple consisting of a dictionary, a key (of type string) not in the dictionary, and a value (of type int), and returns a new dictionary obtain from adding the given key-value pair to the given dictionary.

3.2 List Dictionaries

Task: Implement your module using lists of \((\text{string} \times \text{int})\) tuples to represent dictionaries. Specifically, use the following type:

```racket
type dict = (string * int) list
```
In testing your TestListDictionary, you’ll need to test its keys procedure. What if the keys are "x" and "y"? Then the keys procedure could return either ["x" ; "y"] or ["y" ; "x"]. What should you test against? Answer: if you sort the keys, you can just compare against a single standard answer, namely ["x" ; "y"]. OCaml includes a sorting procedure in its List module, but it requires, as a first argument, a function of two arguments x and y that produces −1 if x < y, 0 if x = y, and +1 if x > y. Fortunately, the designers of OCaml have included a function, called compare, that works on strings (as well as other data types). To sort a list of strings, you write

```ocaml
# List.sort compare ["y"; "x"];;
- : string list = ["x"; "y"]
```

That means that to test your keys procedure, you might write something like this:

```ocaml
check_expect (List.sort compare (TestListDict.keys myDict)) ["a"; "b"; "c"
 " ; "d"];;
```

When testing the functionality of TestListDictionary, you might run into a problem coming up with data of type dict to test your procedures with.

Task: To solve this, you should complete the code for kvpair_list_to_dict which takes in a (string * int) list and produces a dictionary. (note this procedure is defined outside the module)

This procedure should create a new empty dictionary and must call insert for each pair in the list, building a dictionary that contains all the pairs in a given list. You can then use this procedure within your other test cases. Note, this procedure should call insert for each pair in the list in the order they are given in the input list.

### 3.3 Tree Dictionaries

#### 3.3.1 Binary Search Trees (BSTs)

Although a list-based dictionary will meet the dictionary specification we require, it’s not necessarily the most efficient choice. In the list implementation, in order to look up an individual key in the environment, we might have to go through every key in the dictionary, taking linear time. Another way to represent a dictionary is to use binary search trees, which would make this procedure run in log n time on average.

If a binary tree is a leaf, then it is automatically a BST. If the tree is an internal node, then the following invariants must hold:

- Every value in the node’s left subtree must be less than the nodes value; and
- Every value in the node’s right subtree must be greater than the nodes value.

#### 3.3.2 Task

Task: Implement your module using BSTs of string * int tuples to represent dictionaries. Specifically, use this type:
type dict =
  | Leaf
  | Node of ((string * int) * dict * dict)

Make sure to take advantage of the binary search tree structure of your dictionary in your lookup procedure.

**Note:** There are a few different ways to implement these procedures. As long as a) your tree maintains the invariants above, b) the runtime of your procedure is reasonable, you can receive full credit.

**Task:** Copy your code for `kvpair_list_to_dict` which takes in a `(string * int)` list and produces a dictionary. The code should be the same for both this and the previous problem. You can again use this procedure within your other test cases.

### 3.4 Timing Experiments

We now have two implementations of dictionaries, and we’d like to compare the performance. We’ve discussed how lookup take $O(n)$ time in the list implementation, but $O(\log(n))$ time on average in our binary search tree implementation.

To help see the real consequences of this, we’ve included two files in the stencil, `time_list_dict.ml` and `time_tree_dict.ml`. Both use your respective implementations, create a dictionary of 15,000 key-value pairs, and then perform lookup on all of them. We divided each file into two parts - the first which inserts everything into the dictionary in random order, and the second which inserts everything into the dictionary in a sorted order.

To run these tests, we time your lookup function. We use a Unix library to do this, which isn’t linked by default. As such, you’ll need to run the following command to run our tests:

```
ocaml unix.cma < time_list_dict.ml

ocaml unix.cma < time_tree_dict.ml
```

**Task:** Open up and try to understand what our code in `time_list_dict.ml` does. Try running it, and seeing how long it takes to lookup all the items. Note, it takes some time to construct the dictionary, and as such, we are only recording the time between the first and last lookup.

**Task:** Comment out part one of `time_list_dict.ml`, and uncomment part two. Part two of this file inserts the items into the dictionary in a sorted order, and then does the same time measurement.

**Task:** Repeat the previous two tasks on `time_tree_dict.ml`.

**Task:** In a file call `analyze_dict.txt`, discuss the actual runtime implications. That is, discuss which implementation (if either) ran faster on the dictionary where we inserted everything in random order, and discuss which implementation (if either) ran faster on the dictionary where we inserted everything in order. Explain why this is the case.

**Note:** You may need to run this on a department machine depending on the exact configuration of your machine.
3.5 A Note on Modules

While writing these procedures, you may notice that equivalent procedures between implementations have different runtimes. Part of the reason we implemented dictionaries using BST’s in the first place was to get the faster $\log n$ lookup time, as opposed to the linear lookup time from the list implementation. However, in getting this faster runtime, the code in the tree implementation is much more complicated than in the list implementation. As the number of bugs is proportional to the number of lines of code, this makes the tree version more difficult to maintain and debug.

A part of almost every problem in computer science is deciding what data structure to use, while considering these tradeoffs.

For example, in an application where many small dictionaries are being created, it might be more efficient to use the list implementation. In an application where you are looking up keys in an enormous dictionary, a tree implementation is the way to go. Knowing which data structures to use in specific scenarios comes with practice and knowledge of how long the relevant operations take in those data structures.

Modules allow us to seamlessly switch between different implementations, without changing how users interact with our code. This allows us the ability to create multiple implementations, each of which might be useful in a different situation.

Please let us know if you find any mistakes, inconsistencies, or confusing language in this or any other CS 17 document by filling out the anonymous feedback form: http://cs.brown.edu/courses/csci0170/feedback.