CS320: Survey Of Programming Languages

Due: TBA

Requirements

- You can use all the built-in functions and function in `List` module
- You can implement any number of helper functions as you see fit.
- You can add or remove `rec` as you see fit.
- Please do not change the type signature and the order of the input in the functions that are provided in the template .ml file.
- Delete the `failwith "unimplemented"` before you start working on a function.
- If you fail to implement one of the function, replace your incomplete implementation with `failwith "unimplemented"`, this will prevent any syntax or type checking error when you are trying to run your entire assignment.
- If we encounter syntax/type checking error while running your code, You will not receive *any* credit for this *entire* assignment.
- All correct implementations will not throw any error/exception.

1 File IO

For this question, you will get familiar with some basic read/write from a file.

1. `let write_list_str (f_name: string) (content: string list): unit`.

   This function writes a list of strings (provided as the second argument) in a file (the name of the file is provided as the first argument). The strings have to be written one per line following the order left to right in the list. The function returns `unit`.

2. `let read_list_str (f_name: string): string list`.

   This function takes the name of a file and reads its lines into a list of string, where each line is an element in the list.

   Here is an example, assume that the file contains:

   ```
   Hello
   World
   CS320
   ```

   then the function must return `["Hello"; "World"; "CS320"]`.

2 Functions over ciltrees

We now provide you with a very specific type called `ciltree` and the definition of this type is as follows:

```
type ciltree = L of int list | I of int | T of ciltree * char * ciltree
```

The above definition of a `ciltree` basically tells you that a value of type `ciltree` can be any of the following:

1. a `list of int`
2. an `int`
3. a triple containing a left `ciltree`, a `character`, and a right `ciltree`

Here some examples of `ciltree` values:

```
ex1 = T (T (I 1, 'a' , T (I (-34), 'b', L [-21; 53; 12])), 'c', T (I (-18), 'd' ,
I 1))

ex2 = T (T (T (I 31, 'h', L [9; 34; -45]), 'e', L [70; 58; -36; 28]), 'l', I 3),
      'l', T (I 2, 'o', I 49))

ex3 = T (T (T (L [9; 4; -1; 0; -5], 'c', L [40]), 's', I 1), '3', T (L [420; 69],
      '2', I (-3)))
```


We will use these examples later in this handout to illustrate the functions that you are required to implement.

1. `count_ints (tree: ciltree): int`.

   Implement `count_ints` whose behavior is to recursively traverses the input `ciltree` tree and count how many `int` it contains. We ask you to review the definition of `ciltree` again, and you will notice that `int`s are represented by (nodes with constructor `I`).

   For example, if we run `count_ints` on the previously defined examples we have:

   ```
   count_ints ex1 = 4
   count_ints ex2 = 4
   count_ints ex3 = 2
   ```

2. `map_on_all_lists_elem (func: int -> int) (tree: ciltree): ciltree`.

   Implement `map_on_all_lists_elem` which should traverse a `ciltree` tree and apply `func` on all the *elements* of list nodes in tree.

   Here are three examples on the behaviour of this function:

```
map_on_all_lists_elem (fun n -> n * 2)
  (T (L [1;2;3], 't', L [3;2;1])) = T (L [2;4;6], 't', L [6;4;2])

map_on_all_lists_elem (fun n -> 0)
  (T (T (L [1], 'a', I 10), 't', L [3; 2; 1])) =
  T (T (L [0], 'a', I 10), 't', L [0; 0; 0])

map_on_all_lists_elem ((+) 1)
  (T (T (L [1], 'a', I 10), 't', L [3; 2; 1])) =
  T (T (L [2], 'a', I 10), 't', L [4; 3; 2])
```

3. `shift_right (tree: ciltree): ciltree`.

   Problem:[1]

   Implement the `shift_right` function to perform the following operation to a `ciltree` with the required structure (right most branch contains no less than 3 nodes, and the left most branch contains no less than 2 nodes)

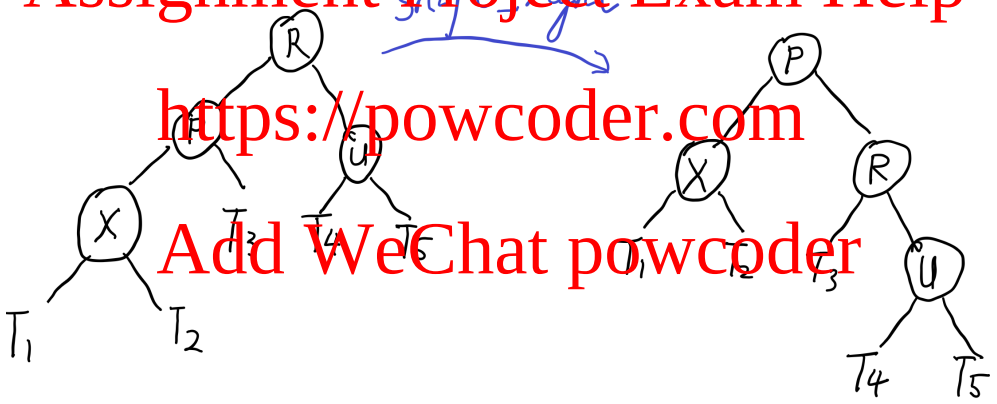

Figure 1: Here we perform a *shift_right* on the node R
. You can assume that T_1, T_2, T_3, T_4, T_5 are some subtrees.

If the tree do not have the required structure, then return the original tree

[1]**Background (optional read, not necessary for solving the problem):** Tree manipulation is very common in implementing balanced tree. In functional programming, we use the power of pattern matching to manipulate tree, instead of the error-prone pointer manipulation. What we are implementing is the left shift operation on tree, which is used in implementation of red black tree (left left case of `https://www.geeksforgeeks.org/red-black-tree-set-2-insert/` red black tree insertion). We will ignore the color change for this problem, since we don't have colors in our tree.

Examples:

```
(*no operation done, because of the left most branch is too short*)
shift_right (L [1; 2; 3]) = (L [1; 2; 3])


shift_right (T (L [1], 't', T (I 1, 'r', T (I 69, 'e', L[14])))) =
  (T (L [1], 't', T (I 1, 'r', T (I 69, 'e', L[14]))))


(*the minimum tree to shift*)
shift_right (T( T (I 12, 'b', L [1]), 'c', T (I 12, 'd', L [34; 96]))) =
T( I 12, 'b', T (L [1], 'c', T (I 12, 'd', L [34; 96])) )


(*a general tree*)
shift_right (
  T(
    T (
      T (I 1, 'r', T (I 69, 'e', L[14])),
      'b',
      T (L [34], 'r', T (L [420;69], 'e', I 44))
    ),
    'c',
    T (T (I 44, 'm', L []), 'd', L [])
  )
) =
T(
  T (I 1, 'r', T (I 69, 'e', L[14])),
  'b',
  T (
    T (L [34], 'r', T (L [420;69], 'e', I 44)),
    'c',
    T (T (I 44, 'm', L []), 'd', L [])
  )
)
```


4. `tree_contains (tree: ciltree) (subtree_to_find: ciltree): bool`

   Given a tree and a subtree to find, `tree_contains` will return whether the given `tree` contains a subtree (a node with all of its descendent) exactly equals the input `subtree_to_find`.

   Examples:

```
(*a tree always contains itself*)
tree_contains (L [1, 2, 3]) (L [1,2,3]) = true
```

```
(*the right most subtree*)
tree_contains (T (L [1], 't', T (I 1, 'r', T (I 69, 'e', L[14]))))
  (T (I 69, 'e', L[14])) = true


(*we can just find one single leaf*)
tree_contains (T (L [1], 't', T (I 1, 'r', T (I 69, 'e', L[14]))))
  (I 1) = true

(*there is no subtree that exactly matches the input*)
tree_contains (T (L [1], 't', T (I 1, 'r', T (I 69, 'e', L[14]))))
  (T (L [1], 't', I 1)) = false
```

3 cil and ciltree

For this question we define a type `cil` as follows:

```
type cil = L of int list | I of int | C of char
```


We ask you now to finish the following tasks:

1. Create a `cil_gtree` which is a general tree[2] where each node in the `cil_gtree` is a `cil`. A non-empty `cil_gtree` would contain both

   - a root node
   - a ordered list of children of the root (where each child in the list is a `cil_gtree`).

   Write the following functions:

   ```
   (*Encoding data to your data type*)
   let mk_empty_cil_gtree: cil_gtree
   let mk_cil_gtree (root_val: cil) (children: cil_gtree list) : cil_gtree

   (*Decoding data from your data type*)
   let cil_gtree_root (tree: cil_gtree): cil option
   let cil_gtree_children (tree: cil_gtree): cil_gtree list option
   ```

   where

   - `mk_empty_cil_gtree` constructs an empty tree.
   - `mk_cil_gtree` constructs a `cil_gtree` by taking in as input the root node value and and all its children. The children of the root node are passed in as a list of `cil_gtree`
   - `cil_gtree_root` returns the root element of the input tree. Think carefully on when to return None.
   - `cil_gtree_children` returns the children of the node as a list. Think carefully on when to return None.

[2] `https://en.wikipedia.org/wiki/Tree_(data_structure)`

2. Implement the following two functions to convert `ciltree` to and from `cil_gtree`

```
let rec cil_tree_to_cil_gtree (tree: ciltree): cil_gtree
let rec cil_gtree_to_cil_tree (tree: cil_gtree): ciltree option
```

When implementing these functions make sure that you cover all the cases of `ciltree` and `cil_gtree`. Note the return types from `let rec cil_tree_to_cil_gtree` and `let rec cil_gtree_to_cil_tree`. Ask yourself why is it important for `let rec cil_gtree_to_cil_tree` to return back a `ciltree option` and not `ciltree`.

4 Submission Instructions

Late submissions will not be accepted. You can use Gradescope to confirm your program adheres to the specification outlined. Only your last Gradescope submission will be graded for your final grade. This means that you can submit as many times as you want before the deadline. If you have any questions, please ask well before the due date on Piazza.

4.1 Gradescope submission instructions

When you log into Gradescope and pick the CS 320 course you will see an assignment called ciltree. You will need to submit a solution written in Ocaml to have full credit for the assignment. The total of points for this assignment is 50. Your program must be a .ml file.

4.2 Additional Resources

For information on how to run OCaml please see the following references:

- `https://caml.inria.fr/pub/docs/manual-ocaml/stdlib.html`
- `https://caml.inria.fr/pub/docs/manual-ocaml/`

You will find resources for these languages on the Piazza course web site, under the Resources page.