Problem Set 7

Deliverable: Submit your responses as a single, readable PDF file on the collab site before **6:29pm** on **Friday, 27 Oct**.

Collaboration Policy (identical to PS5)

For this assignment, you may work in groups of one to three students to write-up a solution together. If you work with teammates, exactly one of your should submit one assignment that represents your collective best work with all of your names and UVA ids clearly marked on it on it. *Everyone on a team should understand everything you turn in for the assignment well enough to be able to produce it completely on your own.* All teammates must review the submissions before it is submitted to make sure you understand everything on it and that your name and UVA id are clearly marked on it.

Preparation

This problem set focuses on recursive data types and structural induction — read Chapter 7 of the MCS book, and Class 15 and Class 16.

Directions

Problems 1–8 are expected for everyone; solve as many as you can. The Programming with Procedures problems are optional (see the note before them).

Tsilly Lists

Consider an alternate way of defining a list from the one we used in Class 15 and 16, where instead of *prepend*, lists are constructed using *postpend* (to avoid confusion, we call our postpended list a *tsil*, and reserve *list* for the original prepended list):

> A *tsil* is either the empty tsil (λ), or the result of postpend(t, e) for some tsil t and object e.

1. Define the meaning of the following operations (similarly to the beginning of Class 16) for the tsil: last : $Tsil \to Object$, frest : $Tsil \to Tsil$, and empty : $Tsil \to Boolean$.
2. Provide a constructive definition of *length* for the tsil data type.
3. Prove that there is an equivalent tsil for every list. (Your answer should include a clear definition of what *equivalent* means.)

Structural Induction on Trees

In Class 16, we defined a recursive data type, lists, that only relied on *one* (smaller) list in its recursive definition. Here we give a recursive definition for the data type of *binary* trees. In a binary tree, each node can have 0,1,or 2 children, and each node has a label of its own that (in this case) is always a natural number.

The main two operators corresponding to the base and the construct cases are are:

$$\begin{aligned} &\textbf{null} : \textit{Tree} \\ &\text{node} : \textit{Tree} \times \mathbb{N} \times \textit{Tree} \to \textit{Tree} \end{aligned}$$

We also define the following operations on binary trees.

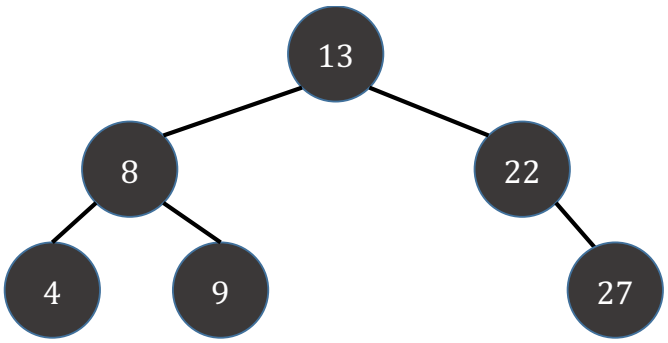
The meaning of the operations is defined for all trees t_1, t_2, and all $n \in \mathbb{N}$, by:

$$\begin{aligned} \text{label} : \textit{Tree} \to \mathbb{N}: &\quad \text{label}(\text{node}(t_1, n, t_2)) \to n \\ \text{left} : \textit{Tree} \to \textit{Tree}: &\quad \text{left}(\text{node}(t_1, n, t_2)) \to t_1 \\ \text{right} : \textit{Tree} \to \textit{Tree}: &\quad \text{right}(\text{node}(t_1, n, t_2)) \to t_2 \\ \text{empty} : \textit{Tree} \to \{\mathbf{T}, \mathbf{F}\}: &\quad \text{empty}(\textbf{null}) \to \mathbf{T} \\ &\quad \text{empty}(\text{node}(t_1, n, t_2) \to \mathbf{F} \end{aligned}$$

4. The *height* of a tree is the *maximum* distance (number of edges) from its root (the one node that has no parent node) to a leaf. For example the hight of the tree at the bottom of this page is 2. Provide a *constructive* definition of *height* for our binary tree type. (Hint: the height of node(**null**, n, **null**) is 0. The height of **null** should also be 0.)

5. Prove that the maximum number of nodes in a binary tree of height h is $2^{h+1} - 1$.

The in-order traversal of a binary tree is a list of the node labels in the order they appear from left-to-right across the tree. For example, the in-order traversal of the tree shown below would be the list $(4, 8, 9, 13, 22, 27)$.



We can define traverse to produce a (prepend) list as follows (note the $+$ operation here is list concatenation, as defined in Class 16) :

$$\text{traverse}(\textbf{null}) = \textbf{null}$$
$$\text{traverse}(\text{node}(t_1, n, t_2)) = \text{traverse}(t_1) + \text{prepend}(n, \text{traverse}(t_2))$$

6. Prove that for all trees t with n nodes, the result of traverse(t) is a list of length n.

Ordered Binary Trees

Here we would like to define an *OrderedBinaryTree* as a data type where for each node with label n, all of the children in the left sub-tree have labels *smaller* than n, and all the children in the right sub-tree have labels *larger* than n.

– **Base case: null** $\in$ *OrderedBinaryTree*.

– **Constructor case:** if $t_1, t_2 \in$ *OrderedBinaryTree* and $n \in \mathbb{N}$, and $(\text{empty}(t_1) \vee \text{maximum}(t_1) < n)$ and $(\text{empty}(t_1) \vee \text{minimum}(t_2) > n)$, then $\text{node}(t_1, n, t_2) \in$ *OrderedBinaryTree*.

You may assume all the other tree operations (including traverse from question 5) are defined for *OrderedBinaryTree* s also.

7. Define the minimum : *OrderedBinaryTree* $\to \mathbb{N}$ and maximum : *OrderedBinaryTree* $\to \mathbb{N}$ operations used in the definition of *OrderedBinaryTree* above.

8. ($\star$) Prove that $\forall t \in$ *OrderedBinaryTree*. traverse(t) is an ordered list. (A list, $p = (p_1, p_2, \cdots, p_n)$ is an ordered list if $\forall i \in \{1, \cdots, n-1\}. p_i < p_{i+1}$.)

Programming with Procedures

> These problems are *optional*, and provided to give students who are interested some experience with functional programming which will make you a more powerful, snazzy, and prolific programmer. You do not need to do them to earn "gold star" level credit on this assignment, and nothing on the exams will depend on them. You will receive "bonus" credit on this assignment for turning in good answers to these questions.

These questions assume you have some experience programming in Python, but are sadly lacking in previous experience using procedures as parameters and results and realize that you cannot be a true kunoichi programmer without becoming adept with programming with procedures.

Download: pairs.py (if the link in the PDF file doesn't work, use *https://uvacs2102.github.io/docs/pairs.py*).

In pairs.py, we defined `make_pair` and various procedures for building and using lists. You should download this code, run it in your favorite Python3 environment, and make sure you understand it.

9. Define a function `list_tostring(lst)` that takes a list (constructed using the `list_append` function from `pairs.py`) as its input and returns a string representation of that list. For example, `list_tostring(list_prepend(1, list_prepend(2, list_prepend(3, None))))` should print out `[1, 2, 3]`. (You can use `str(x)` to turn any Python object `x` into a string.)

10. Define a function `list_map(fn, lst)` that takes as inputs a function and a list, and returns a list that is the result of applying the input function to each element of `lst`. For example, `list_map(lambda x: x + 1, list123)` should return the list `[2, 3, 4]`.

11. Define a function `list_accumulate(fn, lst, base)` that takes as inputs a function, a list, and a base value, and returns the result of applying the function through the list. For example, `list_accumulate(lambda a, b: a + b, lst, 0)` should return the sum of all the elements in the list, and `list_accumulate(lambda a, b: a * b, lst, 1)` should return their product, and `list_accumulate(lambda a, b: b + 1, lst, 0)` should return the length of the list.

12. Define `list_map` (as in problem 10) using `list_accumulate`.

