CMPT 383: Vitamin #3

Anders Miltner
miltner@cs.sfu.ca

Due Oct 4

Introduction

This Vitamin is here to get you accustomed to defining your own data types to represent data.

This submission will be autograded. There are some portions of the assignment that are ungraded, and some that will be graded. We provide a (partial) test suite for partial validation. You can run these tests by opening a terminal in the v2 directory, and running `stack test`.

We have included explicitly some imports, and have omitted some. If you import additional functions, you may get a zero on the assignment. Specifically, we have imported the "elem" and "nub" functions. These functions will be very useful to use, and we suggest searching them on Hoogle.

1 Defining Your Data Type

In this Vitamin you will create a data type for a "Switchable Stack." A switchable stack acts as a standard LIFO stack. Namely, the most recently pushed element should be the first one popped. However, this stack can be in two states: active or inactive. If the stack is active, it acts as a normal stack. If the stack is inactive, elements can be pushed onto it as usual, but when popping – it behaves like an empty stack.

There is one extra caveat for this stack, it does not permit duplicates! If an element is going to be added to the stack, make sure it is not already in the stack. If it is in the stack, do nothing. If it is not in the stack, add it in the standard LIFO way.

Your first task is to define a data type for the stack. You can use any built-in data types that do not require additional imports. This includes, but is not limited to: lists, Maybe, Bools, Ints, and Floats.

2 Basic Switchable Stack Functions

Next you should build some basic functions for the switchable stack. These functions are empty, push, pop, setInactive, and setActive. The types of these functions are provided for you. A description of these functions is provided inline, and a partial test suite is provided in the test folder.

3 Advanced Switchable Stack Functions

Next you should build some basic functions for the switchable stack. These functions are mapState and popWhere.

The mapState function should edit all the elements in the stack with the provided function. Thus, if the function is from type a to type b, the resulting stack should have elements of type b in it. Remember to maintain the no-duplicates invariant! After applying the function, you should be sure to remove duplicates.

The popWhere function breaks the LIFO nature of the stack. It pops all the elements that satisfy the provided predicate. Make sure not to return anything, nor edit the stack, if the stack is inactive.