Damian Franco

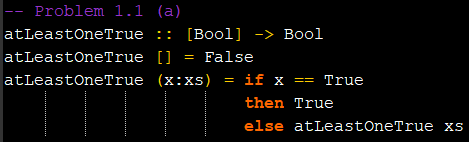
dfranco24@unm.edu

101789677

CS-558 001

Homework 1

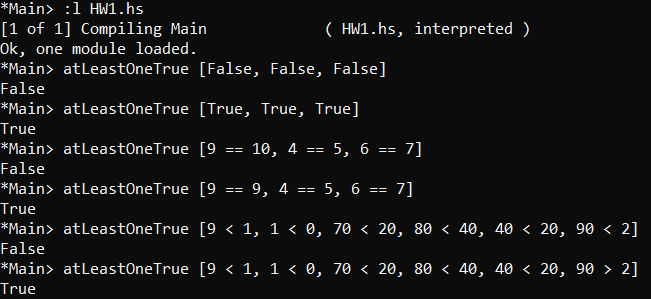
1.1 Functions on lists

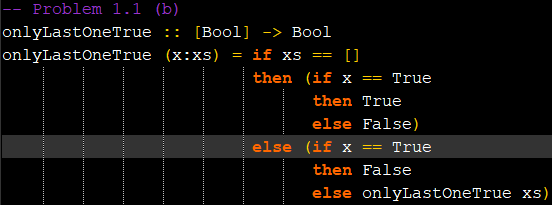
1. For this problem we were tasked to write a function that would parse through a list of type Bool to check if at least one of the boolean expressions within it is true. This function also returns type Bool where False would be returned if there are *no* True boolean expressions are found and True would be returned if there is at least one True boolean expressions are found within the list. Below is the code to the function that was written to fulfill the requirements.

The function above takes a list of booleans and returns a boolean stating whether there is at least one true boolean expression within the list. This function uses recursion to achieve the parsing through the elements of the list.

If a *non*-empty list is passed as an argument then the function will first check if the head of the list *x* is a True boolean expression. If it is True then it can be assumed that there is at least one true boolean expression in the list which would return the boolean True and stop the recursion. If the current head is False then recursion will be used on the tail of the list *xs* to parse through to the rest of the list to check if any other elements are True.

If an empty list is passed as an argument of the function then that indicates the end of the list. If the end is reached then it can be assumed that there were no True boolean expressions due to the if-then-else condition checking on the elements before. This would return False due to the lack of True boolean elements.

To test this function, I used lists of varying sizes with some lists consisting of multiple True boolean expressions, some with exactly one True boolean and others with no True booleans. Below you can see some of the test cases I used to test the functionality of the function. Through these tests, I concluded that this function is fully functional.

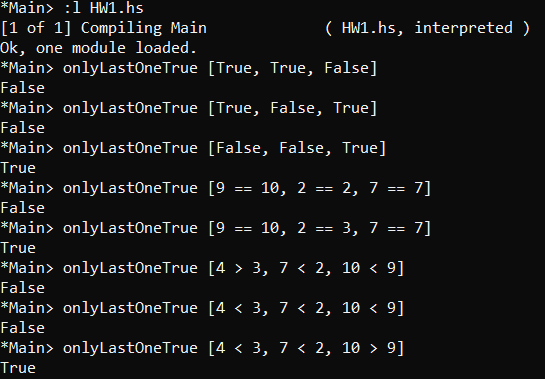
1. This problem asked us to write a function that would check if the last element of a given list is a True boolean expression. The function would return True if and *only if* the last element of the list is True, if not then it would return false. Below is the code for the function.

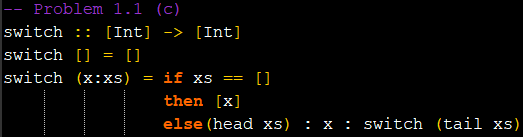
The function above will take a list of boolean expressions as an input and return a boolean stating True or False based on the final element of the list. There is no guard to this function when it comes to checking for an empty list because that is handled within the first if statement when checking if the tail is empty. This function uses recursion to parse through the list of booleans.

First, the function checks if the tail *xs* of the input list is empty. This effectively checks if the head *x* is the last element of the list or if there is a need to keep parsing through the list.

If the tail is empty, then it can be assumed that the current head *x* is the last element of the list. The function will then check if the last element of the list *x* is True or False and will return True if it is True and False otherwise.

If the tail is *not* empty then the function needs to check the current head to see if that boolean expression is True. If the current element is True then we will stop the recursion and return False. We do this since we only want to return True if and *only if* the last element is true and this is telling us that there are other True elements within the list. If the current head is False, then we will recurse with the tail of the list *xs* to continue to parse through the list.

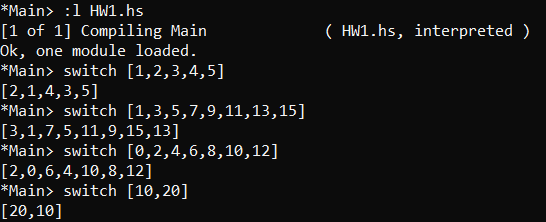
To test this function, I used similar lists that I used for problem (*a*) to test the functionality. With the use of lists of varying sizes with some lists consisting of a single True boolean expression as the final element of the list, some with True booleans at other indices that were not the last index and others with no True booleans. Below you can see some of the test cases I used to test the functionality of the function. Through these tests, I concluded that this function is fully functional.

1. This problem asked to create a new function that will take a list of integers and return a list of integers with a switch of the element positions in the new list. The current element must switch positions with its neighbor to its right and this will continue with the next two elements of the list until there are no more elements that can switch. Below is the code for this function.

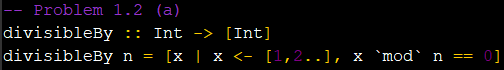
The function above uses recursion to parse through the list of integers. There is a strong use of the “cons” operator (*:*) that helps switch the elements of the list. Also other functions like head and tail are used to get certain elements of the list.

First a guard was added to stop the occurrence of an infinite loop by checking if an empty list is being passed through. If an empty list is the parameter being passed through then an empty list will be returned which will then be appended to the end of the list.

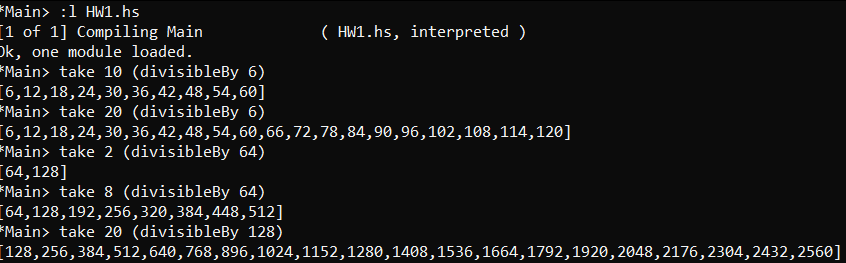
If a *non*-empty list was passed as an argument of this function then there is a check on the tail of the list *xs*. If the tail of the list is empty then the function will return a list consisting of nothing but the head of the list *x*. This is returned to append the final element of the input list to the new list. If the tail of the list is notempty then the “cons” statement of the new list will be run. This statement will first add the head of the tail of the list (neighbor of current head) as the current element to append to the list. Then the current head will be added which will effectively do the “switch” of neighboring elements to the list. Lastly, a recursive call will be appended to the list to continue to recurse through the list until there are no more elements to switch.

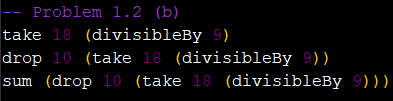
To test this function, I used the example test that was given in the homework write up and tests that were similar to the example. This allowed me to track where the functionality succeeds and fails. Some tests had an even number of elements and some had an odd number of elements to check for failure in functionality as well. Through all of the tests below, you may see that all will successfully switch elements with their neighbors. 

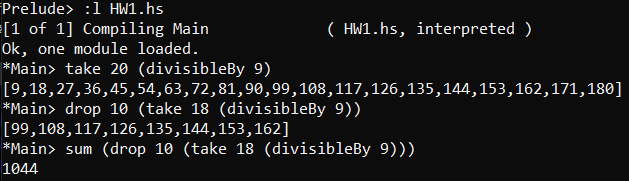
1.2 Infinite Lists

1. For this problem we were tasked to create a function that takes an integer input and returns a list of integers that are exactly divisible by the input integer. The function must use a list comprehension to enable the use of infinite lists to find the correct output. Below you can see the code I wrote to achieve this goal.

First the problem is taking an input *n* which is of type integer and returns a list of integers as well. The list compression first only returns the element *x* from the infinite list where *x* is an element of the infinite list. The filter here on the end of the list comprehension statement is doing all the heavy lifting here. This is where the input element is used to filter out every element that is exactly divisible by *n* using the mod function. If *x* modulus *n* is equal to 0 then it can be assumed that that element is exactly divisible by *n*. That will then be “filtered” through to the *x* that is returned within the list. That is how I achieved this problem's goal.

For testing this function I used the take function to avoid infinitely looping through a list of the output. By inputting multiple integers into the divisibleBy function, we can see if the functionality is correct when we take some elements from the final list. I also noted that the output list is an infinite list as well because it can print out any number of elements within the list. All the numbers below seem to indicate that the functionality is correct.

1. For this problem, the objective was to use the divisibleBy function to calculate the sum of the 11th through the 18th number of the list that is returned when you enter the integer 9 in the divisibleBy function. Below you can see my process on how I achieved the goals of this problem. 

The first line of haskell code in the picture is a test to get a list of length 18 with elements that are divisible by 9. This effectively lets me get the 18 elements which fulfills the first goal of the function. The next line then drops the first 10 elements of the list of 18 elements so the list will then have the 11th through the 18th element. The last list will then simply call the sum function which sums all the elements within a list. Below you can see the test I used to check functionality of this statement.

1. Within part (b), an infinite list is utilized to create a list of all the possible integers that are exactly divisible by the input *n*. The use of an infinite list is possible in Haskell due to the programming language using lazy evaluation. Lazy evaluation indicates that expressions in Haskell are not evaluated until their results are needed by other computations or called upon by the source code. Infinite lists are able to be used due lazy evaluation, otherwise memory usage would be impossibly large to handle, but there is no memory usage until the infinite list is called upon. This is why there was an ability to take a call for an exact amount of elements within the list when I was doing the take functions without having a memory overload. Laziness is a very important part of Haskell programming and I expect that we will continue to explore it more within the class.