CSC330 Summer 2004 Assignment 2

George Tzanetakis

May 28, 2004

1 Overview

PLEASE READ CAREFULLY THIS DESCRIPTION AS MISSING SMALL DETAILS MIGHT COST YOU

A random 25% sample of the students will be orally interviewed by me regarding their submission. Make sure you have written everything you submit. For any function you are trying to write, you can request what is the desired output for a particular input by email. Solve the parts of the assignment in the order they are given as they get progressively harder. If you are stuck somewhere move to the next part and come back later.

The careful design and documentation (i.e comments) of your code will be important factors in your grade. If there is a bug it's better to report it than hide it. Be honest, precise and clear and you shall be rewarded. NEVER (at least in this class) sacrifice clarity for efficiency of execution unless explicitely asked to do so. PROVIDE A SET OF WELL-DESIGNED TEST CASES FOR EVERY PART OF THE ASSIGNMENT.

The assignment is worth 10 points (10% of the final grade) and will be graded in using half-point intervals. The overall documentation and packaging of your submission according to the instructions provided in the web page (READ THEM CAREFULLY) will be worth 1 point. Please submit ONLY one file named **assignment2.sml** with the source code for all the parts clearly documented. *Late assignment policy*: If the assignment is submitted within three days from when it was due, you will get half the grade you would get if you had submitted on time. You will not be able to submit after three days. (exceptions to this rule by request)

The main goal of the assignment is to familiarize you with SML and especially the use of higher-order functions and list recursion. I hope you find the assignment interesting and helpful.

2 Part 1 (1 pt)

Write the following support functions that you might need in the other parts of the assignment. Helpful builtin functions that you might need to use are: explode (converts a string to a list of characters, implode (converts a list of characters to a string) and length (returns the length of a list).

- *tabulate(i,j)* that returns the list of integers from i to j. For example tabulate(2,8) is [2,3,4,5,6,7,8]
- *min(x,y)* that returns the minimum of two numbers and based on it *min3(x,y,z)* that returns the minimum of three numbers. For example min3(5,2,3) is 2.
- *zipwith f (xs,ys)* that takes as argument a function of a 2-tupple and two lists and applies the function by taking one item from each list forming a tuple, applying the function and returning a list of the results. If the lists have unequal lengths the extra items that can not be paired are ignored. For example zipwith (fn(x,y)=>x+y) ([1,2,3],[1,2,3]) is [2,4,6] and zipwith (fn(x,y)=¿x+y) ([1,2,3],[1,2,3,4,5]) is also [2,4,6];
- *zip (xl,yl)* that takes as argument a tupple of lists and returns a list of tuples formed by pairing each corresponding elemnt of the lists. For example zip([1,2,3],["a","b","c"]) is [(1,"a"), (2,"b"), (3,"c")]. If the lists have unequal lenghts the extra items that can not be paired are ignored. Use the function *zipwith* to define zip.
- *zipcount(ls)* that returns a list of tupples who first element is a counter and the second element is the list item. For example zipcount([#"a",#"b", #"c"]) is [(0,#"a"),(1,#"b"),(2,#"c")].
- *copy(n,ls)* that returns a list of n repetitions of ls. For example the result of copy(4,[1,2,3]) is [1,2,3,1,2,3,1,2,3,1,2,3]
- *dropwhile p ls* which takes as argument a predicate (a function that returns a boolean value) and drops the first items of a list that match the predicate. For example the result dropwhile (fn(x)=¿x mod 2 = 0) [2,4,5,2,4] is [5,2,4].

3 Part 2 (2 pt)

In this part you will implement two well known sorting algorithms and compare their performance for sorting list of real numbers.

The following code can be used to generate a list of 30000 real numbers.

```
local val a = 16807.0 and m  = 2147483647.0
in fun nextrand seed =
       let val t = a * seed
       in t - m * real(floor(t/m)) end
end;

fun randlist (n,seed,tail) =
    if n=0 then (seed, tail)
    else randlist (n-1, nextrand seed, seed::tail);

val (r,ls) = randlist(30000,5.0,[]);
```

Your task is to implement two function *insort* and *quicksort* that can be used to sort the ls list of real numbers. Both insort and quicksort will have type *fn: real list -> real list.* In both cases sorting is in ascending order.

The idea behind insertion sort is simple. Items are picked from the input list and are added to a new list maintaining the invariant that the new list is sorted. If the item to be inserted is less than the first item of the sorted list under construction it can be appended on the front. If it is not it is inserted the tail of the remaining under construction sorted list.

Quick sort, invented by C.A.R Hoare, was among the first efficient sorting algorithms. It works by divide and conquer:

- Choose some value a, called the **pivot** from the input
- Partition the remaining items into two parts: the items less than or equal to a, and the items greater than a
- Sort each part recursively, then put the smaller part before the greater

To implement quicksort use a helper function partition (for each partition step) that takes as input a left partition list, a right partition list and the items that haven't been visited. In each call to partition the head of the items is appended to either the left or right partition depending to it's relation with pivot a until the items list is empty.

Compare the performance of insort and quicksort for a large lists of random numbers. Which one is faster ? Why ? (try to explain informally not just say one is O(something) and the other is O(somethingelse))

Examples of running the functions:

```
- insort([5.0,2.0,3.0,1.0,4.0]);
val it = [1.0,2.0,3.0,4.0,5.0] : real list
- quicksort([5.0,2.0,3.0,1.0,4.0]);
val it = [1.0,2.0,3.0,4.0,5.0] : real list
```

4 Part 3 (3 pt) Big Numbers

(IMPORTANT YOU ARE NOT ALLOWED TO USE DIRECT RECURSION IN THIS PART - ONLY HIGHER-ORDER FUNCTIONS SUCH AS foldl, foldr, map)

The bultin operations of arithmetic in most programming languages can only handle integers in some restricted range. For numbers outside this range the operations are not well-defined. One way round the problem is to construct our own package of functions for computing with integers of arbitrary size. In this part, your task is to define variable-length non-negative integer addition and multiplication.

The main idea is to represent a non-negative integer x as a non-emply list of "digits". So for example the number 145 will be represented as [1,4,5]. The is the usual way of representing numbers with the most significant digit first. This representation is not unique since an arbitrary number of leading zeros can be added to an integer without changing its value.

Write a function *strep* that removes the leading zeros from a list. Write a function *allign* that takes two numbers in list format and adds the necessary leading zeros to make them equal in length. Now we are ready to define function *vadd* which adds two variable-length large non-negative integers.

The idea is to allign the two numbers and add them digit by digit the result then needs to be normalized by propagating the carry to ensure that each digit is less than 10. For example [9,5] ++ [5] is [9,10] which needs to be normalized to [1,0,0]. The carry needs to be propagated from the least significant digit to the most significant digits (Hint: use foldr) and any leading zero must be removed using *strep*.

First write a function *vadd* that takes two variable-length non-negative integers represented as lists and performs the addition. Then convert that function to the infix ++ so that you can type in the ml prompt: [9,5] ++ [5] and it will work.

The next task is to write function *vmul* that multiplies two variable length non-negative integers represented as lists. Similarly convert vmul to an infix version using the ** symbol. For example: [2,0] ** [5] is [1,0,0]. The number after the infix keyword is precedence so by doing infix 6 ++ and infix 7 ** you will get the correct precedence. For example [5] ++ [2] ** [3] will be [1,1] instead of [1,0].

To implement multiplication write the following helper functions: *dmul* multiply an integer represented by a list with a single digit. Use a higher-order function to apply *dmul* for each digit of one number and the other number creating the list of partial sums. (do a multiplication on paper to remember the partial sums). Then the last step is to go over the list of partial sums shifting appropriately and adding using *vadd*. Again use a higher-order function for that purpose.

5 Part 4 Edit Distance (3 pt)

The words *computer* and *commuter* are very similar as one can tranform the first to the second by **changing** only one letter. The word *sport* can be changed into *sort* by **deleting** p and in the other direction by **inserting** p.

The edit distance of two string, s1 and s2, is defined as the minimum number of point mutations (change, insert, delete a single letter) required to change s1 into s2. There are many applications of edit distance such as file revision, spelling correction, plagiarism detection, molecular biology, speech recogntion and query-by-humming.

The edit distance of a string to the empty string is the length of the string (delete every letter). Consider the edit distance between two strings string1 and string2 with first characters c1 and c2 respectively and remaining characters s1 and s2. For example if string1 = sport and string2 = hello then c1 = s, c2 = h, s1 = port and s2 = ello. Then the edit distance D between string1 and string2 is the minimum of the following three cases:

- D(s1,s2) + (if c1 = c2 then 0 else 1) (* change or no change *)
- D(string1, s2) + 1 (* insertion one way *)
- D(s1, string2) + 1 (* deletion one way *)

Here is an example of computing edit distance

```
A = acgtac  gtacgt
    || |||  |||| |
B = acatacttgtac t
      ^   ^^    ^
      |   ||    delete
      |   insert*2
      |
      change

D A B = 4
```

These are the only possible cases of editing and by taking the least expensive of the three (the min) we can find the edit distance.

Implement directly this definition of edit distance in ML. (hint: use the explode and implode functions to convert string to lists of characters)

This direct implementation is very slow for long strings. The technique of *dynamic programming* can be used to significanlty speed up the computations.

The idea is to use a two-dimensional matrix m[0..—s1—, 0..—s2—] to hold the edit distance values (—— is used to denote the length of a string).

```
  m[i,j] = d(s1[1..i], s2[1..j])
  m[0,0] = 0
  m[i,0] = i, i= 1..|s1|
  m[0,j] = j  j= 1..|s2|
  m[i,j] = min( m[i-1,j-1] + if (s1[i] = s2[j]) then 0 else 1,
                m[i-1, j] + 1,
                m[i,j-1]  + 1)
```

The matrix can be computed row by row and each row only depends on the previous row and the strings to be compared. Because distance are only computed once propagating values using this matrix is much faster than the direct implementation of the recursion.

In an imperative language an array would be used to hold the matrix. The following illustration shows the process of computing the edit distance using the matrix. Each rows is calculated left to right using information from the previous row. The value of the “current” element depends on the north and northwest elements of the previous row and the previous element on the same row (west). Basically these three dependencies correspond to change (northwest), insertion and deletion and the minimum is chosen.

For example for the value 2* (g row, t column) of the matrix bellow is calculated as follows min(1+1, 2+0 (the row,column string characters are the same, 1+3) = 2.

```
     a c a t  -> B
   0 1 2 3 4 ...
a: 1 0 1 2 3
c: 2 1 0 1 2*
g: 3 2 1 1 2
t: 4 3 2 2 1
. ....
|
V
A
```

Your task is to write the same dynamic programming algorithm using higher order functions and NOT USING AN ARRAY. Also NO DIRECT RECURSION should be used for this part only higher-order functions. The main idea is to create a function *doRow* that given a row (the "previous" row) and a tuple of boundary elements propagates from left to right the values of the new row. Write an appropriate *updElem* function and using a higher-order function to implement *doRow*.

For example with input 0,1 (the boundary conditions) and [1,2,3,4] the first row the result of this step would be [0,1,2,3] (the second row of the matrix). In order to compute this value it is necessary to know the character from the A string corresponding to the row and the whole string B. Hint: you might find the function zipCount that you wrote in part I useful to initialize the process. Combining this function

Another higher-order function using doRow can be used to propagate the row changes down the matrix. In the end the requested value can be found as the southeast corner of the matrix. Basically, one higher-order function with the appropriate arguments propagates the changes for each row and another one propages each row down the column direction.

Compare the performance of the two implementation using the following test cases.

```
val test1a = "acgtacgtacgt";
val test1b = "acatacttgtact";
val test2a = copy 2 test1a;
val test2b = copy 2 test1b;
(* if you see no difference increase the number of copies *)
```

6 Extra Credit

There is no limit to how many extra credit points you get and they can be used to improve your midterm/final or other assignment grades. Extra credit problems are not necessarily hard but many times they are. In addition in many cases I haven't solved them myself so I don't know exactly how hard they are. In addition, they are not as fully specified as the main assignment as part of the challenge is to work with less information. Because of that grading is more subjective.

- (2pt) Implement variable length subtraction and division. Be careful about your design decisions and interpretation of the results you get.
- (1pt) For the edit distance instead of working by rows and columns it is possible to work on diagonals. Make a functional version of a edit distance program that uses diagonals.
- (2pt) Write functions insert, delete, change that do point mutations to strings. Write a function that takes as input a string and a list of functions (IMPORTANT list of functions) and returns the result of applying each function one after the other to the input. Based on the dynamic programming algorithm you have implemented extend your code so in addition to the minimum distance it returns the list of functions that need to be applied for achieving the minimum distance. Verify that these point mutations indeed make the strings equal.

HAVE FUN AND I HOPE YOU ENJOY THIS ASSIGNMENT AS MUCH AS I ENJOYED PREPARING IT