$Id: asg4-ocaml-dc.mm,v 1.1 2011-04-26 13:29:36-07 - - $

1. Overview

In this assignment, you will implement a desk calculator in Ocaml, a language with strong static type checking. Your program will be a strict subset of **dc**(1), although it will not have all of its functions. Begin by reading the man page for **dc**(1) and experimenting with it. Study its input format, output format, error messages, and exit status.

Your program will read the single file (if specified) as does **dc** and then read **stdin** Implement the following **dc** functions: **+ - * / % ^ c d f l p s**. Note that all of these letters are lower case. Do not implement any of the upper case letters except **Y**. Your program should probably respond in some reasonable way to **Y**, for debugging purposes, but does not have to.

2. Implementation Notes

(1) You may not use the **Num** or **Big_int** modules in the Ocaml library. Instead, you will implement your own version of bigint by representing an integer by a product of a sign with a list of integers. The largest value of int in Ocaml is 1073741823 ($2^{30}-1$), which is one bit less than what you might normally expect. This is because one bit is used in each word for tagging.

(2) The ideal representation therefore would be to use eight-digit numbers in a list. However, in order to make sure that the lists are working, you can store only three digits in each element of a list. This wastes storage, but possible makes the representation easier. Since arithmetic operations proceed from the lowest order digit to the highest, represent your numbers with the lowest order digit at the front of the list and the leftmost digit at the end.

(3) Since you will be adding numbers, you will need to be able to store space for the carry bit in an int, hence one digit less than the maxmimum. And for ease in printing, a radix of 10^k is easiest rather than a binary radix. The **dc** utility actually uses character arrays with two decimal digits per byte.

(4) ***Do not*** use any loops in your program. All iteration should be done via recursion, and whenever possible, by using higher-order functions like map.

(5) First implement input and output of numbers. Make sure your output duplicates **dc** for very large numbers. Note that an underscore prefixing a number makes it negative. The minus sign is strictly for subtraction.

(6) Next, implement addition and subtraction. To do this, you will need two functions **add** and **sub** which just compare signs and then call **add'** or **sub'** as appropriate to do the work on their absolute values. When you subtract, make sure that the first argument is always the larger one.

(7) You will need a function **cmp** which returns a comparison value in the same way as does **strcmp** in C. This can move from the low order digits to the high order digits tail recursively and stop at the end of the shorter list, or by maintaining an actual comparison when the two lists turn out to be the same length.

(8) Make sure that you always canonicalize your answers by deleting leading 0 digits. This is only an issue with absolute subtraction, since addition can only lengthen the number. All other operations are implemented in terms of addition and subtraction.

(9) The ancient Egyptians used hieroglyphics for writing, and thus multiplication and division would have been difficult, but they had a system where both operations were done only by repeated addition. These algorithms are about 5000 years old. See the references in the accompanying file, **egyptian-muldiv.html**

(10) To implement multiplication, you add appropriate elements of the right column. To implement division, you add appropriate elements of the left column. The remainder is just whatever is left over after finishing the division, so your division function should return two results as a tuple, namely the quotient and remainder, and the main module then ignores the one not wanted.

(11) Addition and subtraction will thus run at speed $O(n)$. Multiplication and division will run at speed $O(n \log_2 n)$.

(12) Exponentiation will then be trivial, since it is a simple matter to call the other functions. And square root can be done using Newton's method. See the sample function in this directory and then translate it to using **bigint**s. Note that you are dealing only with integers, So carefully check with **dc** for the boundary cases.

3. What to Submit

Makefile, **bigint.ml**, **bigint.mli**, **maindc.ml**, and **scanner.mll** Note that **scanner.ml** is a generated file and should be made by the **Makefile**. Also, **dc.ml** is a debugging tool, not to be submitted. Testing will be done on the **ocamlrun** script **ocamldc**, which should be runnable from the commandline.

Program testing: Test data will be fed to **dc**(1) as well as to your program and the output will be checked with **diff**(1).