

Standard ML Programming Assignment
The BigintB Numeral System

1 Introduction

Given a list of integers $L = [a_{l-1}, \dots, a_1, a_0]$ of length $l > 0$ we may interpret it as representing a polynomial $p(x) = a_{l-1}x^{l-1} + \dots + a_0$ with integer coefficients. We may also find the value of the polynomial for particular (integer) values of the variable x . Further if $x = B > 1$ we may also think of the list L as representing a non-negative integer written in base B if all the elements of the list have values in the range $0..(B-1)$. The value of such a list is then defined as

$$value(L, B) = \sum_{i=0}^{l-1} a_i * B^i \quad (1)$$

Of course, for any value $C > B$, we could also interpret the list as a numeral in base C whose value is given by $value(L, C) = \sum_{i=0}^{l-1} a_i * C^i$. Therefore in order to give these numerals a unique interpretation we need to specify the base as well. But once we are given a base $B > 1$, the numeral would have a unique value and then it does not matter whether the individual elements a_i in the list are positive, negative, zero or even have values greater than B or less than $-B$. In all cases we assume the value given by equation (1). The pair (L, B) is called a **numeral** in base B .

But for convenience it might be a good idea to have a standard format which ensures that the numeral conforms to the usual conventions adopted for representing values in the place-value system of representation¹.

So what about the empty list? We simply adopt the convention that the empty list represents the number zero (regardless of the base) and is in fact, the standard representation of the number zero.

This would allow us to go beyond the word-length of the machine and also the constraints imposed by particular bases and allow us to work *across different bases with arbitrarily large-sized integers*, in the negative domain too. It would then allow us to do computations over large numbers.

2 The Problem Specification

1. Define a new type

```
type bigintb = int list * int
```

which is meant to represent the type of numbers written as lists of integers along with another integer (> 1) which is an indication of the base on which the value of the list is defined (as in equation (1)). $([], B)$ is a representation of the number zero in base B for any integer value of $B > 1$.

2. Define the exception `InvalidBase` if the base is any integer less than 2.
3. Define a function

¹e.g. there should be no leading zeroes, all the coefficients should be non-positive for a negative number and all the coefficients should be non-negative for a positive number etc.

`normalize: bigintb -> bigintb`

such that for any $(L, B) : \text{bigintb}$, with $B > 1$, `normalize` yields an element (N, B) such that

- $\text{value}(L, B) = \text{value}(N, B) = v$ (say)
- $(N, B) = ([\], B)$ if $v = 0$
- If $N = [d_{n-1}, \dots, d_0]$ for some $n > 0$ then
 - $d_{n-1} \neq 0$ (i.e. no leading 0s in the representation)
 - $0 \leq d_j < B$ for all $0 \leq j < n$ whenever $v > 0$ and
 - $0 \geq d_j > -B$ for all $0 \leq j < n$ whenever $v < 0$.

A list N that satisfies the above constraints is said to be in **normal form** and is a standard representation of the number v in base B .

4. Define a function

`convert: bigintb * int -> bigintb`

such that if `convert((L, B), C) = (M, C)` for any valid base C then (M, C) is the normal form in base C of the numeral (L, B) .

5. Define the functions (arithmetic operations)

`add, sub, mult, div, mod : bigintb * bigintb -> bigintb`

representing respectively the operations $+$, $-$, $\times$, `div` and `mod` on integers with the added condition that for `bop` $\in \{\text{add, sub, mult, div, mod}\}$, if `bop((L, B), (M, C)) = (N, D)` then

- $D = \max(B, C)$,
- (N, D) is in normal form

6. Define the boolean valued functions (arithmetic comparisons)

`eq, lt : bigintb * bigintb -> bool`

which check for value equality and “less than” respectively.

Note:

1. You are not allowed to change any of the names given in **font**. You are not even allowed to change upper-case letters to lower-case letters or vice-versa.
2. You may define any new auxiliary functions if you like besides those mentioned.
3. You need to think of the *most efficient way* of implementing the various functions given in the signature so that the function results satisfy their definitions and properties.
4. Since the evaluator is going to look at your source code before evaluating it, you must explain your algorithms in the form of ML comments, so that the evaluator can understand what you have implemented.
5. Follow standard indentation of code as shown in the examples in the class.
6. Do not add any more decorations or functions or user-interfaces in order to impress the evaluator of the program. Nobody is going to be impressed by it.
7. There is a serious penalty for copying. If it is felt that *there is too much similarity in the code between any two persons, then both are going to be penalized equally*. So please set permissions on your directories, so that others cannot copy your programs.