```
$Id: asg5c-dc-stackbignum.mm,v 1.14 2012-03-07 19:58:13-08 - - $
/afs/cats.ucsc.edu/courses/cmps012b-wm/Assignments/asg5c-dc-stackbignum
```

1. Overview

In this assignment you will implement a subset of the **dc**(1) arbitrary precision calculator. For specifications, read the man page for that utility, and experiment by running **dc** itself. You will implement six of its operators: **+**, **-**, *****, **c**, **f**, **p**. Your program will be an executable image called **a5dc**.

2. Code: Module `debugf`

Contains macros useful for debugging. This has been completed.

3. Code: Module `token`

Contains the hand-coded scanner.

(a) `token_ref new_token (FILE*);`
Creates a new token scanner to be used for scanning a file.

(b) `int scan_token (token_ref);`
Reads a new token from the input and returns **EOF**, **NUMBER**, or the character code itself as its return value. Its buffer is updated with the lexical information for a number.

(c) `char *peek_token (token_ref);`
Allows the text of a number to be borrowed for conversion purposes.

4. Code: Module `stack`

Uses the linked list implementation of a stack to hold **bigints**.

(a) `stack_ref new_stack (void);`
Creates a new stack.

(b) `void free_stack (stack_ref);`
Frees the data associated with a stack, provided that it is empty.

(c) `void push_stack (stack_ref, stack_item);`
Pushes a new item onto the stack.

(d) `stack_item pop_stack (stack_ref);`
Pops an item from the stack, provided that the stack is not empty.

(e) `stack_item peek_stack (stack_ref, int index);`
Peeks into the stack at a specified index without changing the stack. Index 0 is the top of the stack. The index must be less than the length of the stack.

(f) `bool is_empty_stack (stack_ref);`
Checks if the stack is empty.

(g) `int length_stack (stack_ref);`
Returns the length of the stack, 0 if empty.

(h) `bool is_stack (stack_ref);`
Checks to see if the pointer really points at a stack.

5. Code: Module `bigint`

Implements arbitrary precision arithmetic with the operations add, sub, and mul.

(a) `bigint_ref new_bigint (size_t length);`
Creates a new **bigint** with a capacity to hold length digits.

(b) `bigint_ref from_string_bigint (char *string);`
Creates a new **bigint** from a given string.

(c) `void free_bigint (bigint_ref);`
Frees a **bigint**.

(d) `void print_bigint (bigint_ref);`
Prints a **bigint** in the same format as **dc**(1).

(e) `bigint_ref add_bigint (bigint_ref left, bigint_ref right);`
Adds to **bigint**s together and returns the sum as a new **bigint**.

(f) `bigint_ref sub_bigint (bigint_ref left, bigint_ref right);`
Subtracts two **bigint**s and returns the differences as a new **bigint**.

(g) `bigint_ref mul_bigint (bigint_ref left, bigint_ref right);`
Multiplies two **bigint**s and returns the product as a new **bigint**.

(h) `bool is_bigint (bigint_ref);`
Checks to see if the pointer really is a **bigint**.

6. Code: Module `main`

The main module scans options, then reads tokens. For each token, it performs the required operation or prints an umimplemented message.

7. Big integer implementation

Following is a more detailed discussion of how to implement the **bigint** module.

(a) Before attempting to implement **bigint**, perform each of the three operations on paper, reminding yourself how to perform the operations without a calculator.

(b) A **bigint** consists of an array of digits, with index 0 being the least significant digit, and the end of the array containing the most significant digit. Each byte contains a single digit between 0 and 9 inclusive. The length field specifies the length of the array, and the digits field specifies the number of significant digits in the array.

(c) Addition, if the signs are the same: call **do_add** to actually perform the addition and return a new **bigint**. Then set the sign to be the sign of one of the arguments.

(d) Addition, if the signs are different: call **do_sub** with the larger number as its left operand and the smaller number as the right operand. Then set the sign to that of the larger number.

(e) Subtraction: if the signs are different, call **do_add**, otherwise call **do_sub**.

(f) **Do_add** is called from either the addition or subtraction function to do the array work. Note that it is marked **static** and is not called outside of the module.

(g) **Do_add** allocates a new **bigint** with space for a number of digits one larger than the largest operand. Then it loops across each array from index [0] to the end, adding and carrying as is done by hand|(::

```
sum = left->buffer[index] + right->buffer[index] + carry;
result->buffer[index] = sum % 10;
sum /= 10;
```

There is a little extra trickiness at the high end of the shorter number.

(h) **Do_sub** allocates a new **bigint** whose size is the same as the left operand, and then performs the subtraction instead of additionC:

```
diff = left->buffer[index] - right->buffer[index] - carry;
if (diff < 0) diff += 10;
result->buffer[index] = sum % 10;
sum /= 10;
```

After computing the result, trim off high-order zeros.

(i) Multiplication proceeds by allocating a new **bigint** whose length is equal to the sum of the lengths of the other two operands. Then in $O(nm)$ speed, where n and m are the lengths of the

numbers, perform an outer loop over one argument and an inner loop over the other argument, adding the new digits to the product, as you would by hand.

(j) Note that `malloc`(3) returns uninitialized storage, but `calloc`(3) sets its allocated storage to 0, so `new_bigint` calls `calloc`, not `malloc`.

8. Testing your program

Your program should write exactly the same output to both `stdout` and `stderr` as does `dc`(1), provided that inputs do not contain those facilities of `dc` that your program is not expected to imitate. For example :

```
dc <test.in >test-dc.out 2>test-dc.err
a5dc <test.in >test-a5dc.out 2>test-a5dc.err
diff test-dc.out test-a5dc.out
diff test-dc.err test-a5dc.err
```

Both of the `diff`(1) commands should produce no output for comparing `stdout`, and only a difference in the name of the ELF for `diff`ing `stderr`.

9. What to submit

Submit `Makefile`, `README`, and all C source and header files necessary for the grader to build your program with the command `make`. If you are doing pair programming, see the additional requirements.