CS360 Lab #2
Due 11:59PM, Tuesday, September 3, 1996

This assignment concentrates on using doubly-linked lists. See the lecture notes in the directory `/blugreen/homes/plank/cs360/notes/Fields` for instruction on how to use the dlist library.

`myreverse` and `mydc`

Implement `myreverse` and `mydc`, which are defined with the following man page.

`Myreverse` should be trivial – read each line into a dlist, and then when the file is done, print the lines backwards. You will have to use `malloc()` and `strcpy()` or `strdup()` here — if you do not, there is something wrong with your code. You should also free the dlist and its associated memory after printing each file.

`mydc` will be more difficult, but will give you excellent practice in handling dlists.

Working Programs

Working versions of `myreverse` and `mydc` are in the directory `/blugreen/homes/plank/cs360/bin`. If you have a question about what your program should do, first see what these programs do.

Strategy

Here is how I went about writing `mydc`. Many students like having road maps for writing program, and I will provide them when I can. By the end of the course, you should be better at designing programs for yourself.

Whenever you are writing largish programs, you should try to write *and test* them incrementally. `Mydc` lends itself well to this. After each step below, make sure that you test the program to make sure it does what you think it does.

- Write the main body of the program that reads standard input into an `IS`, and then processes each word individually. For each word, I have a large `switch` statement on the first letter. If the first letter is a digit or '-', then I convert the word to an integer, and put it at the beginning of my `stack`, which is of course, a dlist. If the first letter is 'f', I print the stack. If the first letter is 'q', I quit. Test this so far.
- Implement the arithmetic opeations (+, -, *, /, %). Note that you'll have to deal with what happens when the first character is '-': is it a number or the subtraction operation? Also,

you have to deal with the following error conditions: division or mod by zero, not enough elements on the stack. Test again.

- Implement 'p', 'd', 'Q', and 'c'. Test after implementing each.
- Now, implement 's', and update 'f' to print out non-empty stacks. What I did here was have an array:

  ```
  Dlist stacks[256];
  ```

  I initialize all of these dlists to be empty. When the user says for example "`sa`", then I push the value onto `stacks['a']`. You have to check to see if the desired stack is an alphabetic character (use `isalpha()`) first. In this way, only 52 of the 256 `stacks` can be used. That may seem wasteful, but it is tolerable. If you wish to implement this differently, feel free to. However, you should use an array or two arrays of dlists for these stacks. Test your work so far.
- Implement 'C' and 'A'.
- Implement 'S', 'l', and 'L'. Test after each one.
- Put in error checking if you haven't done so already.

```
MYREVERSE(1)                 USER COMMANDS                  MYREVERSE(1)

NAME
     myreverse - reverse lines and display

SYNOPSIS
     myreverse [ filename...  ]

DESCRIPTION
     myreverse workds just like mycat, except it prints the lines
     in reverse order.  For multiple files, just print each
     file in reverse order (i.e. don't print the files in
     reverse order).

     Note, there is no -n argument with myreverse.
```

```
MYDC(1)                    USER COMMANDS                   MYDC(1)

NAME
     mydc - desk calculator

SYNOPSIS
     mydc

DESCRIPTION
     mydc is an arithmetic package.  It operates on integers.  The
     overall structure of mydc is a stacking (reverse Polish) calculator.
     Input comes from standard input.

     The following input constructs are recognized:

  Commands
     number  Push an integer onto the  stack.

     +  - /  *  %
             The top two values on the stack  are:  added  (+),
             subtracted   (-),  multiplied  (*),  divided  (/),
             or remaindered (%).   The  two entries are popped off
             the stack and the result is pushed in their place.

     sx      x can be any single alphabetic character (and case should
             be distinguished).  This character is the name of a
             separate stack that can be used to hold values.
             Thus, you have access to 52 stacks other than the
             main stack.  All stacks start as empty.  sx says to push
             the value from the top of the main stack onto stack x.
             The value on the top of the main stack remains there.

     Sx      Treat single character x as a stack and push the value from
             the top of the main stack onto it.  The value is also popped
             off the main stack.

     lx      Push the value from the top of stack x to the top of the
             main stack.  The value remains on top of stack x.

     Lx      Push the value from the top of stack x to the top of the
             main stack.  The value is also popped off stack x.

     d       Duplicate the top value on the stack.

     p       Print the top value on the stack.  The  top  value
             remains unchanged.

     f       Print all values on the main and all other stacks.
             If a stack besides the main stack is empty, do not
             print it.

     q       Exit the program.

     Q       Pop the top value on the stack.

     c       Clear the main stack (i.e. pop off all values)

     Cx      Clear stack x.

     A       Clear all stacks.

EXAMPLE

     INPUT: 5 6 + p                 OUTPUT: 11
     INPUT: 3 4 5 - * p 2 - p       OUTPUT: -3
                                            -5
     INPUT: c 2 3 + d Sa Sb         OUTPUT: main stack: empty
            4 5 * d Sa Sb f                 a: 20 5
            La La - Lb Lb - * p             b: 20 5
                                            225
     INPUT: c p f                  OUTPUT: main stack: empty
                                           main stack: empty
```