```c
/* lab3.c
 * Christopher Brant
 * MP3
 * Subject: ECE222-1,#3
 *
 * Purpose:  The purpose of this machine problem is to learn how string
 *           functions operate, as well as to work with changing ascii
 *           characters to integers and back again, while performing
 *           operations on the integers in the interim there.
 *
 * Assumptions:
 *  #1   The user is prompted to enter a pseudo arithmetic command.  The
 *       input must be verified to be grammatically correct.
 *
 *  #2:  The string and character type library cannot be used under
 *       any circumstances.  You are encouraged to develop your own
 *       functions to perform any similar operations that are needed.
 *
 *  #3   No changes to the code in main.  Your code must be placed in
 *       functions.  Additional functions are encouraged.
 *
 * Bugs:
 *
 * Notes:
 *
 * See the ECE 222 programming guide
 *
 * If your formatting is not consistent you must fix it.  You can easily
 * reformat (and automatically indent) your code using the astyle
 * command.  In a terminal on the command line do
 *
 *     astyle --style=kr lab3.c
 *
 * See "man astyle" for different styles.  Replace "kr" with one of
 * ansi, java, gnu, linux, or google to see different options.  Or, set up
 * your own style.
 */

// do not include any additional libraries
#include <stdio.h>

// do not change these constants
#define MAXLINE 80
#define MAXOPER 13

// named constants and strings
enum operations { NOOP, ADD, MUL, DIV, POW};
const char *operation_str[] = {"Invalid", "+", "*", "/", "^"};

// function prototypes
int process_input(const char *input, char *op_left, char *op_right);
void calc_output(const char *op_l, int op, const char *op_r, char *result);
int op_position(const char *input);
int op_type(const char *input, int op_pos);
int in_length(const char *input);
char *s1(const char *input, int op_pos, char *op_left);
char *s2(const char *input, int op_pos, int string_length, char *op_right);
void addchar(const char *l_op, const char *r_op, char *result, int ll, int rl);
void mullchar(const char *l_op, const char *r_op, char *result, int ll, int rl);
void divchar(const char *l_op, const char *r_op, char *result, int ll, int rl);
void powchar(const char *l_op, const char *r_op, char *result, int ll, int rl);



// do not change any code in main.  We are grading based on the format
// of the output as given in the printfs in main.
int main()
{
    char input_line[MAXLINE];
    char left_operand[MAXOPER];
    char right_operand[MAXOPER];
    char answer[MAXOPER];
    int  operator;

    printf("\nMP3: Arithmetic on GF(47) with + * / ^ using letters\n");
    printf("Commands:\n\tabc+bbc\n\tturtle/frog\n\ttiger^one");
    printf("\tOperands are no more than 12 letters and no spaces\n");
    printf("\tCtrl-d to quit\n\n");
    printf("> ");

    // each call to fgets collects one line of input and stores in input_line
    // BEWARE: fgets includes the end-of-line character '\n' in input_line
    while (fgets(input_line, sizeof input_line, stdin) != NULL) {

        // clear for next round
        left_operand[0] = right_operand[0] = answer[0] = '\0';

        // check for valid grammar
        operator = process_input(input_line, left_operand, right_operand);

        if (operator == ADD || operator == MUL
                || operator == DIV || operator == POW) {

            // print parsed input
            printf("'%s'", left_operand);
            printf(" %s ", operation_str[operator]);
            printf("'%s' => ", right_operand);

            // perform pseudo arithmetic
            calc_output(left_operand, operator, right_operand, answer);

            // print result
            printf("'%s'\n\n", answer);
        } else {
            printf("# %s", input_line);
        }
        printf("> ");
    }
    printf("\nGoodbye\n");
    return 0;
}

/* Parse input of the form SOS where S is a string and O is a character.
 *
 * A string S must consist of up to 12 valid symbols a-z and A-U.
 * The operand O must be one character from: + * / ^
 * Any other characters found in the input, including spaces, are
 * grammatically incorrect and invalidate the input.
 *
 * There must be no spaces anywhere in the input, including between
 * either SO, OS, or leading or trailing spaces.
 *
 * Input: The input string is collected using fgets.  Recall the end-of-line
 *        character is included in the input string and marks the end of
 *        the input.  This string must not be changed.
 *
 * Output: There are three outputs from this function.
 *
 *   The return value is one of NOOP, ADD, MUL, DIV, POW which are
 *      named constants.  If the input is invalid for any reason
 *      then the output must be NOOP.  Otherwise the return value
 *      corresponds to operand O.
 *
 *   If the input is grammatically correct, then two strings are also
```

```
 *   returned, one for each of the left and right operands.  If the input
 *   in invalid the two output strings are undefined.
 */
int process_input(const char *input, char *op_left, char *op_right)
{
    int op = 0;
    int op_pos = op_position(input);
    int strlen = in_length(input);
    // Make sure that if op_pos returns 0 that we get invalid input

    if (op_pos != 0)
    {
        op = op_type(input, op_pos);                    // Determines what operatio
n
        op_left = s1(input, op_pos, op_left);           // Parses op_left
        op_right = s2(input, op_pos, strlen, op_right); // Parses op_right
    }

    // If the strings are invalid, we set the operator option to invalid
    if (op_left[0] == '\0' || op_right[0] == '\0' || strlen >= 78)
        op = 0;

    // Returns the correct operator input
    return op;
}

/* Pseudo mathematical opertions on the two operands work as follows.
 *
 * Each character is converted to an integer in the range 1...46, where a is 0,
 * b is 1, c is 2, ..., z is 25.  The operation is then performed using
 * math on a finite field with no carries.
 *
 * If the two input strings are not the same length, then each output character
 * beyond the length of the shorter string should be a copy of the character
 * from the longer string but with the opposite case.
 *
 * Input: The two operand strings and the operator are assumed to be valid (and
 *        are verified as valid in the parse_input function).
 *
 * Output: The final string generated by the above rules is stored in the
 *         output string named result.  The input strings must not be
 *         changed.
 */
void calc_output(const char *l_op, int op, const char *r_op, char *result)
{
    int i;

    // Find the lengths of each operand locally
    int length_left = in_length(l_op);
    int length_right = in_length(r_op);

    // The four following conditionals execute the operations
    if (op == 1)
        addchar(l_op, r_op, result, length_left, length_right);

    else if (op == 2)
        mullchar(l_op, r_op, result, length_left, length_right);

    else if (op == 3)
        divchar(l_op, r_op, result, length_left, length_right);

    else if (op == 4)
        powchar(l_op, r_op, result, length_left, length_right);


    /* The following conditionals flip the last characters
       in the event of different string lengths */
    if (length_left < length_right)
    {
        for(i = length_left; i < length_right; i++)
        {
            if (r_op[i] <= 'U' && r_op[i] >= 'A')
                result[i] = r_op[i] + ('a' - 'A');

            else
                result[i] = r_op[i] - ('a' - 'A');

            result[i+1] = '\0';
        }
    }

    else if (length_right < length_left)
    {
        for(i = length_right; i < length_left; i++)
        {
            if(l_op[i] <= 'U' && l_op[i] >= 'A')
                result[i] = l_op[i] + ('a' - 'A');

            else
                result[i] = l_op[i] - ('a' - 'A');

            result[i+1] = '\0';
        }
    }
}

// This function determines the placement and validity of the operator
int op_position(const char *input)
{    int i, op_pos = 0, op_num = 0;
    for (i = 0; (input[i] != '\n' && input[i] != '\0' && i < 78); i++)
    {
        if (input[i] == '+' || input[i] == '*'  || input[i] == '/' || input[i] == '
^')
        {
            op_num++;
            op_pos = i;
        }

        // If more than one operator or more than 12 characters in the first string
...
        if (op_num != 1 || op_pos > 12)
            op_pos = 0;
    }

    return op_pos;
}

// This function determines the type of the character the operator is
int op_type(const char *input, int op_pos)
{
    int type;

    if (input[op_pos] == '+')
        type = 1;

    if (input[op_pos] == '*')
        type = 2;

    if (input[op_pos] == '/')
        type = 3;

    if (input[op_pos] == '^')
        type = 4;
```

```
    return type;
}

// This function determines the overall length of the input string
int in_length(const char *input)
{
    int i, length = 0;
    for (i = 0; (input[i] != '\0' && input[i] != '\n'); i++)
        length++;

    return length;
}

// This function parses and validates the left-hand operand from the input string
char *s1(const char *input, int op_pos, char *op_left)
{
    int i, j;

    for (i = 0, j = 0; i < op_pos; i++)
    {
        if ((input[i] <= 'z' && input[i] >= 'a') || (input[i] <= 'U' && input[i] >=
 'A'))
        {
            op_left[j] = input[i];
            j++;
        }

        else
        {
            j = 0;
            break;
        }
    }

    op_left[j] = '\0';

    return op_left;
}

// This function parses and validates the right-hand operand from the input string
char *s2(const char *input, int op_pos, int string_length, char *op_right)
{
    int i, j = 0;

    if (string_length - op_pos <= 13 && string_length - op_pos > 0)
    {
        for (i = op_pos + 1, j = 0; i < string_length && input[i] != '\n' && input[
i] != '\0'; i++, j++)
        {
            if ((input[i] <= 'z' && input[i] >= 'a') || (input[i] <= 'U' && input[i
] >= 'A'))
                op_right[j] = input[i];

            else
            {
                j = 0;
                break;
            }
        }
    }

    op_right[j] = '\0';

    return op_right;
}

// This function adds the operands in a finite field
void addchar(const char *l_op, const char *r_op, char *result, int ll, int rl)
{
    int length, i, op1, op2, sum;

    if (ll > rl)
        length = rl;

    else if (rl > ll)
        length = ll;

    else
        length = ll;

    /* Changes the character to an integer from 0-47 and
       after finding the sum, it changes it back to the
       corresponding character */
    for (i = 0; i < length; i++)
    {
        op1 = l_op[i] - 'a';

        if (l_op[i] >= 'A' && l_op[i] <= 'U')
            op1 = l_op[i] - 'A' + 26;

        op2 = r_op[i] - 'a';

        if (r_op[i] >= 'A' && r_op[i] <= 'U')
            op2 = r_op[i] - 'A' + 26;

        sum = (op1 + op2) % 47;

        if (sum < 26)
            result[i] = sum + 'a';

        else if (sum > 25)
            result[i] = sum - 26 + 'A';
    }

    result[i] = '\0';
}

// This function multiplies the operands in a finite field
void mullchar(const char *l_op, const char *r_op, char *result, int ll, int rl)
{
    int length, i, op1, op2, prod;

    if (ll > rl)
        length = rl;

    else if (rl > ll)
        length = ll;

    else
        length = ll;

    /* Changes the character to an integer from 0-47 and
       after finding the product, it changes it back to the
       corresponding character */
    for (i = 0; i < length; i++)
    {
                op1 = l_op[i] - 'a';

        if (l_op[i] >= 'A' && l_op[i] <= 'U')
            op1 = l_op[i] - 'A' + 26;

        op2 = r_op[i] - 'a';

        if (r_op[i] >= 'A' && r_op[i] <= 'U')
```

```
            op2 = r_op[i] - 'A' + 26;

        prod = (op1 * op2) % 47;

        if (prod < 26)
            result[i] = prod + 'a';

        else if (prod > 25)
            result[i] = prod - 26 + 'A';
    }

    result[i] = '\0';
}

// This function divides the operands in a finite field
void divchar(const char *l_op, const char *r_op, char *result, int ll, int rl)
{
    int length, i, j, op1, op2, quot;

    if (ll > rl)
        length = rl;

    else if (rl > ll)
        length = ll;

    else
        length = ll;

    /* Changes the character to an integer from 0-47 and
       after finding the quotient, it changes it back to the
       corresponding character */
    for (i = 0; i < length; i++)
    {
                op1 = l_op[i] - 'a';

        if (l_op[i] >= 'A' && l_op[i] <= 'U')
            op1 = l_op[i] - 'A' + 26;

        op2 = r_op[i] - 'a';

        if (r_op[i] >= 'A' && r_op[i] <= 'U')
            op2 = r_op[i] - 'A' + 26;

        for (j = 0; j < 47; j++)
        {
            if (op1 == (op2 * j) % 47)
                quot = j;
        }

        if(op2 == 0)
            quot = 0;

        if (quot < 26)
            result[i] = quot + 'a';

        else if (quot > 25)
            result[i] = quot - 26 + 'A';
    }

    result[i] = '\0';
}

// This function raises one operand to the power of another in a finite field
void powchar(const char *l_op, const char *r_op, char *result, int ll, int rl)
{
    int length, i, j, op1, op2, res;

    if (ll > rl)
        length = rl;

    else if (rl > ll)
        length = ll;

    else
        length = ll;

    /* Changes the character to an integer from 0-47 and
       after finding the result, it changes it back to the
       corresponding character */
    for (i = 0; i < length; i++)
    {
                op1 = l_op[i] - 'a';

        if (l_op[i] >= 'A' && l_op[i] <= 'U')
            op1 = l_op[i] - 'A' + 26;

        op2 = r_op[i] - 'a';

        if (r_op[i] >= 'A' && r_op[i] <= 'U')
            op2 = r_op[i] - 'A' + 26;

        res = op1;

        for (j = 1; j < op2; j++)
        {
            res *= op1;
            res %= 47;
        }

        if (op2 == 0)
            res = 1;

        if (res < 26)
            result[i] = res + 'a';

        else if (res > 25)
            result[i] = res - 26 + 'A';
    }

    result[i] = '\0';
}
```