Programming Assignment 3 - part III
tcalc
DUE: Friday April 1 by 10am

In this portion of the assignment, you will add a spreadsheet-like feature to the table-processing code base you have been working on.

Summary: The tcalc program reads in a table (as in part II) from stdin and an expression from a specified file; this expression is evaluated for each row and the results form a new column which is added to the table; the resulting table (with one additional column) is then printed to stdout.

An Example First

The left box below is an example of a table file in the format you are already familiar with. Assume it has the name scores.txt. In the right box is a "formula file"; assume it has the name avg.

This formula file specifies that a new column is to be created with column label "Exam avg" and populated with the average of columns 2 and 3 in the input table.

```
# an input file
# filename: scores.txt
Name:  exam1 : exam2
Bob:   60.0   70.0
Alice: 80.0   90.0
```

```
# tcalc formula file
# filename: avg

Exam avg: / + $2 $3 2
```

The important line in the formula file is "Exam avg: / + $2 $3 2":

- The label to be used for the calculated column is first followed by a colon
- After the colon is a prefix expression for the calculated

column.
- The expression is evaluated for each row in the table.
- $2 and $3 correspond to the values in columns 2 and 3 respectively.
- The expression is in prefix notation. This particular expression corresponds to (($2 + $3) / 2) in infix notation.
- Notice that there is white space between each component ("token") of the expression. This is a requirement for any syntactically correct expression and should make your life easier as a programmer. If, for example, the expression were given as "/+$2$3 2" the tcalc program would report a syntax error.

Using these files, the tcalc program would be run as below.

```
$ ./tcalc avg < scores.txt > scores_avg.txt
```

There is exactly one command line argument giving the name of the formula file. This argument is required. Otherwise, the program reads the table from standard input just like the tsort program.

After the run, the output (redirected into file scores_avg.txt) is as below.

```
Name:  exam1 : exam2:   Exam avg
Bob:   60.0     70.0    65.0
Alice: 80.0     90.0    85.0
```

Note that the output is a new table which is readable by programs like tsort and tcalc itself.

Also note that it does not bother carrying along comment lines (would be nice, but certainly not a priority for us).

Syntax of Expressions and Operators

Expressions are, naturally, given as strings. You will process a given string one “token” at a time.

Tokenizing:

As mentioned in the previous section, we simplify matters by requiring that tokens are *white-space separated*. This implies that a utility like strtok can be used in a straightforward way to read an input string one token at a time.

Legal Expressions:

Expressions are given in ***prefix notation***. Legal arithmetic formulas are formed according to the following recursive rules:

EXPRESSION TYPE	**SYNTAX RULES**	**DISCUSSION**
BASE CASE 1: Numerical constants.	Any token that can be parsed as a float is an expression. For example, sscanf with the %f specifier can successfully parse the token.	
BASE CASE 2: Column Specifiers	A token of the form **$<N>** is an expression where <N> is a positive integer	Examples: $2 $5 Discussion: expressions are always evaluated with respect to a particular row. A column specifier evaluates to the numerical value of the specified

		column. For example, $2 evaluates to the value in column 2 of the row being processed.
ADDITION	If <E1> and <E1> are expressions then `+ <E1> <E2>` is an expression	E1 and E2 can be any expressions. Not just floats.
SUBTRACTION	If <E1> and <E1> are expressions then `- <E1> <E2>` is an expression	E1 and E2 can be any expressions. Not just floats.
MULTIPLICATION	If <E1> and <E1> are expressions then `* <E1> <E2>` is an expression	E1 and E2 can be any expressions. Not just floats.
DIVISION	If <E1> and <E1> are expressions then `/ <E1> <E2>` is an expression	E1 and E2 can be any expressions. Not just floats. (See error handling section regarding division by zero)

Logical and Comparator Operators

Every expression in our “little language” evaluates to a floating point number.

This fact will also be true for logical operators with the following C-like interpretation:

- Zero is interpreted as false.
- Anything that is non-zero is interpreted as true.

This unifies things for us a bit.

OPERATOR	**SYNTAX**	**Evaluation / semantics**
LESS THAN <	If <E1> and <E1> are expressions then < <E1> <E2> is an expression	Evaluates to 1 if <E1> is less than <E2>; zero otherwise.
GREATER THAN >	If <E1> and <E1> are expressions then > <E1> <E2> is an expression	Evaluates to 1 if <E1> is greater than <E2>; zero otherwise.
EQUAL =	If <E1> and <E1> are expressions then = <E1> <E2> is an expression	Evaluates to 1.0 if <E1> is equal to than <E2>; 0.0 otherwise.
CONDITIONAL TERNARY OPERATOR ?	If <E1>, <E1> and <E3> are expressions then ? <E1> <E2> <E3> is an expression	Evaluates to <E2> if <E1> is (evaluates to) non-zero; Evaluates to <E3> otherwise
AND &	If <E1> and <E1> are expressions then & <E1> <E2> is an expression	Evaluates to 1.0 if <E1> **and** <E2> both are (evaluate to) non-zero. Evaluates to 0.0 otherwise
OR |	If <E1> and <E1> are expressions then | <E1> <E2> is an expression	Evaluates to 1.0 if <E1> **or** <E2> is (evaluates to) non-zero. Evaluates to 0.0

		otherwise (both evaluate to 0.0)
NOT !	If <E> is an expression then ! <E> is an expression	Evaluates to 1.0 if <E> is (evaluates) to 0.0; Evaluates to 0.0 otherwise (when <E> is non-zero).

Examples

GOAL	**EXPRESSION (actually line in file would have the column label and a colon before the expression).**
Sum of columns 2, 3 and 4	`+ + $2 $3 $4`
Maximum of columns 2 and 3	`? > $2 $3 $2 $3`
pass/fail: 1.0 if column 2 greater than 50. 0.0 otherwise	~~? > $2 50.0~~ `> $2 50.0`
pass/fail 2: 1.0 if column 2 greater than or equal to 50. 0.0 otherwise	`! < $2 50.0`
1.0 if column 2 is less than 50 or column 3 is greater than 100. 0.0 otherwise	`| < $2 50 > $3 100`
Absolute value of column 2	`? < $2 0.0 * -1.0 $2 $2`

Error Conditions

Can't Even Start Errors:

- formula file not given as command line argument
- Filename given, but cannot be opened for reading
- File given and can be opened, but there is no line in the file corresponding to a new column to be computed.

Syntax Errors: Any formula file that does not conform to the syntax rules in the above tables, should be rejected and no output produced.

Examples:

INPUT FILE	DESCRIPTION OF SYNTAX ERROR
+ \$2 \$3	The expression is ok, but it is missing the column label. These would be ok (2nd one just has an empty column label): Sum: + \$2 \$3 : + \$2 \$3
Sum: +\$2\$3	Tokens of expression not white-space separated.
Label: + + \$2 \$3	First + operator is followed by only one expression, not two. (try converting it to infix)

Semantic Errors: An expression may syntactically well-formed but still fail when actually evaluated in the context of a row -- sort of a "runtime error." The two such errors that a fully function tcalc program should handle are described below.

ERROR	**DESCRIPTION**	**PROGRAM BEHAVIOR**
Column number out of bounds.	Only columns 2 through the number of columns in the given table make sense in an expression. Recall that column 1 is always the row name and so does not have a numerical value.	Program terminates with a suitable error message. No table output is produced.
ADVICE: Do not make handling division by zero a first priority!		
Division by Zero	We all know what division by zero is...	It's not obvious what the "right" behavior should be. Should the entire tcalc run fail because one row happens to result in a division by zero? Maybe yes, maybe no. We will say "NO". Our rule: when an expression results in a division by zero for some row, it will evaluate to **DBL_MAX** which is the largest double. It is a constant defined in float.h. It is our version of "infinity" So, you will need #include <float.h>

Discussion / Tips / Misc. Details

- Remember that the formula file is given as a single command-line argument. You hopefully know how to open and read from such files right? fopen and fclose are the key library functions.
- You have ***not*** been given a specific set of new functions to implement for the **tbl** ADT. It is up to you to decide how to extend the **tbl** ADT to enable the required functionality.
- Nevertheless, give some careful thought to the **division of labor** between the tbl ADT and your application program.
- As a thought experiment, imagine that next week or next month, you want to write another program/extension which *filters* the rows of a given table based on a given expression. Would the decisions you make today result in such a program being easy or hard?
- Recursion, recursion, recursion: use recursion to parse/evaluate expressions.
- When parsing, a recursive call will "consume" some tokens; when that recursive call returns, somehow, you need to account for the tokens consumed (i.e., so your next operation starts in the correct position in the "token stream.") There is more than one way to accomplish this (strtok will do it automatically if used correctly, but there are other ways). Just think carefully about this issue!
- Splitting into tokens has been made as simple as I think is possible. Take full advantage of this simplicity.
- All of the operators are single characters. Take full advantage of this to simplify your code!
- Remember to make sure that your printed table is also readable as a table (e.g. by tsort or by tcalc itself to append yet another column).

Deliverables and Submission

Your submission will be an extension of your submission for Part-2 of this assignment.

If your part-2 submission was not fully functional, you can and should take this opportunity to get it fully working. It will be evaluated as mentioned in class. (Further, part-3 will be evaluated under the assumption that part-2 works, so you need to get part-2 working for part-3 to be evaluated).

In addition to the source files specified for part-2, you will include all source files you created for part-3 (probably tcalc.c) and make sure that the makefile supports the target tcalc.