Program 1

CMPS 390

Jacob Tucker

2/16/23

Exercise 6:

First, I will go over the project structure. Since I chose to implement this program in C, I made the program more modular and easier to read by splitting it into 5 different C source files. The main source file is *program1.c*, which imports all the other files by including their header files. First, a *stack.c* file was created to be used by all the other source files. Then, a separate C file was made for each step of the validate, translate, evaluate process. Also, the input is read from the *input.txt* file.

```
dys@DESKTOP-658TEVJ:~/src/390/stack$ ls
evaluate.c  input.txt   stack.c  translate.c  validate.c
evaluate.h  program1.c  stack.h  translate.h  validate.h
```

input.txt

```
1  (5+3)
2  (5+3)*8+(2)
3  (5+3)*((2+4)*3))
4  (((4)))
5  (1/2)*(2+3)/2
6  (2+3)*(8*9)/2)
7  ((2+3)*((8*9)/2)
8  (2+3)*((8*9)/2)
```

The main source file, *program1.c*, has the following implementation:

program1.c

```c
#include "stack.h"
#include "validate.h"
#include "translate.h"
#include "evaluate.h"
#include <stdio.h>
#include <string.h>

void checkExpr(char expr[]) {
  char* postfixExpr;
  int answer;

  if (isValid(expr, strlen(expr))) {
    printf("Valid expression: %s\n", expr);
    postfixExpr = translate(expr, strlen(expr));
    printf("Translation: %s\n", postfixExpr);
    answer = evaluate(postfixExpr, strlen(postfixExpr));
    printf("Answer: %i\n\n", answer);
  }
  else {
    printf("Invalid expression: %s\n\n", expr);
  }
}

int main() {
  char fileName[] = "input.txt";
  char line[MAX_SIZE];
  FILE* file;
  file = fopen(fileName, "r");

  while (fgets(line, sizeof(line), file)) {
    if (line[strlen(line) - 1] == '\n') { line[strlen(line) - 1] = '\0'; }
    checkExpr(line);
  }

  fclose(file);
}

```

It begins by including the header files from all the other C source files I made, and it also imports the *stdio.h* library for printing to the terminal and the *string.h* library for getting the length of strings.

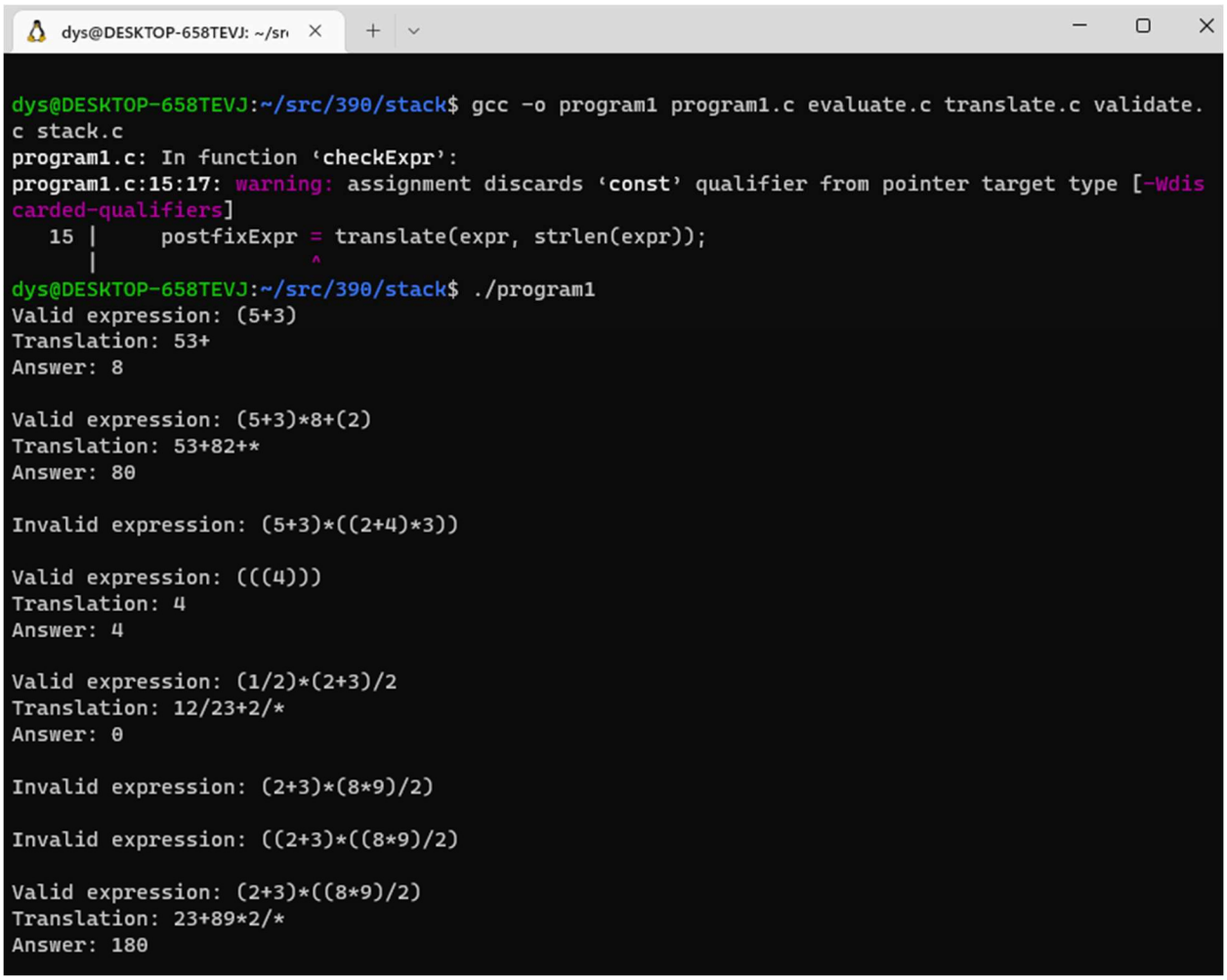
The *main()* function first initializes a string *filename* to refer to file the input will be parsed from. Then a string of size *MAX_SIZE* is declared. *MAX_SIZE* is a constant defined in the *stack.h* header file used to determine the maximum size of the array in the stack struct. Then a pointer to a file simply named *file* is declared. It is then assigned the address returned by the *fopen()* function, which opens the *input.txt* file in read mode.

The *main()* function then loops until the *fgets()* function is done reading from the *input.txt* file. The *fgets()* function reads a line from the provided file, stores the contents of that line in the string *line*, and

repeats this until it encounters the end-of-file marker or an error. When this happens, it then returns a null value, terminating the loop. Inside the loop, the first line checks if the last character of the retrieved string is a line break. If so, then it replaces it with a null value instead, which allows other functions reading the string to see when the string terminates. The last thing done in the loop is passing the string to the *checkExpr()* function.

The *checkExpr()* function accepts a string as an argument and has no return value since it outputs its results to the terminal with *printf()*. It first declares a character pointer *postfixExpr* that will be used to point to the beginning of a string representing the expression in the postfix format. It also declares an int *answer* to store the answer obtained from evaluating the postfix expression. The function then checks if the string it was passed is a valid expression by calling the *isValid()* function that is imported via the *validate.h* header file. If the expression is invalid, it simply prints the expression and says that it is invalid. If it is valid, it then prints the expression and that it is valid, and it also translates it into a postfix expression by calling the *translate()* function, which is imported from the *translate.h* header file, and assigning its return value to *postfixExpr*. The postfix expression is then printed to the console as well. Then, *postfixExpr* is passed into the *evaluate()* function, which is imported from the *evaluate.h* header file, and the output is stored in the *answer* variable. The answer is then printed to the output stream and the function is finished.

Once the loop has passed every line form the input file to the function to be checked and evaluated, it terminates and the *main()* function closes the file. That is all for the main source file. Here is its output:



Note that there is a warning given by the GCC compiler. I was unable to find a way to take care of this warning, and since it isn't an error and the program seems to work fine, I gave up and left it. Limitations of this programs ability to evaluate expressions are that it used integer values and therefore will round answers. It will also not necessarily follow the correct order of precedence for mathematical operations without proper parenthesis placement.

Now I will discuss the other files. Let's start with *stack.c* and *stack.h*.

stack.c

```c
#include <stdio.h>

#define MAX_SIZE 255

union StackElement {
  int intValue;
  char charValue;
};

struct Stack {
  int top;
  union StackElement contents[MAX_SIZE];
};

struct Stack newStack() {
  struct Stack NewStack;
  NewStack.top = -1;
  return NewStack;
}

int isEmpty(struct Stack *stack) { return stack->top == -1; }

void push(struct Stack *stack, union StackElement elem) {
  if (stack->top == MAX_SIZE) { printf("Error: Stack overflow\n"); }

  stack->contents[++stack->top] = elem;
}

union StackElement pop(struct Stack *stack) {
  union StackElement elem;

  if (isEmpty(stack)) { printf("Error: Stack underflow\n"); }

  elem = stack->contents[stack->top--];
  return elem;
}

void showStack(struct Stack *stack) {
  for (int j = 0; j != (stack->top + 1); ++j) {
    if (stack->contents[j].charValue) {
      printf("%c ", stack->contents[j].charValue);
    }
    else {
      printf("%i ", stack->contents[j].intValue);
    }
  }
  printf("\n");
}
```

```c

int runTest(int received, int expected, int testCount, int testsPassed) {
  if (expected == received) { testsPassed++; }
  else { printf("Test %d Failed: Expected %d but got %d\n", testCount, expected, received); }
  return testsPassed;
}

void testStack() {
  int testsPassed = 0;
  int testCount = 1;
  struct Stack testStack = newStack();
  struct Stack *testStackPtr = &testStack;

  testsPassed = runTest(isEmpty(testStackPtr), 1, testCount, testsPassed);
  testCount++;

  union StackElement a;
  a.intValue = 5;
  push(testStackPtr, a);
  testsPassed = runTest(isEmpty(testStackPtr), 0, testCount, testsPassed);
  testCount++;

  union StackElement b;
  b.charValue = 'a';
  push(testStackPtr, b);
  testsPassed = runTest(pop(testStackPtr).charValue, 'a', testCount, testsPassed);
  testCount++;

  testsPassed = runTest(testStack.contents[testStack.top].intValue, 5, testCount, testsPassed);

  printf("%d/%d tests passed\n", testsPassed, testCount);
}

// int main() {
//   testStack();
// }

```

Since this exercise is to validate, translate, and evaluate expressions from a text file using stack methods, it requires a stack that can hold char values and a stack that can hold int values. I wanted to create a generic stack that would work with either data type rather than creating a separate stack implementation for each that uses mostly the same code. It seems that the traditional way to do this in C is to use void* pointers to point to values of yet undetermined types and later typecasting them to the correct type. However, since we only need to worry about being generic for 2 types in the program, I chose to utilize the *union* type instead. This allows me to create an stack of union elements, each of which can have a char or int value.

The *stack.c* file begins with including the *stdio.h* library in order to print error messages and test information. I then define a constant *MAX_SIZE* which is used to determine the maximum size of the array in the stack struct. The union *StackElement* is the defined with its members being an int to represent its int value and a char to represent its char value. The stack struct is then defined with 2 members, an int to represent the index of the top of the stack and an array of type StackElement.

To initialize a new stack I added function *newStack()* that returns a stack with its *top* member initialized to -1 to indicate that the stack is empty. The next function, *isEmpty()*, takes a pointer to a stack and returns 1 if the stack is empty and 0 if it isn't by checking if the *top* member of the stack is equal to -1.

Next, a *push()* function is defined. It accepts a pointer to a stack and a *StackElement* to push onto the stack. It first checks if the stack is at capacity by seeing if the *top* attribute is at the *MAX_SIZE*. If so, then it will print a message indicating that a stack overflow has happened. If not, then it increments the *top* attribute and then adds the *StackElement* to the *contents* array in the stack at the index equal to *top*. Since that step is done in a single line, the prefix increment operator is used to ensure that *top* increments before it is used as the index for *contents* array.

The *pop()* function is then defined, which stacks a pointer to a stack and then returns the element popped from it. It also performs a check to start with, ensuring that the stack isn't empty before attempting to pop an element. If this happens, it prints a message saying that a stack underflow has happened. Using the postfix decrement operator, *top* is used an index to grab the element about to be popped from the *contents* array and then decremented.

The *showStack()* function also takes a pointer to a stack as an argument and loops over each element printing its value. Since each stack element is a union type, they each have 2 potential values. Since each individual stack should have the same value types for all of its elements, i.e. there shouldn't be a stack in this program that has both some char elements and some int elements, my approach for printing them is to check if there is a char value for the elements. If there isn't then the elements were meant to be int values, and they are printed accordingly. If there are, then they were meant to be char values and will be printed accordingly. This allows int values of 0 to be printed appropriately as well.

The next two functions, *runTest()* and *testStack()*, work together to test if the stack is behaving as expected. The *runTest()* function was added to reduce the repeated code for each test. The *testStack()* function stars by initializing counters to track the number of tests performed and passed. A new stack *testStack* and a pointer to it are then initialized. The first tests checks if the stack is empty after initialization. It is run by passing a received value, an expected value, and the test counters to the *runTest()* function. The *runTest()* function compares the received value, which is the result of running the *isEmpty()* function with the *testStackPtr* pointer as an argument in this case, to the the expected value, which is a 1 to indicate that the stack is empty. If the expected value is received, then the test is passed and *testsPassed* is incremented by 1 and returned. The *testCount* counter is also passed to this function so it will know which test failed if one does fail. If the test fails, it will print which test failed, what it expected to get as a result of its tests, and what it received during the test instead.

The *testStack()* function then pushes a value onto the stack and then checks if it is empty again. The next test involves pushing a second element onto the stack and then popping it off and seeing if the expected element was popped. The last test pops the last element off the stack and checks if it is the expected value. This checks both integer and character values with the two pops.

The *main()* function for this file is commented out because this is used as a library file for *program1.c* and other source files to use. This is also the case for the *main()* functions in all other library source files in this project. If uncommented, the *testStack()* function outputs the following:

```
dys@DESKTOP-658TEVJ:~/src/390/stack$ gcc -o stack stack.c
dys@DESKTOP-658TEVJ:~/src/390/stack$ ./stack
4/4 tests passed
```

If I purposefully cause two of the tests to fail for demonstration purposes, the output looks like this:

```
dys@DESKTOP-658TEVJ:~/src/390/stack$ gcc -o stack stack.c
dys@DESKTOP-658TEVJ:~/src/390/stack$ ./stack
Test 1 Failed: Expected 0 but got 1
Test 3 Failed: Expected 97 but got 98
2/4 tests passed
dys@DESKTOP-658TEVJ:~/src/390/stack$
```

stack.h

stack.h > MAX_SIZE

```c
#define MAX_SIZE 255

union StackElement {
  int intValue;
  char charValue;
};

struct Stack {
  int top;
  union StackElement contents[MAX_SIZE];
};

struct Stack newStack();

void push(struct Stack *stack, union StackElement elem);

union StackElement pop(struct Stack *stack);

int isEmpty(struct Stack *stack);

void showStack(struct Stack *stack);

void testStack();

```

As for the *stack.h* file, it simply contains the *MAX_SIZE* constant, *StackElement* union, *Stack* struct, and some of the function declarations from the *stack.c* file so that they can be imported by other source files. The other header files are simpler since there are less functions to export to other files, as seen here:

translate.h

```
translate.h > translate(char *, int)
1    const char* translate(char* expr, int exprLength);
```

validate.h

```
validate.h > isValid(char *, int)
1    int isValid(char* expr, int exprLength);
```

evaluate.h

```
evaluate.h > evaluate(char *, int)
1    int evaluate(char* expr, int exprLength);
```

Next, I will go over the *validate.c* file.

validate.c

```c
#include "stack.h"

int isValid(char* expr, int exprLength) {
  struct Stack pStack = newStack();
  struct Stack* pStackPtr = &pStack;
  union StackElement elem;

  for(int i = 0; i != exprLength; ++i) {
    if (expr[i] == '(') {
      elem.charValue = expr[i];
      push(pStackPtr, elem);
      }
    else if (expr[i] == ')') {
      if (isEmpty(pStackPtr)) { return 0; }
      else { pop(pStackPtr); }
    }
  }

  return isEmpty(pStackPtr);
}

// int main() {
//   char expr[] = "(2+3)*(8*9)/2)";
//   printf("%i\n", isValid(expr, sizeof(expr) - 1));
// }
```

The *validate.c* file imports from *stack.h* and has a single function *isValid()* which accepts a pointer to the beginning of a string and the length of the string. It initializes a new stack and a *StackElement*. It then iterates over the string passed to it and checks for open parenthesis and close parenthesis. If there is an open parenthesis, the *StackElement*s char value is set to be the open parenthesis and then it is pushed onto the stack. If it is a close parenthesis, then a check is done to see if the stack is empty. If it is, then the expression is invalid because there were too many close parentheses, so the function returns 0 to indicate that it is not valid. If not, then the stack is popped to remove an open parenthesis from it. Once the iteration is complete, the expression will be considered valid if the stack empty because that means all of the open parentheses were popped off. So the truth value of whether the stack is empty or not is returned.

Next, I will cover *translate.c*.

translate.c

```c
#include "stack.h"

char* translate(char* expr, int exprLength) {
  struct Stack postStack = newStack();
  struct Stack opStack = newStack();
  struct Stack* postStackPtr = &postStack;
  struct Stack* opStackPtr = &opStack;
  union StackElement elem;
  static char postfixExpr[MAX_SIZE];

  for(int j = 0; j != exprLength; ++j) {
    if (expr[j] >= '0' && expr[j] <= '9') {
      elem.charValue = expr[j];
      push(postStackPtr, elem);
    }
    else if (expr[j] == ')' && !isEmpty(opStackPtr)) {
      elem = pop(opStackPtr);
      push(postStackPtr, elem);
    }
    else if (expr[j] == '+' || expr[j] == '-' || expr[j] == '*' || expr[j] == '/') {
      elem.charValue = expr[j];
      push(opStackPtr, elem);
    }
  }

  while(!isEmpty(opStackPtr)) { push(postStackPtr, pop(opStackPtr)); }

  for(int j = 0; j != (postStack.top + 1); ++j) {
    postfixExpr[j] = postStack.contents[j].charValue;
  }

  postfixExpr[postStack.top + 1] = '\0';

  return postfixExpr;
}

// int main() {
//   char expr[] = "(((1 + 3) / (3 - 1)) + 5)";
//   char* postfixExpr = translate(expr, sizeof(expr)-1);
//   printf("%s\n", postfixExpr);
// }
```

This file also includes *stack.h* to make use of the stack struct and functions. This file only has one function, *translate()*. Like *validate.c*, it accepts a pointer to a string and the length of the string as arguments. It then initializes two stacks, one for holding operators and the other for holding the postfix expression, a *StackElement* for pushing elements onto the stacks, and a static string of size *MAX_SIZE* that will end up holding the end result of the translation. It is made static so that it is not deleted once the function ends, allowing its contents to still be pointed to by the code for the evaluation step.

The function loops over the expression check each character to see if they are a number, a close parenthesis, or an operator. If it is a number, the *StackElement* is set to have its value and is then pushed onto the *postStack*. If it is an operator, the *StackElement* is set to have its value and it is then pushed onto the *opStack*. If it is a close parenthesis, the *StackElement* is set to the return value of popping the *opStack* and then pushed onto the *postStack*, moving the top item from the stack of operators to the stack for the postfix expression.

Once that loop is finished, another loop is used to pop each item off the operator stack and push them onto the postfix stack. Then, to get the contents off postfix stack and into a string in the correct order, the values must be pulled as if the stack were a queue instead. To do this, I iterated over the stack and used a counter that started at 0 and incremented each iteration to take the stack element at the index of that counter and assign its value to the character of the string *postfixExpr* at that index. This copies the stack from the “bottom” to the “top”, assigning each element to the string. Then, I ensure that the *postfixExpr* string is null terminated before returning it so that the evaluation code can more easily detect the end of the string.

The last file to cover is *evaluate.c*.

```c
#include "stack.h"

int evaluate(char* expr, int exprLength) {
  struct Stack numStack = newStack();
  struct Stack* numStackPtr = &numStack;
  union StackElement elem;
  int j, x, y;

  for(j = 0; j != exprLength; ++j) {
    if (expr[j] >= '0' && expr[j] <= '9') {
      elem.intValue = (expr[j] - '0');
      push(numStackPtr, elem);
    }
    else {
      y = pop(numStackPtr).intValue;
      x = pop(numStackPtr).intValue;

      if (expr[j] == '+') { elem.intValue = x + y; }
      if (expr[j] == '-') { elem.intValue = x - y; }
      if (expr[j] == '*') { elem.intValue = x * y; }
      if (expr[j] == '/') { elem.intValue = x / y; }

      push(numStackPtr, elem);
    }
  }

  return pop(numStackPtr).intValue;
}

// int main() {
//   char expr[] = "13+31-/5+";
//   printf("%i\n", evaluate(expr, sizeof(expr) - 1));
// }
```

This file also imports *stack.h*, and its only function accepts a pointer to a string and the length of the string as arguments. It starts by initializing a stack to hold the integers and a pointer to the stack. It declares a *StackElement* and 3 ints. It then loops over the postfix expression it was passed. Since a correctly translated postfix expression only contains numbers and operators, that is what it checks for. If the character is a number, it converts the number into an int by subtracting it from the char *'0'*, which has the same value as the offset of each char representation of numbers from 0 to 9. The number is then assigned to the int value of the *StackElement* and then pushed onto *numStack*. If the l character isn't a

number, then it is an operator. In that case, it pops two values off *numStack*. The first is assigned to *y* and the second is assigned to *x*. Then, depending on which operator the character is, the *StackElement*s int value is set to equal the x [operator here] y. This order ensures that the subtraction and division is done correctly. The *StackElement*, which now contains the solution to the operation that was performed, is then pushed onto *numStack*. This repeats for every character, and once the iteration is finished, there will only be one value left on the stack. This value will be the answer to the evaluation of the expression, so it is popped off the stack and returned.