ELEC 278: Fundamentals of Information Structures
Lab 4: Queues and Recursion

Fall 2023–Instructors: Ni & Mertin

October 22, 2023

Please read the entire document to understand the requirements and process for completing the lab.

1 Objectives

The objectives of this lab are to further develop your understanding of queues and recursion, and their use in real-world applications, by completing an implementation of an interpreter for a simple scripting language[1] This program will recursively processes the input text, reading in instructions which manipulate a queue. At the end of the lab, your program will be able to run scripts which perform simple calculations with numbers using a queue, such as the following example (included as test case 4 in the starter code):

```
x = 8;
ENQ(1);
ENQ(0);
WHILE (0 < x) {
    y = DEQ;
    ENQ(y + DEQ);
    ENQ(y);
    x = x + -1;
}
ASSERT(y = 21);
```

This script computes the 8^{th} element of the Fibonacci sequence and confirms that the result is 21. You will build up to this final program by starting with the provided template code, which provides all of the structure of the program, and fill in the necessary parts, demonstrating your understanding of queue data structures and recursive algorithms.

2 Instructions

Download the file `lab4.zip` from OnQ and unzip it. Open the `lab4` folder that you extracted in either CLion or VS Code (note: you need to make sure you open the correct folder, which is, the one that directly contains the code files). Then, complete the following tasks.

This lab is due 11:30 AM (end of the lab session) on Thursday, November 9 for students in all sections. The scheduling of lab sessions is as follows:

- Tuesdays 12:30–2:30
- Wednesdays 11:30–1:30
- Thursdays 9:30–11:30

[1] A "scripting language" is a simple programming language which usually only has a minimal amount of features.

2.1 Task 1: Queue Implementation

The file `queue.c` contains starter code for an implementation of a circular queue containing integers. The type definitions, function declarations and documentation are given `queue.h`. Fill in each "todo" in the file:

1. Implement `enequeue` to enable entries to be added to the queue.
2. Implement `dequeue` to enable entries to be removed from the queue.
3. Implement `queue_empty` to check whether the queue is empty.

This code will need to take care of all aspects of updating the queue data structure, including reallocating memory where needed. It should shrink the allocated array in some way when the queue size decreases; the exact way you choose to do this (i.e., when and by how much you shrink the allocation) is up to you.

2.2 Task 2: Start Interpreter

The file `engine.c` contains starter code for an implementation of an interpreter for a simple scripting language, which operates on integer values using a queue and two variables (`x` and `y`). This language is specified as follows:

- The input is a sequence of (zero or more) statements, which are executed in the order in which they appear.
- A statement is one of the following:
  - An enqueue statement, `ENQ ( e ) ;`, where `e` is an expression (defined below). This statement evaluates `e` and enqueues the resulting value in a queue. For example:
  - An assert statement, `ASSERT ( c ) ;`, where `c` is a condition (defined below). This statement evaluates `c` and fails if it is false.
  - An assignment statement, `x = e ;` or `y = e ;`, which evaluates `e` and saves the result in the given variable.

  A typical scripting language may have many types of statements, including "if" statements, loops, function calls, etc. However, in this lab, we are limiting the scope of statements to only a few needed to make a very simple language which works for demonstration purposes. In Task 3, we will extend the language with loop statements.
- An expression is one of the following:
  - A variable, `x` or `y`, which evaluates to the current value of that variable.
  - An integer, with a possibly leading sign symbol.
  - The dequeue expression, `DEQ`, which causes a value to be removed from the queue and fails if the queue was empty.

  As with statements, a typical scripting language may support many types of expressions, including full arithmetic formulae with addition, subtraction, multiplication, division, etc. Here, we are starting with just numbers, variables and the "dequeue" operation; however, we will add support for addition in Task 3.
- A condition is one of the following:
  - The empty condition, `EMPTY`, which is true if the queue is currently empty.
  - An equals condition, `e1 = e2`, which is true if the expressions `e1` and `e2` evaluate to the same value.
  - A less than condition, `e1 < e2`, which is true if the expression `e1` evaluates to a value which is strictly less than the value of the expression `e2`.

Note that, in the case of the equals and less than conditions, evaluating the condition requires first evaluating the expressions on either side of the symbol; where applicable, the left expression should be evaluated before the right expression. For example, evaluating the condition `DEQ < DEQ` means dequeueing two elements from the queue, and then checking that the *first* element which was dequeued is less then the *second*; in that particular case, the evaluation order matters!

Take a look at the type definitions and function declarations in `engine.h` first. The type `struct context` defines part of the current *state* of the running script: what is currently in the queue and in each of the variables. The code in `main.c` initializes this to an empty queue and values of zero in the variables. The type `struct error` defines what information is communicated in case of an error. The 5 functions declared in this file implement the core of the interpreter, by handling the different kinds of constructs that could be expected to appear at a given point in the input. Two of these, `run_statements` and `parse_integer`, are already implemented for you; you can use their implementations as reference and examples when implementing the others.

Fill in each "todo" labelled "task 2" in `engine.c`:

1. Implement the three kinds of statements described above in `run_statement`.
2. Implement the three kinds of expressions described above in `eval_expression`.
3. Implement the three kinds of conditions described above in `eval_condition`.

Note the parameters of each of these functions. Each takes a context pointer `struct context *ctx`, which allows the queue and variables to be accessed and updated as needed. Each also takes a double character pointer `const char **input`; this means that the target of this pointer, `*input`, is itself a pointer to a sequence of characters (i.e., can be used as a string). A double pointer is used so that the character pointer itself can be updated; when the function returns successfully, `*input` should point to the first character *after* the construct that was just handled (e.g., the first character after the statement in the case of `run_statement`).

Finally, each function also takes an error pointer `struct error *err`. The function should return `true` if it is successful; if it is unsuccessful, it should return `false` and also set `err->pos` and `err->desc` to the position of the error within the input string (often this would be `*input`) and a human-readable description of the problem, respectively. See `parse_integer` for an example of this. Also, see how `run_statements` handles this as well; since it will only ever return `false` if either the call to `run_statement` or the recursive call to `run_statements` returns `false`, it does not explicitly set the contents of `*err`; it leaves it up to those calls to do so if and when they return `false`.

Note that, like in C, spaces, tabs, new lines, and other whitespace characters between symbols should be ignored; you can use the helper function `skip_whitespace`, declared in `util.h`, to facilitate this. You can also use the other function declared in that file `starts_with`, to assist in parsing the input. After finishing this task, you should be able to run the first 2 test cases successfully.

2.3 Task 3: Extend Interpreter

In this task, you will extend the scripting language as follows:

- A statement can now additionally be a loop statement, `WHILE ( c ) { stmts }`, where `c` is a condition and `stmts` is a sequence of statements. This statement implements a standard "while" loop by repeatedly evaluating `c`, exiting (successfully) if it is false, and executing `stmts` if it is true.
- An expression can now additionally be an addition expression, `e1 + e2`, which evaluates to the sum of the values of the expressions `e1` and `e2`. As with conditions in task 2, where applicable, the left subexpression should be evaluated before the right subexpression.
- A condition can now additionally be a not condition, `! c`, which evaluates to the opposite of the value of the condition `c`.

Fill in each "todo" labelled "task 3" in `engine.c`:

1. Implement the additional kind of statement described above in `run_statement`.
2. Implement the additional kinds of expression described above in `eval_expression`.
3. Implement the additional kind of condition described above in `eval_condition`.

This task will require using recursion in the implementations of these functions; for example, parsing an expression may require recursion in the case where it happens to be an addition expression. You will, however, need to make sure you don't create a situation with infinite recursion; for example, if `eval_expression` recursively calls itself, passing the same position in the string, to try to parse the left subexpression of addition, this will repeat infinitely many times! There are a few possible ways to get around this; here are two of them:

- Create a separate kind of construct, with a separate function to evaluate it, for an expression which is not an addition expression; call this a "guarded expression". Now, an expression is either a guarded expression (i.e., what used to be the definition of expression), or an addition expression `g + e`, where `g` is a guarded expression and `e` is an ordinary expression. This means that the recursive call to parse an expression only happens for `e`, which is closer to the end of the input than the whole addition expression, ensuring that the recursion is going to terminate at some point.
- In the implementation of `eval_expression`, use the fact that any expression starts with an expression which is not an addition expression; therefore, you can start with your existing logic for handling expressions from Task 2, and then check to see if a `+` symbol appears next, and again only make a recursive call for the expression after the `+`.

Ultimately, you can choose any approach which works and makes sense to you. After you complete this task, you should be able to run all test cases successfully.

3 Running the Code

As with Lab 3, you can run the "All CTest" option in CLion, or press the "Run CTest" button in the bottom bar of VS Code to automatically run the program against each of a set of test inputs (in the `test-cases` folder). This does not exhaustively check that your code is perfectly correct, but it is a good starting point. You can also run the program normally and type in your own input.

4 Marking Criteria

After completing all tasks, call over a graduate TA to mark the lab. Lab 4 has 10 marks in total:

- Is your implementation of queues in Task 1 correct? Does it correctly grow and shrink the allocation as needed? (2 marks)
- Is your implementation of the three interpreter functions in Task 2 correct? Do they correctly handle the input as specified? (3 mark)
- Is your extension of the three interpreter functions in Task 3 correct? Do they correctly handle the input as specified? (3 mark)
- Is your code sufficiently well-formatted and commented so that the purpose of each variable/field/parameter and the reason for each function call or data structure manipulation is clear? Refer to the guidance on the course assignment for expected levels of commenting. (2 marks)