The Queue Programming Language

A programming language to allow simple queue manipulation

Sam Messina
Ganesh Koripalli
Mohammed Abdulkadir

Features

1. Manipulate a single queue per program
2. Queue elements may be integers from 0 - 10
3. All major queue methods are implemented
    (a) Two versions of our language: compiled and interpreted
4. Displays parsing and analysis data in real time as it runs

Available Methods

ADD

ADD will add an element to the back of the queue

REMOVE

REMOVE will remove an element to the front of the queue

PEEK

PEEK will display the element at the front of the queue

LENGTH

LENGTH will display the current length of the queue

EMPTY

- Boolean expression to be placed inside an IF statement.
- Evaluates to true if the queue is empty.

NOT_EMPTY

Works like EMPTY only evaluates to true if the queue is not empty.

VIEW

Shows the queue in its current state

IF ()

- Can only take EMPTY or NOT_EMPTY as an argument.
- The line following the IF statement will be evaluated only if the IF statement is true.

BNF

```
<line>         ----> <expression>;

<expression>   ----> ADD <element> |
                     REMOVE        |
                     PEEK          |
                     LENGTH        |
                     VIEW          |
                     IF (<if_expr>)

<if_expr>      ----> EMPTY         |
                     NOT_EMPTY

<element>      ----> <int>

<int>          ----> 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10
```

How It Works - Compiled Version

Get The Tokens

1. Read in a text file to a string
2. Pass the source code string to Parser.java
3. Parser.java finds all the tokens and puts them in an array

4. The array is passed back to our main driver

Output

```
############## Parsing.... #################
Next token is ADD
Next token is 1
Next token is ;
Next token is ADD
Next token is 5
Next token is ;
Next token is ADD
Next token is 9
Next token is ;
Next token is ADD
Next token is 8
Next token is ;
Next token is VIEW
Next token is ;
Next token is REMOVE
Next token is ;
Next token is VIEW
Next token is ;
Next token is PEEK
Next token is ;
Next token is LENGTH
Next token is ;
Next token is VIEW
Next token is ;
Next token is IF
Next token is (
Next token is NOT_EMPTY
Next token is )
Next token is ;
Next token is VIEW
Next token is ;
Next token is IF
Next token is (
Next token is EMPTY
Next token is )
Next token is ;
Next token is VIEW
Next token is ;
```

Analyze The Tokens

This stage combines token analysis and writing to "machine code" (java)

1. The token array is passed into LexicalAnalyzer.java
2. Instructions are converted from our language's tokens to java code to be run
3. LexicalAnalyzer writes out a string of java code to a file, including
   (a) The Queue class that will act as our Queue model for our program
   (b) The Main Driver for our compiled program
   (c) Instructions gathered from token analysis
4. The new string of java code is written into output.java

Output

```
############## Analyzing.... #################
Next line of execution: queue.add(1);

Next line of execution: queue.add(5);

Next line of execution: queue.add(9);

Next line of execution: queue.add(8);

Next line of execution: queue.view();

Next line of execution: queue.remove();

Next line of execution: queue.view();

Next line of execution: queue.showFirst((Integer) queue.peek());

Next line of execution: queue.getLength(queue.size());
Next line of execution: queue.view();

Next line of execution: queue.view();

Next line of execution: queue.view();
```

Compile The Program

1. output.java is compiled to output.class using Runtime.exec().
2. output.java is deleted, leaving only output.class
3. output.class acts as our executable, the output from our pseudo-compiler

Output

```
############## Compiling.... #################


############## Done! #################
Your file is compiled. You can run it by running:

  java output

Happy queueing!
```

Use Case Example

```
$ java Queue myfile.queue
$ java output
```

How It Works - Interpreted Version

All The Steps At Once

- The logic behind the interpreted version is nearly identical to that of the compiled version.
- The major difference is the order in which everything runs
- No more separate parsing, analyzing, compiling, and running.

The Giant Loop

- Like the compiled version, our source code is translated into a string an passed to our Interpreter.java
- Interpreter has one loop that runs through the source code, parsing, analyzing, and executing as it goes.
- Once a token is found, it is analyzed.
- If analysis finds an instruction to run, the instruction will be run right away.