



# The Queue for Numeric Transformation

An Esoteric Programming Language by Dan Hetrick

Version 0.053

## Table of Contents

qunt.pl, the QUNT Compiler/Shell.....	2
The Queue.....	3
The Buffer and Conditionals.....	4
Displaying Text.....	5
The Shell.....	5
User Input.....	6
Examples.....	7
fibonacci.q.....	7
celsius.q.....	8
greater.q.....	8
greater2.q.....	9
average.q.....	9
QUNT Commands.....	10
License.....	11

## qunt.pl, the QUNT Compiler/Shell

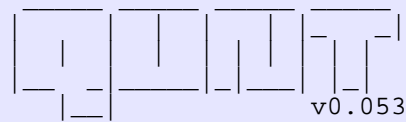
QUNT (pronounced "kyūnt", not...the *other* way) is an esoteric programming language influenced by Brainfuck. It features 26 different commands, uses a "queue" metaphor for data manipulation, and has very terse syntax. For example, here's a Fibonacci sequence generator written in QUNT. It will generate and display the first 10 iterations of the sequence:

```
>0&?>1&?+?:{7<+?:}
```

The official QUNT Compiler/Interpreter, **qunt.pl**, is written in Perl, and requires a default installation of Perl to run it and the compiled code produced by it. If called without any arguments, **qunt.pl** enters shell mode, where you can enter QUNT commands or statements and have them immediately executed.

The shell is the easiest way to write, debug, and test QUNT code. Tools are provided to execute code, view code that has been previously written, and write entered code to file.

```
localhost:~ user$ qunt.pl
```



QUNT Interactive Shell

Enter QUNT commands to be executed, or "dump" to show entered code

To write entered code to file, pass a filename as an argument to "dump"

Use the command "clear" to delete entered code from memory

.

If **qunt.pl** is executed with the word "debug" as the only argument, it will still enter shell mode, only it will display the Perl code generated by each QUNT command/statement, as well as executing the compiled code. For example, given the QUNT statement **>1>1+?** (which adds 1 and 1 together, and prints the result), this would be displayed in the shell:

```
. >1>1+?
*   push(@QUEUE,1);
*   push(@QUEUE,1);
*   $BUFFER = 0;
*   foreach my $e (@QUEUE){
*       $BUFFER += $e;
*   }
*   if($BUFFER){}else{ $BUFFER = 0; }
*   print $BUFFER."\n";
```

```
2
```

If a valid filename is passed as an argument to **qunt.pl**, the QUNT code contained in that file is compiled to Perl, and the resulting code is printed to STDOUT. If the code in the last example were saved to a file named **oneandone.q**, you could compile it with:

```
localhost:~ user$ qunt.pl oneandone.q
#
# ┌───┬───┬───┬───┐
# │   │   │   │   │
# └───┴───┴───┴───┘
#
#
# QUNT Compiler v0.053
# This program is the output of a compiler; please don't edit it

use strict;
use warnings;

my @QUEUE = (); my @WORK_QUEUE_COPY = (); my $BUFFER = 0;

sub qunt_is_number {
    my $n = shift;

    if ($n =~ /^\\d+$/) { return 1; } if ($n =~ /^-?\\d+$/) { return 1; }
    if ($n =~ /^[+-]?\\d+$/) { return 1; } if ($n =~ /^-?\\d+\\.?.*\\d+$/) { return 1; }
    if ($n =~ /^-?(?:\\d+(?:\\.\\d*)?|\\.\\d+)$/) { return 1; }
    if ($n =~ /\\D/) { return undef; } return undef;
}

push(@QUEUE, 1);
push(@QUEUE, 1);
$BUFFER = 0;
foreach my $e (@QUEUE){
    $BUFFER += $e;
}
if($BUFFER){}else{ $BUFFER = 0; }
print $BUFFER."\\n";

localhost:~ user$
```

A QUNT program can also be directly interpreted, without having to pipe it to Perl, by using the "run" argument to **qunt.pl**. Execute **qunt.pl** with the word "run" as the first argument, and the filename of the program you want to interpret as the second. If you're having trouble remembering all these command-line options, you can execute **qunt.pl** with the word "help" as the only argument to display usage information.

Piping code to **qunt.pl** is possible. Any code piped to **qunt.pl** is immediately interpreted, rather than compiled to source.

## The Queue

QUNT is based around the "queue": a sequence of numbers, of variable length, that is manipulated and operated upon. Besides the queue, QUNT features a "buffer"; the results of mathematical operations is stored in the buffer, buffer values can be added to the queue, and queue values can be stored in the buffer. Initially, the buffer is set to zero. This is the core concept of QUNT. The only command that does not operate on either the queue or the buffer is the **\$** command, which is used to display text (see

*Displaying Text*, below).

Think of the queue as a list of numbers. You can print these numbers, or perform mathematical operations on them. Math works differently in QUNT than in other programming languages. All math operations are performed on the entire queue, rather than on one or two values at a time. For example, addition is fairly normal. Given a queue with the values `[1,2,3]`, if we perform the addition operation (with the `+` command), the result "6" will be stored in the buffer; that is,  $1+2+3 = 6$ . Things get a little stranger when we use other mathematical operations. Given a queue with the values `[3,2,3]`, if we were to perform the exponential operation (with the `^` command), the result "729" will be stored in the buffer. That's because  $(3^2)^3 = 729$ . Each operation is performed on every value in the queue, in sequence.

Let's try another example, this time with the subtraction command, `-`. Given a queue with the values `[10,5,3,2]`, if we issue the `-` command, we end up with the result "0" in the buffer, because  $((((10-5)-3)-2) = 0$ .

The division command, `/`, works like the others. Given a queue with the values `[10,5,1]`, if we issue the `/` command, we end up with "2" in the buffer, because  $((10/5)/1) = 2$ .

## The Buffer and Conditionals

All command results that do not directly manipulate the queue are stored in the buffer. Conditional statements, that is, "if...then" statements, compare a given value to the buffer. For example, let's write a program that does a math operation, and displays text depending on what the output is. Our program will calculate  $2+2$ , and display a message if the answer is "4", and a different message if it is not:

```
>2>2+(4$84$104$101$97$110$115$119$101$114$119$97$115$102$111$117$114$10$"@ $84$104$101$97$110$115$119$101$114$119$97$115$110$111$116$102$111$117$114$10)
```

Save this to a file named `twoandtwo.q`. If we compile and execute the program, this is what happens:

```
localhost:~ user$ qunt.pl run twoandtwo.q
The answer was four
localhost:~ user$
```

If the value had *not* been four, it would have printed "The answer was not four".

The `~` command works differently depending on how many times it has been called inside a conditional statement. The first time it is called, it will execute the code following if the buffer is greater than the original condition. The second time it is called, it will execute the code following if the buffer is less than the original condition. For example, let's write a program that adds  $1+1$ ; if the result is "2", it will print the result. If the result is greater than "2", it will print the queue contents. If the result is less than two, it will exit without printing anything:

```
>1>1+(2?~%~!)
```

In pseudocode:

```
Push "1" onto the queue
Push "1" onto the queue
Perform an add operation
If the result is 2, print the buffer
If the result is greater than two, print the queue contents
If the result is less than two, exit
```

For a simple "else" branch, use the `"` command. For example, let's write a program that adds 1 and 1 together, and displays the queue contents if the answer is "2", and exits if not:

```
>1>1+(2%"!)
```

## Displaying Text

Because all command arguments must be numbers, displaying text in QUNT can be difficult. The `$` command is used for this; the command converts its argument from an ASCII code to an ASCII character, and displays it. To make it easier to display text, the compiler/shell has a special command-line mode. If `qunt.pl` is passed the argument "text", followed by the text to convert, it will generate QUNT code to display the desired text. For example:

```
localhost:~ user$ qunt.pl text Hello, world!
$72$101$108$108$111$44$32$119$111$114$108$100$33
localhost:~ user$
```

The code generated will display "Hello, world!".

## The Shell

For writing and testing QUNT code, the shell can be a useful asset. With the shell, you can write code and execute it immediately, without having to compile it or write it to file. The shell features two special commands, `dump` and `clear`. With the `dump` command, you can display all the code you've entered before and optionally write it to file. To simply display what code you've previously entered, enter the "dump" command without any arguments:

```
. >1>1+?
2
. dump
>1>1+?
```

To write the previously entered code to a file, just pass a filename as the first argument to the `dump` command. For example, to write the code in the last example to a file named `oneandone.q`, you could do:

```
. >1>1+?
2
. dump oneandone.q
Wrote dumped code to oneandone.q
.
```

The QUNT shell features one more command: **clear**. This deletes all the code that was previously entered (every time you enter QUNT code into the shell, it is saved; this makes it easier to grab a list of commands that you might want to use in a program).

## User Input

QUNT programs can get user input by using the **.** command. When used, this will allow the user to type in a number; this number will be stored in the buffer, where it can be placed in the queue, displayed, etc. For example, let's write a program that accepts user input; it will get three numbers from the user, and then display the average of those numbers:

```
$80$108$101$97$115$101$32$116$121$112$101$32$105$110$32$116$104$114$101$101$32$110$117$109$98$101$114$115$58$10. . . . :+@:>3/$84$104$101$32$97$118$101$114$97$103$101$32$111$102$32$116$104$101$115$101$32$110$117$109$98$101$114$115$32$105$115$32?
```

Save the complete program to a file named **average.q**, and run it:

```
localhost:~ user$ qunt.pl run average.q
Please type in three numbers:
1
2
3
The average of these numbers is 2
localhost:~ user$
```

## Examples

Here are some examples of QUNT code. The first example is a Fibonacci Sequence<sup>1</sup> generator. It will display a welcome message, then generate and display the first 10 iterations of the sequence:

```
$84$104$105$115$32$119$105$108$108$32$103$101$110$101$114$97$116$101$32$116$104$101$32$102$105$114$115$116$32$116$101$110$32$105$116$101$114$97$116$105$111$110$115$32$111$102$32$116$104$101$32$70$105$98$111$110$97$99$99$105$32$83$101$113$117$101$110$99$101$32$97$110$100$32$100$105$115$112$108$97$121$32$116$104$101$109$58$10>0&?>1&?+?:{7<+?:}
```

In pseudocode, this would be:

```
Print "This will generate the first ten iterations of the Fibonacci Sequence and display them:"
Push "0" onto the queue
Copy the last value in the queue into the buffer
Print the buffer
Push "1" onto the queue
Copy the last value into the queue into the buffer
Print the buffer
Perform an add operation
Print the buffer
Push the buffer value onto the queue
Loop the following seven times
    Delete the first value on the queue
    Perform an add operation
    Print the buffer
    Push the buffer value onto the queue
End loop
```

If saved to a file named **fibonacci.q** and executed, this produces:

```
localhost:~ user$ qunt.pl run fibonacci.q
This will generate the first ten iterations of the Fibonacci Sequence and display them:
0
1
1
2
3
5
8
13
21
34
localhost:~ user$
```

---

<sup>1</sup> [http://en.wikipedia.org/wiki/Fibonacci\\_number](http://en.wikipedia.org/wiki/Fibonacci_number)

Our second example is a program that converts a number into from Celsius to Fahrenheit degrees. The only flaw in the program is that if you try to convert "0", it will display an error message and exit.

```
$69$110$116$101$114$32$84$101$109$112$58$.
(0$69$114$114$111$114$33$32$73$110$112$117$116$32$99$97$110$110$111$116$32$98$101$
32$101$120$97$99$116$108$121$32$122$101$114$111$10!)
$67$101$108$99$105$117$115$58:>32-|:>5*||;/<`%
$70$97$104$114$101$110$104$101$105$116$58;>9*@;>5/@;>32+@;%!
```

If saved to a file named **celsius.q** and executed, this produces:

```
localhost:~ user$ qunt.pl run degrees.q
Enter Temp:32
Celcius:32
Fahrenheit:89.6
localhost:~ user$
```

**celsius.q** was written by Will Munslow, and submitted via an Internet forum.

Our third example is a clever program that takes two numbers in as input, and determines which one of the numbers is larger, or if the two values are equal. This is probably the most complicated program in the manual, and was written before the comparison operator `~` was created:

```
$101$110$116$101$114$32$102$105$114$115$116$32$118$97$108$117$101$46$46$46$46$32.
$101$110$116$101$114$32$115$101$99$111$110$100$32$118$97$108$117$101$46$46$46$32
:.-:-
(0
$116$104$101$32$110$117$109$98$101$114$115$32$97$114$101$32$101$113$117$97$108$46$
13$10! )
>0&;-:>0<<
{
(0 +(0 >0>0>1&|<< " >1>0&| ) )
(1 +(0 >2&|<< " >1& ) )
(2 &(0 +$109$97$120$61?! ) `(0 +$109$97$120$61?! ) >3&| |<>0 )
(3 `:<& (0 >1&|;&|;>4&| " >3&| ) )
(4 &| (0 >2&| " ;>4&| ) )
}
```

If saved to a file named **greater.q** and executed, this produces:

```
localhost:~ user$ qunt.pl run greater.q
enter first value.... 25
enter second value... 50
max=50
localhost:~ user$
```

Please note that if especially large numbers are entered, it may take the program some time to determine which value is greater. However, if you let it run long enough, it will produce the right result.

**greater.q** was written by Léon Planken, and was submitted via an Internet forum.



Since the creation of the `~` operator, Mr. Planken has written a newer, easier to understand version of **greater.q**:

```
$101$110$116$101$114$32$102$105$114$115$116$32$118$97$108$117$101$46$46$46$46$32.:
$101$110$116$101$114$32$115$101$99$111$110$100$32$118$97$108$117$101$46$46$46$32.:
-
(0$116$104$101$32$110$117$109$98$101$114$115$32$97$114$101$32$101$113$117$97$108$4
6$13$10!)
$109$97$120$61(0~`~&)?
```

It performs the same thing as the previous version, only done much more simply. It also works much faster than the previous version, even with very large numbers. If saved to a file named **greater2.q** and executed, this produces:

```
localhost:~ user$ qunt.pl run greater2.q
enter first value.... 10
enter second value... 9
max=10
localhost:~ user$
```

**greater2.q** was written by Léon Planken, and was submitted via an Internet forum.

The fifth example takes in any number of numbers, calculates their average, and displays it to the user.

```
$101$110$116$101$114$32$110$117$109$98$101$114$115$44$32$111$110$101$32$112$101$11
4$32$108$105$110$101$44$32$101$110$100$32$119$105$116$104$32$48$13$10
>0{.(0')>1&|;}
&(0$110$111$32$110$117$109$98$101$114$115$32$101$110$116$101$114$101$100$13$10)
>0&;-:
<{`0')<}
-:
<{`0')<}
-:<<`<:/
$97$118$101$114$97$103$101$61?
```

If saved to a file named **average.q** and executed, this produces:

```
localhost:~ user$ qunt.pl run average.q
enter numbers, one per line, end with 0
355
123
55
0
average=177.666666666667
localhost:~ user$
```

**average.q** was written by Léon Planken, and was submitted via an Internet forum.

## QUNT Commands

There are 26 different commands/operators in QUNT. All whitespace in a QUNT program is ignored. All command arguments must be numeric. Arguments are placed directly after the command; for example, to initiate a loop that will execute 3 times, you would use { 3.

<i>Command</i>	<i>Accepts argument?</i>	<i>Description</i>
.	<i>no</i>	Gets input from the user and stores it in the buffer.
!	<i>no</i>	Exits the program.
%	<i>no</i>	Prints the contents of the queue; each queue value is printed with a newline.
\$	<i>yes</i>	The argument to this command is treated as an ASCII code. It is converted to an ASCII character and printed.
<	<i>no</i>	Removes the first value from the queue.
	<i>no</i>	Removes the last value from the queue.
&	<i>no</i>	Set buffer to the last queue value.
^	<i>no</i>	Set buffer to the first queue value.
:	<i>no</i>	Adds the buffer value to the end of the queue, and resets the buffer to zero.
;	<i>no</i>	Adds the buffer value to the beginning of the queue, and resets the buffer to zero.
?	<i>no</i>	Prints the buffer value, followed by a newline.
@	<i>no</i>	Clears the queue.
+	<i>no</i>	Adds all values in queue in sequence.
*	<i>no</i>	Multiplies all values in queue in sequence.
^	<i>no</i>	Exponentiates all values in queue in sequence.
/	<i>no</i>	Divides all values in queue in sequence.
-	<i>no</i>	Subtracts all values in queue in sequence.
#	<i>no</i>	Performs a modulus operation on all values in queue in sequence.
{	<i>optional</i>	Begins a loop. If an argument is passed to the command, the loop will be repeated a number of times equal to the argument. If no argument is passed, the loop will be infinite.
}	<i>no</i>	Ends a loop.
'	<i>no</i>	Exits a loop.
(	<i>yes</i>	Begins a conditional code block. The block will only be executed if the buffer is equal to the value passed to the command as an argument.
)	<i>no</i>	Ends a conditional code block.
~	<i>no</i>	Branches a conditional code block. The first time it is called within a conditional, the following code will be executed if the buffer is greater than the original condition. The second time, the following code will be executed if the buffer is less than the original condition.
"	<i>no</i>	Branches a conditional code block ("else").
>	<i>yes</i>	Adds a value onto the end of the queue. The argument passed to it is the value added to the queue.

## License

*Copyright (c) 2013, Dan Hetrick  
All rights reserved.*

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- \* Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- \* Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.