The Q-BAL Programming Language

Keith Gaughan

April 13, 2004

Abstract

Q-BAL is a programming language that Ben Yackley and Michael Shulman invented on a whim, based on the question 'What would it be like if a language were based on queues rather than stacks?' The acronym stands for Queue-BAsed Lanugage. This language is not designed to be useful, just fun.

Contents

1 Introduction

1.1 Introduction to Queues

A Queue is a FIFO (First-In, First-Out) data structure, as opposed to a LIFO (Last-In, First-Out) structure such as a stack. A queue is like a list, or a line. The first one to get in line is the first to get out of line. The items get removed from the top of the queue and entered at the bottom. As in a stack, when something is taken off the top of a queue it is removed from the queue (except when the preservation prefix operator is used—see below).

All objects in Q-BAL are queues or functions. For now, we will just deal with queues of numbers (see below for functions and compound data types). A queue of numbers is a simple FIFO structure of integers (Q-BAL has no floating point capacity as yet).

1.1.1 Declaring Queues of Numbers

An object is declared by a declaration statement, as follows:

```
Q queue_name
```

This declares a queue of numbers named `queue_name`. The name is an alphanumeric sequence which can include letters, numbers, and underscores, but cannot begin with a number. There is another part to the declaration: the type. For now, the only type we'll be using is `Q`, meaning a queue of numbers.

1.1.2 Initialization

A queue may be initialized to a certain list of numbers. This is done using the `=` initialization operator and the `{ , }` literal queue operators. An example follows:

```
Q squares = {1,4,9,16,25}
```

The first number inside the `{ , }` comes at the top of the queue and the last comes at the bottom. Literal queues can also be used in expressions and assignments (see below).

1.2 Attachment

1.2.1 The attachment operators

All non-declarations in Q-BAL are assignments or attachments. That is, all statements that do not create a queue or function are classed as assignments or attachments and are thus built around the assignment operator, `=`, or one of the two attachment operators, `->` and `<-`.[1] We will discuss attachment first.

Of the two attachment operators, the second is identical to the first except with reversed arguments. That is to say, `x -> y` is equivalent to `y <- x`.

It is a good idea to be consistent in which you use, unless for some reason the clarity is improved by switching, or you are trying to confuse people. The queue from which the arrow points is known as the source, and the one to which it points is known as the destination. Attachment is also known as appending, or "sticking on the bottom."

1.2.2 What they do

What the first attachment does is to pop the top of `x` and append it to the bottom of `y`. That is, this code:

```
Q x = {1,2,3}
Q y = {4,5,6}
x -> y
y -> x
```

Will end up with `x = {2,3,4}` and `y = {5,6,1}`.

1.2.3 Arithmetic

Arithmetic may be performed on the number after it is popped and before it is attached. More than one number may be popped off the same or different queues and arithmetic may be performed on them. Reading from left to right in an expression, numbers are popped off queues in the order they appear. This is important if more than one number is to be popped off one queue in an expression. One queue can appear on both the source and destination sides, such as in the attachment `x + 1 -> x`, which serves as an increment if `x` is a one-element queue.

The arithmetic operators are `+`, `-`, `/`, `\`, `|`, and `^`. Of these, `+` adds, `-` subtracts, `/` finds the quotient, `|` finds the remainder, `\` multiplies (the opposite of division, naturally), and `^` exponentiates. Remember, these are all integer operators. The parenthesis operators, `(` and `)`, are used to separate operations and override the natural order of operations (which is `^ \ / | + -`). They are also often used for clarity.

[1] Unicode character /u2192 (and /u2090 the other way) is a nice arrow. Unfortunately most computers are still running on ASCII or ANSI so we are unable to type unicode characters in our programs. :(Hence we use the two characters `->`.

1.2.4 Logic

The traditional logical operators may also be applied. They return 1 if true and 0 if not true. They are used in program control (see program counter, below). They are ==, <, >, <= or =<, >= or =>, `!=`, and `!` (logical NOT turns nonzero into zero and zero into one).

1.3 Assignment

The assignment sets one queue *equal* to another. It does not modify the source queue (the one on the right), but the destination queue is completely overwritten by the source queue. An example will illustrate:

```
Q x = {1,2,3}
Q y = {4,6}
y = x
```

This example produces `x = {1,2,3}` and `y = {1,2,3}`. As it shows, the assignment operator does not care about how many elements are in the destination queue. There is no "flipped" form of the assignment operator as there is with the attachment operators; the left is always the destination and the right is always the source.

1.4 The Basic Prefix Operators

Prefix operators are an important part of Q-BAL. For the most part they control access to a queue or force different types of access to a queue. They have higher precedence than any other operators and are evaluated from inside to outside.

1.4.1 The Preservation Operator (*)

The preservation operator returns the top element of a queue without that element being removed from the queue.

1.4.2 The Number Operator (#)

The number operator returns the number of elements in the queue it is applied to. It is often used by functions to determine if they have enough arguments to execute (see functions, below).

1.4.3 Other Prefix Operators

The `&`, `@`, `~`, and `:` operators really only have meaning when used with functions and will be discussed in that section. The `$` and `%` operators only have meaning when used with different data types and will be discussed in that section.

1.5 The Use of the Null

In either an attachment or assignment, either the source or the destination (or theoretically both, but this would be useless) may be left blank. This is called the "null queue." The null queue is never modified and is considered to have no elements. Here are the possible combinations:

```
x ->
-> x
x =
= x
```

Of these, the fourth is the only one with no use whatever. The first pops an element from `x` and discards it, and the third erases `x` completely (i.e. makes it a null queue.). The second does nothing to `x` if it is a queue, but if `x` is a function it makes the function's code execute without storing anything in the input queue (see functions, below).

1.6 Comments

A comment is, as in any programming language, a statement ignored by the compiler/interpreter. The sole function of comments is to make the program more readable to the programmer and anyone else. In Q-BAL all characters between a backquote (`) and carriage return/linefeed are considered comments and ignored by the compiler or interpreter.

1.7 Compiler/Interpreter Directives

As in other languages (especially C) the programmer in Q-BAL can instruct the compiler or interpreter in certain ways. In Q-BAL any line *beginning* in a period (`.`) is considered a directive. There must be one letter after the period designating what type of directive it is, then a space and any parameters required by the directives. At the time this is being written there are only three directives:

```
.I "included.qbl"
.P ;-1 -> ;
.M print 'out?
```

The first tells the compiler/interpreter to include a specified file at that point. The pathname can be relative or absolute. The second changes the end-of-line statment from the default `;+1 -> ;` to anything the programmer specifies. (We'll find out about `;` and program control later.) The third creates a macro, like C's `#define` statement. Macros and other predefined objects are generally made with ALL CAPS.

2 Input and Output

2.1 Stdin and Stdout

2.1.1 The Predefined Queues `in` and `out`

Input and output to stdin and stdout—that is, from the keyboard or to the screen—are handled by a group of predefined queues. The simplest of the group are `in` and `out`. `in` can only be a source and `out` can only be a destination. Numerical input and output is handled with attachments, as follows:

```
Q x
in -> x
x + 1 -> x
x -> out
```

This will input a number, add one to it, and output it in numerical format. Note that at the end of this code snippet `x` is a null queue again (has no elements).

2.1.2 ASCII Character input and output

Character ASCII values can be input and output with the `'in` and `'out` predefined queues. `'in` reads a character from stdin, and `'out` outputs a character to stdout, in much the same way as before. They both convert to or from ASCII values, because queues only store numbers.

2.1.3 String and multiple number input and output

So far we have only read one number or character at a time. But input and output would be tedious indeed if this were the only way. (Not that that would have stopped us from leaving it at that if it were in any way difficult to engineer longer input and output, however.) However, we can use the assignment operator rather than the attachment operators. Therefore, this code:

```
Q x
x = 'in
'out = x
```

Will read a string from stdin until return is pressed and output it to stdout. Note that at the end of this code snippet `x` still contains the string, as the assignment operator does not modify the source. `in` and `out` are most often used with attachment, while `'in` and `'out` are most often used with assignment, but either can be used with either.

2.1.4 Literal Strings

The double quotes `"` are the literal string operator. A literal string such as `"This is a string"` is equivalent to the queue of the ASCII values of the characters in the string (which, for the example, is a 16-element queue).

2.2 Variable Input and Output

The predefined queues `in`, `out`, `'in`, and `'out` pretty much cover the keyboard and the screen (stdin and stdout). However, those are not the only sources of input and output that a program will want to utilize. Q-BAL provides variable input and output for those who want to use other sources of input and directions of output.

2.2.1 The Predefined Variable I/O Queues

The Variable I/O system introduces five new predefined queues. They are `?`, `in?`, `out?`, `'in?`, and `'out?`. The first, represented by a single question mark, is a queue of all the sources and destinations the programmer may want to use. It begins the program as a null queue, and while it is a null queue the use of the other four will cause an error to be generated. Variable I/O is initialized by putting a number (usually defined by the `.M` directive to be a descriptive word) into the queue `?` as follows:

```
PRINTER -> ?
```

Assuming that earlier in the program there was a `.M PRINTER x` statement, where `x` is whatever number the compiler/interpreter decides represents the printer, this statement initializes the variable I/O queues. The other four variable I/O queues function exactly like their standard I/O counterparts except that they input from and output to only whatever area is listed at the top of the `?` queue.

2.2.2 I/O include files

You may have noticed that the last section was rather vague on what numbers represent which sources of input and destinations for output. This is because the representation is dependent on the compiler/interpreter. Luckily, the compiler/interpreter is required to provide include files (usually `.inc` or `.qbi`) which include the neccessary `.M` statements for common devices such as printers, mice, modems, etc. Also sure to be included are library functions which handle these devices, so that it is generally unneccessary to access them directly. (Don't worry, we'll get to functions in a bit.)

3 Program Control

3.1 The Program Counter

There are no if, then, while, for, etc. loops or even a goto statement in Q-BAL. Program control is handled by explicitly manipulating the program counter, a predefined queue represented by a semicolon (`;`). The program counter normally has one element in it, representing a line number (after all `.I` inclusions: see below), starting from 1 at the first line. Blank lines and comments are not counted. So the following line:

```
0 -> ;
```

Will reset program execution to the beginning of the program. But in reality, it is never necessary to know the exact line number of any given line. Labels and relative branching are usually sufficient, as we will discover later.

3.2 Program Execution

You may have been wondering why the code snippet in the previous section was `0 -> ;` and not `1 -> ;`. Normally, program execution follows this pattern: The compiler/interpreter compiles/interprets the command at the line number at the top of the program counter, then executes the statement `;+1 -> ;` and repeats. If not disturbed, therefore, execution will begin at line one (since `; = {1}` at the start) and continue through the program one line at a time. Therefore you must always set the program counter to the number one less than the line number you want to branch to.

3.3 Relative Branching

To imitate if, while, for, etc. statements, we only need to be able to say "go back `x` lines if such-and-such is true." This is done by multiplying `x` by the result of a logic operation, which is 0 if false and 1 if true. The code snippet

```
Q x
in -> x
; - 2 \ (x != 0) -> ;
```

Illustrates this concept, and will get input repeatedly until the number 0 is entered. Remember that after the line the statement `;+1 -> ;` is executed, so you must set `;` to the line number one before the line number of the line you want execution to jump to.

3.4 Multithreading

Very rarely, you may want to put two numbers into the program counter. This will cause execution to jump back and forth between two areas of the program, executing one statement, then another at a completely different address, then the statement after the first, then the one after the second, and so on.

3.5 Ending Execution

Program execution can be stopped by the statement `; ->` if there is only one number in the `;` queue or `; =` if there are more (except from inside a function as noted below. From inside a function these statements return from the function, and `:; ->` is required to end execution.). The compiler/interpreter will assume the program is over if there is no more code, so this statement is not really necessary. (A function will also return without it at the end of the code.)

3.6 Line Labels

As anyone who has programmed in primitive BASIC will know, it can be extremely cumbersome to have to know line numbers in order to control program flow. Fortunately, it is possible to "label" a line in Q-BAL, although (like so many other things in Q-BAL) it is not designed directly into the language. Consider this code:

```
Q LABEL
...
LABEL = ;
...
; = LABEL
```

The first statement functions as the "label," storing the program counter value (i.e. line number) at that point. It does not modify the program counter. The second statement reassigns that line number to the program counter, causing execution to jump to the line just after the label. Voila! And all without knowing what that line number is. Note that the second statement does not modify the queue `LABEL`, allowing it to be used again and again.

4 Introduction to Functions

4.1 What is a Function?

A function is the only other object type in Q-BAL, besides the queue. We will start with functions on numbers, as we did with queues of numbers. A function is composed of three elements: an input queue, an output queue, and an instruction queue.

A function is *called* (although that term is not usually used in Q-BAL) whenever an object is placed on its input queue. This is normally done by treating the function name as if it were a queue name and attaching something to it. When this occurs, a *sub-program-counter* is invoked and the instruction queue of the function activates.

4.2 Declaring Functions

A function on numbers is declared in the same way as a queue of numbers, except with an `F` instead of a `Q` for the type, as follows:

```
F func_name
```

The same restrictions on the name of a function apply as on the name of a queue.

4.3 Initializing Functions

The instruction queue of a function is the only part that can be initialized. It is done as follows:

```
F square
 Q x
 *in -> x
 x \ in -> x
 x -> out
F
```

Don't worry about what that function does; we'll get to that in the next section.

4.4 The Code of a Function

Any statement valid outside a function is valid inside a function, and some others as well, with a few modifications. When the program counter is referred to inside a function, it refers to the function's sub-program-counter. When the predefined queues `in` or `out` are referred to, they refer to the input and output queues of the function. For this reason, one can attach to `in` or read from `out`, although it is considered bad form in most cases. A function can declare its own local variables, but they are reinitialized every time the function executes (see advanced functions for how to prevent this).

4.5 The Higher-Level Prefix Operator (`:`)

The higher-level prefix operator, when used inside a function, forces the name to refer to the object of the same name *outside* the function. For example, `:;` inside a function will refer to the original program counter, while `:in` will refer to stdin rather than the function's input queue.

4.6 Calling a Function

Whenever the name of a function is used on the source side of an expression it refers to the output queue, and is treated as if it were a normal queue. Whenever it is used as the destination, it refers to the input queue, and again is treated as a normal queue.

5 Advanced Functions

5.1 The Input/Output Override Prefix Operators (& and @)

From outside the function, it is possible to store to the output or read from the input. It is generally considered bad form to do so, but is sometimes neccessary. This is done with the Input/Output Override prefix operators. The `&` operator forces use of the input queue, and the `@` operator forces use of the output queue. When the `&` operator is used to store to the input queue (even if it would not be necessary) the function's code does *not* execute. Execution can then be forced later by storing a null queue to the function.

5.2 The Instruction Queue

Remember that a function consists of *three* queues? An input queue, an output queue, and an instruction queue? Well, the instruction queue isn't called a queue for nothing. You can manipulate it just like any other queue, as long as you use the instruction queue override prefix (`~`). Statments are enclosed in brackets `[]` to keep them as one item, but otherwise can be treated just like numbers (except that you can't do arithmetic with them, of course). The ability to manipulate the instruction queue can create very interesting self-modifying code. See Compound Data Types for examples of how this can be used.

5.2.1 The `code` Predefined Queue

There is another predefined queue, like `in` and `out`. It is `code` and it refers to the instruction queue of the current function, just as `in` and `out` refer to the input and output queues of a given function. In the main program, it refers to the main instruction queue.

5.3 Object Oriented Programming in Q-BAL

OOP (Object Oriented Programming) can be done in Q-BAL, but it is primitive compared to other languages such as C++. Then again, Q-BAL is not a object-oriented language, so it's surprising that you can do it at all. It's really a consequence of the capabilities of functions, and not a real designed capability of the language.

A function can act as an object. If it needs to store "member variables" it can do so in a static local variable (see above). The instruction queue can be set to do different things depending on how many arguments are put in the input queue at the same time (or even what type of argument: see variable data types, below), thus acting as different functions. Data can also be stored in the output queue. One function can be assigned to another of the same data type, replacing the target's instruction queue, input queue, and output queue.

5.4 Variable I/O and Functions

Since each function has its own `in` and `out` queues, it stands to reason that it would also have the corresponding variable I/O queues, and this is in fact the case. Each function has its own of all five variable I/O queues, and they function exactly the same as in the main program. Furthermore, if a function's `?` queue is nonempty, then input to the device it specifies is considered input to the function and will cause execution of the instruction queue. In this way, Q-BAL can create an event-driven program. For this purpose, there is a stdin handle that can be stored to `?`, even though the main program does not require it. Stdin can be accessed from a function through `:in` but this will not cause function execution. Also note that in order for this to work, the `?` queue is static; that is, unchanged between function calls.

5.4.1 File Handling

File handling is done through variable I/O, but since the programmer needs to tell the computer what file to open before I/O is possible, it is handled through a function rather than a macro, and now that we know all about functions, we can cover it. `FILE` is a predefined function which takes as input a string which is the filename, with either a local or global path. If the filename does not exist, `FILE` will create it. It then

returns a "file handle" which can be stored to ?, allowing I/O to that file. File I/O using FILE is sequential and appending. That is, if ?={filehandle}, then attachments from in? read one character at a time from the top of the file, sending an EOF when the file is over, and attachments to out? append characters to the end of the file. Assignments from in? read one line at a time, and assignments to out? write one line at a time (as is standard with I/O queues). There are also other miscellanous file-handling predefined functions such as FQ DELETE, which takes a queue of files to delete.

5.5 Static Local Variables

Functions can declare local variables, in the normal manner of variable declaration, but these variables do not keep their values when the function is called multiple times. Many functions can get around this by keeping values on the input or output queues, but sometimes this isn't sufficient, especially for OOP-like functions. One way to get around this is by declaring a queue outside the function that is only used from inside that function, but this is cumbersome.

5.5.1 The ? is Static

Each function has its own ? queue which remains unchanged from calling to calling, like a static variable. It is therefore possible to store single values in the ? queue between function callings, as long as the function does not utilize variable I/O. This is dangerous, however, because if the number stored happens to be the handle for some device, then the function may be inadvertently called by input to that device.

5.5.2 Dynamic Allocation

There is yet another area that variable I/O can handle: dynamic allocation of memory. C or C++ programmers will be familiar with this concept. To dynamically allocate memory for an object (of *any* data type), use the function F ALLOC. This function takes as input a string indicating the data type of the desired object (for example, FQQ) and outputs a handle that can be stored to a ? queue. When this handle is on top of the ? queue, then the variable I/O queues access the dynamically created object. in? functions exactly like &x and out? exactly like @x, for an object named x. The queues 'in? and 'out? are not well defined when ? holds the handle of a dynamic variable. When a function dynamically creates variables, they are static because their handle (presumably in the ? queue) is also static. This is the best way for a function to create static variables.

6 Compound Data Types

6.1 What are compound data types?

There are other types of queues and functions in Q-BAL than queues of numbers and functions on numbers. There are queues of queues, or functions on queues, or queues of functions. These are declared with a variable number of Qs and Fs for the type section of a declaration. For example, FQ sort would declare a function on queues. In general, a "queue" or "function" refers to a queue of numbers or a function on numbers. An example of a use of a queue of queues follows:

```
QQ x
Q y
y = 'in
*$y -> x
; - 3\(#y != 0) -> ;
$x -> 'out
; - 2\(#x != 0) -> ;
; ->
```

This complete program will read strings until a null string is entered (return with no other characters), and then print them all out in the same order they were entered, followed by a blank line. (Don't worry about understanding it now; we'll refer back to it as we go on.)

6.2 The "Greatest Common Denominator"

When an assignment or attachment includes different data types, the "Greatest Common Denominator Rule" is used to determine how the data is manipulated. The GCD of two types is the largest string of Qs and Fs that is common to both, on the right side. In the case of attachment, the GCD must be shorter in length than the shorter of the two types, while in assignment it can be equal to the shorter. The GCD is the longest data type that is manipulated. For any data type, if the GCD is more than one letter shorter than the type in question (if the operation is attachment) or if it is at all shorter (if the operation is assignment), then the excess is put onto the top of the queue (or the output queue for functions.) An example will serve to make this clearer:

```
FQQ x
QQ y
FQ z
...
y -> x
y = x
z = y
y -> z
x -> z
```

Admittedly, you aren't likely to run across anything nearly this complex. But it serves a good illustration. Let's take this in order. The first three statements declare x a function on queues of queues, y a queue of queues, and z a function on queues. The fourth line is an ellipsis, indicating that there is probably code in between, but what it is doesn't matter. Now we're at the confusing part.

The first attachment has a GCD of Q (it would be QQ, but an attachment GCD must be shorter than the shortest, which in this case is QQ). Therefore it works on queues. It pops the first queue off y (y is a queue of queues) and appends it to—what? Well, excess is put onto the output queue, so it appends it to the top queue of queues on the output queue of x, which is a function on queues of queues.

The second statement is an assignment with a GCD of QQ. It sets y (a queue of queues) equal to the top queue of queues in the output queue of x (a function on queues of queues) without popping it off x.

The third statement is an assignment with a GCD of Q. It sets the top of the output queue of z (a function on queues) equal to the top of y (a queue of queues) without popping it off y.

The fourth statement is an attachment with a GCD of Q working the same way between the same variables as the third statement. This one pops the top queue off y (a queue of queues) and appends it to the bottom of the input queue of z (a function on queues).

The fifth statement is an attachment with a GCD of Q. It pops the top queue off the top queue of queues of the output queue of x (a function on queues of queues) and appends it to the bottom of the input queue of z (a function on queues).

Got it? Good!

6.3 The Input/Output Override Prefix Operators Revisited

Remember the I/O override operators & and @ from advanced functions? They can apply to multiple data types as well. Normally, excess is put on the top, or the top of the output queue, but these can force it to be put on the bottom, or on the input queue. To this end, they can be applied to queues as well as functions. When used in this way to functions, sometimes they must be applied twice. The default is @ or @@, the top of the output queue. & or && will force it to the bottom of the input queue. &@ will force the bottom of the output queue, and @& the top of the input queue. Needless to say, these are generally considered bad form. But hey, who cares?

6.4 Variable Data Types

An object may be declared as a type ending in X, such as QX or FQX. The X stands for "anything". QX means a queue of anything, and FQX means a function on queues of anything. When used in an expression, consider the X by default to be whatever will make the GCD the longest. If two X-types are used in the same expression, make both the Xs equal to whatever is at the top of the source.

6.5 The String Prefix Operator ($)

The String prefix operator $ forces the GCD to a higher level. One use of the string on the source will raise an attachment to the level of the smaller, if possible. For example:

```
QQ x
Q y
...
$y -> x
```

This code will leave y a null queue and append to x (a queue of queues) what it used to be. The string operator often leaves its operand a null queue. When used other than this, it causes things to be strung together. For example:

```
Q x
Q y
...
$x -> y
```

This code will string together all of x and append it to y. If `x = {1,2,3}` and `y = {1,4,9}`, then after this code executes `y = {1,4,9,1,2,3}` and x is a null queue. In general, be logical: it's usually pretty obvious what the string operator should do in any given situation.

6.6 The Diminish Prefix Operator (%)

The Diminish Prefix Operator % is the opposite of the string operator. It lowers the GCD by one letter. It is even more intuitive than the string operator and needs no example.

7 Andy's Comments (Appendix)

This is a modification to the Q-BAL language to support multithreading.

Functions were previously declared as:

```
context f_name
    = {
    Qexpr code
        = {
        function_code
        ...
        }
    Qchar in
    Qchar out
    }
```

However, this format is not conducive to multi-threading due to the case of concurrent function calls. In other words, what if two threads are executing the same function or context at once?

The resolution to this problem is to have *implied* in and out queues, defined by the code declaration, as in C++, so that data space can be allocated on an as-need, per-thread basis.

So the above function definition changes to the following:

```
context f_name
    ={
    Qexpr f_name {in_types}
        ={
        func_code
        } out_type
    }
```

By calling the context's name as a function, then the context will call its member function that has identical arguments, so supporting function overloading (type-sensitive calls).

An example of the above implemented in a real class would be as follows:

```
context square
    ={
    Qexpr square {Qfloat inputs}
        ={
        inputs^2 => outputs
        } Qfloat outputs
    }
```

Thus to square a queue, you merely call

```
Qfloat someData = square(sourceQueue)
```

Another critical modification that must occur to do true multithreading, is restructuring of the ip, now a context.

One thing to remember, is that all calls to ip will be *from* ip =)

```
context ip
    ={
    QQexpr private code
    Qlong private handle
    Qexpr private currentCode

    Qexpr createThread {}
        ={
        :@code -> code
        ; -- Assign unique handle -- how??
        }
    Qexpr create {}
        ={
        program.code -> code
        handle = {0}
        }
    Qexpr deleteThread {}
        ={
        code ->
        handle ->
        }
     Qexpr exec {Qchar arg}
        ={
        @code -> currentCode
        ~code -> code
        handle -> handle
        } int returnVal
```

```
Qexpr synch {}
   ={
   ; -- this func makes threads wait at this place until
   ; -- all the threads reach here.
   }
}
```

Also, I made a context called “program” which is, well, the whole program. all it consists of is Qexpr data. also, the above does not have a modifiable eol. Seriously, though, why do you need to be able to make eol = `{[ip - 1 -> ip]}`?? Also I killed the program counter because, well, I can’t directly access data in a random place in a queue.

`ip.create` is called upon program start, by the program context.

8 Revision Notes (Appendix)

8.1 Primitive Types

`char`	Character
`int`	Integer
`float`	Floating point
`expr`	Expression or statement
`type`	Type of object
`var`	Variable type
`Q_`	Queue of _s (nestable to get QQ_, etc.)

You cannot declare a primitive type variable without a `Q` (or `F`; see below) in front of it. I know that’s a throwback to the old cuteness that Andy complained about, but I can’t bear to take it out because it would un-queue-ify Q-BAL so much.

8.2 Prefix Operators

`@`	Peek	Returns top without popping it
`#`	Count	Number of elements in queue
`$`	Top	Considers the top of the queue
`%`	Bottom	Considers the bottom of the queue
`:`	Higher-level	Context containing current context
`.`	In current context	As opposed to local variables
`_.`	In context named _	Subcontext of current one
`?`	Type of object	Returns a type variable
`~`	Evaluate	Executes expressions and statements, returning their value.

- `$` and `%` are used mostly with nested `Q`s
- Can also “execute” type variables, returning a variable of that type, which can then be stored to a var type, or another type if you want to risk a type mismatch.

8.3 Arithmetic/Logic Operators

`+ - * / \ ^`	Guess what these do!
`& | !`	Logical and/or/not (nonzero = true, 0 = false)
`== != < > <= >=`	More logic operators

8.4 Statements

`->`	Attachment	Pops off source and appends to destination
`=`	Assignment	Sets dest equal to source
`=>`	Repeated attachment	Repeats `->` a number of times equal to #source
`~>`	Repeated attachment with recycling	Same as `=>` except before each attachment, @source `->` source is executed

All these abort immediately if any types are incompatible. They return the value of the destination queue after all operations, so `x -> y -> z` is possible.

`;` Comment (ignored)

8.5 Literals

`{,,}`	Queue literal (nestable)
`"..."`	String literal = Qchar
`'.'`	Character literal = char
`[...]`	Expression literal = expr
`‘...‘`	Type literal = type

Qexpr can be literalized either as `{[...],[...]}` or as

```
{
...
...
}
```

with carriage returns taking the place of `],[` in the literal. Qtype can be initialized in the same way, rather than `{‘...‘,‘...‘}`.

8.6 Compound Types

`context`	Execution context	Function
`class`	Template for contexts	
`F_`	Function on _	Specialized context

Contexts and classes can be declared without a `Q` or `F` in front, because they are already queue-like compound types.

8.7 Predefined Queues

These queues occur in all contexts automatically. They can be redefined to initialize them (all) or have different types (in and out only).

```
Qexpr code = {}
Qexpr eol  = {[.ip + 1 -> .ip]}
Qint  ip   = {1}
Qexpr exec = {[#.in > 0]}
Qvar  in   = {}
Qvar  out  = {}
```

These queues all have special places in the use of the context. Whenever @exec is true (nonzero), then execution of code begins, counting lines with ip. eol is executed at the end of each line. Whenever the context name is used as a source, it actually refers to out, and when it is a destination, it refers to in. Note that with all default definitions, the code will execute line by line whenever something is input.

8.8 To Define a Context (or Class)

```
context context_name = {
  Qexpr code = {
    ...
    Qint local_variable
    ...
  }
  Qchar in
  Qfloat out
  Qint context_variable
}
```

Local variables are recreated whenever the code is executed. context variables stay the same for a given context instance. classes are defined the same as contexts, and instances of a class are declared as follows:

```
context context_name = ~class_name
```

8.9 Functions

```
Fchar function_name = {
  ...
  ...
  ...
}
```

Looks exactly the same to the interpreter as

```
context function_name = {
  Qexpr code = {
    ...
    ...
    ...
  }
  Qchar in
  Qchar out
}
```

It's just a kind of shorthand.

8.10 Anatomy of a Variable Reference

```
[#~?@]?[$%]*[:]*[.]?    all followed by the variable name.
```

I hope I got the regexp right, plus 2 things I didn't know how to do in it:

1. you can have `~@` meaning execute top without popping.
2. `.` can be preceeded by a context-name to specify which sub-context.

8.11 Problems

How could you program it so the code executes whenever output is demanded, rather than whenever input is recieved?[2]

[2] I now think this can be fixed with some modification of the multithreading model in Andy's comment.