

The Book of QUACKERY

[build do] is quackery

*a lightweight, open-source
language for recreational
and educational programming*

Foreword

On the 30th of November, 2018, I posted on Facebook that I had an idea for a programming language to be called Quackery, along with a first bit of tentative code towards that goal. Today, on the 30th of November, 2020, I find myself finishing *The Book of Quackery* by writing the foreword. The seeds for it were planted a lot earlier, in the Christmas of 1979, when my parents gifted me a copy of the then new *Gödel, Escher, Bach: an Eternal Golden Braid*, by Douglas Hofstadter. I was seventeen, and, if I recall correctly, I had finished reading it for the first time by the New Year. This was what triggered my interest in computer programming, and in the following years, while lecturers at Plymouth Polytechnic were teaching me BASIC, Fortran and COBOL, I taught myself Pascal, Lisp and Logo, languages closer to the spirit of Gödel, Escher, Bach, and, to my mind, far more exciting and mind expanding.

After college I bought a Jupiter Ace, a tiny computer similar to the Sinclair ZX81 in design, but with a significant difference; rather than running BASIC, it ran a language called Forth, which I knew of only by repute, as a difficult and obtuse language. I found it to be anything other than that. Yes, much of it was unfamiliar, but it was constructed on familiar principles, and had a simplicity and elegance in its conception that challenged much of the received wisdom I had been taught at college regarding the complexity of designing and implementing a programming language. It reminded me very much of Lisp because in many ways the two languages are two sides of the same coin. I would describe their relationship like this.

Forth and Lisp are mirror images. Their opposites include postfix v. prefix, static allocation v. dynamic allocation, explicit v. implicit stack. The primary point of coincidence is that executing a Forth word and evaluating a Lisp function are both depth first tree traversals. They are both interactive and extensible, and simple enough to understand all at once.

Lisp comes from academia and the lambda calculus, a world where more abstract means more fundamental, and computing is mathematics made real with information processing technology.

Forth comes from pragmatism and electrical engineering, where more fundamental means closer to the silicon, and software is hardware by other means.

In short, they are as different as Church and Turing, which is to say demonstrably equivalent.

The path of one's life is seldom a straight line, and I set computer programming to one side to pursue other things for several decades, until the winding paths I had followed crossed with programming again, and I had the inclination to take it up once more as a pastime.

What took me by surprise is that my early interests had not so much withered from lack of use as sat on the back burner of my subconscious, gently simmering away until they were one thing, a language that encapsulates everything I find amazing about computing, and that retains the features I most love about programming; being simple enough to expound in a short book, whilst running on modern hardware rather than the clunky keyboards and tiny screens of the 1980s, which I do not look back on with such sentimentality.

I dedicate this book to my wonderful wife and children, Maya, Alex, and Laura, whom I love dearly. I thank them for abiding my eccentricities and supporting my endeavours.

I dedicate the language Quackery to those intellectual giants; Moore, McCarthy, Wirth, and Feurzeig, Papert & Solomon, on who's shoulders I have been honoured to stand.

If nothing else, I hope you get from this book the idea that you can set yourself a goal you feel is just beyond your reach, a small mountain to climb, and achieve that goal while enjoying every step of the journey, as I have done.

Contents

What is Quackery?

What to expect of Quackery and The Book of Quackery.

Installing Quackery

How to get Quackery up and running on your system.

Quackery says Hello World!

The obligatory first experience, in a surprising amount of detail.

Quackery is Forwards Lisp

The only barrier to Reverse Polish Notation is unfamiliarity.

The Quackery Stack, More Stacks, and The Other Stack

The hurdle at the start of the learning curve. We take the hurdle slowly.

Stack Comments

A convention worthy of an entire page to itself.

Stackrobatics

A necessary skill you can learn with practice and examples.

Quackery Control Flow

Quackery flows from left to right, with skipping and jumping.

Dealing With Quackery

Strings and nests. Examples to work through.

Problem, Problem, Problem

There are no errors in Quackery. That's because we call them "problems" instead.

Vulgar Quackery

Extending Quackery to include fractions.

Sorting and Searching

Testing a Quackery design decision, and implementing a paradigm outside of its bailiwick.

Quackery is Not Forwards Lisp

...but it could be. Dispensing with jumping around.

What's in a Name?

A summary of some key points before we get to the long list of word behaviours.

Words, Numbers, and Nests

Mostly a contents page for the long list of word behaviours.

Word Behaviours

The long list of what every word does, with example code.

Quackery in Python

49,000 characters (including whitespace) of Python, half of which is Quackery.

Under the Quackery Bonnet

Notes about the Python code.

What is Quackery?

Quackery is a lightweight, open-source programming language suitable for recreational and educational purposes. It combines ideas from Lisp and Forth, so is quite different to many mainstream languages.

From a technical point of view one could think of it as an assembly language for a stack machine (i.e. a computer processor that provides support for two or more stacks rather than the almost ubiquitous registers and call stack model) that directly supports a dynamically allocated memory model (i.e. dynamic arrays), rather than a contiguous array of memory locations, (i.e. RAM). While there are some hardware implementation of stack machines, other than the occasional research paper that considers hardware support for aspects of dynamic memory management, dynamic allocation remains firmly in the province of software, so Quackery runs on a virtual processor.

Virtual stack machines are quite commonplace. Many high level languages, including Java and Python, compile byte code for a virtual stack machine, and languages such as C++ and Rust can compile to WebAssembly, the emerging standard for a virtual stack machine built into web browsers, and currently supported by Chrome, Firefox, Safari and others.

However, describing Quackery as an assembly language fails to convey the full nature of the language. It neither looks nor feels like a conventional assembler, and bears more in common with structured high-level languages. Some idea of its simplicity and power can be gleaned from a few statistics about the implementation presented here, which amounts to thirty two A4 pages of well-spaced 12pt text, totalling a mere 49,000 characters, including spaces and carriage returns.

The first nine pages cover exception handling and define the fifty three machine code operations of the Quackery processor. The next one and a half pages implement the Quackery processor as a depth-first traversal of nested Python lists (dynamic arrays). After that three and a half pages are used to define a basic Quackery compiler from first principles, enabling the rest of Quackery to be defined in Quackery. The “rest of Quackery” is sixteen pages of well formatted Quackery code, (under 26k) that extends the language from the machine code operations stepwise up to include an extensible compiler, decompiler, and REPL (interactive language shell). One more page of Python brings it all together, and the final page allows Quackery (which is defined as a single Python function) to run in the Terminal app.

The Python code is optimised for legibility and makes minimal use of Python libraries, uses a tiny subset of Python 3 and avoids the use of idioms that may be unfamiliar to non-Python programmers wherever reasonable, to make it amenable to re-implementation in other languages.

Expectation Management

Learning stack based programming can present a hurdle for programmers unfamiliar with the paradigm. Fear not! This will be covered in some detail.

Quackery is not overly burdened with data types or a broad variety of data structures, so there are plenty of opportunities for extending the language. Similarly it knows very little about input and output. User I/O is text based, and the file handling words are rudimentary and idiosyncratic. Programming in Quackery is reminiscent of the halcyon days of home computing in the mid-1980s, without the funky, chunky graphics, but with a better keyboard and a much larger screen.

A crude back-of-the-envelope calculation involving a recent Mac mini, the naive recursive Fibonacci benchmark and some abuse of Moore’s Law suggests that Quackery would be considered speedy if it ran at the same speed as this implementation on a similarly priced computer from 25 to 30 years ago.

Implementing Quackery in another language will be easiest if the language has the following things in common with Python; support for mixed type dynamic arrays with automated memory management, first-class functions and arbitrary-precision arithmetic.

Installing Quackery

The Quackery program that accompanies this document runs in the Terminal. It is a Python Script that requires Python 3 to be installed. It was developed on Python 3.6 through to 3.9 on Mac OS 10.13 (High Sierra) and 10.15 (Catalina), and Python 3.6 on Pythonista 3 for iOS. It is recommended that it be run on the most recent stable version of Python 3, if possible. At the time of writing the most recent version is Python 3.9.0. It can be downloaded from python.org. (For experts: Quackery has also been tested on PyPy 3.5.3 (v7.0.0) on Raspberry Pi OS, and PyPy 3.6 (v7.3.1), on Mac OS, and runs 20 times faster than using the regular Python 3 on the Pi 400, and 30 times faster on Macs.)

Mac OS Users

Python 3.9 requires Mac OS 10.9 (Mavericks) or later. The earliest version of Mac OS that is supported is 10.6 (Snow Leopard) and the most recent version of Python that runs on Snow Leopard is Python 3.7.6.

Download the Python 3 Mac OS installer from python.org and run it. To confirm it has worked, open the Terminal app, type

```
python3 --version
```

at the prompt, and press enter. It should respond with “Python 3.9.0” (or whatever version number you installed.)

The following steps are for novice Terminal users. Experienced users may prefer different approaches. You will need to know which shell your Terminal app uses. To find this out, enter

```
echo $SHELL
```

It should respond with either “/bin/zsh” or “/bin/bash”, depending on whether it uses the ZSH shell or the BASH shell.

Put the quackery folder in your Home folder, alongside the system folders; Applications, Desktop, Downloads, and so on. In the Terminal app, enter;

```
echo "alias quackery='cd ~/quackery; python3 quackery.py'" >> ~/.zshrc
```

if Terminal is running the ZSH shell, or

```
echo "alias quackery='cd ~/quackery; python3 quackery.py'" >> ~/.bashrc
```

if Terminal is running the BASH shell.

Now when you open a new Terminal window and type **quackery** it should change the current directory (the Unix word for “folder”) to the quackery directory, and run Quackery. Quackery should print this in the terminal window;

```
Welcome to Quackery.
```

```
Enter "leave" to leave the shell.
```

```
/O>
```

This is the Quackery shell, and the duck’s head prompt **/O>** indicates that it is ready for you to start programming in Quackery.

Raspberry Pi OS and other Linux Users

These instructions are based on the Raspberry Pi 400 (at the time of writing, the only version) but are readily adapted to other Linux systems. I suggest you put the quackery folder that contains this document in the home folder, alongside the Desktop, Documents and Downloads folders, et cetera.

Beginners: Open the Quackery folder and double-click on `quackery.py`. This will open the Thonny interactive development environment for Python. Click the Run icon. Quackery should print this in the lower panel, labeled “Shell”;

```
Welcome to Quackery.
```

```
Enter "leave" to leave the shell.
```

```
/O>
```

This is the Quackery shell, and the duck’s head prompt `/O>` indicates that it is ready for you to start programming in Quackery.

Seasoned hands: Unless you are planning to modify `quackery.py` (make a backup of the file first!) it is more convenient to run Quackery in the Terminal app. Open the app and navigate to the quackery directory with `cd`. Run Quackery in the terminal by entering “`python3 quackery.py`” Assuming the quackery directory is in the Home directory;

```
cd ~/quackery
python3 quackery.py
```

To automate this process, leave Quackery by entering “`leave`” and pressing return twice, then enter

```
echo "alias quackery='cd ~/quackery; python3 quackery.py'" >> ~/.bashrc
```

Now, next time you open the terminal you can just enter “`quackery`”.

Windows Users

This section is currently omitted as I do not have access to a Windows machine. Please contact me if you can provide a similar walk-through to that given for Mac OS and Linux users. Thank you.

All Users

The folder `sundry` contains some examples of code written in Quackery, including `demo.qky`, which contains a small selection of coding classics. It can be run from the terminal by entering `quackery sundry/demo.qky` – assuming you have set up the word `quackery` as an alias as described above.

Quackery and the Keyboard

Before we start, one word of caution. Quackery only understands a limited set of keyboard characters. They are the printable characters

```
0123456789AaBbCcDdEeFfGgHhIiJjKkLlMmNnOoPpQqRrS
sTtUuVvWwXxYyZz()[\]<>~+=-*/^\\_.,:;?!'"%&#$
```

and the whitespace keys `<space>` and `<carriage return>`. Anything else will be turned into a question mark during input and output. The reason for this is discussed in the User I/O section of Word Behaviours.

Quackery says Hello World!

It is traditional when learning a programming language to start with the question “how do you make it say Hello World!”? In Quackery, the answer is; “say "hello world"”. So type that in at the duck’s head prompt, then press enter twice.

```
/O> say "Hello World!"
...
Hello World!
Stack empty.

/O>
```

The question “how did Quackery do that?” has a longer answer, and by the time you have worked your way through this document you should be able to answer it yourself. That journey starts with the next thing people traditionally ask of a programming language, “what is 2+2?”. So type that in.

```
/O> 2+2
...
Unknown word: 2+2
Stack empty.

/O>
```

The first rule of Quackery is; everything is separated by spaces or carriage returns. It assumed that “2+2” is a word, and it is acceptable as such by Quackery standards, but it’s not a word it knows. So let’s try again, with spaces in between the numbers and the +.

```
/O> 2 + 2
...

    Problem: Stack unexpectedly empty.
Quackery Stack:
    Return stack: {[...] 0} {quackery 1} {[...] 6} {shell 5} {quackery 1}

/O>
```

The second rule of Quackery is; don’t fret over problems – they will happen a lot. When you program in Quackery you are communicating directly with hardware, albeit virtual hardware, and very simple hardware in many respects. It’s the sort of straight forward notion of a computer processor that you find in introductory textbooks, not the sophisticated and complex products you would find in a modern computer or mobile device. Consequently, if you inadvertently instruct it to do something impossible, it will report some diagnostic information that will make more sense as you advance in your understanding of Quackery. When a problem occurs, the stack will be emptied for your convenience. It is possible to inflict sufficient damage to Quackery to crash it completely. This will be covered later. To see that in action, you might try entering ' if take (including the apostrophe). That’ll do the trick.

Here is a version of “2+2” that works.

```
/O> 2 2 +
...

Stack: 4

/O>
```

Quackery successfully added two and two, and left the answer on the stack.

You may well be wondering by now; “Quackery”, “shell”, “duck’s head”, is there going to be much of this asinine anatine punnery? Well... there is going to be some. But not much. I mostly got over that a few months into development.

We have seen examples of words, which are any sequence of printable characters, some of which Quackery recognises. To see the full list, enter the word `words` into the shell, and it will display them all. (They are separated into two lists, the longer one has the names of Quackery instructions, like `+` and `echo`, and the shorter one is a list of builders, special words that are part of the Quackery compiler.)

And we have seen examples of numbers. Numbers in Quackery are whole numbers, and they can be as large as you like, as far as practical considerations such as time and computer memory permit. It is entirely possible to extend Quackery to include other sorts of numbers, but this is what you get straight out of the box. Quackery is not “batteries included” (Python PEP 206); its approach is more “Through the use of a few simple tools, my creativity will flourish.” (Julia Cameron, *The Artist’s Way*).

The third thing that Quackery has is a way of grouping things together to make composite objects. We make composite objects by enclosing them in brackets, thus:

```
[ 72 101 108 108 111 32 87 111 114 108 100 33 ]
```

Composite objects in Quackery are called nests, because they can be nested, like Russian nesting dolls. A nest can contain numbers, names and nests. (Internally, in the implementation of Quackery presented here, nests are Python lists, which are, technically speaking, dynamic arrays, but that’s by the by. They’re an ordered sequence of things.) This nest contains a sequence of twelve numbers.

If we were to enter this nest into the shell, it would put each of those numbers on the stack, one after the other, which is probably not entirely useful. It may be useful in some particular circumstance, and it is a valid Quackery program, but it is hard to imagine such a circumstance.

However, if we precede it with an apostrophe, or ‘single quote’, thus:

```
/O> ' [ 72 101 108 108 111 32 87 111 114 108 100 33 ]  
...  
Stack: [ 72 101 108 108 111 32 87 111 114 108 100 33 ]
```

then the nest appears on the stack as a single object. Next, we will enter the word `echo$` (the peso sign or dollar sign, \$, is pronounced “string” in Quackery.)

```
/O> echo$  
...  
Hello World!  
Stack empty.
```

`echo$` interpreted that nest of numbers as a string of characters and echoed it to the screen. Quackery said Hello World! (Again!)

Quackery represents a string of text as a nest of numbers. In the same way that it only understands whole numbers, Quackery only understands a limited range of characters. This is covered in more detail later. For now it is sufficient to know that if a character appears to turn into a question mark, it is because it is outside the range of characters that Quackery understands.

Typing in nests of numbers is not the most convenient way of dealing with text. Luckily there is a builder that knows about strings. It is `$`. `$` understands strings, and converts them into quoted nests of numbers. So we can write `$ "Hello World!" echo$` and that too will echo Hello World!.

The quotation marks " at either end of the text are called the “delimiters”. A delimiter cannot occur inside a string. If you want to make a string with quote marks in it, you would need to use a different delimiter. Quote marks and apostrophes are the most sensible choices for delimiters, but any character that Quackery understands will work equally well, as long as the same delimiter occurs at each end of the string. (The word `cr` is an instruction to print a carriage return - i.e. start a new line.)

```
/O> $ /Hello World!/ echo$ cr
... $ zHello World!z echo$ cr
...
Hello World!
Hello World!

Stack empty.
```

The next stage is to extend Quackery to include a word that echoes `Hello World!` to the screen, followed by a carriage return. We do this by putting the sequence of instructions into a nest with the aid of `[` and `]`, to make it a single item, and then give that item a name, using `is`.

```
/O> [ $ "Hello World!" echo$ cr ] is hello
... hello hello
...
Hello World!
Hello World!

Stack empty.
```

So far we have seen some numbers, four regular Quackery words and five Quackery builders, but didn't really explain how they worked. The regular words were `+`, `echo$`, `cr` and `'`.

- `+` added the two numbers that preceded it and left the result on “the stack”.
- `'` (“quote”) put the thing that followed it, a “nest”, on “the stack”.
- `echo$` printed a string that was on “the stack”. We saw that a string is a “nest” of numbers.
- `cr` started a new line of text on the screen. (“Printed a carriage return.”)

The five builders were `[`, `]`, `$`, `is`, and `say`.

- `[]` worked together to enclose a sequence of things and turned them into a nest.
- `$` took the string that came after it and turned it into a nest.
- `is` took the thing that preceded it, and gave it a name. The name followed `is`.
- `say` took the string that followed it and printed it.

We will discuss the stack at length in a while, but for now it's “the place where things go”, because everything has to be somewhere. If Quackery were a workshop, the stack would be the workbench where you put stuff in order to work on it.

The Quackery compiler, which is called `build`, is straightforward; it recognises regular Quackery words and numbers and lays them down one after another just as a real world bricklayer would lay down a row of bricks. When it encounters a builder word, that's its cue to call in a specialist, a “builder”. Builders extend what the compiler is capable of, to add new and useful tools, like nests and strings, or to gussy up the language by add some decorative trim. This is known as syntactic sugar.

Quackery has a syntactically sugared version of `$ "Hello World!" echo$`. It is the builder called `say` that started this chapter.

How builders work will be described later. First there are some basics to cover, starting with “why `2 2 +`, and not `2+2`”?

Quackery is Forwards Lisp

“What is three times four plus five times six?”

This is how we learn arithmetic at school, and how it is expressed in most computer languages. $3*4+5*6$. Assuming that the language conforms to the “order of operations” convention (i.e. BODMAS, PEMDAS, BEDMAS, or BIDMAS, depending on where you live) a conventional language would do the multiplications first, then the addition. For clarity, one might write $(3*4)+(5*6)$. This is infix notation, which requires knowing the order of operations to decipher (“parse”) it and requires parentheses for some expressions, or to make expressions that don’t strictly require parentheses simpler to understand. Parsing it requires you to jump around the expression, working out in what order to do things.

“What is the sum of the product of three and four and the product of five and six?”

This is the same question, with a more academic ring to it. If you say it out loud you can almost hear a dusty professor peering quizzically over half-moon spectacles. In a parenthesised prefix language such as Lisp it would be written $(+ (* 3 4) (* 5 6))$. Once you are familiar with the notation, parsing it requires finding the deepest nested parentheses and working outwards and backwards, from right to left, again doing the multiplications first, then the addition. It is consistent and logical, but written backwards if you are used to reading text from left to right.

“Take three and four and multiply them. Take five and six and multiply them. Add the two results.”

This time the expression is not a question, it is a step-by-step sequence of instructions. There is no ambiguity, no complicated parsing, just follow the steps one after the other from left to right and get the result. In a postfix language such as Quackery it would be written $3\ 4\ *\ 5\ 6\ *\ +$.

If you worked out the answer to the questions “What is three times four plus five times six?” or “What is the sum of the product of three and four and the product of five and six?” when you saw them, you effectively parsed them into the steps “Take three and four and multiply them. Take five and six and multiply them. Add the two results.” When a compiler for a mainstream procedural language or a Lisp style functional language parses an arithmetic expression, it does the same. Compilers make use of a data structure known as a stack to do this, which is not directly available to the programmer.

Stack based languages such as Quackery assume that you are at least as smart as the computer you are programming and capable of exercising this skill of parsing arithmetic expressions, while making the stack available to you. This means that various optimisations can be applied which are equivalent to the idea of using register optimisations when coding in an assembly language for a register based processor.

These will be covered in the section Stackrobatics, which explores the use of stack management words in Quackery, but first we will look at what a stack is, the various Quackery stacks, and why Quackery has no variables, in the section “The Quackery Stack, More Stacks, and The Other Stack”.

The Quackery Stack, More Stacks, and The Other Stack

1. The Quackery Stack

The stack is, allegedly, the most confusing aspect of stack-based languages. If you had a stacking toy as a toddler, you know what a stack is. If you have played the Towers of Hanoi puzzle game, you have some expertise in dealing with stacks. It's a "last in, first out" data structure. The last ring you put on the stacking toy is the first ring you take off.

The sum from the previous section was $3\ 4\ *\ 5\ 6\ *\ +$. Let's watch Quackery process it. We will start at the left and work our way to the right. Open the terminal app and get the quackery shell up and running. Type in each of the numbers and instructions one at a time, pressing enter twice each time.

```
/O> 3
```

```
...
```

```
Stack: 3
```

```
/O> 4
```

```
...
```

```
Stack: 3 4
```

```
/O> *
```

```
...
```

```
Stack: 12
```

```
/O> 5
```

```
...
```

```
Stack: 12 5
```

```
/O> 6
```

```
...
```

```
Stack: 12 5 6
```

```
/O> *
```

```
...
```

```
Stack: 12 30
```

```
/O> +
```

```
...
```

```
Stack: 42
```

To summarise, the numbers 3 and 4 appear on the stack, which builds up from left to right across the screen. $*$ takes the 3 and 4 off the stack and replaces them with 12, which is $3*4$. (This is standard behaviour for Quackery – we say that words “consume their arguments”.) 5 and 6 go on the stack above the 12, and again $*$ multiplies them, replacing them with the result, 30. Finally $+$ adds the top two items on the stack, 12 and 30, and replaces them with the result.

Things other than numbers can go on the Quackery stack, as we saw in Quackery Says Hello, where there was a string on the stack. We'll see this again in Dealing With Quackery. For now, numbers will suffice.

2. More Stacks

Quackery has no variables. It does, however, have ancillary stacks, which can be used like variables.

To illustrate their use as local variables, we will consider a simple problem.

As an avid box collector specialising in cuboid boxes, you like to maintain some statistics in your catalogue of interesting boxes. For each box, you note the length, width and height of each box in centimetres, then calculate their volume, surface area, and the length of all their edges, and note that down too. The calculations are rather laborious, so you have decided to make some Quackery words to perform the arithmetic for you.

Volume is easy enough. Say the dimensions of a box are 3, 4, and 5, then entering `3 4 5 volume` should return the answer 60. ($=3*4*5$.) First we'll put 3 4 5 on the stack, then do a `*` and see what happens.

```
/O> 3 4 5
... *
...
Stack: 3 20
```

As we might have expected, `*` has multiplied the top two items on the stack and replaced them with the result. A second multiplication should multiply that result by 3, giving a final result of 60.

```
Stack: 3 20
/O> *
...
Stack: 60
```

We have enough now to define a word called `volume`, using `[` and `]` to group the two `*`s into a nest, and `is` to give the nest a name. As diligent Quackery programmers we will also write a “stack comment” for our new word, to remind ourselves what our word needs to find on the stack, and what it will replace those stack items with. Comments can be entered using the builders `(` and `)`, which ignore what is between them.

First we will make sure the stack is empty with the word `empty`, which empties the stack. It's a useful tool during development, when the stack can end up littered with all manner of detritus, in this instance the number 60.

```
/O> empty
... [ * * ] is volume ( length width height --> volume )
... 3 4 5 volume
...
Stack: 60
```

Calculating the length of the edges is similar. It's four times the sum of the length, width, and height.

```
/O> empty
... [ + + 4 * ] is edges ( length width height --> length-of-edges )
... 3 4 5 volume
...
Stack: 60
```

This approach is not going to work for calculating the surface area, as we need to use the length, the width and the height of the box more than once, and these numbers are removed from the stack as soon as we do some arithmetic on them. We could use ancillary stacks here. As we will find out, it's not the best approach, but it is a good way to introduce ancillary stacks, so here we go.

An ancillary stack can be created using `[stack] is name-of-stack`. We will create three, called `length`, `width`, and `height`.

```
[ stack ] is length
[ stack ] is width
[ stack ] is height
```

The word `put` will put a number on the stack onto an ancillary stack. So if we enter `5 height put`, the 5 will be moved from the stack to the ancillary stack called `height`.

`share` puts a copy of the topmost item of an ancillary stack onto the stack.

`release` removes the topmost item of an ancillary stack.

There are some other ancillary stack words, which are covered later, but these are sufficient for now.

The surface area of a box of is $((length*width)+(length*height)+(height*width))*2$. So we can define `surface` as

```
[ height put
  width put
  length put          ( note a )
  length share  width share *
  length share  height share *
  height share  width share * ( note b )
  + + 2 *          ( note c )
  height release          ( note d )
  width release
  length release ] is surface ( height width length --> surface-area )
```

note a, at this point we have moved all three parameters to their respective ancillary stacks.

note b, now the stack contains `length*width length*height height*width`

note c, now the stack contains the surface area of our box.

note d, it is important to tidy up ancillary stacks when you are finished with them.

We should test code as soon as it is written. (The surface area of our 3 4 5 box is 94 cubic centimetres.)

```
/O> 3 4 5 area
...
```

```
Stack: 94
```

`area` is remarkably wordy compared to `edges` and `volume`, and as noted, we will get to a better way to define it, but for now we will continue with this approach, mostly in order to reinforce the point that ancillary stacks are stacks, not variables. It is also worth noting that sometimes an ancillary stack is the right approach, and the occasional use of an ancillary stack is fine. In fact there is one ready defined, called `temp`, which some system words use but is fine for general use as a local variable, as long as you tidy up after yourself.

Other system ancillary stacks include **base**, which is used to specify the base for reading and displaying numbers during input and output. It has a default value of **10**, but can be temporarily overridden with **put** and **release**. It is not a local variable, a better description would be an overridable global constant. To give an ancillary stack a default value, include it in the definition of the ancillary stack, thus:

```
[ stack 10 ] is base
```

(Useful tip: to see what is on an ancillary stack during development, enter the name of the stack and the word **copy** in the shell.)

```
/O> 11 temp put
... 22 temp put
... 33 temp put
... temp copy
...
```

```
Stack: [ stack 11 22 33 ]
```

```
/O> empty
... temp release
... temp release
... temp release
... temp copy
...
```

```
Stack: [ stack ]
```

What we are seeing here is that ancillary stacks store their contents within their own definitions, and that ancillary stack words like **put** and **release** modify those definitions. This makes ancillary stack words potentially dangerous, as they can be abused to modify definitions that should not be modified. Quackery does not prevent this – it is as permissive a language as it reasonably can be, but if you’re going to practice mischief like that, expect consequences.

Also expect consequences from tampering with system ancillary stacks with names like **with.hold** or **sort.test** – the naming convention suggests that the stacks are specific to particular words (**with** and **sort** respectively), and using them outside of that context is generally A Bad Idea, Don’t Do It. Usually this is because the system is using these ancillary stacks to convey information around while keeping the stack uncluttered for the programmer to use.

Continuing with our box collector example, after using the words **volume**, **edges**, and **area** for a while you decide it would require less typing if you combine them into one word – **stats**. To improve it a tiny bit we will introduce one more ancillary stack word, **take**, which is the opposite of **put**; it removes an item from an ancillary stack and puts it on the stack. So **temp take** is equivalent to **temp share temp release**. This allows us to tidy-up as we go along, so to speak.

```
[ height put    width put    length put
  height share  width share  length share  edges
  height share  width share  length share  surface
  height take   width take   length take   volume ] is stats
```

```
( height width length --> length-of-edges surface-area volume )
```

```
/O> 3 4 5 stats
...
```

```
Stack: 48 94 60
```

3. The Other Stack

Before we continue with our Dimensions of a Box problem, we'll look briefly at the *other* stack.

The overwhelming majority of processors have hardware support for a stack, often referred to as “the stack”, which would be somewhat confusing in Quackery, where “the stack” refers to the Quackery stack, as described above. So it will always be referred to as the “return stack” in this document. It allows the Quackery processor to keep track of nesting as it traverses a program.

Nesting comes in two forms, implicit and explicit. In the nest named **stats**, defined on the previous page, every word in that nest is the name of a nest. This is implicit nesting. Explicit nesting is where a nest is visibly embedded inside another nest, thus: (here, **a**, **b**, **c**, et cetera are arbitrary Quackery items; names, numbers, or nests).

```
[ a b [ c d ] e f ]
```

This is a nest that contains five items, **a**, **b**, the nest **[c d]**, **e**, and **f**. In order to traverse that nest from left to right it first does **a**, then **b**, then the nest **[c d]**, then **e**, then **f**. The processor maintains a number indicating which of the five items it is currently processing, and a pointer indicating which nest it is currently processing. In order to process a nested nest, such as **[c d]** (or an implicitly nested nest – any of **a**, **b**, **c**, **d**, **e**, or **f** may be named nests) it copies the item number and nest pointer to the return stack, and retrieves them once it has finished traversing the nested nest.

This differs from the majority of other processors, which typically maintain a single pointer to the current address being processed. Another difference is that the Quackery processor is jealously protective of its return stack, and provides only limited ways of modifying its contents, giving Quackery structured control flow at the virtual hardware level.

The final difference is that the return stack is only used for control flow purposes. “Stack frames” – a feature of the majority of high level languages, which are used to transfer arguments to functions, return results and store local variables – are not required in Quackery as these functionalities have been factored out into the Quackery stack and the ancillary stacks, where they are accessible to the programmer.

It should be noted that these design choices are a source of inefficiency in the implementation of the virtual Quackery processor presented here, which emphasises simplicity and legibility over speed of execution. However, in a theoretical hardware implementation, or in a virtual machine coded in a more pedal-to-the-metal language, not having to maintain stack frames makes for a language which need not discourage extremely short or deeply nested function definitions for the sake of efficiency.

For further reading on Stack Processors I recommend “Stack Computers: the new wave” by Philip J. Koopman, Jr., https://users.ece.cmu.edu/~koopman/stack_computers/

In my lay understanding of Stack Processors I envisage a possible hardware Quackery processor as a 64 bit machine, with 60 data bits (and consequently a 60 bit address space) and 4 metadata bits. The metadata would encode four types of data in two bits; pointer to nest (i.e. Jump to Subroutine), pointer to bignum nest, bignum nest, and operator (machine code instruction.) One bit would flag the end of a nest (i.e. Return from Subroutine) or bignum nest, and one bit would be reserved for garbage collection.

(Bignum nests are not covered in this document. Python successfully hides the implementation of its bignums from the programmer, so the details are not necessary for understanding Quackery. I use the term “nest” here to suggest they are somewhat similar to Quackery nests.)

Stack Comments

This document and its accompanying files follow the convention of providing stack comments for Quackery words. Although not compulsory you are encouraged to adopt this convention as a minimum requirement for commenting words.

They take the form (*before-stack* --> *after-stack*) where *before-stack* indicates the stack items that are consumed by the word, listed with the top of stack at the right, and *after-stack* indicates the stack items returned by the word, in the same order.

The stack items are represented in various ways.

Where they are discussed in accompanying text, the letters a, b, c, and so on are used.

In the program listings provided, single characters are used to indicate the item types consumed and returned by the word. (Terms not previously described will be described in due course.)

```
x -- any item

n -- a number
b -- a number being used as a boolean
f -- a number being used as an ordered set of booleans (f = flags)
c -- a number being used as a character

[ -- a nest
$ -- a nest being used as a string
s -- a nest being used as an ancillary stack

* -- a variable number of stack items
```

A few words have take or return a variable number of stack items. Generally words should take a fixed number of items. Words that take a variable number should be defined sparingly and only when justified. Examples of predefined words in this category include **pack** and **unpack**, described below.

Other words have variable stack effects because they “do” other words, nest or numbers. “doing” a word will be described in due course. The stack comments of these words assume that the thing being “done” has no effect on the stack. (-->). Often this will not be the case.

The accompanying PDF crib-sheet, “Quackery Quick Reference” uses a mix and match approach to stack comments to try to cram as much useful information as possible into a single page.

When it is meaningful, the items listed in a stack comment should use names that are context specific, just as in a conventional language one should give helpful names to a function’s arguments.

As should be apparent by now, stack commenting is a bare-bones convention that only covers the simplest of cases (which should be the overwhelming majority of your code), and is not an alternative to properly documenting code for the benefit of others, and for yourself when you return to it having forgotten the details of how the code works.

Stackrobatics

Returning to the Dimensions of a Box problem, we could conclude that they work and are therefore not in need of fixing, but there is something of a discrepancy between the words **volume** and **edges**, which are remarkably short, and the words **surface** and **stats**, which are noticeably lengthier and use ancillary stacks.

Is it possible to shorten these two words and make them more efficient at the same time? Yes. It requires learning a skill that is peculiar to stack based languages; stack management or stack optimisation, jocularly referred to as stackrobatics or stack juggling.

In a register based processor it is a sensible optimisation to try to use the very fast memory within the processor – the registers – as much as possible, rather than moving data back and forth between the processor and RAM. Similarly, in stack based languages it is good practice to avoid excessive use of ancillary stacks or their equivalent, and favour the exclusive use of the stack as far as is reasonable.

To this end, Quackery provides a collection of stack management words, which are listed in the upper left of the handy Quackery Quick Reference PDF, the one page crib sheet that comes with Quackery.

```
dup ( a --> a a )
drop ( a --> )
swap ( a b --> b a )
rot ( a b c --> b c a )
unrot ( a b c --> c a b )
over ( a b --> a b a )
nip ( a b --> b )
tuck ( a b --> b a b )
2dup ( a b --> a b a b )
2drop ( a b --> )
2swap ( a b c d --> c d a b )
2over ( a b c d --> a b c d a b )
pack ( a b 2 --> [ a b ] )
unpack ( [ a b ] --> a b )
dip ** ( a b c --> a**b c )
```

Spend some time experimenting with these in the shell. If you like, leave out **pack**, **unpack** and **dip** for the moment, they will be explained presently. Put a bunch of numbers on the stack, enter one of the stack management words and see what happens. Can you replicate the behaviour of one of the words by using two or more of the other words? (As a reminder, **empty** will empty the stack, if you want to tidy up while you are playing.)

We can categorise them as duplicating, disposal, reordering, composite, and extending.

The duplicating words, **dup** and **2dup** circumvent the “words consume their arguments” principle. They are the “we will need this again” words.

The disposal words, **drop** and **2drop** do the opposite. They are the “we no longer need this” words. One common use is after a looping structure. Jumping ahead to introduce one of the looping words, **times** does the thing that follows it a specified number of times. **example** will echo the string on the top of the stack to the screen five times, keeping the the string on the stack using **dup**, and then disposing of it afterwards using **drop**.

```
/O> [ 5 times [ dup echo$ ] drop ] is example ( $ --> )
... $ "Hello " example
...
Hello Hello Hello Hello Hello
Stack empty.
```

The reordering words, **swap**, **rot**, **unrot**, and **2swap** serve a couple of purposes.

Words consume stack items and return results in a specific order. If they consume more than one item, occasionally the order is unimportant, as with **+** and *****, but usually it matters, and here is a coding tip; when defining a word, if there is a natural reading for the order of arguments, consider using that first; it will make the word usage a lot easier to remember. A good example is **/mod**, which divides one number by another and returns the result of the division, and the remainder. “a divided by b is c remainder d”, e.g. “twenty divided by six is three remainder two”.

```
/O> 20 6 /mod
...
Stack: 3 2
```

You may well need to reorder stack items before a word to fit the word’s requirements.

Or you might need a result that was computed earlier and has become buried under one or two stack items, or to bury a stack item until it is required. And here is another coding tip; sometimes the order in which you compute things is not important. If you can use a stack item when it is at or near the top of the stack, do so. It helps keep the stackrobatics to a minimum. So if you find a definition is needing a lot of stack management words, consider the order in which things need to be done, and where the order of operation is flexible. Try some variations and go with the one that requires the least stack juggling.

The composite words, **over**, **nip**, **tuck**, and **2over**, combine elements of duplication or disposal with reordering. But let’s not get hung up on this impromptu taxonomy of stack management words. Mostly it’s just a way of saying “think about what you want to achieve and if it’s getting messy try a few different combos to see what works best for a given situation.” The composite words acknowledge that sometimes two or even three stack management words are needed in a row, and address some of the more commonly occurring situations. (Parenthetically, the words thus far are the core words in many implementations of the Forth programming language, so there is a basis to claiming that they are a reasonable choice for a set of stack management tools.) Four stack management words in a row should raise a red flag, telling you that it is very likely that there is a better approach than the one you have chosen. Start by breaking the definition down into subtasks and coding each subtask separately. (This is called factoring. A well factored program has lots of short definitions, each of which does a single, well defined task, and each of which can be reused elsewhere. This is the ideal situation. In practice, not everything factors well, but mostly it does. Don’t lean on this observation as an excuse for poor factoring!

Here I have to put my hands up and admit that the portion of Quackery that is coded in Quackery sometimes falls short of these ideals. Some of the definitions, in particular **build** and **unbuild**, are undeniably long by Quackery standards, and on a couple of occasions there are four stack management words in a row. In my defence I note that the longer words were developed as a set of well factored shorter words, for ease of development, then combined into one long definition as the factored out portions did not satisfy the reusability criterion, and “not cluttering up the dictionary with words that did not serve a useful purpose” was very high on the list of criteria. With regard to overly complicated stack juggling; where this occurs, a significant amount of time was spent trying to simplify the code before I conceded that maybe this was one of those rare cases where it was justified.

The extending words, **dip**, **pack** and **unpack**, are not lifted from the Forth lexicon. **dip** has a precedent in the concatenative language Joy, and **pack** and **unpack** to some extent stand in for the Forth words **pick** and **roll**. They are the “extending” words because among their uses they allow one to reach further down into the stack than the words mentioned so far.

dip temporarily removes the top item from the stack and returns it to the stack after the item that follows it in the code has been performed. It is equivalent to **temp put ... temp take**, except it has its own ancillary stack; it doesn't use **temp**.

```
/O> empty
... 1 2 3
... dip dup
...

Stack: 1 2 2 3
```

```
/O> empty
... 4 5 6
... dip [ over + ]
...

Stack: 4 9 6
```

As the second example illustrates, the thing that follows it can be a nest, and it is not restricted to stack management words. Note that “double dipping” requires the second **dip** to be in a nest.

```
/O> empty
... 7 8 9 0
... dip [ dip + ]
...

Stack: 15 9 0
```

dip can mean “I won't need this for ages, so let's shove it well out of the way for now.” Consequently the nests that follow it can be quite long. There are instances of this in the Quackery source code.

pack and **unpack** are the “there's just too much happening on the stack” words. **pack** takes a number of items off the stack, puts them into a nest and puts the nest on the stack, reducing that number of stack items to a single item. **unpack** reverses the operation of **pack**.

```
/O> 5 6 7 8 9
... 4 pack
...

Stack: 5 [ 6 7 8 9 ]

/O> unpack
...

Stack: 5 6 7 8 9
```

With our “Dimensions of a Box” problem in mind, we might guess that **3dup** could be a useful addition to the set of stack management words, and define it using **dip**, **pack**, and **unpack**.

```
/O> [ 3 pack
...   dup
...   dip unpack
...   unpack ] is 3dup ( a b c --> a b c a b c )
... empty 1 2 3
... 3dup
...

Stack: 1 2 3 1 2 3
```

In practice we will see that `3dup` is not required, but we can use the central idea of it, so defining it was a worthwhile exercise.

Returning to calculating the surface area of a box, we note that the order of arguments on the stack is not significant, as only addition and multiplication are required, and there is a symmetry to the formula. Taking width, length and height as `a`, `b`, and `c`, the formula is;

$$((a*b)+(a*c)+(b*c))*2$$

which can be simplified to

$$((a*(b+c))+(b*c))*2$$

This reduces the number of arithmetic operations by one, which will make our task a little simpler.

Let's put some numbers on the stack and start working the problem. 5, 6, and 7 can be `a`, `b`, and `c` respectively. `2dup *` will calculate `b*c` (which equals 42) and leave `b` and `c` on the stack for further use.

```
/O> 5 6 7
... 2dup *
...
Stack: 5 6 7 42
```

Now we can `dip` the 42 out of the way and work on `a*(b+c)` by adding `b` and `c`, then multiplying by `a`, (which equals 65).

```
/O> dip [ + * ]
...
Stack: 65 42
```

Finally we add `a*(b+c)` to `b*c` and double the result.

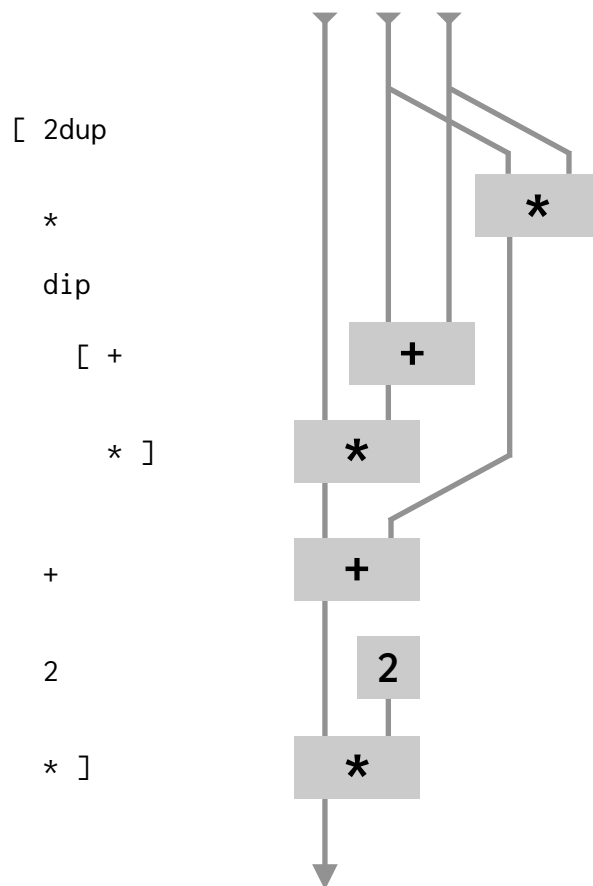
```
/O> + 2 *
...
Stack: 214
```

That works, so let's put it all together in a nest, give it a name, and test it;

```
/O> empty
... [ 2dup *
...   dip [ + * ]
...   + 2 * ] is surface ( a b c --> s )
... 5 6 7 surface
...
Stack: 214
```

I don't know, but it is possible that if you are new to the world of stack processors you are thinking "Oh, that's supposed to be 'easy', is it?", and have that uneasy feeling that I am looking at you askance, sympathetically, like you just admitted you were baffled by doorknobs. It is true that old hands can sometimes forget how tricky it seems at first, but my experience was that it is much like learning to ride a bicycle; seemingly impossible until suddenly it 'clicks' and you're sailing along, perhaps not ready to enter the Olympics, but definitely moving away from perilous wobbling and barked elbows at a respectable speed.

One way of thinking about the stack that may help is with structured data flow diagrams, where the stack builds up from left to right, as with stack comments.



This raises a fairly deep point, that the stack management words are more than just optimisations, they embody the fact that Quackery, and stack based languages in general, have both structured control flow and structured data flow.

Moving on, let's look again at **stats**. Having defined **3dup**, let's use it. We'll put 1 2 3 on the stack for our height, width and length and, as we will need those numbers three times, we'll **3dup** them twice.

```
/O> 1 2 3 3dup 3dup
```

```
...
```

```
Stack: 1 2 3 1 2 3 1 2 3
```

Then we need to use **edges**, **surface**, and **volume** to compute the results, **dipping** each result out of the way as we go.

```
/O> edges dip [ surface [ dip volume ] ]
```

```
...
```

```
Stack: 6 22 24
```

This works but the results are in reverse order. We could fix that and define **stats** as;

```
[ 3dup 3dup volume
  dip
  [ surface
    dip edges ] ] is stats ( a b c --> e s v )
```

We *could* define it like that, but somehow the definition didn't sit right with me. Mostly because I know how **3dup** works, and there's unnecessary **packing** and **unpacking** going on. After a little experimentation, I settled on;

```
[ 3 pack
  dup  unpack edges
  over unpack surface
  rot  unpack volume ] is stats ( a b c --> e s v )
```

Walk through the code to see how it works, and convince yourself that it does work. Then decide for yourself which version you prefer, or see if you can devise a version that you like better. 'Best' without context is subjective.

At the moment, if you recall, the context of this exercise is "box collector cataloging small boxes", so realistically, "first version that works" is best. No optimisation is required.

If we change the context to "mathematician working with extremely large numbers" then optimisation becomes necessary. With very large numbers, multiplication can take a very long time.


Just for fun, try it. Define `[dup dup * *] is cubed`, then enter `9 cubed`. When the result comes back (almost instantaneously), enter `cubed` again to cube the result, then enter `cubed` again to cube that result, and keep doing it until you're twiddling your thumbs waiting for the next result. What did it take until you were getting bored? Possibly around ten to twelve steps, depending on your machine and your patience. For some areas of mathematics these numbers are small potatoes.

Note that this slow-down is not a problem caused by Quackery running on a virtual stack processor, or even a problem caused by Python running on a virtual stack processor; the multiplication routine is coded in hand-optimised C, which compiles to native machine code. Multiplying two very large numbers is inherently slow. There are clever algorithms that ameliorate the problem for humongously large numbers, but even those are just stopgaps.

Assuming the numbers are sufficiently large to justify spending an evening treating the "dimensions of a box" problem like a Sudoku puzzle, i.e. a mental exercise pastime, I came up with this final version of **stats** that minimises the number of multiplications. (1 << and 2 << are equivalent to 2 * and 4 * respectively, but a little faster.) The comments show the state of the stack at the end of each line on the left, with the state of the ancillary stack **temp** on the right hand side of the comment.

(a b c	-->	edges	surface	volume)
((a+b+c)*4	((a*(b+c))+(b*c))*2	a*b*c)
[2dup *		(a b c b*c)
temp put		(a b c		b*c)
+		(a b+c		b*c)
2dup *		(a b+c a*(b+c)		b*c)
temp share		(a b+c a*(b+c) b*c		b*c)
+ 1 <<		(a b+c surface		b*c)
unrot		(surface a b+c		b*c)
over + 2 <<		(surface a edges		b*c)
unrot		(edges surface a		b*c)
temp take		(edges surface a b*c)
*]	is stats	(--> edges surface volume)

Quackery Control Flow

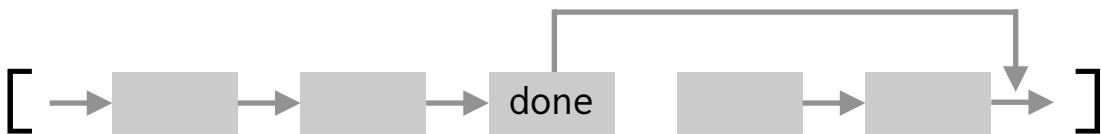
 this is an item, i.e. a number, a word, or a nest



usually, proceed from left to right



again means “jump to the start of the nest”



done means “jump to the end of the nest”



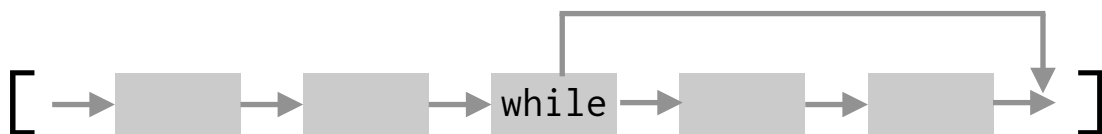
if means “skip over the next item unless the top of stack is **true** (not 0)”



iff means “skip over the next two items unless the top of stack is **true**”



else means “skip over the next item”



while means “jump to the end of the nest unless the ToS is **true**”



until means “jump to the start of the nest unless the ToS is **true**”

There are no syntax rules for the control flow words. Mix and match them to make useful combos.

Dealing With Quackery

This is an illustration of string and nest processing in Quackery. The file `cards.qky` in the folder `sundry` contains the code presented here. It can be added to Quackery by entering `$ 'sundry/cards.qky' loadfile` in the shell. The goal is to extend Quackery so that code like

```
newpack +jokers 3 riffles 4 players 7 deal
```

constitutes a meaningful Quackery program. (This program would create a pack of cards with two jokers, give it three riffle shuffles and deal four players seven cards each.)

Before we start coding the set of card handling words, it will turn out that a division word that rounds up rather than down will be useful here, so we'll define that first and call it `/up`. `/mod` returns the whole number result of division and the remainder, as with primary school maths. (i.e. "ten divided by three equals three remainder one.") `if 1+` adds one to the result if the remainder is not zero.

```
[ /mod if 1+ ] is /up ( n n --> n )
```

A pack of cards is represented by a nest of the numbers 0 to 51. "`[]`" puts an empty nest on the stack. The word `times` does the next item the number of times specified by the number on the top of the stack. In this example, `i^` will return 0 the first time `times` does `[i^ join]`, it will return 1 the second time, 2 the third time, and so on up to 51. `newpack` returns a nest of the numbers 0 to 51.

```
[ [] 52 times [ i^ join ] ] is newpack ( --> [ ] )
```

Games sometimes include one or both jokers, one of which comes before all the other cards, the other of which comes after. So we will define three words which add a low joker, a high joker and both jokers to the pack respectively.

```
[ -13 swap join ] is +lowjoker ( --> [ ] )
```

```
[ 64 join ] is +highjoker ( [ --> [ ] )
```

```
[ +lowjoker +highjoker ] is +jokers ( [ --> [ ] )
```

The numbers -13 and 64 are not arbitrary. Later we will define a word `rankfirst` where this is explained. For now it is sufficient to note that -13 is less than 0, and 64 is more than 51, and to know that their positions as first and last card will not be affected however the cards are sorted.

The rank of a card is represented by a number in the range 0 to 12. `rank` returns a string corresponding to a card's rank. The reason for the `do` at the end of the definition is covered in the description of `$` in the Building Words section of Word Behaviours.

`table` turns a nest into a lookup table, just as `stack` turns a nest into an ancillary stack. A `table` will take a number from the stack and return an item in the table corresponding to that number. Here the string "ace" is the zeroth item in the table, and "king" is the twelfth.

```
[ [ table
  $ "ace"    $ "two"    $ "three"
  $ "four"   $ "five"   $ "six"
  $ "seven"  $ "eight"  $ "nine"
  $ "ten"    $ "jack"   $ "queen"
  $ "king" ] do ] is rank ( n --> $ )
```


The suit of a card is represented by a number in the range 0 to 3. `suit` returns a string corresponding to a card's suit.

```
[ [ table
  $ "clubs" $ "diamonds"
  $ "hearts" $ "spades" ]
do ] is suit ( n --> $ )
```

A card number (other than the jokers, which have neither suit nor rank) is composed from a suit number and a rank number by multiplying the suit number by 13 and adding the rank number (this is “suit first encoding”), so that applying `13 /mod` to a card number will return the rank number of the card on the ToS, and the suit number as 2oS. This means that the nest created by `cards` represents a pack of playing cards sorted first by suit and then by rank, i.e. first the clubs, ace to king, then the diamonds, ace to king, then the hearts, ace to king, and finally the spades., ace to king.

If we had chosen to represent a card by multiplying the rank number by 4 and adding the suit number, (“rank first encoding”) so that applying `4 /mod` to a card number would have returned the suit number of the card on the ToS and the rank number as 2oS, then the nest returned by `cards` would have represented a pack of playing cards sorted first by rank, and then by suit, i.e. first the aces, the Ace of Clubs, Ace of Diamonds, Ace of Hearts, Ace of Spades, then the twos and so on.

`card` takes a card number and returns the card's name as a string. It treats the jokers as special cases.

```
[ dup -13 = iff
  [ drop $ 'low joker' ] done
dup 64 = iff
  [ drop $ 'high joker' ] done
13 /mod rank
$ ' of ' join
swap suit join ] is card ( n --> $ )
```

A pack or hand of cards is represented by a nest of `cards`. `hand` returns a string of the names of each of the cards in a hand, separated by carriage returns.

```
[ [] swap
  witheach
  [ card join
    carriage join ] ] is hand ( [ --> $ )
```

`echocard` and `echohand` display a card and a hand of cards on the screen respectively.

```
[ card echo$ ] is echocard ( n --> )
[ hand echo$ ] is echohand ( [ --> )
```

The word `sort` sorts a nest of numbers into ascending order, so would sort a hand of cards into suit first order, but there is no built-in equivalent to sorting into rank first order. We will use the phrases `sortwith bysuit` and `sortwith byrank` to give more meaningful code.

The comparison `bysuit` can be used with `sortwith` to sort a hand of cards by suit and then by rank.

```
[ > ] is bysuit ( n n --> b )
```

rankfirst takes a card numbered with suit first encoding, and returns the same card if it were numbered with rank first encoding. See **suit**, above. If **rankfirst** is applied to a joker, the joker will retain its correct order in the sequence, as `-13 rankfirst` returns `-1` and `64 rankfirst` returns `52`.

```
[ 13 /mod 4 * + ] is rankfirst ( n --> n )
```

The comparison **byrank** can be used with the word **sortwith** to sort a hand by rank and then by suit.

```
[ rankfirst swap rankfirst < ] is byrank ( n n --> b )
```

cut splits a pack or hand at some **random** position within the nest, swaps the two halves and joins them back together. As **random** will crash if presented with an argument of `0` we check that the pack is not empty first. (Cutting a pack of zero cards is not entirely meaningful, but takes no time. Later we won't write defensive code for a similar error when dealing cards, despite the fact that dealing zero cards at a time would be very silly. The test for potential problems is impossibility, not meaningfulness.)

```
[ dup [] = if done
  dup size random split
  swap join ] is cut ( [ --> [ )
```

Scarne's Cut is named after its inventor, the master magician John Scarne, who devised it as a method of protecting US troops from card sharps during World War II. It consists of pulling a block of cards from the centre of the pack and putting them on the top before performing a regular cut. The first step can be repeated several times. We will use the phrases **scarne cut** or e.g. **3 times scarne cut** to perform a single Scarne cut or a repeated Scarne cut.

scarne splits the pack twice, initially in the first* two thirds of the pack, and then in the portion on the top of the stack. It swaps the top and middle and joins the three pieces. Note the use of **/up** to avoid passing a zero to **random**.

```
[ dup [] = if done
  dup size 2 * 3 /up
  random split
  dup size dup iff
    [ random split
      dip swap join ]
  else drop
  join ] is scarne ( [ --> [ )
```

*The first card in a nest is treated as the top of the pack, so splitting the pack puts the upper portion of the pack second on the stack, and the lower portion on the top of the stack.

players creates a nest of empty hands, one hand for each player. The word **of** creates a nest of a specified number of specified items. Imagine an army quartermaster saying "Boots, two of." before equipping a recruit with a pair of boots. Here **of** creates a nest of `n` empty nests, which serve as hands of cards. `[]` puts an empty nest i.e. "`[]`" on the stack, **nested** puts it inside a nest. i.e. "`[[]]`", and if `n` is 4, **of** will join four copies of this together; "`[[[]] [[]] [[]] [[]]]`".

```
[ [] nested swap of ] is players ( n --> [ )
```

`dealby (a b c d --> e f)` deals from a pack of cards, `a`, into a nest of player's hands `b`, returning the nest of hands, `f`, on ToS, and the talon (the undealt cards), `e`, underneath it on 2oS. It deals `d` cards at a time, and gives each player `d` cards, `c` times.

Using a small pack of ten cards to illustrate this, we will deal twice to each of three players, one card at a time, which will leave four cards in the talon.

```
/O> ' [ 0 1 2 3 4 5 6 7 8 9 ] ' [ [ ] [ ] [ ] ] 2 1 dealby
...
Stack: [ 6 7 8 9 ] [ [ 0 3 ] [ 1 4 ] [ 2 5 ] ]
```

If we try to deal more cards than there are in the pack, the first hands in the nest of hands will receive the extra cards, as would happen in a real life deal. Here we try to deal three cards to each of four players, again one card at a time, with the same short pack of ten cards.

```
/O> ' [ 0 1 2 3 4 5 6 7 8 9 ] ' [ [ ] [ ] [ ] [ ] ] 3 1 dealby
...
Stack: [ ] [ [ 0 4 8 ] [ 1 5 9 ] [ 2 6 ] [ 3 7 ] ]

[ temp put
  over size * times
    [ over [] = iff
      [ 1 split swap join ]
    else
      [ swap
        temp share split
        swap rot behead
        rot join
        nested join ] ]
  temp release ] is dealby ( [ [ n n --> [ [ ]
```

In order to better understand this code we will step through the code for `deal`, using the shell to see what is happening. For brevity we will use a short pack of 5 cards, dealing two cards, one at a time, to three players.

```
/O> ' [ 0 1 2 3 4 ] ' [ [ ] [ ] [ ] ] 2 1
... temp put
...
Stack: [ 0 1 2 3 4 ] [ [ ] [ ] [ ] ] 2
```

The first line just moves the number of cards to be dealt each time to the ancillary stack `temp`. We will need to refer to this number in the middle of the dealing loop, and it is convenient not to have it cluttering up the stack.

```
/O> over size *
...
Stack: [ 0 1 2 3 4 ] [ [ ] [ ] [ ] ] 6
```

`over size *` calculates the number of times we need to deal, or attempt to deal, a card from the pack into a player's hand. It may seem a little dimwitted to attempt to deal six cards when there are only five in the pack, but as we will see, it serves a purpose.

In order to step through the loop we will remove the 6 with `drop`, then skip to the first line within the `times` loop.

`over [] = iff` looks to see if the ToS (the pack) is empty, without changing the stack. At the moment the pack is not empty, so we will follow the `else` branch of the `iff`.

```
Stack: [ 0 1 2 3 4 ] [ [ ] [ ] [ ] ]

/O> [ swap temp share split
...   swap rot behead
...   rot join
...   nested join ]
...

Stack:[ 1 2 3 4 ] [ [ ] [ ] [ 0 ] ]
```

The first card in the pack, `0`, has moved from the pack into a player's hand. We could step through this line by line to see the mechanics of the move, but instead we will leave that as an exercise for the reader and see the next trip around the loop, noting that the pack is still not empty.

```
/O> [ swap temp share split
...   swap rot behead
...   rot join
...   nested join ]
...

Stack: [ 2 3 4 ] [ [ ] [ 0 ] [ 1 ] ]
```

The next three trips round the loop yield:

```
Stack: [ 3 4 ] [ [ 0 ] [ 1 ] [ 2 ] ]

Stack: [ 4 ] [ [ 1 ] [ 2 ] [ 0 3 ] ]

Stack: [ ] [ [ 2 ] [ 0 3 ] [ 1 4 ] ]
```

Each time round the loop, the first player's hand in the nest of hands (ToS) is moved from the front to the end of the nest, and the first card in the pack (2oS) is moved to the end of the player's hand. One might envisage the player's hands as arranged on a rotating turntable, with a robot dealer adding a card from the stack to a hand as it passes beneath it.

After five times around the loop, the pack is empty, and there is one more trip around the loop to perform, this time taking the first clause of the `iff`.

```
/O> [ 1 split swap join ]
...

Stack: [ ] [ [ 0 3 ] [ 1 4 ] [ 2 ] ]
```

This time the players in the nest of players have cycled without having any cards added, moving the first nest to the end, leaving them in the required order, i.e. with the players who received the extra cards at the front of the nest. This would correspond to the first hand in the nest being to the left of the dealer in a real life game, and is the reason for wanting to step through the loop six times rather than five.

In this example the talon (the remains of the pack, 2oS) is empty.

Mostly we will want to deal one card at a time, so `deal` gives a simplified notation for doing this.

```
[ 1 dealby ] is deal ( [ [ n --> [ [ ] ]
```

For example, to deal seven cards to each of four players from a new pack, one would write;

```
newpack 4 players 7 deal
```

A few games involve dealing two or more cards at a time. One example of this is Euchre, where five cards are dealt to each player, as packets of two and three cards. **dealeach** deals each player a specified number of cards.

```
[ 1 swap dealby ] is dealeach ( [ [ n --> [ [ )
```

For example, to deal two cards to each player, and then three cards to each player, one would write, (assuming there is a pack and a nest of hands on the stack);

```
2 dealeach 3 dealeach
```

The final dealing word we will define, **dealall**, simply deals all the cards in the pack, one at a time, to the players.

```
[ over size deal ] is dealall ( [ [ --> [ [ )
```

undeal reverses the action of dealing, taking the cards from each player one at a time and returning them to the pack so that the cards are in the same order as before **dealall**. It also removes the nest of players from the stack.

```
[ [ ] swap
  [ behead
    1 split
    dip [ swap dip join ]
    dup [ ] = iff
    drop
  else
    [ nested join ]
    dup [ ] = until ]
  drop swap join ] is undeal ( [ [ --> [ [ )
```

[] swap puts an empty nest under the nest of player's hands. At the end, **swap join** will append this to the pack of cards that **undealall** expects to find underneath the nest of players' hands.

The **until** loop nest removes the first player's hand from the nest of players' hands (**behead**), and the first card from that hand (**1 split**) and adds it to the nest placed underneath the nest of players' hands (**dip [swap dip join]**). It then checks if the player's hand is empty (**dup [] = iff**) and either discards the empty hand (**drop**) or adds it to the end of the nest of players' hands (**[nested join]**). It repeats the loop until all the players' hands have been discarded (**dup [] = until**).

Once out of the loop, the now empty nest of players hands is discarded (**drop**) and as noted previously, **swap join** adds the cards collected from the players' hands to any cards that were previously undealt.

Undealing cards is not a common practice in real life, but **undeal** can be used in combination with other words defined in this suite can provide a variety of ways of shuffling cards.

divvy takes a different approach to sharing out all the cards to the players. Rather than dealing the cards one or more at a time, it gives each player all their cards at once.

```
[ over size over size
  tuck /mod
  rot over -
  dip [ over 1+ swap of ]
  rot swap of
  join
  swap [] 2swap
  witheach
    [ split
      unrot nested join
      swap ]
  unrot
  witheach
    [ dip behead join
      nested join ] ] is divvy ( [ [ --> [ [ )
```

For example, if there are 52 cards in the pack and 5 players, each with an empty hand, the first two players will receive 11 cards each, and the remaining three will receive 10 cards each. `over size over size` puts the number of cards and the number of players on the stack whilst leaving the nest of cards and the nest of players' hands on the stack.

`tuck /mod rot over -` calculates the number of players to receive eleven cards and the number to receive ten. The number of players remains on the stack. (For the moment we will leave the nest of cards and the nest of players off the stack and focus on their sizes.)

```
/O> 52 10 tuck /mod rot over -
...
Stack: 10 2 3
```

`dip [over 1+ swap of]` and `rot swap of` use these numbers to construct two nests. `join` joins them together into a nest that describes how the cards should be distributed amongst the players.

```
/O> dip [ over 1+ swap of ]
... rot swap of
... join
...
Stack: [ 11 11 10 10 10 ]
```

At this point in the code the stack contains a nest of 52 cards, in order, a nest of five players' empty hands, and a description of how the cards should be divided amongst the players.

(In this example, the players' hands are empty at the start. This is not a prerequisite. Try stepping through the code after say dealing a couple of cards to each player with `newpack 2 dealeach` to see how it handles that.)

```
Stack: [ 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24
25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48
49 50 51 ] [ [ ] [ ] [ ] [ ] [ ] [ ] [ 11 11 10 10 10 ]
```

swap [] 2swap reorganises the stack and adds an empty nest into the mix.

```
Stack: [ [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14
15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38
39 40 41 42 43 44 45 46 47 48 49 50 51 ] [ 11 11 10 10 10 ]
```

Now the top two items on the stack are the pack of 52 cards and the description of how those cards should be distributed. `with each [split unrot nested join swap]` will distribute the cards into nests in the empty nest that is now third on stack, leaving the now depleted pack of cards on the top of the stack.

```
/O> with each
... [ split
...   unrot nested join
...   swap ]
...
```

```
Stack: [ [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ 0 1 2 3 4 5 6 7 8 9 10 ] [ 11 12 13 14
15 16 17 18 19 20 21 ] [ 22 23 24 25 26 27 28 29 30 31 ] [ 32 33 34 35 36
37 38 39 40 41 ] [ 42 43 44 45 46 47 48 49 50 51 ] ] [ ]
```

`with each` removes the nest that is on the ToS, then repeats the thing that follows once for each item in the nest it removed, putting an item from the nest it removed on the stack at the start of each repetition. So, with `[11 11 10 10 10]` on ToS, `with each [split unrot nested join swap]` is equivalent to

```
11 split unrot nested join swap
11 split unrot nested join swap
10 split unrot nested join swap
10 split unrot nested join swap
10 split unrot nested join swap
```

As previously, `unrot` prepares the stack for the upcoming `with each`, which joins the nest of divided up cards into the nest of players' hands, leaving the empty pack of cards underneath the nest of players' hands.

```
/O> unrot
... with each
... [ dip behead join
...   nested join ]
...
```

```
Stack: [ ] [ [ 0 1 2 3 4 5 6 7 8 9 10 ] [ 11 12 13 14 15 16 17 18 19 20 21
] [ 22 23 24 25 26 27 28 29 30 31 ] [ 32 33 34 35 36 37 38 39 40 41 ]
[ 42 43 44 45 46 47 48 49 50 51 ] ]
```

We can use `newpack`, `players` and `divvy` to create a pack of cards suitable for playing Euchre by stripping out some of the cards. Euchre packs have several variations. Here we will create a 25 card pack, which is Aces, nines, tens, Jacks, Queens, and Kings, and a high ranking joker.

euchrepack starts with a **newpack** and uses **4 players divvy** to separate it into a nest of nests of suits. It then uses a **witheach** clause to remove the twos to eights from each suit (**1 split 7 split nip**) and **join** the remaining cards together, and then **join** the suit back into the pack. Finally it adds a high joker to the pack.

```
[ newpack
  4 players divvy
  witheach
    [ 1 split 7 split
      nip join
      join ]
  +highjoker ]      is euchrepack ( --> [ ] )
```

gather gathers up the cards in a nest of players' hands and adds them to the pack. It reverses the action of **divvy** and removes the nest of players' hands from the stack.

```
[ [ ] swap witheach join
  swap join ]      is gather ( [ [ ] --> [ ] )
```

Shuffling a pack of cards is already provide for with the word **shuffle**. **shuffle** will thoroughly randomise a pack of cards, as far as a pseudo-random number generator is able to do this, which makes it somewhat unrealistic. The shuffling methods employed in real life card games do not lead to all 52 factorial possible reorderings of the pack by a long stretch. This has a material impact on game play, so we will code a few popular shuffles.

The weave or riffle shuffle, in its most exact incarnation where the pack is divided in two halves and cards are taken from each half one at a time, is known to mathematicians as the faro shuffle, and is of interest to card mechanics for one particular mathematical property, the ability to move the top card to a desired position within the pack by a number of faro shuffles. The faro shuffle comes in two flavours, the in shuffle and the out shuffle, depending on which half of the pack is selected from first.

faro-out does an out shuffle, **faro-in** does an in shuffle.

```
[ 2 players divvy undeal ] is faro-out ( [ ] --> [ ] )

[ 2 players divvy
  1 split swap join
  undeal ]      is faro-in ( [ ] --> [ ] )
```

faro takes two numbers in addition to a pack of cards (3oS), the 2oS is the number of in and/or out shuffles to be performed, and the ToS specifies how many cards down the uppermost card in the pack should have moved once the specified number shuffles have been performed. Note that placing a card in a specified position requires sufficient shuffles. "Sufficient shuffles" means as many shuffles as binary digits are required to represent the desired position numerically. Moving a card from the top of the pack, position 0, to the bottom of the pack, position 51, would require at least six shuffles, as the decimal number 51 is 110011 in binary. It works by examining the binary representation of the number, using each successive bit in the number to select either an in shuffle or an out shuffle.

```
[ unrot times
  [ over i bit & iff
    faro-in else faro-out ]
  nip ]      is faro ( [ n n --> [ ] )
```

Explanations and mathematical analyses of how The Technique (as magicians sometimes call it) works can be found readily enough on the web.

A common shuffling method is dealing out the pack to some number of players, then gathering up the piles of cards.

```
[ players dealall gather ] is piles ( [ n --> [ )
```

`piles` can be made slightly less deterministic by randomly reordering the piles with `shuffle` before gathering them.

```
[ players dealall  
  shuffle gather ] is mixpiles ( [ --> [ )
```

The final method of shuffling that we will consider is the imperfect riffle, where sometimes more than one card slips through the shuffler's fingers during the weave.

```
[ stack 5 ] is riffskill ( --> [ )
```

The ancillary stack `riffskill` represents the ability of the shuffler to perform a perfect riffle. This is “best guess” coding. Without performing a statistical analysis of a large number of real life riffles from a representative sample of card players, which is beyond the scope of a simple tutorial, there is no way of knowing how accurate it is. However, a while spent trying out various options suggests that it is reasonable to assume that a `riffskill` range of 1 to 10 goes from butterfingers lummo at 1 to near mastery of the riffle at 10. The default value of 5 appears to provide a reasonably adept shuffler who is not a card mechanic. (A setting of 0 will provide a perfect riffle every time, with the only random element being whether an in shuffle or an out shuffle is performed.)

`riffle` splits the pack into two halves and randomly swaps them on a coin toss (2 `random`) before entering a `while` loop nest, which, while there is one or more cards in the uppermost of the two half packs, takes a card from that half-pack, adds it to the empty nest placed on the stack in readiness at the start of `riffle` and then swaps the half-pack. Once the loop is exited the empty half-pack is dropped and any remaining cards in the non-empty half are joined to the pack.

```
[ [] swap  
  dup size 2 /up split  
  2 random if swap  
  [ dup [] != while  
    behead nested  
    dip rot join unrot  
    riffskill share 1+ random  
    1 != if swap  
    again ]  
  drop join ] is riffle ( [ --> [ )
```

Some proportion of the time, determined by `riffskill`, `riffle` will fail to swap the two halves, which is equivalent to letting more than one card slip through the shufflers' fingers. With a `riffskill` of 1, two or more cards will slip through about half the time, three or more cards will slip through a quarter of the time, four or more cards one eighth of the time and so on. With a `riffskill` of 2, the odds are 1:3 for two or more cards, 1:9 for three or more cards, 1:27 for four or more and so on.

Finally, `riffles` will perform a specified number of `riffle` shuffles on a pack of cards.

```
[ times riffle ] is riffles ( [ n --> [ )
```

Development of this suite of words could continue indefinitely, but the goal of illustrating string and nest processing in Quackery has already been more than achieved, so by way of “exercises for the reader” I suggest the wikipedia article on shuffling, where several more are described, such as the Overhand, the Mongean and the Mexican spiral.

Problem, Problem, Problem

“everyone can master a grief but he that has it”
Much Ado About Nothing — *W. Shakespeare*

By now you will probably have encountered some of Quackery’s problem messages, and possibly managed to crash out of the Quackery shell with a Python error message. Mostly the shell can recover from problems, but the open and lenient nature of Quackery does mean that it is not practical, (and quite possibly not even possible) to make it entirely bulletproof.

Problems in Quackery fall into four categories; virtual hardware problems, where the Quackery virtual processor has been asked to do something it cannot process; compilation problems, where the compiler encounters badly formed Quackscript; system damage problems, where some aspect of Quackery has been made non-viable, and pseudo-virtual hardware problems.

Virtual Hardware Problems

Stack unexpectedly empty.
Expected nest on stack.
Expected number on stack.
Return stack unexpectedly empty.
Bailed out of Quackery.
Cannot divide by zero.
Cannot ** by a negative number:
Cannot << by a negative number:
Cannot >> by a negative number:
Found a '"' at the end of a nest.
Unexpectedly empty nest.
Cannot access an item outside a nest.
Cannot remove an immovable item.
Quackery was worried by a python on the stack.
Quackery was worried by a python in the nest.

Stack Problems

Stack unexpectedly empty.
Expected nest on stack.
Expected number on stack.

These are self-evident. You can’t take something from an empty stack. If an operator takes two arguments, there need to be at least two items on the stack. Similarly you can’t do nest editing on a number, or arithmetic on a nest.

Return Stack Problems

Return stack unexpectedly empty.
Bailed out of Quackery.

You are unlikely to encounter “Return stack unexpectedly empty.” It is a remnant from an early phase of the development of Quackery, and remains in the code for the benefit of anyone seeking to modify the Python code of Quackery, or recode it in a different language. If you manage to generate this problem purely coding in Quackery, please let me know how.

“Bailed out of Quackery.” can be caused by intemperate use of]bailby[.

Arithmetic Problems

Cannot divide by zero.
Cannot ****** by a negative number:
Cannot **<<** by a negative number:
Cannot **>>** by a negative number:

These are self-evident. **/mod**, **/** and **mod** are the most likely culprits when it comes to division by zero.

Raising a number to a negative power would give a non-integer result, and the Quackery virtual processor only knows about integers.

<< and **>>** are described in the action Bitwise Logic below. If you require a word that will bit-shift in either direction, **<<>>** will give left-shift with a negative *n*, and right shift with a positive *n*.

```
[ dup 0 < iff [ negate << ] else >> ] is <<>> ( f n --> f )
```

Nest Problems

Found a **"'"** at the end of a nest.
Cannot access an item outside a nest.
Unexpectedly empty nest.
Cannot remove an immovable item.

Several operators assume that they are not the last item in a nest. **if** and **else** assume there will be at least one item following them, and **iff** assumes two items after it. If you do put them at the end of a nest, however, it is not a problem. They just consume their arguments (in the case of **if** and **iff**) and the Quackery virtual processor realises that the end of the nest has been reached and continues in its usual fashion. Neither will the compiler complain that you have done a strange thing, as it is entirely possible that the nest is destined to be joined to another nest, meaning the words would no longer be at the end of the nest.

"'", on the other hand, not only *assumes* another item in the nest, it *requires* one.

“Cannot access an item outside a nest.”

peek, **poke** and **pluck** require the specified item in a nest to exist. If the nest is not large enough for that to be true, they will report “Cannot access an item outside a nest.”

split and **stuff** are tolerant of references to positions outside of a nest, and will treat overly large positive and negative numbers as meaning the end and the start of the nest respectively.

“Unexpectedly empty nest.” will only be encountered if you try to apply **take** one too many times to something which is not a **stack** or a **table** (or more generally, anything which has been made **immovable**). As noted elsewhere, this is not recommended practice as it goes against the principle that nests should be immutable.

“Cannot remove an immovable item.” will be reported if you try to take one to many items from an **immovable** nest, such as a **stack** or **table**. The word **immovable** exists purely to detect this problem; without it ancillary stack underflow can change the behaviour of a word in bewildering ways by turning an ancillary stack into a no-op. This was the single most problematical bug during the development of Quackery. You do not need that grief in your life.

Python Problems

Quackery was worried by a python on the stack.
Quackery was worried by a python in the nest.

If you see these problems reported, somehow a Python object other than a function, list or number has wriggled into the Quackery environment. As with “Return stack unexpectedly empty.” this should never happen outside of modifying the Python code that Quackery runs on, and is a remnant from development.

When a virtual hardware problem is encountered within the Quackery shell, Quackery will attempt a soft reboot, clearing the return and data stacks, but leaving the dictionary nests and ancillary stacks unchanged, and reporting the problem along with the contents of the data and return stack immediately prior to the reboot.

When a virtual hardware problem is encountered while quackery is running a .qky file from the terminal, Quackery will attempt to report the problem along with the contents of the data and return stack before handing control back to the terminal.

When a virtual hardware problem is encountered while quackery is being used as a function within a Python program, it will raise a `QuackeryError` with a string containing the same diagnostic information as the two previous circumstances and hand control back to the Python.

Compilation Problems

Unexpected "]."
"is" needs something to define.
"is" needs a name after it.
"builds" needs something to define.
"builds" needs a name after it.
Unfinished comment.
'Unexpected ")'.
"forward" needs a name after it.
"resolves" needs something to resolve with.
"resolves" needs a name to resolve.
Unknown word after “resolves”:
<word> is not an unresolved forward reference.
"char" needs a character after it.
"\$" needs to be followed by a string.
Endless string discovered.
"say" needs to be followed by a string.
"hex" needs a number after it.
<word> is not hexadecimal.
Unknown word: <word>
unfinished nest.

These are discussed in the the section The Building Regulations, and in the descriptions of the building words that follow it.

When a compilation problem is encountered in the Quackery shell, it is reported and the shell continues as normal. `build` will restore the dictionary and protected ancillary stacks to their state prior to invoking `build`. When a compilation problem is encountered while quackery is running a .qky file from the terminal, Quackery report the problem and hand control back to the terminal. When a compilation problem is encountered while while quackery is being used as a function within a Python program, Quackery will return a string reporting the problem as the result of evaluating the Quackery function.

System Damage Problems

System Damage problems are reported when Python raises an exception that is not trapped by Quackery. Quackery prints “Quackery system damage detected.” to the terminal, followed by the exception that Python raised, and then forcibly hands control back to the operating system using the python function `sys.exit()`.

Mostly this will be a “maximum recursion depth exceeded” exception, caused by damage to the dictionaries that significantly impairs the ability of Quackery to report virtual problems, causing a problem to be detected when reporting a problem.

Note that not all dictionary damage causes this; many types of damage just cause Quackery to behave weirdly. Entering “`' pack take drop`” in the shell, for instance, will reverse the order of the items on the stack every time it is displayed by the shell, amongst other things. Weird behaviour can include an endless error loop. For instance if the soft reboot succeeds, but the dictionaries are damaged in such a way that `shell` causes a virtual hardware problem then that problem will be reported over and over, ad infinitum. And of course a poorly coded program can get into an endless loop without a problem being detected. In either instance pressing Control C will terminate Quackery.

Other causes of system damage reports could include an exception being raised by the file handling words, which would suggest there is a problem not with Quackery but with the file.

Finally, the other Quackery compiler, the one coded in Python, `build()`, who’s job is to compile the predefined words in Quackery, does its own error checking for badly formed Quackery. `build()`’s problem reports are terser and even less helpful than those generated by the Quackery Quackery compiler, `build`.

Again, this is a remnant of the development process, and left in as an aide to future developers. If you are minded to modify the predefined Quackery words, it is recommended to thoroughly test the new definitions in the shell first, if possible, using the Quackery Quackery compiler

(Also bear in mind that the full set of building words is not present in the Python Quackery compiler. Specifically, only `is`, `(`, `)`, `forward`, `resolves`, `char`, `$` and `hex` are defined, and the builders dictionary can only be extended with Python functions, not Quackery words.)

Pseudo-Virtual Hardware Problems

Occasionally, a problem may arise which is technically a compilation problem but cannot reasonably be detected during compilation. For this reason Quackery has the word `fail`, which will cause Quackery to terminate as if a virtual hardware problem had been detected whilst running a program.

The sole example of this is the problem report “Unresolved reference.”

Words are mostly added to Quackery’s vocabulary by specifying a behaviour and then naming it with `is` or `builds`. Sometimes it is desirable to reverse this order of events and name a word with, for example “`forward is example-name`” before specifying its behaviour with, for example “`[(some behaviour)] resolves example-name`”. This is known as forward referencing.

Using `example-word` before its behaviour has been specified would be problematical, but treating this as a compilation problem would be unreasonable, as it could be resolved in a future invocation of the Quackery compiler. For this reason, the initial behaviour of a word named with `forward`, until such time as it is resolved, is to generate a pseudo-virtual hardware problem that reports an unresolved reference.

Extending Quackery

Programming in Quackery *is* extending Quackery. `is` and `builds` add new definitions to the Quackery dictionaries, increasing its functionality and that of the Quackery compiler respectively. It is also meta-extensible. The dictionary system is open to extension; words could be added to enable editing the dictionaries or adding new dictionaries, and new compilers could be defined to introduce, for example, infix notation for arithmetic expressions.

Python Extensions

Some functionality is outside the scope of Quackery as defined here. It has no graphical or audio capabilities beyond sounding a system alert on systems that recognise “\a” as the bell character. Similarly it is ignorant of VT100 escape sequences for giving more control over text output than treating it as a simple teletype. Nor is it aware of other sorts of I/O, such as via the GPIO on a Raspberry Pi, to give just one example.

This out-of-scope functionality can be added to Quackery by editing the Python 3 source code. With a little familiarity with the Python language this should mostly be a straightforward task, consisting of importing any relevant Python libraries and adding new operators to Quackery, following the pattern of the existing operators; `fail` through to `sharefile`.

1. Add a new Python function to the section of code containing the existing operator functions, bearing in mind that operator functions need to
 - Check that any necessary arguments are on the stack and retrieve them.
 - If necessary, convert the arguments from Quackery data types, i.e numbers and nests (which are Python ints and lists) into the required Python data types.
 - Execute whatever Python functions are required.
 - If necessary, handle any error conditions that may arise.
 - If necessary, convert any results from Python data types into Quackery data types.
 - Push any results onto the stack.
2. New Quackery operators need to be added to the Python `operators` dictionary, the Quackery nest of strings called `namenest`, and the Quackery table called `actions`.

The order of words in the Quackery data structures is not important, but it is imperative that they both have the same order.

Quackery Libraries

The Python dictionaries can be extended from a file by use of “\$ ‘my-file-name.qky’ loadfile”, where `my-file-name` is the name of a text file of Quackery code (Quackscript).

`loadfile` has functionality to prevent a file from being loaded twice, so a file of Quackscript which is to be treated as a library can be loaded from other Quackscript files without unnecessary duplication. Specifically, `loadfile` will look for a word called `my-file-name.qky`, and ignore the request to load and compile that file if it finds that word in `namenest`.

To enable this functionality, a file of Quackscript that is to be treated as a library should start with the definition;

```
[ this ] is my-file-name.qky
```

(Any definition would suffice. Using “[this]” is a convention, not a requirement.)

To load a library called “my-library.qky” within another file use;

```
[ $ 'my-library.qky' loadfile ] now!
```

(Using **now!** will ensure that the definitions in **my-library.qky** are compiled immediately and hence available to definitions within the file that loaded it.)

The Quackery shell looks for a file called **extensions.qky** in the same directory as **quackery.py** and automatically loads it during startup, if it exists, and announces it is doing so by printing “Building extensions.” after “Welcome to Quackery.”

By default extensions are switched off by the expedient of the supplied extensions file being named “extensionsX.qky”. To enable this functionality, edit the name of the file by removing the X.

The supplied extensions file loads the library **bigrat.qky**, described below, and redefines **sortwith**, **sort** and **sort\$** to use a better sorting algorithm (merge sort rather than insertion sort) without changing the functionality of those words.

Extending Quackery by adding definitions to the “**predefined**” string in **quackery.py** is possible, and will compile faster as the Python Quackery compiler runs significantly faster than the compiler defined in Quackery. However the Python Quackery compiler is not extensible, so some compiler extensions (for example, **say**) will not be available, and any problematic code will cause Python to forcibly exit with even less helpful diagnostics than those provided by the Quackery Quackery compiler. If you must do it, develop your code in the shell, which uses the Quackery Quackery compiler, before editing it into to the **predefined** string.

As with new operators, new definitions need to be added to the the Quackery nest of strings called **namenest**, and the Quackery table called **actions**, or, for words that extend the Quackery Quackery compiler, to the nest of strings called **buildernest**, and the table called **jobs**.

The Vulgar Arithmetic Library

The Vulgar Arithmetic Library, **bigrat.qky**, short for (bignums rational”) adds rational arithmetic words (aka vulgar fractions) to the Quackery dictionary.

It can be loaded into the Quackery shell either by entering **\$ 'bigrat.qky' loadfile**, or automatically at startup by renaming the file called “extensionsX.qky” as “extensions.qky”.

To load it within a Quackery file use **[\$ 'bigrat.qky' loadfile] now!**.

The words it adds are;

```
2put gcd reduce sqrt
```

```
v+ -v v- v* 1/v v/ n->v proper improper vabs v0= v0< v< overflow vsqrt  
round approx= $->v vulgar$ proper$ +zero -zeroes -point point$
```

```
v.cf v.initcf v.nextcf v.numbers v.nextnumber v.getnumber v.denoms  
v.nextdenom v.getdenom
```

`2put (a b c -->)`

`2put` moves the stack items `a` and `b` to the ancillary stack `c`, keeping `b` above `a`.

`1 2 temp 2put` is equivalent to `swap temp put temp put`.

The converse operation `2take` can be defined as

`[dup take swap take swap]` is `2take (c --> a b)`

if required.

Rational numbers are represented by two numbers on the stack, with the numerator as second on stack and the denominator as top of stack. So the vulgar fraction $\frac{3}{4}$ would be `3 4`. `2put` adds the convenience of moving a rational number to an ancillary stack as a single step. Negative fractions, such as $-\frac{5}{3}$ are represented by making the numerator a negative number; `-5 3`. The denominator of a rational number is always positive. A rational number with a denominator of zero, e.g. $\frac{1}{0}$ represents an arithmetic overflow condition. This is discussed in the word `overflow`, below.

`gcd (a b --> c)`

`gcd` returns the greatest common divisor `c`, of the numbers `a` and `b`. It uses Euclid's algorithm, noteworthy for being one of the oldest algorithms in common usage. Here it is used by the word `reduce`.

`reduce (a/b --> c/d)`

`reduce` takes a rational number `a/b` and reduces it to its simplest terms, `c/d`. For example, the fraction $\frac{25}{100}$ would be reduced to $\frac{1}{4}$. Rational numbers in Quackery are always represented in their simplest terms.

`sqrt (a --> b c)`

`sqrt` takes a number `a`, and returns its integer square root, `b` and remainder, `c`.

For example, the integer square root of 25 is 5 with a remainder of 0, as 25 is a perfect square - i.e. $(5*5)+0 = 25$. The integer square root of 29 is 5 remainder 4, as $(5*5)+4 = 29$. If `a` is a negative number, `b` will be 0 and `c` will be the same as `a`.

`sqrt` uses the quadratic residue method described here:
[rosettacode.org/wiki/Isqrt_\(integer_square_root\)_of_X](http://rosettacode.org/wiki/Isqrt_(integer_square_root)_of_X)

`v+ (a/b c/d --> e/f)`

`v+` takes two rational numbers, `a/b` and `c/d` (i.e. four items on the stack, `a`, `b`, `c`, and `d`) and returns their sum, i.e. `e/f` is equal to $\frac{a}{b} + \frac{c}{d}$.

`-v (a/b --> -a/b)`

`-v` negates a rational number, `a/b` (i.e. multiplies it by -1).

v- (**a/b c/d --> e/f**)

v- takes two rational numbers, **a/b** and **c/d** and returns **e/f**, equal to $\frac{a}{b} - \frac{c}{d}$.

v* (**a/b c/d --> e/f**)

v* returns the product of **a/b** and **c/d**, i.e. **e/f** is equal to $\frac{a}{b} \times \frac{c}{d}$.

1/v (**a/b --> b/a**)

1/v returns the reciprocal of **a/b**, i.e. if **a/b** is $-\frac{3}{2}$, **1/v** returns $-\frac{2}{3}$.

v/ (**a/b c/d --> e/f**)

v/ returns **a/b** divided by **c/d**, i.e. **e/f** is equal to $\frac{a}{b} \div \frac{c}{d}$.

n->v (**a --> a/1**)

n->v converts a whole number, **a**, to its rational number representation, $\frac{a}{1}$.

Vulgar fractions in Quackery are “improper fractions”, which is to say that the numerator can be larger than the denominator, e.g. $\frac{7}{3}$, which is equal to $2 + \frac{1}{3}$. Sometimes it is desirable to separate the whole number part of an improper fraction from the fractional part. The word **proper** does this.

proper (**a/b --> c d/e**)

proper takes a rational number **a/b** and returns the number **c**, and the rational number **d/e**, where **d/e** is greater than or equal to zero, and less than one, and $c + \frac{d}{e} = \frac{a}{b}$.

There is a “gotcha” to watch out for here. If **a/b** is negative, the result may not be what you anticipate. for instance **-7 3 proper** will return **-3 2 3** (i.e. minus-three plus two-thirds), and not **-2 1 3** (i.e. minus two-and-a-third,) which would be the usual way of saying $-\frac{7}{3}$ as a proper fraction. This is done for consistency with the behaviour of other arithmetic operators in Quackery.

The word **proper\$**, below, which converts a rational number to a string for output as a proper fraction, operates the way you would expect, and will convert **-7 3** to the string “**-2 1/3**”.

improper (**a b/c --> d/e**)

improper is the converse of **proper**. It takes a proper fraction in the form produced by **proper** and converts it to its improper representation.

vabs (**a/b --> e/f**)

vabs takes a rational number, **a/b** and returns its absolute value **e/f** (i.e. if **a/b** is negative, it negates it.)

v0= (**a/b --> c**)

v0= returns **1 (true)** if the rational number **a/b** is equal to zero, and **0 (false)** otherwise.

`v0< (a/b --> c)`

`v0<` returns 1 (**true**) if the rational number `a/b` is less than zero, and 0 (**false**) otherwise.

`v< (a/b c/d --> e)`

`v<` returns 1 (**true**) if the rational number `a/b` is less than the rational number `c/d`, and 0 (**false**) otherwise.

`overflow (a/b --> c)`

`overflow` returns 1 (**true**) if the rational number `a/b` is equal to the overflow condition, and 0 (**false**) otherwise.

The reciprocal of zero, which is represented by the rational number $\frac{0}{1}$, is the overflow condition; $\frac{1}{0}$, also known as Undefined or Not a Number (NaN). If you call it “infinity” a mathematician will give you a stern look, and possibly a lecture.

The reciprocal of the overflow condition is the overflow condition, not zero, and more generally, the overflow condition will propagate through a program once it occurs, just as NaN will propagate through a spreadsheet and be displayed as a result on-screen.

`vsqrt (a/b c --> d/e f)`

`vsqrt` takes a rational number `a/b`, and a number `c` specifying the desired precision of the result, and returns its square root as the rational number `d/e`. The boolean `f` indicates whether the result is exact or an approximation.

As not all rational numbers have a rational square root, the number `c` specifies the precision to which the square root should be calculated, expressed as the number of decimal digits to be displayed after the decimal point, if the result were displayed as a decimal number.

(This is assuming that the current **base** is 10 (decimal). If, for example, the current base is 16 (hexadecimal) `c` specified the number of hexadecimal digits after the hexadecimal point.)

If the square root of `a/b` is a rational number and can be expressed exactly within the precision specified by `c`, the boolean `f` will be 1 (**true**), indicating that the result is exact, otherwise it will be 0 (**false**), indicating that the answer is an approximation.

If `a/b` is the overflow condition, then `d/e` will be the overflow condition and `f` will be 1 (**true**).

If `a/b` is negative, then `d/e` will be the overflow condition and `e` will be 0 (**false**).

Even without introducing approximations to irrational numbers, vulgar fractions have the drawback that when performing a large number of calculations the size of the numerators and denominators will tend to increase. For example, the sum $\frac{1}{2} + \frac{1}{3} + \frac{1}{5} + \frac{1}{7} + \frac{1}{11} + \frac{1}{13}$ is equal to $\frac{40361}{30030}$.

```
/O> 1 2 1 3 v+ 1 5 v+ 1 7 v+ 1 11 v+ 1 13 v+  
...
```

```
Stack: 40361 30030
```

This is not particularly useful if the answer you really want is “approximately $\frac{4}{3}$ ”, and unchecked growth can lead to very slow computations, and potentially running out of memory.

Schools, in my experience, circumnavigate this problem by switching to teaching decimal fractions, and encourage pupils to tap the sum into a pocket calculator and announce that the answer is “approximately 1.344022644”. To be fair, approximating decimal fractions is trivial; “truncate and round off the last digit” and approximating rational numbers is somewhat cumbersome, but significantly less so than, say, long division, and easily demonstrated with a pocket calculator, particularly if it is equipped with a π button and a $1/x$ button.

My calculator displays pi as 3.141592653589793, i.e. $\frac{3141592653589793}{1000000000000000}$. Note down the whole number part, 3, subtract it from 3.141592653589793 and find the reciprocal ($1/x$) of 0.141592653589793. It is 7.062513305931046. Note down the whole number part, 7, subtract it from 7.062513305931046 and find the reciprocal. It is 15.99659440668572. Note down the whole number part, 15, subtract it from 15.99659440668572 and find the reciprocal. Repeat this process a couple more times, until you have noted down 3, 7, 15, 1, 292. (Technically, these are the first five terms of the continued fraction of pi, but you don’t need to know that to do the method.)

The first term, 3, is the Biblical approximation to pi. (*1 Kings 7:23 KJV*; “And he made a molten sea, ten cubits from the one brim to the other: it was round all about, and his height was five cubits: and a line of thirty cubits did compass it round about.”)

The first two terms, 3 and 7, give us the best known reasonable approximation, $\frac{22}{7}$, arrived at by taking the reciprocal of the second term, 7; $\frac{1}{7}$ and adding 3 to it.

To get Milü (the approximation of pi discovered by Zǔ Chōngzhī, born 429 AD); $\frac{355}{113}$, we need to repeat the process of taking the reciprocal and adding (i.e. reversing the process of finding the continued fraction) starting with the fourth term, 1. The reciprocal of 1 is 1 ($\frac{1}{1}$). Adding the third term, 15 to this gives $\frac{16}{1}$ and the reciprocal is $\frac{1}{16}$. Adding 7 gives $\frac{113}{16}$, and the reciprocal is $\frac{16}{113}$. Finally, adding 3 give us Milü, $\frac{355}{113}$. This is approximately equal to 3.1415929204, which is a very respectable approximation of pi, being accurate to six decimal places.

This method can be turned into a useful computer algorithm by generating successive approximations at the same time as generating the continued fraction, rather than generating the continued fraction first and then trying different starting points until you find one that gives the desired size of numerator and denominator. (Starting with the fifth term, 292, would yield an approximation of $\frac{103993}{33102}$, so we can comfortably say that $\frac{355}{113}$ is the best approximation with a denominator smaller than 30000.)

Rather than go into fine detail about the algorithm, (Mediant Rounding, from D. E. Knuth’s *The Art of Computer Programming*, Vol 2, Seminumerical Algorithms, 4.5.3, Analysis of Euclid’s Algorithm, exercise 40) I present the reverse-engineered notes I referred to while coding it, below. It is essentially calculating two fibonacci sequences simultaneously, one starting with 0 1 and the other starting with 1 0, and multiplying by the next term of the continued fraction at each step.

The relationship to the fibonacci sequence becomes more apparent when calculating phi, the golden ratio, which is approximated by successive pairs of numbers in the fibonacci sequence, and whose continued fraction is an endless sequence of 1s. This, incidentally, is the worst case instance of the algorithm, where the size of the numbers in successive approximations increases most slowly.

continued fraction of pi starts: 3 7 15 1 292

approximations to pi:	3/1	22/7	333/106	355/113	103993/33102
numerators	3	22	333	355	103993
denominators	1	7	106	113	33102

cf		3	7	15	1	292	
numers	0	1	3	22	333	355	103993

0						
1		0				
3	=	1	*	3	+	0
22	=	3	*	7	+	1
333	=	22	*	15	+	3
355	=	333	*	1	+	22
103993	=	355	*	292	+	333

cf		3	7	15	1	292	
denoms	1	0	1	7	106	113	33102

1						
0		1				
1	=	0	*	3	+	1
7	=	1	*	7	+	0
106	=	7	*	15	+	1
113	=	106	*	1	+	7
33102	=	113	*	292	+	106

`round (a/b c --> d/e)`

`round` takes a rational number `a/b` and a number `c`, and returns a rational number `d/e` which is the closest approximation to `a/b` such that the numerator and denominator are both smaller (closer to zero) than `c`.

`approx= (a/b c/d e --> f)`

`approx=` compares the rational number `a/b` to the rational number `c/d` and returns `true` if the difference between them is less than x^{-y} , where x is the number `e`, and y is the number on the top of the ancillary stack `base`, and `false` otherwise.

In other words, if the current base is decimal and `e` is 3, `a/b` and `c/d` are approximately equal if they are equal when rounded to three places after the decimal point, as $10^{-3} = 0.001$.

`$->v (a --> b/c d)`

`$->v` attempts to convert a string `a` representing a number expressed in decimal point notation in the current base to a rational number. If the conversion succeeds, the number will be returned as `b/c` and the boolean `d` will be `true`. Otherwise `b/d` will be as much of the string as it was able to convert, and `d` will be `false`.

`vulgar$ (a/b --> c)`

`vulgar$` takes a rational number `a/b` and returns a string of that number expressed as a vulgar fraction, e.g `-7 5 vulgar$ echo$` will print “-7/5”.

`proper$ (a/b --> c)`

`proper$` takes a rational number `a/b` and returns a string of that number expressed as a proper fraction, e.g `-7 5 proper$ echo$` will print “-1 2/5”.

The next three words, `+zero`, `-zeroes`, and `-point`, are factored out of `point$`, below, for clarity of code, and because they may conceivably be of use.

`+zero (a --> b)`

`+zero` takes a string `a`, and returns it as string `b` with a zero “0” prepended to it if the first character is a decimal point “.”.

`-zeroes (a --> b)`

`-zeroes` takes a string `a` and returns it as string `b` with all trailing zeroes removed, unless that means that the final character in the string will be a decimal point, in which case it leaves one trailing zero.

`-point (a --> b)`

`-point` takes a string `a` and returns it as string `b` with the trailing decimal point removed, if it had one.

`point$ (a/b c --> d)`

`proper$` takes a rational number `a/b` and a number, `c`, and returns a string of `a/b` expressed in decimal point notation (dependant on the current base), with up to `c` digits after the decimal point. The final digit is rounded up if there are more digits available and the next one is 5 or more. (Or generally, half of the current value on the ancillary stack `base`.)

For example, assuming `base` is 10, `-15 7 10 point$ echo$` will print “-2.1428571429”.

The remaining words in the library, `v.cf`, `v.initcf`, `v.nextcf`, `v.numers`, `v.nextnumber`, `v.getnumber`, `v.denoms`, `v.nextdenom`, and `v.getdenom` are used by `round` and, as indicated by the use of the dotted naming convention, (i.e. `v.xyz`) should be considered private to the library and left well alone.

As with any arithmetic system that involves approximation, care should be taken to mitigate against rounding errors. While Knuth makes the passing observation in *Semi-Numerical Algorithms* that rational numbers approximated using the mediant rounding scheme show a tendency for rounding errors to cancel out, I have not found any definitive research confirming this supposition as fact. So, despite Professor Knuth’s standing, it would be foolish to rely on an “appeal to authority” argument.

As a rule of thumb for casual use of the library, I suggest working to a couple of digits higher precision than you intend to print out, and, if you are using numbers with large integer parts, separating them from the fractional part using `proper` before approximating the fractional part with `round`, and rejoining the two parts with `improper` afterwards. This will mitigate against loss of accuracy in the fractional part when the integer part is large, and against the the possibility of generating the overflow condition when the size of the integer part exceeds the specified size limit.

[this] is bigrat.qky	(BIGNum RAtional -- vulgar arithmetic)
[rot over put put]	is 2put	(n n [-->)
[[dup while tuck mod again] drop]	is gcd	(n n --> n)
[2dup gcd tuck / dip /]	is reduce	(n/d --> n/d)
[1 [2dup < not while 2 << again] 0 [over 1 > while dip [2 >> 2dup -] dup 1 >> unrot - dup 0 < iff drop else [2swap nip rot over +] again] nip swap]	is sqrt	(n --> n n)
[rot 2dup * dip [rot * dip * +] reduce]	is v+	(n/d n/d --> n/d)
[dip negate]	is -v	(n/d --> n/d)
[-v v+]	is v-	(n/d n/d --> n/d)
[dip abs]	is vabs	(n/d --> n/d)
[dup if [swap dup 0 < if [negate -v]]]	is 1/v	(n/d --> n/d)
[rot * dip * reduce]	is v*	(n/d n/d --> n/d)
[1/v v*]	is v/	(n/d n/d --> n/d)
[1]	is n->v	(n --> n/d)
[tuck /mod rot reduce]	is proper	(n/d --> n n/d)
[rot n->v v+]	is improper	(n n/d --> n/d)
[drop 0 =]	is v0=	(n/d --> b)
[drop 0 <]	is v0<	(n/d --> b)
[v- v0<]	is v<	(n/d n/d --> b)
[nip 0 =]	is overflow	(n/d --> b)

[over 0 = iff [drop true] done base share swap ** tuck * dip * tuck * sqrt 0 = dip [swap reduce]]	is vsqrt	(n/d n --> n/d b)	note 1
[stack 0 0]	is v.cf	(--> [])	
[v.cf release v.cf replace v.cf put]	is v.initcf	(n/d -->)	
[v.cf take v.cf take tuck dup if [/mod rot] v.cf 2put]	is v.nextcf	(--> n)	
[stack 0 1]	is v.numers	(--> [])	note 2
[v.numers take tuck * v.numers take + dup unrot v.numers 2put]	is v.nextnumber	(n --> n)	
[v.numers release v.numers take 0 1 v.numers 2put]	is v.getnumber	(--> n)	
[stack 1 0]	is v.denoms	(--> [])	
[v.denoms take tuck * v.denoms take + dup unrot v.denoms 2put]	is v.nextdenom	(n --> n)	
[v.denoms release v.denoms take 1 0 v.denoms 2put]	is v.getdenom	(--> n)	
[temp put 2dup < iff [1/v ' 1/v] else [] unrot v.initcf [v.nextcf dup 0 = dip [dup v.nextdenom swap v.nextnumber max temp share >] or until] v.getnumber v.getdenom rot do temp release]	is round	(n/d n --> n/d)	note 3

```

[ dip
  [ v- vabs proper rot ]
  swap iff
  [ drop 2drop false ]
  else
    [ base share swap **
      n->v 1/v 2swap v< ] ] is approx= ( n/d n/d n --> b )

[ char . over find split
  dup $ '' != if
    [ behead drop ]
  dup size
  base share swap **
  unrot join $->n
  dip [ swap reduce ] ] is $->v ( $ --> n/d b )

[ 2dup overflow iff
  [ 2drop
    $ "overflow" ] done
  reduce number$
  dip
    [ number$
      char / join ] join ] is vulgar$ ( n/d --> $ )

[ 2dup overflow iff
  [ 2drop
    $ "overflow" ] done
  2dup v0< dip
    [ vabs proper
      rot number$ ]
  if [ char - swap join ]
  unrot 2dup v0= iff
    [ 2drop ]
  else
    [ vulgar$ space
      swap join
      join ] ] is proper$ ( n/d --> $ )

[ behead dup char . =
  if [ char 0 swap join ]
  swap join ] is +zero ( $ --> $ )

[ dup size 1 - times
  [ -1 split
    dup $ '0' = iff drop
    else
      [ join
        conclude ] ] ] is -zeroes ( $ --> $ )

[ -1 split dup $ "." =
  iff drop else join ] is -point ( $ --> $ )

```



```

[ 2dup overflow iff
  [ 2drop
    $ "overflow" ] done
  dup 0 = dip
  [ unrot 2dup v0<
    dip
    [ vabs rot
      dup dip
      [ base share
        dup dip
        [ swap 1+ **
          rot * swap / ]
        tuck 2 / + swap /
        number$ ]
      negate split
      char . swap join join
      +zero -zeroes -point ] ]
    rot swap
    if [ 2 split nip ]
    swap
    if [ char - swap join ] ] is point$      ( n/d n --> $ )

```

Notes.

1. `vsqrt` only uses `sqrt` once, because $\sqrt{\frac{x}{y}} = \frac{\sqrt{x}}{\sqrt{y}} = \frac{\sqrt{x}\sqrt{y}}{y} = \frac{\sqrt{xy}}{y}$.
2. `v.numers` through to `v.getnumner` and `v.denoms` through to `v.getdenom` are unashamedly copy-paste coding, in that the routines to generate successive numerators are identical to the routines to generate successive denominators, apart from referencing the ancillary stacks `v.numers` and `v.denoms` respectively, and re-initialising them to 0 1 and 1 0 at the ends of `v.getnumner` and `v.getdenom`. The ancillary stack references could have been arguments to a generalised version of `v.nextnumner` and `v.nextdenom` at the cost of increased stack management. For just two instances I felt the cost was not justified.
3. `2dup < iff`

```

      [ 1/v ' 1/v ]
    else []
    ...
    ...
    ... do

```

The `< iff` test decides if the numerator of the rational number on the stack is less than the denominator or not, if it is, it computes the rational number's reciprocal with `1/v`, and puts a pointer to `1/v` on the stack, so that later on, `do` can reverse that process without doing an additional test. If the numerator is not less than the denominator the `else` clause puts an empty nest on the stack for `do` to do. An empty nest is the Quackery version of a no-op (i.e. a do-nothing word).

4. Coded from the middle outwards, by dealing with the simple case – a positive number with a decimal point required and no trailing zeroes, then wrapping it in code to deal with each exception to the simple case, thoroughly testing at each stage. It makes for somewhat complicated code, but then, the way humans like to see decimal point numbers presented is somewhat complicated to define precisely. As the old saw goes, “This job would be great if it wasn’t for the customers.”

Sorting and Searching

Quackery was conceived when I learned that a data structure that combines the functionality of a Queue and a Stack is sometimes referred to by the portmanteau Quack, which amused me. Soon after learning this, it occurred to me that this could be the basis of a programming language, as the Quack embodies two fundamental characteristics of programming; sequential performance of instructions, and nesting of subroutines. A circularly linked list satisfied the requirement of being able to act as a Quack, and after some consideration it grew to be two circularly linked lists, a body containing data and/or code, and a smaller head containing metadata, the head and body being joined by a link that I named the neck. The name Quackery flowed naturally from these considerations. I knew from the start that in a genealogy of programming languages it would be a child of both Forth and Lisp, and the name had some self-effacing humour; I imagined that someone encountering the language without some experience of its forebears might cry out “But this is Quackery!”, and they would be correct.

After settling on Python 3 as the implementation language I realised that the built in Python data structure it calls a list (actually a dynamic array) also met the criteria and removed the need to create the data structure described above and the low level code required to manage it from scratch. And so the Quack, with its head and body was superseded by the nest as the basis of Quackery, the stacks remained as stacks in the very specific computer science meaning of the term, but only an echo of the queues remained as sequential compilation and execution.

Other early design decisions included the non-inclusion of Go To, even at the virtual machine level, (see “Go To Statement Considered Harmful” by Edsger Dijkstra, March 1968.) The immutability of nests (i.e. the idea that nests cannot be modified, only created and destroyed) came from Lisp, and a limited relaxation of this principle for ancillary stacks allowed Quackery to work without variables. (See “Global Variable Considered Harmful” by William Wulf & Mary Shaw, February 1973.)

In this chapter we will first test the viability of making nests immutable by implementing a couple of in-place sorts, the Gnome Sort, a recent classic, which should probably be the first sort algorithm taught, on account of its simplicity, and a heap sort, mostly because it uses priority queues, thereby reintroducing queues in their strict computer science sense to Quackery. After that we will create a string search algorithm that relies heavily on the Go To statement in its most spaghetti-codifying incarnation, which I first encountered in various incarnations of Microsoft BASIC as ON x GOTO. This was also the source of the word **poke**, which will be used in the sort algorithms, and will be our means of turning immutability on and off for comparisons.

In the next chapter, Quackery is Not Forwards Lisp, we will consider the possibility of restricting branching within nests even further.

Gnome Sort

Gnome Sort is possibly the simplest sorting algorithm to describe, and is very suitable as a first sorting algorithm, so if you have not coded one before I suggest you read the description and try it out before proceeding to the next page. The words **peek** and **poke** will be needed, and are described in the comprehensive Word Behaviour section below, under the subheading Nest Editing. As a hint, you might also look at **times** and **step** under the Control Flow subheading. This description is from the Wikipedia Gnome Sort page at en.wikipedia.org/wiki/Gnome_sort.

Gnome Sort is based on the technique used by the standard Dutch Garden Gnome.

Here is how a garden gnome sorts a line of flower pots.

Basically, he looks at the flower pot next to him and the previous one; if they are in the right order he steps one pot forward, otherwise, he swaps them and steps one pot backward.

Boundary conditions: if there is no previous pot, he steps forwards; if there is no pot next to him, he is done.

— “Gnome Sort - The Simplest Sort Algorithm”. Dickgrune.com

Here is a coding of Gnome Sort to sort a nest of numbers into numerical order, (and hence to sort the characters in a string into Unicode order) along with a quick test to establish that it works.

```
/O> [ dup size times
...   [ i^ 0 > if
...     [ dup i^ 1 - peek
...       over i^ peek
...       2dup > iff
...       [ dip [ swap i^ poke ]
...         swap i^ 1 - poke
...         -1 step ]
...       else 2drop ] ] ] is gnomesort ( [ --> [ ]
...
```

Stack empty.

```
/O> $ 'who knows why the wind blows' gnomesort echo$
...
bdehhhiklnnoosstwwwwwy
Stack empty.
```

The key word here is **poke**, which occurs twice within the **times** looping structure, and which we will take to be typical usage. **poke** takes its name from Applesoft BASIC (and other dialects of rebadged Microsoft BASIC prevalent in home computers of the late 1970s and early 1980s), where I first encountered it. It overwrote a specified address in memory with a specified 8-bit number.

In Quackery the idea of the computer memory (RAM) as a single array of memory locations is abstracted into nests, which can be seen as multiple arrays of memory locations. The Quackery **poke** takes three arguments, the item (a number, nest, or operator) that will replace the contents of one of these memory locations, the nest which is to be modified, and a number specifying the memory location within that nest. And, as nests are immutable, it does not actually change the contents of the nest. Instead it returns a copy of the entire nest, with the relevant location modified.

This sounds terribly inefficient. Imagine sorting the contents of a large nest, and making two copies of the nest every time it goes around the loop and meets the necessary condition to exchange two items. Especially with an already inefficient algorithm such as Gnome Sort, which can make many more trips around the loop than are strictly necessary.

We can put this to the test. Is this inefficiency a major factor, a deal breaker for immutability?

As it happens, **poke** is not used at all in the predefined or core words of Quackery, so we can modify its functionality, making it a word that modifies a nest rather than creating a copy, without affecting the way any other part of Quackery works.

The Python 3 definition of **poke** includes the line “**nest = from_stack().copy()**”, with the suffix “**.copy()**” forcing **poke** to have the the immutability property. Removing that suffix will mean that **poke** does not return a copy of the nest passed to it as an argument, but the same nest, modified.

If you wish to test this for yourself by running some timing comparisons, the word **time** returns the number of microseconds since the Unix epoch. (It is not reliable to the microsecond, it does this so it can provide an unpredictable seed for Quackery’s pseudo-random number generator. The Python 3 documentation only guarantees accuracy to the nearest second, but in all probability using it to make measurements to the thousandth of a second for casual purposes should be fine.

```
time ( code-you-are-timing ) time swap - 1000 / echo say " milliseconds"
```

In tests I have conducted, there has been only a marginal improvement of between 1 and 2 percent.

Without understanding the inner workings of the Python interpreter in detail, it is wise to view this conclusion with some scepticism, and perhaps conduct a different test that does not assume that `.copy()` always forces a copy of the nest to be made. There are possible optimisations that mean it need only make a copy when it has to, and if there is only a single reference to the nest passed to `poke` it could get away with modifying the nest whilst keeping the appearance of immutability.

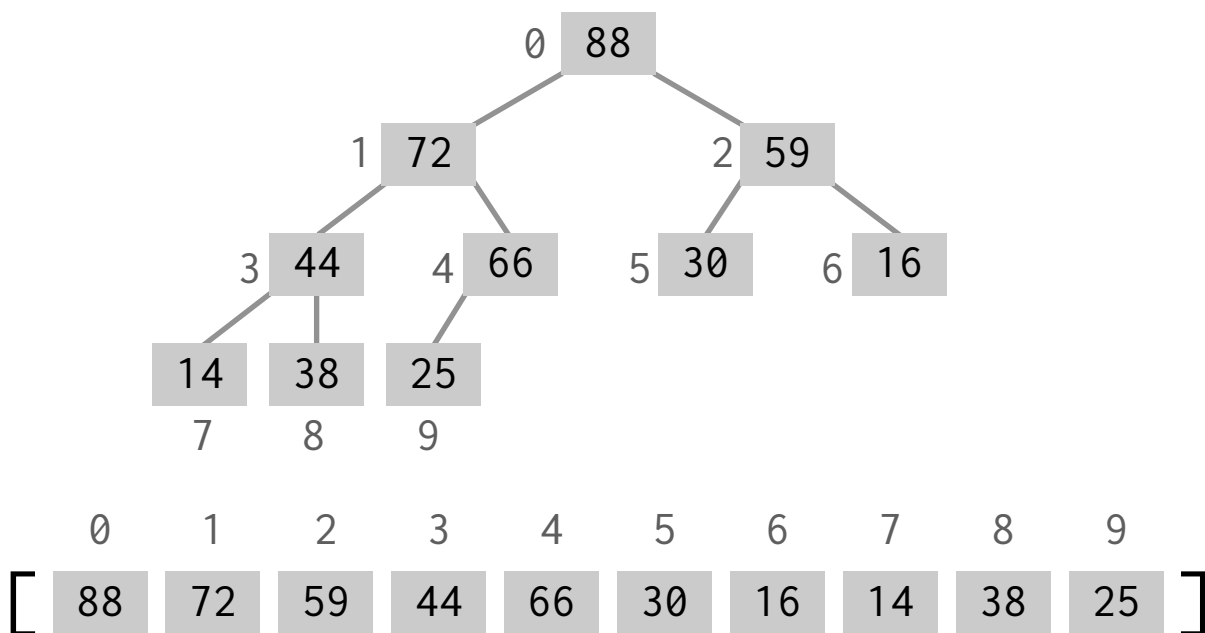
Heap Sort

Heap Sort uses a nearly balanced binary tree as a priority queue represented as a heap, which is an nearly balanced binary tree in a nest. Let's break that down. The code for the heap sort can be found in the folder `sundry`, in the file `heapsort.qky`.

A priority queue is what you use in a triage situation, dealing with things as they arrive, not in the order that they arrive, but prioritising them in order of urgency. For the sake of this illustration, we will say that smaller numbers are “more urgent” than larger numbers.

This is an nearly balanced binary tree in which the higher up the tree you are, the “more urgent” the numbers are, with the “most urgent” (i.e. smallest) number at the top. You can make and maintain a nearly balanced binary tree by add new nodes in the order indicated by the grey numbers adjacent to the nodes, and removing nodes in the reverse order.

It is a min heap, because each node has a smaller number in it than the nodes below it that it is connected to (the node's children). We say it satisfies the min heap condition. A max heap would be the same except the number in a node would be larger than those of its children.



The key takeaways from this diagram are;

1. That by knowing the size of the nest and the position of a given node within the nest we can calculate whether it has a parent node, and the position of the parent node in the nest, and whether it has zero, one, or two children, and the positions of the left and right child in the nest, with simple arithmetic operations (doubling, halving, adding and subtracting one) and comparisons (greater than and less than). This will allow us to move from node to node.
2. If we remove the zeroth item in the nest or add an item to the end of the nest it will no longer satisfy the heap condition, and will need to be corrected.

A preliminary coding of the priority queue routines turned out to be quite stack intensive, leading to an unnecessary amount of stack management words. So we will relieve the stack pressure by storing the priority queue on an ancillary stack called `pq` and define routine specific versions of `peek` and `poke`.

```
[ stack ] is pq ( [ --> [ )
[ pq share swap peek ] is pq.peek ( n --> x )
[ pq take swap poke pq put ] is pq.poke ( n x --> )
```

With that taken care of, we can define the relationships between nodes of the binary tree.

To move from a node to its parent, add one, divide by two and subtract one.

```
[ 1+ 2 / 1 - ] is parent ( n --> n )
```

Every node has a parent except for the one in position 0.

```
[ 0 > ] is has-parent ( n --> b )
```

To move from a node to its left child, double and add one.

```
[ 2 * 1+ ] is child ( n --> n )
```

A node has a child if that child is within the nest.

```
[ child pq share size < ] is has-child ( n --> b )
```

To move from a left node to its sibling right node, add one.

```
[ 1+ ] is sibling ( n --> n )
```

Every left node has a sibling if that node is within the nest.

```
[ sibling pq share size < ] is has-sibling ( n --> b )
```

To make the priority queue routines general purpose, we will specify the test for “more urgent” at run time, by putting it on an ancillary stack called `comparison`, and performing that test with `pq.compare`. For example, the appropriate comparison for a min heap of numbers where smaller numbers are “more urgent” than larger ones is `>`, which will return true if the top of stack is “more urgent” than the second on stack.

```
[ stack ] is comparison ( [ --> [ )
[ comparison share do ] is pq.compare ( x x --> b )
```

Before moving on to the heap management words, you may have spotted an obvious optimisation in Gnome Sort. When moving a flower pot towards the start of the row of flower pots, he repeatedly exchanges a flower pot with its neighbour. It would have required less effort to pull the chosen flower pot forward and then slide flower pots to the right until there is a gap where the pulled-forward pot is destined to end up. We did not make that optimisation for the gnome sort, but for the heap we will be creating a space and then moving it up or down the binary tree, much as one moves the empty space around in the classic 15 Puzzle.

Note this if you augment these notes with other tutorials (a good idea), and also note that most use an array that starts with the first element, rather than the zeroth, so the arithmetic will differ slightly.

toheap – We can add an element, *x*, to a heap, *h*, by joining it to the end of the heap and then restoring the heap property by walking the element up the binary tree until we encounter a parent node that is more urgent than it, or we get to the top of the tree, which does not have a parent node.

```
[ over size
  rot 0 join pq put
  [ dup has-parent while
    dup parent
    rot over pq.peek
    2dup pq.compare iff
    [ 2swap unrot pq.poke ]
    again
    rot 2drop swap ]
  pq.poke pq take ]          is toheap      (      h x --> h      )
```

fromheap – We can remove the zeroth element from a heap, *h*, by noting its contents and replacing it with the last element of the heap, then walking it down the tree much as **toheap** walks *n* item up the tree, with the added complication that when moving downwards one has to choose between the two siblings if a node has two children. We choose the more urgent one.

The process of restoring the heap condition in this manner does not just work with the zeroth element – if the heap condition is satisfied for all the descendants of any given node, *n*, the process, known as “heapify” can be applied starting at *n*. As it will be used in the word **makeheap** this part of **fromheap** has been factored out into the word **pq.heapify**, the **pq.** portion of the name indicating that it expects to find the heap on which it will act on the ancillary stack **pq**.

```
[ dup pq.peek swap
  [ dup has-child while
    dup child
    dup has-sibling if
    [ dup sibling pq.peek
      over pq.peek
      pq.compare if sibling ]
    dip over dup pq.peek
    rot dip dup pq.compare iff
    [ rot pq.poke ]
    again
    2drop ]
  pq.poke ]          is pq.heapify      (      n -->      )

[ behead
  over [] = if done
  swap -1 split
  swap join pq put
  0 pq.heapify
  pq take swap ]          is fromheap    (      h --> h v      )
```

comparison, **toheap** and **fromheap** provide all the functionality of a priority queue. The remaining words, **makeheap**, **hsortwith** and **hsort** use the priority queue functionality to implement an efficient sorting mechanism. However, it should be noted that heap sorts are not stable. (If two elements have the same value, their relative order is not preserved. This can be important if you want to sort a nest of data structures that can be sorted in more than one way – e.g. some records that have both an alphabetic ordering and a date ordering. One might want that sorting the records into alphabetic order and then sorting the result by day would maintain the alphabetic order of records within each day.

Quackery is equipped with a stable insertion sort suitable for sorting small nests, and loading **extensions.qky** improves on this with a stable merge sort for longer nests that falls back to insertion sort for nests with sixteen or fewer items.

makeheap heapifies an unsorted nest. Note that **pq.heapify** need only be applied to the upper half of the binary tree, as nodes which have no children automatically satisfy the heap condition. This is the first step in the two step heap sort process. **makeheap** assumes that the ancillary stack **comparison** has been loaded with a suitable test for urgency.

```
[ dup pq put
  size 2 / times
    [ i pq.heapify ]
  pq take ] is makeheap ( [ --> h ] )
```

The second step in the heap sort process is to remove every item from the priority queue in order of priority by repeatedly calling **fromheap** and appending the items it returns to a nest.

hsortwith will sort a nest using the comparison that follows it in the Quackery code. It uses **]'** to provide this syntactic sugaring and give it the same usage as the provided word **sortwith**.

```
[ ]' [ comparison put
  [ ] swap makeheap
  dup size times
    [ fromheap
      nested rot join
      swap ]
  drop
  comparison release ] is hsortwith ( [ --> [ ] )
```

hsort illustrates the usage of **hsortwith** and sorts a nest of numbers into ascending order. Usage is the same as the provided word **sort**.

```
[ hsortwith > ] is hsort ( [ --> [ ] )

/O> [ ] 10 times [ 90 random 10 + join ]
... cr dup echo cr hsort echo cr
...

[ 97 93 85 73 35 44 75 98 28 58 ]
[ 28 35 44 58 73 75 85 93 97 98 ]
```

Stack empty.

Returning to the topic of the possible inefficiency of immutable nests, the definition of **pq.poke** can be changed from

```
[ pq take swap poke pq put ] is pq.poke ( n x --> )
```

to

```
[ pq share swap poke pq replace ] is pq.poke ( n x --> )
```

without changing its functionality. The latter version ensures that **poke** does copy the nest passed to it. Comparative timings sorting large nests again show a change in how long it takes to sort a large nest of numbers with **hsort** on the order of two or three percent. That there is only a small change suggests that possibly it does not copy the entire nest, but instead records only the differences, the “deltas”, between the original and the new version. Again, without delving deep into the implementation details of Python 3 and PyPy3, this is uncertain. However it does reassure us that it is not an overwhelming factor. There are plenty of small optimisations possible in the implementation presented here which, if all applied could make it run significantly faster. However I feel this is not necessary in its intended role as a recreational programming language, and at odds with its ambitions as an educational language.

Finite Automata String Search

While it is generally accepted that `goto` statements can lead to unstructured programs that are difficult to debug and hard to maintain there are use cases that are best served by unstructured code. An extreme example of this is in coding Finite State Automata, machines that were in use long before computing was invented. We will look first at the example of a mechanical vending machine.

One stage in the process of designing a vending machine is to consider what coins may be fed into it before it vends, say, a chocolate bar. We will start with a specification that chocolate bars cost eighty pence, that the machine can accept ten, twenty and fifty pence coins, and that when it receives exactly eighty pence it will vend a chocolate bar, but if that amount is exceeded without ever being exactly eighty pence it will return the coins and not vend a chocolate bar.

From this specification we can draw a state table which will help when designing the mechanism. The machine starts in the initial state, **state 0**, where it is awaiting **80** pence. Depending on the value of the first coin it receives, it will move to one of three other states, where it is awaiting **70**, **60**, or **50** pence. From whichever state it is in it can move to one of three other states, again depending on the value of the next coin it receives. There are eight possible states where it is awaiting further coins, namely awaiting **80**, **70**, **60**, **50**, **40**, **30**, **20**, or **10** pence, which we will call states **0** to **7**, the success state **S!**, where it has received exactly **80** pence and will vend a chocolate bar, and the failure state, **f!** where it has received more than **80** pence and will return the coins received. Once it has reached and acted on states **S!** or **f!** it returns to state **0**.

coin received:				
	10	20	50	
state 0 - awaiting 80p	s1	s2	s5	
state 1 - awaiting 70p	s2	s3	s6	
state 2 - awaiting 60p	s3	s4	s7	
state 3 - awaiting 50p	s4	s5	S!	
state 4 - awaiting 40p	s5	s6	f!	
state 5 - awaiting 30p	s6	s7	f!	
state 6 - awaiting 20p	s7	S!	f!	
state 7 - awaiting 10p	S!	f!	f!	

Say a customer puts a **20** pence coin, a **10** pence coin and a **50** pence coin into the machine, in that order. The machine is initially in state **0**. Reading from the state table we can see from the **state 0** row that the **20** pence coin will cause the machine to enter state **2**, from the **state 2** row that the **10** pence coin will cause the machine to enter state **3**, and from the **state 3** row that the **50** pence coin will cause the machine to enter the success state, **S!**, and vend a chocolate bar before returning to state **0**. Hurrah!

The Infinite Monkey Theorem states that, given a typewriter and all eternity, a monkey will, at some point, recreate the entire works of William Shakespeare.

As we do not have that long, we will address the Finite Monkey Puzzle, which asks, if a monkey equipped with a faulty typewriter, such that it can only type the letters A, B, N, and R types a single page of text, what is the probability that it will reference the Beach Boys by typing BARBARAANN somewhere in that page?

Rather than answer that question, we will create the state table for a machine capable of recognising the sequence of letters BARBARAANN when it is fed one letter at a time from the page typed by our monkey, much as the chocolate bar vending machine can recognise a sequence of coins totalling eighty pence.

(In passing I note that this apparently silly task is equivalent to the more serious task of searching the human genome for a specified sequence of nucleotides, so has a beneficial real-world application.)

This will bring us quite close to coding a string search algorithm, with the main difference that rather than having an alphabet of four characters, we will use the 95 characters that Quackery recognises.

For this state table we will use a nest of Quackery nests as this is the format we plan to use in the code;

```
( 'A' 'B' 'N' 'R' )
[ [ 0 1 0 0 ] ( state 0 - waiting for 'B' )
  [ 2 1 0 0 ] ( state 1 - waiting for 'A' )
  [ 0 1 0 3 ] ( state 2 - waiting for 'R' )
  [ 0 4 0 0 ] ( state 3 - waiting for 'B' )
  [ 5 1 0 0 ] ( state 4 - waiting for 'A' )
  [ 0 1 0 6 ] ( state 5 - waiting for 'R' )
  [ 7 4 0 0 ] ( state 6 - waiting for 'A' )
  [ 8 1 0 0 ] ( state 7 - waiting for 'A' )
  [ 0 1 9 0 ] ( state 8 - waiting for 'N' )
  [ 0 1 -10 0 ] ] ( state 9 - waiting for 'N' )
```

The success state in this state table is indicated by a negative number, `-10`. This will be a useful choice when we come to write the code. (I know this because, having already actually written the code, I have the gift of hindsight.)

Of particular interest is the column for the letter 'B'. If a B is received when it is not the letter that the state machine is waiting for, it mostly it jumps back to state 1 rather than state 0 as other undesired letters do. This is because the undesired letter may well be the first letter of the sequence we are looking for, whereas an undesired 'A', 'N', or 'R' is definitely not the first letter of BARBARAANN. The exception is state 6, where an undesired B causes a jump back to state 4. This is because when we reach state 6 we know that the previous six letters were BARBAR, so this B could be the fourth letter in BARBARAANN. To confirm this is correct, try following the sequence of letters B A R B A R B A R A A N N through the state table as we followed the sequence of coins 20 10 50 through the state table in the first example.

Also note that there is no failure state indicated in this table. This is because in this example failure would consist of running out of letters to feed to the state machine before reaching the success state. We will test for this separately in the code.

A string search routine that only looks for the sequence “BARBARAANN” is of limited utility, and I would not want to hand-craft a state table every time I wanted to search for a different substring. Full disclosure – I didn’t hand-craft the state tables above, instead I adapted them from the output of a routine coded in Quackery that takes a string and returns a state table. This is the first part of the string search algorithm. The second part is a routine that takes a string to search and a state table and acts as a state machine, moving from state to state according to the sequence of characters in the string until it reaches the success state or runs out of characters (the failure state). It returns the number of characters it had to process before reaching a success or failure state.

The two routines are, in essence, a compiler (the first routine) that takes a textual description of a problem (the string to be matched) and produces a chunk of code (the state table) that can be acted on by a processor (the second routine) to reach a success or failure state. This indicates the position of the string to be matched within the string to be searched. (In common with `find` and `findseq` the failure state is represented by returning the length of the string to be searched, so `found` can be used to test for failure. This is a remarkably powerful programming technique, and the basis of the Unix tool `grep`, which compiles complex state tables from regular expressions to match patterns within strings. The code presented here is a first step in developing a `grep` in Quackery. It is almost sufficiently powerful to qualify as a full-blown programming language, being just a hair shy of Turing equivalent. (Technically, this is a *deterministic* state machine – it will always either succeed or fail. Turing equivalence requires *non-determinism*; the possibility that one could present it with a program that will not be able to determine success or failure and hence run forever.

In Finite State Machine terminology, the set of possible input events is called an alphabet. In this instance the alphabet is the set of printable characters and **space**. For the purposes of the string search, carriage return will be treated as equivalent to the space character. We start by creating a nest of all the characters Quackery recognises so that we can step through them one at a time when required. The word **constant** ensures that the nest is created only once, at compile time.

```
[ [] 95 times
  [ i^ space +
    join ] ]      constant is alphabet      (      --> $      )
```

(**constant** could also have been used in the card dealing routine **newpack**, earlier, in case you had wondered about something like that at the time. The reason it wasn't is that I had yet to define the word **constant** when that chapter was written.)

In order to construct a state table, we will need to consider all the possibilities. We will look at one element of the finite money state table in detail, the one where we have reached state 6, and the next character in the string we are searching in is B.

```
( 'A' 'B' 'N' 'R' )
...
...
...
[ 7  4  0  0  ]      ( state 6 - waiting for 'A' )
```

As we have reached state 6 we know that the previous six characters in the string we are searching in were **BARBAR**, and appending a B will give us **BARBARB**, which is seven characters long. We will compare this to the first seven characters of the string we are searching for, **BARBARA**.

```
BARBARB
BARBARA
```

This is not a match, so we clip one character off the start of **BARBARB**, and one character off the end of **BARBARA**, and compare again.

```
BARBARB
BARBARA
```

Again this is not a match, so we repeat the process.

```
BARBARB
BARBARA
```

Again this is not a match, so we repeat the process..

```
BARBARB
BARBARA
```

Now we have a match of four characters, so the number we need to record in the state table at that point is 4, the number of characters by which the two strings overlap.

```
[ [ 2dup != while
    -1 split drop
    swap 1 split
    unrot drop again ]
  drop size ]      is overlap      (      $ $ --> n      )
```

The next word, `eachend`, takes the string matched so far, plus the next character that we are hoping to find, (i.e. for the sixth row of the state table, “BARBARA”) and also the total length of the string we are searching for, `n`, (i.e. for “BARBARAANN”, `n` equals 10. It calls `overlap` with, for example, “BARBARA”, on the top of stack, and the same string with each possible end character substituted into “BARBARA”, i.e. “BARBARA”, “BARBARB”, “BARBARN”, and “BARBARR” second on stack. (Except with every character in `alphabet`, not just A, B, R, and N.)

The number returned by `overlap` is compared to the total length of the string we are searching for (in the example, 10) and if necessary, the returned number is negated, as a means of indicating that a match for the string has been found, so the success state has been achieved.

```
[ temp put [] swap
  alphabet witheach
    [ over -1 poke
      over overlap
      dup temp share
      = if negate
      swap dip join ]
  drop temp release ]      is eachend      (      $ n --> [      ] )
```

`buildfsm` builds the state table for the Finite State Machine by calling `eachend` for as many lines of the state table as there are characters in the string being searched for.

```
[ [] swap
  dup temp put
  size times
    [ temp share
      i 1+ split drop
      temp share size eachend
      nested swap join ]
  temp release ]      is buildfsm      (      $ --> [      ] )
```

`nextcharn` takes a string, and returns -1 and an empty string if the string passed to it is empty, otherwise it returns the position in `alphabet` of the first character in the string (remembering that the carriage return character is treated as equivalent to the `space` character), and the string passed to it with the first character removed.)

`nextcharn` is used to feed the finite state machine processor `usefsm` with successive characters from the string being searched in, and to indicate to it that the failure state has been achieved if there are no characters left in the string.

```
[ dup [] = iff -1
  else
    [ behead
      dup carriage = if
        [ drop space ]
        space - ]
  swap ]      is nextcharn      (      $ --> n $      )
```

`usefsm` is the processor for the Finite State Machine. It takes the string to be searched and a state table, and moves from line to line in the state table as determined by successive characters from the string, until either it runs out of characters to process (the failure state) or achieves the success state.

It returns the starting position of the first instance of the string being searched for (as encoded in the state table) in the string being searched in, if it has been found, or the length of the string being searched in if there were no instances of the string being searched for.

```
[ swap dup size
  swap temp put
  swap 0
  [ over swap peek
    temp take
    nextcharn
    temp put
    dup 0 < iff
      [ 2drop 0 ] done
    peek
    dup 0 < until ]
  nip
  temp take size - + ]      is usefsm      (      $ [ --> n      )
```

`find$` takes a string to be searched for, (the substring) from which it will build a state table, and then use it to search within the string on the top of the stack, returning the position of the substring within the string, or the negate of the string, if the substring is not present. First, however, it checks to see if the substring is the empty string, as that would crash the finite state machine. We could cause Quackery to crash with a suitable problem report using `fail` if the substring is empty, but as it is not an entirely unreasonable thing to ask (like asking Quackery to divide by zero) instead we will return a result of `0`. (It is reasonable to assert that there is an empty string at the start of every string, as `any-string 0 split` will return an empty string and `any-string`.)

```
[ over [] = iff
  [ 2drop 0 ]
  else
    [ swap buildfsm
      usefsm ] ]      is find$      (      $ $ --> n      )
```

`buildfsm`, with its nested loops, is a relatively slow process, but when the string being searched is very large, and the string being searched for is short and the alphabet used is quite small, that is a small price to pay for the benefit of `usefsm` only looking at each character once. In `find$` `buildfsm` is used every time `find$` is used, but in a word that finds not just the first instance of the substring, but every instance, `buildfsm` need be only used once, further reducing its overhead. The code for the finite state machine can be found in the folder `sundry`, in the file `fsm.qky`.

This brings us to the end of our excursion into the world of sorting and searching. In the next chapter we will ask, if we do not require `GOTO`, what other control flow words can we do without?

After that final plunge into “Computer Science Quackery Style”, (a reference to the rather excellent book *Computer Science Logo Style*, by Brian Harvey: people.eecs.berkeley.edu/~bh) it is possible, that the rest of this document, which describes every word in the Quackery lexicon, (and looks in some detail at several of them, with examples of usage), and then presents the entire code for Quackery, implemented in Python 3 and Quackery, will seem like a walk in the park by comparison.

Quackery is Not Forwards Lisp

In this section we will ask the rhetorical question, What if Quackery didn't have the control flow words introduced in the section Control Flow Words, namely **again**, **done**, **if**, **iff**, **else**, **until** and **while**; could we manage without them?

It's an intriguing proposition. In the section The Other Stack we established that running a Quackery program consists of proceeding from left to right, through a nest, using the return stack to keep track of nested nests. In Control Flow Words this simple rule was weakened, by making it "usually, proceed from left to right" and then introducing some ways to break that rule by skipping over items and jumping to the start and end of a nest, proving a simple but powerful and flexible set of control flow words.

How this is achieved, and how additional control flow words can be created using the "meta control flow operators" is covered the appropriate part of Word Behaviours, below, but for now we will mention briefly that **done** is not, as you might suspect, an operator, but a nest, and it is defined as `[]done[]` is done.

`]done[` (pronounced meta-done") is one of the meta-control flow operators. As you will recall in Control Flow Words, nesting causes the the item number and nest pointer of the nest currently being traversed to be moved to the return stack. `]done[` removes them from the return stack.

This idea, of operating indirectly on the pointers in the Quackery virtual processor by use of the meta control flow words, along with the ideas that this section covers, will be used in Behaviour of Words to explain the behaviour of words such as **times** and **witheach**, which proved so useful in the preceding section, Dealing With Quackery.

Control flow boils down to two ideas, selection and repetition. We can use `'` and `do` along with `peek` to select between two courses of action...

```
/0> true ' [ [ say "the top of stack was 'false'" cr ]  
...      [ say "the top of stack was 'true'" cr ] ]  
... swap peek do  
...  
the top of stack was 'true'
```

Stack empty.

```
/0> false ' [ [ say "the top of stack was 'false'" cr ]  
...      [ say "the top of stack was 'true'" cr ] ]  
... swap peek do  
...  
the top of stack was 'false'
```

Stack empty.

This code takes advantage of the fact that the logical values **false** and **true** are represented in Quackery as 0 and 1 respectively, and that `peek` returns the nth item in a nest, with item numbers starting at 0.

It is a little odd that the 0th item in the nest is the "false" option, and the 1st item the "true" option, but that can be reversed with the word **not**, which turns **true** into **false** and vice versa.

With this refinement, we can now define a word to select between two options as;

```
[ swap not peek do ]          is 'ifelse ( b [ --> )
```

and demonstrate that it is functionally equivalent to ... iff ... else ...;

```
/O> [ ' [ [ say "true" cr ]  
...     [ say "false" cr ] ]  
...   'ifelse ]          is test1 ( b --> )  
...
```

Stack empty.

```
/O> [ iff [ say "true" cr ]  
... else [ say "false" cr ] ] is test2 ( b --> )
```

Stack empty.

```
/O> 2 2 = test1  
... 2 2 = test2  
... 2 3 = test1  
... 2 3 = test2  
...  
true  
true  
false  
false
```

Stack empty.

Now that we have 'ifelse, we can define 'if by noting that “if [say "true"]” is equivalent to “iff [say "true"] else [(do nothing)]”.

```
[ nested  
  ' [ [ ] ] join  
  'ifelse ]          is 'if ( b x --> )
```

If x is the nest “[say "true"]”, then 'if turns it into “[[say “true”] []]” and 'ifelse selects between “[say "true"]” and “[]”.

```
/O> [ stack 4 ] is lights  
... lights share 4 = ' [ say "There are four lights." cr ] 'if  
...  
There are four lights!
```

Stack empty.

```
... lights share 5 = ' [ say "There are four lights." cr ] 'if  
...  
Stack empty.
```

The next stage is to achieve repetition without using again to jump back to the start of a nest by building nests that include themselves as an item. Imagine a model village. There is one near where I live called Bekonscot, the first such scale model ever built, portraying aspects of life in 1930s England. It has the usual items you would find in a such a scale model, all a twelfth of the size of their real life counterparts; a high street with a church and a pub, a train station, a cricket pitch, and, of course, a model village, specifically, a model of Bekonscot. It includes itself. Inside the model of Bekonscot are all the items you find in Bekonscot, at 1:144 scale, including of course, Bekonscot.

Nests do not need to include themselves quite that literally. They just need to be able to refer to themselves, just as they can refer to other nests by name, using their entries in the names and actions dictionary.

The word **this** is what a nest calls itself. It's behaviour is "Put a pointer to this nest on the stack."

```
/O> [ [ 10 random echo cr ] this ]
...
3

Stack: [ [ 10 random echo cr ] this ]

/O> do
...
9

Stack: [ [ 10 random echo cr ] this ]

/O> do
...
4

Stack: [ [ 10 random echo cr ] this ]
```

... and as long as we keep entering **do**, the same thing will keep happening. It's the basis of an endless loop, which we will call 'forever - all we have to do is build a nest with "this do" at the end.

As with 'if, we will start the definition with **nested**, not only for consistency, but because the nest it finds on the stack could have words like **while** and **done** in it, which rely on the end of the nest not having stuff arbitrarily added to it, as they right to the end of the nest, wherever it is.

```
[ nested
  ' [ this do ]
  join do ]      is 'forever ( x --> )
```

We can test 'forever with something like ' [10 random echo cr] 'forever and watch a stream of random numbers scroll rapidly up the screen. (The **cr** may be necessary, depending on your system. On my Mac text isn't echoed to the screen until there is a carriage return or some internal buffer is filled, which can take a while. In Pythonista on my iPad text is echoed immediately.) Press Control C to break out of this endless loop.

But... there is a potential problem which will be revealed with more thorough testing. Replacing "10 random" with the development tool **nestdepth** (which shows how much is on the return stack at any given time) will show that it is increasing by one with every repeat. So 'forever should really be called 'keepgoinguntilyourunoutofmemoryforthereturnstack, which isn't ideal unless your system has infinite memory. We can fix this by including the meta control flow word **]done[**, which, as noted above, removes a nest pointer and item number from the return stack. **nestdepth** counts this pair as one thing. (The technical term for this fix is "tail-call optimisation".)

```
[ nested
  ' [ ]done[
    this do ]
  join do ]      is 'forever ( x --> )
```

" [... ...] 'forever" is much like "[... ... again]". We can develop this into a word much like "[... ... until]" by noting that 'if is a conditional version of **do** – it does the thing on the top of stack, but only if the second on stack is true.

As a first pass we could try;

```
[ nested
  ' [ this 'if ]
  join do ]      is 'until ( x --> )
```

It's close, but not quite there. Testing shows that the logic is reversed, we need a **not** before this **'if**, and as before the return stack fills up rapidly. Worse than with **'forever** in fact – this time it increases by three each time on account of the extra nesting that **'if** wraps around its argument, so we would need **]done[]done[]done[** to fix it. As it happens, there is another meta control word, **]bailby[**, which allows us to say that a little more tersely, as **3]bailby[**.

Almost there. The return stack depth increases by three every time... except for the first time. Uh-huh. This we can fix by adding some extra nesting around the **do**, leaving us with a finished version of;

```
[ nested
  ' [ 3 ]bailby[
    not this
    'if ]
  join
  [ [ do ] ] ] is 'until ( x --> )
```

Testing this shows it now works as desired.

```
/O> ' [ nestdepth echo sp
...    4 random dup echo cr
...    0 = ] 'until
... say " finished"
10 3
10 1
10 0 finished
```

The return stack depth is constant at **10**, it repeats until **4 random** returns a **0**, and we can see that it doesn't remove too many return stack items at the end, because **say " finished"** works just fine.

Having completed **'until** we can move on to **'while**, which, as you may suspect, will be equivalent to **[... .. while again]**.

The nest passed to **'until** takes the form **[(do some actions and then leave a boolean on the stack)]**, which we can abbreviate to **[actions test]**. With **'while** the test comes before the action, and we need to separate the actions from the test, so the nest passed to **'while** will take the form **[[actions] [test]]**, much like the nest passed to **'ifelse**.

'until does **actions test actions test** repeatedly, stopping when test returns true, so the actions are done at least once. **'while** differs in that **actions** might not be done at all. So we will start by doing **test**, and if that returns true, then we will do **actions test actions test** repeatedly until **test** returns true. Stepping through **'while** as left as an exercise for the reader, to consolidate your understanding of the ideas presented in this section.

```
[ dup dip [ 1 peek do ]
  swap
  ' [ [ nested
    ' not join
    'until ]
    [ drop ] ]
  'ifelse ]      is 'while ( [ --> )
```


We have one final exercise before we draw some conclusion from all of this, and that will be to use `'while` to create an equivalent to `times`. `times` is an all-singing, all-dancing word that comes with extras such as `i` and `i^`. We will forego those extras here, although it would not be too hard an exercise to add them if you wished. The idea is to build a nest to pass to `'while` where the test portion checks if a countdown has reached 0, and decrements it.

```
[ stack ]                is 't.countdown (--> [ ] )

[ swap 't.countdown put
  nested
  ' [ 't.countdown share
    0 >
    -1 't.countdown tally ]
  nested join
  'while
  't.countdown release ]    is 'times ( n x --> )
```

Testing...

```
/O> 3 ' [ say "It works! " ] 'times
... 0 ' [ say "This should not print at all." ] 'times
...
It works! It works! It works!
Stack empty.
```

Conclusion.

Quackery is a hybrid language; mostly it falls into the category of functional languages, of which the original was Common Lisp.

The use of a stack, which gives Quackery a Reverse Polish notation, and its control flow words, which are procedural in nature rather than functional, come from Forth, the original concatenative language.

More recent concatenative languages, such as Joy and Factor, eschew the Forth control flow style in favour of a system much more like that presented here, making them more purely functional languages, albeit with a reverse Polish notation.

My personal preference is for the Forth inspired control flow words, but if you prefer the style presented in this section, then by all means use it. Or use both. Each has its strengths. `if`, `iff`, `else`, `again`, `done` et cetera, are truly mix and match, and where a control structure calls for multiple exit points, they can be created quickly and easily. Their functional equivalents are less flexible in that respect, but having their tests and actions on the stack rather than embedding them within a nest during coding allows for an entirely different sort of flexibility. And both word sets can be extended to create new control flow words.

If nothing else, having followed this through will give you a basis for understanding some of the deepest and most bewildering (at least for me) ideas in computer science, those of the Lambda calculus in general and the Y-Combinator in particular, should you encounter them.

What's in a Name?

A summary of some of the key points of Quackery, in preparation for describing every word in detail.

A Quackery word is a sequence of printable characters that occurs in either the **builders & jobs** dictionary or the **names & actions** dictionary. Quackery words are names for Quackery objects, of which there are three types: numbers, operators, and nests. The **builders & jobs** dictionary contains the names of words that the Quackery compiler, **build**, uses to do various jobs during compilation, and the **names & actions** dictionary contains all the rest. The **builders** are:

```
[ ] is builds ( ) forward resolves char $ say hex now! constant
```

Numbers are used to represent information, either by their numerical value or by association with some other type of information. For example, 1 and 0 are used to represent the Boolean logical values **true** and **false**, and in text strings numbers indicate letters, digits, punctuation marks and suchlike.

Operators correspond to “opcodes”, the computer operations that process information; adding numbers, manipulating the stack, and so on. There are 54 operators in Quackery that provide basic functionality and an interface to the external world via the screen, keyboard and text files. Their names are:

```
fail nand = > 1+ + negate * /mod ** << >> & | ^ ~ time stacksize
nestdepth return dup drop swap rot over ]done[ ]again[ ]if[ ]iff[ ]else[
]'[ ]this[ ]do[ ]bailby[ put immovable take [] split join find poke peek
size nest? number? operator? quid emit ding input putfile releasefile
sharefile
```

(It should be noted that this selection is to some extent arbitrary, and different sets of operators could function equally well.)

Nests provide structure to Quackery. A nest is a sequence of zero or more words, numbers and nests., enclosed in brackets; []. They are how Quackery represents composite data and sequences of instructions for the Quackery processor to perform. The processor functions by traversing a nest from left to right. When the processor encounters a number it puts that number on the stack, when it encounters an operator it performs that operation, and when it encounters a nest, it traverses that nest before continuing in the nest it was previously traversing. When it gets to the end of the first nest it started traversing, the program ends. The meta control flow operators (i.e. those with names that are reverse bracketed:]...[) are used to modify the order of traversal.

Programming in Quackery consists of extending the language by adding new words to the language, and creating sequences of instructions for the processor to traverse. For example,

```
[ dup 1+ * ] is sum-of-evens ( n --> sum_of_first_n_even_numbers )

6 sum-of-evens echo
```

adds the word **sum-of-evens** to Quackery, and then uses it to calculate the sum of the first 6 even numbers and print the result on the screen.

Words, Numbers, and Nests

Words

A Quackery word is a sequence of printable characters that occurs in either the `builders & jobs` dictionary or the `names & actions` dictionary.

Numbers

Numbers in Quackery are a sequence of digits, optionally preceded by a minus sign. Numbers are integers, and decimal by default. The behaviour of a number is to put its value on the stack.

Nests

A nest is a sequence of zero or more words, numbers, and nests, separated by whitespace characters (spaces and carriage returns) that starts with the word `[` and ends with the word `]`.

Stack Management Words

```
dup drop swap rot unrot over nip tuck 2dup 2drop 2swap 2over pack unpack
dip dip.hold
```

Arithmetic Words

```
1+ + negate abs - * ** /mod / mod
```

Comparison Words

```
= oats != < > min max clamp within $< $>
```

Boolean Logic Words

```
true false not and nand or xor
```

Bitwise Logic Words

```
~ & | ^ << >> bit 64bits 64bitmask rot64
```

Pseudorandom Number Words

```
random randomise shuffle
prng prng.a prng.b prng.c prng.d initrandom
```

Ancillary Stack Words

```
stack put take release share replace move tally temp base decimal
```

Control Flow Words

```
done again if iff else until while
]done[ ]again[ ]if[ ]iff[ ]else[ ]do[ ]'[ ]this[
' do this table
recurse decurse depth
times i i^ step refresh conclude
times.start times.count times.action
```

Character and String Words

space carriage upper lower printable qacsfot digit char->n number\$ \$->n
trim nextword nest\$

Nest Editing Words

[] nested join split size peek poke pluck stuff behead of reverse reflect
copy makewith witheach with.hold

Searching and Sorting Words

matchitem findwith find findseq found sortwith sort sort\$
match.test match.nest sort.test

User Input and Output Words

input sp cr emit echo\$ wrap\$ echo ding

File Management Words

putfile takefile sharefile releasefile replacefile loadfile

Exception Handling Words

protect backup]bailby[bail bailed message
history backupwords restorewords releasewords
protected fail

To-do Stack Words

to-do new-do add-to now-do do-now not-do

Internal Representation Words

quid operator? number? nest? name? builder? immovable

Dictionary Words

names actions builders jobs namenest actiontable buildernest jobtable

Building Words

build quackery [] is builds forward resolves char \$ say constant hex
now! () unbuild quackify unresolved nesting b.nesting b.to-do

Time Words

time

Development Tool Words

empty words shell leave stacksize echostack nestdepth return return\$
echoreturn

Word Behaviours

Stack Management

`dup drop swap rot unrot over nip tuck 2dup 2drop 2swap 2over pack unpack dip dip.hold`

`dup (a --> a a)`

`dup` duplicates the ToS (top of stack), `a`. `a` can be a number, a word or a nest (as with all stack management words, unless otherwise stated.)

As with all words that require items on the stack, if there are too few items on the stack the processor will crash, reporting that the stack is unexpectedly empty.

`drop (a -->)`

`drop` removes the ToS from the stack.

`swap (a b --> b a)`

`swap` exchanges the two uppermost items on the stack.

`rot (a b c --> b c a)`

The 3oS (3rd on Stack), `a`, is brought to the ToS and the upper two items move down the stack correspondingly.

`unrot (a b c --> c a b)`

The converse operation to `rot`. ToS is moved down to 3oS, 3oS and 2oS move up the stack accordingly. `unrot` is equivalent to `rot rot`, and similarly, `unrot unrot` is equivalent to `rot`.

`over (a b --> a b a)`

`over` places a duplicate of the 2oS, `a`, on the ToS. It is equivalent to `swap dup unrot`.

`nip (a b --> b)`

`nip` removes the 2oS `a`, from the stack. It is equivalent to `swap drop`.

`tuck (a b --> b a b)`

`tuck` puts a duplicate of the ToS, `b`, underneath the 2oS. It is equivalent to `dup unrot`.

`2dup (a b --> a b a b)`

`2dup` puts duplicates of the 2oS and ToS, `a` and `b`, on the stack. It is equivalent to `over over`.

`2drop (a b -->)`

`2drop` removes the ToS and 2oS from the stack. It is equivalent to `drop drop`.

`2swap (a b c d --> c d a b)`

`2drop` exchanges ToS and 2oS with the third and fourth items on the stack.

`2over (a b c d --> a b c d a b)`

`2drop` places duplicates of the 3oS and 4oS on the stack.

`pack (* a --> [*])`

`pack` expects a number, `a`, on ToS, and that number of items on the stack beneath `a`. It replaces those items with a nest containing them. For example, if the stack contains

12 13 14 15

then

4 pack

will cause the stack to contain the nest

[12 13 14 15]

If `a` is zero or a negative number, `pack` will put an empty nest, `[]`, on the stack.

`unpack (a --> *)`

`unpack` expects a nest `a`, on the ToS. It removes the items from the nest, onto the stack. For example, if `a` is `[12 13 14 15]` then `unpack` will put 12 13 14 15 on the stack.

`dip (a --> a)`

`dip` temporarily moves the ToS, `a`, out of the way, performs the item following it, and then returns the ToS to the stack. In the illustrative shell dialogue below, `empty` empties the stack.

```
/O> 1 99 dip dup
...
```

Stack: 1 1 99

```
/O> empty 1 2 3 99 dip [ swap rot ]
...
```

Stack: 3 2 1 99

```
/O> empty 1 2 3 99 dip [ + + ]
...
```

Stack: 6 99

Note that (for example) `dip dip swap` will cause Quackery to crash. For “double dipping”, make the second `dip` and the item that follows it into a single item by putting them in a nest:

```
/O> 1 2 88 99 dip [ dip swap ]
...
```

Stack: 2 1 88 99

`dip.hold (--> a)`

`dip.hold` is the ancillary stack (see Ancillary Stack words, below) to which `dip` temporarily moves the ToS.

Arithmetic

`1+ + negate abs - * ** /mod / mod`

(The arithmetic words expect numbers on the stack unless otherwise stated.)

`1+ (a --> b)`

`1+` adds 1 to the ToS.

`+ (a b --> c)`

`c` equals `a` plus `b`.

`negate (a --> b)`

`b` equals `a` multiplied by `-1`.

`abs (a --> b)`

`b` is the absolute value of `a`. (i.e. if `a` is less than zero, negate it.)

`- (a b --> c)`

`c` equals `a` minus `b`.

`* (a b --> c)`

`c` equals `a` multiplied by `b`.

`** (a b --> c)`

`**` is the exponentiation operator. Not to be confused with `^`, which in Quackery is a bitwise logic operator.

`c` equals `a` raised to the power `b`. `b` cannot be a negative number.

`/mod (a b --> c d)`

`/mod` is division with remainder as taught at primary school, also known as Euclidian Division. "a divided by b equals c remainder d". (Forth programmers should note that it does not return the results in the same order as Forth does.)

For example, 42 divided by 10 = 4 remainder 2.

The behaviour of `/mod` with negative numbers is known as "floor division". Technically, it rounds towards negative infinity.

So `-42 10 /mod` will put `-5 8` on the stack, not `-4 2` as might be expected. The rationale for this behaviour is that `a` can be consistently recreated by multiplying `c` by `b` and adding `d`, so $(-5 * 10) + 8 = -42$, whereas $(-4 * 10) + 2 = -38$.

Similarly, `42 -10 /mod` returns `-5 -8`, and $(-5 * -10) + -8 = 42$, and `-42 -10 /mod` returns `4 -2`, and $(4 * -10) + -2 = -42$.

`/ (a b --> c)`

`/` is equivalent to `/mod drop`, so `a` divided by `b` equals `c`.

`mod (a b --> c)`

`mod` is equivalent to `/mod nip`, so `a mod b` equals `c`. (`mod` is short for “modulo”, computerese for “remainder after division by”.)

Comparison

`= oats != < > min max clamp within $< $>`

`= (a b --> c)`

`=` takes two things from the stack, `a` and `b`, and leaves 1 (i.e. `true`) on the stack if they are equivalent, and 0 (i.e. `false`) otherwise. Numbers are equivalent if they are the same number. 42 is always equivalent to 42, and never equivalent to 23. Similarly for operators, `over` is always equivalent to `over`, and never equivalent to `rot`, or to `[swap dup rot swap]`, despite this nest having the same behaviour as `over`. Pointing this out may seem overly pedantic, but it is a necessary preamble to discussing equivalency of words and nests.

If you put the word `rot` and the nest `[unrot unrot]` on the stack by using “`'`” to “quote” them (described in the Control Flow words, below) and then apply `=` to them, thus;

`' rot ' [unrot unrot] =`

the stack will have 0 (i.e. `false`) on it, because although `rot` and `[unrot unrot]` do the same thing, `rot` is the name of an operator, and an operator is never equivalent to a nest.

However, if you put the word `unrot` and the nest `[rot rot]` on the stack and apply `=` to them, thus;

`' unrot ' [rot rot] =`

the stack will have 1 (i.e. `true`) on it, because `unrot` is defined with:

`[rot rot] is unrot`

So `unrot` is the name of a nest containing two `rots`, and is therefore equivalent to another nest containing two `rots`.

However, the `shell` will show `unrot` on the stack as `unrot`, but will show the nest `[rot rot]` as `[rot rot]`, apparently different, despite their equivalence. Most of the time this distinction is unimportant. Nonetheless confusion can arise occasionally, which brings us to `oats`.

`oats (a b --> c)`

`oats` takes two things from the stack, `a` and `b`, and leaves 1 (i.e. `true`) on the stack if they are one and the same, and 0 (i.e. `false`) otherwise.

In the movie *The Matrix*, there is a scene where Neo sees a cat walk past a doorway, twice. It was important that he decide whether he saw two identical cats, one after the other, or the same cat twice. The Quackery word `=` would not be sufficient for this task but `oats` would.

If we use `dup`, `over`, `tuck` or `2dup` to make duplicates of nests on the stack, `oats` will reveal that they are not two identical nests; they are the same nest, twice.

```
/O> ' [ 23 24 25 ] dup oats
```

```
...
```

```
Stack: 1
```

However; if we create two identical looking nests, they are not one and the same.

```
/O> ' [ 23 24 25 ] ' [ 23 24 25 ] oats
```

```
...
```

```
Stack: 0
```

The key to understanding this is to know that when we say something is “on the stack” or “in a nest”, we are using imprecise language. Strictly speaking, stacks and nests contain *pointers* to Quackery objects. (Pointers are much like links on web pages. A page may contain two or more links to one page (i.e. the links point to one and the same page), or it may contain links to two or more identical pages on different websites (i.e. the links point to pages that are equivalent but not one and the same).

The majority of words in Quackery take account of this distinction, so if we use `join` from the nest editing words to append a number to a nest, it does not modify the nest, rather it creates a new nest that is the result of appending a number to the specified nest.

```
/O> ' [ 1 2 3 4 ] dup
```

```
...
```

```
Stack: [ 1 2 3 4 ] [ 1 2 3 4 ]
```

```
/O> 999 join
```

```
...
```

```
Stack: [ 1 2 3 4 ] [ 1 2 3 4 999 ]
```

The ancillary stack words (described below) of necessity do not create new instances of the stack (or nest) they are applied to, which can lead to surprising behaviour. In the following illustrative shell dialogue we use the ancillary stack word `put` rather than the nest editing word `join` to append a number to a nest, whilst keeping the original nest on the stack so that we can see that it also changes.

```

/O> ' [ 1 2 3 4 ] dup
...

Stack: [ 1 2 3 4 ] [ 1 2 3 4 ]

/O> 999 over put
...

Stack: [ 1 2 3 4 999 ] [ 1 2 3 4 999 ]

```

This behaviour can be circumnavigated by judicious use of the word **copy**, which replaces the top of stack with a new copy of itself. When the ToS is a nest, the copy is a pointer to a new nest, not one and the same as any existing nest.

```

/O> ' [ 1 2 3 4 ] dup copy
...

Stack: [ 1 2 3 4 ] [ 1 2 3 4 ]

/O> 999 over put
...

Stack: [ 1 2 3 4 ] [ 1 2 3 4 999 ]

```

For this reason is prudent to only use the ancillary stack words for managing ancillary stacks. (See **table** in the Control Flow words and **resolves** in the Builder words for exceptions to this guideline.)

Also note that **oats** may provide surprising results when applied to numbers. Internally, Python 3 represents numbers in a variety of ways depending on their size, and sometimes it conserves memory by making two numbers that are equal “one and the same”, and sometimes it does not. While the programmer is protected from any adverse effects of this optimisation, it does mean that applying **oats** to two equal numbers is not a particularly meaningful act.

Two instances of a Quackery operator, whether generated by duplication (“**dup**” et cetera.) or by quoting (“**'**”) are always one and the same. (i.e. **oats** will always return **true**.)

!= (a b --> c)

!= takes two things from the stack, **a** and **b**, and leaves 1 (i.e. **true**) on the stack if they are equivalent, and 0 (i.e. **false**) otherwise. **!=** does the same as **= not**.

< (a b --> c)

< takes two numbers from the stack, **a** and **b**, and leaves 1 (i.e. **true**) on the stack if **a** is less (closer to negative infinity) than **b**, and 0 (i.e. **false**) otherwise.

> (a b --> c)

> takes two numbers from the stack, **a** and **b**, and leaves 1 (i.e. **true**) on the stack if **a** is more (closer to positive infinity) than **b**, and 0 (i.e. **false**) otherwise.

`min (a b --> c)`

`min` takes two numbers from the stack, and leaves the smaller (closer to negative infinity) number on the stack.

`max (a b --> c)`

`max` takes two numbers from the stack, and leaves the larger (closer to positive infinity) number on the stack.

`clamp (a b c --> d)`

`clamp` takes three numbers from the stack, `a`, `b`, and `c`. If `a` is less than `b`, it leaves `b` on the stack, if `a` is larger than `c`, it leaves `c` on the stack. Otherwise (i.e. if `a` is equal to either `b` or `c`, or lies between `b` and `c`) it leaves `a` on the stack.

```
3 4 6 clamp returns 4
4 4 6 clamp returns 4
5 4 6 clamp returns 5
6 4 6 clamp returns 6
7 4 6 clamp returns 6
```

`within (a b c --> d)`

`within` takes three numbers from the stack, `a`, `b` and `c`, and leaves 1 (`true`) on the stack if `a` is within the range `b` to 1 less than `c`, inclusive (i.e., `a` is larger than or equal to `b`, and smaller than `c`), and 0 (`false`) otherwise.

```
3 4 6 within returns false
4 4 6 within returns true
5 4 6 within returns true
6 4 6 within returns false
7 4 6 within returns false
```

`$< (a b --> c)`

`$<` takes two strings from the stack, `a` and `b`, and leaves 1 (i.e. `true`) on the stack if `a` comes before `b` according to its QACSFOT ordering (see `qacsfot`, in the Character and String words, below) and 0 (i.e. `false`) otherwise. QACSFOT ordering is similar to dictionary ordering, in that the strings are compared character by character from left to right, until a difference is found. Spaces and carriage returns are considered identical and come before the numbers 0 to 9. Then come the letters, in the order `AaBbCc` and so on, and other characters come after letters.

These are in QACSFOT order: 0, 1, 10, 2, 20, Bat, bat, caT, cat, catch, \$#!%!

`$> (a b --> c)`

The same as `$<`, except `true` if `a` comes after `b`, and `false` otherwise.

Boolean Logic

true false not and nand or xor

true (--> 1)

true puts the number 1 on the stack.

false (--> 0)

false puts the number 0 on the stack.

not (a --> b)

not expects a number, a, on the stack. It replaces it with 1 if a is 0, and with 0 otherwise. As with all the Boolean logic words, it treats arguments of 0 as false and other numbers as true.

and (a b --> c)

and expects two numbers, a and b, on the stack. It replaces them with 1 if neither a nor b are 0, and with 0 otherwise.

nand (a b --> c)

nand expects two numbers, a and b, on the stack. It replaces them with 0 if neither a nor b are 0, and with 1 otherwise. nand is equivalent to and not.

or (a b --> c)

or expects two numbers, a and b, on the stack. It replaces them with 0 if both a and b are 0, and with 1 otherwise.

xor (a b --> c)

xor expects two numbers, a and b, on the stack. It replaces them with 1 if one of a and b is 0 and the other is not 0, and with 0 otherwise.

Other Boolean Operations

If required, material equivalence ($a \Leftrightarrow b$), can be defined as:

[xor not] is <=> (a b --> c)

(This is the same as =, except that it expects two numbers on the stack and will report a problem if it finds a nest or an operator.)

If required, material implication ($a \Rightarrow b$) can be defined as:

[swap not or] is implies (a b --> c)

If required, nor can be defined as:

[or not] is nor (a b --> c)

Bitwise Logic

`~ & | ^ << >> bit 64bits 64bitmask rot64`

The bitwise logic words operate on the internal binary representations of numbers. To illustrate their behaviours we will define the word `echo8bits` to display the least significant 8 bits of a number on the screen. So `0` will be displayed as `00000000`, `1` as `00000001`, `2` as `00000010`, `3` as `00000011`, `4` as `00000100`, all the way to `255` (i.e. hex `FF`) as `11111111`. The word `echoline` will display “-----” on the screen.

```
[ 8 times
  [ 2 /mod swap ]
  drop
  8 times echo ]      is echo8bits ( n --> )

[ char - 8 of echo$ ] is echoline (  --> )
```

Also, we will use the ancillary stack base and the builders word `now!` to temporarily change Quackery's default base to binary and return it to decimal during compilation.

```
/O> [ 2 base put ] now!
... 00101010 echo8bits
... [ base release ] now!
...
00101010
```

`~ (a --> b)`

`~` is bitwise **not**. It takes a number from the stack and inverts the bits, so `0`s become `1`s and vice versa.

```
/O> [ 2 base put ] now!
... 11010101 dup echo8bits cr
... [ base release ] now!
... echoline cr
... ~ echo8bits cr
...
11010101
-----
00101010
```

`& (a b --> c)`

`&` is bitwise **and**. It takes two numbers from the stack and returns the result of performing **and** on all of their bits.

```
/O> [ 2 base put ] now!
... 10111110 dup echo8bits cr
... 01101010 dup echo8bits cr
... [ base release ] now!
... echoline cr
... & echo8bits cr
...
10111110
01101010
-----
00101010
```

| (a b --> c)

| is bitwise **or**. It takes two numbers from the stack and returns the result of performing **or** on all of their bits.

```
/O> [ 2 base put ] now!  
... 00001010 dup echo8bits cr  
... 00100010 dup echo8bits cr  
... [ base release ] now!  
... echoline cr  
... | echo8bits cr  
...  
00001010  
00100010  
-----  
00101010
```

^ (a b --> c)

^ is bitwise **xor**, not to be confused with raising a number to a power, which is ******. It takes two numbers from the stack and returns the result of performing **xor** on all of their bits.

```
/O> [ 2 base put ] now!  
... 00001111 dup echo8bits cr  
... 00100101 dup echo8bits cr  
... [ base release ] now!  
... echoline cr  
... ^ echo8bits cr  
...  
00001111  
00100101  
-----  
00101010
```

<< (a b --> c)

<< is logical shift left. It takes two numbers from the stack, **a** and **b** and returns the result shifting the bits of **a** by **b** places to the left, inserting 0s in the **b** empty least significant bits.

```
/O> [ 2 base put ] now!  
... 00010101 dup echo8bits cr  
... echoline cr  
... [ base release ] now!  
... 2 << echo8bits cr  
...  
00010101  
-----  
01010100
```

>> (a b --> c)

<< is logical shift right. It takes two numbers from the stack, **a** and **b** and returns the result shifting the bits of **a** by **b** places to the right, discarding the **b** least significant bits.

```
/O> [ 2 base put ] now!  
... 10101001 dup echo8bits cr  
... [ base release ] now!  
... echoline cr  
... 2 >> echo8bits cr  
...  
10101001  
-----  
00101010
```

bit (a --> b)

bit expects a non-negative number, **a**, and returns a number, **b**, with all its bits set to 0 except for the **a**th bit, which is set to 1. Bits are counted from the least significant, which is bit 0. So 0 **bit** returns (in binary) 1, 1 **bit** returns 10, 2 **bit** returns 100 and so on.

The remaining bitwise logic words, **64bits**, **64bitmask**, and **rot64**, are included as they are used by the Pseudorandom Number words, but may feasibly be of use elsewhere.

64bits (a --> b)

64bits expects a number, **a**, and returns a number, **b**, equal to **a** with the 64 least significant bits unchanged, and all higher bits being set to 0. To put it another way, **64bits** masks out all the bits to the left of the 63rd bit. (Counting starts with the 0th bit.)

64bitmask (--> a)

64bitmask puts the binary number
111 on the
stack. (That's sixty four 1s.)

rot64 (a b --> c)

rot64 takes two numbers, **a** and **b**, from the stack. **b** should be in the range 0 to 64 inclusive. **a** should be in the range 0 and the number returned by **64bitmask**, inclusive. **a** will be masked to force it into that range. If **b** is equal to 0 or 64, then **c** is equal to masked **a**, otherwise **c** is equal to masked **a** with the most significant **b** bits moved down to be the **b** least significant bits, and the higher bits moved up accordingly. So, as hex **DEFACEABADFACADE** is a 64 bit number and the hexadecimal digits **DEFACE** occupy the 24 (decimal) most significant bits,

hex **DEFACEABADFACADE** 24 **rot64**

will leave the hexadecimal number **ABADFACADEDEFACE** on the stack.

Pseudorandom Numbers

```
random randomise shuffle
prng prng.a prng.b prng.c prng.d initrandom
```

The Pseudorandom Number words are adapted from the 64 bit variant of “A Small Noncryptographic PRNG” by Bob Jenkins, which is in the public domain. The code and notes about it can be found at <http://burtleburtle.net/bob/rand/smallprng.html>.

`random (a --> b)`

`random` expects a number, `a`, in the range 0 to $2^{64}-1$ inclusive, and returns a pseudorandom number `b` in that range. The numbers that it returns are (believed to be) evenly distributed within the range, and part of a very long cycle.

`randomise (-->)`

`randomise` reinitialises the PRNG (PseudoRandom Number Generator) with a different and (realistically speaking) unpredictable number each time it is called. This word should be used before a series of numbers are generated by `random` if different results are desired each time. One invocation of `randomise` is sufficient each time Quackery is used.

`shuffle (a --> b)`

`shuffle` expects a nest, `a`, and uses `random` to create a new nest, `b`, with the same contents as `a`, but in a different order. For example:

```
/O> ' [ 0 1 2 3 4 [ 5 6 7 ] ] shuffle
...

Stack: [ 4 3 2 [ 5 6 7 ] 0 1 ]

/O> shuffle
...

Stack: [ 0 2 1 4 [ 5 6 7 ] 3 ]
```

Note that nests within the shuffled nest are not shuffled.

The rest of the Pseudorandom Number words are intended for use only by `random` and `randomise`, but briefly, `prng (--> a)` returns a pseudorandom number in the range 0 to $2^{64}-1$ inclusive, `prng.a prng.b prng.c prng.d` are ancillary stacks used by `prng`, and `initrandom (n -->)` is used to kickstart the PRNG, initially from a specified number, and by `randomise` from an unpredictable number returned by `time`.

Ancillary Stacks

stack put take release share replace move tally temp base decimal

For discussion of using the ancillary stack words on Quackery objects other than ancillary stacks, see the descriptions of **table** and of the comparison words **=** and **oats**.

stack (--> a)

stack is used to make an ancillary stack. It should be the first word of a nest. It returns **a**, a pointer to that nest for use by the ancillary stack words **put**, **take**, **share**, **replace**, **release**, **move**, and **tally**. Any items in the nest after **stack** are not performed, they are the initial contents of the ancillary stack. For example,

```
[ stack [ 52 50 ] ] is example1
```

creates an ancillary stack called **example1** with one item on it, the nest [52 50].

put (a b -->)

put transfers an item, **a**, from the Quackery stack to an ancillary stack, **b**.

Continuing with the example, after doing

```
' [ 68 78 65 ] example1 put
```

the ancillary stack **example1** will contain the two items

```
[ 52 50 ] [ 68 78 65 ]
```

with [68 78 65] being the top of the ancillary stack.

take (a --> b)

take transfers the top item, **b**, of an ancillary stack, **a**, to the Quackery stack.

Continuing with the example, after doing

```
example1 take
```

the ancillary stack **example1** will contain [52 50] as its top of stack, and [68 78 65] will be on the top of the Quackery stack.

release (a -->)

release removes the top item of an ancillary stack, **a**. It is equivalent to **take drop**.

share (a --> b)

share puts a duplicate of the top item, **b**, of an ancillary stack, **a**, on the Quackery stack, without removing it from the ancillary stack. It is equivalent to

```
dup take dup rot put
```

`replace (a b -->)`

`replace` replaces the top item of ancillary stack `b` with `a`. It is equivalent to

`dup release put`

`move (a b -->)`

`replace` transfers the top item of ancillary stack `a` to ancillary stack `b`. It is equivalent to: `a take b put`.

The following Quackscript `moves` three numbers from ancillary stack `a` to ancillary stack `b`, following the rules of the Tower of Hanoi game, (i.e a larger number can never be on top of a smaller number) and then prints the ancillary stack `b` using `copy echo`.

```
/O> [ stack 3 2 1 ] is a
... [ stack ] is b
... [ stack ] is c
... a b move
... a c move
... b c move
... a b move
... c a move
... c b move
... a b move
... b copy echo
...
[ stack 3 2 1 ]
Stack empty.
```

`tally (a b -->)`

`tally` expects a number `a`, and an ancillary stack, `b`, with a number as its top of stack. It adds the number `a` to the top of ancillary stack `b`. `tally` could be used, for example, to maintain a running total without cluttering up the Quackery stack.

`temp (--> a)`

`temp` is an ancillary stack used by various predefined Quackery words. All the predefined words that use `temp` observe the convention of only `take`-ing as many items as they `put`, and not using any of the ancillary stack words `take`, `release`, `share`, or `replace` without first `put`-ting, so `temp` is available for other uses.

`base (a -->)`

`base` is an ancillary stack used by Quackery to determine the current base for conversion between numbers and characters, and numbers and strings. The default base is decimal. Note that `build` overrides the current default base and sets it to decimal. For a demonstration of changing `base` whilst `building`, see the description of the bitwise logic words. `base` should be between 2 (binary) and 36 (hexatridecimal) inclusive.

`decimal (-->)`

`decimal` overrides the current base and sets it to decimal. It is equivalent to `10 base put`, so should be followed by `base release` to observe stack conventions.

Control Flow

```
done again if iff else until while
]done[ ]again[ ]if[ ]iff[ ]else[ ]'[ ]do[ ]this[
' do this table
recurse decurse depth
times i i^ step refresh conclude
times.start times.count times.action
```

Structured Control Flow

```
done again if iff else until while
```

When the Quackery processor performs a nest of Quackery code, it starts at the `[` and proceeds one item at a time until it ends at the `]`, unless it encounters a word that modifies the flow of control by jumping to the start or the end of the nest, or by skipping over one or two items. Conditional jumps and skips are dependent on the item on the top of the Quackery stack. The structured control flow words can be used without restrictions of order or number within a nest.

```
done ( --> )
```

`done` causes a jump to the end of the nest. (Actually, it exits the nest immediately, which has the same effect as jumping to the end of the nest but is a smidgeon faster.)

```
/O> [ $ 'Print this once.' echo$
... done
... $ 'Do not print this.' echo$ ]
...
Print this once.
```

```
again ( --> )
```

`again` causes a jump to the start of the nest.

```
[ cr $ 'Print this forever.' echo$ again ]
```

will print line after line of “Print this forever.” on the screen until the computer stops working or someone presses Control C, whichever happens first.

```
if ( a --> )
```

`if` will cause the processor to skip over the next item if `a` is 0 (i.e. false).

```
/O> true if [ $ 'true' echo$ ]
... false if [ $ 'false' echo$ ]
...
true
```

```
iff ( a --> )
```

`iff` will cause the processor to skip over the next two items if `a` is 0 (i.e. false).

```
/O> false true true true
... [ iff [ $ 'true ' echo$ ] again ]
...
true true true
```

else (-->)

else will cause the processor to skip over the next item.

```
/O> true  iff [ $ 'true'  echo$ ]
...      else [ $ 'false' echo$ ]
...
true

/O> false iff [ $ 'true'  echo$ ]
...      else [ $ 'false' echo$ ]
...
false
```

until (a -->)

until will cause a jump to the start of the nest if **a** is zero (i.e. **false**.) It is equivalent to **not if** again.

```
/O> true false false false
... [ dup
...   iff [ $ 'true '  echo$ ]
...   else [ $ 'false ' echo$ ]
...   until ]
...
false false false true
```

while (a -->)

while will causes a jump to the end of the nest if **a** is zero (i.e. **false**.) It is equivalent to **not if done**.

```
/O> false true true true
... [ while $ 'true ' echo$ again ]
...
true true true
```

Meta Control Flow

`]done[]again[]if[]iff[]else[]'[]do[]this[`

While **while** is equivalent to **not if done**, it cannot be defined as

```
[ not if done ] is while
```

because the **done** will cause a jump to the end of the nest that it occurs in, rather than the nest in which the **while** occurs. The definition of **while** is

```
[ not if ]done[ ] is while ( b --> )
```

where `]done[` can be thought of as granting the “jump to end of nest” behaviour to **while**. Similarly, **done**, **again**, **if**, **iff**, **else**, and **until** are defined with:

```
[ ]done[ ]      is done (  --> )
[ ]again[ ]     is again (  --> )
[ ]if[ ]        is if    ( a --> )
[ ]else[ ]      is else  (  --> )
[ not if ]again[ ] is until ( a --> )
```

The rest of the meta control flow words are illustrated in the following descriptions, with the exception of `]bailby[`, which is illustrated in the exception handling words section.

Deferment

```
' do this table
```

“Saying and doing are two things.”

Matthew Henry, 1662–1714

(Also see the `To-do Stack` section for another sort of deferment that uses `'` and `do`.)

```
' ( --> a )
```

`'` “quotes” the item that follows it, which is to say that the word, nest or number after `'` is put on the stack rather than being performed.

```
/O> ' 21
```

```
... ' [ dup + ]
```

```
...
```

```
Stack: 21 [ dup + ]
```

Note that in this example, quoting the number 21 is redundant, since the behaviour of numbers is to put their value on the stack. We might describe this behaviour by saying that “numbers quote themselves”.)

`'` acquires its behaviour from `]'`. It is defined with: `[]'[] is '`.

```
do ( a --> )
```

`do` undoes the behaviour of `'`; it takes the ToS, `a`, and performs it.

```
Stack: 21 [ dup + ]
```

```
/O> do
```

```
...
```

```
Stack: 42
```

Working together, `'` and `do` give deferment; the ability to say something now and do it later.

Note that `doing` a number is also redundant – numbers `do` themselves.

```
/O> 42 do
```

```
...
```

```
Stack: 42
```

`do` acquires its behaviour from `]do`. It is defined as:

```
[ ]do[ ] is do
```

```
this ( --> a )
```

this puts the nest it is in on the stack.

```
/O> 3 times [ this echo sp ]
...
[ this echo sp ] [ this echo sp ] [ this echo sp ]
Stack empty.
```

this acquires its behaviour from]this[. It is defined as:

```
[ ]this[ ] is this
```

```
table ( a --> b )
```

table takes a number, a, from the stack, and leaves a Quackery object, b, on the stack in its place. Like stack, it needs to be the first item of a nest. The object b, is the ath object following table in the nest, counting from 0. If a is negative, it returns the -ath item counting backwards from the end of the nest, with the end item numbered -1. So:

```
0 [ table 10 11 12 ] would leave 10 on the stack,
1 [ table 10 11 12 ] would leave 11 on the stack,
-1 [ table 10 11 12 ] would leave 12 on the stack, and
-2 [ table 10 11 12 ] would leave 11 on the stack.
```

Used in conjunction with do, table provides multiway branching behaviour illustrated here with the word monthdays.

divides returns true if the number x is exactly divisible by the number y and false otherwise.

```
[ mod 0 = ] is divides ( x y --> b )
```

leap returns true (i.e. 1) if the number y is a leap year number, and false (i.e. 0) otherwise.

```
[ dup 400 divides iff
  [ drop true ] done
  dup 100 divides iff
    [ drop false ] done
  4 divides ] is leap ( y --> b )
```

monthdays takes a year (y) number and month (m) number (counting from January as 1), and returns n, the number of days in the specified month.

```
[ 1 -
  [ table
    31 [ dup leap 28 + ]
    31 30 31 30 31 31 30
    31 30 31 ]
  do nip ] is monthdays ( y m --> n )
```

Taking April 1970 as an example, (i.e. 1970 4 monthdays) table will replace the 3 on the ToS with 30, do will replace that with 30, (taking advantage of the redundancy of doing numbers) and finally nip will remove the year number (1970) from 2oS, leaving 30 on the ToS.

Taking February 2020 as an example, (i.e. 2020 2 monthdays) **table** will replace the 1 on the ToS with the nest [dup leap 28 +], **do** will do that nest, (i.e. **dup** will duplicate the year number, **leap** will remove the uppermost duplicate and replace it with 1, as 2020 is a leap year, which will be added to 28 by +, giving 29 as the ToS) and finally **nip** will remove the year number (2020) from 2oS, leaving 29 on the ToS.

As with **stack**, **table** uses **]this[** to refer to its enclosing nest, and **]done[** to jump over the other items in the nest.

tables can be extended dynamically using the ancillary stack word **put**, which will add an extra item at the end of the table.

```
[ table 10 11 12 13 ] is example
' example 14 put
```

is equivalent to

```
[ table 10 11 12 13 14 ] is example
```

and in general the ancillary stack words operate on **tables** in the same manner as they operate on ancillary stacks. For discussion about creating other words that can be modified using ancillary stack words, see **immovable** in the notes on internal representation words below.

Recursion

```
recurse decurse depth
```

```
recurse ( --> )
```

recurse causes its enclosing nest to **do** itself. It is equivalent to **this do** and is defined as:
]this[do] is recurse

For example, the naive recursive Fibonacci function:

```
[ dup 2 < if done
  dup 1 - recurse
  swap 2 - recurse
  + ] is fibonacci ( a --> b )
```

Taking it line by line,

dup 2 < if done	if a is 0 or 1, then b is 0 or 1 respectively, otherwise
dup 1 - recurse	find the Fibonacci number of a-1 and
swap 2 - recurse	the Fibonacci number of a-2 and
+	add them together.

`decurse (-->)`

`decurse` has the same behaviour as `recurse`, except that is limited in the number of recursive calls it can perform by the number on the top of the ancillary stack `depth`, which tracks the number of remaining recursive calls available to it.

```
/O> 10 depth put
... [ depth share echo sp
...   decurse
...   depth share echo sp ]
... depth release
...
10 9 8 7 6 5 4 3 2 1 1 2 3 4 5 6 7 8 9 10
```

If `depth` is set to `-1`, `decurse` behaves in the same way as `recurse`, i.e. without limit, and the current depth of recursion can be determined with `depth share 1+ negate`.

For recursive structures that require more than a single enclosing nest, see the building words `forward` and `resolves`.

`depth (--> a)`

`depth` is a system ancillary stack used by `decurse`, as described above.

Iteration

```
times i i^ step refresh conclude
times.start times.count times.action
```

`times (a -->)`

`times` repeats the behaviour of the item that follows it `a` times, unless modified with `step`, `refresh` or `conclude`. If `a` is less than 1, it is repeated 0 times.

```
/O> [ times [ char * emit ] ] is stars ( n --> )
... 5 stars
...
*****
```

`dice` returns the result of rolling a six sided dice `a` times and adding the dots. It is used in the code illustrating `refresh` and `conclude` below.

```
[ 0 swap
  times
  [ 6 random 1+ + ] ] is dice ( a --> b )
```

`i (--> a)`

While `times` is repeating, `i` returns a number indicating the number of repeats remaining after the current iteration, if the step size is 1. (See `step`.)

```
/O> 10 times [ i echo sp ]
...
9 8 7 6 5 4 3 2 1 0
```


`i^ (--> a)`

While `times` is repeating, `i^` returns a number indicating the number of repeats performed prior to the current iteration, if the step size is 1. (See `step`.)

```
/O> 10 times [ i^ echo sp ]
...
0 1 2 3 4 5 6 7 8 9
```

`step (a -->)`

While `times` is repeating, `step` specifies the step size as `a`, which affects `i` and `i^` as illustrated. The default step size is 1.

Be wary of the hidden “gotcha”. `times` maintains a countdown from the initial value passed to it down to zero. It decrements this countdown at the start of the loop. `step` works by adjusting the countdown, taking the decrement of the countdown at the start of the next iteration into account, which is why it affects the behaviour of `i` and `i^` immediately.

```
/O> 10 times [ i echo sp 2 step ] cr
... 10 times [ i^ echo sp 2 step ]
...
9 7 5 3 1
0 2 4 6 8

/O> 10 times [ 2 step i echo sp ] cr
... 10 times [ 2 step i^ echo sp ]
...
8 6 4 2 0
1 3 5 7 9

/O> 1024 times [ i^ 1+ dup step echo sp ]
...
1 2 4 8 16 32 64 128 256 512 1024
```

`refresh (-->)`

While `times` is repeating, `refresh` resets the iteration countdown to the number originally passed to `times`. As with `step`, the behaviour of `i` and `i^` is affected immediately.

`sixes (a -->)` rolls one dice until a number of sixes have been rolled consecutively.

```
[ times
  [ 1 dice
    dup 6 != if
      refresh
    echo sp ] ] is sixes ( a --> )

/O> 5 times [ 2 sixes cr ]
...
6 5 6 6
1 1 4 6 1 4 1 4 4 6 1 5 6 3 5 1 3 3 1 5 3 3 6 6
5 5 5 2 3 5 4 5 1 4 5 3 3 5 1 3 3 4 2 3 3 5 1 2 6 6
2 2 5 2 4 4 2 6 6
5 4 5 1 1 6 2 6 6
```

(This example was chosen for brevity. Typically it takes longer than this to roll two sixes.)

`conclude (-->)`

While `times` is repeating, `conclude` sets the iteration countdown to zero. As with `step`, the behaviour of `i` and `i^` is affected immediately.

`snakeeyes (a -->)` rolls two dice at most `a` times, but ends prematurely if snake eyes (two ones) is rolled.

```
[ times
  [ 2 dice
    dup 2 = if
      conclude
    echo sp ] ] is snakeeyes ( a --> )

/O> 5 times [ 20 snakeeyes cr ]
...
10 4 8 4 7 10 10 4 9 6 10 8 11 7 8 12 9 7 3 10
6 6 3 5 11 5 8 8 8 6 5 4 8 9 4 10 6 9 11 2
4 5 4 8 5 7 12 3 11 10 10 8 4 3 5 7 8 4 5 7
2
6 10 5 6 3 8 9 3 2
```

`times.start`, `times.count`, and `times.action` are ancillary stacks used by the iteration words.

Characters and Strings

`space carriage upper lower printable qacsfot digit char->n number$ $->n trim nextword nest$`

Quackery recognises the following printable characters

```
0123456789AaBbCcDdEeFfGgHhIiJjKkLlMmNnOoPpQqRrS
sTtUuVvWwXxYyZz()[\]<>~+=-*/^`|_.,;?!'""%&#&$
```

and represents them by their Unicode code points (i.e. the numbers 33 to 126 inclusive.)

The space character is represented by the number 32, and the number 13 (i.e. carriage return) represents whichever character causes the start of a new line of text in the host operating system.

All other characters are converted to the number 63 (i.e. a question mark) when inputted to Quackery from the keyboard or text files, and all other numbers are converted to a question mark (i.e. code point 63) when output as characters from Quackery to the screen or to a text file.

Strings of characters are represented as nests of numbers, so most of the nest editing words work with strings. The searching and sorting words **find** and **sort\$** work with strings. The user input and output words with the exception of **ding** work with characters or strings. The file management words work with strings. The building words **char** and **\$** put a character and a string on the stack respectively.

The words described in this section are specific to handling characters and strings, and do not fall into other categories.

carriage (--> 13)

carriage puts the number that represents a new line or carriage return on the stack.

Incidentally, as with other words that act as numerical constants, it is defined as **[13] is carriage**, rather than **13 is carriage**, which would also work, so that the decompiler (unbuild) gives a more meaningful decompilation. This can be demonstrated in the shell, which displays the stack by decompiling its contents.

```
/O> 23 is twentythree
... [ 42 ] is fortytwo
... ' twentythree
... ' fortytwo
...
```

Stack: 23 fortytwo

space (--> 32)

space puts the number representing the space character on the stack.

upper (a --> b)

upper takes a character, **a**, from the stack and puts a character, **b**, on the stack. If **a** is a lower case (minuscule) letter, **b** is its upper case (majuscule) equivalent, otherwise **b** is the same as **a**.

lower (a --> b)

lower takes a character, **a**, from the stack and puts a character, **b**, on the stack. If **a** is an upper case (majuscule) letter, **b** is its lower case (minuscule) equivalent, otherwise **b** is the same as **a**.

printable (a --> b)

printable expects a character **a** on the stack, and returns **false** if it is **space** or **carriage**, and **true** if it is one of the printable characters listed above.

qacsfot (a --> b)

qacsfot maps a character **a** onto the Quackery Arbitrary Character Sequence For Ordered Text, (QACSFOT) which is slightly less arbitrary than the Unicode code points for the set of characters that Quackery recognises. For non-printable and unrecognised characters it returns 0, for printable characters it returns a number **b** in the range 1 to 94 according to its position in QACSFOT, which is:

```
0123456789AaBbCcDdEeFfGgHhIiJjKkLlMmNnOoPpQqRrS
sTtUuVvWwXxYyZz()[\]<>~=-+*/^\\_.,:;!'"`%&#&$
```

`digit (a --> b)`

`digit` takes a number `a` from the stack and if `a` is in the range 0 to 35 inclusive, puts the corresponding alphanumeric digit on the stack. The numbers 0 to 9 correspond to the characters “0” to “9”, and the numbers 10 to 35 correspond to the characters “A” to “Z”.

`char->n (a --> b)`

`char->n` takes an alphanumeric character `a` from the stack, and returns a number `b` corresponding to `a` as per `digit`. `char->n` treats lower case letters as equivalent to upper case letters. If `a` is not a valid digit in the current base, `char->n` returns -1 to indicate an unsuccessful conversion.

`number$ (a --> b)`

`number$` takes a number `a` and converts it to its string representation `b` in the current base.

`$->n (a --> b c)`

`$->n$` takes a string `a` and attempts to convert it to a number in the current base. If the conversion is successful `c` will be `true` and the number `b` will be the numerical value of `a`. If the conversion is not successful `c` will be `false` and the number `b` will be the numerical value of `a` up to the character in the string `a` that was not a valid digit in the current base.

`trim (a --> b)`

`trim` takes a string `a` and returns a string `b`, the same as string `a` but with leading whitespace characters (i.e. not `printable` characters) removed.

`nextword (a --> b c)`

`nextword` takes a string `a` which should have had any leading whitespace removed with `trim`, and returns two strings, `b` and `c`, where `c` is a string of the characters before the first whitespace character in `a`, and `b` is the rest of the string `a`.

```
/O> $ "    first    and the rest"
... trim nextword echo$ cr
... echo$
...
first
    and the rest
```

`nest$ (a --> b)`

`nest$` takes a string, `a`, and turns it into a nest of strings, `b`, by stripping out extraneous whitespace and separating each word in the string into its own nest.

Nest Editing

`[]` nested join split size peek poke pluck stuff behead of reverse reflect
copy makewith witheach with.hold

`[] (--> a)`

`[]` creates an empty nest.

`nested (a --> b)`

`nested` creates a nest `b` and puts `a` in it.

`join (a b --> c)`

`join` creates a nest `c` which is the concatenation of `a` and `b`.

```
/O> ' [ 1 2 ] ' [ 3 4 ] join echo cr ( a and b are nests )
... 12 34                join echo cr ( neither are nests )
... ' [ 1 2 ] 34         join echo cr ( only a is a nest )
... 12 ' [ 3 4 ]        join echo cr ( only b is a nest )
...
[ 1 2 3 4 ]
[ 12 34 ]
[ 1 2 34 ]
[ 12 3 4 ]
```

`split (a b --> c d)`

`split` expects a nest `a` and a number `b`, and returns two nests, `c` and `d` formed by dividing nest `a` in two at position `b`.

0	1	2	3	4	5	6	Zero and positive positions.
↓	↓	↓	↓	↓	↓	↓	
[100	101	102	103	104	105]
↑	↑	↑	↑	↑	↑	↑	
-7	-6	-5	-4	-3	-2	-1	Negative positions.

When position `b` is at the start of nest `a` (i.e. `0` or `-7` above) nest `c` will be an empty nest and nest `d` will be the same as nest `a`.

When position `b` is at the end of nest `a` (i.e. `6` or `-1` above) nest `c` will be the same as nest `a` and nest `d` will be an empty nest.

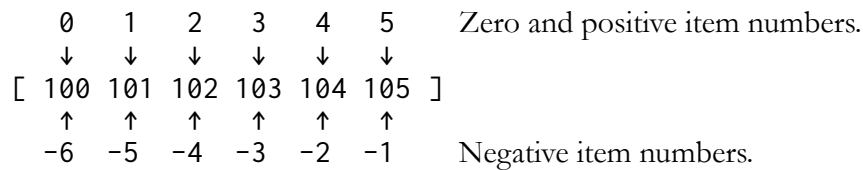
`size (a --> b)`

`size` expects a nest, `a`, and returns the number of items it contains, `b`. Nests within `a` count as a single item.

```
/O> ' [ 0 1 2 [ 30 31 32 ] 4 ] size
...
Stack: 5
```

peek (a b --> c)

peek expects a nest **a** and a number **b**, and returns the **b**th item of **a**. **a** is not modified.



poke (a b c --> d)

poke expects an item **a**, a nest **b**, and a position **c**, and returns a nest **d**, which is identical to **b** except that the item at position **c** has been replaced with item **a**.

pluck (a b --> c d)

pluck expects a nest **a** and a number **b**, and returns **c**, a copy of **a** with the **b**th item removed, and **d**, the **b**th item of **a**. Item numbers are as illustrated in **peek**.

```
/O> ' [ 10 11 12 13 ] 2 pluck
...
Stack: [ 10 11 13 ] 12
```

stuff (a b c --> d)

stuff takes an item, **a**, a nest **b**, and a number **c**, and returns a copy of **b** with **a** inserted into the **c**th position. Position numbers are as illustrated in **split**.

```
/O> 999 ' [ 0 1 2 3 ] 2 stuff
...
Stack: [ 0 1 999 2 3 ]
```

behead (a --> b c)

behead takes a nest, **a**, and returns a nest, **b**, a copy of **a** with the 0th item removed, and **c**, the 0th item of **a**.

```
/O> ' [ 10 11 12 13 ] behead
...
Stack: [ 11 12 13 ] 10
```

of (a b --> c)

of creates a nest, **c**, made of **b** copies of **a** joined together.

```
/O> ' dup 3 of
...
Stack: [ dup dup dup ]

/O> ' [ 3 swap ] 2 of
...
Stack: [ 3 swap 3 swap ]
```

`reverse (a --> b)`

`reverse` takes an item `a` from the stack and if it is a nest, returns a reversed copy. `b`. If `a` is number or an operator, `b` is the same as `a`.

```
/O> ' [ rot [ 3 tuck ] ] reverse echo
...
[ [ 3 tuck ] rot ]
```

Note that the nest within the nest, `[3 tuck]`, is not reversed.

`reflect (a --> b)`

`reflect` is like `reverse`, except that it digs down into nested nests. It takes an item `a` from the stack and if it is a nest, returns an enthusiastically reversed copy, `b`. If `a` is number or an operator, `b` is the same as `a`.

```
/O> ' [ rot [ 3 tuck ] ] reflect echo
...
[ [ [ over swap ] 3 ] rot ]
```

Note that not only did `reflect` reverse the nested nest `[3 tuck]`, it dug down into the definition of `tuck` (`[swap over]`) and reversed a copy of that too.

While this may suggest itself as a way to see the sequence of operators involved in a word (for example, `' clamp reflect reflect` reveals that `clamp` is equivalent to `[rot [[over over] > []if[] swap drop] [[over over] [swap >] []if[] swap drop]]`) it comes with a caveat:

Nests which are named using `forward` `is` and `resolves` are mostly defined that way so that they can include themselves, either directly or indirectly. (Science fiction fans are directed to the Doctor Who minisode “Space”, where the TARDIS materialises inside itself, for an analogy.) This situation will not be detected by `reflect`, or other Quackery words that dig down into nests within nests, leading to a situation where nothing appears to be happening as the processor is trapped in an endless loop where it is likely to be filling up memory with increasingly large nests. Pressing Control C will resolve this impasse by forcibly terminating both Quackery and the host language.

It is not possible to create such self-containing nests using the nest editing words alone, but it is quite easy to `put` an ancillary stack on itself. Here we `put` the ancillary stack `temp` on itself twice, then use `copy echo` to see the contents of `temp`.

```
/O> temp temp put
... temp temp put
... temp copy echo
...
[ stack temp temp ]
```

Finally we use `put` to put an empty nest inside itself, and allow the `shell` to display it by leaving the result on the stack. Unlike `echo`, the `shell` limits the depth of nesting when it shows the stack (indicated by “...”) so it does not get stuck in an endless loop.

```
/O> [] dup dup put
...
```

Stack: [[[[[...]]]]]

The following word demonstrates the use of nest editing words to construct nests with specific behaviours. The basic idea, that one can `join` two nests of `do`-able Quackery code to make a new nest of Quackery code is quite straightforward. Practical examples can be somewhat less straightforward.

`makewith (a --> b)`

`makewith` takes an item, `a`, from the stack, and constructs a nest, `b`, around it. The behaviour of `b` is to apply the behaviour of `a` to each item of a nest that it finds on the stack.

```
[ nested
  ' [ dup with.hold put
    size times ]
  ' [ with.hold share
    i^ peek ]
  rot join nested join
  ' [ with.hold release ]
    join ]          is makewith ( a --> b )

/O> 0 ' [ 1 2 3 4 5 ]
... ' + makewith
... dup echo
... do
...
[ dup with.hold put size times [ with.hold share i^ peek + ]
with.hold release ]
Stack: 15
```

Taking the `shell` interaction line by line:

```
0 ' [ 1 2 3 4 5 ]
```

This is setting up the stack for the demonstration. The plan is to construct a nest that takes the ToS, `[1 2 3 4 5]`, and adds each item in turn to the 2oS, `0`, leaving the sum of the numbers in the nest on the stack.

```
' + makewith
```

Construct a “with” nest around the “+” operator,

```
dup echo
```

and `echo` it to the screen to show what `makewith` constructed,

```
do
```

and demonstrate that it works by applying it to the items on the stack.

The nest that `makewith` constructed, (laid out nicely.)

```
[ dup with.hold put
  size times
  [ with.hold share
    i^ peek + ]
  with.hold release ]
```

The sum of 1, 2, 3, 4, and 5.

```
15
```



```
witheach ( a --> )
```

witheach takes the item following it and applies its behaviour to each item in the nest, **a**, that it expects on the stack. Continuing with the example from **makewith** of finding the sum, **b**, of the numbers in a nest of numbers, **a**, **sum** can be defined as,

```
[ 0 swap witheach + ] is sum ( a --> b )
```

and testing it in the shell...

```
/O> ' [ 1 2 3 4 5 ] sum echo
...
15
```

witheach is defined as `[]'[makewith do] is witheach`.

Pro tip: as **makewith** constructs a **times** loop, things that it constructs the loop around, such as a nest following a **witheach**, can use **i**, **i^**, **step**, **refresh** and **conclude**.

```
with.hold ( --> a )
```

with.hold is an ancillary stack used by nests created with **makewith**.

Searching and Sorting

```
matchitem findwith find findseq found sortwith sort sort$
match.test match.nest sort.test
```

```
matchitem ( a b c --> d )
```

matchitem builds on the idea demonstrated in **makewith** of constructing nests with useful behaviours on the fly by taking a word from the stack, constructing a nest that includes that word and passing it to **makewith** for further editing. A nest created with **matchitem** will examine each item in a nest, **a**, until it finds one that meets a criterion specified by **b**, then do **c**, (typically some stack tidying-up action dependant on the behaviour of **b**) and finally put the position, **d**, of the item that matched the criterion on the stack. **b** should expect to find an item of nest **a** on the stack and leave **true** on the stack if the criterion was met, and **false** otherwise. If **b** requires any other items on the stack, they should still be on the stack after doing **b**, and **c** should remove them.

For example, if the aim is to find the first item in a nest which is a number, then **b** could be the operator **number?**, which leaves **true** on the stack if the ToS was a number, and **false** if it was a nest or word. As **number?** only takes one argument no tidy-up is required, so **c** can be an empty nest, which is the Quackery equivalent of a no-op (do-nothing) instruction. In this demonstration, **a** is a nest containing a word as item 0, a nest as item 1 and a number as item 2.

```
/O> ' [ dup [ 0 1 ] 23 ] ' number? ' [ ] matchitem
...
```

Stack: 2

If no item in nest **a** meets criterion **b**, **matchitem** will leave the **size** of nest **a** on the stack.

```
/O> ' [ dup * 1+ ] ' number? ' [ ] matchitem
...
```

Stack: 3

`findwith (a --> b)`

`findwith` takes the functionality of `matchitem` and gives it a less flexible but more intuitive usage, by placing the words that specify its behaviour after the word `findwith` rather than expecting them to be on the stack. So the example given for `findwith` of searching a nest for the first instance of a number becomes,

```
/O> ' [ dup [ 0 1 ] 23 ] findwith number? [ ]  
...  
Stack: 2
```

`findwith` is defined as

```
[ ]'[ ]'[ matchitem ] is findwith
```

In the example above, the first `]'[` puts `number?` on the stack, and the second `]'[` puts the empty nest `[]` on the stack above `number?`.

`find (a b --> c)`

`find` expects a nest `b` on the ToS, and underneath that the number, word, or nest `a` to search for in `b`. If `a` is an item in `b`, it returns the position of the first occurrence of `a` in `b`, and if not, returns the size of `b`.

```
/O> 12 ' [ 10 11 12 13 ] find  
...  
Stack: 2
```

Commentary: `find` could be defined as follows, but in practice it is coded as an operator, i.e. in Python 3 code. A major design criterion in Quackery is that as much of Quackery be written in Quackery as is reasonable, to facilitate porting to other host languages, and so that elements such as the extensible compiler, `build`, the command line interpreter, `shell`, and the tools used to create them are exposed to the programmer, facilitating the creation of variations on either, should the programmer desire. As migrating `find` from Quackery to Python 3 gave a sixfold increase in compilation speed and a more responsive feel to `shell`, this was felt to be an acceptable concession.

```
[ findwith [ over = ] drop ] is find
```

`findseq (a b --> c)`

`find` expects a nest `b` on the ToS, and underneath that a sequence of numbers, words, or nests, enclosed in a nest `a`, to search for in `b`. If `a` is a subsequence within `b`, it returns the position of the start of first occurrence of `a` in `b`, and if not, returns the size of `b`.

```
/O> ' [ 12 13 14 ] ' [ 10 11 12 13 14 15 ] tuck findseq  
... split  
...  
Stack: [ 10 11 ] [ 12 13 14 15 ]
```

`found (a b --> c)`

`found` expects a nest, `a`, and a number, `b`, on the stack, and returns `true` if `b` is less than the size of `a`, and `false` otherwise. `found` can be used in conjunction with `find` or `findseq` as in this example, which searches for the string “found” in the nest of strings called `names` and reports where it is in the nest, or if it was not in the nest.

```
/O> $ "found"
... dup names find
... dup names found iff [ say "item number " echo ]
...                       else [ say "not found" drop ]
... drop
...
item number 74
```

`sortwith (a --> b)`

`sortwith` takes a nest, `a`, and returns a nest, `b`, containing the same items as `a` in order. The order is determined by the item following `sortwith`, which should take two items for comparison, and return a boolean. So if `sortwith` is followed by `<`, the items in the nest will be sorted so that each item is less than its predecessor.

```
/O> ' [ 4 2 1 3 ] sortwith <
...

Stack: [ 4 3 2 1 ]
```

In the implementation of Quackery presented here, `sortwith` is defined as an insertion sort, the algorithm used when sorting playing cards into a hand, taking each card in turn and inserting it into the correct position in a hand of cards.

```
[ ]'[ sort.test put
[] swap witheach
[ swap 2dup findwith
[ over
sort.test share
do ] [ ]
nip stuff ]
sort.test release ] is sortwith
```

`sort (a --> b)`

`sort` takes a nest of numbers, `a`, and returns a nest, `b`, with the same numbers as `a` in ascending numerical order.

`sort$ (a --> b)`

`sort` takes a nest of strings, `a`, and returns a nest, `b`, with the same strings as `a` in ascending QACSFOT order.

`match.test match.nest sort.test (--> a)`

These are ancillary stacks used by the searching and sorting words.

User Input and Output

```
input sp cr emit echo$ wrap$ echo ding
```

The nature of Quackery's connection to the outside world via the keyboard and the Terminal app in which the `shell` runs is determined largely by the design criteria that it should be portable between various environments, and that the implementation should be as simple as is possible. As Python offers no simple and portable means of detecting or reading individual keypresses, the fundamental unit of keyboard input is a line of text ending with the enter key being pressed. Similarly, characters displayed in the Terminal are only reliably displayed when a carriage return or equivalent is sent.

Additionally, while Python 3 supports Unicode, operating systems do not necessarily support the full range of Unicode characters, and the complexities of distinguishing between control characters, whitespace characters, combining characters, surrogates and so on and the limitations inherent in a monospace font as typically used by Terminal apps has led to the decision to support a minimal subset, namely the ASCII printable characters, the space, and the carriage return.

The bell code (Unicode code point 7) is supported by the vast majority of operating systems if not all (e.g. a system without audio capabilities would clearly not support it) and is useful in alerting a user to, for example, the completion of a lengthy computation. As is not a character so much as a piece of functionality which has been incorporated for historical reasons into the character set, this has been implemented as a separate Quackery word (`ding`) which causes an environmentally determined alert sound.

On operating systems that support it, the functionality of `input` is extended by importing the Python 3 readline extension module, which enables use of the up and down arrow keys to access a history of previously typed lines of text. This is a convenience for the Quackery programmer using the `shell` that could not be otherwise implemented given the limitations on detecting and reading keypresses.

```
input ( a --> b )
```

`input` takes a string, `a`, outputs it to the terminal and awaits user input via the keyboard terminated by the return or enter key, and returns the user input, not including the terminating keypress as a string, `b`. As with all i/o, any character in `a` or `b` that is not a either a space, carriage return or printable ASCII character (i.e. is not represented by 13 or a number in the range 32 to 126 inclusive is replaced by ? (i.e. character number 63.)

(User input is indicated here by underlined text.)

```
/O> $ "What is your name? " input
... say "Welcome, " echo$ cr
...
What is your name? It is Arthur, King of the Britons.
Welcome, It is Arthur, King of the Britons.
```

```
sp ( --> )
```

`sp` outputs a space character to the terminal.

```
cr ( --> )
```

`cr` causes the terminal to start a new line for text output.

`emit (a -->)`

`emit` outputs a character, `a`, to the terminal.

`echo$ (a -->)`

`emit` outputs a string of characters, `a`, to the terminal.

`wrap$ (a b -->)`

`emit` outputs a nest of strings, `a`, such as that created by `nest$`, to the terminal, starting on a new line and without splitting a string over two lines. The number `b` specifies the maximum number of characters displayed on a line. If a string within `a` is more than `b` characters long it will be displayed on one line, exceeding the specified maximum number of characters.

`echo (a -->)`

`echo` outputs the top of stack, `a`, to the terminal. It uses `unbuild` to decompile `a` and then displays the resulting string. Note when using `echo` to display numbers that `unbuild` decompiles numbers in the current `base`, whereas `build` compiles numbers in decimal unless temporarily overridden during compilation by use of the builder words `hex` or `now!`.

In this illustration, we will put the hexadecimal number `FEED` (i.e. decimal `65261`) on the stack first, then the decimal number `64206` (i.e. hexadecimal `FACE`) on top of it, then echo the `ToS` in hexadecimal and the `2oS` in decimal.)

```
/O> hex FEED      ( will be compiled as hexadecimal )
... 64206         ( will be compiled as decimal   )
... 16 base put   ( current base is now hexadecimal )
... echo sp
... base release  ( current base is back to decimal )
... echo sp
...
FACE 65261
```

`ding (-->)`

`ding` causes a system dependant alert sound, assuming the system supports audio, system alerts are enabled in the operating system and in the terminal app, and sound is not muted.

Note that some systems limit the number of alert sounds that can be made in rapid succession. Also the system bell may not sound until a `cr` has been sent. This too is system dependant.

File Management

`putfile takefile sharefile releasefile replacefile loadfile`

Files in Quackery are a variant on ancillary stacks, with two limitations. Firstly they can only contain strings, and secondly they can contain at most one item. Unlike ancillary stacks they do not need to be declared, so there is no equivalent to `[stack]` for files. If a file exists, it contains one item. If the file is empty, it contains an empty string. Files that do not exist contain zero items. To put something that is not a string into a file it should first be turned into string of Quackery (a Quackscript) using `quackify`, so that `loadfile`, `build`, or `quackery` can reconstruct the original item from it.

`putfile (a b --> c)`

`putfile` attempts to put a string, `a`, in a file named `b`, where `b` is a string. If `putfile` succeeds, (i.e. the file `b` did not already exist) then it returns `true` (1) to indicate success. If the file `b` did already exist then it returns `false` (0), indicating an overflow condition.

Assuming the file `pangram.txt` does not already exist;

```
/O> $ "A very bad quack might jinx zippy fowls."  
... $ "pangram.txt" putfile  
... $ "Eight ducks with pyjamas squeeze in an overfull box."  
... $ "pangram.txt" putfile  
...
```

Stack: 1 0

Now the file `pangram.txt` exists and contains the text “A very bad quack might jinx zippy fowls.” as indicated by the first invocation of `putfile` returning 1. It does not contain the text “Eight ducks with pyjamas squeeze in an overfull box.” as indicated by the second invocation of `putfile` returning 0.

`takefile (a --> b c)`

`takefile` attempts to remove a string, `b`, from a file named `a`, where `a` is a string. If the file exists then `b` is the contents of the file, the file is deleted and `c` is `true` (1), indicating success. If the file does not exist then `c` is `false` (0) and `b` is the name of the file, `a`.

Assuming that the file `pangram.txt` has been created as above;

```
/O> $ "pangram.txt" takefile  
... echo sp echo$ cr  
... $ "pangram.txt" takefile  
... echo sp echo$ cr  
...  
    1 A very bad quack might jinx zippy fowls.  
    0 pangram.txt
```

`sharefile (a --> b c)`

`sharefile` attempt to copy a string, `b`, from a file named `a`, where `a` is a string. If the file exists then `b` is the contents of the file, the file is *not* deleted and `c` is `true` (1), indicating success. If the file does not exist then `c` is `false` (0) and `b` is the name of the file, `a`.

`releasefile (a --> b)`

`releasefile` attempt to delete a file named `a`, where `a` is a string. If the file existed then `b` is `true` (1), indicating success. If the file did not exist then `b` is `false` (0).

`replacefile (a b --> c)`

`replacefile` attempt to replace the contents of a file named `a`, where `a` is a string, with a string, `b`. If the file existed then `c` is `true` (1), indicating success. If the file did not exist then `c` is `false` (0).

`loadfile (a -->)`

`loadfile` attempts to copy a Quackscript (a string of Quackery code) from a file named `a`, where `a` is a string, to the stack, then `build` and `do` it. If the file does not exist then it will display “file not found: “filename”” in the terminal, where `filename` is the string `a`.

Note that if there is a word in the `names & actions` dictionary that has the same name as the file, `loadfile` will do nothing. This enables the programmer to create a Quackscript file that will not load a second time in the same Quackery session, by having the Quackscript create a word with same name as the file. By convention this should be the first word defined in the file, and should be in the form:

```
[ this ] is myfile.qky
```

Exception Handling

```
protect backup ]bailby[ bail bailed message  
history backupwords restorewords releasewords  
protected fail
```

The construct “`iff a else b`” allows Quackery to choose between one of two options when the correct choice is known in advance. The exception handling words cater for another possibility, that it is impossible or impractical to determine the correct choice beforehand, so you need to try `a`, and if that doesn’t work, put everything back the way it was and do `b` instead. Putting everything back the way it was requires first making a backup of how things are so that they can be restored later if necessary. In Quackery this means at least saving the state of the system stacks; the stack, the return stack, and various ancillary stacks.

Say that at some point, while doing `a`, it becomes apparent that this option either is or is not working out as anticipated. So at that point in the code there will be “`if bail`” or an equivalent construct, where `true` on the ToS indicates that `a` was the wrong choice. It could be deeply nested, or inside an iteration – say `times` or `witheach` – wherever it becomes apparent that `a` was not the right option to take, and option `b` should be taken instead. For the sake of the following illustrative code, say that `a` expects 2 items on the stack.

```
2 backup a bailed if b
```

`backup` puts sufficient information on the `history` ancillary stack to enable `bailed` to revert the `protected` ancillary stacks to their current state if `bail` is invoked while doing `a`, and to exit `a` in an orderly fashion. To achieve this it needs to know how many stack items `a` expects on the stack, (i.e. 2).

`bailed` removes the information left by `backup` from the `history` ancillary stack, reverts the system stacks and returns `true` if `bail` was invoked, and `false` otherwise.

protect (-->)

protect adds the ancillary stack that follows it to the nest of **protected** ancillary stacks that **backup** uses.

```
[ stack ] is examplestack
  protect examplestack
```

backup (a -->)

a is the number of stack items that **backup** should copy to the **history** ancillary stack. **backup** records the following information on **history**; the sizes of the protected ancillary stacks, copies of **a** stack items, the size of the stack, the size of the return stack, and **false**, to indicate to **bailed** that **bail** has not been invoked.

]bailby[(a -->)

See the Meta Control Flow section above for discussion of reverse bracketed (“meta”) words. **a** is the number of nests that **bail** (or other words that invoke **]bailby[**) should exit from. It works by removing the appropriate number of items from the return stack. For example,

```
5 ]bailby[ is equivalent to ]done[ ]done[ ]done[ ]done[ ]done[
```

nestdepth]bailby[will cause Quackery to exit. Use **]bailby[** with extreme caution.

bail (-->)

As described in the discussion of Exception Handling above. **bail** replaces the **false** on the top of **history** with **true** to indicate that **bail** has been invoked and causes control flow to move to immediately after the word or nest following the most recently evaluated **backup**.

bailed (--> a)

As described in the discussion of Exception Handling above. **bailed** clears the items left on **history** by **backup**, and restores the stack and protected ancillary stacks to their former state if **bail** has been invoked. **a** is **true** if **bail** was invoked and **false** otherwise.

message (--> a)

message is an ancillary stack. It is not protected and will not be reverted to a former state by **bailed**. It can be used to convey information regarding why **bail** was invoked to the code that is executed if **bail** was invoked. To extend the “2 backup a bailed if b” example above to include a string to be displayed in the terminal as bailing information, if **bail** would become;

```
if [ $ "<insert reason for bailing here>" message put bail ]
```

and 2 backup a bailed if b would become;

```
2 backup a bailed if [ message take echo$ b ]
```

build (described in the Building Words section, below) uses the **message** ancillary stack.

history (--> a)

history is the ancillary stack used by **backup** to convey the necessary information from **backup** to **bailed** to enable **bailed** to revert the system to the state prior to **backup** being invoked if **bail** was invoked.

backupwords (-->)
restorewords (-->)
releasewords (-->)

backup and **bailed** only save and restore the states of the system stacks and protected ancillary stacks. These words save and restore the state of the Quackery dictionaries by copying them to and taking them from the **history** ancillary stack. **build** (described in the Building Words section, below) uses them.

If, in the “2 backup a bailed if b” example above the behaviour of **a** includes modifying the Quackery dictionaries (other than by invoking **build**) then

```
2 backup a bailed if b
```

should be extended to

```
backupwords
2 backup
a
bailed iff
  [ restorewords b ]
else releasewords
```

protected (--> a)

The ancillary stack **protected** carries a nest listing the ancillary stacks which are protected. At startup, and until the nest has been modified by **protect**, they are;

```
base temp depth to-do dip.hold b.nesting
times.start times.count times.action
with.hold mi.tidyup mi.result sort.test
```

fail (a -->)

fail expects a string, **a**. It exits Quackery and raises a Python **QuackeryError** exception in the same format as the exceptions raised by the Quackery virtual processor when it encounters problematic code, where **a** is a description of the problem encountered.

```
/O> $ "This must be Thursday." fail
...
```

Quackery crashed.

```
      Problem: This must be Thursday.
Quackery Stack:
  Return stack: {[...] 0} {quackery 1} {[...] 26} {shell 5}
{quackery 1}
```

To-do Stack

`to-do new-do add-to now-do do-now not-do`

To-do lists are a familiar idea; even if we do not write them down we often have in mind lists of things to attend to when it is opportune; when we are at the shops, when we have some free time, when it stops raining and so on. We notice that the stock of teabags is running low and needs replenishing, a cake needs eating, the lawn needs mowing, and so on, so we add each to its relevant list in order to do it later.

Similarly, a program can detect that something or things will need to be done later, and Quackery enables this with to-do stacks. It comes equipped with a to-do stack, called `to-do`. More to-do stacks can be added if required, they are just ancillary stacks put to a particular use.

The first step is to initialise the to-do stack with `new-do`. Imagining that we have appropriate word “buy” defined, which buys specified quantities of various items at a shop, we might compose a to-do stack thus; (cakes, teabags and bananas in this hypothetical example each put a stock-number on the stack, buy expects to find a quantity and a stock number on the stack.)

```
to-do new-do
  2 cakes   ' buy 2 to-do add-to
 80 teabags ' buy 2 to-do add-to
  5 bananas ' buy 2 to-do add-to
```

The ' preceding buy puts the word buy on the stack, and the 2 tells `add-do` how many items to take from the stack in addition to buy in readiness for when we get around to doing the `to-do` stack.

Later, when we are at the shop, we can buy all the items added to `to-do` with

```
to-do now-do
```

In regular stack fashion, `now-do` will do the things on the `to-do` stack in last-in, first out-order, and buy the bananas first, then the teabags, then the cakes. If the actions on a `to-do` stack need to be done in the same order as they were added, then `do-now` should be used instead of `now-do`. If the need to do a `to-do` stack vanishes, then `not-do` does the equivalent of crumpling up a to-do list and tossing it in the wastepaper basket.

Note that after applying `now-do`, `do-now` or `not-do` to a `to-do` stack, it will need to be reinitialised with `new-do` before being used again. Also note that `buy` does not put anything onto the stack. Words put on to `to-do` stacks must take the specified number of arguments from the stack and must not return any results to the stack.

For a non-hypothetical example of the use of a `to-do` stack, see `build` and `resolves` in the Building section, below.

```
to-do ( --> a )
```

`to-do` is an ancillary stack, provided for use as a general purpose to-do stack.

```
new-do ( a --> )
```

`new-do` expects an ancillary stack `a` and initialises it as a to-do stack by putting `done` on it, which acts as a marker indicating that there are no more actions on this stack to be done by `now-do` or `do-now`, or to be removed by `not-do`.

```
add-to ( a* b c d --> )
```

add-to puts an action to be performed, **b**, along with the zero or more stack items **a*** (the number of items is indicated by **c**), onto an initialised to-do stack **d**. The action **b** (which can be either a word or a nest) should expect **c** number of items on the stack, and leave zero items on the stack.

For example, the builder word **resolves** adds the word **replace** to the to-do stack **to-do**, along with the two items that **replace** will expect to find on the stack when it is performed. The relevant line in the definition of **resolves** is

```
' replace 2 to-do add-to
```

(The overall stack effect of this line of code is to remove two items from the stack which have been put there earlier in the definition. They are the stack items that **replace** expects.)

```
now-do ( a --> )
```

now-do expects an initialised to-do stack **a**. It removes items from **a** put there by **add-to** or **new-to** one at a time and performs each of them until it removes and performs the **done** left there by **new-to**, which causes it to finish.

```
do-now ( a --> )
```

do-now has the same functionality as **now-do**, except that it performs the actions on the to-do in the same order as they were added by **add-to**.

```
not-do ( a --> )
```

not-do expects an initialised to-do stack **a**. It clears items from **a** down to and including the first **done** that it encounters, deinitialising it.

Internal Representation

```
quid operator? number? nest? name? builder? immovable
```

```
quid ( a --> b )
```

quid (“Quackery Identifier”) returns a number, **b**, which is uniquely associated with the Quackery item (number, operator, or nest) **a**. The **quid** of a Quackery item will remain the same during the lifetime of the object, but not over multiple uses of Quackery. (i.e. if you run Quackery now,

```
' dup quid echo
```

might leave 4483186472 on the stack, but if you leave the shell and run Quackery again, it might leave 4380065576 on the stack this time.)

See the description of **oats**, (defined as [**quid swap quid =**] **is oats**) for some discussion of when two Quackery items are distinct from one another, and when they will return the same identifier.

```
operator? ( a --> b )
```

operator? takes an item **a** and returns **true** if **a** is an operator, and **false** otherwise.

`number? (a --> b)`

`number?` takes an item `a` and returns `true` if `a` is a number, and `false` otherwise.

`nest? (a --> b)`

`nest?` takes an item `a` and returns `true` if `a` is a nest, and `false` otherwise.

`name? (a --> b)`

`name?` takes a string `a` and returns `true` if `a` is the name of a Quackery word, and `false` otherwise.

`builder? (a --> b)`

`builder?` takes a string `a` and returns `true` if `a` is the name of a Quackery builder, and `false` otherwise.

`immovable (-->)`

Attempting to **take** one more item from an ancillary stack than the ancillary stack contains will cause Quackery to crash.

```
/O> [ stack 3 4 5 ] is example
... example take echo sp
... example take echo sp
... example take echo sp
... example take echo sp
...
5 4 3
Quackery crashed.
```

Problem: Cannot remove immovable item.

This is because `stack` is defined as an immovable nest. An immovable nest has the word `immovable` as its zeroth item.

```
[ immovable ]this[ ]done[ ] is stack ( --> s )
```

`immovable` is a no-op; performing `immovable` does nothing. It exists to prevent the ancillary stack words `take`, `release`, and `replace` (which **releases** an item before **putting** a new item in its place) from removing too many items from an ancillary stack, which would cause Quackery to crash in a way that would be difficult to debug.

As tables can be modified on the fly using ancillary stack words, `table` is also an immovable word, by virtue of having `immovable` as its zeroth item.

```
[ immovable
  dup -1 > if 1+
  ]this[ swap peek
  ]done[ ]          is table ( n --> x )
```

Dictionary

names actions builders jobs namenest actiontable buildernest jobtable

Quackery has two dictionaries, which associate the words that make up Quacksript with the behaviours of those words. The larger of the two dictionaries is the **names & actions** dictionary. The **names** nest, which resides on the ancillary stack **namenest**, contains the words that represent the regular Quackery words such as **dup**, **/mod**, **again**, **split**, **ding**, **takefile**, and so on. **actions** is a table with holds the behaviours; the operators, nests, and numbers, of those words.

The smaller of the two dictionaries is the **builders & jobs** dictionary. The **builders** nest resides on the ancillary stack **buildernest**, and **jobs** is a table. Both are similar in function to their **names & actions** equivalents, but hold the words and behaviours of directives for the Quackery compiler, **build**. These are **[**, **]**, **is**, **builds**, **(**, **)**, **forward**, **resolves**, **char**, **\$**, **say**, **hex**, and **now!**.

The behaviour of **build** is covered in more detail below, but briefly, it finds the position of each word in a Quacksript in either **builders** or **names** passes that position to either **actions** or **jobs** respectively, which return the associated operator, nest, or number to be appended to the current nest being compiled, in the case of regular Quackery words, or performed using **do** in the case of compiler directives.

The tables; **actions** and **jobs**, can be searched with **find** and extended using **put**, via **actiontable** and **jobtable**, which inhibit their **table** behaviour by quoting them.

names (--> a)

a is the nest of strings that contains the names of regular Quackery words, ordered from most recently defined to first defined.

```
/O> names 65 wrap$
... cr names size echo $ " names" echo$
...
```

```
quackify replacefile takefile loadfile words empty wrap$ leave
shell echostack echoreturn return$ echo unbuild nesting quackery
build releasewords restorewords backupwords unresolved b.to-do
b.nesting message jobtable jobs builder? builders buildernest
actiontable actions name? names namenest nest$ oats bailed bail
backup history shuffle random randomise initrandom prng prng.d
prng.c prng.b prng.a rot64 64bits 64bitmask $->n char->n sort$ $>
$< qacsfort sort sortwith sort.test not-do do-now now-do add-to
new-do to-do unpack pack reflect reverse nextword trim printable
found findwith matchitem mi.result mi.tidyup echo$ witheach
makewith with.hold number$ decimal base digit lower upper sp
space cr carriage findseq behead stuff pluck of table temp step
refresh conclude i^ i times times.action times.count times.start
abs decurse depth 2over 2swap dip dip.hold protect protected
nested move tally replace release share stack while until recurse
do this ' copy clamp max min else iff if done again 2drop 2dup
within unrot tuck bit mod nip / - < xor != or and not true false
sharefile releasefile putfile input ding emit quid operator?
number? nest? size poke peek find join split [] take immovable
put ]bailby[ ]do[ ]this[ ]'[ ]else[ ]liff[ ]if[ ]again[ ]done[
over rot swap drop dup return nestdepth stacksize time ~ ^ | & >>
<< ** /mod * negate + 1+ > = nand fail
212 names
```

`actions (--> a)`

`a` is the table of behaviours associated with the regular Quackery words, ordered as with `names`.

```
/O> 0 actions echo say " >>> " 211 actions echo
...
quackify >>> fail
```

`builders (--> a)`

`a` is the nest of strings that contains the names of the compiler directive words, ordered from most recently defined to first defined.

```
/O> builders 65 wrap$
... cr builders size echo $ " builders" echo$
...

constant now! hex say $ char resolves forward ) ( builds is ] [
14 builders
```

`jobs (--> a)`

`a` is the table of behaviours associated with the compiler directive words, ordered as with `jobs`.

```
/O> 0 jobs echo say " >>> " 13 jobs echo
...
[ over [] = if [ [ [ ]'[ ] [ 34 99 111 110 115 116 97 110 116 34 32
110 101 101 100 115 32 115 111 109 101 116 104 105 110 103 32 98
101 102 111 114 101 32 105 116 46 ] ] message put bail ] dip [ -1
pluck do dup number? not if [ ' ' nested swap nested join nested ]
join ] ] >>> [ [ [ ]'[ ] [ 91 ] ] b.nesting put [] swap ]
```

(Note that, unlike the illustrative code for `actions`, where we see the names of the behaviours echoed, here we see the decompiled behaviours. This is because `unbuild`, which `echo` uses to find the names associated with operators and named nests, only searches the `names` & `actions` dictionary, so does not find `constant` or `[`, the zeroth and thirteenth items in the `jobs` table, and therefore assumes they are nameless and decompiles them.)

`namenest (--> a)`

`namenest` is the ancillary stack on which `names` resides.

`actiontable (--> a)`

`actiontable` puts a pointer to `actions` on the stack so it can be searched with `find` and extended with `put`.

`buildernest (--> a)`

`buildernest` is the ancillary stack on which `builders` resides.

`jobtable (--> a)`

`jobtable` puts a pointer to `jobs` on the stack so it can be searched with `find` and extended with `put`.

Building

```
build quackery
[ ] is builds forward resolves char $ say hex now! ( )
unbuild quackify
unresolved nesting b.nesting b.to-do
```

Building is the Quackery term for assembling or compiling a program from the source text (“Quackscript”).

```
build ( a --> b )
```

build takes a string **a**, the Quackscript, and returns a nest, **b**, which can be performed by **do** to “run the program.”

Any string **a** is a valid Quackscript, in the sense that **build** will always return a nest **b** that can be performed by **do**. However, the majority of possible strings will return a nest that reports a problem with the string. (Here we assume that “rinnzekete” has not been defined as a Quackery word, which can be classified problematic; in this case the problem probably being a mismatch of expectation and reality on the part of the programmer.)

```
/O> $ "21 21 + echo rinnzekete"
... build dup echo cr cr
... do
...
[ ' [ 85 110 107 110 111 119 110 32 119 111 114 100 58 32 114 105
110 110 122 101 107 101 116 101 ] echo$ ]
```

Unknown word: rinnzekete

Another possible problem is having a **[** without a corresponding **]**.

```
/O> $ "[ 1764 42 / echo"
... build dup echo cr cr
... do
...
[ ' [ 85 110 102 105 110 105 115 104 101 100 32 110 101 115 116
46 ] echo$ ]
```

Unfinished nest: [

There are nineteen ways in which Quackscripts can be problematic, including containing an unknown word and having an unfinished nest. The other seventeen are discussed in the descriptions of the builder words below.

When a problem is detected, either by **build** or by a builder word, compilation is ceased immediately by **bail** being invoked, resetting any **protected** ancillary stacks in use to their state prior to **bailing**. This does not include the **message** ancillary stack, which is used to carry a description of the problem encountered (e.g. “Unknown word: rinnzekete” or “Unfinished nest: [”) to the code at the end of **build** that is performed when **bail** is invoked. In addition to using the exception handling words **backup** and **bailed** to restore the states of various stacks, **build** uses **backupwords**, **restorewords**, and **releasewords** to restore any changes made to the dictionaries by builder words. Also, the **b.to-do** ancillary stack is initialised as a to-do stack ready to convey actions necessary to revert any other changes to the system not covered by the exception handling words. See **resolves** for an example of this.

The process of **building** a nest occurs in three phases, the setup phase, the main loop and the clean-up phase.

During the setup phase **build** prepares for **bail** to be invoked, puts an empty string on the ancillary stack **b.nesting** and initialises the to-do stack **b.to-do** (see below for discussion of these), uses **decimal** to put **10** on the ancillary stack **base**, and puts an empty nest, “the target nest”, on the stack underneath the Quackscript string **a**, “the source text”. This will become the returned nest, **b**.

In the main loop, **build** removes any leading whitespace from the source string with **trim**, checks to see if it is now an empty string, which ends the loop, then removes the first word it encounters from the Quackscript with **nextword** and attempts to process it.

It searches the **builders & jobs** dictionary first, and if it finds the word, it does the associated job and then restarts the main loop.

If the word is not a building word, it searches the **names & actions** dictionary, and if it finds the word, appends it to the target nest with **nested join** and then restarts the main loop.

If it has not found the word in either dictionary, it attempts to parse it as a decimal number with **\$->n**. If this is successful it appends the number to the target nest and then restarts the main loop.

If the word is not in either dictionary and is not a number, then it is an unknown word, and **bail** is invoked with an appropriate problem report on the **message** ancillary stack.

The clean-up phase first checks the **nesting** ancillary stack to ensure that the nests are well formed (and if not, invokes **bail** with an appropriate problem report), and **releases** the **base** ancillary stack.

Then it checks if **bail** has been invoked with **bailed**. If **bail** was invoked, it performs any actions on the **b.to-do** stack, constructs an appropriate problem report nest to return, and restores the dictionaries. Otherwise it clears the **b.to-do** stack and **releases** the dictionary backups, leaving a successfully compiled nest on the stack.

Notes

Dictionaries are searched from most recently added to least recently added, so when a word is added to either dictionary that is the same as one already present, the new one will be found by **build** thereafter.

Because **build** searches for builder words before regular Quackery words, the builder words are effectively reserved, as while it is possible to define a regular Quackery word with the same name as a builder word, it will not override the builder word.

Because the dictionaries are searched before attempting to parse a word as a decimal number, it is not a good idea define a word with only the digits **0** to **9** in its name.

quackery (a -->)

[build do] is quackery

Quackery in a nutshell.

The builders, `[`, `]`, `is`, `builds`, `(`, `)`, `forward`, `resolves`, `char`, `$`, `say`, `hex`, `constant`, and `now!` are invoked by `build` when it encounters them in a Quackscript.

Discussion of the builders will refer to their definitions in the source code, where they are named `b.[`, `b.]`, `b.is`, `b.builds`, `b.(`, `b.)`, `b.forward`, `b.resolves`, `b.char`, `b.$`, `b.say`, `b.hex`, `b.constant`, and `b.now!` respectively. These names are used to differentiate them from the equivalent compiler directives used by the Python Quackery compiler (i.e. the one defined using Python code) `build()`, which exists solely to build the predefined words in Quackery that are not primitives. The Python Quackery compiler is discussed elsewhere. This discussion relates to the Quackery Quackery compiler (i.e. the one defined using Quackery code) `build`.

Unlike the Python Quackery compiler, `build()`, The Quackery Quackery compiler, `build`, can be extended by adding new builders using `builds`. Builders must comply with The Building Regulations.

The Building Regulations

- When a builder is invoked, the top of stack will be a string of Quackscript (the source string) and the second on stack will be a nest under construction (the target nest). This should still be the case after the builder has done its job.
- Builders that use text from the Quackscript string on the ToS should check for problematic Quackscript (for example, if a builder requires some text but the string is empty then there is a problem) and if a problem is found should report it by putting a descriptive string on the message ancillary stack and invoking `bail`. Examples of this can be found in the definitions of `is`, `builds`, `(`, `forward`, `resolves`, `char`, `$`, and `hex`.
- Builders that use items from the nest under construction, (2nd on Stack) should check for problematic nests (for example, if a builder requires the nest to contain at least one item but the nest is empty then there is a problem) and if a problem is found should report it by putting a descriptive string on the message ancillary stack and invoking `bail`. Examples of this can be found in the definitions of `is`, `builds`, and `resolves`.
- Embracing builders that, for example, start and end the construction of a nest, as `[` and `]` do, should make use of the ancillary stack `b.nesting` to ensure that the nesting in the Quackscript is properly balanced. To do this, the builder that starts an embrace (an “opening brace”) should `put` a string containing its name on `b.nesting`, and the builder that ends the embrace (a “closing brace”) should `take` the top of `b.nesting` and check that it is a name that should be there. For example, `]` expects the string `“[”` to be on `b.nesting`.

If a closing brace finds an empty string on `b.nesting` it should report that with the descriptive message `“Unexpected “x”.”`, where `x` is the closing brace’s name, and if it finds an unacceptable string `y` on `b.nesting` it should report that with `“Nest mismatch: y x”` for consistency with other problem reports generated in this manner. An example of this behaviour can be found in the definition of `]`.

Embraces are not restricted to pairs of builders; more complex relationships are permitted. (For example a nesting structure that brings Regex type functionality might permit one or more centre braces with a behaviour that reflects the functionality of the regex alternation operator.) Any nesting structure must be able to be validated by use of the `b.nesting` ancillary stack.

- Builders that modify parts of Quackery that are not backed up by the Exception Handling words used in `build` should provide a recovery mechanism using the to-do stack `b.to-do` to undo their modifications if `bail` is invoked by another builder or by `build` later in the compilation. An example of this behaviour can be found in the definition of `resolves`.

`[(a --> b a)`

The builder `[` is the opening brace of the matched pair `[` and `]`. It places an empty nest `b` underneath the source string `a`, making `b` the target nest.

In accordance with The Building Regulations it puts the string “[” on the ancillary stack `b.nesting`.

`] (a b c --> d c)`

The builder `]` is the closing brace of the matched pair `[` and `]`. It appends the target nest `b` to the previous target nest `a` using `nested join`, making the target nest `d`, which is underneath the source string `c`.

In accordance with The Building Regulations it takes a string from the ancillary stack `b.nesting`. If that string is empty, it invokes `bail`, reporting “Unexpected ”]” via the ancillary stack `message`.

If the string is not empty and is not “[” it invokes `bail`, reporting “Nest mismatch: y]”, where `y` is the string that was on `b.nesting`. (Note that for this to occur a second matched pair of builders would have to be defined – `[` and `]` is the only matched pair that Quackery has predefined.)

`is (a b --> c d)`

The builder `is` extends Quackery by (1) removing the item most recently added to the target nest, `a`, and adding it to the front of the `actions` table and (2) removing the next word from the source text, `b`, and adding it to the front of the `names` nest. `c` and `d` are the modified target nest and source text respectively.

In accordance with The Building Regulations, `is` reports the following problems.

If the target nest is empty, it reports “`"is" needs something to name.`”

If the source string is empty or contains only whitespace, it reports “`"is" needs a name after it.`”

`builds (a b --> c d)`

The builder `builds` extends the Quackery compiler by (1) removing the item most recently added to the target nest, `a`, and adding it to the front of the `jobs` table and (2) removing the next word from the source text, `b`, and adding it to the front of the `builders` nest. `c` and `d` are the modified target nest and source text respectively.

In accordance with The Building Regulations, `build` reports the following problems.

If the target nest is empty, it reports “`"build" needs something to name.`”

If the source string is empty or contains only whitespace, it reports “`"build" needs a name after it.`”

forward and **resolves** provide a mechanism to create more complex recursive structures than are within the capability of **recurse** and **decurese**. One example of this is mutually recursive words.

Mutual recursion can be illustrated with two words, **odd** and **even**.

Oddness and evenness for non-negative whole numbers can be defined by asserting that such numbers are either even or odd, that 0 is an even number, that a number one less than an odd number is an even number, and that a number one less than an even number is an odd number.

The word **odd** takes a non-negative number and returns **true** if the number is odd, and **false** if the number is even.

The word **even** takes a non-negative number and returns **true** if the number is even, and **false** if the number is odd.

If the number passed to **odd** is 0, it returns **false**. Otherwise it subtracts one from the number and passes it to **even**. It can be defined with:

```
[ dup 0 =  
  iff [ drop false ]  
  else [ 1 - even ] ] is odd ( n --> b )
```

If the number passed to **even** is 0, it returns **true**. Otherwise it subtracts one from the number and passes it to **odd**. It can be defined with:

```
[ dup 0 =  
  iff [ drop true ]  
  else [ 1 - odd ] ] is even ( n --> b )
```

However, these definitions have a problem. The definition of **odd** assumes that **even** is defined, and the definition of **even** assumes that **odd** is defined. Whichever we define first will report that the other is undefined. The way around this dilemma is with **forward** and **resolves**, which allow a word to be named before it is defined.

```
forward is odd  
[ dup 0 =  
  iff [ drop true ]  
  else [ 1 - odd ] ] is even ( n --> b )  
  
[ dup 0 =  
  iff [ drop false ]  
  else [ 1 - even ] ] resolves odd ( n --> b )
```

```
forward ( a b --> c b )
```

The builder **forward** appends the nest [**unresolved**] to the target nest **a**, ready to be removed by **is** and returning the updated target nest **c**.

If you need to use the builders **forward** and **resolves** when defining a new builder, do not use the phrase “**forward builds my-new-builder**” as it will not work as you may anticipate.

Instead use “**forward is my-new-builder**”, and then use the phrase “**my-new-builder builds my-new-builder**” after “**resolves my-new-builder**”, which may seem odd, and leaves a redundant entry in the names and actions dictionary, but does work.

`resolves (a b --> c d)`

The builder `resolves` modifies a word defined with `forward is` by replacing the `unresolved` in its definition with the item most recently added to the target nest, `a`. It expects to find the name of an `unresolved forward` word at the front of the source text `b`. `c` and `d` are the target nest and source text with the most recent item and the name removed.

In accordance with The Building Regulations, `resolves` has the following behaviours.

If the target nest is empty, it reports `"resolves" needs something to resolve with.`

If the source string is empty or contains only whitespace, it reports `"resolves" needs a name to resolve.`

If the word that follows it in the source string is not defined, it reports `"Unknown word after "resolves": x"`, where `x` is the undefined word.

If the word that follows it in the source string is defined, but the definition is not `[unresolved]`, it reports `"x" is not an unresolved forward reference.`, where `x` is the incorrectly defined word.

If the name that follows it in the source text is an `unresolved forward` word, in addition to replacing the `unresolved` in its definition, it adds a nest that will undo that replacement to the to-do stack `b.to-do`, as that is a modification that is not automatically reverted when `bail` is invoked.

`char (a b --> c d)`

The builder `char` takes the next printable character from the source text `b` and appends the number representing its Unicode code point to the target nest `a`, returning the updated target nest `c` and source text `d`.

```
/O> char A emit
... char B emit
... char C emit
...
ABC
```

In accordance with The Building Regulations, `resolved` reports the following problem.

If the source string is empty or contains only whitespace, it reports `"char" needs a character after it.`

`$ (a b --> c d)`

The builder `$` creates a string from the characters that follow it in the source text `b` and appends a nest that will place that string on the stack to the target nest `a`, returning the updated target nest `c` and source text `d`.

In detail, the string it creates consists of the all characters in the source text between the first and second instance of the delimiting character. The delimiting character is the first printable character following `$`. Typically the delimiting character might be `"` or `'`, but it can be any of the 94 printable characters that Quackery recognises. Thus `$ "this"`, `$ 'is'`, `$ |a|`, and `$ QstringQ` are all valid representations of strings, and represent the four strings `this`, `is`, `a`, and `string`.

The nest \$ compiles is of the form illustrated by the following shell dialogue.

```
/O> ' $ "A B
... C D"
... dup echo cr
... do echo$
...
[ ' [ 65 32 66 13 67 32 68 ] ]
A B
C D
```

The rationale for \$ compiling a single nest rather than compiling a ' followed by a string (e.g. [65 32 66 13 67 32 68]) is that this allows tables of strings to be created as per the following illustrative code.. day\$ returns the name of a day as a string. The argument n is the number of days since a Sunday.

```
[ 7 mod
[ table $ "Sunday"    $ "Monday"
        $ "Tuesday"  $ "Wednesday"
        $ "Thursday" $ "Friday"
        $ "Saturday" ] do ]      is day$ ( n --> $ )
```

Note that when a nest created with \$ is quoted using ' or included in a table it is necessary to use do to convert the nest placed on the stack into a string.

In accordance with The Building Regulations, \$ reports the following problems.

If the source string is empty or contains only whitespace, it reports “"\$" needs to be followed by a string.”

If there is no second instance of the delimiting character, it reports “Endless string discovered.”.

say (a b --> c d)

say has the same functionality as \$, except that instead of leaving the string that follows it on the stack, it echoes it to the screen. Also, it reports "say" needs to be followed by a string., instead of "\$" needs to be followed by a string.

The nest that say compiles is of the form [' [65 32 66 13 67 32 68] echo\$]

hex (a b --> c d)

The builder hex adds the hexadecimal number that follows it in the source text b to the target nest a, returning the updated target nest c and source text d. A hexadecimal number is represented by an optional - sign, followed by one or more hexadecimal characters, which are the digits 0 to 9 and the letters A to F and a to f. Both the upper case and lower case letters represent the hexadecimal digits equal to 10 to 15 in decimal.

In accordance with The Building Regulations, hex reports the following problems.

If the source string is empty or contains only whitespace, it reports “"hex" needs a number after it.”

If the printable characters following hex are not a valid hexadecimal number it reports “"xyz" is not hexadecimal.”, where xyz is the invalid text.

`constant (a b --> c d)`

The builder `constant` removes the item most recently added to the target nest `a` and does it, returning the updated target nest `c`. The item it does should leave a single item on the stack (`--> x`), and otherwise be in accordance with The Building Regulations. The item `x` will be added to the target nest `a` as a literal.

To illustrate this, the word `smallprime` returns `true` if `a` is a prime number less than `1000`, and `false` otherwise. (By definition only positive integers can be prime, hence the `0 max`.)

```
[ 0 max bit
  [ 0
    ' [ 2 3 5 7 11 13 17 19 23 29 31 37
        41 43 47 53 59 61 67 71 73 79 83 89
        97 101 103 107 109 113 127 131 137 139 149 151
        157 163 167 173 179 181 191 193 197 199 211 223
        227 229 233 239 241 251 257 263 269 271 277 281
        283 293 307 311 313 317 331 337 347 349 353 359
        367 373 379 383 389 397 401 409 419 421 431 433
        439 443 449 457 461 463 467 479 487 491 499 503
        509 521 523 541 547 557 563 569 571 577 587 593
        599 601 607 613 617 619 631 641 643 647 653 659
        661 673 677 683 691 701 709 719 727 733 739 743
        751 757 761 769 773 787 797 809 811 821 823 827
        829 839 853 857 859 863 877 881 883 887 907 911
        919 929 937 941 947 953 967 971 977 983 991 997 ]
    witheach
      [ bit | ] ] constant
    & 0 > ] is smallprime ( n --> b )
```

`smallprime` works by converting the number `n` to a bit pattern where every bit is `0` except for the `n`th bit, and comparing it to a bit pattern where every bit is zero unless the bit position corresponds to a prime number less than `1000`. Expressed as a decimal number it is 301 digits long.

As this is a somewhat cumbersome number to put in a Quackery word it is computed from a nest of the prime numbers less than `1000` (the section of the definition that starts `[0` and ends `bit |]`). Performing this computation every time the word `smallprime` is invoked is pointless as it always returns the same number, so it is followed by the word `constant`, indicating that it can be computed once, during compilation.

We can confirm that this has happened by decompiling `smallprime` in the shell.

```
/O> ' smallprime copy
...

Stack: [ bit
1360396709629698212581594767974700056400491703929229463506337903156
3183503598513586036297222193352555538909171790625246962409298703309
8638245598338502064276858734705184535549642030768571939159874046323
9251581515774735157478399074962378414221254602604809196179277244169
127525629985231081713895503177900 & 0 > ]
```

In accordance with The Building Regulations, `constant` reports the following problem.

If the target nest `a` is empty it reports ““constant” needs something before it.”

now! (a b --> c d)

The builder `now!` removes the item most recently added to the target nest `a` and `does` it, returning the updated target nest `c`. The behaviour of the item it `does` should be in accordance with The Building Regulations.

```
/O> [ $ "starting compilation" echo$ cr ] now!  
... $ "performing target nest" echo$ cr  
... 10 times [ i^ echo sp ] cr  
... [ $ "ending compilation" echo$ cr ] now!  
...  
starting compilation  
ending compilation  
performing target nest  
0 1 2 3 4 5 6 7 8 9
```

((a --> b)

The builder `(` removes characters from the source text until it finds and removes a `)` which is not adjacent to a printable character.

In accordance with The Building Regulations, `$` reports the following problems.

If it does not find a `)` without adjacent printable characters it reports “Unfinished comment.”

) (-->)

The builder `)` does not need to be defined, or could do nothing, but instead it has the punctilious and arguably pernicky behaviour of invoking `bail` with the message “Unexpected `)` `).`”

Quackery comments are wrapped in the builders `(` and `)`. `(` and `)` are not an embracing pair in the sense described in The Building Regulations as they are non-nesting.

```
( Below is some code that has been commented  
  out ... )
```

```
( [ swap rot ] is twist ( a b c --> c b a )
```

```
( Note that the non-nesting behaviour allows  
  the code to be commented out with just the  
  "(" at the start as the comment is ended  
  by the ")" at the end of the stack comment.  
  Also note that the closing parenthesis in  
  the previous sentence does not end this  
  comment as it is adjacent to printable  
  characters, namely the quotation marks on  
  either side of it. )
```

`unbuild (a --> b)`

`unbuild` is the Quackery decompiler. It takes a Quackery object `a` and returns a string of Quackscript `b` that will recompile as `a` if passed to `build`.

Limitations

- `unbuild` does not handle nests created by imtemperate use of Ancillary Stack words well. It does not detect endless loops and will attempt to construct a string of infinite length, which will eventually cause Python to crash in an undignified manner unless the user presses Control C and terminates the program abruptly in the meantime.

This can be ameliorated by putting a small positive number on the ancillary stack `nesting`. (`nesting release` should be invoked afterwards to restore `unbuild` to its default behaviour.) This will limit the depth of nesting that `unbuild` decompiles to, with nests beyond the limit being represented by “[. . .]” in the returned string. If this happens `build` will not be able to recompile `a` from the returned string.

- `unbuild` cannot infer that builders other than `[` and `]` were employed in the construction of a nest passed to it, so the returned string will not have niceties such as comments, strings will be represented as for example `[' [65 66 67]]` rather than `$ "ABC"` and so on.

`quackify (a --> b)`

`quackify` takes an item `a` and returns a Quackscript `b`. `b` would recreate `a` if passed to `quackery` or loaded from a file by `loadfile`.

`quackify` uses `unbuild` to generate the Quackscript, so shares its limitations.

`unresolved (-->)`

`unresolved` causes Quackery to fail with a Python `QuackeryError` exception describing the problem as “Unresolved reference.”

`nesting (--> a)`

`nesting` is an ancillary stack which can be used to restrict the depth of nesting decompiled by `unbuild`.

`b.nesting (--> a)`

`b.nesting` is an ancillary stack used by `build`.

`b.to-do (--> a)`

`b.to-do` is a to-do stack used by `build`.

Time

`time`

`time (--> n)`

`n` is the number of microseconds since the Unix epoch (00:00:00 UTC, 1st of January 1970). Accuracy is system dependant, and not guaranteed to be better than to the second.

Development Tools

`empty words shell leave stacksize echostack nestdepth return return$ echoreturn`

`empty (... x y z -->)`

`empty` drops every item from the stack.

`words (-->)`

`words` displays a list of all the Quackery names, followed by the builders. Both lists are ordered from first defined to most recently defined.

`shell (-->)`

`shell` is an interactive language shell or REPL (“Read Evaluate Print Loop”).

It is an endless loop that initially displays a duck’s head prompt `/O>` to indicate it is awaiting user input, and accumulates a string of entered Quackscript line by line, displaying the continuation prompt `. . .`, until the user presses enter twice. It passes the entered Quackscript to `quackery`, invokes `echostack` (below) with `nesting` set to 5, and then loops.

`leave (-->)`

`leave` is used to leave the `shell` by breaking out of its endless loop.

`stacksize (--> a)`

`stacksize` returns the number of items on the stack.

`echostack (-->)`

If the stack is empty, displays the text “Stack empty.”, otherwise displays the text “Stack: ” followed by the contents of the stack, with each item decompiled with `unbuild`.

`nestdepth (--> a)`

`nestdepth` returns the number of address pairs on the return stack. As with most processors, the virtual Quackery processor keeps track of where it has got to by means of a return stack, (a.k.a. call stack). Typically an address is a location in memory, but as Quackery is based on a memory model implemented as dynamic arrays (i.e. nests), an address pair consists of a pointer to a nest and an offset to an item within the nest, akin to a street name and house number.

`return (--> a)`

`return` puts a nest containing a copy of the return stack on the stack.

`return$ (--> a)`

`return$` returns a representation of the return stack with address pairs wrapped in curly brackets “{” and “}” and unnamed nests indicated by “[. . .]”

`echoreturn (-->)`

`echoreturn` is equivalent to `return$ echo$`.

Quackery in Python

```
# [                                     quackery                                     ]

import time
import sys
import os
try:                                     # note 1
    import readline
except:
    pass

class QuackeryError(Exception):
    pass

def quackery(source_string):

    """ Perform a Quackery program.
        Return the stack as a string. """

    def failed(message):
        traverse(build(""" stacksize pack
                           decimal unbuild
                           return$
                           nestdepth ]bailby[ """))
        returnstack = string_from_stack()
        thestack = string_from_stack()
        raise QuackeryError('\n          Problem: ' + message +          # note 3
                           '\nQuackery Stack: ' + str(thestack)[2:-2] +
                           '\n  Return stack: ' + str(returnstack))

    def isNest(item):                                     # note 4
        return(isinstance(item, list))

    def isNumber(item):
        return(isinstance(item, int))

    def isOperator(item):                                # note 5
        return(isinstance(item, type(lambda: None)))

    def expect_something():
        nonlocal qstack
        if qstack == []:
            failed('Stack unexpectedly empty.')

    def top_of_stack():
        nonlocal qstack
        return(qstack[-1])

    def expect_nest():
        expect_something()
        if not isNest(top_of_stack()):
            failed('Expected nest on stack.')

    def expect_number():
        expect_something()
        if not isNumber(top_of_stack()):
            failed('Expected number on stack.')
```

```

def to_stack(item):
    nonlocal qstack
    qstack.append(item)

def from_stack():
    nonlocal qstack
    expect_something()
    return(qstack.pop())

def string_from_stack():
    expect_nest()
    result = ''
    for ch in from_stack():
        if ch == 13:
            result += '\n'
        elif 31 < ch < 127:
            result += chr(ch)
        else:
            result += '?'
    return(result)

def string_to_stack(str):
    result = []
    for ch in str:
        if ch == '\n':
            result.append(13)
        elif 31 < ord(ch) < 127:
            result.append(ord(ch))
        else:
            result.append(ord('?'))
    to_stack(result)

def qfail():
    message = string_from_stack()
    failed(message)

def stack_size():
    nonlocal qstack
    to_stack(len(qstack))

def qreturn():
    nonlocal rstack
    to_stack(rstack)

def dup():
    a = from_stack()
    to_stack(a)
    to_stack(a)

def drop():
    from_stack()

def swap():
    a = from_stack()
    b = from_stack()
    to_stack(a)
    to_stack(b)

```

note 6

```

def rot():
    a = from_stack()
    swap()
    to_stack(a)
    swap()

def over():
    a = from_stack()
    dup()
    to_stack(a)
    swap()

def nest_depth():
    nonlocal rstack
    to_stack(len(rstack)//2)

def to_return(item):
    nonlocal rstack
    rstack.append(item)

def from_return():
    nonlocal rstack
    if rstack == []:
        failed('Return stack unexpectedly empty.')
    return(rstack.pop())

true = 1

false = 0

def bool_to_stack(qbool):
    to_stack(true if qbool else false)

def nand():
    expect_number()
    a = from_stack()
    expect_number()
    bool_to_stack(from_stack() == false or a == false)

def equal():
    expect_something()
    a = from_stack()
    expect_something()
    bool_to_stack(a == from_stack())

def greater():
    expect_number()
    a = from_stack()
    expect_number()
    bool_to_stack(from_stack() > a)

def inc():
    expect_number()
    to_stack(1 + from_stack())

def plus():
    expect_number()
    a = from_stack()
    expect_number()
    to_stack(a + from_stack())

```

note 7

```

def negate():
    expect_number()
    to_stack(-from_stack())

def multiply():
    expect_number()
    a = from_stack()
    expect_number()
    to_stack(a * from_stack())

def qdivmod():
    expect_number()
    a = from_stack()
    if a == 0:
        failed('Cannot divide by zero.')
    expect_number()
    results = divmod(from_stack(), a)
    to_stack(results[0])
    to_stack(results[1])

def exponentiate():
    expect_number()
    a = from_stack()
    if a < 0:
        failed('Cannot ** by a negative number: ' + str(a))
    expect_number()
    to_stack(from_stack() ** a)

def shift_left():
    expect_number()
    a = from_stack()
    if a < 0:
        failed('Cannot << by a negative number: ' + str(a))
    expect_number()
    to_stack(from_stack() << a)

def shift_right():
    expect_number()
    a = from_stack()
    if a < 0:
        failed('Cannot >> by a negative number: ' + str(a))
    expect_number()
    to_stack(from_stack() >> a)

def bitwise_and():
    expect_number()
    a = from_stack()
    expect_number()
    to_stack(a & from_stack())

def bitwise_or():
    expect_number()
    a = from_stack()
    expect_number()
    to_stack(a | from_stack())

```

```

def bitwise_xor():
    expect_number()
    a = from_stack()
    expect_number()
    to_stack(a ^ from_stack())

def bitwise_not():
    expect_number()
    to_stack(~from_stack())

def qtime():
    to_stack(int(time.time()*1000000))

def meta_done():
    from_return()
    from_return()

def meta_again():
    from_return()
    to_return(-1)

def meta_if():
    expect_number()
    if from_stack() == 0:
        to_return(from_return() + 1)

def meta_iff():
    expect_number()
    if from_stack() == 0:
        to_return(from_return() + 2)

def meta_else():
    to_return(from_return() + 1)

def meta_literal():
    pc = from_return() + 1
    return_nest = from_return()
    if len(return_nest) == pc:
        failed(''Found a "" at the end of a nest.'')
    to_stack(return_nest[pc])
    to_return(return_nest)
    to_return(pc)

def meta_this():
    pc = from_return()
    return_nest = from_return()
    to_stack(return_nest)
    to_return(return_nest)
    to_return(pc)

```

```

def meta_do():
    expect_something()
    the_thing = from_stack()
    if isOperator(the_thing):
        the_thing()
    elif isNumber(the_thing):
        to_stack(the_thing)
    elif isNest(the_thing):
        to_return(the_thing)
        to_return(-1)
    else:
        failed('Quackery was worried by a python on the stack.')

def meta_bail_by():
    expect_number()
    a = 2*(from_stack())
    if a <= len(rstack):
        for _ in range(a):
            from_return()
    else:
        failed('Bailed out of Quackery.')

def qput():
    expect_nest()
    a = from_stack()
    expect_something()
    b = from_stack()
    a.append(b)

def immovable():
    pass

def take():
    expect_nest()
    a = from_stack()
    if len(a) == 0:
        failed('Unexpectedly empty nest.')
    if len(a) == 1:
        if isNest(a[0]):
            if len(a[0]) > 0:
                if a[0][0] == immovable:
                    failed('Cannot remove an immovable item.')
    to_stack(a.pop())

```

note 8

note 9

note 10

```

def create_nest():
    to_stack([])

def qsplit():
    expect_number()
    a = from_stack()
    expect_nest()
    b = from_stack()
    to_stack(b[:a])
    to_stack(b[a:])

def join():
    expect_something()
    b = from_stack()
    if not isNest(b):
        b = [b]
    expect_something()
    a = from_stack()
    if not isNest(a):
        a = [a]
    to_stack(a+b)

def qsize():
    expect_nest()
    to_stack(len(from_stack()))

def qfind():
    expect_nest()
    nest = from_stack()
    expect_something()
    a = from_stack()
    if a in nest:
        to_stack(nest.index(a))
    else:
        to_stack(len(nest))

def peek():
    expect_number()
    index = from_stack()
    expect_nest()
    nest = from_stack()
    if index >= len(nest) or (
        index < 0 and len(nest) < abs(index)):
        failed('Cannot access an item outside a nest.')
    else:
        to_stack(nest[index])

def poke():
    expect_number()
    index = from_stack()
    expect_nest()
    nest = from_stack().copy()
    expect_something()
    value = from_stack()
    if index >= len(nest) or (
        index < 0 and len(nest) < abs(index)):
        failed('Cannot access an item outside a nest.')
    else:
        nest[index] = value
        to_stack(nest)

```



```

def qnest():
    expect_something()
    bool_to_stack(isNest(from_stack()))

def qnumber():
    expect_something()
    bool_to_stack(isNumber(from_stack()))

def qoperator():
    expect_something()
    bool_to_stack(isOperator(from_stack()))

def quid():
    expect_something()
    to_stack(id(from_stack()))

def qemit():
    expect_number()
    char = from_stack()
    if char == 13:
        print()
    elif 31 < char < 127:
        print(chr(char), end='')
    else:
        print('?', end='')

def ding():
    print('\a', end='')

def qinput():
    prompt = string_from_stack()
    string_to_stack(input(prompt))

```

```

def putfile():
    filename = string_from_stack()
    filetext = string_from_stack()
    try:
        f = open(filename, 'x')
        f.close()
    except FileExistsError:
        to_stack(false)
    except:
        raise
    else:
        try:
            f = open(filename, 'w')
            f.write(filetext)
            f.close()
        except:
            raise
        else:
            to_stack(true)

def releasefile():
    filename = string_from_stack()
    try:
        os.remove(filename)
    except FileNotFoundError:
        to_stack(false)
    except:
        raise
    else:
        to_stack(true)

def sharefile():
    dup()
    filename = string_from_stack()
    try:
        f = open(filename)
        filetext = f.read()
        f.close()
    except FileNotFoundError:
        to_stack(false)
    except:
        raise
    else:
        drop()
        string_to_stack(filetext)
        to_stack(true)

```

```

operators = {
    'fail':      qfail,      # (    $ -->    )
    'nand':      nand,       # (    b b --> b    )
    '=':         equal,      # (    x x --> b    )
    '>':         greater,    # (    n n --> b    )
    '1+':        inc,        # (    n --> n    )
    '+':         plus,       # (    n n --> n    )
    'negate':    negate,     # (    n --> n    )
    '*':         multiply,   # (    n n --> n    )
    '/mod':      qdivmod,    # (    n n --> n n  )
    '**':        exponentiate, # (    n n --> n    )
    '<<':        shift_left,  # (    f n --> f    )
    '>>':        shift_right, # (    f n --> f    )
    '&':         bitwise_and, # (    f f --> f    )
    '|':         bitwise_or,  # (    f f --> f    )
    '^':         bitwise_xor, # (    f f --> f    )
    '~':         bitwise_not, # (    f --> f    )
    'time':      qtime,      # (    --> n    )
    'stacksize': stack_size,  # (    --> n    )
    'nestdepth': nest_depth,  # (    --> n    )
    'return':    qreturn,    # (    --> [    )
    'dup':       dup,        # (    x --> x x   )
    'drop':      drop,       # (    x -->      )
    'swap':      swap,       # (    x x --> x x   )
    'rot':       rot,        # (    x x x --> x x x )
    'over':      over,       # (    x x --> x x x )
    ']done[':    meta_done,   # (    -->      )
    ']again[':   meta_again,  # (    -->      )
    ']if[':      meta_if,     # (    b -->      )
    ']iff[':     meta_iff,    # (    b -->      )
    ']else[':    meta_else,   # (    -->      )
    ']"[':       meta_literal, # (    --> x    )
    ']this[':    meta_this,   # (    --> [    )
    ']do[':      meta_do,     # (    x -->      )
    ']bailby[':  meta_bail_by, # (    n -->      )
    'put':       qput,       # (    x [ -->      )
    'immovable': immovable,   # (    -->      )
    'take':      take,       # (    [ --> x    )
    '[]':        create_nest, # (    --> n    )
    'split':     qsplit,     # (    [ n --> [ [  )
    'join':      join,       # (    x x --> [    )
    'find':      qfind,      # (    x --> b    )
    'peek':      peek,       # (    [ n --> x    )
    'poke':      poke,       # (    x [ n -->      )
    'size':      qsize,      # (    [ --> n    )
    'nest?':     qnest,      # (    x --> b    )
    'number?':   qnumber,    # (    x --> b    )
    'operator?': qoperator,  # (    x --> b    )
    'quid':      quid,       # (    x --> n    )
    'emit':      qemit,      # (    c -->      )
    'ding':      ding,       # (    -->      )
    'input':     qinput,     # (    $ --> $    )
    'putfile':   putfile,    # (    $ --> b    )
    'releasefile': releasefile, # (    $ --> b    )
    'sharefile': sharefile}  # (    $ --> $ b   )

```

```
qstack = []
```

```
rstack = []
```

```

current_nest = []

program_counter = 0

def traverse(the_nest):
    nonlocal current_nest
    nonlocal program_counter
    nonlocal rstack
    current_nest = the_nest
    program_counter = 0
    while True:
        if program_counter >= len(current_nest):
            if rstack == []:
                break
            else:
                program_counter = from_return()
                current_nest = from_return()
                program_counter += 1
                continue
        current_item = current_nest[program_counter]
        if isNest(current_item):
            to_return(current_nest)
            to_return(program_counter)
            current_nest = current_item
            program_counter = 0
        elif isOperator(current_item):
            current_item()
            program_counter += 1
        elif isNumber(current_item):
            to_stack(current_item)
            program_counter += 1
        else:
            failed('Quackery was worried by a python in the nest.')

def isinteger(string):
    numstr = string
    if len(numstr) > 0 and numstr[0] == '-':
        numstr = numstr[1:]
    return numstr.isdigit()

def next_char():
    nonlocal source
    if len(source) > 0:
        char = source[0]
        source = source[1:]
        return(char)
    else:
        return('')

def next_word():
    result = ''
    while True:
        char = next_char()
        if char == '':
            return(result)
        if ord(char) < 33:
            if result == '':
                continue
            return(result)
        result += char

```

note 12

note 13

```

def one_char():
    while True:
        char = next_char()
        if char == '':
            return(char)
        if ord(char) < 33:
            continue
        return(char)

def get_name():
    name = next_word()
    if name == '':
        sys.exit('Unexpected end of program text.')
    return(name)

def check_build():
    nonlocal current_build
    if len(current_build) == 0:
        sys.exit('Unexpected naming.')

def qis():
    nonlocal operators
    nonlocal current_build
    check_build()
    name = get_name()
    operators[name] = current_build.pop()

def qcomment():
    word = ''
    while word != ')':
        word = next_word()
        if word == '':
            sys.exit('Unfinished comment.')

def endcomment():
    sys.exit('Unexpected end of comment.')

def unresolved():
    sys.exit('Unresolved forward reference.')

def forward():
    nonlocal current_build
    current_build.append([unresolved])

def resolves():
    nonlocal current_build
    name = get_name()
    if name in operators:
        if operators[name][0] != unresolved:
            sys.exit(name + ' is not a forward reference.')
        check_build()
        operators[name][0] = current_build.pop()
    else:
        sys.exit(' Unrecognised word: ' + name)

```

```

def char_literal():
    nonlocal current_build
    char = one_char()
    if char == '':
        sys.exit('No character found.')
    current_build.append(ord(char))

def string_literal():
    nonlocal current_build
    delimiter = ''
    result = []
    while delimiter == '':
        char = next_char()
        if char == '':
            sys.exit('No string found.')
        if ord(char) > 32:
            delimiter = char
            char = ''
    while char != delimiter:
        char = next_char()
        if char == '':
            sys.exit('No end of string found.')
        if char != delimiter:
            result.append(ord(char))
    current_build.append([[meta_literal], result])

def ishex(string):
    hexstr = string
    if len(hexstr) > 1 and hexstr[0] == '-':
        hexstr = hexstr[1:]
    for char in hexstr:
        if char not in '0123456789abcdefABCDEF':
            return False
    return True

def hexnum():
    nonlocal current_build
    word = get_name()
    if not ishex(word):
        sys.exit(word + " is not hexadecimal.")
    current_build.append(int(word, 16))

```

```

builders = {'is':      qis,
            '(':      qcomment,
            ')':      endcomment,
            'forward': forward,
            'resolves': resolves,
            'char':    char_literal,
            '$':      string_literal,
            'hex':     hexnum}

current_build = []

source = ''

the_nest = []

def build(source_string):
    nonlocal source
    nonlocal the_nest
    source = source_string
    nesting = 0

    def sub_build():
        nonlocal nesting
        nonlocal current_build
        the_nest = []
        while True:
            current_build = the_nest
            word = next_word()
            if word == '':
                return(the_nest)
            elif word == '[':
                nesting += 1
                the_nest.append(sub_build())
            elif word == ']':
                nesting -= 1
                if nesting < 0:
                    sys.exit('Unexpected end of nest.')
                return(the_nest)
            elif word in builders.keys():
                builders[word]()
            elif word in operators.keys():
                the_nest.append(operators[word])
            elif isinstance(word, int):
                the_nest.append(int(word, 10))
            else:
                sys.exit('Unrecognised word: ' + word)

    the_nest = sub_build()
    if nesting > 0:
        sys.exit('Unfinished nest.')
    return(the_nest)

```

```
predefined = r"""
```

```
( note 16 )
```

[0]	is false	(-->	b)
[1]	is true	(-->	b)
[dup nand]	is not	(b	-->	b)
[nand not]	is and	(b b	-->	b)
[not swap not nand]	is or	(b b	-->	b)
[= not]	is !=	(x x	-->	b)
[not swap not !=]	is xor	(b b	-->	b)
[swap >]	is <	(n n	-->	b)
[negate +]	is -	(n	-->	n)
[/mod drop]	is /	(n n	-->	n)
[swap drop]	is nip	(x x	-->	x)
[/mod nip]	is mod	(n n	-->	n)
[1 swap <<]	is bit	(n	-->	n)
[swap over]	is tuck	(x x	-->	x x x)
[rot rot]	is unrot	(x x x	-->	x x x)
[rot tuck > unrot > not and]	is within	(n n n	-->	b)
[over over]	is 2dup	(x x	-->	x x x x)
[drop drop]	is 2drop	(x x	-->)
[]again[]	is again	(-->)
[]done[]	is done	(-->)
[]if[]	is if	(b	-->)
[]iff[]	is iff	(b	-->)
[]else[]	is else	(-->)
[2dup > if swap drop]	is min	(n n n	-->	n)
[2dup < if swap drop]	is max	(n n n	-->	n)
[rot min max]	is clamp	(n n n	-->	n)
[dup nest? iff [] join]	is copy	([-->	[)
[]'[]	is '	(-->	x)
[]this[]	is this	(-->	[)

[]do[]	is do	(x -->)
[]this[do]	is recurse	(-->)
[not if]again[]	is until	(b -->)
[not if]done[]	is while	(b -->)
[immovable]this[]done[]	is stack	(--> s)
[dup take dup rot put]	is share	(s --> x)
[take drop]	is release	(s -->)
[dup release put]	is replace	(x s -->)
[dup take rot + swap put]	is tally	(n s -->)
[swap take swap put]	is move	(s s -->)
[[] tuck put]	is nested	(x --> [])
[stack []]	is protected	(--> s)
[protected take] '[nested join protected put]	is protect	(-->)
[stack] protect dip.hold	is dip.hold	(--> s)
[dip.hold put] '[do dip.hold take]	is dip	(x --> x)
[rot dip rot]	is 2swap	(x x x x --> x x x x)
[dip [dip 2dup] 2swap]	is 2over	(x x x x --> x x x x x x)
[stack] protect depth	is depth	(--> s)
[depth share 0 != while -1 depth tally]this[do 1 depth tally]	is decurse	(-->)
[dup 0 < if negate]	is abs	(n --> n)
[stack] protect times.start	is times.start	(--> s)
[stack] protect times.count	is times.count	(--> s)
[stack] protect times.action	is times.action	(--> s)

```

[ ]'[ times.action put
  dup times.start put
  [ 1 - dup -1 > while
    times.count put
    times.action share do
    times.count take again ]
  drop
  times.action release
  times.start release ]      is times      (      n -->      )

[ times.count share ]      is i      (      --> n      )

[ times.start share i 1+ - ] is i^      (      --> n      )

[ 0 times.count replace ]   is conclude (      -->      )

[ times.start share
  times.count replace ]     is refresh   (      -->      )

[ times.count take 1+
  swap - times.count put ]   is step     (      --> s      )

[ stack ]                   is temp      (      --> s      )
protect temp

[ immovable
  dup -1 > +
  ]this[ swap peek
  ]done[ ]                   is table     (      n --> x      )

[ [] unrot
  dup 0 = iff 2drop done
  [ 2 /mod over while
    if [ dip [ tuck join swap ] ]
    dip [ dup join ]
    again ] 2drop join ]     is of       (      x n --> [      )
                                                                    ( note 17 )

[ split 1 split
  swap dip join
  0 peek ]                   is pluck     (      [ n --> [ x      )

[ split
  rot nested
  swap join join ]           is stuff     (      x [ n --> [      )

[ 0 pluck ]                   is behead   (      [ --> [ x      )

[ over size over size
  dup temp put
  swap - 1+ times
  [ 2dup over size split
    drop = if
    [ i^ temp replace
      conclude ]
    behead drop ]
  2drop temp take ]         is findseq   (      [ [ --> n      )

```

[13]	is carriage	(--> c)
[carriage emit]	is cr	(-->)
[32]	is space	(--> c)
[space emit]	is sp	(-->)
[dup char a char { within if [32 -]]	is upper	(c --> c)
[dup char A char [within if [32 +]]	is lower	(c --> c)
[dup 10 < iff 48 else 55 +]	is digit	(n --> c)
[stack 10] protect base	is base	(--> s)
[10 base put]	is decimal	(-->)
[\$ '' over abs [base share /mod digit rot join swap dup 0 = until] drop swap 0 < if [\$ '-' swap join]]	is number\$	(n --> \$)
[stack] protect with.hold	is with.hold	(--> s)
[nested ' [dup with.hold put size times] ' [with.hold share i^ peek] rot join nested join ' [with.hold release] join]	is makewith	(x --> [)
[]'[makewith do]	is witheach	([-->)
[witheach emit]	is echo\$	(\$ -->)
[stack] protect mi.tidyup	is mi.tidyup	(--> s)
[stack] protect mi.result	is mi.result	(--> s)
[mi.tidyup put over size mi.result put ' [if [i^ mi.result replace conclude]] join makewith do mi.tidyup take do mi.result take]	is matchitem	([x x --> n)

[]'[]'[matchitem]	is findwith	([--> n)
[size <]	is found	(n [--> b)
[space >]	is printable	(c --> b)
[dup findwith printable [] split nip]	is trim	(\$ --> \$)
[dup findwith [printable not] [] split swap]	is nextword	(\$ --> \$ \$)
[dup nest? if [[] swap witheach [nested swap join]]]	is reverse	(x --> x)
	forward is reflect	
[dup nest? if [[] swap witheach [reflect nested swap join]]]	resolves reflect	(x --> x)
[[] swap times [swap nested join] reverse]	is pack	(* n --> [)
[witheach []]	is unpack	([--> *)
[stack] protect to-do	is to-do	(--> s)
[' done swap put]	is new-do	(s -->)
[dip [1+ pack] put]	is add-to	(* x n s -->)
[[dup take unpack do again] drop]	is now-do	(s -->)
[1 split reverse join now-do]	is do-now	(s -->)
[[dup take ' done = until] drop]	is not-do	(s -->)

```

[ stack ] is sort.test ( --> s )
protect sort.test

[ ]'[ sort.test put
  [ ] swap witheach
    [ swap 2dup findwith
      [ over sort.test share
        do ] [ ]
      nip stuff ]
    sort.test release ] is sortwith ( [ --> [ ] )

[ sortwith > ] is sort ( [ --> [ ] )
[ 32 127 clamp 32 -
  [ table
    0 86 88 93 94 90 92 87 63 64 75 73 82 74 81 76
    1 2 3 4 5 6 7 8 9 10 83 84 69 72 70 85
    91 11 13 15 17 19 21 23 25 27 29 31 33 35 37 39
    41 43 45 47 49 51 53 55 57 59 61 65 78 66 77 80
    89 12 14 16 18 20 22 24 26 28 30 32 34 36 38 40
    42 44 46 48 50 52 54 56 58 60 62 67 79 68 71 0 ]
  ] is qacsfot ( c --> n )

[ [ dup $ ' = iff false done
  over $ ' = iff true done
  behead rot behead rot
  2dup = iff [ 2drop swap ] again
  qacsfot swap qacsfot > ]
  unrot 2drop ] is $< ( $ $ --> b )

[ swap $< ] is $> ( $ $ --> b )

[ sortwith $> ] is sort$ ( [ --> [ ] )

[ upper 47 - 0 44 clamp
  [ table
    -1 0 1 2 3 4 5 6 7 8 9 -1 -1 -1 -1
    -1 -1 -1 10 11 12 13 14 15 16 17 18 19 20 21
    22 23 24 25 26 27 28 29 30 31 32 33 34 35 -1 ]
  dup 0 base share
  within not if [ drop -1 ] ] is char->n ( c --> n )

[ dup $ ' = iff [ drop 0 false ] done
  dup 0 peek char - =
  tuck if [ behead drop ]
  dup $ ' = iff [ 2drop 0 false ] done
  true 0 rot witheach
    [ char->n
      dup 0 < iff [ drop nip false swap ]
      else [ swap base share * + ] ]
  rot if negate
  swap ] is $->n ( $ --> n b )

```

```

(    adapted from 'A small noncryptographic PRNG' by Bob Jenkins    )
(    https://burtleburtle.net/bob/rand/smallprng.html              )

[ hex FFFFFFFFFFFFFFFF ]      is 64bitmask      (      --> f      )

[ 64bitmask & ]                is 64bits        (      f --> f      )

[ dip 64bits 2dup << 64bits
  unrot 64 swap - >> | ]      is rot64          (      f n --> f      )

[ stack 0 ]                    is prng.a         (      --> s      )
[ stack 0 ]                    is prng.b         (      --> s      )
[ stack 0 ]                    is prng.c         (      --> s      )
[ stack 0 ]                    is prng.d         (      --> s      )

[ prng.a share
  prng.b share tuck
  7 rot64 - 64bits swap
  prng.c share tuck
  13 rot64 ^ prng.a replace
  prng.d share tuck
  37 rot64 + 64bits prng.b replace
  over + 64bits prng.c replace
  prng.a share + 64bits
  dup prng.d replace ]        is prng            (      --> n      )

[ hex F1EA5EAD prng.a replace
  dup prng.b replace
  dup prng.c replace
  prng.d replace
  20 times [ prng drop ] ]    is initrandom      (      n -->      )

hex DEFACEABADFACADE initrandom

[ time initrandom ]           is randomise        (      -->      )

[ 64bitmask 1+
  over / over *
  [ prng 2dup > not while
    drop again ]
  nip swap mod ]              is random          (      n --> n      )

[ [] swap dup size times
  [ dup size random pluck
    nested rot join swap ]
  drop ]                      is shuffle          (      [ --> [      )

```

```

[ stack ]                is history      (      --> s      )

[ protected share
  [ dup [] != while
    -1 split 0 peek
    size history put again ]
  drop
  pack dup history put unpack
  stacksize history put
  nestdepth history put
  false history put ]      is backup      (      n -->      )

[ history release
  nestdepth
  history take
  - ]bailby[
  true history put ]      is bail      (      -->      )

[ history take iff
  [ stacksize
    history take
    history share
    size - - times drop
    history take unpack
    protected share
    reverse
    [ dup [] != while
      -1 split 0 peek
      dup size
      history take -
      [ dup 0 > while
        over release
        1 - again ]
      2drop again ]
    drop true ]
  else
    [ protected share
      size 3 + times
      [ history release ]
      false ] ]      is bailed      (      --> b      )

```

```

[ quid swap quid = ]          is oats          (      x x --> b      )

[ [] swap
  [ trim
    dup size while
    nextword nested
    swap dip join again ]
  drop ]          is nest$          (      $ --> [      )

[ stack ]          is namenest        (      --> s      )

[ namenest share ]          is names        (      --> [      )

[ names find names found ]    is name?        (      $ --> b      )

                                forward is actions    (      n --> x      )

[ ' actions ]          is actiontable    (      --> x      )

[ stack ]          is buildernest    (      --> s      )

[ buildernest share ]          is builders    (      --> s      )

[ builders find
  builders found ]          is builder?    (      $ --> b      )

                                forward is jobs        (      n --> x      )

[ ' jobs ]          is jobtable    (      --> [      )

[ stack ]          is message        (      --> s      )

[ stack ]          is b.nesting    (      --> s      )
protect b.nesting

[ stack ]          is b.to-do        (      --> s      )

[ $ '[' b.nesting put
  [] swap ]          is b.[          (      [ $ --> [ [ $      )

[ b.nesting take dup
  $ '' = if
    [ $ 'Unexpected "]"'. '
      message put bail ]
  dup $ '[' = iff drop
  else
    [ $ 'Nest mismatch: '
      swap join $ ' ' join
      message put bail ]
  dip [ nested join ] ]          is b.]          (      [ [ $ --> [ $      )

```



```

[ over [] = if
  [ $ '"is" needs something to name.'
    message put bail ]
dup $ '' = if
  [ $ '"is" needs a name after it.'
    message put bail ]
nextword nested
namenest take
join
namenest put
dip
  [ -1 pluck
    actiontable take
    1 stuff
    actiontable put ] ]      is b.is      (      [ $ --> [ $      )

[ over [] = if
  [ $ '"builds" needs something to name.'
    message put bail ]
dup $ '' = if
  [ $ '"builds" needs a name after it.'
    message put bail ]
nextword nested
buildernest take
join
buildernest put
dip
  [ -1 pluck
    jobtable take
    1 stuff
    jobtable put ] ]      is b.builds      (      [ $ --> [ $      )

[ trim nextword
dup $ '' = if
  [ $ 'Unfinished comment.'
    message put bail ]
$ ') ' = until ]      is b.(      (      [ $ --> $ [      )

[ $ 'Unexpected ")"'. '
message put bail ]      is b.)      (      [ $ --> $ [      )

```

```

[ $ 'Unresolved reference.'
  fail ] is unresolved ( --> )

[ dip
  [ ' [ unresolved ]
    copy nested join ] ] is b.forward ( [ $ --> [ $ ] )

[ over [] = if
  [ $ '"resolves" needs something to resolve with.'
    message put bail ]
  dup $ '' = if
    [ $ '"resolves" needs a name to resolve.'
      message put bail ]
    dip [ -1 split ]
    nextword dup temp put
    names find
    dup names found not if
      [ $ 'Unknown word after "resolves": '
        temp take join
        message put bail ]
    actions
    dup ' [ unresolved ] = not if
      [ char " temp take join
        $ '" is not an unresolved forward reference.'
        join
        message put bail ]
    rot 0 peek over
    replace
    ' unresolved swap
    ' replace 2 b.to-do add-to
    temp release ] is b.resolves ( [ $ --> [ $ ] )

[ 1 split
  over $ '' = if
    [ $ '"char" needs a character after it.'
      message put bail ]
    dip join ] is b.char ( [ $ --> [ $ ] )

[ dup $ '' = if
  [ $ '"$" needs to be followed by a string.'
    message put bail ]
  behead over find
  2dup swap found not if
    [ $ 'Endless string discovered.'
      message put bail ]
  split behead drop
  ' ' nested
  rot nested join
  nested swap dip join ] is b.$ ( [ $ --> [ $ ] )

[ dup $ '' = if
  [ $ '"say" needs to be followed by a string.'
    message put bail ]
  $ '$' builders find jobs do
  dip
    [ -1 pluck
      ' echo$ nested join
      nested join ] ] is b.say ( [ $ --> [ $ ] )

```

```

[ 16 base put
  nextword dup
  $ ' ' = if
    [ $ '"hex" needs a number after it.'
      message put bail ]
  dup $->n iff
    [ nip swap dip join ]
  else
    [ drop
      char " swap join
      $ ' ' is not hexadecimal.'
      join message put bail ]
  base release ] is b.hex ( [ $ --> [ $ ] )

[ dip [ -1 split ] swap do ] is b.now! ( [ $ --> [ $ ] )

[ over [] = if
  [ $ '"constant" needs something before it.'
    message put bail ]
  dip
    [ -1 pluck do
      dup number? not if
        [ ' ' nested swap
          nested join
          nested ]
    join ] ] is b.constant ( [ $ --> [ $ ] )

[ ' [ namenest actiontable
  buildernest jobtable ]
  witheach
    [ do share copy
      history put ] ] is backupwords ( --> )

[ ' [ jobtable buildernest
  actiontable namenest ]
  witheach
    [ do dup release
      history swap move ] ] is restorewords ( --> )

[ 4 times
  [ history release ] ] is releasewords ( --> )

```

```

[ backupwords
  b.to-do new-do
  1 backup
    [ $ '' b.nesting put
      decimal
      [] swap
      [ trim
        dup $ '' = iff drop done
        nextword
        dup builders find
        dup builders found iff
          [ dip [ drop trim ]
            jobs do ] again
        drop
        dup names find
        dup names found iff
          [ actions nested
            nip swap dip join ] again
        drop
        dup $->n iff
          [ nip swap dip join ] again
        drop
        $ 'Unknown word: '
        swap join message put bail ]
      base release
      b.nesting take dup
      $ '' = iff drop
    else
      [ $ 'Unfinished nest: '
        swap join message put bail ] ]
    bailed iff
      [ drop b.to-do now-do
        restorewords
        ' ' nested
        message take nested join
        ' echo$ nested join ]
    else
      [ b.to-do not-do
        releasewords ] ]      is build      (      $ --> [      )
[ build do ]                is quackery      (      $ -->      )

```

```

[ stack -1 ]                is nesting      (      --> [      )

                                forward is unbuild (      x --> $      )

[ nesting share
  0 = iff [ drop $ '...' ] done
  $ '' swap
  dup number? iff
    [ number$ join ] done
  actiontable share
  behead drop
  [ dup [] = iff
    [ drop false ] done
    behead
    rot tuck oats iff
      [ drop size 2 +
        actiontable share
        size swap -
        names swap peek join
        true ] done
    swap again ]
  if done
  dup nest? iff
    [ $ '[' rot join swap
      [ dup [] = iff drop done
        behead
        -1 nesting tally
        unbuild
        1 nesting tally
        space join
        swap dip join again ]
      $ ']' join ] ] resolves unbuild (      x --> $      )

[ unbuild echo$ ]           is echo          (      x -->      )

[ $ ''
  return -2 split drop
  witheach
    [ dup number? iff
      [ number$ join
        $ '}' join ]
    else
      [ $ '{' swap dip join
        actiontable share
        findwith
          [ over oats ] drop
        dup actiontable share
        found iff
          [ 1 - names swap
            peek join
            space join ]
        else
          [ drop $ '[...] '
            join ] ] ]
  -1 split drop ]           is return$      (      --> $      )

[ return$ echo$ ]          is echoreturn    (      -->      )

```

```

[ stacksize dup 0 = iff
  [ $ 'Stack empty.' echo$ drop ]
else
  [ $ 'Stack: ' echo$
    pack dup
    witheach [ echo sp ]
    unpack ]
cr ] is echostack ( --> )

[ cr $ '' $ '/O> '
  [ input
    dup $ '' != while
    carriage join join
    $ '... ' again ]
  drop
  quackery
  5 nesting put
  cr echostack
  nesting release again ] is shell ( --> )

[ cr randomise 12 random
  [ table
    $ 'Goodbye.' $ 'Adieu.' $ 'So long.'
    $ 'Cheerio.' $ 'Aloha.' $ 'Ciao.'
    $ 'Farewell.' $ 'Be seeing you.'
    $ 'Sayonara.' $ 'Auf wiedersehen.'
    $ 'Toodles.' $ 'Hasta la vista.' ]
  do echo$ cr cr
  3 ]bailby[ ] is leave ( --> )

[ stacksize times drop ] is empty ( all --> )

[ tuck temp put
  witheach
    [ dup size
      rot + dup
      temp share > iff
        [ cr drop dup size ]
      else sp 1+ swap echo$ ]
  drop temp release ] is wrap$ ( [ n --> )

[ names reverse 70 wrap$ cr
  builders reverse
  70 wrap$ cr ] is words ( --> )

[ dup name? iff drop
  else
    [ dup sharefile not if
      [ $ |$ 'file not found: "|
        swap join
        $ |"' echo$| join ]
      nip quackery ] ] is loadfile ( $ --> )

[ dup sharefile iff
  [ swap releasefile ]
  else [ drop false ] ] is takefile ( $ --> $ b )

[ dup releasefile iff
  putfile
  else [ 2drop false ] ] is replacefile ( $ $ --> b )

```

```
[ nested ' [ ' ]
  swap join
  decimal unbuild
  base release ]           is quackify      (      x --> $      )
                                     ( note 18 )
```

```
$ "quackify replacefile takefile loadfile words empty wrap$ leave
  shell echostack echoreturn return$ echo unbuild nesting quackery
  build releasewords restorewords backupwords unresolved b.to-do
  b.nesting message jobtable jobs builder? builders buildernest
  actiontable actions name? names namenest nest$ oats bailed bail
  backup history shuffle random randomise initrandom prng prng.d
  prng.c prng.b prng.a rot64 64bits 64bitmask $->n char->n sort$
  $> $< qacsfoot sort sortwith sort.test not-do do-now now-do
  add-to new-do to-do unpack pack reflect reverse nextword trim
  printable found findwith matchitem mi.result mi.tidyup echo$
  witheach makewith with.hold number$ decimal base digit lower
  upper sp space cr carriage findseq behead stuff pluck of table
  temp step refresh conclude i^ i times times.action times.count
  times.start abs decurse depth 2over 2swap dip dip.hold protect
  protected nested move tally replace release share stack while
  until recurse do this ' copy clamp max min else iff if done
  again 2drop 2dup within unrot tuck bit mod nip / - < xor != or
  and not true false sharefile releasefile putfile input ding emit
  quid operator? number? nest? size poke peek find join split []
  take immovable put ]bailby[ ]do[ ]this[ ]'[ ]else[ ]iff[ ]if[
  ]again[ ]done[ over rot swap drop dup return nestdepth stacksize
  time ~ ^ | & >> << ** /mod * negate + 1+ > = nand fail"
nest$ namenest put
```

```
[ table
  quackify replacefile takefile loadfile words empty wrap$ leave
  shell echostack echoreturn return$ echo unbuild nesting quackery
  build releasewords restorewords backupwords unresolved b.to-do
  b.nesting message jobtable jobs builder? builders buildernest
  actiontable actions name? names namenest nest$ oats bailed bail
  backup history shuffle random randomise initrandom prng prng.d
  prng.c prng.b prng.a rot64 64bits 64bitmask $->n char->n sort$
  $> $< qacsfoot sort sortwith sort.test not-do do-now now-do
  add-to new-do to-do unpack pack reflect reverse nextword trim
  printable found findwith matchitem mi.result mi.tidyup echo$
  witheach makewith with.hold number$ decimal base digit lower
  upper sp space cr carriage findseq behead stuff pluck of table
  temp step refresh conclude i^ i times times.action times.count
  times.start abs decurse depth 2over 2swap dip dip.hold protect
  protected nested move tally replace release share stack while
  until recurse do this ' copy clamp max min else iff if done
  again 2drop 2dup within unrot tuck bit mod nip / - < xor != or
  and not true false sharefile releasefile putfile input ding emit
  quid operator? number? nest? size poke peek find join split []
  take immovable put ]bailby[ ]do[ ]this[ ]'[ ]else[ ]iff[ ]if[
  ]again[ ]done[ over rot swap drop dup return nestdepth stacksize
  time ~ ^ | & >> << ** /mod * negate + 1+ > = nand fail ]
```

```
resolves actions      (      n --> x      )
```

```
$ "constant now! hex say $ char resolves forward ) ( builds is ] ["
nest$ buildernest put
```

```
[ table
  b.constant b.now! b.hex b.say b.$ b.char b.resolves
  b.forward b.) b.( b.builds b.is b.] b.[ ]
```

```
resolves jobs ( n --> x )
```

```
"""
```

```

traverse(build(predefined)) # note 19
while(True):
    to_stack([ord(char) for char in source_string])
    try:
        traverse(build('quackery'))
    except QuackeryError as diagnostics:
        if __name__ == '__main__' and len(sys.argv) == 1:
            print(diagnostics)
            continue
        else:
            raise
    except Exception as diagnostics:
        print('Quackery system damage detected.')
        print('Python reported: ' + str(diagnostics))
        sys.exit()
    else:
        traverse(build('stacksize pack decimal unbuild'))
        result = ''
        for ch in (qstack[0][2:-2]):
            result += chr(ch)
        return(result)

```



```

if __name__ == '__main__':
    if len(sys.argv) > 1:
        filename = sys.argv[1]
        try:
            f = open(filename)
            filetext = f.read()
            f.close()
        except FileNotFoundError:
            print('Cannot find file "' + filename + '"')
        else:
            try:
                print(quackery(filetext))
                print()
            except QuackeryError as diagnostics:
                print()
                print('Quackery crashed.')
                print()
                print(diagnostics)
                print()
            except Exception as diagnostics:
                print('Quackery system damage detected.')
                print('Python reported: ' + str(diagnostics))
                sys.exit()
    else:
        print()
        print('Welcome to Quackery.')
        print()
        print('Enter "leave" to leave the shell.')
        quackscript = r"""
$ 'extensions.qky' dup name? not
dip sharefile and iff
[ cr say 'Building extensions.' cr quackery ]
else drop

shell """

    try:
        quackery(quackscript)
        print()
    except QuackeryError as diagnostics:
        print()
        print('Quackery crashed.')
        print()
        print(diagnostics)
        print()

```

Under the Quackery Bonnet

A few notes about the code. Mostly things that were not immediately obvious about Python, as I was new to the language when I started coding, and took longer than they needed to for me to figure out, and a couple of things that I feel the need to mention about the coding style and Quackery code.

1. This gives the shell a handy way of accessing previously entered text via the up and down arrows. It is not supported by Python for Windows, hence the `try`.
2. When a virtual processor problem arises, Quackery generates a `QuackeryError` and passes diagnostic information back to Python. The simplest way to format the diagnostics is with a short Quackery program, compiled with the Python Quackery compiler `build()`, and run using the Virtual Processor, `traverse()`. During development these tools were not available, so the Quackery Stack and Return Stack were printed as Python lists.

Using Quackery code relies on large portions of the Quackery dictionary being viable after the crash, so there are circumstances where this will not work. This is addressed in the Python script at the end of the file, which deals with the possibility of an exception being raised while handling an exception, leading to infinite recursion, for each of the various ways that Quackery can be used (as a function inside a Python program, in the terminal using the Quackery shell, or with a filename passed to it as an argument in the terminal shell.)

3. “`str(thestack)[2:-2]`” If you’re not familiar with Python, this colon syntax is called “slice notation”. It’s difficult to search online for information about syntax based functionality when you don’t have the right word for it.
4. There are a lot of very short functions. The Python code is more “Quackery style” than “Pythonic” – this is a side effect of thinking Quackery while coding Python. Hopefully, this makes the code more readable for you, not less.
5. Quackery nests are Python lists, Quackery numbers are Python ints, and Quackery operators are Python functions, but there is no test in Python to say “is this a function” so we have to say “does this look like this example of a Python function. (Python does not distinguish between named functions and lambda functions.) `isoperator()` is not a strict test for Quackery operators, as a Python function which is not a Quackery operator would pass the test, but the Quackery compilers will not compile any Python functions other than the ones they find in their dictionaries, so that’s OK.
6. “`nonlocal`” – Variables are not declared in Python. If Python encounters a word it has not seen before, it will assume that it is a local variable unless it has been told otherwise. Hence it has the declarations `global` (not used in this code) and `nonlocal`, which tells it that, in this instance, the variable `qstack` is local to the function `quackery()`, of which `to_stack()` is a subfunction.
7. “`x if y then z`” is Python’s ternary operator. Evaluate `y`. If `y` returns `True`, evaluate `x` and return the value `x` returns, otherwise evaluate `z` and return the value `z` returns.
8. Quackery’s method of treating a function as data is with `'` and `do`, so if you have defined a word `xyz`, `' xyz do` is equivalent to `xyz`. Python handles this by ripping the brackets off a function and sticking them on a variable.

So if you have defined a function `xyz()`, and `abc` is a variable, then

```
abc = xyz
```

```
abc()
```

is equivalent to `xyz()`

9. How to define a no-op in Python.
10. Really quite a lot of nested tests, just to allow programmers to abuse **take** by applying it to non-immovable nests. This is the cost of combining leniency with error checking.
11. “**operators**” is a slight misnomer for this dictionary. This page of the code does provide a handy summary of the Quackery operators, prior to defining the virtual processor, **traverse()**, on the next page, but the Python Quackery compiler that follows will use it as a dictionary of all the predefined Quackery words, not just the operators.
12. The Quackery virtual processor is a classic depth first binary tree traversal. It differs in that Python lists are arrays, not linked lists, so traversing the right branch consists of incrementing the variable **program_counter**, and in that rather than being a recursive function the return stack is explicit and needs to be managed. This allows the meta control flow operators to modify the return stack.

The actions of pushing a number onto the stack, jumping to a subroutine and returning from a subroutine are implied by the presence of a number, the presence of a nest, and the end of the array respectively. This simplifies the compiler and allows us to easily construct nests under program control generally, but requires that we explicitly test for numbers, nests and operators in the virtual processor.

13. The Python Quackery compiler differs in several ways from the Quackery Quackery compiler. This is the first – it nibbles away at the source string one character at a time, whereas the Quackery Quackery compiler bites off one word at a time. Some of the differences are because different languages lend themselves to different approaches, other differences are because when I wrote the Quackery Quackery compiler I had the experience of having written a Quackery compiler before.

The Python compiler works fine in its task of compiling the predefined portion of Quackery written in Quackery and does not need fixing, so I did not retrospectively modify the Python Quackery compiler in light of what I had learned implementing the Quackery Quackery compiler.

14. The Python Quackery builders dictionary is exclusively for the use of the Python Quackery compiler, and provides the minimal functionality required to compile the string of predefined Quackery words that follows the compiler in the source code. **builds** is not present as the Python Quackery compiler uses **is** to aggregate all the definitions into the Python operators dictionary. Words destined for the Quackery builders & jobs dictionary are named with **b.** at the start of the name to differentiate them from their counterparts in the Python Quackery builders dictionary.

The compiler directives **[** and **]** are not present in the Python builders dictionary as the Python Quackery compiler treats them as a special case. This is down to the order in which the compiler was developed. I wanted to be able to create nests before I put the compiler directives mechanism in place, and because it allows the compiler to be defined simply, as a recursive function.

15. Strictly speaking, **sub_build()** is the Python Quackery compiler. **build()** is just a harness to make it easier to check for balanced nesting.
16. The **r** in **r ""** stands for “raw” – it allows us to include backslashes in a string without Python treating them as a special character. Strings defined with **r ""** cannot have the sequence of characters **""** inside them.

17. The definition of `of` may seem unnecessarily convoluted when

```
[ [] swap times [ over join ] nip ] is of ( x n -> [ ] )
```

would work just fine. The convoluted method is more efficient when creating very large nests. It is a variation on Russian peasant multiplication.

https://en.wikipedia.org/wiki/Ancient_Egyptian_multiplication#Russian_peasant_multiplication

18. Having added all the definitions of the predefined portion of Quackery to the Python operators dictionary, the final step is to populate the Quackery names & actions and builders & jobs dictionaries from it.

19. Finally, after almost 46k of sub-functions and other preamble, here is the definition of the Python function `quackery()`

1. Build the quackery dictionaries by compiling the predefined string with the Python Quackery compiler and then traversing the resulting nest.
2. Load the Python string passed to `quackery()` as a parameter onto the Quackery stack as a Quackery string.
3. Compile and run the string “quackery”, which compiles the string on the nest using the Quackery Quackery compiler (`build`) and traverses it with “do”.
4. Return the Quackery stack as a Python string as the result of evaluating the Python function `quackery()`.

The rest of the definition addresses what to do if an exception is raised.

```
if __name__ == '__main__' and len(sys.argv) == 1: treats the Quackery shell as  
a special case, when Quackery crashes in the shell with a Quackeryerror, it is restarted.
```

20. The final page is run if Quackery is run from the terminal, rather than the function `quackery()` being invoked from a Python script.