Algorithmic Functions as Mathematical Expressions

Jake Saville

March 1, 2017

1 Introduction

Most equations and mathematical functions are defined by a process, a list of inputs and an output. By definition, a a function performs the same process on the inputs, regardless of their values. For example, consider the function $f(x) = 3x$. The input here is x and the process is *multiply by 3*. This process is the same regardless of the value of x. However, often this is not enough to calculate what we need to calculate, as the process we need to apply to the inputs might need to be different depending on their values. At this point, one might abandon the use of a mathematical function altogether and branch into the domain of programming and algorithms. These tools provide the use of conditional statements, loops and boolean (true or false) values. An example of this would be the Collatz [1] function. This function halves the input if the input is even, and triples it and adds 1 if the input is odd. As a mathematical function, this would be wordy and unsatisfying; it may look something like

$$\begin{aligned} f(x) &= \frac{x}{2} \text{ if } x \text{ is even} \\ &= 3x + 1 \text{ if } x \text{ is odd} \end{aligned}$$

This is an issue because it would be difficult to pass this function to a computer program such as Wolfram Alpha[2] for computation. Another issue is that this uses english words (or the language of choice for the mathematician writing) and one of the main benefits of writing equations is that they are universal. In this paper, we will discuss how we could write this (or indeed any function which would require loops or conditionals) as one single analytical equation, recognisable by any mathematician (despite being somewhat less readable in some cases).

2 Boolean Logic

If we are to step into the domain of logical conditions, we will need to establish a method of indicating whether a the result of a condition is true or false. In this paper, we will be using the number 0 to mean a logical false, and 1 to mean a logical true. Please note that this is profoundly different from the field of boolean algebra. In boolean algebra, there exist no values other than 0 and 1, and they have properties different to their properties in regular arithmetic. When we use 0 and 1 in this paper they can refer to either a logical state *or* the values normally associated with

[1] http://www.mathworld.wolfram.com/CollatzProblem.html
[2] https://www.wolframalpha.com

them. Both cases are treated exactly the same and have the same properties, however they have different connotations to the mathematician. That is to say that a logical true plus a logical true will equal the same value as the number 1 plus the number 1; both cases equal the number 2.

3 Functional Notation

In this paper, we will use slightly different notation to that in the rest of mathematics, however that is simply for readability and clarity, and any function defined herein can be written like a regular function and still be just as correct. For the purposes of this paper, we will allow function names to be multiple characters in length, so as to be more descriptive of their purpose. As such, function parameters must be encased in parentheses when nesting functions, e.g. $second(first(x))$. Also, we will define functions with what will be referred to as a *function stub*. This serves no inherent mathematical purpose and it does not affect the way in which the function operates, but instead creates a framework to communicate what type of inputs the function takes, and what type of output it creates. The symbol n shall be used to represent a regular number, whereas b shall be used to represent a boolean value (see above section on boolean logic). For example, a logical *or* operation takes two boolean values, and returns a boolean value. As such, its stub shall look like this:

$or(b, b) \to b$

Similarly, a function which takes a number and a boolean value, and returns another number will have a stub like below:

$function(n, b) \to n$

Of course, the stub gives no indication of the actual process carried out by the function, and so is used in conjunction with a regular function definition. For example, if we were to define a function named *triple* which would take an input and return a number of triple its value, we would define it like this:

$triple(n) \to n$
$triple(x) = 3x$

With the first line being the function stub, defining the types of its inputs and output, and the second line defining the process. Please be aware that the variable symbols (x in the above example) do not need to be either b or n.

4 Basic Functions

Let's return to the previous example of the Collatz function. This is quite a simple case because there is an existing mathematical method of determining whether a given number is even or odd. This is achieved using the modulo operator, which finds the remainder of a division. The expression $x \bmod 2$ will be equal to 0 if x divides by 2 with a remainder of 0 (that is to say that x is even) and will be 1 if x divides by 2 with a remainder of 1 (x is odd). Assuming we are working with only integers, these are the only two values the expression can equal. As such, we can treat them as boolean values. A 1 means that x is odd and a 0 means that x is even. We can therefore define the following function:

$isOdd(n) \to b$
$isOdd(x) = x \bmod 2$

Now what we need is a function which returns one number if a boolean value is true, and another number if it is false. We will call this function if, and its stub will be $if(b, n, n) \to n$. If its boolean parameter, which will be called x, is true, it will return the first number parameter, which will be called a, otherwise it will return the second number parameter, which will be called b.

Before defining the function, we will first consider the expression $x(a - b)$, which uses the same variables x, a and b defined above. It is apparent that when x is 0, this expression will equal 0 and when x is 1, it will equal $a - b$. However, with more complex expressions it can be tricky to keep track of the different values the expression can have. As such, we can use a truth table to visualise the relationship between an expression and its boolean variables. The left hand columns will contain the different values of each of the boolean variables and the right hand columns will contain the values of an expression with the boolean variables as defined in their row. For example, the above expression $x(a - b)$ will have the following truth table:

x	$x(a-b)$
0	0
1	$a-b$

A truth table can contain several different expressions on the right hand side. This can be used to keep track of a very complicated expression at the same time as various other simpler expressions which make it up. For example, we could use a more complicated expression such as $x(a - b) + b$. We can use the existing column in the truth table to help calculate the rows of this new column.

x	$x(a-b)$	$x(a-b)+b$
0	0	b
1	$a-b$	a

As you can see above, the expression $x(a - b) + b$ equals a if x is 1, and b if x is 0, which is what we needed for our if function. We can now define it.

$if(b, n, n) \to n$
$if(x, a, b) = x(a - b) + b$

Remember that, as with regular functions, parameters passed to these functions don't need to be values, but can also be expressions or nested functions. We can now define our Collatz function.

$isOdd(n) \to b$
$isOdd(x) = x \bmod 2$

$if(b, n, n) \to n$
$if(x, a, b) = x(a - b) + b$

$collatz(n) \rightarrow n$
$collatz(x) = if(isOdd(x), 3x+1, \frac{x}{2})$

We can now substitute the *isOdd* and *if* functions into the *collatz* function like so:

$$\begin{aligned} collatz(x) &= if(isOdd(x), 3x+1, \frac{x}{2}) \\ &= if(x \bmod 2, 3x+1, \frac{x}{2}) \\ &= (x \bmod 2)(3x+1-\frac{x}{2}) + \frac{x}{2} \end{aligned}$$

And we can now simplify to

$collatz(x) = (x \bmod 2)(\frac{5x}{2}+1) + \frac{x}{2}$

We can check that this works using a truth table

x	$x \bmod 2$	$(x \bmod 2)(\frac{5x}{2}+1)$	$(x \bmod 2)(\frac{5x}{2}+1)+\frac{x}{2}$
Even	0	0	$\frac{x}{2}$
Odd	1	$\frac{5x}{2}+1$	$3x+1$

5 Checking Equality

Please note that for the rest of this paper, previously defined functions such as *isOdd*, *if*, *collatz* or any other function we define will not be redefined when used. For your reference, there will be a list of all functions defined at the end of the paper.

5.1 Equality to Zero

In order to define a function which checks if two numbers are equal, we must first define one which checks whether a given number equals 0. We will call this function *equalsZero* and its stub will be $equalsZero(n) \rightarrow b$.

Consider the expression $\frac{x}{x+1}$. A table of values for this expression is shown below

x	0	1	2	3	4
$\frac{x}{x+1}$	0	0.5	$0.\dot{6}$	0.75	0.8

You will note that when x is 0 the expression equals 0 and that when x is not 0 the expression equals some number between 0 and 1 (non-inclusive). However, if x is -1, then the expression is $\frac{-1}{0}$ which involves division by zero. To prevent this, we can change the expression to $\frac{|x|}{1+|x|}$ to make sure that the positive value of x is used. The above table of values still applies, as does the assertion that when x equals 0 the expression equals 0, otherwise it equals some value between 0 and 1. Now all we must do is round the expression up to the next highest integer using a ceiling function: $\lceil \frac{|x|}{1+|x|} \rceil$. This results in the following values:

x	-1	0	1	2	3	4
$\frac{x}{x+1}$	0.5	0	0.5	$0.\dot{6}$	0.75	0.8
$\lceil \frac{\lvert x\rvert}{1+\lvert x\rvert} \rceil$	1	0	1	1	1	1

This is close to our *equalsZero* function, except negated. This expression equals 0 if x equals 0, otherwise it equals 1. We will call this function *doesntEqualZero*.

$doesntEqualZero(n) \to b$
$doesntEqualZero(x) = \lceil \frac{|x|}{1+|x|} \rceil$

Now we need a function which negates a boolean value. We will call this function *not*, and it is defined as follows:

$not(b) \to b$
$not(x) = 1 - x$

And now we can define *equalsZero* as

$$\begin{aligned} &equalsZero(n) \to b \\ &equalsZero(x) = not(doesntEqualZero(x)) \\ &\qquad = not(\lceil \frac{|x|}{1+|x|} \rceil) \\ &\qquad = 1 - \lceil \frac{|x|}{1+|x|} \rceil \end{aligned}$$

We can check this with a table of values.

x	-1	0	1	2	3	4
$\frac{x}{x+1}$	0.5	0	0.5	$0.\dot{6}$	0.75	0.8
$\lceil \frac{\lvert x\rvert}{1+\lvert x\rvert} \rceil$	1	0	1	1	1	1
$1 - \lceil \frac{\lvert x\rvert}{1+\lvert x\rvert} \rceil$	0	1	0	0	0	0

5.1.1 Checking Divisibility

One simple use for the *equalsZero* function is checking whether a number a divides perfectly by another number b. All we have to get the remainder of the division using the modulo operator and check if it is equal to 0.

$$\begin{aligned} &divides(n, n) \to b \\ &divides(a, b) = equalsZero(a \bmod b) \\ &\qquad = 1 - \lceil \frac{|a \bmod b|}{1 + |a \bmod b|} \rceil \end{aligned}$$

In this particular case (and certain others) we can remove the absolute pipes, as the modulo operator can only ever produce a positive number or zero.

$divides(a, b) = 1 - \lceil \frac{a \bmod b}{1+(a \bmod b)} \rceil$

5.2 General Equality

Now that we have defined a function to check if a number is 0, it is trivial to make one to check if two numbers are equal. This question can be phrased as ”is the difference between two numbers equal to zero?". The solution is as follows

$$\begin{aligned}
&equals(n,n) \rightarrow b\\
&equals(a,b) = equalsZero(a-b)\\
&\qquad\qquad = 1 - \lceil \frac{|a-b|}{1+|a-b|} \rceil
\end{aligned}$$

6 Checking Inequality

We've created a function for checking if a is equal to b, but we can also create a function to check whether a is less than b or a is greater than b. We'll start with the former. If a is less than b, then $a-b$ will be negative. Otherwise it will be positive or 0. This means that if the value of $a-b$ is not equal to its absolute value then a is less than b.

$$\begin{aligned}
&lessThan(n,n) \rightarrow b\\
&lessThan(a,b) = not(equals(a-b, |a-b|))\\
&\qquad = not(1 - \lceil \frac{|a-b-|a-b||}{1+|a-b-|a-b||} \rceil)\\
&\qquad = 1 - (1 - \lceil \frac{|a-b-|a-b||}{1+|a-b-|a-b||} \rceil)\\
&\qquad = \lceil \frac{|a-b-|a-b||}{1+|a-b-|a-b||} \rceil
\end{aligned}$$

On the other hand, if a is greater than b, it implies that b is less than a. As such it is easy to define a *greaterThan* function

$$\begin{aligned}
&greaterThan(n,n) \rightarrow b\\
&greaterThan(a,b) = lessThan(b,a)\\
&\qquad = \lceil \frac{|b-a-|b-a||}{1+|b-a-|b-a||} \rceil
\end{aligned}$$

Inclusive inequalities, such as less than or equal to and greater than or equal to, are also easy to define from here. If a is less than or equal to b, it simply means that a is not greater than b.

$$\begin{aligned}
&lessThanOrEqualTo(n,n) \rightarrow b\\
&lessThanOrEqualTo(a,b) = not(greaterThan(a,b))\\
&\qquad = not(\lceil \frac{|b-a-|b-a||}{1+|b-a-|b-a||} \rceil)\\
&\qquad = 1 - \lceil \frac{|b-a-|b-a||}{1+|b-a-|b-a||} \rceil
\end{aligned}$$

Similarly, if a is greater than or equal to b, it means that a is not less than b.

$greaterThanOrEqualTo(n, n) \to b$
$$\begin{aligned} greaterThanOrEqualTo(a, b) &= not(lessThan(a, b)) \\ &= not(\lceil \frac{|a-b-|a-b||}{1+|a-b-|a-b||} \rceil) \\ &= 1 - \lceil \frac{|a-b-|a-b||}{1+|a-b-|a-b||} \rceil \end{aligned}$$

7 Logical Operators

We've already defined a *not* function which negates a boolean value, but there are several other logical operators which we can implement in this way.

7.1 And Operator

Other than *not*, *and* is perhaps the easiest logical operator to implement. It takes two boolean variables and only returns 1 is both of the parameters are 1. Otherwise, it returns 2. This can be achieved by multiplying the two boolean values, as shown in the following truth table.

x	y	xy
0	0	0
0	1	0
1	0	0
1	1	1

It is therefore defined as follows:

$and(b, b) \to b$
$and(x, y) = xy$

7.2 Or Operator

or is a little bit more difficult to implement. Like *and*, it takes two boolean parameters. It will return 1 if either or both of its inputs are 1. That is to say that the sum of its inputs are anything other than 0. It can be defined as

$or(b, b) \to b$
$$\begin{aligned} or(x, y) &= doesntEqualZero(x + y) \\ &= \lceil \frac{|x+y|}{1+|x+y|} \rceil \\ &= \lceil \frac{x+y}{1+x+y} \rceil \end{aligned}$$

Note that we can remove the absolute pipes because the sum of two boolean values can never be anything other than 0, 1 or 2. The *or* function is shown below in a truth table.

x	y	$x+y$	$\frac{x+y}{x+y+1}$	$\lceil\frac{x+y}{1+x+y}\rceil$
0	0	0	0	0
0	1	1	0.5	1
1	0	1	0.5	1
1	1	2	$0.\dot{6}$	1

7.3 Xor Operator

Perhaps the most complicated logical is *xor* (or exclusive *or*). It takes two boolean variables and will return 1 if either *but not both* of its parameters are 1. In other words, it returns 1 if its two parameters have different values.

$$\begin{aligned}
&xor(b, b) \to b \\
&xor(x, y) = not(equals(x, y)) \\
&\qquad = not(1 - \lceil\frac{|x-y|}{1+|x-y|}\rceil) \\
&\qquad = 1 - (1 - \lceil\frac{|x-y|}{1+|x-y|}\rceil) \\
&\qquad = \lceil\frac{|x-y|}{1+|x-y|}\rceil
\end{aligned}$$

Its truth table is shown below.

x	y	$x-y$	$\lvert x-y\rvert$	$\frac{\lvert x-y\rvert}{1+\lvert x-y\rvert}$	$\lceil\frac{\lvert x-y\rvert}{1+\lvert x-y\rvert}\rceil$
0	0	0	0	0	0
0	1	-1	1	0.5	1
1	0	1	1	0.5	1
1	1	0	0	0	0

8 Loops

Loops are often essential to implementing an algorithm. In most programming languages, there are two types of loop: *For* loops and *While* loops. *For* loops are much simpler to implement as an analytical function, and as such we will discuss them first. However, both types of loops (or systems which act similarly) are indeed possible.

8.1 For Loops

Consider the following Python code:

```python
total = 0
for k in range(10,100,5):
    total += sub(k)
```

In this snippet, imagine that the function *sub* is defined somewhere else in the program. What this code does is use a counter variable k which starts with a value of 10 and increases by 5 until it reaches 100. For each different value of k, the value of $sub(k)$ is added to the total. This functionality is similar to a sigma in mathematics. The same piece of code can be written as the following expression:

$$\sum_{k=0}^{\frac{100-10}{5}} sub(10+5k)$$

There are two major differences between this and the code that one must bear in mind. The first is that instead of k having the values of 10, 15, 20, 25 … it instead has the values of 0, 1, 2, 3 … and we use the expression $10+5k$ to create the correct values. This expression will be referred to as the *counter expression* The second difference is that in the code, k is non-inclusive of the upper limit. That is to say that the largest value k will have is 95. Whereas, in the expression, k is inclusive of the upper limit and when k has its maximum value of 18, the counter expression will equal the actual upper limit of 100. We can generalise for any lower bound, upper bound and jump to create the following function:

$$forSum_{sub}(n,n,n) \to n$$
$$forSum_{sub}(l,u,j) = \sum_{k=0}^{\lfloor\frac{u-l}{j}\rfloor} sub(l+jk)$$

Here, l is the lower bound, u is the upper bound and j is the jump. Note that we added a floor function to the top of the sigma, so that if the difference between the upper and the lower bounds does not divide evenly by the jump, the value of the counter expression will get as close to the upper bound as possible without going over. In other words, the value at the top of the sigma is equal to the number of jumps which need to be performed to get from the lower bound to the upper bound, if that is not an integer (for example if 2.4 jumps needed to occur), we will instead perform the next integer down of jumps (in this case 2). You will also see that there is some subtext beneath the function name. This represents the name of the function over which we are iterating. This is necessary because it will change depending on the purpose of the loop, and function names cannot be passed as parameters. When using this function, we will replace *sub* with the name of the function over which we wish to iterate. For example, if we wanted the sum of the first x square numbers, we might go about it as follows:

$$square(n) \to n$$
$$square(x) = x^2$$

$$sumOfSquares(n) \to n$$
$$\begin{aligned} sumOfSquares(x) &= forSum_{square}(1,n,1) \\ &= \sum_{k=0}^{\lfloor\frac{n-1}{1}\rfloor} square(1+1k) \\ &= \sum_{k=0}^{n-1} square(k+1) \\ &= \sum_{k=0}^{n-1} (k+1)^2 \end{aligned}$$

We can also find the product, looping over a function using a pi symbol. It would be equivalent to the following Python code:

```
total = 1
for k in range(10,100,5):
        total *= sub(k)
```

And the function itself would be defined almost identically to before, as

$$forProduct_{sub}(n, n, n) \to n$$

$$forProduct_{sub}(l, u, j) = \prod_{k=0}^{\lfloor \frac{u-l}{j} \rfloor} sub(l + jk)$$

It is also worth noting that sub, or indeed whichever function over which you are iterating, can only take one parameter. Naturally, this is of little consequence as functions can be substituted, but in the interests of convention and clarity it is worth baring in mind.

8.2 While Loops

While loops are slightly more problematic as they do not contain a finite set of numbers over which to loop. As such, we can not use sum or product notation, and instead we will use recursive functions. Also, not only do we need a function over which to iterate, but we also need a function which (depending on its parameters) will return a boolean value indicating whether or not to continue the loop. The stub of the *whileSum* function will look like this:

$$whileSum_{sub_{cond}}(n, n) \to n$$

In which sub is the function over which we are iterating, $cond$ is the function which tells us whether to continue the loop, and the two parameters are the counter variable (which will be called x) and jump (which will be called j)respectively. We need our function, if $cond(x)$ returns a 1, to return $sub(x)$ plus the value which the function returns with x increased by j. Otherwise, to return 0 and terminate the recursion.

$$\begin{aligned}
&whileSum_{sub_{cond}}(n, n) \to n \\
&whileSum_{sub_{cond}}(x, j) = if(cond(x), sub(x) + whileSum_{sub_{cond}}(x + j, j), 0) \\
&\qquad = cond(x)(sub(x) + whileSum_{sub_{cond}}(x + j, j) - 0) + 0 \\
&\qquad = cond(x)(sub(x) + whileSum_{sub_{cond}}(x + j, j))
\end{aligned}$$

An example use of this would be a function which finds the sum of the squares of every third number starting from x until it lands on a multiple of 10. We will call this function $sumSqThirdUntilDividesTen$. We can define it as follows:

$$\begin{aligned}
&doesntDivideTen(n) \to b\\
&doesntDivideTen(x) = not(divides(x, 10))\\
&\qquad\qquad\qquad\quad = 1 - (1 - \lceil \frac{x \bmod 10}{1 + (x \bmod 10)} \rceil)\\
&\qquad\qquad\qquad\quad = \lceil \frac{x \bmod 10}{1 + (x \bmod 10)} \rceil
\end{aligned}$$

$$\begin{aligned}
&sumSqThirdUntilDividesTen(n) \to n\\
&sumSqThirdUntilDividesTen(x) = whileSum_{square_{doesntDivideTen}}(x, 3)\\
&\qquad = doesntDivideTen(x)(square(x) + whileSum_{square_{doesntDivideTen}}(x + 3, 3))\\
&\qquad = doesntDivideTen(x)(square(x) + sumSqThirdUntilDividesTen(x + 3\\
&\qquad = \lceil \frac{x \bmod 10}{1 + (x \bmod 10)} \rceil (sumSqThirdUntilDividesTen(x + 3))
\end{aligned}$$

Similar to *whileSum*, we can define a function *whileProduct*, which will multiply the terms rather than add them.

$$\begin{aligned}
&whileProduct_{sub_{cond}}(n, n) \to n\\
&whileProduct_{sub_{cond}}(x, j) = if(cond(x), sub(x) \cdot whileSum_{sub_{cond}}(x + j, j), 1)\\
&\qquad\qquad\qquad\qquad\quad = cond(x)(sub(x) \cdot whileSum_{sub_{cond}}(x + j, j) - 1) + 1
\end{aligned}$$

The main problem with representing *while* loops as mathematical functions is that they are, by their very nature, recursive. Although this is not catastrophic, it does mean that an algorithm using a *while* loop may need multiple functions, as substitution may not always be possible. Also, all of the other functions defined in this paper do not need to inherently take the form of a function, but could instead be part of an equation, a formula or just an expression. However, now that recursion is introduced, a function is necessary and the solution becomes less versatile. As such, I would recommend if at all possible to rephrase any algorithm in terms of for loops as opposed to *while* loops.

9 Example - Primality Test

We can create a function which checks whether a number is prime or composite, returning 1 if it is prime and 0 otherwise. It is worth noting that this is definitely not the fastest primality tester to compute, nor is it the most elegant. However, it does a good job of demonstrating the methods used in this paper.

A number x is prime if and only there are no numbers less than x which, when x is divided by them, result in a remainder of 0. We can take the product of the remainders of all such divisions, and if any of them are indeed 0, then the result will be 0. Otherwise the number is prime or 1. Therefore, we need also to check whether x is greater than 1. The function is defined as follows:

$$\begin{aligned}
&rem_x(n) \to n\\
&rem_x(y) = x \bmod y
\end{aligned}$$

$isPrime(n) \to b$
$$\begin{aligned}
isPrime(x) &= and(greaterThan(x,1), doesntEqualZero(forProduct_{rem_x}(2, x-1, 1)))\\
&= and(\lceil \frac{|1-x-|1-x||}{1+|1-x-|1-x||} \rceil, doesntEqualZero(\prod_{k=0}^{x-3} rem_x(2+k)))\\
&= and(\lceil \frac{|1-x-|1-x||}{1+|1-x-|1-x||} \rceil, doesntEqualZero(\prod_{k=2}^{x-1} (x \bmod k)))\\
&= and(\lceil \frac{|1-x-|1-x||}{1+|1-x-|1-x||} \rceil, \lceil \frac{\prod_{k=2}^{x-1}(x \bmod k)}{1+\prod_{k=2}^{x-1}(x \bmod k)} \rceil)\\
&= \lceil \frac{|1-x-|1-x||}{1+|1-x-|1-x||} \rceil \lceil \frac{\prod_{k=2}^{x-1}(x \bmod k)}{1+\prod_{k=2}^{x-1}(x \bmod k)} \rceil
\end{aligned}$$

10 Conclusion

In this paper we have set out a framework for turning algorithms blocks of code which utilise conditional statements, boolean logic and loops into analytical standalone mathematical functions, making them easier to process, graph and transport, although they may have a slightly clunkier appearance.

11 List Of Functions

Below is the list of all functions defined in this paper in order of appearance.

$triple(n) \to n$
$triple(x) = 3x$

$isOdd(n) \to b$
$isOdd(x) = x \bmod 2$

$if(b, n, n) \to n$
$if(x, a, b) = x(a-b) + b$

$collatz(n) \to n$
$collatz(x) = (x \bmod 2)(\frac{5x}{2} + 1) + \frac{x}{2}$

$doesntEqualZero(n) \to b$
$doesntEqualZero(x) = \lceil \frac{|x|}{1+|x|} \rceil$

$not(b) \to b$
$not(x) = 1 - x$

$equalsZero(n) \to b$

$equalsZero(x) = 1 - \lceil \frac{|x|}{1+|x|} \rceil$

$divides(n, n) \rightarrow b$
$divides(a, b) = 1 - \lceil \frac{a \bmod b}{1+(a \bmod b)} \rceil$

$equals(n, n) \rightarrow b$
$equals(a, b) = 1 - \lceil \frac{|a-b|}{1+|a-b|} \rceil$

$lessThan(n, n) \rightarrow b$
$lessThan(a, b) = \lceil \frac{|a-b-|a-b||}{1+|a-b-|a-b||} \rceil$

$greaterThan(n, n) \rightarrow b$
$greaterThan(a, b) = \lceil \frac{|b-a-|b-a||}{1+|b-a-|b-a||} \rceil$

$lessThanOrEqualTo(n, n) \rightarrow b$
$lessThanOrEqualTo(a, b) = 1 - \lceil \frac{|b-a-|b-a||}{1+|b-a-|b-a||} \rceil$

$greaterThanOrEqualTo(n, n) \rightarrow b$
$greaterThanOrEqualTo(a, b) = 1 - \lceil \frac{|a-b-|a-b||}{1+|a-b-|a-b||} \rceil$

$and(b, b) \rightarrow b$
$and(x, y) = xy$

$or(b, b) \rightarrow b$
$or(x, y) = \lceil \frac{x+y}{1+x+y} \rceil$

$xor(b, b) \rightarrow b$
$xor(x, y) = \lceil \frac{|x-y|}{1+|x-y|} \rceil$

$forSum_{sub}(n, n, n) \rightarrow n$

$$forSum_{sub}(l, u, j) = \sum_{k=0}^{\lfloor \frac{u-l}{j} \rfloor} sub(l + jk)$$

$square(n) \rightarrow n$
$square(x) = x^2$

$$sumOfSquares(n) \to n$$
$$sumOfSquares(x) = \sum_{k=0}^{n-1}(k+1)^2$$

$$forProduct_{sub}(n,n,n) \to n$$
$$forProduct_{sub}(l,u,j) = \prod_{k=0}^{\lfloor \frac{u-l}{j} \rfloor} sub(l+jk)$$

$$whileSum_{sub_{cond}}(n,n) \to n$$
$$whileSum_{sub_{cond}}(x,j) = cond(x)(sub(x) + whileSum_{sub_{cond}}(x+j,j))$$

$$doesntDivideTen(n) \to b$$
$$doesntDivideTen(x) = \lceil \tfrac{x \bmod 10}{1+(x \bmod 10)} \rceil$$

$$sumSqThirdUntilDividesTen(n) \to b$$
$$sumSqThirdUntilDividesTen(x) = \lceil \tfrac{x \bmod 10}{1+(x \bmod 10)} \rceil (sumSqThirdUntilDividesTen(x+3))$$

$$whileProduct_{sub_{cond}}(n,n) \to n$$
$$whileProduct_{sub_{cond}}(x,j) = cond(x)(sub(x) \cdot whileSum_{sub_{cond}}(x+j,j) - 1) + 1$$

$$rem_x(n) \to n$$
$$rem_x(y) = x \bmod y$$

$$isPrime(n) \to b$$
$$isPrime(x) = \lceil \tfrac{|1-x-|1-x||}{1+|1-x-|1-x||} \rceil \lceil \tfrac{\prod_{k=2}^{x-1}(x \bmod k)}{1+\prod_{k=2}^{x-1}(x \bmod k)} \rceil$$