

# On the Unification of Mathematical Notation and Programming Notation

Rhys Goldstein

August 7, 2008

## Abstract

A carefully selected set of mathematical notations, intended for both human interpretation and computer execution, could render irrelevant the distinction between mathematical notation and programming notation. The argument is made that it would be both feasible and advantageous to program computers with mathematical formulas embedded in documents, as an alternative to code.

## Introduction

It seems to be human nature to perceive a long-standing convention, however arbitrary it was when first adopted, as being the sole correct way of doing something. We tend to follow such a convention without questioning its merits, without considering its alternatives.

An interest in programming language design led me to explore traditional mathematical notation, and for the first time I became aware of its many weaknesses. Even the most frequently obeyed conventions have flaws. Consider, for example, the following two expressions.

$$a(b + c)$$

$$f(x + y)$$

Most would interpret the first as the product of a number  $a$  and the sum of  $b$  and  $c$ . And the second expression appears to be the result of a function  $f$  when applied to the sum of  $x$  and  $y$ . But note the ambiguity. One could justifiably interpret  $a$  as a function and  $f$  as a number. Convention has us using the same notation for multiplication and function application. In the example above we are spared from confusion by naming conventions; the name  $f$ , for instance, is generally used to represent a function. Nevertheless, human interpretation of mathematical formulas would be more reliable with the adoption of an alternative convention, such as the use of a dot for multiplication.

$$a \cdot (b + c)$$

That there are unnecessary flaws in traditional mathematical notation is a fact so obvious to me now, it seems absurd that I had once thought otherwise. Perhaps it is the age and universal adoption of mathematical conventions that mislead me. I imagined that the notations I was taught all my life came about in a series of profound discoveries, when in fact they were the result of arbitrary decisions. In any case, the realization that mathematical notation could be improved led me to a promising idea about computer programming.

First note that there is, at present, a very pronounced distinction between mathematical notation and programming notation. While modern programming languages adopt some mathematical conventions, they are generally restricted to lines of monospace text, include dozens of reserved words, and introduce features such as sequences of statements, loops, and the ability to re-assign variables.

Programming language features seem to be necessary, as we all know traditional mathematical notation is not well-suited to computer execution. But few realize that traditional mathematical notation is not particularly well-suited to human interpretation either. And this observation leads to the idea that, if one decides to adopt unambiguous mathematical notations to improve human interpretation, the opportunity arises to make those notations sufficient for computer execution as well. Notations specific to programming would then lose their necessity, and in their absence the distinction between mathematical notation and programming notation would become irrelevant.

I have identified a set of “unified” mathematical notations suitable for both human interpretation and computer execution. Their detailed description is deferred to another document, but in this paper I will use these notations in examples to make a case for the unification of mathematical notation and programming notation. Focusing first on the presentation of algorithms in academia, and then on the design of interactive software, I will argue that it would be both feasible and advantageous to write programs with mathematical formulas embedded in documents. For all but the lowest-level applications, computer code as we know it might then become obsolete.

## The Presentation of Algorithms in Academia

Academic researchers almost invariably describe their algorithms two times over. One description is formal, written in a programming language, and intended primarily for computer execution. Below is an example: an implementation of Newton’s method for approximating the root of a function  $f$  given its derivative  $g$ .

```
def Newton(f, g, x0, eps, N):
    x = x0
    i = 0
    while i < N:
        x_new = x - f(x)/g(x)
        if abs(x_new - x) < eps:
            return x_new
        x = x_new
        i = i + 1
    return None
```

If researchers were satisfied with their code as a means of presenting their algorithms in lectures and publications, they would need not prepare an alternative description. And admittedly, the odd researcher does publish some of his or her code. But typically, researchers seem to be of the opinion that regardless of which popular programming language they use, and regardless of their programming style, no amount of colouring or commenting can

make their monospace ASCII text presentable. Hence there is a need for that other description, which is almost invariably informal, composed of a mixture of prose and mathematical expressions, and intended only for human eyes. Below is Newton's method again, as it might appear on paper.

Let  $f$  be a continuous function, let  $g$  be its derivative, and let  $x_0$  be an initial guess at a root of  $f$ . Starting with  $x = x_0$ , an improved guess  $x'$  is calculated as follows.

$$x' = x - \frac{f(x)}{g(x)}$$

This improved guess then becomes the current guess, and the calculation is repeated. The process ends successfully when two consecutive guesses differ by less than some arbitrary threshold  $\epsilon$ .

$$|x' - x| < \epsilon$$

The process fails if the guesses do not converge after an excessive number of iterations.

There is something problematic with this convention of writing formal code, and separate informal documentation. And the problem goes beyond the time wasted producing multiple descriptions of the same thing. The real consequence is the failure of publications to fully specify the algorithms they describe. Formulas like those above omit details, and the surrounding text is ambiguous by nature. For a simple algorithm like Newton's method, this may not seem like a serious issue. But for more complex algorithms, omissions and ambiguities severely detract from a researcher's contribution. His or her work cannot be reproduced by others. Errors in the researcher's programs are concealed, and therefore go undiscovered. In many publications one may learn that a new algorithm was invented, and read that its implementation produced excellent results, but will never be provided with the formal specification necessary to use or evaluate the algorithm oneself.

It is neither fundamentally correct nor fundamentally incorrect to make a distinction between mathematical notation and programming notation. The distinction comes with both advantages and disadvantages. But we must recognize it as a disadvantage that the distinc-

tion between notations compels a researcher to produce two complementary descriptions of an algorithm instead of one description suitable for both communication and computation. Should we choose to unify our notations, executable programs fit for publication become feasible.

What follows is a combined description and implementation of Newton's method, as it might look if mathematical notation and programming notation were the same thing. We still see mathematical formulas surrounded by text, but the formulas are written using the unified notations I referred to in the introduction.

The function *Newton* attempts to approximate a root of its argument function  $f$  using  $g$ , the derivative of  $f$ . Starting with an initial guess  $x_0$ , a new guess is calculated on each iteration. If two consecutive guesses differ by less than the threshold  $\epsilon$ , the latter guess is the final approximation and the result of *Newton*. If  $N$  iterations pass before a final approximation is found, however, the procedure is terminated and *Newton* results in  $\emptyset$ .

$$Newton([f, g, x_0, \epsilon, N]) := loop([x_0, 0])$$

The recursive function *loop* takes as arguments first the current guess  $x$  then number of completed iterations  $i$ . Note that *Newton* itself is defined as the result of *loop* given the initial guess  $x_0$  obtained after 0 iterations. As indicated by the conditional expression below, *loop* results in  $\emptyset$  if  $N$  iterations have been completed.

« *Newton* »

$$loop([x, i]) := \begin{pmatrix} i = N & \rightarrow \emptyset \\ i < N & \rightarrow seek \end{pmatrix}$$

If fewer than  $N$  iterations have been completed, the process continues with the evaluation of *seek*. The result depends on whether the new guess  $x'$  differs from the current guess  $x$  by less than  $\epsilon$ . If it does, then  $x'$  is the final approximation. Otherwise *loop* is applied recursively to the

new guess  $x'$ , with  $i + 1$  iterations having passed.

$\ll \dots; loop \gg$

$$seek := \left( \begin{array}{l} |x' - x| < \epsilon \rightarrow x' \\ |x' - x| \geq \epsilon \rightarrow loop([x', i + 1]) \end{array} \right)$$

The new guess  $x'$  is calculated as follows.

$\ll \dots \gg$

$$x' := x - \frac{f(x)}{g(x)}$$

The “context identifiers” ( $\ll \dots; loop \gg$ , for example) are unconventional. I adopt them to control the scope of each variable. When formally describing algorithms, it is not practical to adhere to traditional mathematical notation and define all variables with a global scope. I have also altered the notation for conditional expressions: conditions on the left point to their respective results. One may notice a few other unusual features, but for the most part the notations used above are similar to those of traditional mathematics and need no explanation.

Compare this description of Newton’s method with the first version written in code. The four mathematical formulas are, in my opinion, more presentable than the nested structures and statements introduced with the reserved words `def`, `while`, `if`, and `return`. But the ability to execute the program has not been sacrificed. As the application of *Newton* requires no more than straight-forward substitution and reduction operations, the embedded formulas above could be automatically extracted from the document and compiled. The resulting software could be used to evaluate an expression like the one below.

$$Newton([sin, cos, 3, 0.0001, 100])$$

Unlike the informal publication-style description of Newton’s method shown earlier, this new version omits no details. The original statement that the “improved guess then becomes the current guess, and the calculation is repeated”, is now formally specified using the new set of mathematical notations. We now know with certainty, for example, that the condition  $g(x) = 0$  has not been handled. The publication of details like this will allow

researchers to find potential bugs in each others' programs, improving reliability.

A unification of mathematical notation and programming notation would render coded mathematical expressions obsolete. When modifying a program, we would no longer have to parse, in our heads, unappealing text like the following.

```
1/(sqrt(2*pi)*sigma)*exp(-(x - mu)**2/(2*sigma**2))
```

Instead we would read rendered expressions like the one below. (Note the dots used for multiplication.)

$$\frac{1}{\sqrt{2 \cdot \pi} \cdot \sigma} \cdot e^{-\frac{(x-\mu)^2}{2 \cdot \sigma^2}}$$

A unification of notations would undermine the distinction between software design and implementation. For a researcher, I see this as a significant advantage. The design and implementation of an algorithm ought to be a single activity, striving to produce a document that serves as both a program and a publication. This new style of programming would promote the communication of formal ideas both within academia, and between academia and industry.

## The Design of Interactive Software

Admittedly, a skeptical reader might not yet be convinced that mathematical formulas are a viable alternative to computer code. They sufficed for Newton's method, as shown, but the execution of that algorithm was strictly computational. When a program is interactive, it must output information to a human or some computer process, and respond accordingly when information is input. Traditional mathematical notation was not designed to represent such events, or "side effects" as they are called. The following examples will demonstrate, however, that by introducing a few additional rules governing computer execution, mathematical formulas can in fact be used to design interactive software.

Consider a function, named *login*, that requests from the user first a username then a password. Upon receiving these values, the function attempts to gain access to some computer system. If access is obtained, *login* results in a truthful value. Otherwise the message "login failed" is output, and the login procedure is repeated from the beginning. The function

*login* terminates with a false result if, during the input of the username or password, the user chooses to abort the procedure.

Below is an implementation of *login* in typical programming notation. We introduce as primitives (from some hypothetical library, say) the functions *output*, *input*, and *access*. The *output* function displays its argument and has no result. The *input* function also displays its argument, but then waits for an input value from the user to adopt as its result. The user may decline to input a value, in which case *input* results in *None*. Using its *username* and *password* arguments, *access* attempts to gain access to some computer system. It results in a boolean to indicate whether it was successful.

```
def login():
    while true:
        username = input("Username: ")
        if username == None:
            return false
        password = input("Password: ")
        if password == None:
            return false
        accessed = access(username, password)
        if accessed:
            return true
        output("login failed")
```

This and other examples were written in Python. Hopefully experts in the language will note that while *input* conflicts the “built-in” function of the same name, the above still serves its purpose as an example of typical code. In terms of aesthetics, Python features one of the harder sets of programming notation to argue against. But it is not my intent to promote mathematical formulas by offering unnecessarily obscure code as the alternative. Instead, I will try to argue that there is a strong case for programming with formulas even when compared with reasonable coding styles and conventions. Below is a combined description/implementation of *login*, defined using my unified notations. We again adopt the primitives *output*, *input*, and *access*. The *output* function now results in  $\emptyset$ , as does *input* in the event that the user declines its request.

The recursive *login* function first requests a username from the user, resulting in  $\perp$  if the user declines.

$$login() := \left( \begin{array}{l} username \equiv \emptyset \rightarrow \perp \\ username \not\equiv \emptyset \rightarrow verify \end{array} \right)$$

$$username := input("Username : ")$$

Having received a username, a password is requested. A decline, as before, results in  $\perp$ .

$\ll login \gg$

$$verify := \left( \begin{array}{l} password \equiv \emptyset \rightarrow \perp \\ password \not\equiv \emptyset \rightarrow check \end{array} \right)$$

$$password := input("Password : ")$$

With both a username and a password, access to the computer system is requested. If accessed is obtained, *login* results in  $\top$ .

$\ll \dots; verify \gg$

$$check := \left( \begin{array}{l} accessed \rightarrow \top \\ -accessed \rightarrow retry \end{array} \right)$$

$$accessed := access([username, password])$$

If access is not obtained, a message is output and *login* is executed recursively to repeat the procedure.

$\ll \dots; check \gg$

$$retry := \left[ \begin{array}{l} output("login failed") \\ login() \end{array} \right] (1)$$

What is particularly nice, in this case, is the ease with which a procedure can be broken down and represented with a set of very small formulas. I am of the opinion that because mathematical notation was invented for only human interpretation, it has maintained the

property that any expression can be extracted from the larger expression that contains it. Computer code has not maintained an analogous property with its blocks of code, as the reader will realize after an attempt to partition the Python version of *login*.

It would not be easy for any modern-day programmer to give up reserved words, loops, variables that change value, jump statements like `return`, and other programming notations that have for so long dominated the discipline. And yet the use of mathematical formulas is a sound alternative. As demonstrated by a final example, even a graphical user interface (GUI) can be programmed in this manner. Suppose the *login* function is now invoked by the click of a “Login” button. The username and password are no longer requested in sequence, but taken instead from text boxes that the user is free to edit. We keep the primitives *output* and *access*, but in place of *input* adopt GUI component functions named *panel*, *label*, *text\_box*, and *button*.

The function  $panel_{login}$  results in a login panel to be used to gain access to a computer system.

$$panel_{login} () := panel ([\text{“components”} \rightarrow components])$$

The components of the login panel include an editable text box for the username, an editable text box for the password, and a login button. Both text boxes have associated labels.

$\ll panel_{login} \gg$

$$components := \left[ \begin{array}{l} label(\text{“Username :”}) \\ text\_box_{username} \\ label(\text{“Password :”}) \\ text\_box_{password} \\ button([\text{“Login”}, login]) \end{array} \right]$$

The username text box is editable. The password text box is also editable, but the text is concealed.

« ... ; components »

$$text\_box_{username} := text\_box ([\textit{“editable”} \rightarrow \top])$$
$$text\_box_{password} := text\_box \left( \left[ \begin{array}{l} \textit{“editable”} \rightarrow \top \\ \textit{“conceal”} \rightarrow \top \end{array} \right] \right)$$

The *login* function, invoked by the login button, attempts to access the computer system using the values entered into the username and password text boxes. If unsuccessful, a message is output.

« ... »

$$login () := \left( \begin{array}{ll} accessed & \rightarrow \emptyset \\ -accessed & \rightarrow output(\textit{“login failed”}) \end{array} \right)$$
$$accessed := access \left( \left[ \begin{array}{l} text\_box_{username}(\textit{“text”}) \\ text\_box_{password}(\textit{“text”}) \end{array} \right] \right)$$

Modern programmers often focus on the production of reusable libraries of code. As these libraries may be incorporated into a vast number of different projects, they tend to become large and complex. The adoption of multiple large libraries in a single project often leads to compatibility problems that are difficult to resolve. I believe that by replacing code with executable mathematical formulas embedded in documents, programmers would concentrate more on the communication of their ideas and less on the production of functionality. A set of formulas could be reused in the same manner as code, of course. But instead of including in their project a library of 50,000 lines of code they never read, programmers might choose to adopt only those formulas that address their needs.

Another problem with modern software development is the fact that when a project is discontinued, its code is rarely salvageable. In particular, if the programming language used in that project falls out of fashion, the code will almost certainly be neglected. But a document that completely and formally describes a program, using familiar mathematical notations, may remain useful long after the abandonment of the software it defines.

## Conclusion

In all honesty, I feel that the case for the unification of notations is strongest in the context of academic research. In this field, at present, computer code is essentially re-written with mathematical formulas anyway, though only for the purpose of publication. The mathematical formulas that describe a program might as well be the program itself, in which case researchers could dispense with code. Though not quite as obvious, the argument for programming with mathematical formulas is still compelling for the design of interactive software. With tools for rendering, editing, and executing formulas, programmers would have the means to write programs for others to appreciate. Given these advantages, one might ask why mathematical notation and programming notation are not already the same thing. The explanation is that, while we have been willing to invent numerous programming languages with different notations, we rarely depart from established conventions when it comes to mathematics. Our habits lead us to regard mathematical notation as something that cannot be changed, and under this self-imposed constraint the unification of notations is impossible.

## Credit

I first realized that traditional mathematical notation could be improved after reading Edsger Dijkstra's manuscript "The notational conventions I adopted, and why" [1]. It was at this point, while designing a new programming language, that I gave up the text editor in favour of pencil and paper. Equally influential was Eric Hehner's "from Boolean Algebra to Unified Algebra" [2]. Hehner shows how one can dispense with traditional symbols such as  $\neg$ ,  $\Leftarrow$ ,  $\Rightarrow$ ,  $\oplus$ ,  $\exists$ , and  $\forall$ , without sacrificing boolean algebra in the process. Such dramatic simplifications would make mathematical notation considerably easier for children to learn and for professionals to adopt as a programming tool.

Donald Knuth can be credited for the idea that programs should exist in documents intended for both human interpretation and computer execution [3]. To make "literate programming" feasible, Knuth devised a way to partition computer code so that each part of a program could be described by its surrounding text. Mathematical formulas are much easier to partition than code, however, and have been embedded in documents before computers existed. The style of programming I promote could be called "literate programming with mathematical formulas".

But of course the use of rendered mathematical formulas for programming is not my idea either. The Fortress programming language represents perhaps the most notable recent effort to make programming look like mathematics [4]. In all but appearance, however, Fortress is very much like other modern programming languages. Employing over 70 reserved words, it includes sequences of statements, loops, the ability to re-assign a variable, the ability to jump out of a control structure, and object-oriented features.

I do not have a strong objection to loops, incidentally. But loops are of little use without the ability to assign a new value to an existing variable. And when a statement that re-assigns a variable is extracted into a function, one is compelled to pass the variable "by reference". Pandora's Box is now open, for throughout our programs we must face the possibility that different variables have shared identities. Basic mathematical transformation become invalid; as a result, even when faced with strictly computational algorithms, we

tend to abandon mathematical reasoning and rely on tests. The alternative to loops is to use recursive functions, thereby avoiding the complications associated with references. The feasibility of this approach is demonstrated most effectively by “functional programming” languages. It is this style of programming, and no invention of my own, that allowed the mathematical formulas in my examples to fully describe algorithms and interactive procedures.

The new programming style I propose would require the integration of these preexisting and previously-demonstrated ideas: Fortress-like rendering to capture the appearance of mathematical notation, functional programming to capture its semantics, and literate programming to enhance its presentation. But first we need to select a set of unambiguous mathematical notations, and in that effort we should turn to the work of Dijkstra and Hehner.

## References

- [1] E. W. Dijkstra. The notational conventions I adopted, and why (EWD 1300). 2000.
- [2] E. C. R. Hehner. from Boolean Algebra to Unified Algebra. *the Mathematical Intelligencer*, 26(2):3–19, 2004.
- [3] Donald E. Knuth. Literate Programming. *The Computer Journal*, 27(2):97–111, 1984.
- [4] Eric Allen, David Chase, Joe Hallet, Victor Luchangco, Jan-Willem Maessen, Sukyoung Ryu, Guy L. Steele Jr., and Sam Tobin-Hochstadt. The Fortress Language Specification Version 1.0. 2008.