From Lego, Java and Mathematica *to* feature interaction

Harold Thimbleby

Gresham Professor of Geometry

Notes for a lecture given at Gresham College on 13 March 2003

We're all familiar with construction kits like Lego. A pile of Lego bricks can take you to almost anywhere your imagination wants to go. You can build houses or cars, spaceships or robots. You can either start sticking bricks together and see what happens, or you could sit down and work out what bits you need and how many. It is possible to work out how many bricks you will need for a design before you start, but hardly anybody does — it's far easier to push things together and see what you can build. Often as you build things, you wonder why Lego isn't made with a brick shaped *like this,* so it would fit *just here.* Perhaps if you had planned your building, you wouldn't have got to this impasse — but it's not really a problem as you can easily dismantle part of the model and have another go. Eventually you'll get something to work that is good enough, and you'll have had lots of fun. And if the model isn't good enough you can patch it up with a few reinforcements.

What do we learn from this?

- It's easier to make than design
- It's much easier to test than plan
- It's easier to rebuild than build right first time
- Sometimes the bits don't fit
- Sometimes you wonder why special bits that you need aren't made
- It's great fun being creative
- You wouldn't make a real building out of a toy.

All these points are obvious and familiar from our childhood. The point is: this familiar experience is just like programming, the topic of this Gresham lecture. Programs are built out of building blocks, which fit together into many ways — much like Lego bricks.

Like building Lego models, computer programs are very easy to write, at least if you don't pre-plan what you are going to do. You can work out what programs are going to do, but it is usually so difficult that it is easier to build them, then try them out, and then tinker with them to get them to work as you want. Sometimes the bricks fit nicely, and sometimes we will wonder why they are made the shapes they are.

Don't panic!

Some of the discussion in these notes gets quite technical. Please jump ahead to the penultimate section 'Feature interaction' (starting on the last page) if you want to skip any of the messy details. However, the difficulty of the discussion is something that needs to be seen to be believed. That programming is a very complex activity is one of the points we are making — and the details are there if you want to take issue with my conclusions!

Mindstorms

Lego is an amazingly versatile toy. It comes in various flavours, suitable for people of all ages. There are even Lego professionals, earning a living out of this stuff. Lego Mindstorms is a computerised version of Lego, and the product line we shall build our lecture on. It has been said that Mindstorms is a breakthrough in technological toys. With its building elements you can create contraptions that would make Leonardo da Vinci jealous (Erwin, 2001). Your mechanisms can have sensors, motors and lights operated by a programmable brick, the Lego Robotic Command Explorer (RCX). Things built with the RCX can run on their own, explore the environment, interact with each other or be dynamic works of art. The RCX has been used with young children through to postgraduate university students. For our purposes, Mindstorms will bring our programming is playing with bricks analogy to life.

Although the RCX can be programmed in several languages (including Java) we've chosen NQC (Baum, 2003) because it is easily available and it is a textual language (rather than a pictorial language) so it is easily compared with the other two languages we will be looking at. NQC means Not Quite C.

Computer programs can be put together much like Lego bricks. I've programmed the RCX with some demonstrations of how easy it is to put 'bricks' together in programs.

The first program simply makes a light flash. (All robots *must* have flashing lights.)

The 'brick' to make a light flash is called a ***task***, and it can be programmed in NQC as follows:

```
task flash()
{     On(lamp); Wait(300); Off(lamp); Wait(300);
}
```

… except that this only makes the lamp come on for 3 seconds (i.e., for 300 hundredths of a second) then switch off. To get the lamp to flash, rather than come on and go off once, this sequence of instructions must be repeated endlessly:

```
task flash()
{     while( true )
      {     On(lamp); Wait(300); Off(lamp); Wait(300);
      }
}
```

This 'task' is a bit like a 'brick.' When we build a robot, it will probably have other tasks to do (such as seeking out light, moving) and we can add each task easily. The Lego RCX is designed so that tasks work nicely like this: they run independently, and a programmer can introduce more tasks like these to do new things in the robot.

However, like you can run out of Lego bricks, you can run out of tasks! The RCX can only do eight tasks at once, so if you have a complicated robot project, you have to design it to fit with the few tasks you have available in your programming kit — or you have to buy a completely different kit, or use two RCX computers together. Or something.

Tasks are a rather complex sort of progamming brick, although a sort of brick that fits together with other bricks quite cleanly. For the rest of this lecture, we'll look at ***assignment***, which is a much simpler sort of brick. (Programmers will know that tasks and assignments fit together in very interesting ways, which unfortunately we won't have time to discuss.)

Variables

Most programming languages have variables, which behave a bit like mathematical variables. When we write `x` in a program, this generally means "the value of `x`" — just like when we write x in a mathematical equation we mean to refer to the value of x. So, x+1 means one more than the value of x.

In mathematics, everywhere we say x we mean the same thing, namely the same x. So $x=2x-1$ is an equation, meaning x is twice itself minus one. Because x is the same thing everywhere in this equation, we can subtract it from both sides and leave the equation still an equation. So $x-x=2x-1-x$ means the same thing as the original equation. A little more jiggling and we get $0=x-1$, then we can conclude $x=1$. This final equation means the same as the equation we started with. Indeed, if x is one, then one equals twice one (i.e., two) minus one: in other words, one equals one, or more pedantically $x=1$ is equivalent to $x=2x-1$, and it just happens that they are both true.

To summarise, in mathematics we have rules for reasoning that, in particular, allow us to manipulate equations to get simpler forms that are equivalent. One of the main rules we use is that names, such as x, mean the same thing everywhere they occur. If the meaning of a name like x changed gratuitously, meaning different things in different places, it would not be possible to reason reliably about equations.

Programming is rather different. Names do not mean the same thing everywhere, and this is both a source of strength and a source of weakness. It is a source of strength because we want computer programs, particularly ones inside robots, to change the world: things have to change. Even in maths, the $x=2$ I am now writing about is not the same $x=1$ I wrote about earlier in these lecture notes. It's obviously useful to change you mind about what you are talking about — it would be very restrictive if every x in these notes had to mean exactly the same thing everywhere! On the other hand, being able to change our mind is a source of weakness because it becomes very much harder to reason precisely about what we mean, let alone what programs should do. It is a very real weakness, as the evidence from all of the failing and buggy programs we know about only too well testifies. Even expert programmers find programming difficult.

We can solve the mathematical conundrum of the last paragraph, where we wanted x to mean different things in different places by introducing a new concept, ***scope***. We now say every name means the same thing in the same scope. The first use of x was in the same scope as $x=2x-1$, and the second use of x was in a different scope, where we wanted $x=2$. In each of these two scopes, x means one thing, but the two scopes are quite different uses of x. That's not too hard to understand. 'Scope' is a formal name for a concept we've all been using implicitly since school days. When doing some schoolbook exercises we expect the answer x of Question 2 to be a different x than the answer to Question 3. Each exercise, then, is in a different scope. In fact, the scopes are rather complex: if the book provides answers in the back, then an answer $x=2$ (or whatever) for Question 3 is to be understood to be in the same scope of x as in Question 3 itself. So scopes may be spread out in complex ways, but they should never overlap — we should always be clear which x we are talking about.

Once we have the notion of scope clear, we can be clear that when we write x it means a particular x, provided we are clear about the scope being used. And of course (though we shall see some counter-examples later), when we define a new scope, we are defining a new use of a name which is not to be confused with other uses of the same name.

Now in a programming language we can write what looks like an equation, such as `x=2x-1` but this is an assignment, and it means something very different. Here `x` is not the same thing in each place it occurs. As an assignment, it means *make* `x` twice what it *was* and subtract one. So if `x` was 1, now `x` would still be 1 (coincidentally); but if `x` was 2, now `x` would be 3. However if we read `x=2x-1` as a mathematical equation, `x` is never 2 nor 3. If we read `x=2x-1` as an assignment, `x` can be any number, and after doing the assignment, `x` will be one less than it was doubled. In short, assignment is an instruction (for a computer) to do something, whereas an equation (for a mathematician) is a statement of fact. Yet we still want a mathematical understanding for the assignment (so we can think clearly about what it means). Somehow the two uses of `x` in the assignment are same sort of thing, but they are being used differently. The two uses of `x` are in the same scope, but they somehow have different meanings.

In normal mathematics, every variable like x has a value. The solution to the assignment problem is to introduce two sorts of value: left values and right values. On the left hand side, `x` means the object `x` itself. On the right hand side, `x` means what the value of the object `x` is. What does this mean? It means the mathematical value of `x` is no longer one or two, but the object that `x` is — this is its left value. Everywhere `x` is written in an assignment, it refers to the same object. But on the right hand side of the assignment, we mean by `x` the value in that object, and on the left hand side we mean the object *itself* as a container for values. So if we think of objects as boxes, the right hand value is "get me what's in the box," and the left hand value means "the box itself." Overall, the assignment means "put a value (on the right) into the box (on the left)." We've got a mathematical sense back because now `x` where ever it occurs means the same thing: it always refers to the same object, but on the left and right sides of = we refer to it in different ways. In contrast, in maths, $x=x$ has the name x meaning exactly the same thing on both sides of the equation: in maths, = is symmetric. In programming languages, as an assignment = is an instruction and it is not symmetric. [Those last two sentences could have ended with the pedantic hedge 'usually,' since the notations of both maths and programming languages are designed to do what we want them to do, and we don't always want them to behave conventionally.]

In languages like Java and C, we write the assignment `x=x+1` to mean 'increase `x` by 1.' With the help of the idea of left and right values, we can explain clearly what is happening (we can also explain more advanced features like &).

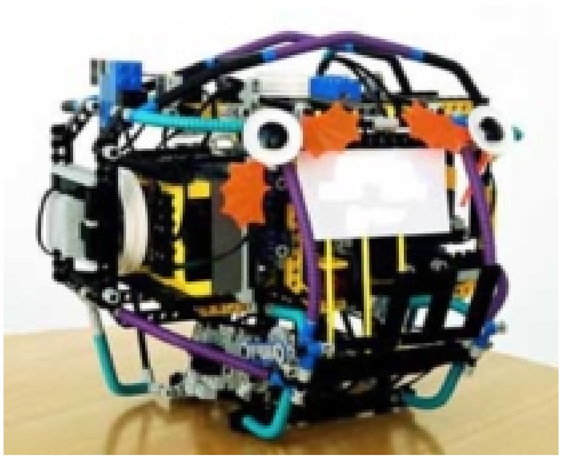
Sticking assignments together

In mathematics, an equation like $x=x+1$ is generally false; that is, there is no x such that one more than it is the same thing (though x could be infinite). But we could write $b=(x=x+1)$ as a 'bigger' equation and expect this to mean here that b is false. In the programming languages C and Java there is a similar idea, though since assignments are actions, their value is taken to be the value assigned. So `x=2` is an assignment that means 'make `x` equal to 2' and it also has the value 2 itself. So `w=(x=2)` has the effect of making both `x` and w equal to 2. (Since (w=x)=2 makes no sense in a programming language, w=x=2 is always taken to mean w=(x=2), that is, assignment is right associative.)

Back to building blocks. If x=2 is a brick, then w=x=2 is sticking two bricks together, namely the assignment for x and the assignment of *that* to w. In the RCX programming language NQC, this way of sticking bricks together is not allowed. So trying to say w=x=2 results in a programming error. It's as if NQC does not have this shaped brick. As in Lego, one can usually get round the problem of a missing brick. In NQC, one would have to write: x=2; w=x for what can be more concisely written in C as w=x=2.

Java

A robot called Jitter, built in Lego Mindstorms and programmed in Java, worked on the International Space Station in 2001. Interestingly, the 16MHz 48k Lego RCX is a more powerful computer than NASA had in the Apollo Lunar Lander module. Things have changed — now children have access to what adults could only dream about in the 1960s! I think Jitter was made of standard Lego, except it was glued together to stop any unfortunate accidents in space.



Java is a much more sophisticated language than NQC or C. Assignments work as we've described them, except that Java has a *very* sophisticated idea of what scope is. Variable names can be written in the form x.y, and this means "the value of y understood in the scope of the object x." The object x itself is evaluated in a scope, and Java allows long scope descriptions, such as u.v.w.x.y if they are needed.

Let's not worry about how Java defines scopes (or lots of other interesting things Java does, which don't concern us here), but we'll start with the following definitions:

```
class Cat { int paws = 4; }
class LameCat extends Cat { int paws = 3; }
```

which create two related scopes, one of Cats, where paws is a variable (called a field in Java) which has the value 4, and one of LameCats, which are related to Cats, but which give paws the value 3.

Let's create a new variable: Cat tiddles = new LameCat(). This is legitimate, because lame cats were defined as a sort of cat. Within the scope of tiddles we can refer to paws, as in tiddles.paws. Now tiddles is a lame cat,[*] so we might expect the assignment p=tiddles.paws to put 3 into p, but in fact p becomes 4.

What has happened is that making tiddles a Cat has *hidden* the definition of paws defined by LameCat. Eh? As the book on Java says (p76), "... where fields are concerned, it is hard to think of cases in which hiding them is a useful feature." In other words, the bricks of Java combine in confusing ways, which even the designers of Java acknowledge. "Eh?" by the way, is an abbreviation for an explanation that the margins of these notes are not big enough to contain.

Experienced Java programmers are likely to blame me for this confusion rather than the design of Java. We can escalate the problem, however. Let's suppose that cats can their count their paws:

[*] A Java programmer can confirm this by trying tiddles instanceof LameCat, which is true.

```
class Cat
{    int paws = 4;
     void countPaws()
     {    System.out.println("Cat has "+paws+" paws."); }
}
class LameCat extends Cat
{    int paws = 3;
     void countPaws()
     {    System.out.println("LameCat has "+paws+" paws."); }
}
```

We've now asked the cats to 'think aloud' when we ask them to count their paws (…with one more clause we could get them to count claws).[#] If we now ask our `tiddles`, we can get it to say, "`LameCat has 3 paws.`" Now that means that the method `countPaws` uses the lame cat to do the thinking, but it isn't looking at the same paws as `tiddles` did before! Eh? This looks inconsistent, and should certainly make a programmer pause for thought — and it's a problem that arises because the designers of Java have put in a 'brick' they don't think is a really useful feature! In a sense, Java is a programmer's Lego: it looks like you can build cool things with it, but by this analysis, it doesn't look strong enough to make real-world things out of, certainly not if you want them to be reliable enough to use. See Thimbleby (1999) for other such issues.

Mathematica

Mathematica is a programming language with a very impressive built-in knowledge of maths. It is designed for people who want to do mathematics. In any other sort of language getting the maths to work would have to be programmed explicitly. For example *Mathematica* can solve equations directly. If we wanted to solve the equation $x=2x-1$ we used earlier in *Mathematica*'s notation we'd just write `Solve[x==2x-1,{x}]`, and we'd get the answer 1 directly.[†] *Mathematica* therefore combines a programming language idea of names, which it needs for assignments, and a more normal mathematical idea of names, which it needs for solving equations. The two ideas can clash with quite surprising results.

From the *Mathematica* book (p289, 4th edition) we are told about the relation between the replacement operator `/.` and the assignment operator `=`. Essentially, `a=b` does an assignment so that henceforth `a` is taken to have the value `b` (this is like NQC or Java). In contrast the `/.` operator (called '`ReplaceAll`') defines a specific scope where a rule, such as `a` having the value `b`, is applied. In essence but, as we'll see, *not* in detail, the assignment `x=1` followed by `y` (which might be an expression containing `x`) means much the same as the expression `y/.x→1`, because in both cases any `x` in `y` is given the value `1`. The latter is close in spirit to the more mathematical turn of phrase, y **where** $x=1$. As in everyday maths we might write, say, $x+1$ **where** $x=2$ to mean 3, in *Mathematica* we can write `x+1/.x→2` and get 3 too — as we'd expect.

The *Mathematica* book gives the following example: `(1+x)`^{6}`/.x→3-a` which results in `(4-a)`^{6}, because it means "take `x` to be `3-a` in the expression `(1+x)`^{6}," so *Mathematica* finds the result to be `(4-a)`^{6}. A simpler example would be `f[x]/.x→a` which gets `f[a]`. Whatever `f[x]` is, `f[x]/.x→a` means work it out for `x=a`.

Continuing with the examples from the *Mathematica* book, it now tries the assignment `x=3-a` and repeats the calculations (though with a different exponent, 7 instead of 6). Now `(1+x)`^{7} gets `(4-a)`^{7} and indeed, we can do `f[x]` after the assignment and get `f[3-a]`, as we would expect, because after doing `x=3-a`, `f[x]` must be `f[3-a]`.

[#] What's the difference between a cat and a comma? One has claws at the end of its paws, the other is a pause at the end of a clause.

[†] As in C and Java, in *Mathematica* = means an assignment, and the assignment `x=2x-1` is not an equation: instead, it means "make `x` one less than twice what it was." Since = means assignment, *Mathematica* uses == to mean equals — hence the apparent verbosity in our `Solve` example. The `{x}` is needed because we have to tell *Mathematica* to solve for `x`, even though there is no other variable in the equation — but usually there will be.

It's a fair comparison to try `f[x]` instead of the book's examples of `+` and raising to a power, since `+` and so on are just functions, in this case with built-in names `Plus` and `Power`: for example, `FullForm[(1+y)`^{6}`]` shows us the underlying form `Power[Plus[1,y],6]`. Note that I had to use `y` here because if I'd used `x`, it would have had the value `3-a` hanging around from the last assignment we had done to `x`. Indeed, the *Mathematica* book warns about the problems of forgetting to remember you've assigned values to variables: it's the single most common mistake in *Mathematica* programming, apparently.

Now let's define our own function for `f`, so it works out factorials: `f[n_]:=n*f[n-1]; f[0]=1;` This means that (in ordinary mathematical language) $f(n)$ is defined to be $n\times f(n-1)$ and $f(0)$ is 1. This is a perfectly reasonable definition of factorials, and indeed one directly based on an example from the book (don't worry about the *two* forms of assignment used here!). And in *Mathematica* it works correctly: `f[3]` gets 6, as indeed it should, as 3!=6;and `f[4]` gets 24, and so on.

Let's try the sorts of examples the book gives, but using our function `f`. Surprisingly, using a rule, `f[x]/.x→3` gets `$RecursionLimit::reclim: Recursion depth of 256 exceeded.` (which is said twice) and then incorrectly returns the result `0` (rather than `$Failed`). In contrast, using an assignment, `x=3; f[x]` gets `6` correctly, without any problems. So clearly assignment and rules work differently, despite the explanation in the book. Eh?

Deassignment

Mathematica uses a special deassignment operator: the notation `x=.` means remove any value assigned to `x`. Here's an example of how it might be used: after the assignment `x=3` the expression `x+1` will get `4`, but after the deassignment `x=.` (even after `x=3`) `x+1` will get `1+x`. In other words, after a deassignment, `x` becomes a symbol rather than a variable denoting a specific value, and *Mathematica* cannot calculate `x+1` numerically, so it does it symbolically.

In an ordinary programming language like Java, although variables can change values (as a result of assignments) variables cannot have their values *removed* from them. The reason *Mathematica* needs this feature is because it doesn't otherwise distinguish between mathematical variables and program language variables. In *Mathematica*, you must use deassignment if you want to use a variable as a mathematical variable after you've used it even once as a program variable. If you forget, something innocuous like `Solve[x==2x-1,{x}]` — which we used as an example above — results in an error or an incorrect result, and possibly a problem that you don't notice (and find hard to sort out when you do).

It gets very confusing. Doing the assignment `x=3` makes `f[x] = 6` (because 3!=6) and of course `g[x]` gets `g[3]`, because we haven't told *Mathematica* what `g` is, so it can't do any better than symbolically say `g[3]`. As we would expect, `g[x]/.x→4` is `g[4]`, by using the rule that `x` is `4`. Yet `f[x]/.x→4` is `6`, which is giving us the value for `f[3]`, not for `f[4]`, which is `24`. Somehow the rules for getting the value of `x` change depending on whether it is used in defined functions. Eh?[‡]

It gets worse. It goes wrong, in different ways depending on the context, when `x` looks like it is bound as a formal parameter, which seems to be bizarre. Unfortunately there isn't space to discuss this deep problem here. If you have *Mathematica* handy, compare the behaviour of `f[x_]=x^2` and `f[x_]:=x^2` when `x` has and hasn't been assigned a value; then look up the book to see what it says. Try it again inside modules and blocks, and you'll get different results.

[‡] For experienced *Mathematica* programmers, it is worth noting that deassignment is often not what you want to do (you may not want to discard a variable's value you've gone to some trouble to calculate), so *Mathematica* provides numerous *ad hoc* functions like `Hold`, `HoldForm`, `HoldPattern`, `Unevaluated`, `Evaluate` and `ReleaseHold` so that you can more finely control whether a variable is treated as a symbol or as referring to a value, *etc*. You can also `Clear`, `Remove` or `Unset` names. And so on; new functions are added regularly. The use of such functions gets *exceedingly* obscure when combined with rules like `x→4` (which also come in several forms!). It's arguable that this complexity could have been avoided if the bricks had only been sensibly shaped to start with.

Feature interaction

I don’t understand what is going on, whether I read the *Mathematica* book or do experiments. I suspect the implementers of *Mathematica* had problems too. The big picture is easy to appreciate with our analogy of bricks: if the specification of bricks is sufficiently complex they cannot be made exactly as required. Over time, the specification may be changed to conform better to the bricks actually supplied, but with revised specifications the next production run of bricks will introduce new variations — and so on. If you have a building made a few years ago, you won’t be able to get replacement bricks for it.

The killer is that in the real world we can’t talk about bricks in isolation. We might be happy with the house we live in and we don’t intend to change it; but when we come to upgrade our car, we find we have to rebuild the garage. When we buy the bricks for the garage, we discover the kitchen stops working, so we fix that, and then we discover the tables in the dining room won’t handle the place setting we’ve used for years. We probably need to get a bigger car to buy a new table! The problem is called ***feature interaction***, and its prime cause is having too many curious features to start with.

Conclusions

The analogy made between playing with child’s building bricks and programming has been made, and we pursued in some detail one programming building brick, the assignment statement as it occurs in a few different programming languages: NQC, Java and *Mathematica*. There’s a lot more to programming than this, of course — indeed, some programming languages do without this particular building brick completely (like Meccano is an alternative construction kit that doesn’t use bricks).

When a child builds a house in Lego, you don’t automatically think the child will become an architect or a structural engineer, though you may spot some flair. However imaginative the house was, you wouldn’t confuse the model house for a solution to the British housing shortage. And certainly you wouldn’t think the house could be made full size and still stand up. We make all of these mistakes with computer programming. Many programming failures in the real world happen because ‘toy’ programs don’t scale up to the real-world tasks given to them. Since they were built rather than designed, they have no structural integrity and have to be exhaustively tested — to fix numerous bugs. But this is an endless task. If we built real houses out of Lego, it too would be a pointless task — though perhaps not as pointless as building real cars out of Lego! To build real houses, use real bricks (and real builders).

When programming languages, such as Java and *Mathematica*, have complex rules (such as the ones we have discussed for assignment) programmers have no choice but to build programs and then play with them to see whether they will work as intended. Bricks fit together in weird ways. Although this lecture has only given very brief examples (and a lot has been left unsaid) it should be clear that the meaning of programs is not at all clear. A programmer would have a very difficult, if not impossible, job if they tried to design what they wanted to build before starting. Like a Lego builder, it’s far easier just to start by building than to start by thinking.

If we want better computer systems, we could start by using programming languages that have reliable properties. Whereas builders wouldn’t waste time using Lego, there are very few programmers out there who realise that they could ask for different and better bricks.

References

K. Arnold, J. Gosling & D. Holmes, *The Java Programming Language*, 3rd. edition, Addison-Wesley, 2000.

D. Baum, *Definitive Guide to Lego Mindstorms*, 2nd. edition, APress, 2003.

B. Erwin, *Creative Projects with Lego*, Addison-Wesley, 2001.

H. Thimbleby, “A Critique of Java,” *Software—Practice & Experience*, **29**(5):457–478, 1999. Available at `www.uclic.ucl.ac.uk/harold`

S. Wolfram, *The Mathematica Book*, 4th. edition, Cambridge University Press, 1999.