# The End of Innocents

## Strategies for a 21st Century Pied Piper

### or, Why Kids Think Computers Aren't Cool Any More, And How To Cope

Reuben Thomas

11th–12th March 2006; revised 20th March 2006, 13th May 2008

## A Unique Privilege and A Common Predicament

We (by "we", I mean members of my generation, that commonly denoted "X") lived through a magic time. In 1975 the first computer marketed at individuals was launched in kit form; by 2000 most middle-class homes had at least one PC, and many users spent longer using the internet each day than they did watching television. In between was a time when computers were simple enough to be understood from top to bottom: one could buy a machine, or even build it from components, then learn to program it, and achieve results comparable with commercial software for the same machine.

Yet, only a few years later, children are growing up taking for granted hugely powerful and complex computers that they use for diverse entertainment and social purposes, but rarely show any interest in taking the lid off. (It is noteworthy that whereas in the 70s one had to assemble a computer before one can program it, one now has, virtually at least, to disassemble it.) Why does this matter? One could argue that programming is simply becoming like the majority of adult disciplines, one to which adults, not children, are attracted (and one to which young adults will flock readily enough when they realise the impracticality of becoming a racing driver or fireman). Kids don't love maths or law either, but we still get our mathematicians and lawyers. If only this were so, but I chose my sample adult disciplines carefully: mathematics and law are different. We do in fact have a shortage of mathematicians, and it's getting worse. We also already have a shortage of programmers (in fact, there has never been a superabundance), and it too will only increase.

What's special about programming? Programming is a mathematically-based discipline: it requires the ability to reason formally, that is, to use systems of

axioms and rules of inference to prove conclusions. Most disciplines have some element of formal reasoning, often called "logic", but do not utterly rely on it. It's also worth noting that "formal reasoning", or "mathematics" for short, has little to do with numbers (perusal of an elementary number theory text will convince doubters), although it is an ability whose lack should be bemoaned as much as deficiency in numeracy and literacy are. Formal reasoning skills, like numeracy and literacy, are best learned young, as adults find them much harder to learn than children.

Seeing how hard it is to excite children about maths and science, why should computing be any different? Before getting too depressed, let's examine what happened in the "magic time".

## The Magic of Childhood

Why is childhood special? It is the child's limitations that make the world a magical place: their limited understanding and power simultaneously fills the world with arbitrary terrors (or in the managed modern world, tedium) and fanciful delights. (Note how once again ontogeny recapitulates phylogeny.) Similarly, it was primarily the limitations of computers, in their own childhood, that made them magical, particularly to children, who perhaps subconsciously related to their immaturity. But first, let us quickly consider the advantages of computers circa 1985 as objects of interest for budding programmers. "Home computers", as they were often called, had two key advantages for budding programmers: they switched on instantly, like all the best toys, and they had a built-in programming language, usually a dialect of BASIC. These two features meant that you could start writing a program in seconds, with an ease often mourned today by those old enough to remember. The other attributes of home computers which encouraged programming were mostly negative. (This should not be surprising: "necessity is the mother of invention" is just a special form of the more general observation that creativity is often proportional to the degree of restriction under which it operates.) The most important were:

**BASIC is all you got** Commercial games were often not much better than what you could write yourself. (Children were of course mostly interested in writing games!)

**Computers were simple** Simple enough to be understood top to bottom. Children enjoy the feeling of mastery this gives, and the more ambitious learnt how to delve into the machine's internals.

**Computers were small** Small enough that a good game could still be short enough to type in by hand and learn from by example. Even commercial software usually had a single author; many programmers were still teenagers when they started earning money from programming.

**Computers were slow** Slow enough that one needed to understand the machine thoroughly to obtain speed or special effects which are now part

and parcel of the complex hardware and vast programming libraries that programmers command.

**Computers were isolated** Perhaps most importantly, home computers were mostly not networked. Communication and computation may be theoretically equivalent, but given the choice between browsing the web and writing a program, most people will choose the former. Communications channels focus attention on the message even when that includes the medium; the mechanism is generally ignored.

Generation X literally grew up with computers. We don't necessarily understand them better than those who came after (or before) us, but we have a unique and unreproducible perspective, as brief as that of the pioneers and perhaps as important. (Let's hope we get some good memoirs from this generation.)

Given that this special time is over, what can we really do to fire the imagination of the next generation of programmers, other than the tried-and-tested tricks of the other formalists, namely, getting children to play games on our terms, and sneaking formal thinking into everyday contexts? In fact, we do have one spark of magic left: while most formal subjects have a purely platonic existence, computers are a physical embodiment of programming; not only that, but they are reactive. Our programs have a life of their own, and that is still attractive to children. We'll now look at some concrete methods to put this into practice.

## Harnessing The Remnant Spark

We'll now examine several ways to get children interested in programming, and look at some bad as well as good ideas, to try to give a rounded overview of the field.

### We Don't Heed No Education

Educational software can (and should) be disposed of quickly. At best, it's the naïve result of a misguided belief that the only thing wrong with Kennedy's Latin Primer is that it's not animated and fluffy enough; more often it's lazy hypocrisy typical of initiates that what was good enough for them is good enough for the next round of novices (but if it's on a computer it's modern, right?). At its worst it's one of the more dangerous manifestations of the techno-idiocy that seems to think that humans can be replaced by machines. When children don't see through it and switch off immediately, they end up feeling duped and risk losing respect for education.

### There's No Turning Back The Clock

Some people think that if only we had computers that behave as they did in the 1980s, children would still be excited by them. Sadly, it wouldn't work. The childhood of computers I described above was preceded by the childhood

of electronics, which similarly captured a generation through electronic kits. However, by the 1980s building your own radio simply wasn't exciting to most children, who probably owned at least one themselves, and with whose functionality, size and stylishness kits could never hope to compete. Fortunately, it turned out that society simply didn't need that many electronic engineers. (It's worth noting that outside the context of this article, namely, the developed world, building simpler, smaller, cheaper computers might well be a winning strategy, and not just for children. Nicholas Negroponte's technically ingenious but politically fatally flawed eMachine initiative to build sub-$100 PCs for the developing world is arguably woefully under-ambitious: why not a sub-$10 computer? As so often, by trying to build the simplest thing possible, we might come up with something simple and reliable enough for the real world.)

## We Must Speak Their Language

The modest success of Logo illustrates what I believe is the most promising approach: engage children with specially tailored programming languages and environments. However, recently examples of this have focused on the wrong thing: trying to keep children's attention, they have, like educational software, concentrated on eye (and ear) candy. This just makes them look incompetent next to good commercial games, and in any case, as good games writers know, what really matters is the gameplay. Also, systems like Alice and KPL make the same mistake as Logo of letting children know that they are designed especially for them. By the time children are mature enough to learn to program, they think they want to be treated like adults.

What should a programming environment for kids be like? Logo points the way: it should be based on a formally sound language, and it should be strongly connected to the real world. The closest thing in existence is Carl Sassenrath's REBOL, which has many excellent features:

**Simple and uniform** It is based on LISP, but a more functional notation with a smattering of infix operators avoids the uncontrolled proliferation of parentheses. In short, it retains the wonderful purity, elasticity and "code is data" magic without being physically painful to look at.

**Small and ubiquitous** It runs on all common PC operating systems, and consists, in common incarnations, of a single smallish binary.

**Real-world datatypes** REBOL understands quantities like currencies, colours, IP addresses and dates.

**Strong connections to the real world** Similarly, REBOL comes with simple programming interfaces for many internet protocols, including HTTP and SMTP, and a fully-featured GUI system, all in the small standard distribution package. Most current areas of interest can be explored in a few lines of code, even more easily than in BASIC of yore (more like the souped-up late-80s and early 90s dialects STOS and AMOS, themselves hugely popular, though rather late in the day, and once again aimed at creating games, that is to say, fun programming).

REBOL is a good start, but there are a few other facilities that would help to grab a child's attention:

**Internet integration** REBOL's internet support is rather low-level; it has not yet caught up with the recent explosion of web services, in particular as defined by Google. It should be possible to easily fetch data such as exchange rates, atomic weights, cometary orbits and parrot plumage and use them in programs, or to use pictures from Flickr for feature recognition, maps from Google Maps for population simulation, or one's email account on Yahoo for textual analysis. The calculation language Frink has the beginnings of this sort of understanding. REBOL runs in parallel with the web; these days it's hard to imagine how something that can't actually run in a browser could be a real winner (although being able to run separately too would be a great bonus).

**Simulation** It should be much easier to build large-scale simulations: a flock of birds, a planetary system, a molecular assembly, a barter economy. As well as general facilities for discrete and continuous simulations, a wide range of physical laws and other governing equations should be built in, along with an understanding of units (like Frink's).

**Ubiquity** The programming environment should follow the child around, from desktop to phone to laptop, from home to school. Technologically a solved problem, it still needs to be made practical and usable. It should be at least as portable as a favourite doll.

**Socialisation** Recent popular computing technologies have all become popular because they support socialisation. A programming language that a child could use side-by-side with other activities would be much more likely to be used.

**Integration** A corollary of the previous two characteristics is that the programming environment should not be a separate thing, but integrated with all the software a child uses, so that it is never far from their attention.

In summary, I think that the best way to hook children is to give them an attractive and ubiquitous programming tool. The importance of teaching it to them is easy to overstate: children do much of their learning on their own or from each other, and this has historically been true of programming in particular. However, precisely because children are increasingly using computers to communicate with each other and find out about the world, computing environments have become an important factor in the socialisation of children. In a way they are even more important than, for example, the school environment, since they will live much of their lives in cyberspace. Thus, before concluding, I'd like to adopt a slightly more fanciful stance and explore ways and senses in which we might set computers up to educate and socialise children more generally.

## Hacking Barbarians

Children, as Roald Dahl observed when writing on rail safety, are ignoble savages. Like most savages, they do not like authority, and they are not stupid. Sadly, they often end up resigned to authority and acting stupid, or worse. One of the things children like best about computers is precisely that they can be tools for working around authority, and one of the things that society needs most is adults who, while they respect authority, are competent and passionate subversives. We should be thinking about ways in which kids can (surreptitiously if necessary) be encouraged to do social hacking. Such encouragement probably needs to be hidden from the kids themselves, and from their teachers and parents; the best way to do this is to ensure that everyone is trying to do it. Guerrilla education is perhaps the best general substitute for that rarest of gems, the loved and respected guru, whom few can hope to have. (Why is so little written on this subject?)

## Enlightenment: We Are Doomed

Despite our little remaining magic, we in computing are as doomed as the mathematicians and scientists; all we can hope to do is parlay the percentages into a slight edge; in any case, it's not in our long-term interest to beat the other formalists, as we're all in this together. To be learnt properly, computing needs to be learnt young, so we must strive to implant the seed of formal thought in as many young minds as we can reach. It's a worthy enterprise: intelligent society depends on it (and not, as is often erroneously implied, merely technological society, though those who despise the latter often fail to understand the former). But, without giving up hope of one day finding a simple solution, teaching children the love of formal thought is in the NP-complete category of educative problems, because it means getting them to believe that creativity is better than consumption, and those who do so have attained enlightenment.

## Acknowlegements

Eben Upton, as Director of Studies in computer science at St John's College, Cambridge, brought to my attention the fall in applicants to Cambridge and in particular his difficulty in finding good candidates to read computer science. Julian Midgley commented on the first version.

## Lacunæ and further reading

Eben Upton believes (private communication, March 2006) that bribery will work; his give-away TV-compatible ABC computer is based on this premise. I believe this will only be successful if it makes smart kids who would have directed their energies elsewhere reconsider.

*The Little Coder's Predicament* by _why (`https://github.com/hacketyhack/hacketyhack/wiki/The-Little-Coder%27s-Predicament`), a similar, earlier

essay to which I was directed in July 2006, endorses my language-centric approach.

*Why Johnny Can't Code* by David Brin (http://www.salon.com/tech/feature/2006/09/14/basic/), who notes the problem, but has the misguided idea that it's important to learn BASIC in order to understand the "deeper patterns" underlying programming (yes, programmers should understand how computers work, but that's not the big problem, nor is one of the worst languages ever designed a solution). The article has inspired KidBASIC (http://kidbasic.sourceforge.net/).

James MacKenzie (private communication, January 2008) suggests that I've overlooked the importance of being able to program hardware directly, as one could, say, from BBC BASIC (with peeking and poking), but can't from the standard installations of most of the first languages one is likely to use today. That may be true; I didn't think of it because I didn't play with hardware on microcomputers when I was learning to program. It seems reasonable to imagine, though, that many or even most children would be more excited by playing with real gadgetry than merely making pretty pictures on their screen. I'm not sure however that even gadgets are as exciting as dealing with the big wide world via the internet; the Polish boy who derailed his local tram system would seem to agree (http://www.theregister.co.uk/2008/01/11/tram_hack/), though he is a bad example of another benefit of introducing programming via the internet, which is that it leads naturally via communication with people and real-world data sets to consideration of the wider world and adult involvement in it, in a way that programming video games doesn't.

Bill Thompson outlines the problem after a history lesson in *Who will write tomorrow's code?* (http://news.bbc.co.uk/1/hi/technology/7324556.stm).