Informatics in Education, 2007, Vol. 6, No. 2, 307–320

© 2007 *Institute of Mathematics and Informatics, Vilnius*

Should LOGO Keep Going FORWARD 1?

Ken KAHN
Animated Programs, Oxford University, and London Knowledge Lab of the Institute of Education
23-29 Emerald Street, London WC1N 3QS, United Kingdom
e-mail: toontalk@gmail.com

Received: August 2007

Abstract. LOGO has been evolving in incremental steps for 40 years. This has resulted in steady progress but some regions of the space of all programming languages for children cannot be reached without passing through unacceptable intermediate designs. What are the ultimate aims of LOGO? What criteria and aesthetics should be used in determining which areas of the design space are most promising? What would the ideal programming language look like? Would a family of special-purpose languages be more effective than a single language?

In looking to the future what can we learn from the history of LOGO? What can we learn from other programming systems for children? Alan Kay is leading a new project entitled, "Steps toward the Reinvention of Programming". What are its strengths and weaknesses?

We can conceptualise the design alternatives as defining an n-dimensional space. Some dimensions represent major alternatives for syntax, others for dealing with concurrency, others for the underlying computational models, and others for features of the programming environment.

The goal of this paper is to spur a discussion of these issues. I will present my personal opinions based upon 30 years of research experience in this field.

Key words: future of LOGO, Smalltalk, ToonTalk, StageCast Creator, AgentSheets, NetLogo, StarLogo.

The Trajectory of LOGO

> I could not agree more with you about current Logo being out of date and am planning to immerse myself in thinking about what a language for 2005 (or so) could be.
>
> Seymour Papert (personal email to the author, June 14, 1999)

The Past

In the last 40 years LOGO has moved FORWARD 1 many times. Sometimes it has been cloned and copies have headed off in somewhat different directions. Colour was added. 3D was added to some branches of the LOGO tree. Object-orientation and multiple turtles were incorporated. Concurrent processes were supported. Advanced user interface gadgets were added. There have been many valuable improvements.

Sometimes language designers imagined languages that could not be reached incrementally from LOGO and new child-oriented programming languages appeared else-

where in the design space. Smalltalk (Kay, 1993) was probably the first language that was inspired by the LOGO "philosophy" but not part of the language evolution. Boxer (diSessa, 1997) and ToonTalk (Kahn, 2001b) are other examples. These languages borrowed some of the powerful ideas of LOGO but did not grow out of LOGO itself. Other languages such as AgentSheets (Repenning *et al.*, 2000), StageCast Creator (Smith *et al.*, 2000), and Alice (Conway *et al.*, 2000) share many of the goals and ideas underlying LOGO but developed without explicit influence from LOGO.

In some cases, separate language trees have branches that share the same region of design space. The Etoys system of Squeak/Smalltalk (Allen-Conn and Rose, 2003), for example, is similar to some modern LOGO systems.

Possible Futures

One future is for LOGO to continue to improve incrementally. The difficulty here is that radical improvements are not incremental and are frequently disruptive. The intermediate points in the design space are often not viable: being "neither fish nor fowl".

Another future is described by Alan Kay and colleagues at the Viewpoints Research Institute in a recently funded 5-year project (hereafter called the VPRI Project) to reinvent programming, especially for children (Kay *et al.*, 2006). It proposes to build a new computing system and programming language with many innovative properties: self-explanatory systems, separation of intent and optimisations, a self-modifiable implementation, extreme portability, well-integrated and transparent tools and operating system functionality, and a classless prototype-oriented object model.

Another future is that papers such as this one sparks discussions in the wider community that lead eventually to the design and implementation of a new language (or languages).

What are the Ultimate Goals?

What are our ultimate goals in designing the ideal programming language for children (and other learners)? One answer is that we want to give kids (and as many of them as possible) the power to express themselves computationally. Alan Kay expressed it well in his 1984 *Scientific American* article (Kay, 1984):

> The protean nature of the computer is such that it can act like a machine or like a language to be shaped and exploited. It is a medium that can dynamically simulate the details of any other medium, including media that cannot exist physically. It is not a tool, although it can act like many tools. It is the first metamedium, and as such it has degrees of freedom for representation and expression never before encountered and as yet barely investigated. Even more important, it is fun, and therefore intrinsically worth doing.
>
> ... Computers are to computing as instruments are to music. Software is the score, whose interpretation amplifies our reach and lifts our spirit. Leonardo da Vinci

called music "the shaping of the invisible", and his phrase is even more apt as a description of software.

We want a language that is truly general purpose and not based upon a limited special-purpose computation model. Or, as discussed below, maybe the goal shouldn't be a single programming language but a family of languages.

Besides giving children the ability to express themselves in fundamentally new and powerful ways, we also want to give them objects to think and learn with. This is the deep idea underlying Papert's *Mindstorms* (Papert, 1980).

Thirdly, we want to give children new mathematically interesting objects to think *about* as well as think *with*. One can direct one's thoughts at a programming language itself and appreciate it as a mathematical object and as a model of computation. Perhaps Lisp and its underlying lambda calculus are at least as worthy of study as, say, trigonometry.

I think LOGO has compromised the goal of providing a language to think about by giving priority to other goals. Lisp and Prolog, for example, can describe themselves in very short programs. LOGO, with its distinction between functions and commands, its syntax, and the special forms (*IF*, *WHILE*, etc.) of many dialects, make it much harder to implement LOGO in LOGO. And it makes it harder to think about how LOGO works and what it is.

The VPRI Project led by Alan Kay aims to build a language and system that is simpler and better suited for self-description and self-inspection. It also is exploring the possibility of making the system *self-explanatory*. Their plan is that the system can explain its own structure and operations. Using AI techniques, it will automatically generate explanations to support end-user exploration.

Whose Design Aesthetics?

The design of LOGO was largely based upon Lisp. The design of Lisp was largely based upon the *lambda calculus*, a branch of mathematics. This makes Lisp (and to a lesser extent LOGO) a language that is not only a good tool for expressing programs but an object to think about and with. One consequence of this is that Lisp is well-suited for meta-programming. Programs can construct other programs. Programs can reflect upon themselves. This is a consequence of the small powerful kernel underlying Lisp that is based upon lambda calculus. Alan Kay in a talk at Stanford in 2003 described Lisp as "a mathematical object that can see itself". The idea of a self-referencing language kernel is one of the inspirations behind the VPRI Project. The kernels of some programming languages such as Lisp, Prolog, concurrent constraint programming languages, and functional programming languages have a mathematical beauty. A very small basis set can generate incredible richness. In contrast, languages like Smalltalk, Oz, Python, and Java were designed by computer scientists. The beauty and elegance of these languages seems more akin to that of engineering than mathematics. We either have to choose which aesthetic will drive the design or take the gamble that we can design something that simultaneously is a mathematical and engineering jewel.

If one is pursuing mathematical beauty then basing the language design on well-established mathematics is much easier and safer than also inventing new mathematics. For example, a language can be based upon lambda calculus which is a theory of functions or instead on a calculus of processes such as the *pi calculus*. Perhaps we should be building languages based upon processes rather than functions since processes are more valuable and fundamental. If so, the design of the ideal programming language for children should focus upon doing concurrency right and not be as concerned about functional programming.

It may be that in practice even the mathematically beautiful programming languages are complex engineering artefacts. Maybe the structure and elegance of the kernel matters little if programmers need to put most of their efforts into understanding large engineered libraries of useful components. Not much real programming is done completely bottom up from language primitives. And even if someone builds a large program "from scratch", much of their cognitive effort typically goes into designing and using the higher level chunks of code.

I think this situation is analogous to the idea of reductionism in science and philosophy. It is a great achievement that one can understand everything in the universe in terms of atoms, or elementary particles, or quarks, or super strings. It is important that things can, "in principle", be reduced to primitive elements. In theory, it can explain why you can't put a round peg in a square hole. But usually it is not the level at which informative, helpful, or satisfying explanations come from. Similarly, I think it is great if one can see that a complex program bottoms out ultimately in a small set of primitives even if most of one's thinking about programs occurs at higher levels of organization. In science one discovers new entities and laws that emerge at different levels. Maybe we should be looking for these emergent properties at higher levels of software as well.

The VPRI Project aims to go one step beyond an elegant software kernel by connecting the kernel primitives to the physical hardware in a transparent and customisable fashion. By building upon some clever bootstrapping techniques they plan to build self-describing kernels that also describe mappings to the machine language of the target machines. The hope is that curious non-professionals will be able to understand the entire system down to the metal.

A very Brief Introduction to ToonTalk

I began designing and building ToonTalk in 1992 (Kahn, 1996). I was inspired by Seymour Papert's description of LOGO as taking the best ideas from computer science about programming languages and environments and "child-engineering" them (Papert, 1977). I took computational ideas from the concurrent constraint programming field (Saraswat, 1993) and user interface ideas from computer games and made a language that looks and feels like a game but is really a powerful concurrent programming language. The child enters a virtual world and constructs programs by training animated robots to manipulate boxes, birds, trucks, numbers, and various other tools. Programs are constructed by demonstration with examples (Kahn, 2001a). The fundamental idea is that sophisti-

cated computational abstractions can be made accessible by providing concrete analogues without loss of expressive power.

Design Aesthetic of ToonTalk

Is ToonTalk an example of a design guided by engineering aesthetics? It appears to be in the engineer/computer scientist family of languages. It has many more primitive elements and constructs than say lambda calculus underlying Lisp or the Horn clauses underlying most logic programming languages. And yet its design was directly inspired by concurrent constraint languages which are mathematical beauties. What happened?

ToonTalk has very few true building blocks. There are pads (atomic data), boxes (compound data structures), birds/nests (communication channels), and robots (program fragments). What about all the other things in ToonTalk? They are ways of expressing certain kinds of actions, not things themselves. A truck is not really a part of a ToonTalk computation but is a way of expressing the spawning of a new process. The Magic Wand is not a thing but a way of expressing the copying of other things. The helicopter is a way to monitor an ongoing computation at different scales and locations. Notebooks were designed as a way to obtain the functionality of a file system for persistence and sharing. But notebooks can also be used as an alternative to boxes and from a mathematician's view this overlapping of functionality is ugly.

ToonTalk often operates at a level below that of the concurrent constraint programming languages. In Janus (Saraswat *et al.*, 1990), you can understand communication as the asking and telling of constraints. In ToonTalk, the corresponding constructs are waiting for things to arrive on a nest (asking) or giving things to birds (telling). Despite a direct mapping between Janus and ToonTalk, it is hard to see the underlying constraint programming of ToonTalk.

Is the ToonTalk world too rich? Would a sparser set of primitives lead to a language that is a better object of study and contemplation? Would a sparser ToonTalk have more layers of software? Could these higher-level components be concretised as well as ToonTalk currently does? Many good questions remain.

The Ideal Programming Language

After 15 years of ToonTalk work and experience I now imagine an ideal language as retaining many of the concepts and strengths of ToonTalk with these differences:

1. **Tiny kernel**. The kernel of the language should be as elegant and powerful as possible. The design of a language like ToonTalk could be guided by the aesthetics of mathematicians.
2. **Multiple representations**. ToonTalk's support for programming with concrete examples in an action-oriented fashion using metaphorical analogues of computational abstractions could be augmented with corresponding static pictorial representations such as comic books (Kindborg, 2001) and a textual or tile-based symbolic language. Discussed in detail below.

3. **Composing components**. One lesson from several research projects that I have participated in (Playground (Hoyles, 2002), WebLabs (Kahn *et al.*, 2005), BBC (Kahn *et al.*, 2006), ReMath, and Constructing2Learn (Kahn, 2007)) is that a very effective way to enable beginners to quickly build programs they care about is to provide them with a library of easily composable high-level components. These components should run as autonomous processes with no or minimal interfacing. Children can start programming "in the middle" and move "up" by composing program pieces and move "down" to understand, modify, or create new components.
4. **Exact arithmetic**. ToonTalk's exact rational arithmetic (Kahn, 2004) could be augmented with exact irrational numbers (Boehm, 2004). Rather than mislead or confuse children with "leaky abstractions" for numbers (e.g., limited precision floating point numbers), we could provide them with exact implementations of rational and real numbers. One technical challenge with computable irrational numbers is that the determination of whether a number is equal to another may not terminate.
5. **Exact geometry**. Perfect or ideal geometry could be supported. We should explore whether it is feasible to build geometrical objects that are pixel perfect at any scale or size. Imagine a circle or a fractal consisting of dimension-less points that glow. As one zooms in on an increasingly small portion of the object, the screen pixels are computed accurately. At extreme scales, such as a googol-fold zoom, the system may begin to slow down as the exact rational or real arithmetic may become relatively expensive. Turtle geometry could also be implemented using exact real arithmetic (or an approximation that ensures that no anomalies are observable). Such an implementation would make "real" and "concrete" the idealizations of geometry.
6. **Time travel**. A better version of time travel could be supported. ToonTalk provides users with the ability to travel in time (Kahn, 2006). They can go back to an earlier time in their session and replay or revise the past. These time travel records can be copied and shared with others. The implementation sometimes imposes an unacceptable performance penalty upon the system but in other cases it is very useful for undoing, reviewing, and creating demos for others. A new implementation could dramatically reduce the performance cost in many cases. ToonTalk's time travel enables one to replay the past from the viewpoint it was viewed originally. A more general and useful version would allow one to "move the camera" while replaying and maybe to join the scene as an (observer-only) ghost until such time as one is ready to fork the time line.
7. **Debugging tools**. Children are rarely given a powerful set of debugging tools. Too often they have few alternatives to adding *SHOW* or *PRINT* statements to their programs. Time travel is a powerful debugging tool but it is not a complete solution. Children should have the kinds of tools that a modern Integrated Development Environment such as Eclipse offers. The challenge here is to child-engineer the user interface without reducing its power.
8. **Run everywhere**. Programs can run on mobile devices, web browsers, and robotic construction kits. While I expect that programs will typically be constructed and

debugged from inside a virtual world, they could run as well in browsers, mobile devices, and other platforms. ToonTalk programs currently can be automatically converted to Java and run as applets in browsers. Perhaps something similar could be done to run on other platforms as well. It probably isn't practical to expect the programming environment to run on these more limited devices or contexts but when an application can be isolated from the environment it should be possible to generate a stand-alone version with much lower run-time requirements.

9. **For professionals too**. ToonTalk is rarely used by professional programmers to accomplish their tasks. Perhaps the ideal language for children and non-professionals should provide support for professionals as well. NetLogo has succeeded in simultaneously supporting students and researchers in building and exploring simulations of multi-agent models.
10. **Distributed implementation**. ToonTalk's model of concurrent and distributed computation should be preserved but with a more general and flexible implementation. A distributed persistent implementation (perhaps building upon distributed hash tables) could provide a foundation for inter-process communication to achieve peer-to-peer distribution, persistence, and scalability. This could provide a unique and powerful means of collaboratively programming over the network.
11. **Inside a virtual world**. Programming would take place within a 3D persistent shared on-line virtual world with an integrated physics engine. Discussed in detail below.

Another idea is that the language should run with zero installation, possibly as a web service accessed via a browser. The storage of programs on the web service would enable students to seamlessly continue to work on projects as they move from school to home and back again. The implementation constraints entailed by web services will make the above list of enhancements even more challenging to accomplish.

How should the Ideal Language Support Concurrent and Distributed Computation?

Over the last ten years, I've had many technical discussions about why I believe that ToonTalk's model of concurrency is better than that provided by multi-threaded LOGO implementations (including StarLogo and NetLogo) and Squeak.

Very briefly, what we want are not multiple processes that cause side effects on the environment, but processes that also consume and produce data. These processes need ways to communicate, synchronize, and coordinate via primitives that are accessible by non-expert programmers. The way ToonTalk does this enabled, for example, the computational exploration by children of infinity and cardinality (Kahn *et al.*, 2005).

Concurrency combined with destructive operations upon shared data and variables leads to race conditions and other very hard to track down bugs. Attempts to introduce locks and atomic actions add complexity and the risk of deadlock. I strongly believe that destructive operations upon shared data should not be part of any language with a general model of concurrency. And non-general models of concurrency such as that of StarLogo and NetLogo are useful only in limited situations.

Furthermore, I believe we want to support children in building distributed computations. This need not add complexity to the system if the model of concurrency generalizes to processes running on different computers communicating over a network, as is the case in ToonTalk. Despite limitations of the partial implementation of distributed computing in ToonTalk, various networked games have been built such as *Battleships* and two-player *Pong*. Swiss high school students have implemented a chat system in ToonTalk.

Also the communication mechanism used by programs should be of use for human-to-human communication as well. In the Playground Project young children used ToonTalk's long-distance birds to exchange their ToonTalk games as well as text messages.

The concurrency of a programming language should have a solid theoretical foundation such as that offered by the pi calculus.

What is lost by making computational abstractions concrete as ToonTalk does?

Let's consider an example. Suppose we want a robot that can take a stream of incoming objects and produce two outgoing streams each containing every other element. Fig. 1 is a series of pictures of the training of such a robot.

Here's the equivalent robot in pseudo-code:

```
to Split In Out1 Out2
   if In receives X then
      send X on Out1 and
```

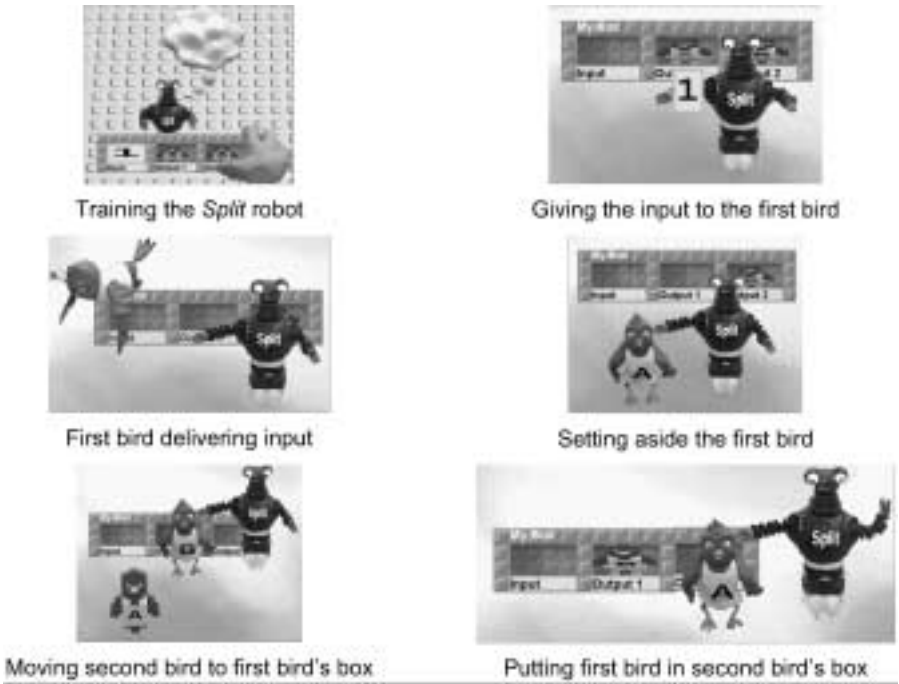

Fig. 1. A robot splitting a stream of numbers.

```
    Split In Out2 Out1 // notice the arguments are swapped
end
```

I believe there are many children and adults that understand how the *Split* robot works, especially after watching it in action, will find the equivalent *Split* procedure perplexing. And maybe an even smaller proportion of those who can program the *Split* robot could construct the *Split* procedure.

But have those who can program the *Split* procedure lost something by using ToonTalk instead? One thing they may have lost is the ability to see "at a glance" what the program does without running it. A series of pictures as in Fig. 1 helps but currently there are no tools for producing them automatically. While ToonTalk can generate a verbal description of a robot, it is often at too low a level of detail.

One answer to the question of what is lost is a formal representation of the program. This is only of value to those who are good at thinking using abstract formalisms. But what about those learners who aren't good at using abstract formalisms but would become so if they put in the effort to learn an abstract programming language?

Another answer is that the ideal language should support multiple representations that range from ToonTalk-like concretizations, to series of pictures, to symbolic representations. And the language should support the easy movement between these representations for the same program fragment. There are many challenges to designing a seamless multi-representation language without comprising any of the individual representations.

One Language or Many?

The community has been producing, and will continue to produce, many programming languages for children. Should we design a coordinated suite of languages and offer that to children or should we focus our resources on the "ideal" language? The argument for multiple languages is that each one may offer different strengths and ways of thinking about computation. The set of ideal languages for children may include those based upon different paradigms of programming as well as different levels of concreteness. Special purpose languages may have a role in such a basket of languages since they may provide superior support for certain classes of problems.

Multiple languages that are too similar are counter productive. The LOGO community has suffered from many incompatible dialects; (Boytchev, 2007) lists 173 LOGO dialects.

Ideally different languages should interoperate together well. Microsoft's Common Language Runtime may provide support for tight integration. In some cases it may even be possible for some languages to emit equivalent code for import into a different language.

If these languages can be tightly integrated then is it best to think of them instead as a single multi-paradigm, multi-layered language?

Some have argued for different languages for different age ranges of children. My experience with ToonTalk is that this is not necessary. Children, as young as 3, have built ToonTalk programs (Morgado, 2003), while university students have used it to explore

concurrent algorithms. By having a diverse user base, children of different levels of experience and expertise are more likely to help each other than if they are using different languages.

One danger of multiple languages is that they can obscure how apparently different things may be fundamentally similar. If, for example, a model of the spread of diseases and a model of the spread of technological innovations are built in the same language then the similarity of these processes should be easier to perceive.

Where should Program Construction Take Place?

This may seem like a strange question. Some people construct programs on paper and then enter them into a computer. Some people type their programs into their favourite text editor. Others use editors specialized for the programming language. Some programming languages support program construction from within the programming environment. A recent trend here is to use composable tiles representing program fragments as in Etoys (Allen-Conn and Rose, 2003) and Alice (Conway *et al.*, 2000).

ToonTalk is unique in that program construction takes place from within a game-like virtual world. It takes place, not by supporting the editing of textual or pictorial programs from within a virtual world, but instead by taking direct "everyday" actions in this world. The ToonTalk world is cute and playful and popular with pre-teens but it isn't a place where people spend much time for purposes other than constructing, debugging, and running ToonTalk programs. Some children spend many hours decorating houses, filling notebooks with artwork, playing with text-to-speech engines, and doing mathematical explorations with ToonTalk's exact rational arithmetic. But "being there" is rarely the main purpose of visiting the ToonTalk world.

How might persistent shared on-line virtual worlds fit into the big picture? *Second Life* is a nice example of such a place where the "residents" have built a great variety of places and things in this world (Ondrejka and Cook, 2005). *Croquet* has a similar vision (Smith *et al.*, 2003). About ten million people regularly visit such places. People visit these places primarily for entertainment and social reasons but there are more "serious" activities.

What if one could construct programs from within these worlds in a manner similar to ToonTalk programming? It could be done in a way that is quite similar to how ToonTalk currently works (including carefully designed concrete analogs to all computational abstractions) but with these differences:

1. **3D**. ToonTalk uses sprite animation and $2\frac{1}{2}$dimensional graphics to provide a virtual world. This simplifies but limits things. 3D enables the explorations of many topics in science and mathematics. And it enables the programming of very popular classes of games. It supports a wider set of contexts for activities. The challenge is to make a 3D world that can be navigated and programmed by all. Progress has been made in making 3D programming broadly accessible with AgentCubes (Repenning and Ioannidou, 2006), Alice (Conway *et al.*, 2000), Elica (Boytchev, 2003), and forthcoming versions of NetLogo and StarLogo.

2. **Realistic physics**. Many virtual worlds today have "physics engines" that support collisions, gravity, friction, rigid body motion (with joints), and more. It is unreasonable to expect such physics engines to be built by children. But they can be customised and parameterised by all. A programmable world with a built-in physics engine enables many explorations in sciences (e.g., sports science or mechanics). And it opens up a whole new class of games that children can create.
3. **Virtual communities**. Inhabitants of a shared on-line virtual world can provide help to each other in ways that are currently feasible only in face-to-face encounters. One can meet and build things together. Someone half way around the world can help others build and debug a program by being in the same place, manipulating the same objects, and conversing the whole time. People currently collaborate in ways that are more awkward using web sites, email, and the like. These communities can also have out-of-world support from associated "Web 2.0" sites.
4. **Living with your creations**. One is more motivated to build things that work in the place where one spends one's time. If people inhabit these spaces for reasons other than programming, then programming becomes a tool for enhancing their "living space". Their creations are things they can enjoy while they are in a virtual world for social or entertainment reasons, rather than objects that exist only when they are in some programming system. Examples include virtual pets, useful gadgets, kinetic sculptures, and long-lasting simulations.

Where does Turtle Geometry Fit into the Picture?

Despite my love of turtle geometry, it is not part of ToonTalk. In 1995, I tried to add it and began user testing. The idea was that there was a bird for each picture. If you gave the bird a box containing the text pad *FORWARD* and the number pad *100* then the bird would deliver the message to the picture and it would move forward 100 units in the direction of its current heading. Non-programming adults and a fourth-grade class seemed to understand the concepts of ToonTalk (despite its primitive state) but were unhappy with this message passing interface. It was too indirect and clumsy.

I redesigned ToonTalk to include remote controls to support picture programming in a more direct intuitive manner. For example, a sensor for the horizontal position of a picture appears just as numbers do (except it has an animated marquee to indicate its special status). One can manipulate this remote control as an ordinary number and the associated picture's position changes accordingly. This also works in the other direction: if the picture is moved, the number is updated. This scheme is similar to the property sheets used in a huge variety of software where the properties show the current state of an object and can be edited to change the object. In ToonTalk the equivalent of a property sheet is broken up into small pieces corresponding to single properties. This provides a nice "declarative" interface for objects. Children as young as six in the Playground Project made heavy use of these ToonTalk remote controls (Hoyles, 2002). But these remote controls only support Cartesian geometry and don't support turtle geometry.

So why not add *FORWARD* and *RIGHT* sensors to ToonTalk? The equivalent of a *RIGHT* sensor would be straightforward; just add a sensor for the heading of a picture. The equivalent of *RIGHT 90* would be to drop a number pad with *90* on the heading sensor and the current heading will be incremented by 90 degrees. But what sensor would play the role of *FORWARD*? This stumped me for many years. The best idea I have is to introduce a sensor for the picture's distance to where you want it to be (in the direction of its current heading). So if you dropped *100* on such a sensor the picture would move forward 100 steps. The really odd thing is that this sensor would always display 0 since as soon as it is changed the picture moves and it is where you instructed it to be. Maybe a better version would be a sensor for the distance to its "goal". If its speed is infinite then it will also always display 0 but if it is given a finite speed it will glide towards its goal and the value displayed will decrement at the speed until it reaches zero. Some other design issues remain such as how to integrate the trails left by turtles' pens. Are they new objects or are they alterations of the surface they are drawing on?

I now believe that rather than choosing between message passing and declarative user interfaces (property sheets/remote controls) that they should coexist at different levels. The primitive low-level way of manipulating pictures could be via message passing as I originally intended a dozen years ago. This will probably be used only by more advanced users, particularly those enhancing the system itself. The ideal system should provide primitive support so that remote controls can be built within the system. They would behave much as they currently do in ToonTalk but their implementation would be transparent. Ordinary users could inspect, edit, and create new kinds. The challenge here isn't just to implement sensors within the language but to do so in a manner that non-experts can understand. This is one of the challenges the VPRI Project led by Alan Kay is addressing.

More generally, I think a good way to proceed is to open the development of new "primitives" to a wide community. The kernel of the system can be small but include a way to "plug in" modules written in nearly any programming language. This could be based upon an improved version of ToonTalk's foreign birds that provides a way to use birds to interface external code. The bird or message passing level of interface can then be built upon to provide higher-level functionality if need be.

Conclusions

In the 40 years since LOGO was born there has been good incremental progress. Computer science has also made substantial progress in models of computation and user interfaces. Building upon this progress and drawing upon our experiences with LOGO, Smalltalk, Boxer, AgentSheets, StageCast Creator, ToonTalk, Alice, StarLogo, NetLogo, and Scratch we should be discussing where to go next. We should jump to promising regions of the design space rather than making slow and steady turtle-like progress. Rather than *REPEAT n [FORWARD 1]* we need to determine a good (n-dimensional) *heading* and go *[RIGHT heading FORWARD 1000]* to the next generation of programming languages for children of all ages.

References

Allen-Conn, B.J., and K. Rose (2003). *Powerful Ideas in the Classroom Using Squeak to Enhance Math and Science Learning*. Viewpoints Research Institute.

Boehm, H.-J. (2004). *The Constructive Reals as a Java Library*. HP Laboratories Palo Alto, HPL-2004-70. `http://www.hpl.hp.com/techreports/2004/HPL-2004-70.pdf`

Boytchev, P. (2003). Turtle metamorphoses (From FD 1 to 3D Animated Characters). In *9th European Logo Conference EuroLogo 2003*, Porto, Portugal.

Boytchev, P. (2007). *LOGO Tree Project*. `http://www.elica.net/download/papers/LogoTreeProject.pdf`

Conway, M., S. Audia, T. Burnette, D. Cosgrove and K. Christiansen (2000). Alice: lessons learned from building a 3D system for novices. In *SIGCHI Conference on Human Factors in Computing Systems*, The Hague, The Netherlands, April 2000. ACM Press, New York, pp. 486–493.

diSessa, A.A. (1997). Twenty reasons why you should use Boxer (instead of Logo). In M. Turcsányi-Szabó (Ed.), *Learning & Exploring with Logo*: *Proceedings of the Sixth European Logo Conference*, Budapest, Hungary, pp. 7–27.

Hoyles, C., R. Noss and R. Adamson (2002). Rethinking the Microworld Idea. *Journal of Educational Computing Research*, **27**(1–2), 29–53.

Kahn, K. (1996). ToonTalk – an animated programming environment for children. *Journal of Visual Languages and Computing*, June.

Kahn, K. (2001a). Generalizing by removing detail: how any program can be created by working with examples. In H. Lieberman (Ed.), *Your Wish Is My Command*: *Programming By Example*. Morgan Kaufmann.

Kahn, K. (2001b) ToonTalk and Logo – is ToonTalk a colleague, competitor, successor, sibling, or child of Logo? In *EuroLogo Conference*, August 2001.

Kahn, K. (2004). *The Child-Engineering of Arithmetic in ToonTalk, Interaction Design and Children Conference*. College Park, Maryland.

Kahn, K., E. Sendova, A.I. Sacristan and R. Noss (2005). Making infinity concrete by programming never-ending processes. In *7th International Conference on Technology and Mathematics Teaching*, Bristol, UK.

Kahn, K., R. Noss, C. Hoyles and D. Jones (2006). Designing digital technologies for layered learning. *Informatics Education – The Bridge between Using and Understanding Computers*, *Lecture Notes in Computer Science*. Springer Berlin/Heidelberg.

Kahn, K. (2006). Time travelling animated program executions. In *Software Visualisation Conference*, Brighton, UK.

Kahn, K. (2007). Comparing multi-agent models composed from micro-behaviours. In *Third International Model-to-Model Workshop*, Marseille, France, March 2007.

Kay, A. (1984). *Computer Software*. Scientific American, September 1984.

Kay, A. (1993). *The Early History of Smalltalk. History of Programming Languages*, II, ACM.

Kay, A., D. Ingalls, Y. Ohshima, I. Piumarta and A. Raab (2006). *Steps Toward The Reinvention of Programming a Compact and Practical Model of Personal Computing as a Self-Exploratorium*, VPRI Research Note RN-2006-002. `http://www.viewpointsresearch.org/pdf/NSFproposal.pdf`

Kindborg, M. (2001). Representing ToonTalk programs as comic strips. In *Playground International Seminar*, Porto, Portugal.

Morgado, L., M.G. Cruz and K. Kahn (2003). Taking programming into Kindergartens. In *EuroLogo Proceedings*, Portugal, August 2003.

Ondrejka, C., and J. Cook (2005). Brace for Impact: how user creation changes everything. In *Games, Learning and Society Conference*, Madison, Wisconsin.

Repenning, A., A. Ioannidou and J. Zola (2000). AgentSheets: end-user programmable simulations. *Journal of Artificial Societies and Social Simulation*, **3**(3).

Repenning, A., and A. Ioannidou (2006). AgentCubes: raising the ceiling of end-user development in education through incremental 3D. In *IEEE Symposium on Visual Languages and Human-Centric Computing*.

Papert, S. (1977). *MIT Logo Project Meeting Notes*.

Papert, S. (1980). *Mindstorms*. Basic Books, New York.

Saraswat, V.A., K. Kahn and J. Levy (1990). Janus: a step towards distributed constraint programming. In *North American Conference on Logic Programming*. MIT Press, Cambridge.

Saraswat, V. (1993). Concurrent constraint programming languages. Doctoral Dissertation Award and Logic Programming Series. The MIT Press.
Smith, D.C., A. Cypher and L. Tesler (2000). Novice programming comes of age. *Communications of the ACM*, **43**(3), 75–81.
Smith, D., A. Kay, A. Raab and D. Reed (2003). Croquet – a collaboration system architecture. In *First Conference on Creating, Connecting and Collaborating through Computing*.

K. Kahn has been engaged in research in computer programming since before he received his doctorate from MIT in 1979. After a short period exploring programming languages for children he turned towards the design and implementation of very high-level programming languages embodying ideas from object-oriented programming, logic programming, constraint programming, concurrent programming, distributed computing, and visual programming. In 1992, Ken returned to programming languages for children when he founded Animated Programs whose mission is to make computer programming child's play. He designed and built ToonTalk, an animated programming language for children. He was a researcher in two large-scale European research projects that built upon ToonTalk. He is currently a senior researcher at Oxford University and a visiting fellow and researcher at the London Knowledge Lab.

Ar turėtų LOGO ir toliau judėti PRIEKIN 1?

Ken KAHN

Jau 40 metų sparčiai plėtojama Logo programavimo kalba. Šiame straipsnyje iškeliami svarūs probleminiai klausimai, susiję su Logo programavimo kalbos projektavimu: programos stipriosios ir silpnosios vietos, sąsajos su kitomis programomis, Logo projektavimo įtaka informatikos plėtrai ir pan. Straipsnio tikslas – paskatinti pasvarstyti iškeltas problemas, parodyti išsamų Logo raidos mokslinį kelią. Autorius, turėdamas 30 metų patirtį šioje srityje, gvildena šias problemas ir pateikia pagrįstus samprotavimus.