Real-Time Programming and the Big Ideas of Computational Literacy

by

Christopher Michael Hancock

A.B., Mathematics
Harvard College, 1983

Ed.M.
Harvard Graduate School of Education, 1987

Submitted to the Program in Media Arts and Sciences,
School of Architecture and Planning,
in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy in Media Arts and Sciences

at the

Massachusetts Institute of Technology

September 2003

Signature of Author ______________________________
MIT Program in Media Arts and Sciences
August 19, 2003

Certified by ______________________________
Mitchel Resnick
LEGO Papert Associate Professor of Learning Research
Thesis Supervisor

Accepted by ______________________________
Andrew B. Lippman
Chair, Departmental Committee on Graduate Students
Program in Media Arts and Sciences

Real-Time Programming and the Big Ideas of Computational Literacy

by

Christopher Michael Hancock

Submitted to the Program in Media Arts and Sciences,
School of Architecture and Planning, on August 19, 2003
in Partial Fulfillment of the Requirements for the Degree of
Doctor of Philosophy in Media Arts and Sciences

ABSTRACT

Though notoriously difficult, real-time programming offers children a rich new set of applications, and the opportunity to engage bodily knowledge and experience more centrally in intellectual enterprises. Moreover, the seemingly specialized problems of real-time programming can be seen as keys to longstanding difficulties of programming in general.

I report on a critical design inquiry into the nature and potential of real-time programming by children. A cyclical process of design, prototyping and testing of computational environments has led to two design innovations:

- a language in which declarative and procedural descriptions of computation are given equal status, and can subsume each other to arbitrary levels of nesting.

- a "live text" environment, in which real-time display of, and intervention in, program execution are accomplished within the program text itself.

Based on children's use of these tools, as well as comparative evidence from other media and domains, I argue that the coordination of discrete and continuous process should be considered a central Big Idea in programming and beyond. In addition, I offer the theoretical notion of the "steady frame" as a way to clarify the user interface requirements of real-time programming, and also to understand the role of programming in learning to construct dynamic models, theories, and representations. Implications for the role of programming in education and for the future of computational literacy are discussed.

Thesis Supervisor: Mitchel Resnick
Title: LEGO Papert Associate Professor of Learning Research

Real-Time Programming and the Big Ideas of Computational Literacy

by

Christopher Michael Hancock

Certified by ____________________

Henry Lieberman
Research Scientist
MIT Media Laboratory

Certified by ____________________

Andrea A. diSessa
Chancellor's Professor of Education
University of California at Berkeley

Acknowledgements

Mitchel Resnick, my advisor and teacher, supported this work generously through its many twists and turns. It has been a privilege to learn from him and to be a member of his working group.

Andy diSessa has built a multifaceted body of work that I very much admire, and that has informed my work at almost every turn. I thank him for this, and for his confidence in and encouragement of my work.

Henry Lieberman taught me to focus on the essentials.

Three friends in particular helped to make this project happen:

Cynthia Solomon, my collaborator and muse, encouraged me all the way and repeatedly made sure I didn't hide from kids too long.

Robert Kwok's dependable handling of camera work, logistics, and countless other details ensured that this project finished on time. Robert also conceived and built the seesaw apparatus. Thank you Robert!

Bakhtiar Mikhak helped me to find my way into this work, and supported it in many different ways, of which the most important has been his friendship.

My gratitude also to all of these:

the participating children and their parents

the undergraduate assistants who have helped over the years: Warren Chia, Sam Davies, Sandra Galdamez, Dan Kim, Hana Kim, Adam Mullett.

Tim Gorton, Chris Lyon, and the Media Lab's Grassroots Invention group for their fine Tower technology and gracious technical support.

past and present members of the Lifelong Kindergarten group, especially Andy Begel and Brian Silverman, and all my fellow constructionists at the Media Lab.

John and Kenny and the whole gang at the pool.

…and many others!

Thanks to Eleanor Toren, Jerry Heath, Dr. Kumsan Song, Dr. Joseph Song, Donia Goss, Solana Goss, and Juciane Asunciao, who have all given so generously on the home front.

Thanks to Peter Hancock, with whom I have shared a love of language and thinking for 42 years.

And to Patricia Song, Ph.D. (2003): We did it!

Contents

Chapter 1: Introduction

PROLOGUE

Nick, aged 15, and Tor, 14, are programming a robotic seesaw. Along the length of the seesaw runs a track for a ball. The challenge: program the seesaw to roll the ball back and forth along the track, without letting it run off either end. The robot is connected to a computer by a cable. It has a motor to tilt the track this way and that, and I have just helped them mount a distance sensor at one end to detect the position of the ball.

Once the distance sensor, which is color-coded white, has been mounted, the boys' first order of business[1] is to check its readings. At the computer, they write

```
        P: white_distance()
```

Immediately, this line of code begins showing the continually updated current value of the variable P (for position) that they have created. They slide the ball up and down the length of the seesaw, checking that the sensor continues to "see" and respond to the ball. It does.

When encountering this challenge for the first time, learners often try programming the seesaw to raise the left side when the ball is on the left, and raise the right side when the ball is on the right. Nick and Tor tried this a half hour ago, and they have since decided that almost as soon as the ball gets going in one direction, they need to start adjusting the tilt against it, or it will fly off the end. They need to attend to the ball's direction rather than its position. So Nick adds a second line to their program, which now looks like this:

```
    P: {47} white_distance()
    SPEED: {-4} delay(P,20) - P
```

The new line computes, at each moment, the distance (negative distance, but this doesn't matter) the ball has traveled in the last 2 seconds. They move their hands up and down the length of the track. It works: When they move their hands slowly to the left, the flickering numbers for SPEED are negative, and when they move their hands to the right, they get positive numbers. Now they can tell the ball's direction. They are ready for their first attempted solution.

```
    P: {47} white_distance()
    SPEED: {-4} delay(P,20) – P
    WHEN SPEED < 0
        DRIGHT()
    WHEN SPEED > 0
        DLEFT()
```

(… where DRIGHT—"down right"—and DLEFT are simple subprocesses they have defined to turn the motor one way or the other.)

They turn these lines on, and the seesaw leaps into action, without waiting for the ball to appear on the track. The seesaw jerks this way and that spasmodically. The story on the screen makes the reason clear. As sensor readings fluctuate, SPEED is virtually never zero, but rather a small negative or positive number. Consequently, at any

[1] This vignette is excerpted and distilled from a longer story. Prior to this episode, the boys had worked for a while with a different sensor arrangment, a series of light sensors embedded under the track. Some code from this earlier phase remained (inert) in their program for a while. For clarity of exposition, I am omitting some twists and turns in the boys' process.

moment either DRIGHT() or DLEFT() is lit up, and the device can never rest. Nick changes the zeroes to –8 and 8, respectively, and the apparatus calms down.

Now they can test the seesaw's response to the ball. They try this in various ways, sometimes letting the ball roll freely, other times nudging it one way or another, and sometimes holding it in their hands, waving it up and down the length of the track. The robot's response is confusing. Sometimes it seems to be responding to the motion of the ball, other times it seems not to be paying attention. When they roll the ball from one end to the other, the robot waits much too long, finally starting to change the tilt just as the ball is approaching the end. Nick has an idea to account for the sluggishness. He changes the time delay to 10 so that SPEED now measures change over the last second, instead of the last two seconds. This improves the response somewhat. Now, when the ball is started from one end of the ramp, the robot succeeds occasionally in reversing its direction once, and sometimes even attempts to reverse it again, before losing control. They are making progress.

```
P: {19} white_distance()
SPEED: {11} delay(P,10) – P
WHEN SPEED < -8
    DRIGHT()
WHEN SPEED > 8
    DLEFT()
```

Nick and Tor still have a long way to go to meet this difficult challenge. Changing a two-second delay to a one-second delay was a good step, but the boys will need to continue inching closer to a concept of instantaneous speed in order to get adequate responsiveness from their robot. The boys have not yet come to terms with seesaw calibration, so their program has no way to find a level position. Gears grind noisily when the motor attempts to push past the limit on either side. And to refine their control of the ball they will need to get deeper into the relationships among tilt, acceleration and deceleration. Beyond that, to avoid recurring runaway, they made need to think about making the apparatus anticipate as well as respond to the ball's motion.

But we have perhaps seen enough of the story to make a few observations that help reveal where this dissertation is headed.

First, the seesaw setup is plainly a rich learning environment, and shows potential to be even richer. The apparatus is fun, and the ball-control challenge is compelling. The boys are tackling deep questions about speed and acceleration, in a context where gradual mastery of these ideas is immediately empowering.

Second: the boys are engaging these ideas through programming. Programming a robot creates a context where understanding can be translated into power.

Third: the program is, in part, a model. Just how this kind of model compares to other models we will have to work out, but most importantly, it is a "live" model. It is built to reflect aspects of that situation as the situation is unfolding.

Fourth: the model is partly expressed in terms of dynamic variables, which contribute both intelligibility and power to the boys' evolving solution. These variables create a logically and perceptually stable representation (a "steady frame," as I will call it) for monitoring , interpreting, and responding to a changing situation. Speed is a particularly well-known dynamic variable with a well-established place in our culture and in school curriculum. Later we will see examples of much less celebrated variables, some of them categorical rather than numerical, some of then invented on the spot, playing equally significant roles in student projects, and helping to illuminate the essential strategies we use, in programs and in our discourse and understanding, to master change by containing it.

Finally, we must make note of the specific programming language, Flogo II, which helps to make this project possible. That Flogo II does not figure prominently in the story as I have just told it, is, paradoxically, one indication of just how important its design is to the richness of this learning environment. Any other language would have had a larger role in the drama, and some would have stolen the show entirely. Among the essential ingredients in Flogo II's comparative transparency is that it runs "live." The code displays its own state as it runs. New pieces of code join into the computation as soon as they are added. In the story so far, we can see the value of liveness in quickly resolving questions, such as: is the sensor hooked up correctly? Is it registering the ball all along the track? What range of values is it reporting? Which direction corresponds to positive speed, and which to negative? Why is the first program producing such jittery motion?

Live programming allows a much more open and fluid relationship among the program, the robot and its environment, and the programmer. Taking up the idea of another boy who also worked with the seesaw, I will later encourage Nick and Tor to write two more lines:

```
WHEN LEFT_ARROW() DLEFT()
WHEN RIGHT_ARROW() DRIGHT()
```

This is all it takes to allow Nick and Tor to drive the robot motor directly using the arrow keys on the computer keyboard. Reflecting on their own strategies to control the ball will help them generate new ideas and insights for a programmable solution.

The time is ripe for a renewed and expanded epistemological investigation of the intellectual substance of programming, and its potential educational values. But our view of programming as an intellectual endeavor is filtered through the available programming languages and the kinds of activities they support, and these remain deficient in some very important respects. The goal of Flogo II is to change the nature of computer programming in ways that will help us take the epistemological investigation farther. The story of the design research that led to Flogo II will occupy many of the pages that follow. The generative liveness and other properties achieved by Flogo II could not have been grafted onto an existing language, but required a thorough reworking of the computational paradigms appropriate for learners. Children's use of Flogo II (and its predecessor, Flogo I), takes up the remainder, as I report on what these prototype languages have so far helped to reveal about some essential ideas and broader values of programming—as it has been, and as it may soon become.

CONTEXT: THE PLACE OF PROGRAMMING IN EDUCATION

Debates of the 1980's about the essence of programming, the challenges in learning it, and its optimal place in education have fallen into relative quiescence—lapsed, one might say, into a fatigued polarization. In the minds of many educators, perhaps representing the mainstream, the question is resolved: programming is a specialized skill, too difficult for most kids, and it is not necessary for realizing the educational potential of digital technologies. At the same time, a small community of researchers, teachers, and other educators persists in pursuing and studying learning opportunities in programming that seem, to them, both profound and rare—profound because of the depth and power of some concepts involved in programming, and rare because of the opportunity to empower children's roles as investigators, creators and designers in relation to almost any educational domain.

Communication between the two points of view has been low, but there are signs that the time may be ripening for a renewed dialog. Research yielding qualified support for the benefits of programming, along with more specific lessons and insights, continues to accumulate (Clements, 1999; diSessa, 2000; Noss&Hoyles, 1996). The arguments in favor of programming have grown in sophistication, and some of the old arguments have found receptive new audiences and compelling new instantiations. Programming has been shown to offer children a new intellectual landscape, with inviting routes into geometry (Papert, 1980), algebra (Hoyles&Noss, 1996), and physics (Sherin, 2001), and access to frontier areas of robotics (Resnick, 1993; Miller & Stein, 2000) and complex systems modeling (Resnick, 1994; Wilensky&Stroup, 2000). These demonstrations have helped to displace the old straw-man argument for programming, viz.: that it might bring about some nebulous, generalized improvement in children's problem-solving abilities. A considerably more specific case can now be made that programming provides an excellent immersion for the skills and practices of engineering; that it makes mathematics applied, concrete, and empowering; that the core concepts of programming belong in any meaningful definition of computational literacy; and that programming provides a rich medium for creative expression and for cultivating the skills of design; and that as a general model-building and toolmaking tool it supports constructionist learning across the curriculum (e.g. Papert, 1980; Kafai, 1995; Kay, 1991; diSessa, 2000).

The past decade has brought new programming languages and environments that extend beyond the terms of the old debates. Newer general-purpose programming languages, beginning with Boxer (diSessa, 2000), and now including ToonTalk (Kahn, 1996) and Smalltalk's reincarnation Squeak (www.squeakland.org) are radically different from the Logo and Basic of old, and even Logo and Basic have changed in significant ways. There are also new programming languages for more specialized purposes, such as the modeling languages StarLogo (Resnick, 1994; Wilensky, 1995), NetLogo (Wilensky & Stroup, 2000), Agentsheets (Repenning, 2000), and KidSim (ref), the introductory geometry construction language Turtle Math (Clements, 1999), and the virtual reality entertainment authoring language Alice (Cooper, Dann, & Pausch, 2000). In small ways and large, all of these developments reframe the programming process and enable new learning trajectories which remain largely unexamined. The envelope has also been pushed by new contexts of use. Children have been programming for new kinds of purposes, including robotics (Martin, 1996), the design of games and tutorials (Harel & Soloway, 1991; Kafai, 1995; Hoyles & Noss, 1999), the construction of virtual worlds and communities (Bruckman, 1998), and the creation of virtual reality entertainment (Cooper, Dann, & Pausch, 2000). Meanwhile, many other new environments have collectively smudged the boundaries of the category "programming." These include commercial construction and puzzle games such as The Incredible Machine (Sierra), Widget Workshop (Maxis), Yoiks! (TERC), and MindRover(Cognitoy).

Many of these boundary-blurring tools can be found in schools: for example, geometry tools, (Cabri, Geometer's Sketchpad), data analysis tools (Fathom), simulation environments like Model-It (Jackson et al., 1999), spreadsheets, and web and multimedia authoring tools. In fact, it is increasingly clear that many of the most interesting educational tools, old and new, gain much by being programmable. Meanwhile, research and development involving programming continues quietly, under more acceptable banners such as "scripting" and "modeling." In short, while programming may have been removed from the mainstream educational agenda, it has proven difficult to banish it altogether. Will it continue to lurk on the fringe, and under the hood? Or do these developments suggest a potential return to educational legitimacy?

Perhaps the broadest articulated vision for a new role for programming comes from diSessa (2000), who envisions an increasing amount of discourse, within education and beyond, taking place within a “computational medium” with scope and effects fully comparable to those of paper-based media over the last several hundred years. The most important difference between diSessa’s computational medium and the computer-based media in wide use today is programmability at the user level. On the social and individual level, investment in computational literacy will yield rich returns. Kay has captured a similar idea in his characterization of computers as a “metamedium” (Kay, 1991).

However far off realization of these visions seem, socially and politically, it is in intention of the current work to take them seriously, and to inquire into the technologies and understandings that will help to increase their feasibility.

Advocates of programming have unfinished work to do

I count myself firmly among the programming partisans. I find many of the arguments persuasive and examples compelling. Programming in its current forms has, I am convinced, a tremendous amount to offer young people, given the right conditions. But I believe that a renewed dialog with the mainstream will require a stronger case than now exists. Most importantly, the continuing problem of the difficulty of programming, particularly in contexts of interest to children, has not yet been adequately dealt with. In addition, the need has not been fully met for a clear, comprehensive, and coherent account of the intellectual substance of programming, and its relation to major educational and cultural values.

The failure of programming to take hold in most schools reflects a complex web of factors. Bringing anything into school that is perceived as a large addition to the curriculum is a monumental task. Moreover, the full educational implications of programming run strongly counter to a long standing (and recently resurging) tendency, deeply embedded in school culture, to view education in terms of topics, facts and skills. Nonetheless, undeniably intertwined with such factors is the simple fact that programming has been hard to learn, for both students and teachers (duBoulay, 1989; Pea, Soloway, & Spohrer, 1987; Perkins&Martin, 1986).

The problem of programming’s difficulty has, if anything, become thornier in recent years. The kinds of programming tasks that were found to be difficult in the 1980's--averaging or sorting lists of numbers, producing drawings on the screen, interacting with users via text prompts and typed responses—are still difficult, though less so. But the world has moved on to very different kinds of programming challenges. Many children (boys, especially) want, even yearn, to make video games. Yet with the available tools, only the most persistent can produce a working game, and a very crude one at that. This is a shame, because a good educational opportunity is being missed.

If the difficulty of video game programming is a shame, the difficulty of robotics programming is, in my opinion, a major embarrassment to the cause of educational programming. Across the country, in afterschools and many other extracurricular settings, we find not only throngs of interested children, eager to invest hours of effort in programming a robot or other device to behave autonomously, but also phalanxes of smart and committed volunteers and educators working to help them. Yet once again, as I will detail in Chapter 3, the results (vis-à-vis programming, specifically) are meager and a major educational opportunity is being missed. Why?

What video game and robot programs have in common is that they interact with an independent environment that proceeds according to its own schedule. They are *real-*

time programs, meaning they must respond to events as they occur, and not just when the program asks for them. Real-time programming is hard for experts as well as novices. The nub of the problem is process coordination: real-time problems, by definition, require that multiple, semi-independent processes inside and outside the program be able to share information and influence each other's actions.[2] A mature, cognitively congenial set of tools for creating such coordination does not yet exist.

A solution to this problem is urgently needed, not only to capitalize on the educational opportunity in children's desire to program robots and video games, but also because of the wider implications of such tools for the expressivity of a universal computational medium. With better real-time programming in the hands of children, teachers, and educational developers, there is reason to hope for a flowering in the development of all kinds of dynamic tools, representations, and construction and inquiry environments ("microworlds"—Papert, 1980) at a new level of richness, dynamism and immediacy. To understand why this might be so, consider that, in terms of the programming requirements, an animated microworld is just a kind of video game, and a digitally enhanced laboratory apparatus—a controllable seesaw, for example—is just a kind of robot.

"Misconceptions Unite!"

I mentioned two gaps in the case for programming. Difficulty was the first; the second is the need for a coherent account of programming as an intellectual endeavor. Here we have something far less than a scandal, but we do face a challenge. The same new developments that help to earn programming also create something of an identity crisis for programming. With the greatly increased diversity of tools and activities to which the label might arguably be applied (and the still greater diversity of forms likely yet to be created), one can ask whether programming is likely to remain a useful, coherent category at all, distinct from the various themes and concepts that appear among subsets of these disparate environments.

"Scripting" is a term that has gained currency in recent years. Educational software with some programmability is described as having "scripting capability." In part, a meaningful distinction is being made: a scripting language promises greater simplicity in return for reduced expressive power. But there is a rhetorical agenda as well: to reassure the listener that no dangerously substantial ideas lurk within. The idea is afoot that students will have no need for the specialized skills of programming, because they can just "script." This is thin gruel for growing minds. With no sense of a growing competence and power to which students' tool-related programming experiences, large and small, might synergistically contribute, the scriptable tool scenario promises a balkanized educational technology, where students' ability to "go beyond the representation given"[3] does not get off the ground[4]. The much preferable alternative, argued in detail by diSessa (2000), would include educational tools in, or interface them to, a unifying "computational medium." Every episode of scripting would then also be an investment in what

[2] I say "by definition" because even in the minimal case, a single-threaded program must coordinate with the independent process constituted by external events and inputs.

[3] Paraphrasing Bruner (1973)

[4] The component architecture strategy that some have advocated (e.g. Roschelle, Kaput, Stroup, & Kahn, 1998; see also Kynigos et al., 1997) is an inadequate solution to this problem, I believe. This is not the place for a full discussion, but the core problem with the component model is that tool interconnections are fixed in advance by the developer. For full flexibility of tool interconnections, you need a programming language. In fact, you need a real-time programming language.

diSessa rightly calls a new computational literacy, and I am here calling programming.

Some of the former challenges of programming—syntax, for instance—are gradually yielding to improved language designs and user interfaces. Will everything that students of programming once struggled to learn turn out to have been an artifact of immature technology? Surely not. One is not going too far out on a limb to predict that topics such as data structures and Boolean logic, and concepts such as variable, argument (or parameter), algorithm, and even loop, will have an enduring role. But we need an understanding of these and many other issues that transcends individual languages.

For example, a well-known misconception among students learning about looping is the expectation that iteration can end in mid-loop, the instant a termination condition becomes true. And yet in this document I will present a language with looping constructs that work in just this way, are accepted as perfectly reasonable by inexperienced programmers, and are used by them to build effective robot behaviors. Does this mean that the old "misconception" can now be discarded as another artifact of the arbitrary requirements of an immature medium? Actually, in this case and many others, I think something much more interesting happens. What used to appear as a relatively isolated misconception is transformed into a part of a more general learning challenge with clearer connections to the rest of the world. In this case, for example, children still need (eventually) to master the differences between the two kinds of looping and the appropriate uses of each , along with parallel distinctions throughout the language. But in the meantime, they are free to use a construct that matches their intuitions and meets their needs. The unifying “big idea,” which will be elaborated in later chapters, concerns the different ways that processes, actions, and events can interact.

To build the case for programming as a new literacy, then, we need a more detailed account of the conceptual challenges of programming, and their connections to broader educational goals. This epistemological enterprise is in part an empirical undertaking, requiring comparative studies of learning issues across different kinds of programming environments—something Clements (1999) has called a “gaping hole” in the literature. But it is also a design challenge, a quest for an educationally optimal delineation and mapping of programming as an intellectual endeavor. Both research and design naturally proceed in dialectic with work on new programming environments themselves.

One might ask whether dialog with mainstream educators and the institution of school is an advisable or necessary way to bring the benefits of programming to children. Writers have considered whether the culture of School is intrinsically incompatible with programming and the constructionist ethos associated with it (Hoyles,1993, Papert, 2002), as it has seemed to be in the case of so many educational "reforms" (Tyack & Cyban, 1996). The institution of School as we know it is under attack from many quarters, and some have prophesied its doom. Developers of children's programming, meanwhile, focus most of their efforts on informal education contexts such as museums, community technology centers, home schoolers, robotics leagues, and home use of commercial products. These venues are important in their own right, and may also be the thin edge of a wedge back into school. But if one takes seriously the notion of computational literacy growing from a critical mass and variety of programming opportunities, then the time that children spend in school needs to figure centrally in the plan. There is much at stake in the educational frameworks and technologies now taking shape. Programming must be on the table, and it must be the very best "programming" we can muster.

RESEARCH GOALS AND METHOD

The tone of the foregoing has been, I hope, no more polemical than necessary to motivate and contextualize the goals of the present work. This dissertation seeks to amplify the values of children's programming, and further the epistemological enterprise around programming, by asking:

Q1: How can language design change the nature of real-time programming in order to make it more accessible and rewarding for learners?

This design question is the major focus of the thesis. Note that the question is posed so as to require not only a good design but also a critical account of the principles underlying its successes and failures. If we can succeed, to some degree, in deepening learners' engagement with real-time programming, we are then in a better position to investigate:

Q2: What does learners' engagement in real-time programming begin to reveal about the intellectual substance of real-time programming, and of programming in general?

A full investigation of this second question is beyond the scope of this project. The treatment here will be exploratory. Within the project as a whole, this question provides one additional test, on a "proof of concept" level, of the language design and the principles I attach to it, and of my claims about the educational importance of real-time programming.

Research Method

> The function of criticism is the reëducation of perception of works of art; it is an auxiliary in the process, a difficult process, of learning to see and hear. The conception that its business is to appraise, to judge in the legal and moral sense, arrests the perception of those who are influenced by the criticism that assumes this task.
> (Dewey, 1958, p.324)

The work reported here is conceived as a *critical design inquiry*. It is increasingly remarked that educational research is properly concerned not only with how things are, but how they might be (e.g. Collins, 1999). Growing interest among educators in the relationship between design and educational research and development is reflected in the increasing volume of papers and projects addressing it (e.g. Gorard, 2002; Hoadley, 2002; Kelly&Lesh, 2002, JLS special issue). Much of this discussion builds on seminal papers by Collins (1992) and Brown (1992), introducing the notion of a "design experiment," in which research on learning proceeds in dialectic with efforts to fashion and test learning materials and learning environments. The present project does combine design with attention to student learning. However, a typical design experiment concludes with a design evaluation based on some measurement of student learning. The present study addresses a question that is logically prior to such measurement, namely: what kind of learning should we be looking for, and why? I will be developing some recommendations about what ideas are worth children's time. This is not the sort of question that can be resolved by an experiment in the commonly understood sense of the term[5]. To what test, then, can my designs and recommendations be put? The appropriate paradigm is criticism.

Papert's (1987) has called for a new genre of literature to be called "software criticism." Here the model is the work of art criticism—criticism not in the sense of

[5] Experimental data can, of course, be relevant in judging an epistemological recommendation, but only within a broader web of considerations.

disparagement, but in the sense of aiming "to understand, explicate, to place in perspective." In comparison to experiments designed to isolate the effects of a single treatment variable, Papert argued, computer criticism is far more relevant to the construction of a radically different educational future. Since then, few people seem to have accepted Papert's invitation[6]. However, as the complement to an ambitious exploratory design project such as this one, I find Papert's paradigm more helpful for coming to terms with the multiple influences on how a designer poses problems, generates possibilities, and makes choices. The design experiment model seems better suited to a "normal science" phase (Kuhn, 1970) of design, where the design problem has become more circumscribed. Because this project includes both design and a critical assessment of the design and its context, I call this undertaking a critical design inquiry.

Papert's paper is a brief work of advocacy and repartée, not a detailed manual. However, Eisner (1991) develops a strongly resonant model of qualitative research, also based on the paradigm of art criticism (especially as analyzed by Dewey), which helps one to fill in much of the missing detail and framework. I have discussed elsewhere (Hancock, 2001b; 2002) some of the rationale and characteristics of a more elaborated software criticism. The essentials are easily summarized in terms of software criticism's work, function, and validity criteria.

The work of software criticism. In Eisner's rendering, criticism combines four kinds of work: description, interpretation, evaluation, and thematics. I have already ventured a description and interpretation of the current place of programming in education. In addition, I will be offering descriptions of the programming challenges faced by children, and the design problems faced by language developers, interpretations of these in terms of conceptual, epistemological and technical issues, and evaluative discussion of several programming languages, including the ones I am presenting. I will also develop two related themes, "steps versus processes" and "the steady frame," that connect many of the project's concerns.

The functions of software criticism are to…

- clarify context and values
- develop new categories and finer distinctions
- reveal unsuspected dimensions of choice
- pursue generative insight and practical understanding
- pick out revealing details and telling examples

Through these sorts of functions, it is my hope that this thesis will be helpful in opening up new possibilities for what programming might become, and identifying the implications of these possibilities for the future of education.

A work of computer criticism gains *validity* by means of the following tests (my four-part answer draws heavily on a similar three-part answer by Eisner).

Test #1: Illumination

We noted earlier that criticism aims to educate the perception of its readers. Whether it has succeeded in doing this is something that a reader can judge for herself. Does the reader find that she can now discern patterns, connections, or dimensions of the phenomenon better than before? Eisner calls this criterion "referential adequacy":

[6] For examples of the genre (in my estimation—the authors don't mention software criticism), see Harvey (1991), Gough, (1997), and diSessa(1997a, 1997b, 2000).

> Criticism is referentially adequate to the extent to which a reader is able to locate in its subject matter the qualities the critic addresses and the meanings he or she ascribes to them. In this sense criticism is utterly empirical; its referential adequacy is tested not in abstractions removed from qualities, but in the perception and interpretation of the qualities themselves. (Eisner, 1991, p.114)

It will be up to the reader, then, to judge whether the themes, interpretation and designs offered here are true to the phenomena described, and help, from the reader's point of view, to illuminate the interplay of programming, media, and learning.

Test #2: Corroboration

Critical explications are more reliable when they are supported by a variety of kinds of evidence, from a variety of sources. In this dissertation I seek corroboration for my themes and interpretations by comparing known phenomena across a range of contexts, including diverse programming languages as well as linguistics, philosophy, music education, mathematics education, and sports.

In the case of a critical design inquiry, corroboration also comes longitudinally, in the story of the twist, turns, and insights that brought the design to its current state. My design narrative (Hoadley, 2002) accordingly forms a substantial portion of this document.

Test #3: Implementation

The implementation of working language prototypes provides strong evidence for the feasibility and coherence of the principles they embody. The value of those principles gains confirmation from their observed role in children's programming success, and from evidence of substantial thinking and learning.

Test #4: Consensus and Dialog

A critical work gains validity when it secures what Eisner calls "consensual validation," defined as "agreement among competent others that the description, interpretation, evaluation [and / or] thematics … are right." Eisner stresses that a community should not expect or pursue full agreement among all critics. Each work can still be judged according to how well it presents a particular point of view, and how well it contributes to the advance of understanding on the community level. This judgment emerges through the combined efforts of "competent others." To such a process, this work is respectfully submitted.

Chapter 2: Situating the Design Goals

A central concern of computer criticism is the role of larger assumptions, interpretations, values and commitments in the design of learning technologies. In this chapter I briefly review two major programming environments for learners, and discuss specific design features in terms of their connection to these broader themes. This project does not aim to develop a programming environment as complete as those described here. In tackling the problems of real-time programming, however, it does aim to generate solutions and approaches that might become part of such an environment. In this chapter, strongly contrasting language designs can be seen to reflect very different assumptions—about the nature of programming power, the factors that make programming hard to learn and do, its intellectual essence and broader connections, and the role it might play in children's education and in society. Against this background we will be able to situate and clarify some of the major design criteria driving the present work.

TOONTALK

ToonTalk is a programming environment built for children, and positioned as, among other things, a competitor and potential successor to Logo (Kahn, 2001b). Its founding goals are (i) to be learnable by children without the involvement of a teacher, and (ii) to be "a very powerful and flexible programming tool" (Kahn, 1996). ToonTalk's power comes from its underlying computational model, called concurrent constraint programming. The strategy for learnability begins with ToonTalk's fundamental form: a virtual play environment that resembles a video game more than it does a conventional computer tool.

A ToonTalk programming session begins with a user-controlled helicopter hovering over a toylike city. I bring the helicopter in for a landing. My avatar (a simple cartoon person) jumps out, with a cute little four-legged toolbox trundling along behind. I steer my avatar into a building and sit him (or her) down. Now the floor is my workspace, and I work through the avatar's arm. The toolbox hurries over and opens up, offering me bins of numbers, text, container boxes, fresh robots that I can train, nests with birds for sharing data between processes, scales for specifying comparison relationships, trucks for dispatching new processes, a bomb for obliterating this workspace, and a notebook for saving work. That may seem like a long list, until we note that this is the only palette in the whole environment. There are no menus at all. Everything you do in ToonTalk you do via the nine items in the toolbox, plus a handful of hovering characters (a vacuum for erasing, a wand for copying, a pump for scaling views, Marty the Martian for help, a mouse for arithmetic), spatial navigation among buildings that represent workspaces, and a few special keys (arithmetic keys, <escape> for exiting, function keys for summoning characters). A few more advanced tools and options are tucked away in the notebook. This is a triumph of economy. With names reminiscent of Pee Wee's Playhouse—Dusty the vacuum, Tooly the toolbox, Marty the Martian, Pumpy the pump—the characters present a somewhat grotesque first impression, but win one over with their faithful service and whimsically individual styles. One quickly grows comfortable in this little world.

ToonTalk is clearly a technical tour de force. Its most important contribution to the discourse in children's programming is, I believe, that it greatly widens the space of possibilities of what programming, and programming environments, might be. ToonTalk looks like no other programming language, and that helps to shake us all out of our complacency. ToonTalk's interface, in particular, is highly flexible and dynamic. The stiff minuet of conventional user interaction is wonderfully loosened up, into a kind improvisational dance, and, remarkably, the result is not chaos. Objects obligingly shrink and grow to improbable size as needed. Sidekick characters faithfully tag along, following the user's movements without getting in the way. Programs, tools and user are all able to pursue their separate activities without tripping each other up.

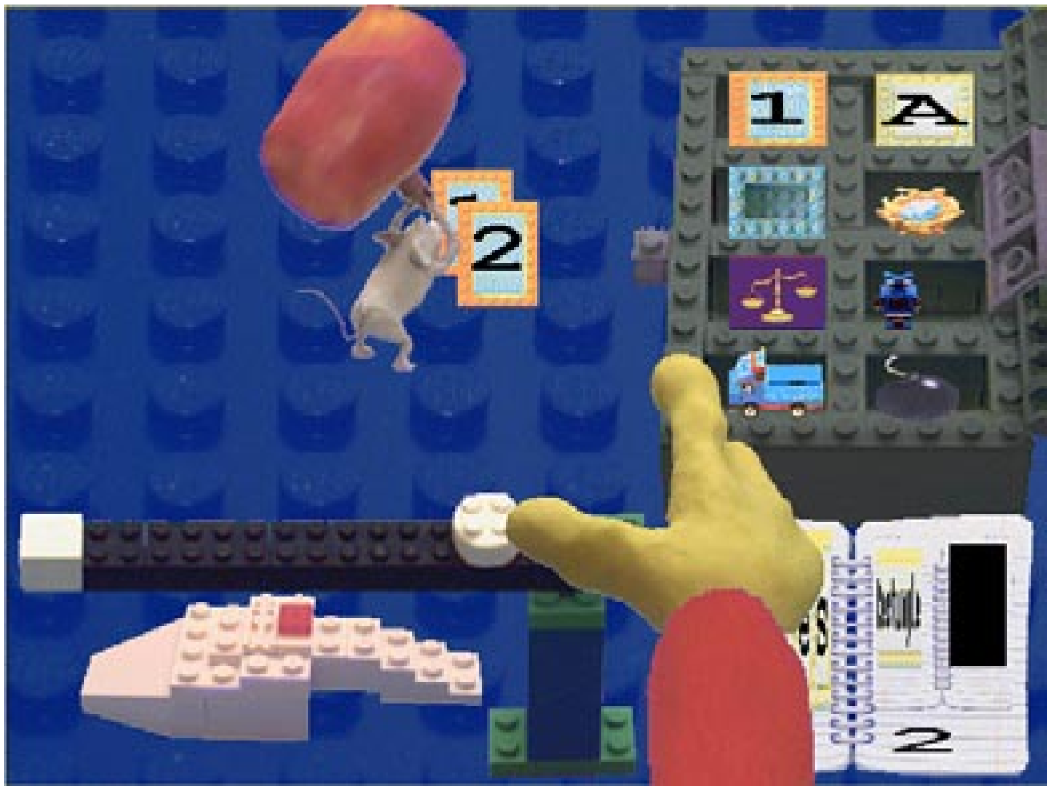


Fig. 1: When numbers need to be added in ToonTalk, a mouse runs up and hammers them together. Also visible in the picture are ToonTalk's copy wand and vacuum cleaner (lower left), Pump for size control (below avatar thumb), notebook, and toolbox. Illustration from Kahn (1996).

ToonTalk's characters and props are the foundation of its pedagogical strategy. The rationale is that what makes computational concepts hard is their alien form: "… maybe the difficulty is not in the concepts per se, but their lack of accessible metaphors" (Kahn, 1996). Accordingly, ToonTalk language elements all appear as familiar-looking (though not always familiar-acting) objects: a process appears as a house, a method appears as a robot, a comparison test appears as a set of scales, a communication channel appears as a bird and its nest, and so on. Elements of the programming interface—deletion, for example—likewise appear as animated characters.

In comparison to a more simply presented language, such as Logo, getting anything done in ToonTalk takes more time. And not just time, but mindshare. When a normal computer environment works well, its commonly used interface elements quickly vanish into transparency. When closing a window, one doesn't consciously think "now I'll move the arrow over to the close box and press the mouse button;" or even “now I’ll click the close box.” One simply closes the window. Experienced ToonTalk users certainly become more fluent with its operations, and the interface’s ability to play out actions such as house-building very quickly also helps. Still, the activity of erasing something by using a big arm to hold a vacuum cleaner that sucks things into its mouth can be made only so transparent. It is not simply that more clicks are required and more screen activity is generated: animated objects and characters call out to be recognized in a way that simple boxes and arrows do not. This is where the concrete begins to weigh us down.

All this stage business demands screenshare as well. To accommodate it, or perhaps from a desire for consistency and simplicity, Kahn is willing to sacrifice fundamental facilities like the ability to see and edit the program that one has written. Yes: incredibly, this is a programming language where the programs cannot be seen.[7] It’s not that programs don’t exist, or aren’t central to the construction process: they do, and they are. But a program is laid down by demonstrating a series of actions to a robot, who memorizes them. The only way to see the program again is when the robot acts out what it has memorized. If I want to change a method I’ve defined, my only option is to retrain the robot by acting out the entire procedure again. Ordinarily, programmers think as they code, often editing as they go (“Let’s see, start by doing X, then Y, no wait—first do W…”). This is impossible in ToonTalk[8]. For the nitty gritty business of programming, ToonTalk is extremely frustrating and inefficient.

In view of this, Kahn’s recurring emphasis that ToonTalk implements a “powerful” underlying computational model and allows one to do “real, advanced programming,”[9] can be perplexing. As best I can understand it, the terms “powerful” and “advanced” do not apply to the program behaviors that children are able to construct (these are usually exceedingly simple) but rather to the sophisticated algorithms that can be expressed in ToonTalk (a recursive merge sort, for example). This is a very inward-directed notion of power. ToonTalk’s usability problems work against building a large or medium-sized application program in which such algorithms might be applied, so simply demonstrating the algorithm remains a leading paradigm of use. Kahn claims that “ToonTalk is more appropriate for a wider age range than Logo … while 5 year olds enjoy the basic elements of ToonTalk, university students can use ToonTalk to explore and visualize concurrent algorithms and distributed programming” (Kahn, 2001b). But enjoyment, exploration and academic study are not the same as getting results. As a tool for constructing programs that do something, ToonTalk is not optimized to meet the needs of either group, nor those in between.

And what about learnability? Can one really make programming easier to *understand*, while simultaneously making it harder to *do*? ToonTalk’s driving assumption is that programming ideas are hard to understand because they lack direct correlates in the physical world. By rendering the elements of concurrent constraint programming in terms of familiar-looking characters and objects, and by using these to repeatedly act out the processes of a program, Kahn hopes to make programming more accessible

[7] This will not be so incredible to those familiar with the field of programming-by-demonstration, where invisible programs are quite common. However, such PBD systems are generally intended to help users avoid programming, not to help them learn it. If ToonTalk is attempting to help people learn programming by avoiding it, that seems like an uphill battle.
[8] The upcoming v.3 release features multiple Undo, which only partially addresses this problem. Visible, editable programs may be in the works (Kahn, 2001). It will be interesting to see how far this addition can overcome the original assumptions of the language design.
[9] Kahn, 2001b, p.1

and more fun. I can vouch for appeal of the characters and their behavior, for both adults and children. And the ability to watch processes in action no doubt helps children follow the mechanics of a program. But understanding is not a function of surface familiarity. The skills and expectations that kids bring with them about birds, boxes, and vacuum cleaners do not directly prepare them to work with the completely different properties of these familiar-looking objects on the screen. (Indeed, part of the fun is in seeing objects' conventional properties happily violated in this wacky world where mice wield sledgehammers, and houses are built in under two seconds). And watching programs being acted out contributes only a fraction of what one needs in order to construct programs that do what one wants. The core problem of learning to program remains: that is, to develop a repertoire of ways of using the language's given objects to meet a widening range of goals, weaving this repertoire together with a growing understanding of the structural principles that explain how primitives and higher-level constructions work.

ToonTalk's notion of "concrete" is too simplistic. While language primitives may look concrete, the strategies needed to deploy them are anything but. Moreover, opportunities are lacking for ideas to gain concreteness through meaningful, empowering use. In ToonTalk's tutorial game mode, Marty the Martian guides us through the language features by means of a series of challenges. Consider whether one would call these challenges abstract or concrete:

"See if you can make a box with exactly 24 zeroes in it."

"Let's figure out how many seconds there are in a day."

"Let's try to fill a bird's nest with at least five different whole numbers starting from 2."

These tasks may be concrete in the sense of inanimate, disconnected, or heavy, but if one judges concreteness in terms of connection to webs of personal meaning (Wilensky, 1991), these tasks weigh in as highly abstract.

Language or Microworld?

In Morgado's (2001) presentation of very young children's play with ToonTalk, I was struck by one particularly inventive trick. One pair of children had devised a repeating pattern in which a bird carried some datum from one house to another, and then another bird brought it back, and so on. Birds are normally used in ToonTalk for interprocess communication, but here they were used as material to construct an interesting scene. The datum, likewise, served no role except to trigger the birds' flight. This kind of construction is not programming so much as computational play.

Children's own suggestions also seem to push in the direction of play within ToonTalk's imaginary world. Kahn (1999b) reports that children often enthusiastically suggest extensions to this world. They would like to be able to land the helicopter on rooftops as well as on the street, explode houses by landing on them, venture into the water around the island, ride boats, swim, drown, see sharks. Kahn rejects these ideas:

> Very few suggestions from children for changes to ToonTalk have actually been followed. This is because every item or action in ToonTalk supports the fundamental purpose of the system — enabling its users to create programs. Much effort went into making them fun and appealing as well. ToonTalk has a magic wand because there is a need for copying things when programming. Many children find it fun to *play* with the wand, but it also serves an essential function. Blowing up houses by landing on them or drowning, not only do not serve useful functions, but they interfere with the task of programming.

This austerity is sadly misplaced. Having created an appealing virtual world where interesting things happen, Kahn is underutilizing it, because of the mischaracterization of ToonTalk's proper mission. What ToonTalk clearly wants to be is not a programming environment but a computational microworld.

As I discovered firsthand, ToonTalk can be very absorbing. I enjoyed setting myself challenges and learning my way around. My daughter, then aged four, was attracted and intrigued by the friendly characters and their strange, magical behaviors, but stymied by Marty's heavily numerical challenges. Why not embrace the notion of ToonTalk as a superb "toy" programming environment? Add more characters and places. Enlarge the set of computational materials: let programs compute with seashells and ice cream cones, not just numbers and character strings. Replace the scenario of fixing Marty's spaceship computer with a more interesting story that generates less contrived challenges. Instead of making kids build games on raw graphics sheets, give them a suite of programmable effects and interactions to create scenes and games right within the environment. This is a kind of power that ToonTalk is well equipped to offer.

Lessons So Far

The case of ToonTalk helps to bring several of the design priorities of the current work into focus—some by similarity, but many by contrast. To meet the broad goals outlined at the beginning of this chapter, the present work aims toward a programming environment embodying the following properties and principles.

1. Make programming easier to *learn* by making it easier to *do*. ToonTalk sacrifices programmer usability in favor of "concrete" portrayals of computational primitives. But primitives are not the main problem: structures and strategies are. These can best earn a place in learners' minds by providing genuine empowerment, repeatedly. The environment's usability characteristics should maximize learners' ability to achieve results in their programming. It must be a finely honed tool.

2. Support a version of programming that is engaged with the world, useful and able to produce personally meaningful results. This implies, for one thing, a strong emphasis on application programming and user interface construction. More generally, it implies a large set of primitives and a large and extensible repertoire of useful tools. In contrast to ToonTalk's view of sorting, for example, as an opportunity to explore interesting algorithms, this principle suggests that sorting utilities should be ready to hand (which doesn't prohibit thinking about the algorithms when one wants to, of course).

3. The language should reveal its own workings in an intelligible and useful way. ToonTalk is, as Kahn says, "self-revealing." It shows what it is doing as it is doing it. However, as a corollary of principle #1, I seek less in the way of high-bandwidth explanation-oriented graphics, and more in the way of handy tools to reveal the specific aspects of program state that matter to the programmer at a certain moment, e.g. when testing or debugging. In short: Reveal to empower, not to explain.

4. The language should be protean. Programming is, in part, modeling. When writing a program for playing bridge, for example, it's a good idea to organize the program in terms of entities like "card," "hand," and "discard pile," and actions like "shuffle," "deal," and "bid." One might even add some internal displays to help monitor program state, showing current hands graphically, for example. Comparatively bland primitives such as variables, keywords, and subroutine names are readily assimilated into whatever story the particular program needs to tell. A scene full of bombs and birds' nests, on the other hand, leaves less room, on the screen or in the mind, to

maintain these model entities. ToonTalk's program elements are anything but protean, and they illustrate the drawbacks of hogging the metaphor space—drawbacks, that is, if one is interested in programming substantial applications. ToonTalk's approach is more justifiable in a language devoted to programming itself as the object of interest. To say it another way, programming in ToonTalk is opaque: when one looks at a program, one sees ToonTalk. I seek transparency: the ability to look at a program and see what it is saying about the domain of application.

5. Provide strong tools for real-time programming. This is a corollary of principle 2 (engaged with the world), and technically it implies that concurrency should be a first class language property. This is true of ToonTalk's concurrent constraint paradigm, and the benefits to the environment's interface are clear. It is less clear that adequate tools are provided for the coordination challenges of, say, a video game. But I agree with Kahn's (1996) argument that the heritage of single-thread programming has always been artificially limiting, and is especially problematic in view of kids' interest in games and other real-time applications.

6. Support a wide range of users, from beginners to experts. I do not accept Kahn's implication that ToonTalk adequately meets the programming needs of users from age 5 to college. Yet I agree that the proverbial "low threshold, high ceiling" is a highly desirable property. Most programmers work in some kind of community, be it a school, a club, a business, or a network of friends. Some members will be doing more advanced programming than others: if all are working in the same language, there is more sharing of help and inspiration. A language with a high ceiling provides a smooth ramp of challenges and rewards that help to draw interested learners ever onward. Then too, a language that serves advanced users often is simply more polished and evolved as a working environment, and its smoothness and reliability can be felt by all users—as modest home cooks appreciate professional cookware. Finally, a language with a high ceiling is more likely to be extensible. It supports the gradual accumulation of useful tools, which, though built by advanced users, can extend the creative range of beginners.

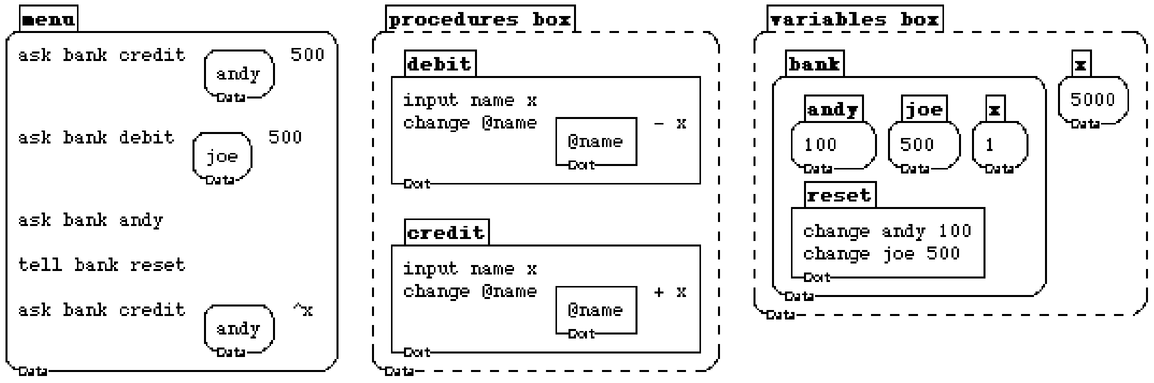
To bring forth more guiding principles it will be helpful to have a second reference point. We turn now to Boxer.

BOXER

Boxer is positioned, even more explicitly than ToonTalk, as a successor to Logo. Boxer appropriates Logo's goal—"to provide the simplest, but most powerful and unconstrained computational resources possible to 'just plain folks' in the service of enhancing learning" (diSessa, 1997a)—some aspects of its syntax, and its trademark turtle graphics. In Boxer, this heritage undergoes a deep restructuring, changing the way programs are presented, organized, and used.

The key to Boxer's uniqueness is its space. The virtual space of a Boxer environment accommodates not only program code, but program data, user interface elements, and other media objects. The space is organized by means of two complementary structures, text and boxes. Boxes can be included in text (where they behave like just another character), and text can be entered in boxes. Boxes can also contain graphics, video, file access, crosslinks into other parts of the document, or links across the Internet. The result is a roomy hierarchical structure of unlimited depth, well suited to organizing a variety of mixed media documents, any portion of which might be executable code. The full capabilities of this flexible structure take some getting used to. Boxer can serve as a vehicle for tutorials or courseware, an email client, a simple alternative to PowerPoint, a file system browser, even an alternative World Wide Web. It is for these reasons that Boxer is labeled a "computational medium," not just a programming language (diSessa, 2000). For diSessa, Boxer represents the emerging potential for a new literacy in our society, potentially as pervasive as print. What makes a literacy both worthwhile and learnable, in diSessa's view, is its repeated use in a wide variety of contexts, some mundane, others profound. Just as reading grocery lists helps to justify and maintain the skills that also let us study the Bill of Rights, so writing a simple script to process email helps to justify, develop, and maintain skills involved in programming a simulation, a mathematical tutorial, or a custom visualization for scientific data. The integration in Boxer of so much material that is not itself code increases the occasions where programming can be useful. Here we have a different model of the nature of programming power and means by which it can be acquired.

As a programming language, Boxer has several interesting properties, many of which flow from its document structure. What stands out at first look at is all the boxes. Boxes demarcate blocks of code (what begin / end does in Pascal, brackets in Logo, or indentation in Python). Boxes also delimit procedure definitions, establish variable storage and bindings, and organize data structures and object (i.e. code + data) structures. One important consequence of all this boxing is that program code and program data can coexist in a coherent organization. In most programming languages, variables are invisible entities somewhere off in computer memory. In Boxer, a variable is a box that can be kept right near the code that uses it. Execute a line that modifies the variable, and you see the effect immediately. If you want a different value in the variable, just edit it. DiSessa calls this Boxer's "concreteness" principle: that "the user of a computer system can pretend that what appears on the screen is the computer system" (diSessa, 1997a). (Compare this with the ToonTalk sense of concreteness as familiar solid object). This kind of concreteness removes a cognitive stumbling block, but also greatly enhances usability, adding openness and fluidity to the processes of coding, testing and debugging. It s a good example of the slogan offered earlier: "reveal to empower."


```
menu
ask bank credit  andy  500
ask bank debit  joe  500
ask bank andy
tell bank reset
ask bank credit  andy  ^x

procedures box
debit
input name x
change @name  @name  - x
credit
input name x
change @name  @name  + x

variables box
bank
andy 100   joe 500   x 1
reset
change andy 100
change joe 500
x 5000
```

Fig 2: Boxer snapshot illustrating coexistence of code, data, and a rudimentary form of user interface (the box of double-clickable lines of code, at left)

Boxer shares with many interpreted languages the ability to execute a single line or other small chunk of code at a time. This feature gains power when the domain of effects of the code (in this case, variables and data structures) are visibly present at the same time and in the same place, allowing for a type of activity that diSessa calls “semi-programming,” the use of incremental coding and execution either to explore and learn, or to gradually accumulate a program’s worth of tested code.[10]

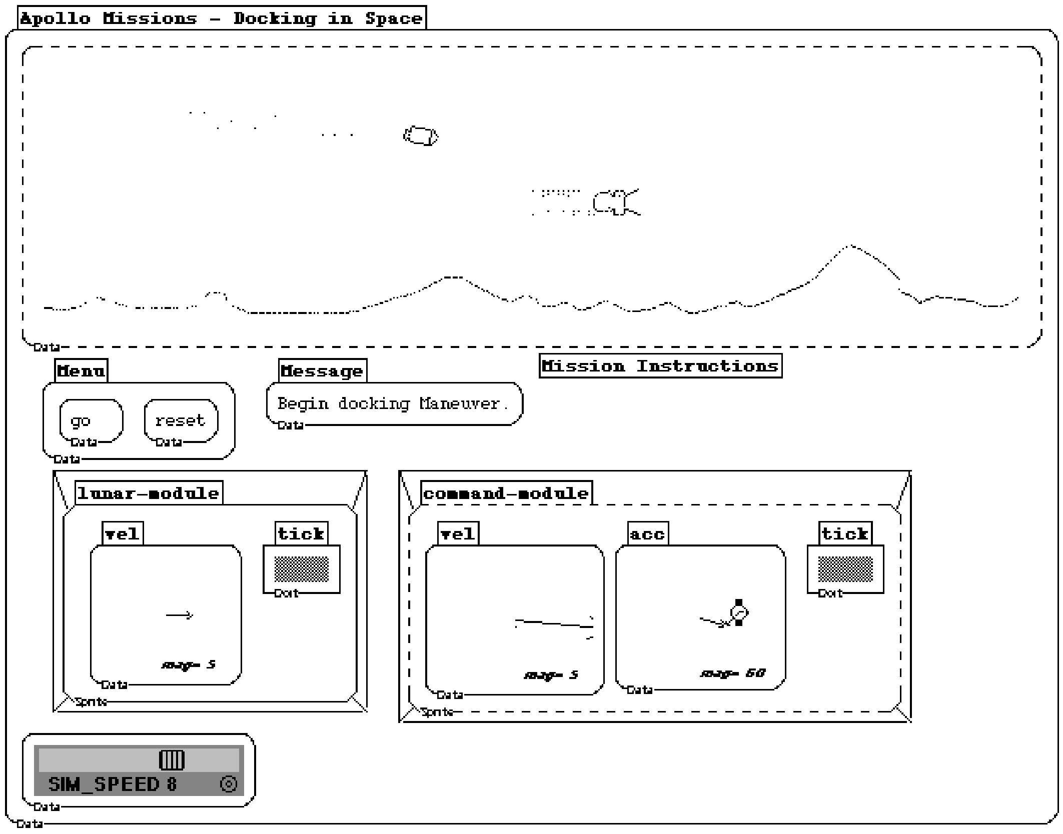


Fig. 3: A simple game to explore the mathematics of motion. The user must successfully dock the command module with the lunar module, using an acceleration vector to control the craft.

There is one more property of Boxer that is important to mention here: the ability to construct and include interface elements directly within the programming space. Fig. 2 shows a very simple instance of this, where a box containing lines of code serves as a rudimentary menu. In Fig. 3 we see a more elaborate interface featuring a slider (bottom left) and interactive graphical views. Even here, the code that drives the game is accessible. The impact of this integration of interface and program within a single space is quite profound. To begin with, it simplifies user interface programming by short-circuiting the cycle of coding and testing. It also allows the programmer to pepper her code with handy little programmer’s tools that may have graphical interfaces. But beyond that, at what one might call the sociological level, it breaks down the barrier between being a programmer and being a user. Clearly it supports the notion of programming on one’s own behalf—whether to develop a simulation or other kind of inquiry device, to process personal data such as email, or to customize one’s work environment. But it also means that any program shared between people comes with a built-in invitation to look inside. It is easy to make a gradual transition from using a program as a black box, to poking around in the code,

[10] Semi-programming first appeared with the Logo turtle, and is also familiar from spreadsheets. Repenning and Ambach (1996) use the term “tactile programming” to describe a related property of programming in Visual AgenTalk.

perhaps twiddling some parameters, then making modifications or additions, or borrowing chunks of code for use elsewhere.

An extension of this model of program sharing is diSessa's notion of "open toolsets" (diSessa, 1997b). A beginning cook's creative range is greatly augmented by packaged sauces and mixes, frozen crusts and doughs, and the like. Gradually one gains the knowledge and confidence to make more parts of the meal from scratch, as desired, but prepared components are always a useful resource. Similarly, open toolsets are domain-specific collections of useful computational components that can be combined in many generative ways. With an *open* toolset, one has the added ability to get inside a component, to customize or adapt it in ways unanticipated by its original creator. An appropriate open toolset can help both beginner and expert get more interesting products up and running more quickly in a particular domain, be it Newtonian physics, text processing, or video control. Open toolsets also have the potential to grow and evolve in an open-source style, through the contributions of those who use them.

The foregoing presentation has surely not concealed my approval of many aspects of the design of Boxer, and the accompanying models of how programming can empower people, how they can learn it, and how computational media might support intellectual and cultural communities in schools and beyond. Of the six principles I put forward earlier, Boxer scores high on all but one (the exception is real-time programming—see below). Indeed, Boxer does so many things right that it has sometimes made me worry for the prospects of programming in schools. If something this good has failed to catch on, will anything? I argued at the opening of this chapter that we need a better kind of programming in order to win educators over. But how much better can programming get? Will schools continue to resist programming no matter how well it is realized? Or alternatively, are my quality judgments way off?

With all of Boxer's strengths, I think one can identify a short list of significant reasons that may account for a good deal of its difficulty in spreading in our culture.

First, there is a collection of issues I'll call *polish and appeal.* As diSessa has acknowledged, Boxer needs an interface makeover. Its visual style is dated and not that pretty to look at. Boxer's interface is not immediately inviting, requiring knowledge of special keys and special places to click. Its repertoire of user interface elements and graphics tools is small and rudimentary. In addition, Boxer lacks a rich set of polished sample programs—games, microworlds, courseware—to draw children and teachers in. Nor is there yet a strong collection of open toolsets for multiple domains.

Second, there is the *user threshold* problem. Boxer has a low threshold for beginning programmers: that is, it is not hard to get started programming in Boxer. But what if your immediate goal is not to start programming but simply to try out a friend's game, or a colleague's physics materials? Unfortunately, the flip side of Boxer's integration of code and interface currently means that you can't really use these things without knowing something about how to navigate Boxer and run code. It's not too hard to learn those things, but the point is, all you wanted to do was use a program, so in practice this can be a very effective barrier. Boxer's user threshold drastically reduces the pool of potential users of Boxer programs, and deprives Boxer of one of its best opportunities to spread: by letting a programming community gradually emerge from within a much larger user community.

The two problems mentioned so far are fairly straightforward: fixing them is just a matter of work (and money). The next two are thornier.

The *real-time problem* is the same general problem that I have identified and that is the subject of this thesis. The dynamism of Boxer programs is severely limited by the lack of tools for creating and coordinating multiple processes. The lunar lander game shown above illustrates the issues at a level of polish. To play the game I invoke the "go" procedure. Then for the duration of one game, my cursor is shaped like a watch (to show that a program is running), which is ugly and makes it hard for me to see where I'm pointing, and the rest of Boxer's interactivity—menus, editing—is disabled. To stop playing I have to press a special interrupt key. These are some of the awkwardnesses of running in a single-thread system. DiSessa says "Boxer has a multithreading system, but we haven't bothered debugging it fully because it hasn't proven useful" (1997a). I suspect that Boxer has simply not been pushed hard in the direction of real-time applications like robotics, video games, or even highly polished interfaces. Yet applications like these would attract many more programmers and users, in addition to opening up new learning opportunities.

As a computational medium, a dynamic Boxer would have a very different feel from the current Boxer. The Boxer universe is relatively static. It is a realm of ideas and contemplation, and not of action. Lunar Lander has the form of a video game, but it isn't a *real* video games; it lacks action or immersion. One does not become truly physically engaged with it. Animated though it is, the feel of playing it is closer to a crossword puzzle than to a computer game.

Questioning the social value of "twitch" video games, one might wonder whether Boxer is better off remaining relatively static. But of course, twitch games are only one manifestation of real-time interaction. The great promise, in general, of a *real-time* computational medium is a new bridge between the intellectual and the physical, between our bodies and our ideas. The revolutionary potential of this bridge is not easily spelled out in words, and we may not fully appreciate it until we see it, but I think it is something that children understand viscerally.

Finally, Boxer faces one more problem, which this work will not help to solve, but which bears thought. I call it the *parallel universe* problem. Boxer represents a grand vision, remarkably fully elaborated, of a computational universe considerably more open and empowering than the one we have. It clearly illustrates how radically empowering programming could become if differently integrated into the fabric of computer systems and institutions. Yet to take full advantage of this now, one has to "move in" to the Boxer world, forsaking valuable tools that are not part of Boxer. DiSessa (1997a) writes "consider how nice it would be … to be able to write a simple program to sort mail for you." But actually, my current mail program sorts my mail just fine most of the time; if I could run a Boxer program on my mail every now and then I might, but the difference is not enough to warrant me switching to Boxer's email system. Boxer also offers "network browsing." The only catch is, all you can browse are other Boxer systems. The World Wide Web that we all know is off limits. I wish that Tim Berners-Lee had conferred with diSessa back in 1989, before developing HTML and all that. If he had, we might have a much more congenial World Wide Web. But now that we've got the one we've got, there's no giving it up.[11]

In short, the opportunity, if there ever was one, to build our entire computational world within a humanely programmable computational medium has passed. The parallel universe of Boxer stands as a demonstration of what might have been, and perhaps a pure model for something that in reality can only be achieved messily and

[11] I don't mean to imply that diSessa wants us to abandon the "real" WWW. But meanwhile, the isolation of the Boxer web from the WWW is a real problem, and the fact that, in my opinion, a Boxer-like web might have been better, is now moot.

partially. Several questions follow from this: Just how far can a programming language now get in permeating the computational universe, and especially that of children? What combination of strategies, technical and social, could get us there? Would it be far enough to gain the critical mass of use needed for a flowering of computational literacy?

Summary of Language Design Principles

Combining the previous list of design principles with major points from the Boxer discussion, we get the following list of desiderata for an educational programming language.

1. Make programming easier to *learn* by making it easier to *do*.
2. Support a version of programming that is engaged with the world, useful and able to produce results.
3. The language should reveal its own workings in an intelligible and useful way.
4. The language should be protean—able to include representations of entities in the application domain.
5. Provide strong tools for real-time programming.
6. Support a wide range of users, from beginners to experts
7. Integrate code, data, and interface elements within a single space.
8. Support development of polished, self-contained applications usable by non-programmers.

Most of these principles will come into play, to some degree, in the designs presented here. Still, I want to be clear that the current project does not aim to produce a complete language fully exhibiting these properties. Rather, this will be a design exploration of the challenges and implications of principle #5, with the remaining principles helping to define the context in which the solution to #5 should eventually fit.

Chapter 3. Robotics Programming and the Design of Flogo I

THE STATE OF CHILDREN'S ROBOTICS PROGRAMMING

Robotics is a relatively new domain for children, but a domain that is quickly gaining interest among educators, researchers, and especially children. In general, children love to build robots and create behaviors for them. In doing so, proponents argue, they can learn on many levels—about science, mathematics, engineering, design, and artistic expression, as well as programming (Druin&Hendler, 2000; Resnick, Berg, & Eisenberg, 2000). Miller and Stein (2000) list several educational goals for robotics, including exposing students to basic engineering principles, providing a motivating context for application of math skills, building problem-solving skills, building teamwork, and promoting interest in science, math, and school in general. Turbak & Berg (2002) advocate creative robotics as a context for learning the "big ideas" of engineering, including "iterative design, real-world constraints, tradeoffs, feedback, complexity management techniques". Such ideas, they argue, are important for all of us, not just professional engineers. (No-one, it should be noted, is advocating robotics as a regular school subject. At the precollege level, robotics is occasionally a classroom enrichment activity, more often an afterschool or club offering.)

While these writers have undoubtedly achieved some of these goals in their own work, a more widespread and longer-lasting impact is by no means assured. The problem is that programming a robot to behave in interesting or intelligent ways can be surprisingly hard. Like generations of AI researchers before them, children often find that goals must be repeatedly scaled back as the complexity of producing seemingly simple behaviors is revealed. Many writings on robotics for kids are still at the stage of early enthusiasm, and critical problems are not fully confronted in the literature (but see Martin, 1994 & 1996). Children's degree of mastery and accomplishment in robotics programming is not easily measured, and little measurement has been attempted. Yet at some level it seems clear that children's robotics programming has not really taken off. For example, Lego Mindstorms™ is sold as a children's product; certainly some children make sustained, rewarding use of it, yet a substantial proportion of purchasers of Mindstorms are actually adult hobbyists, who draw on a deep prior knowledge of computation to get interesting results. In my experience of robot-making here at the Media Lab, children often construct fixed action patterns, perhaps triggered by a single sensor crossing a threshold, or else extremely simple sensor-effector dependencies. The same was true at the 2003 Boston Robotics Olympics, where teams of schoolchildren aged 10-14 had constructed and programmed robots to meet a variety of challenges, such as:

- slowest robot contest: use gearing systems to make your robot go as slowly as possible
- obstacle course: using two touch sensors as steering controls, drive your vehicle through an obstacle course.
- ball delivery: robots are required to pick up a ping-pong ball, carry it across a table surface, and drop it into a cup.

One might expect the ball delivery challenge to involve sensor-based navigation to find the ball or the cup and engage correctly with each. In fact, each robot was programmed with a fixed series of motions which succeeded or failed based on how precisely the robot was initially positioned, how sensitive the dropping action was to variations in the relative positions of cup and robot, and luck. Of the seven challenges, only two involved sensors at all. One was the obstacle course, where the only programming needed is to make each motor output contingent on the corresponding switch. In the other, the robot was required to sweep packing peanuts from a square area marked off by black tape, while remaining within the square. This was one of the most difficult challenges and had far fewer entries than most. No entrants used sensors to detect the peanuts (instead, the robots simply moved and flailed about until the time limit was reached) but two did use sensors to detect the perimeter line. According to an event organizer, this year's challenges required somewhat less sensor use than the previous year's—evidently an adaptation to what the contestants were able to produce with available tools (LEGO Mindstorms RCX and RoboLab software).

For older children, expectations of robotics programming are often not much higher. The FIRST robotics league is a major organizer of robotics events for high school students. Up until now, this has been a contest of manually controlled robot behavior. The coming year's competition will be the first with any requirement that robots operate autonomously—and the requirement will be limited to a special segment of the trial, lasting just 15 seconds. Similarly, the annual submersible robot competition hosted this year at MIT, and attended by college and high school teams from around the country, featured manually controlled robots only.

Some events place a higher emphasis on autonomous control, for example the FIRST Lego League, the KISS Institute Botball program (Miller& Stein, 2000), and the Trinity College fire-fighting robot contest[12]. In these contexts, it appears that younger teams get by through a combination of coping strategies, including substantial programming work by adult mentors, the provision of task-specific program templates, and delegation of programming responsibility to a single knowledgeable team member. With all that, the final behaviors may still be very simple.

Constructing behavior is an essential part of robotics. Propping up children's programming work is a time-limited strategy. The flip side of the story I have told is that children are often fascinated with constructing behaviors for robots, and are willing to work hard at it, and gain a very special kind of satisfaction from getting even the simplest autonomous behavior working. But without a sense of growing power to understand and create interesting behaviors, most children will eventually lose interest. In the absence of broad numerical data, I will venture a guess that while many children who try robotics in one context or another have enjoyable and worthwhile first experiences, far fewer are drawn into sustained work, because behaviors beyond the most elementary ones prove too hard to achieve. Without better programming tools, the prevalent reality of children's robotics will not catch up to current visions, compelling though they may be.

Robotics languages for children

Among robotics languages for learners, only a few are regularly used by children. These are:

Cricket Logo[13], which grew out of LEGO/Logo (Resnick&Ocko, 1991), is a core subset of the Logo programming language, with a few additional primitives for querying sensors and controlling motors. It includes two control structures added with robotics in mind: a "waituntil" construct that pauses execution until a given condition is

[12] http://lor.trincoll.edu/~robot/

[13] http://llk.media.mit.edu/projects/cricket/doc/index.shtml

satisfied, and an implicitly parallel “when” clause, which can be used to set up a single “demon,” e.g.:

```
    when (sensorA < 100) [beep backup]
```

LogoBlocks (Begel, 1996) features a syntax and feature set which are almost identical to Cricket Logo, but syntactic units ride on draggable blocks, coded by color and shape, which can be arranged to form programs and procedures. This iconic interface eliminates common syntactic problems such as balancing delimiters, correct formation of control constructs, and so on. A similar iconic language, RCX Code, accompanies the retail version of the *Lego Mindstorms* robotics kit.

RoboLab (Portsmore, 1999) is a simple iconic language that executes step-by-step programs visually. Control is by means of branches and conditionals. RoboLab is built in LabView, so many people assume it shares LabView’s dataflow paradigm. But RoboLab is a conventional procedural language.

NQC (“Not Quite C”—Baum, 2002) is a small subset of C, augmented with fairly direct access to the firmware capabilities of several robotics microcontrollers, including Lego’s RCX. The language omits many of C’s data types and includes arrays as the only data structure. Still, NQC is rooted firmly in the culture of C, and relatively abstract features and conventions remain: for example, the #define macro-definition facility and the use of named integer constants (e.g. OUT_A / OUT_B for outputs, OUT_FULL / OUT_HALF for power levels) in API calls. *Interactive C* (Newton Research Laboratories) is a complete C environment for several robotics controllers, originating in MIT’s 6.270 robotics challenge course. Both NQC and C allow for line-by-line testing. Both have been used by middle and high school students (Miller&Stein, 2000).

A true dataflow programming model is provided by the commercial robot simulation game *MindRover* (www.cognitoy.com). The language is a variant of the wiring model, in which each wire stands for some dependency between two components. However, the exact dependency cannot be read directly from the wiring diagram; instead, each dependency can be viewed and edited in a dialog accessible via its wire. MindRover programs can be used to program virtual robotic vehicles on the screen; they can also be downloaded to physical robots.

In addition to the foregoing, a number of interesting programming environments have been developed for research purposes. Most of these fall into one of four categories: procedural, dataflow, rule-based, and formula-based (Ambler, 1998). For example, *LegoSheets* (Gindling et al., 1995) is a simple introductory-level robotics programming tool developed within the AgentSheets (Repenning, 2000) programming environment. A LegoSheets program consists of a set of rules associated with the robot’s effectors. There is no provision for higher-level structuring of effector groups: each must be programmed separately. LegoSheets’ explicit design goals include providing a paradigm (rule-based programming) better suited to the non-sequential needs of robotics programming; and supporting “live” real-time execution, to help students make sense of their programs in action. In the same general family is the actor-based children’s language *Cocoa*, which has been applied to robotics in research settings (Heger et al., 1998). More eclectic is the work of Cox and colleagues (Cox et al., 1998; Cox&Smedley, 2000), who have experimented with combinations of dataflow, finite state machines, subsumption architecture, programming by demonstration, and tools for building custom visual environments that in turn allow user to specify behaviors more concretely, in terms of visualizations of the robot and its environment.

At the Media Lab there have been explorations of a variety of programming approaches for robotics, ranging from straightforward adaptations of the Logo language, to multitasking extensions of Logo (Resnick, 1990), iconic program-construction tools (Begel, 1996), and programming by physically wiring up simple logic components (Hogg, Martin, & Resnick, 1991). While not applied to robotics, Travers' (1996) agent-based programming models engage equivalent problems in constructing behavior in onscreen worlds, or what Travers calls "animate systems." Travers' "Dynamic Agents" model, in particular, achieves an interesting kind of power by bringing constraint-based programming into the framework of concurrent, hierarchical agent structures. While children are envisioned as ultimate beneficiaries of an agent-based approach to programming, the models actually developed by Travers are not targeted toward beginning programmers.

FLOGO I

The impetus for the development of Flogo I came from the perception of a potential opportunity to raise the level of children's robotics programming by providing better tools. In particular, the design sought to improve on existing environments by:

1. opening up the computational black box by revealing the executing robotics program
2. supporting the development of larger projects and open toolsets
3. providing better means of managing the concurrent processes involved in robotics programs

The first goal addresses a major handicap afflicting almost all available robotics environments[14]: that to run a program one must first download it to the robot controller, so that program state is completely invisible while the robot is behaving. This situation creates an obvious impediment to comprehension and debugging.

The second goal reflects an observation that considerable research effort in robotics languages for learners has gone into making languages more accessible at the level of language primitives. The drawback of this approach is that the most beginner-friendly primitives may become an impediment as projects become larger. Flogo I sought leverage for beginners in recombinable graphical tools that could be built within the language itself.

The third goal is the thorniest. It is really one of the fundamental problems of real-time programming. I began to appreciate the problem in 1998, when I helped to run a brief robotics workshop[15] for a small group of children aged 10-13, involving some moderately complex programming tasks (simulating the popular electronic toy Bopit[16], in particular). That these behaviors were too hard for young beginners to construct on their own was not surprising. What surprised me was my sense, as we began to work with the children, that the necessary programming tricks were too arcane to be worth children's time. We decided to simply provide the completed program. Around the same time, conversations with Bakhtiar Mikhak about the possibility of modular computational components (both physical and virtual) passing data along wires suggested the possibility of a new visual data flow language, in

[14] The exception is the small prototype environment LegoSheets (Gindling et al., 1995)
[15] Entitled "Concentration," this workshop was a joint project with Bakhtiar Mikhak and Claudia Urrea.
[16] Against a catchy rhythmic background track, this toy issues repeated, randomly chosen action commands ("Bop it!" "Twist it!" "Pull it!"). The player tries to perform each action in time for the next command.

which inherent parallelism would make Bopit, and many other robotic behaviors, simpler to program. The result was Flogo I.

As the name suggests, Flogo I supports the representation of program processes in terms of data flow. Flogo I is not unlike the visual dataflow language LabView (National Instruments), but optimized for use by learners in robotics (differences between Flogo I and LabView will be reviewed at the end of this chapter). Flogo I's distinguishing features include:

• the ability to run on a desktop computer with a "tethered" robot, so that program can be viewed, as well as edited and intervened in, while the robot is behaving.

• a generalized capability for dataflow components to have graphic interfaces that make their activity easier to monitor and control.

• an encapsulation facility allowing nested organization of circuits within components.

• a planned, but not completed, notational language, allowing users to define new components in terms of actions to be taken at each system tick. (Instead, we made do with a LISP-based component definition language, not appropriate for novice use).

• a variety of small innovations designed to make common graphic operations simple—for example, moving one end of a wire, inserting a component on a wire, managing rearrangement of diagrams when opening and closing components.

Why dataflow?

Several considerations give plausibility to the idea of using a dataflow representation for robotics programming:

- The match between dataflow and robotics is especially clear at the robot's sensorimotor periphery: A sensor acts a source of a stream of data values; likewise a motor acts on the basis of a stream of control values.
- The natural parallelism of dataflow programming seems to fit well with the inherent parallelism of many real-time control problems.
- There are successful precedents for the use of dataflow programming in real-time applications (e.g. LabView in scientific instrumentation, Opcode's *Max* in music synthesis and processing).
- Dataflow presents particular opportunities for "liveness" (Tanimoto, 1990) in the programming interface. I will explore this theme in depth later in this document, but the main reasons can be stated simply: (1) an up-to-date display of changing data values on wires can be maintained without disrupting program editing or display; and (2) one can envision the intact parts of a dataflow program continuing to operate meaningfully even as other parts are being edited.

But there were grounds for caution as well as optimism. Visual and dataflow programming paradigms are known to have weaknesses of their own in comparison to textual and procedural ones (see e.g. Green & Petre, 1992; Whitley, 2000). Even scaled-back conjectures that their representations are superior for certain types of tasks have not been easy to confirm (Good, 1996). Flogo I work was conceived as a design exploration through which both the needs of learners of robotics programming and the potential of dataflow to meet those needs might become clearer, with the understanding that major changes or extensions to the dataflow paradigm might ultimately be required.

In conjunction with Flogo I, we used "cricket" control computers, developed at the Media Lab (Martin, Mikhak, et al., 2000). Rather than downloading programs to the cricket's on-board processor as most programming environments do, we kept the cricket tethered to a desktop computer via a serial cable, so that the operation of the program can be seen and adjusted in real time.[17] Robots were built mostly in Lego, with some additional craft materials.

A Simple Flogo I program

Fig. 3 shows a simple Flogo program that controls a vehicle with reflectance sensors for detecting the surface of a table. On the left side of the program are components corresponding to the sensors, with black fill that shifts up and down as sensor values fluctuate. The connection is "live," so, for example, one can hold the robot and move one's hand near and far from the sensor, and watch the components respond onscreen.

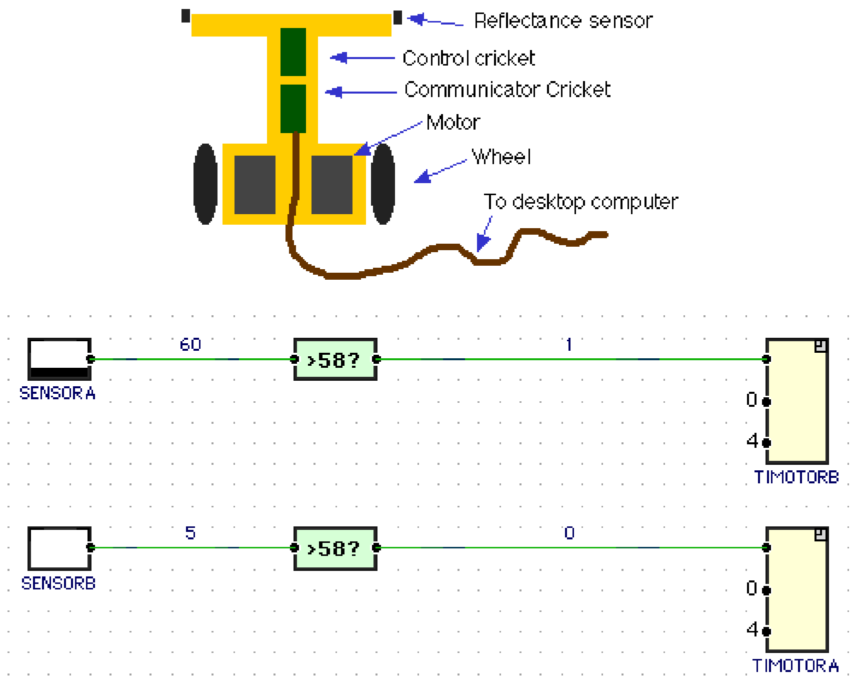


Figure 3. A "Table Explorer" robot and a simple Flogo program to control it. When a sensor moves off the table, the diagonally opposite motor is inhibited.

From the sensor components, wires carry the stream of constantly updated sensor values to two simple threshold units, which convert the data to either 1 (sensor value above 58, therefore sensor over table) or 0 (value below 58, therefore sensor off table). These values, in turn, are passed to the on-off inputs of components that drive the motors. Each sensor is used to control the motor on the opposite side of the robot. The net effect of this simple program is to prevent the robot from driving off the table. If the robot approaches the table edge at an angle, it will turn just enough to drive along the edge.

[17] The cable adds some awkwardness, especially to mobile robots. In principle one could use radio communications instead, but we found the cable adequate for our needs. A natural addition to Flogo I would be the ability to compile programs for download after program development is complete. Since this is unrelated to the project's focus on the programming process, we did not implement it.

What if the robot drives straight to the edge of the table, so that both sensors go off the edge simultaneously? In that case, the program in Fig. 3 will stop the motors in time to prevent a fall, but the robot will remain stopped. Fig. 4 shows a more complex program with two new components inserted in the center area. When both sensors are off the table, these components cause one wheel to drive backwards, causing a reverse turn until one of the robot's sensors is back on the table and normal locomotion can resume. (The understandability of this program for beginners is a topic to which we return in Chapter 4).

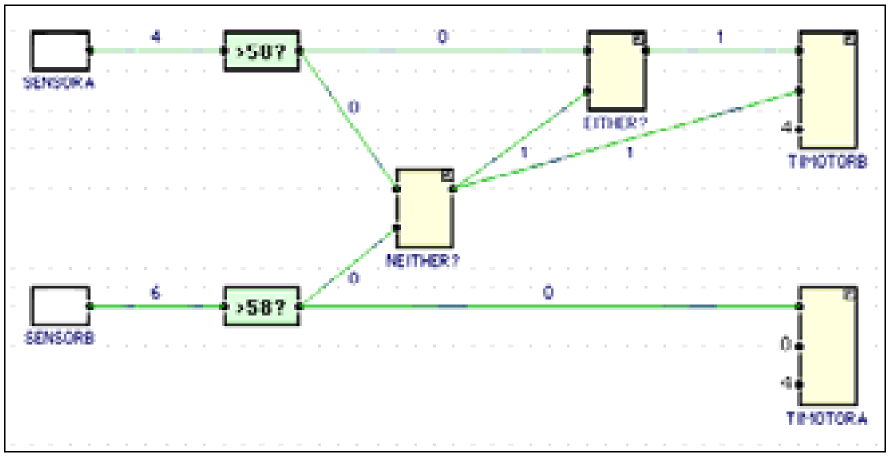


Figure 4. A more complex Table Explorer program. If both sensors go off the table, one wheel turns backwards. (The second input node on the TIMOTORB component controls motor direction).

Some basics of Flogo I are visible in this example; others don't come through in a snapshot. Flogo I components accept inputs on their left side and generate outputs on their right side. Wires carry information in one direction only; the direction is indicated not by arrowheads but by a marquee-style animation on the wire. Multiple wires can branch out from a single node. Each input and output has a name, which is displayed when the mouse passes over (Fig. 5). For example, the three inputs to a TIMOTORA component (stands for Three-Input Motor A) are named, from top to bottom, "on?" "backwards?" and "power." A constant input value can be simply typed in next to the corresponding node. Thus the TIMOTORA component in Fig. 4 may be switched on and off, but it never runs backwards, and always runs at power level 4.

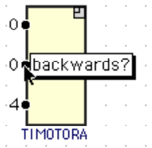
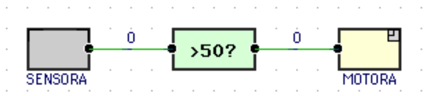


Figure 5: holding the mouse over an input to find out its name

Pulses

Another important entity in Flogo I is the "pulse," which travels on special pulse wires and can be used to signal events and initiate actions. (Pulse wires appear yellow in contrast to the green of ordinary data wires). Figure 6 shows a typical use of a pulse in connecting the behavior of a touch sensor and a light. In Figure 6a, inputs and outputs are connected directly, much as in the earlier examples. This means that the light will stay on just as long as the sensor lever is depressed. When the lever is released, the light goes off. Figure 6b changes this. The signal from the touch sensor goes into a component that detects the crossing of the threshold (the exclamation mark in “>50!” designates this function), and will fire a pulse at the instant that its input crosses above 50. This pulse travels along a pulse wire to a “switch” component, where it triggers a change in the switch’s state (if the switch is off, it turns on; if on, it turns off). With this program the touch sensor behaves like a power button: press it once to turn the light on; press again to turn the light off.

(a) 

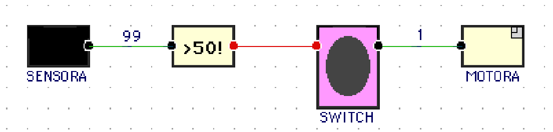
(b) 

Fig. 6. (a) Sensor state fed directly into output state. (b) Use of a pulse-activated switch to hold the output on or off. Wire flashes red momentarily as pulse passes through.

Pulses are the standard protocol for objects that need to be started or stopped, and for objects that perform discrete transactions of some kind, such as incrementing a counter or toggling a switch. A number of Flogo I components help with the management of pulses: they emit pulses at regular intervals, route pulses along different wires, count pulses as they are received. The pulse-delay, which imposes a time delay in the path of the pulse, comes in handy in a number of contexts, such as the idiom in Fig. 7, which emerged early in the Flogo work, for turning an output on for a specified period of time:

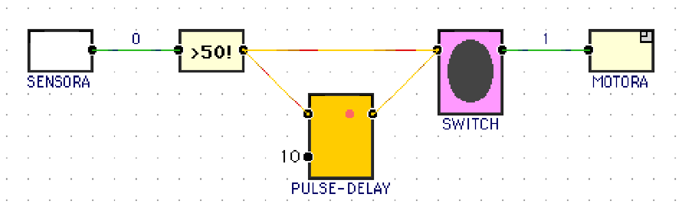


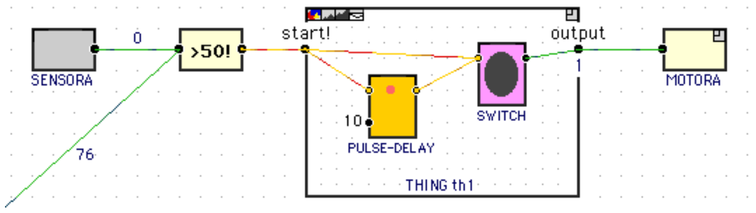
Fig. 7. Using pulses and delays to turn an output, such as a light, on for a fixed period of time. Clicking a touch sensor generates a pulse, which has turned a switch on. A delayed copy of the pulse will arrive at the switch one second later, turning off the output. The delayed pulse appears as a red dot moving across the PULSE-DELAY component.

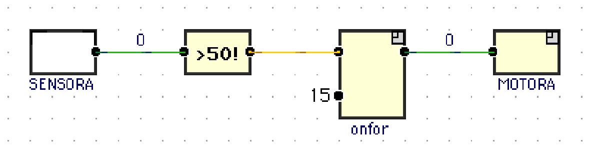
Encapsulation

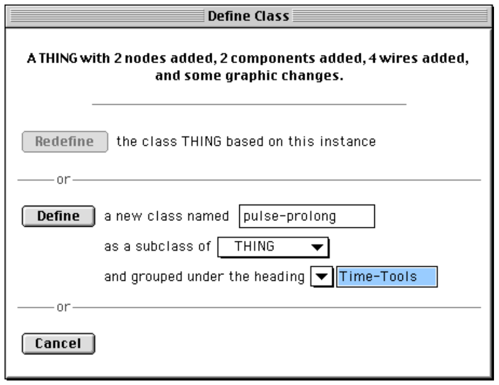
Flogo I’s encapsulation model facilitates hierarchical program organization and language extensibility. Fig. 8 shows how the little trick of using a delay to turn something on and then off again could be made into a standard component, by a series of steps. First, part of a Flogo I program can be enclosed in a box, and wires to and from funneled through input and output nodes. This box can be closed and opened as needed to hide or reveal detail. If desired, this new component can be made the prototype for a new class of components, accessible from the construction palette[18]. Finally, an interactive graphical interface can be defined for the component.

[18] Flogo I includes a subclassing model whereby a class can be defined in terms of its differences from a parent class. For example, “mslider” (motor slider) is a subclass of slider in which the low and high values are preset to the range required by motor s. Yellow buttons are defined as a subclass of buttons with a yellow color scheme. In a traditional object-oriented language, subclasses modify their parent classes solely through the addition of new or shadowing instance variables and methods. In Flogo I, the user is additionally free to change, rearrange or delete elements defined in the superclass, so the meaning of inheritance becomes alittle trickier. In practice, Flogo I’s subclassing model was only lightly used, and when it was, it was most often used as a way to set default parameters, as in the examples mentioned. As an alternative to subclassing, it is easy to make a new freestanding class by copying and

(In the current prototype, however, interface construction is an expert feature that requires writing a small amount of LISP code.)

(a)


(b)


(c)

Define Class

A THING with 2 nodes added, 2 components added, 4 wires added, and some graphic changes.

Redefine the class THING based on this instance

or

Define a new class named pulse-prolong

as a subclass of THING

and grouped under the heading Time-Tools

or

Cancel

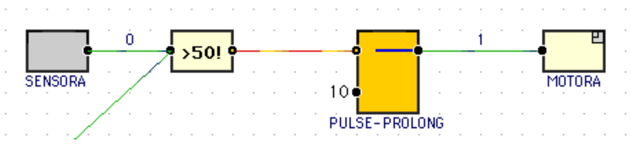
(d)


Fig. 7: Stages of encapsulation in Flogo I. (a) Put a box around working subdiagram, routing wires through input. (b) Object may now be closed. (c) Object becomes prototype for class definition. (d) A graphical interface may be added, using textual methods. [*picture cleanup pending*]

An important property of this sequence is that one has a functioning program at each step of the way. Code initially written and tested as a single construction can be encapsulated and generalized in place. It is reasonable to expect that this low abstraction barrier would both support understanding and encourage frequent use of encapsulation, but this remains a conjecture as it was not a focus of Flogo I user testing. However, Flogo I's encapsulation facility was used extensively by me and by the project's undergraduate assistants, and it served us well. (We used a combination of wiring encapsulation, as shown, with textual method definition, described below). For example, I was able to build Flogo I's interactive timeline editor (see "player" object, Fig. 8) in two hours one evening—in time to be used by children returning the next day. The programming is unusually simple because Flogo I handles the double-

editing any existing component. Another way to get some of the effects of subclassing is to create a new component which includes a subcomponent of the "superclass" type, adding transformations to its inputs and outputs, and shadowing external variables. Given these alternatives, as well as its conceptual complexity, inheritance seems not to be a vital feature for a language like Flogo I. A good system for parameter defaulting would probably be more useful.

buffered animation. All one has to do is straightforwardly describe how the interface should look—and, if desired, how the component should respond to the mouse—at each system tick. Fig. 8 shows some of the components, which have built in Flogo I using this facility.

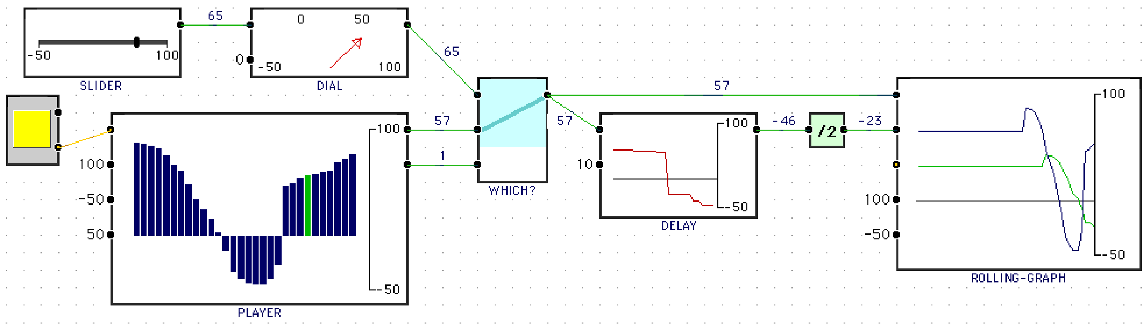


Fig. 8: Flogo I construction with examples of tool components with their own interfaces.

Step-by-step encapsulation, as shown above, depends in part on Flogo I's integrated display of subdiagrams. In most visual dataflow languages that allow encapsulation (e.g. LabView, ProGraph), the interior of an encapsulated diagram must be displayed in a separate window from the main diagram. This has several drawbacks. First, any editing action that brings up a separate window imposes a cost—in time, in the breaking of visual context and activity flow, and in subsequent window management. Second, it becomes difficult to watch the subdiagram's activity in the context of activity in the larger diagram: not only have the two become visually disconnected, but there is a good chance that one will be partially obscuring the other. And third, the encapsulation of part of a diagram becomes a choppier transition. In short, the use of separate windows for subdiagrams makes them harder to create, edit, test, observe, and understand. Flogo I is, to my knowledge, the only dataflow language in which components can be opened in place, without unnecessary disruption of the surrounding circuit.

Two features help to make this possible. First, Flogo I circuits can be displayed at a wide range of scales. By default, subdiagrams open up at about 75% of the scale of the containing view—still quite legible, but better able to fit into the surrounding context. Second, Flogo I takes special measures to ensure that components can be conveniently opened and closed without disrupting the arrangement of neighboring elements. When a component is opened, surrounding objects automatically move out of the way; when it is closed, they return to their previous positions. Positional adjustments propagate as far as necessary to avoid legibility problems—for example, preserving the left-right relationships of connected components, so that wires are not forced to angle "backwards." These measures succeed in being hardly noticeable: subdiagrams can be edited and tested in context, the cost of opening and closing a subcomponent is reduced to a single click, and the environment simply "does the right thing" to make room.

Uses of text in Flogo I

One of the design themes of Flogo I was to explore possible roles for text in a visual dataflow language. If it could be integrated in the right way, text might add convenience and clarity to some computations, and enlarge the language's expressive range. As Flogo I was taking shape, several opportunities arose to combine textual programming with data flow diagrams. These reached varying degrees of completion in the prototype.

Common to all uses of text is the Flogo I name space. The ability to refer to some values by name, rather than receive them via a wire, is sometimes a useful alternative. Names can help reduce clutter, especially for long-distance references; they create more options for diagram arrangement, and they have explanatory value as well. All nodes and subcomponents in a Flogo I component have names. (A third named entity, the method, is described below). Nodes are like variables, and a node's name can be used to refer to its value. The values at nodes are either typed in by the programmer (or user), received along a wire, or set by a component method (see below). Nodes outside the current component can be referred to by compound names that describe the path to that node. For example:

up.temp	refers to a node named temp, one level up in the enclosing diagram.
meter.out	refers to a node named out within a subcomponent named meter.
top.distthreshold	refers to a node named distthreshold that can be found within the topmost diagram. The top level can be a useful place to store important global parameters.[19]

These addressing methods can of course be combined, e.g. top.meter.out or up.up.preprocess.delay.buffer, but it seems unlikely that one would want to do this very often. The names of components have no other current use besides their role in addressing nodes and methods (see below), and as labels.

Node names can be referenced in all contexts where Flogo I allows textual code. There are three principal contexts: unary operations or "ops," lambda-style functional expressions or "funs", and component methods.

Unary Operations

The "op" is a very simple device, but quite a successful one. The op was used extensively by children and helped to meet a clear need that emerged in piloting. The initial problem is simply to make some arithmetic adjustment, such as subtracting 2 or dividing by 10, to a stream of data, often coming from a sensor. In the first Flogo workshop, I noticed that although children knew about Flogo's arithmetic components, they did not seem to think of using these in situations where one operand is a constant. Constructions of the sort shown in Fig. 10(a), for example, although very simple, were rarely originated by children. (I am speaking here of ten-year-old novices using Flogo for just two days. I expect that with more time they would have learned this construction as a standard idiom). They did not seem to fit the way children were envisioning the computation—that is, as a unary (and unitary) operation. A much better fit was provided by the "op" device, which allows one to type in simple half-expression such as +20 or /10 (Fig. 9). As a bonus, the op is more readable, more space-efficient, and quicker to install. (A similar device is found in the experimental language V — Auguston & Delgado, 1997).

[19] A technical note: the designation "top" need not be absolute. Any diagram can be declared to act like a "top" container from the point of view of all the subdiagrams and subsubdiagrams within it. The price of this, however, is that no nodes or methods above the level of this "top" container can be addressed from anywhere inside it: all information must arrive and leave via input and output nodes. A "top" component is thus lexically sealed off from its environment. A corrollary of this is that such a component could, in principle, reside on a separate device.

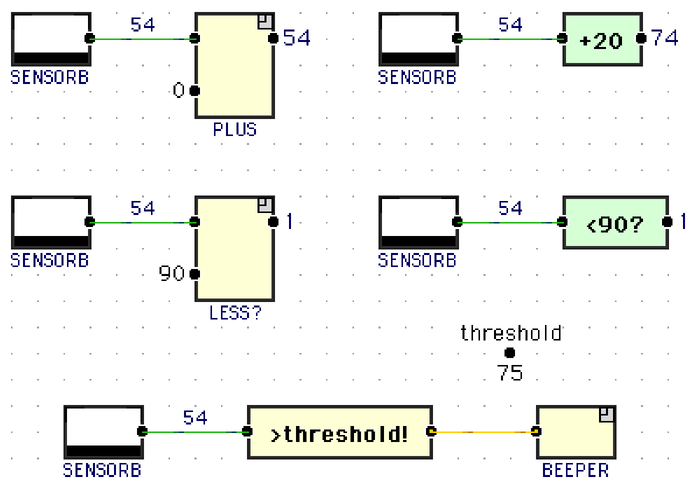


Fig. 9. "Ops" such as those labeled "+20" and "<90?" are easier to create and read than the traditional dataflow equivalents at left. Ops may also signal events, and refer to node values (bottom).

A few small extensions added to the power of the humble op. Comparison operators can be used, as can unary operators such as abs, sqrt, and not. Node values can be used instead of constants. Finally, by appending an exclamation mark to a comparison, one transforms the op into an event detector that will fire a pulse every time the comparison becomes true. This became the standard threshold detection device used throughout Flogo I programming.

Formula Components

The op's big brother is the functional component or "fun." The user can create an arbitrary number of named input nodes, and type in a formula that defines the output in terms of the inputs. The formula can also refer to nodes, as described above. This object is quite similar to LabView's "formula node." The "fun" was developed but not extensively tested.

Once debugged, the "fun" is intended to replace another formula device in Flogo I, the node formula. Like a spreadsheet cell, a node may contain either a static value, or a formula describing how its value should be derived. At each system tick, the formula is re-evaluated to generate an updated value for the node. The facility is useful, but confusing to read, because the formula is not visible. The "fun" improves on the formula node by making the formula visible.

Component Methods

In addition to its visible nodes, component, and wires, a Flogo I component can also have methods. These are not visible on the diagram itself (though it might be better of they were), but accessible via a small envelope icon at the component's upper left. Some component methods are invoked by the system. When a component is first created, if it has a method named "setup," it will be run at that time. If a component has a method named "go," that method will be called once per system tick. Methods can call other methods. For example, one common way to define a go method is

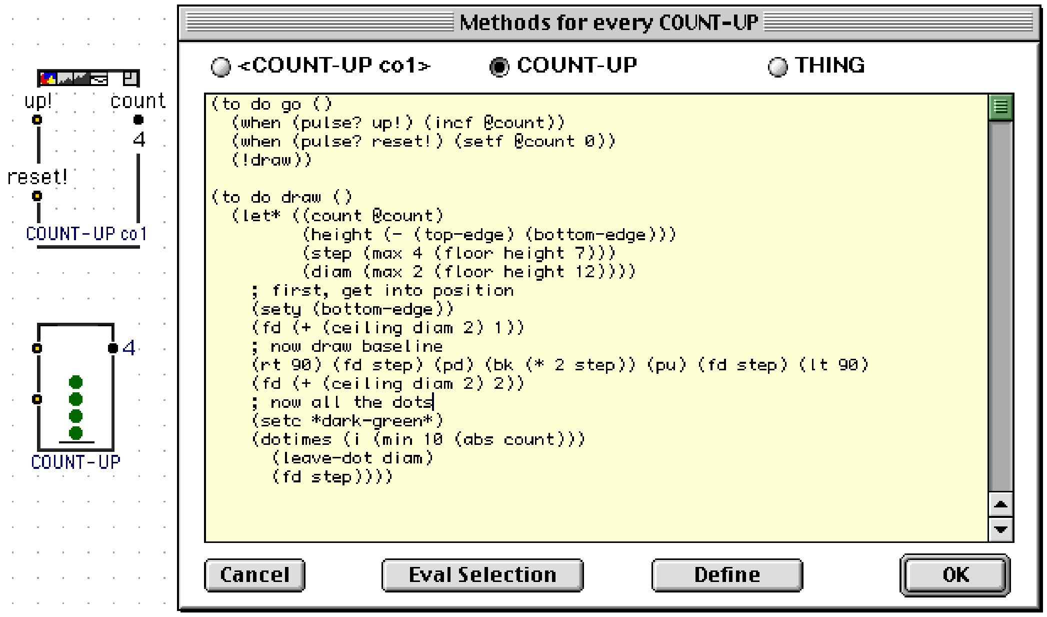
```
(to do go ()
    (!update)
    (!draw))
```

This will in turn invoke two separate methods, one to update the component's state, and another to update its appearance. Methods serve two main uses:

1. They provide the bottom layer of language definition. That is, components can be defined in terms of other components, and so on, but at the leaves of this tree of definitions, one will find textually defined behaviors. For example, the “count-up” component increases its counter each time it receives a pulse on its first input (up!), and resets the counter when it receives a pulse on its second input (reset!). This is achieved via the go method shown in Fig. 12.
2. They are the current means for defining component graphics and user interaction. For example, the code that generates the count-up’s graphic appearance is also shown in Fig. 12.

It is permissible, and occasionally useful, for a component to combine wiring-based and method-based computations. Other means of invoking methods from within a component as well as from outside it, were also envisioned but not prototyped.

As the examples reveal, methods are defined in a variant of LISP (called “flisp”), which includes special notational devices for method calls and node references. This was seen as a temporary stand-in for a future notational Flogo I language, syntactically similar to LOGO, which would be usable by children. The transition to the LOGO-like language would also have been the occasion to build true multitasking into the system. As it stands, methods are run to completion one after another. If any method takes a long time, the whole Flogo I environment waits for it. This was fine for prototyping—we simply wrote no slow methods. But the proper solution is to allow methods a limited number of operations per tick, and let unfinished methods continue at the next tick. (This approach has since been implemented in Flogo II).



```
(to do go ()
  (when (pulse? up!) (incf @count))
  (when (pulse? reset!) (setf @count 0))
  (!draw))

(to do draw ()
  (let* ((count @count)
         (height (- (top-edge) (bottom-edge)))
         (step (max 4 (floor height 7)))
         (diam (max 2 (floor height 12))))
    ; first, get into position
    (sety (bottom-edge))
    (fd (+ (ceiling diam 2) 1))
    ; now draw baseline
    (rt 90) (fd step) (pd) (bk (* 2 step)) (pu) (fd step) (lt 90)
    (fd (+ (ceiling diam 2) 2))
    ; now all the dots
    (setc *dark-green*)
    (dotimes (i (min 10 (abs count)))
      (leave-dot diam)
      (fd step))))
```

Fig. 12: the count-up component and its methods. Two count-ups are shown at left, one opened and one in the normal closed state. The dialog shows the methods for the count-up class. The go method provides basic functionality, incrementing and resetting the counter. The draw method generates the dynamic appearance shown at lower left. In the prototype, methods are defined in a variant of LISP. A more accessible, LOGO-like method language was ultimately envisioned.

How Flogo I differs from LabView

Given the family resemblance between Flogo I and LabView (National Instruments), it is important to note the major differences. While LabView is successful in scientific markets, efforts to use LabView in high schools or with novice students of programming have not taken hold (NI staff, personal communication). It appears that some kind of adjustment is needed to make the power of visual dataflow accessible to learners. The design of Flogo I represents a specific conjecture about what those adjustments might be.

Many of Flogo I's most important differences from LabView relate to ways in which Flogo I endeavors to provide an *open, responsive, and unified* programming environment:

Following one of the principles introduced in Chapter 1, Flogo I integrates code, data, and interface elements within a single space. Many objects under program control also accept user intervention. For example, a switch (Fig. 5) can be toggled by receiving a pulse on a wire, but also by a click of the mouse. In LabView, interface elements do not appear in wiring diagram, but are instead kept in a separate window.

As discussed above, Flogo I is able to display subdiagrams within diagrams rather than in a separate window. This allows subdiagram execution to be monitored and tested in context.

Flogo I programs normally reveal much about their state as they run. Data wires are dynamically labeled with their current values, pulse wires show the transmission of pulses, components show things such as sensor levels, time delay progress, counter status, switch states, and so on. In a special slowed-down mode, LabView can animate the movement of data through the program—a useful feature. However, during normal operation, a LabView wiring diagram is, by default, visually inert. Monitoring can be turned on for specific wires, but that requires the programmer to know what she is looking for. Moreover, LabView has no equivalent to Flogo's use of animation to reveal component state.

While a LabView program is running, one cannot modify it or intervene in its execution. In Flogo I, one can.

These differences all relate to the *liveness* of the programming environment. Tanimoto (1990) distinguishes four levels of liveness, or immediacy of semantic feedback, in programming languages (Fig. 14). According to this scheme, Flogo I is at level 4, because it not only provides immediate feedback to incremental changes in the program (level 3), but also an ongoing portrayal of the program's actions in response to input streams and user interface elements. LabView, because of its distinct modes for editing and running, achieves only level 2.

Flogo I's other main difference from LabView is at the paradigm level. Most dataflow languages, including LabView, represent the movement of data in terms of discrete data values. These little pellets of data are scrupulously accounted for. An operation will not fire until it has received exactly one value at each of its input notes; moreover, firing is delayed so long as any values previously output by the operation remain on the output wires. These rules, which were laid down in the first major paper on dataflow (Davis & Keller, 1982) and reflect dataflow's heritage in functional programming, enforce a strict interlocking of behavior of operations throughout the program. For Flogo's purposes, a looser coupling seemed appropriate. In Flogo I, a wire always has a current value associated with it. Outputs from operations are construed as updates to the wire's current value, rather than as discrete messages to operations downstream. Component computations are triggered not by the arrival of values, but by a system tick. Flogo I components are thus "always on."

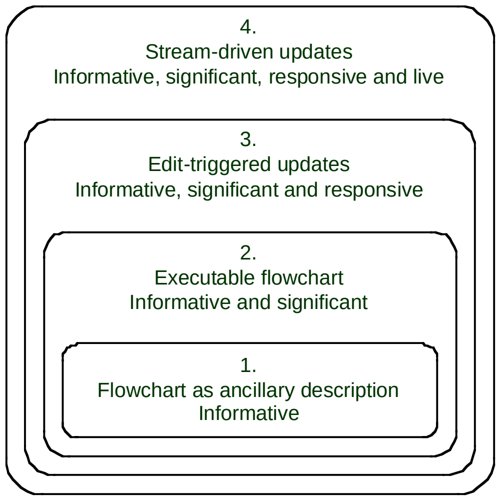

4.
Stream-driven updates
Informative, significant, responsive and live
3.
Edit-triggered updates
Informative, significant and responsive
2.
Executable flowchart
Informative and significant
1.
Flowchart as ancillary description
Informative

Fig. 14 (After Tanimoto, 1990). Tanimoto’s four levels of liveness in visual programming.

Flogo I’s approach has more in common with that advocated by Brooks (1991), who describes a robotics architecture in which components constantly transmit their current information, whether it is needed or not: this extravagance of transmission pays off in simplicity of architecture and elimination of tricky handshaking and coordination issues. Tanimoto’s proposed real-time video processing language VIVA (Tanimoto 1990), likewise adopts an electronic circuit metaphor, in which signals on wires are updated pseudo-continuously, and operators are “simultaneously and continually active.” Like Flogo I, these designs grow out of a primary concern with real-time applications. The pellet model can be used in real-time programming—LabView is a strong case in point. Nonetheless, real-time programming is a context in which this alternative paradigm—let us call it the “circuit model”—gains in interest. The potentially large cognitive impact of the circuit model’s “steady frame” is discussed in the next chapter.

Flogo I’s inclusion of pulse data is a consequence of its circuit model of dataflow. In LabView and other dataflow languages, data values serve a dual role of fixing parameters and triggering execution. Parts of programs can be activated by sending them data, and deactivated by cutting off their supply of data, and individual data values can be transmitted in order to trigger actions or processes. In Flogo I these roles are separated. Data wires carry continuously updated information, while pulse wires are used to signal events.

Chapter 4. Flogo I: Findings and Interpretations

Piloting of Flogo I yielded insights at several levels: about the real-time programming as an intellectual domain; about the kinds of knowledge that children bring to bear on this domain, and about the role of dynamic representations in supporting children's engagement and learning. Considering each of these in turn puts us in a better position to evaluate the strengths and weaknesses of Flogo I. The chapter concludes with reflections on the limitations of Flogo I's visual dataflow paradigm.

PILOTING OF FLOGO I

Flogo I was piloted with sixteen 10- and 11-year-olds with varied, but generally slight, previous experience of Logo Microworlds programming. Eight children participated in a two-day workshop. Eight more, plus one of the first group, participated in a four-day (20 hour) workshop which was preceded and followed by informal clinical sessions in which children, in pairs, were introduced to new language features and took on a variety of programming and problem-solving challenges.

Entitled "The Robotic Arts," the workshops attempted to support a variety of creative agendas, including those of children who might not be drawn to robotics in its more familiar forms. In conjunction with Flogo I, we used "cricket" control computers, part of a long lineage of robotics control devices developed at the Media Lab (Martin, Mikhak, et al., 2000). Physical constructions were built in Lego, with additional craft materials. As well as the customary robotic humanoids and vehicles, children built such things as a model fairground ride, a "gardener's helper" which spreads seeds by spinning, and assorted kinetic sculptures with interesting movement and light patterns. In the second workshop, the group collaborated to construct mechanized versions of scenes from a storybook (Seeger & Hays, 1994); these were used to make a video of the story.

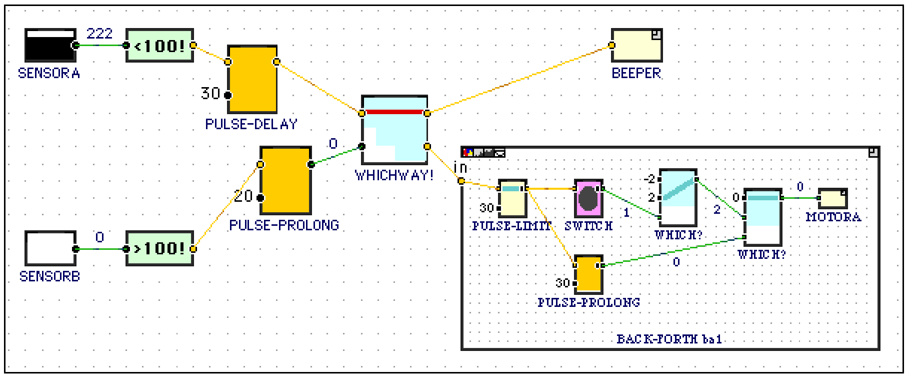


Fig. 1: An "Indiana Jones" puzzle. The challenge is to trigger the motor and open a secret compartment. (This is the second puzzle in a series of three.)

The children mostly used Flogo I to build small programs of their own. In addition, I occasionally presented them with small puzzles, both to probe their understanding and to offer another way to learn about the language. In a series of "Indiana Jones" puzzles, children were presented with a complete Flogo I program controlling a Lego treasure chest (Fig. 1 shows one of these puzzles). The challenge is to manipulate the two sensors in a way that opens the secret compartment. Often children can get the compartment open in a few seconds, simply by aggressive clicking and poking. Only after they do this is the full challenge explained: find a way to open the compartment without letting the beeper go off once (the beeper means Indiana Jones has been bitten by a snake), and whenever you're ready, you can play just once, "for keeps." These rules were readily agreed to and prompted a more careful investigation of the puzzle.

REFLECTIONS ON THE DOMAIN: PULSE VS. DATA IN FLOGO I

Let us look first at what children's use of Flogo I helped to reveal about the intellectual substance of real-time programming. Of course, in robotics there is plenty of substance to choose from, and some of it was on display in Flogo I piloting. To pick one example, children often struggled when confronted with the need to map data values from one scale into another, e.g. a sensor value between 0 and 255 translated to a motor output between 0 and 7 (or worse, –7 and +7). But one issue stood out as particularly related to the problems of real-time control, and that is the focus of this section.

In the preceding chapter I outlined Flogo I's distinction between two kinds of signals. The *data* signal is a steady stream of information representing the variation of some quantity over time. A *pulse* is an instantaneous signal carrying no informational content other than the moment at which it occurs. Pulses are used to report events and to initiate actions.

Children were not always clear on the distinction between pulse and data. A common context for the problem is in situations where the program detects a sensor crossing a threshold value. It is easy to confuse the *event* of the sensor value crossing over the threshold with the *span of time* during which the sensor stays over the threshold value.

So, for example, when students confronted the mystery program in Fig. 1, they would expect the "Which" component to remain in the downward position as long as sensor B was pressed. This would happen while students were actually playing with the program, and manipulating the sensors. They could observe a relationship between the switch position and the sensor's value going under the threshold. They would say things like "the switch needs the sensor to be below 100," and make corresponding action errors, holding down the sensor value well in advance, instead of timing the press. "Backsliding" was observed: when asked to look closely, a subject might correctly state that the pulse happens when the sensor value crosses over, but then repeat earlier errors. Also observed was correct action together with incorrect description: children saying that the switch is open while the sensor value is below 100, but in fact timing their movements to make the value cross under at just the right time.

Related errors occurred in other contexts:

- using the term "pulse" as the equivalent of "true." For example, "This one is more so it sends a pulse to the switch"
- when a pulse was used to start a component's action—initially believing that the path taken by the initiating pulse needed to remain open for duration of the action (whereas it only needs to get through once, and the action will proceed).
- indiscriminate wiring of pulse nodes to data nodes, and vice versa.

There are several layers to sort out in this phenomenon. Of course, part of it is kids' understanding of the pulse / data distinction in Flogo I's ontology. But it also appears that kids are not practiced in distinguishing events and durations when analyzing real-world processes.

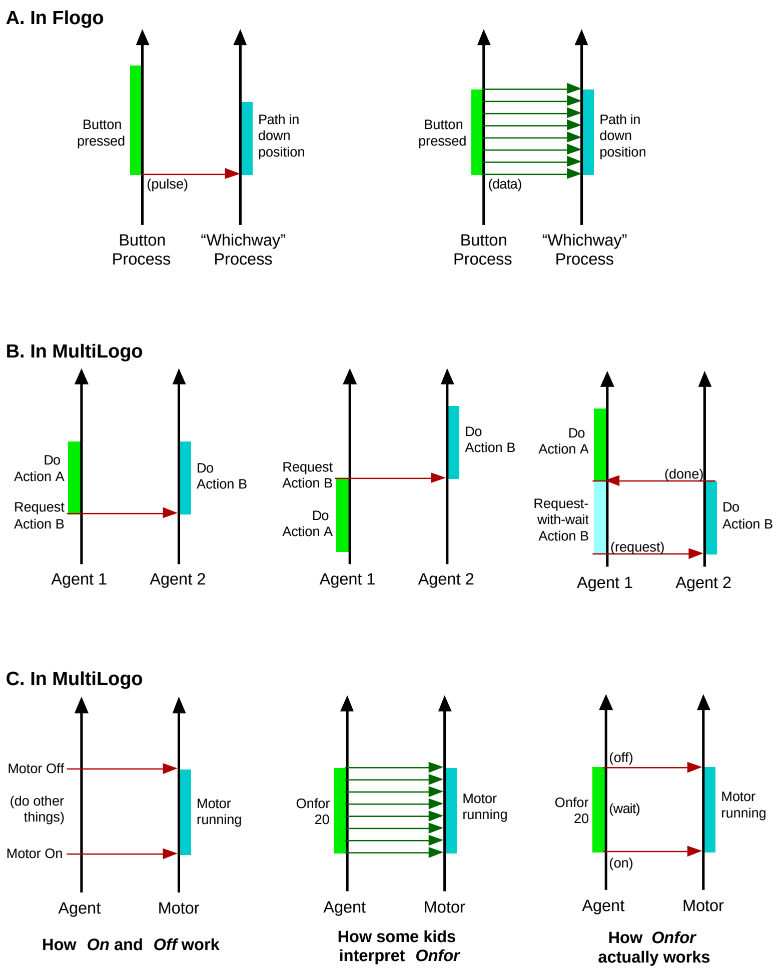


Fig. 2: Flogo's pulse/data distinction compared to process coordination in MultiLogo (Resnick, 1991). Upward arrows indicate the passage of time in distinct threads of computation. A. On left, actual event-triggered relationship between an input button and a "WHICHWAY" switch in one of the Indiana Jones puzzles; on right, a recurring misconception that the switch stays on as long as the button. B. In MultiLogo, learners may not recognize major differences in execution depending on the order and type of interprocess calls. C. If a process is asked to run a motor for a specified time, but then aborted, students expect the motor to stop; in fact, it will run forever.

The distinction between events and durations is an important part of understanding how to use multiprocess programming languages. In a study of children's understanding of MultiLogo, a text-based concurrent programming language, Resnick (1990) observed a variety of "synchronization bugs," in which children either expected that one agent, having asked another agent to do an action, would wait until the action was finished (whereas in fact it would start immediately), or, conversely, that an agent could start a time-consuming action of its own and then ask another agent to do something in parallel (whereas in fact the second agent doesn't get asked until the action is finished). Again, the difficulty is in the general area of sorting out instantaneous versus time-consuming actions, as they are coordinated among different processes. Children also had trouble with agents executing the “onfor” command. They thought that halting the agent would also halt the motor (whereas in fact, halting the agent before it could turn off the motor led to the motor staying on indefinitely). Here, children fail to realize that an apparently continuous engagement of two processes is actually nothing more than two instantaneous interactions. The drawings in Fig. 2 suggest the relatedness of all these confusions.

As described so far, the event-duration and pulse-data distinctions, and their cousins the synchronization bugs, look like issues specific to programming in parallel or multiprocess languages. And they are indeed more visible there. But similar phenomena occur in real-time programming with procedural languages. Fred Martin (personal communication) has observed many difficulties with what he calls (borrowing from related electrical engineering terminology) the “edge/level distinction,” among schoolchildren and college students, programming robots in a variety of procedural languages. Martin has found that students’ robotics programs often reveal a conflation of situations defined by a condition being true, with moments at which a condition *becomes* true.

We can also note the relevance of a well-known misconception of novice programmers, that a while loop will exit anytime its looping condition becomes false, whereas the condition is actually checked just once per iteration. In other words, does the condition apply over the *duration* of the loop, or by means of discrete condition-checking *events*. So even when there really is only one process, the naïve expectation seems to imply multiple processes, so process coordination is still an issue!

Finally, beyond the realm of programming, it is interesting to note a cognate phenomenon in the realm of music analysis. Bamberger (1991) has investigated in depth a dichotomy in children’s invented representations of rhythm. For example, she would have students clap the rhythm corresponding to the verse fragment “five, six, pick up sticks; seven, eight, lay them straight” (Note: I am providing this verse fragment as a way to communicate the rhythm to the reader. In her studies, Bamberger simply clapped it.) Over a wide range of ages, responses often fall into two categories, either:

O O o o o O O o o o

…or:

O O o o O O O o o O

Bamberger calls the first representation figural, the second formal. She reports that producers of one kind of representation have considerable difficulty seeing any merit in the other representation. An appreciation of both representations and the relationship between them is generally arrived at only through an extended process of dialog and reflection, often spanning days if not weeks.

A full interpretation of these representations entails many subtle issues in musical cognition, and will not be attempted here. What is not too hard to discern, however, is a difference in the way the two representations reflect temporal duration. The lower variant (“formal”) consistently describes each clap in terms of the time duration to the next clap. It represents the rhythm as a series of temporal durations. In the upper form, each circle seems more connected to something qualitative about the clap itself, rather than to the duration; hence the inconsistent correspondence between circle size and temporal interval.

Music, like robot behavior, is something that unfolds over time. In trying to describe a temporal process, music students confront a similar problem: integrating event-oriented and duration-oriented information about the process. Both are essential—each notation, above, is only a partial description of the rhythm—yet we have a tendency to try to describe both kinds of information with the same representational tools. This is a problem that any approach to real-time programming, and the learning of real-time programming, must address.

REFLECTIONS ON LEARNING: THE ROLE OF ACTION KNOWLEDGE AND TINKERING

DiSessa (1986) has described how programming competence can be based on a patchwork of partial understandings. The understandings include structural knowledge (knowing something about the mechanics of computation, and the ability to predict, on this basis, how some computations will run) and functional knowledge (context-based "how to" knowledge for achieving desired ends), as well as schemas and expectations carried over from other domains. Development as a programmer will entail expansion, interconnection and reconciliation of these partial understandings, including both structural and functional understandings.

These types of knowledge were apparent in children's use of Flogo I. Here is an example of functional knowledge. All the children eventually learned that if you want a button to turn a motor on indefinitely, instead of just as long as the button is pressed, then you need to put a switch between them. This knowledge worked for them – they could use it to build working programs – even though most were hazy on what kind input the switch takes (pulses that say when to turn on and off) and what kind of output it generates (data about the current on / off status).

Now, what kinds of knowledge, and what ways of gaining knowledge, are we adding to the mix when children have a completely new way to interact with programs? Consider the Indiana Jones puzzles mentioned earlier. As I said, the challenge of playing for keeps, which requires the ability to open the compartment reliably, with no missteps, encouraged careful investigation by the children. However, it would be a mistake to call this investigation a mechanical analysis of the puzzle, by which I mean a tracing of cause and effect based on structural understanding of the primitives. There is some mechanical thinking (varying from learner to learner), but at least as important is a process of gradually identifying useful visual cues, and working out how to coordinate one's actions with these cues.

For example, in solving another Indiana Jones puzzle, one boy developed a mantra, "one...two...three… send it across" – a terse guide to what to do and what to pay attention to in the program. The slow counting marked out how long a particular touch sensor needed to be pressed; the "it" that was sent across was a pulse that needed to get to a target part of the program, past some logical obstacles which the "one…two…three" action would temporarily clear. The word "send" marked the moment at which this pulse should be launched, by touching a second sensor. By borrowing the mantra, his partner, who had made even fewer mechanical interpretations of the puzzle, began to be able to open the compartment as well.

Flogo I's live, graphical interface, I suggest, brings into play a third kind of knowledge and learning: not structural, not functional, but based in action. Action knowledge— rhythms, timings, movements, alertness to specific cues—of course plays a large role elsewhere in children's lives. What is its potential value in programming? By developing their action knowledge of Flogo I, children were

1. Building a practical, action-oriented sense of program processes which can be gradually coordinated with the other kinds of knowledge

2. Honing skills in program reading—not just reading the static program, but “reading” its execution process. When there is a lot going on simultaneously, choosing what to watch is a nontrivial skill.
3. Developing the practice of experimenting with program components and working them for desired effects.

Tinkering

The term “tinkering” has been used in the literature of constructionism to describe an open-ended, exploratory process of programming (Turkle & Papert, 1992; Turkle, 1995). Like the *bricolage* of Levi-Strauss (1966), programming-by-tinkering is a process guided neither by well-formulated plans nor by firm goals. Instead, the tinkerer responds rapidly and flexibly to emerging results of partial constructions, and may arrive at an end product very different from that initially envisioned—if an end product was envisioned at all. Programmers with a solid structural understanding of the programming language may sometimes program by tinkering. However, for the many learners whose structural understanding is not fully developed, tinkering is an approach that helps to leverage satisfying results from incomplete knowledge.

For example, consider the big box, called a “back-forth”, in the lower left of the Indiana Jones program (Fig. 1). One pulse into this box will make the compartment open; the next one will make it close, and so on. Most kids did not try to analyze how it worked, but they knew what it did, and that it was the target in these puzzles. Based only on this exposure, two pairs spontaneously used a back-forth in their own constructions: one to make a car drive back and forth; and the other to make two arms of a kinetic sculpture wave one way, then the other. (Note that these are different from the original application, so it appears that they had abstracted the mechanical essence of the back-forth.)

Figure 3 shows one of these appropriations of the back-forth, for driving a kinetic sculpture. The pulse-repeat sends a pulse to the back-forth once every second (the 10 is in units of 1/10 second). Inside the back-forth, the first component has the job of ignoring any pulses that follow within 3-1/2 seconds of the previous pulse. (This prevents the movement from being interrupted in midcourse.) So only one pulse in four gets through. These details presumably escaped the notice of the programmer, who was happy with the overall behavior.

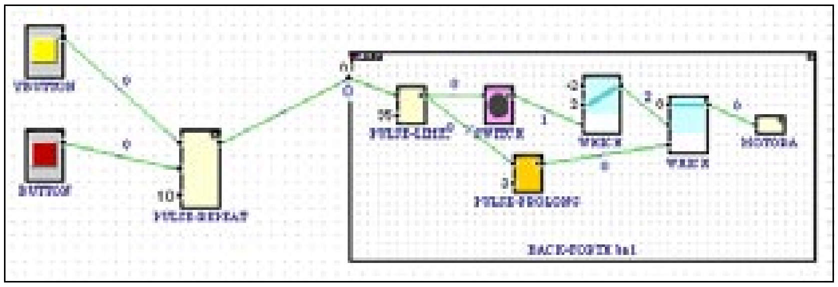


Fig. 3. Example of programming by tinkering: Successfully re-using a Back-Forth in a non-standard way (three out of four input pulses will be ignored).

The fact that the programmer was able to get a successful program working without understanding everything about how it worked, through a process of tinkering, is a good thing. Turkle and Papert (1992) defend programming-by-tinkering as a legitimate manifestation of cognitive style. But tinkering can also play an important role as a leading edge for learning. Tinkering gives learners a chance to succeed, and to gain various kinds of knowledge and experience, while their structural knowledge and planning skills are still fragmentary. Structural knowledge gains not only time to develop, but also a framework of other kinds of knowledge with which it can be "webbed" (Noss & Hoyles, 1997). Programming will become more learnable as we make its deep ideas more accessible, but also as we allow children more time and context to learn them, by supporting tinkering.

Some may worry that too much tinkering may slow learning, by removing the necessity of confronting the deeper structural ideas of the language. I agree that educators need to consider moments and methods to provoke a shift from tinkering to deeper reflection. But this is not an argument against tinkering. It is tinkering itself, after all, that helps to lay a foundation, sometimes over very long time periods, for such teachable (or thinkable) moments. No matter how well one tinkers, structural knowledge can still help one get farther faster. The fact that structural knowledge is empowering implies that children will still value it, and that ways will still be found to promote it.

If we take support for tinkering to be desirable in a programming language, what are some of the characteristics that would contribute to this support? One of them would surely be the ability to test programs, and parts of programs, in the live way that kids investigated the Indiana Jones puzzles. "Action learning" about these components would be one way for the tinkerer to sort out what aspects of execution to pay attention to, and how to adjust for them.

Flogo I has a second language property that appears to support tinkering: a spatial layout of program components in which distinct regions interfere with each other relatively little. Functional and non-functional parts of a program can coexist in the same program: the non-functional part does not prevent the functional part from working. This property can greatly reduce the price of experimentation. It is easy to add new components and wires to a program, without fear that they will introduce damaging side effects. Fig. 4 shows a snapshot of an extended programming session. The functional core of the program is surrounded by partial constructions, some abandoned, others in process. The children worked iteratively, making a few modifications (some planful and some haphazard), testing them onscreen or via the robot, interpreting the results (in terms of structure, or just success), and modifying again. Because they never had to "break" the working parts of the program in order to try new ideas, forward and lateral progress far outweighed regress. By the end of the session, real progress had been made.

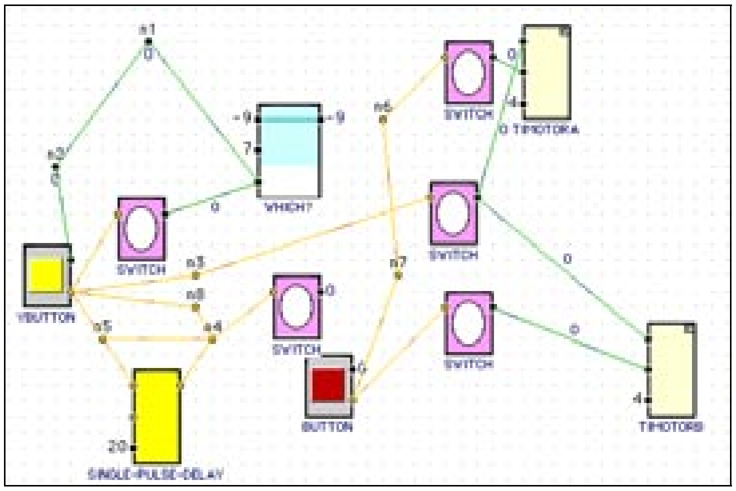


Fig. 4. Another tinkering example. Work-in-progress shows non-functional, experimental and test code coexisting with functional core.

To put in another way, I am suggesting that Flogo I's "live" dataflow representations help to make programs less fragile. The principle could be summed up as follows: if your program behaves *as* you build it, and if each part behaves *where* you built it, then the program breaks less when you tinker with it—and when it does break, it will be easier to do something about it.

A third way that a language can support tinkering is by closing off false leads in the exploratory process. Flogo treatment of data vs. pulse signals illustrates this. During most of the time that children used Flogo I, the difference between data and pulse was simply a matter of interpretation. A pulse was defined as the moment of a signal's transition from 0 to a positive number. There were a number of situations where one could conveniently connect a signal that, conceptually, carries data, to a node where pulses are expected (e.g. connecting a touch sensor component directly to a beeper, or to a "player"—Fig. 3-9). But for every situation in which mixing pulse and data conveniently did the right thing, there were many opportunities for children to connect the wrong nodes—through confusion, by accident, or as a way to explore—and by mystified by the results.

After seeing children's difficulties with pulses and data, I changed Flogo I to make the distinction more explicit. Pulse and data nodes look different, as do the wires between them, and Flogo I refuses to connect a data node to a pulse node. I thought that this might force the issue of pulse vs. data and get kids thinking and talking more about the difference. In a follow-up session with two boys, I presented and explained the new system. Without further discussion of the nature of pulse and data the boys proceeded confidently on their project. Finding, as time went by, that some attempted wirings were now prevented, one of the boys good-naturedly remarked, "these orange nodes are really messing us up… they make it harder." In fact, though, only dead ends were thwarted, and I believe they got their constructions working more quickly than they would have with the earlier version. A language change developed with structural understanding in mind thus turned out, in the short term, to have a different kind of value. With no immediate effect on structural understanding, it nevertheless helped to make the tinkerers' search for useful behaviors more efficient. (Meanwhile, the potential value for structural understanding over the longer term is not ruled out).

REFLECTIONS ON MEDIA: THE STEADY FRAME

The design of Flogo I placed a high priority on "liveness," and, as recounted above, the effort seems to have paid off in opening up new ways for children to approach programming. At this point, the concept of liveness merits a closer look. As we saw above, liveness is defined as a feedback property of a programming environment, and this is the way it is commonly discussed in the literature (e.g. Burnett et al., 1998). And yet, if making a live programming environment were simply a matter of adding "continuous feedback," there would surely be many more live programming languages than we now have. As a thought experiment, imagine taking Logo (or Java, or BASIC, or LISP) as is, and attempting to build a live programming environment around it. Some parts of the problem are hard but solvable, e.g. keeping track of which pieces of code have been successfully parsed and are ready to run, and which ones haven't. Others just don't make sense: what does it mean for a line of Java or C, say a = a + 1, to start working as soon as I put it in? Perhaps I could test the line right away—many programming environments allow that. But the line doesn't make much sense in isolation, and in the context of the whole program, it probably isn't time to run it. In a sense, even if the programming environment is live, the code itself is not. It actually runs for only an infinitesimal fraction of the time that the program runs.

The possibility of meaningful liveness depends on the structure of the way that computation is represented. This section asks: what are the properties of representations that help to make liveness possible? Since programming languages are rather complicated things, we start with a simpler example.

Gregory Bateson (1980) offered a contrast between two kinds of feedback loop. As an example he compares the process of hitting a target with a rifle versus hitting it a machine gun. I find the contrast even clearer between a bow-and-arrow and a water hose, so let me describe it that way. Hitting a bull's eye with an arrow is a careful process: you aim, shoot, string another arrow, carefully correct the aim, shoot again, and so on. It is difficult enough that archery is an Olympic event—a distinction unlikely to be accorded to the challenge of hitting a target with water from a hose. What is it about a hose that make it so much easier to aim? The evident difference is "continuous feedback": as we adjust which way the hose is pointed, the stream of water from us toward the target reflects back as an uninterrupted stream of information about the consequences of our aim. But how does this help? A reasonable first explanation might be that more, faster feedback simply allows us to compress what the archer does into a much shorter time. The rate of feedback surely matters, but does not fully explain the difference. The archer and the waterer are engaged in qualitatively different processes, and the key to the difference is this: the waterer *never stops aiming*. The only movements the waterer needs to make are corrections to a single, evolving aim.

The archer, on the other hand, makes a series of distinct aiming actions. Each time she makes a new shot, she fits a new arrow to the bow, and then undertakes the feat of reconstructing how she aimed the previous shot. This is an essential step, because her previous aim is the basis for any correction she might make. Reconstruction requires a tremendous amount of concentration and skill. It has to be very precise: any correction she makes will be only as good as her approximation of the previous aim. Reconstructing previous shots is a substantially larger source of error in archery than making the right correction (it has to be, otherwise, having hit the bull's-eye once, it would be easy to keep hitting it[20]). Correspondingly, many of the training efforts and equipment advances in archery can be understood as promoting the repeatability of shots.

For the waterer, reconstruction is a non-problem: there is no need to "reload" the hose, the way an archer sets up the next arrow. There is a requirement to maintain the current aim, but this is much more manageable, and maintenance is assisted by availability continuous feedback. If my arm starts to droop, I'll see the stream of water start to drift off that bush at the other end of the flowerbed, and I will correct without even thinking about it.

In other words, continuous feedback only makes sense in response to a continuous action. The archer's firing is not continuous; rather, it is made up of a series of actions that unfold over time; having occurred, they disappear. Continuous feedback is not simply absent: it is meaningless. In watering with a hose, on the other hand, the "shot" has escaped its temporal embedding. An action sequence has been converted to a stable entity—to wit, the holding of a hose nozzle—and the ephemeral sequence of aim, fire, result, has been replaced by a persistent relationship, visibly rendered by an actual arc of water, between nozzle position and where the water is landing.

[20] Cannon ballistics is a bit more like this, because the cannon's aim is mechanically maintained from one shot to the next.

In comparison to archery, hosing creates what I propose to call a *steady frame* for the activity of aiming for a target. A steady frame is a way of organizing and representing a system or activity, such that

(i) relevant variables can be seen and / or manipulated at specific locations within the scene (the framing part), and

(ii) these variables are defined and presented so as to be constantly present and constantly meaningful (the steady part).

If we take the not unreasonable position that aim is a highly relevant variable in aiming, then this definition tells us that hosing is a steady frame for aiming, because aim is constantly present and meaningful via the hose nozzle, while archery is not, because aim is only intermittently represented by the bow and arrow.

Of course, the use of the term "relevant" opens up room for qualitative variation in steady frames, as well as for disagreement. One steady frame may be better than another if it presents more variables that are relevant, or variables that are more relevant, than those offered by the other. There can also be qualitative differences in the ways that variables are represented. For example, a set of knobs to control a hose's aim would be steady, but far less easy to work with.

Aim is not the only relevant variable in aiming. The spot that the arrow or stream of water travels to—let us call it the aim outcome—is just as important. And in general, in any dynamic system or activity we will be interested in relationships among two or more variables, and in ways that some variables can be manipulated to achieve desired effects in other variables. The qualitative judgment of variable relevance depends on this web of relationships. If a variable has no intelligible or usable relationships with any other relevant variables in a system, then we are unlikely to judge it a relevant variable. The relationship between aim and aim outcome is a very well-behaved one—a continuous mathematical function with few reversals—and this too contributes to the value of these variables in working with the system.

The qualitative judgment of variable *representations* also has a basis in these relationships. We will prefer representations that make relationships easier to perceive. For example, if I am unfortunately forced to control hose aim remotely via knobs that operate like polar coordinates, and if I can't see where the water lands with respect to the target rosebush, but must instead look at readout dials showing this information, then it is more helpful if these dials show aim outcome in polar coordinates with respect to the bush, as opposed to x / y offsets from the bush, because these will be easier to correlate with my aim inputs. Best of all, of course, is holding the hose and watching the water. Then my perspective yields a very simple mapping: turning the nozzle to the left makes the water land a little more to the left, and so on. Moreover, the arc of water between my hand and the bush helps me to track the time lag in the aim→outcome relationship, making it easier for me to avoid overcorrections and other lag-related difficulties.

In short: if we are interested in exploring dynamic systems through real-time feedback, many factors will affect the quality of our experience. The notion of the steady frame emphasizes one factor that is, in a sense, a prerequisite for the rest—namely, the structuring of the system in terms of true dynamic variables. If a system that we organize or apprehend in terms of action and event sequences can be recast in terms of persistent variables and relationships, a new space of representational possibilities opens up.

Another example: the oscilloscope.

The oscilloscope's ability to make high-frequency wave patterns visible is based on the way it maps a signal onto two-dimensional space. As the oscilloscope constantly retraces the signal across the display, patterns matching the oscilloscope's retrace frequency become visible. Each phase of the signal, with respect to the tuned frequency, appears within its proper vertical slice in the display. The result is a steady frame that allows us to observe waveform variations in real time. Note that the variables made steady by an oscilloscope are, in a sense, virtual constructs. Consider a single vertical slice of a sound wave pattern: signal level at, say, offset +1 mSec. At a fine enough time scale, we see that this seemingly constant variable is actually based on an intermittent series of momentary values that occur once per oscillation. The phosphorus of the oscilloscope screen and the receptors of the retina work together to maintain our perception of the variable, and indeed the whole waveform, across these time intervals. The waveform presented by an oscilloscope is thus not a given of nature. It is an abstraction, a construct that recasts sequential events in a steady frame—one that has proven indispensable for all kinds of scientific and technical work.

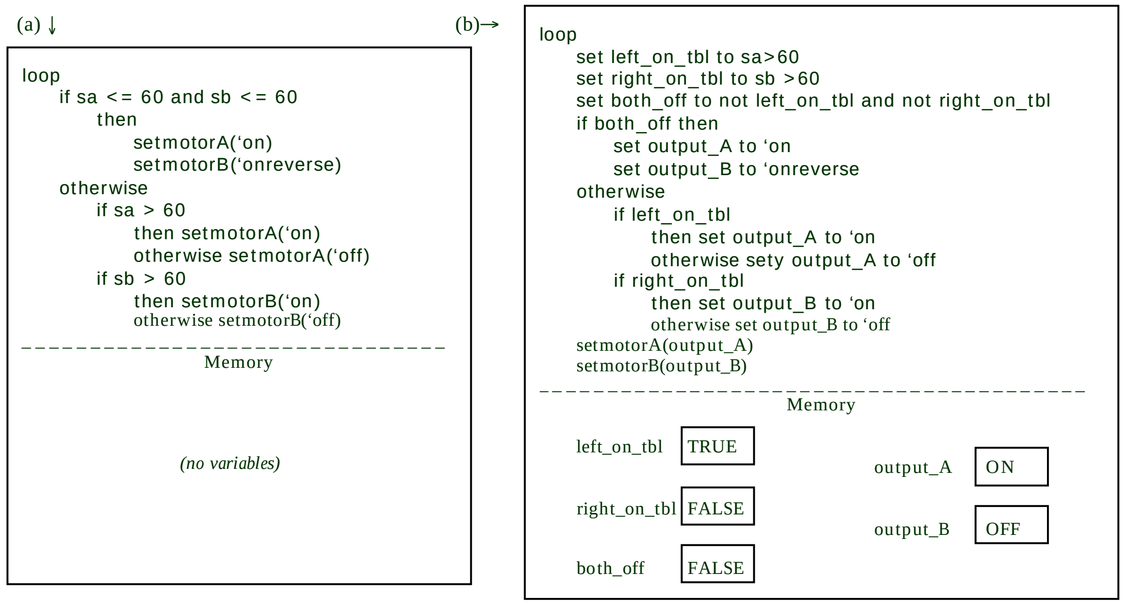
The oscilloscope lets us look at just one frequency at a time (or a few, counting harmonics). Another steady frame device, the real-time Fourier spectrum graph, is able to reveal a whole range of frequencies at once, using a different spatial organization—the trade-off being that for each frequency we see only an amplitude, not a waveform.

Dataflow as a Steady Frame

Returning our attention to programming languages, we can see visual dataflow, and especially Flogo I's circuit-style dataflow, providing a steady frame at two levels.

The first and most significant steady frame is at the level of program execution. As the program runs, we can see input values and derived values changing. Each one is constantly present, each is localized to a wire or component, and each has a visible derivation, as well as a constant interpretation, such as "value from sensor B," "whether the left side is off the table," "whether motor A should go backwards." Because these variables are held steady, we can (given a live programming environment) experiment with them. For example, we can move the robot so that its left sensor crosses back and forth over the table' edge, and watch the effect anywhere in the program.

Just to remind ourselves that this orderly access to a dynamic process is not to be taken for granted, consider some alternative frames for the same information. Suppose we could look inside the computer's CPU as the program ran. All the same data values would pass through at various times, but since multiple quantities use the same space, the effect is a complete jumble. Or imagine watching a LOGO or C program execute. Perhaps the lines light up as they are executed, but this is hard to make sense of as execution flashes by at incomprehensible speed. If we can see into memory and watch the values of variables, we are doing better—but only to the extent that the programmer has chosen to store relevant computational results in variables with steady meanings (an issue we will return to in a later chapter). Almost certainly there are important parts of the program's computation that are not reflected in variable values, but rather in other aspects of program state, such as which clause of a conditional one is in, or the values of transiently evaluated expressions.



(a) ↓

```
loop
    if sa <= 60 and sb <= 60
        then
            setmotorA('on)
            setmotorB('onreverse)
    otherwise
        if sa > 60
            then setmotorA('on)
            otherwise setmotorA('off)
        if sb > 60
            then setmotorB('on)
            otherwise setmotorB('off)
_________________________________
              Memory

          (no variables)
```

(b)→

```
loop
    set left_on_tbl to sa>60
    set right_on_tbl to sb >60
    set both_off to not left_on_tbl and not right_on_tbl
    if both_off then
        set output_A to 'on
        set output_B to 'onreverse
    otherwise
        if left_on_tbl
            then set output_A to 'on
            otherwise sety output_A to 'off
        if right_on_tbl
            then set output_B to 'on
            otherwise set output_B to 'off
    setmotorA(output_A)
    setmotorB(output_B)
___________________________________________
                  Memory

    left_on_tbl   TRUE
                                output_A   ON
    right_on_tbl  FALSE
                                output_B   OFF
    both_off      FALSE
```

Fig. 6.Two procedural programs that compute the equivalent of the Flogo I "Table Explorer" program (Fig. 3-1). Program (b) creates a steadier frame by following a dataflow style of coding, to an unusual degree.

If the programmer has chosen to imitate a dataflow style of programming very faithfully within the procedural language (as in Fig. 6b), then a memory display will give us quite a lot of usable real-time information. However, there are still some problems. First, it takes extra work to match up the variable's value with the code that set it. This is already a problem in Fig. 6 and it increases as the program gets bigger. In a dataflow program we don't have this problem, because data and computation are overlaid. Second, variable values are only part of the story of this program's computation. The other part is control flow. To understand what this program is doing, we need to be able to trace the pattern of flow of execution through the program as external conditions change. Let us suppose that, in our hypothetical programming environment, lines of code are highlighted whenever they are executed, and furthermore the highlighting persists longer than the nanosecond it actually took to execute, so we can see it. With practice we can learn to use this information, but it still requires detective work, because we only see lines flickering here and there, showing that they have been executed recently. We do not have a stable, localized representation (in other words, a steady frame) for the actual path that computation is following through the program.[21] In Flogo I's circuit-like form of dataflow, this is less of an issue, because for components that are always on, there is no control flow to worry about. Deciphering pulse pathways can be analogously tricky, but the use of pulses in a Flogo I program is relatively compartmentalized compared to the way control flows throughout a procedural program, so the problem is usually smaller.

Beyond the level of program execution, Flogo I also provides a steady frame for program development and testing, for reasons discussed earlier: the overlay of computational results directly on the program; the immediate response to program edits; and the localization of computational effects, reducing brittleness. A particularly vivid demonstration of the steadiness of this frame was achieved using

[21] Whether such a representation is possible in principle—perhaps in some oscilloscope-like fashion—is an interesting question that will not be pursued here.

the video record of the computer screen during a 40-minute programming session in which two boys worked on a robot behavior. Speeding up the video by a factor of 40, one can see the program gradually take shape: a central core, then various offshoots, some of which are deleted, others that remain. The fact that this video is intelligible at all at high speed says something very important about the work that Flogo I's representation is accomplishing in holding the boys' project steady for them while they think about it and tinker with it.

To summarize: for real-time applications such as the Table Explorer, Flogo I's live rendition of circuit-style dataflow makes the dynamics of program behavior more accessible, by providing a steady representational frame for the variation of program data, for the relationship between program code and computational effects, and for the development process of the program as a whole. These factors combine to make real-time programming in Flogo I a bit less like archery and a bit more like wielding a garden hose.

Let me be clear, however, that the foregoing is not meant as a case for dataflow programming. Rather, it is an analysis of one specific strength of dataflow representations, which must of course be weighed along with other considerations. Not all real-time programming problems match up with Flogo I's strengths so well. As the next section will explain, when the robot is expected to manage a repertoire of behaviors, Flogo I's steadiness can become too much of a good thing.

REFLECTIONS ON LANGUAGE DESIGN: LIMITS OF EXPRESSIVITY IN FLOGO I

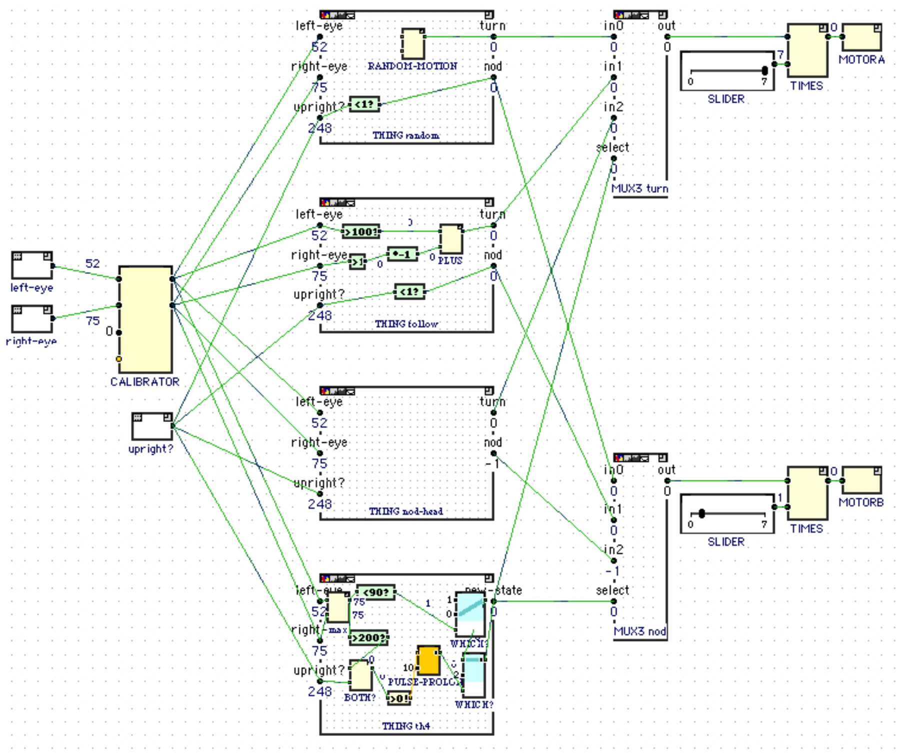


Fig. 7: A Flogo I program to control a Lego bird.

Fig. 7 shows the largest program written in Flogo I to date. Written by an undergraduate project assistant, it controls a model Lego bird equipped with two distance sensors in place of eyes, and motors and sensors to help it turn in place, and lower and raise its neck in a pecking motion. In the absence of stimulation, the bird scans its environment lazily, turning for a while, resting for a while. When a nearby object, such as a person, is detected, the bird now tracks the object's movement, endeavoring to continue facing toward it. If the object comes even closer—for example, if one holds out one's hand as though offering birdseed—the bird makes a pecking motion.

Without going into the details of the code, one can see a general flow: from the sensors at left, branching out to four boxed subdiagrams, then cross-fanning into two large switching boxes which in turn drive the motor outputs. The four major subdiagrams include one each for the bird's three distinct behaviors—scanning, tracking, and pecking—plus, at bottom, a control module with the task of deciding which of the three behaviors to deploy. What is noteworthy in this program is that, as a corollary of Flogo I's "always on" computational model, the circuitry for all three behaviors is, indeed, always on, and always generating motor outputs. The outputs actually used are selected downstream, at which point the rest are discarded.

The general pattern of modules that remain active even as they gain and lose control through downstream selection is a recurring and well-supported theme in the behavioral sciences and in artificial intelligence (Fodor, 1983; Minsky, 1986; Brooks, 1991). However, a language in which downstream selection is the only way of shifting control among different circuits can create difficulties for novice programmers, and for programmers attempting to maintain simplicity and understandability as their programs grow.

Distinguishing behaviors: the MOTORS-SHARE device

As this issue gradually came into focus, I came to see the Table Explorer example (see Figs. 3-1 and 3-2) as a case in point. The second Tabletop Explorer program is best thought of as performing two distinct behaviors: usually it (i) moves forward, using wheel inhibition to turn away from the table's edge; but if both sensors go off the table, it (ii) reverses one wheel in order to turn away from the edge. The program in Fig. 3-2 actually makes this harder to see, because the two behaviors are superimposed on the same diagram. The clarity of Fig. 3-1 is gone, even though the behavior it describes is still a large part of what the program does. But how could one use Flogo I to make the organization clearer? One option is to organize it like the Birdie program, above, but this seems like an inordinate amount of wiring for such a simple program.

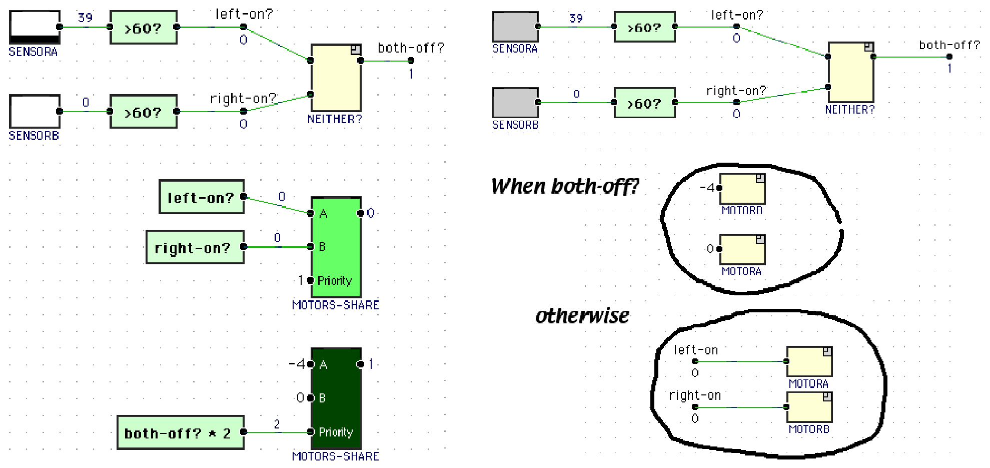


Fig. 7: (a) The Tabletop explorer program reorganized using MOTORS-SHARE output arbitration devices. The lower MOTORS-SHARE is currently active (darker), because its priority input is higher. (b) an imaginary program that seems to express the idea more directly.

To create another option, we developed a new component, called a MOTORS-SHARE. Rather than converging on a single switching point, the motor outputs of each behavior terminate at its own MOTORS-SHARE object, along with a number representing the behavior's current priority. At each moment, the MOTORS-SHARE with the highest current priority is the one that actually drives the motors.

The potential use of this device for program organization is shown in the revised Table Explorer program in Fig. 7(a). The two behaviors are now separated, as in the Birdie program, but some of the wiring complexity of the Birdie problem is avoided. On the input side, sensory information is made available via named nodes. On the output side, MOTORS-SHAREs obviate the need for wires to fan in. Still, this device is not completely successful in clarifying the intent of the program. The priority scheme, in particular, seems rather distant from what one would really like to say, which is: use *this* circuitry when both the wheels are off the table, and *that* circuitry when they aren't. Fig 7b offers an impressionistic rendering of what one would like the language to be able to do.

Expressing sequence: the count-and-beep problem

A similar weakness in the language was suggested by children's struggles with a morning warm-up challenge problem in the second workshop. It read:

> Count-and-beep challenge: Write a program to count how many times the room lights are turned off, and then beep that many times.

This puzzle stumped not only the children, but also the adults in the room, experienced programmers though they were. Everyone was able to count dimmings of the lights. But once people had written that part, a pronounced hesitancy took over. The complexity of a pulse loop with a counter was surely a big part of the problem, but I believe there was another issue as well. Once the light dimmings have been counted, that part of the program should be over. Yet the counting circuitry is still there. In this case it does no harm to have it there, but still, the solution (Fig. 8a) is counterintuitive in that it requires the connection of two circuits that are active at different times. Programmers seemed at a loss as to how to say: first use this circuit, then that one. Once again, an imaginary program can be written that seems to get at this more directly (Fig. 8b).

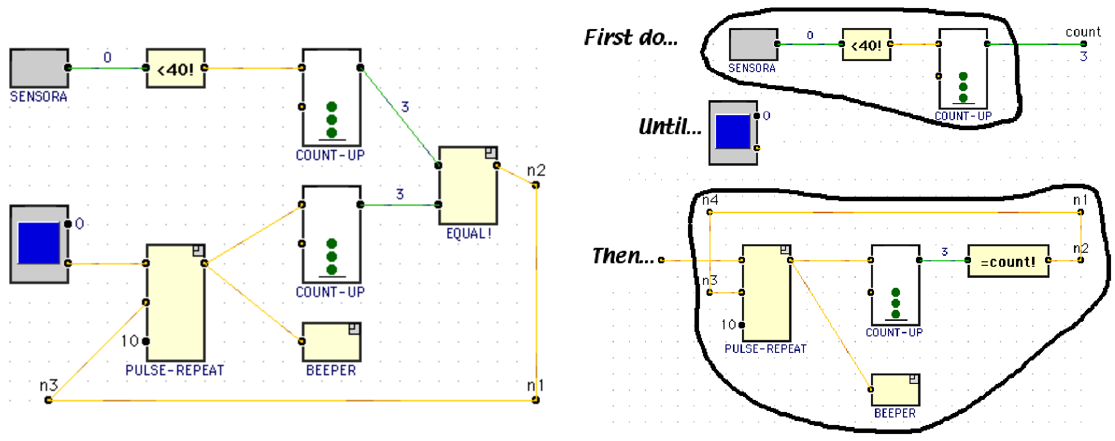


Fig. 8: Real and imaginary solutions to the count-beep problem.

How, then, can one make one part of a diagram active at some times, and another part at other times, while making the overall pattern of control clear? This has been a perennial struggle for designers of dataflow languages. LabView offers patented "structured dataflow" constructs (Kodosky et al., 1991), which work by enclosing alternative subcircuits in a box with a fixed set of inputs and outputs, with mechanisms to establish which alternative is active at any time. The drawback of this approach is that it creates hidden code. Only one subcircuit can be seen at a time, so that a printout of a diagram does not fully represent it, and on-screen reading requires mouse work to page through all such constructs.

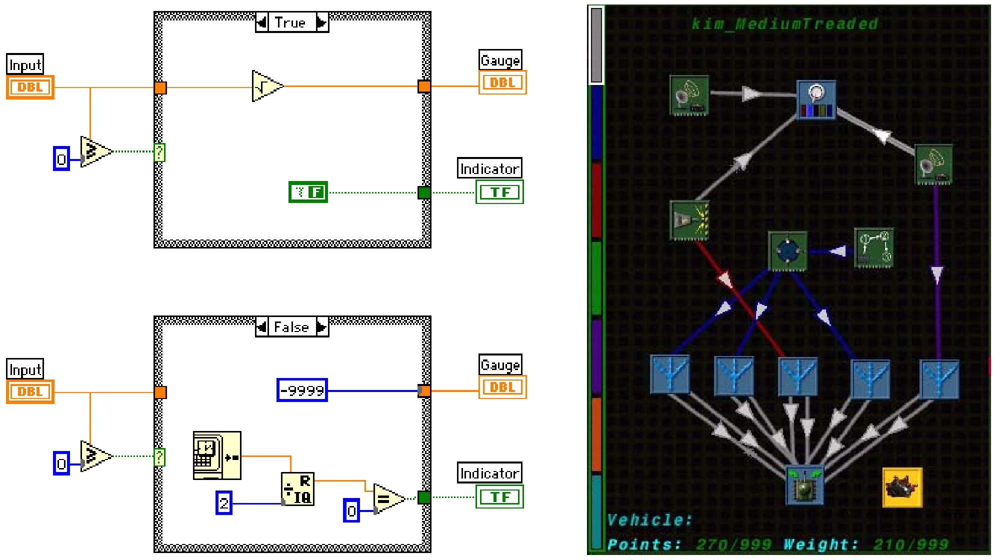


Fig. 9. Two approaches to diagram control. (a) LabView encloses alternative wirings in boxes. (b) MindRover offers colored wires, active only during specific execution modes. Note mode indicator along left edge, and mode selector device at top center of diagram.

The robot programming game MindRover (www.cognitoy.com) offers a novel approach based on the activation and deactivation of wires. A palette of colors is available to designate alternative “modes” of diagram execution. Gray wires are always active, but colored wires are active only when the diagram is in the mode of the same color. Used with care, these low-level materials can produce readable programs. However, superimposing wirings makes any individual wiring harder to pick out, and does not easily express the intent of the high-level control.

In comparison to the available visual methods, textual control structures, of the kind suggested by the imaginary programs 7b and 8b, offer some appealing advantages:

- They gather their constituent parts into a coherent compound entity. (LabView’s boxes come closest in this respect.)
- They can express top-down as well as bottom-up control. (Flogo’s motors-share and MindRover’s mode switches are bottom up devices, embedded within in the circuitry they govern).
- In some cases, at least, they describe the programmer’s intent clearly and straightforwardly.

The irony with Flogo I is that text-based procedural control structures actually exist in the language, but can only be used with text. Flogo I’s diagrams and text each contribute important capabilities for defining process. But their segregation from each other leaves the language with restricted expressivity.

Unification of the expressive powers of Flogo I’s text and diagrams thus became the driving design problem. There are three options:

1. Attempt to achieve the function and clarity of textual control constructs in a purely diagrammatic mode.

2. In addition to embedding of statements in circuits, which is straightforward, allow embedding of circuits within textual constructs, as in the Figs. 7b and 8b.
3. Make a textual equivalent of Flogo I's dataflow capabilities, and integrate these with more traditional procedural constructs.

Option #1 seemed the least attractive, being the most well trodden path. Option #2 might work, although it poses some puzzling challenges, such as how best to edit such a chimera, and how to maintain visual integrity of textual constructs across large expanses of diagram space. Option 3 began to seem potentially the cleanest—provided that the liveness of Flogo I could somehow be preserved in a textual language. This is what Flogo II sets out to do.

WHERE DID FLOGO I GET TO?

Let us briefly review the project's status at this turning point. Recall that Flogo I sought to advance robotics programming for learners by pursuing three design objectives.

1. *Open up the black box of the executing robotics program*. Flogo I does indeed make execution visible as the program runs, and allows user intervention and program editing at the same time. It is thus the first robotics language of any complexity to run fully "live." Moreover, this liveness was observed to play a central role in children's processes of learning and of program development.

2. *Support the development of larger projects and open toolsets.* Here, status is mixed. Flogo I's encapsulation model succeeded in supporting the convenient construction of a wide assortment of useful interactive components. Although encapsulation was not tested with children (this would require a longer engagement, as well as some further software development), Flogo I's ability to encapsulate by gradual steps seems likely to be helpful to learners. However, Flogo I did not immediately promote a major advance in the complexity of student projects. In a four-day workshop, children used Flogo I to build the same simple action patterns that one sees with other languages. Moreover, programs built by advanced users suggest that Flogo lacks adequate constructs for high-level control of circuitry—a barrier to the development of larger projects. In short, objective #2 seems to be held back by objective #3.

3. *Provide better means of managing the concurrent processes involved in robotics programs*. Flogo I integrates procedural text within visual dataflow structures, but not the other way around. To meet the needs of both novices and experts, a more thorough unification of the two paradigms seems necessary.

In addition, several new themes emerged from the piloting work:

- Children's engagement with a live, real-time programming environment brings *action knowledge* into the sphere of programming.
- *Tinkerability* is an important goal in language design.
- The difference between *pulse and data* is an important distinction that appears to resonate with process coordination difficulties noted by other researchers.
- The notion of a *steady frame* helps to clarify the role of Flogo I's language representations in helping users cope with the dynamics of runtime execution, and with exploratory program development.

This project has not exhausted the potential of Flogo I. Further work could have taken it much farther, and the problems identified might have been ameliorated. Both children and adults who used Flogo I have told me that they miss it and would have liked to see work on it continue. What may not have come through in the discussion so far is the character and enjoyability of Flogo I as an environment: the appeal of each visual component, the spacious feeling of the editing grid, the ease of exploring the repertoire of objects. While Flogo II preserves many of the important qualities of Flogo I, it cannot be said to supersede it entirely.

Chapter 5. The Design and Piloting of Flogo II

Flogo II is an integrated response to two design objectives:

- how can we design building blocks for process control and coordination that maximize, in combination, intelligibility, learnability, and the expressive power needed for real-time applications?
- how can the liveness of a language like Flogo I be reproduced in a textual language?

A user of Flogo II experiences its response to both questions as a coherent unity. But for the sake of an orderly exposition, I will present one answer at a time. It is instructive to begin by looking at Flogo II in a way that its users rarely it: as inert words on a page.

FLOGO II'S COMPUTATIONAL PARADIGM

A *procedural* statement describes a computational action that, once initiated, will (at least in principle) come to completion. Exploiting this property, the procedural paradigm of programming works by arranging statements in sequence. The completion of one statement leads to the initiation of the next.

A *declarative* statement describes a rule, relationship, or ongoing activity that pertains or proceeds over an indefinite span of time. Any time bounds on a declarative statement must be imposed from outside the statement itself. Declarative statements are naturally parallel (if they weren't, declarative programs would be one-liners!).

Although absolutely pure procedural or declarative languages are uncommon, a language's primary affiliation is usually clear[22]. Spreadsheets are declarative, though they allow procedural macros. C and Java, though their event handling systems are arguably declarative, are clearly procedural. So is Cricket LOGO, even though it has a declarative WHEN clause that monitors a specific condition throughout program execution. Prolog is declarative. So are dataflow diagrams.

Flogo II is a textual language in which declarative and procedural means of specifying computation are integrated within a single unified paradigm. I say "unified paradigm" because, in the literature, paradigm "integration" often refers to programming environments in which one paradigm is structurally "on top" of the other, and its constructs cannot be subsumed within constructs of the subordinate paradigm (e.g. Cox & Smedley, 2000; Erwig&Meyer, 1995; Ibrahim, 1998). Thus one does not have free access to both paradigms in building up a multi-leveled computational structure. Flogo I itself is an example of such an integration, with declarative diagrams on top and procedural text playing the subordinate role. As in Flogo I, the two paradigms often occupy distinct editing spaces as well.

In Flogo II, procedural and declarative constructs reside in the same editing space, and each kind of statement can be nested within the other kind. Indeed, one sometimes finds multiple layers of nesting within a single line of code. Such a

[22] But not always. For example, concurrent constraint programming as embodied in ToonTalk (Kahn, 1996) seems to define routines procedurally but invoke them declaratively. The procedural / declarative distinction, rather like the analog / digital distinction, has a tendency to blur as one looks more and more closely at the phenomenon.

program implicitly entails the activity of many threads of execution, whose comings and goings are coordinated by the language's control constructs: I call this *structured concurrency*.

In Flogo II, the procedural statements are called *steps* and the declarative statements are called *processes*. The use of the term "process" here is based on its natural language sense of "stuff that goes on," and not in the computer science sense of a single thread of execution. A Flogo II process may entail any number of execution threads; what makes it a process is that it continues until somebody stops it.

Here is a Flogo II version of the familiar "Table Explorer" program from chapters 3 and 4. All statements in this example are declarative, or, in Flogo II's terminology, process statements. (The arrows and labels at right are not part of the program).

```
LEFT_ON: SENSOR('A) > 60
RIGHT_ON: SENSOR('B) > 60
BOTH_OFF: not RIGHT_ON and not LEFT_ON
WHEN BOTH_OFF
    MOTORA('on,'thatway)          } A
OTHERWISE
    WHEN LEFT_ON MOTORA('on)      } B
    WHEN RIGHT_ON MOTORB('on)
```

The first three lines show how dataflow programming can be achieved textually, using variable names in place of wires. The variables LEFT_ON and RIGHT_ON are constantly updated to true or false depending on their corresponding sensor readings. The variable BOTH_OFF, in turn, is constantly updated based on their latest values. The WHEN / OTHERWISE construct is a higher-level control construct of the sort we were seeking in the last chapter. When both sensors are off the table, it makes statement A active and statements B inactive; when either or both sensors are on the table, statement A stops and statements B resume. Because this is a declarative construct, it does not perform a one-shot decision, but rather continues to switch back and forth as needed, as long as the program runs.

Integration of steps and processes

In Flogo II, step statements and process statements can be combined in a single program. However, for the combination to be meaningful, certain rules must be observed. As I mentioned earlier, a series (or "block") of step statements has a natural meaning, viz., that the statements should be executed one after another. A series of process statements is also meaningful: it described processes to be performed simultaneously. However, a mixed block containing steps and processes is not considered meaningful in Flogo II. Steps may appear only in a collection of steps, and processes only in a collection of processes. To include a step (or series of steps) among processes, or vice versa, one must wrap it in a construct that enables it to fit in. To see how this works, consider this Flogo II version of the little count-beep challenge from the previous chapter:

```
let count be 0
until sensor('B) becomes > 50
    EVERY TIME SENSOR('A) BECOMES < 25
        set count to count + 1
repeat count
    beep()
    wait 1 second
```

Unlike the previous program, this one is mostly made of steps. First we make a variable count and set it to zero. Next (until…) we count dimmings until a switch is clicked. Finally we make the beeps. Looking more closely, we see that within the until statement is a process statement: EVERY TIME, etc. (Note the Flogo II convention of presenting processes in uppercase and steps in lowercase). The EVERY TIME statement describes a rule, which can be in force for any period of time. That period of time is here established by the until construct. Thus, until converts a process into a step by putting a time bound around it.

Conversely, the EVERY TIME statement contains a step (set count to count + 1). By establishing the instant or instants at which a step should be initiated, EVERY TIME… creates a process based on that step.

These examples illustrate the basic principles by which Flogo constructs integrate steps and processes. To include a process[23] among steps, establish a way for it to terminate. These constructs all build a step from a process:

for 5 seconds …	
for 1 tenth …	
as long as light < threshold …	
until light >= threshold …	(simple converse of as long as)
until light becomes >= threshold …	(this means something a little different)

(In Flogo II the keyword "becomes" is used to signify the crossing of a threshold, much as the exclamation mark is used in Flogo I.)

To include a step among processes, specify at what point or points it should be initiated. These constructs all build a process from a step:

EVERY 6 SECONDS …	
EVERY TIME LIGHT BECOMES >= threshold …	
AS SOON AS LIGHT BECOMES >= threshold	(means the same as EVERY TIME)
REPEATEDLY …	(do step(s) over and over)
ON START …	(do step(s) each time process starts up)
ON STOP …	(do each time process stops)

In addition to these "cross-paradigm" constructs, Flogo II includes within-paradigm control constructs such as WHEN for processes, if and repeat for steps, and a do/with construct that forms a single step from the parallel execution of two or more blocks of steps.

Encapsulation

To encapsulate code in Flogo II, one currently writes a definition. Procedures are defined in a fairly conventional way, using DEFINE STEP, e.g.

```
define step nbeeps (n)
    repeat n
        beep
        wait 1 second
```

[23] All of these principles apply equally to compound statements. A block of processes is itself a process, and a block of steps is itself a step.

Functions that return results are defined in the same manner. When a defined step includes multiple simultaneous activities, returning a result also terminates those activities. For example, this little program waits, for a limited time, for the user to press sensor a or b, and makes a special motion while waiting:

```
define step wait_for_touch ()
    for 10 seconds
        REPEATEDLY
            for 5 tenths motora('on,'thisway)
for 2 tenths motora('on,'thatway)
        AS SOON AS SENSOR('A) BECOMES > 50
            return 'a
        AS SOON AS SENSOR('B) BECOMES > 50
            return 'b
    return 'timeout
```

Processes are defined similarly. Here is a simple example

```
DEFINE PROCESS TURNLEFT (SPEED)
    MOTOR1('ON,'THISWAY,'SPEED)
    MOTOR2('ON,'THATWAY,'SPEED)
```

As is always the case in blocks of process statements, the order of the MOTOR1 and MOTOR2 statements does not matter. Note that the arguments to a process call are not simply single values but continuously updated streams of values. Thus the TURNLEFT process, if invoked like this..

```
TURNLEFT( SENSORA() / 5 )
```

…will cause the robot to turn at a speed that covaries with a sensor value.

Here is a more advanced example of process definition, in which an output stream is indicated by use of the special variable OUTPUT[24]:

```
DEFINE PROCESS CUMULATIVE_MAXIMUM (SIGNAL)
    ON START set OUTPUT to SIGNAL
    OUTPUT: max(OUTPUT, SIGNAL)
```

(Incidentally, the circular dataflow in the last line causes no trouble. Each updated value of OUTPUT will be derived from its predecessor). Invoking a defined process is similar to class instantiation in object-oriented languages, in the sense that the invocation will have its own persistent set of variables and other state information.

I mentioned that writing a definition is the current way to encapsulate code. A smoother transition from working code to defined routine, as shown for Flogo I, is feasible but has not been fully implemented. Despite this, children have used definitions effectively in Flogo II.[25]

Potential expansions of the language

In the current prototype, process invocations are attached to their place of invocation in the program. For processes to become Flogo II's full answer to objects, what is needed is the ability to create and store them as data—a fairly straightforward

[24] Means for producing multiple outputs have been considered but not implemented. One idea would be for variables within the process call to be accessible by name from outside the call.
[25] Flogo II does offer one helpful shortcut in working with subroutines: the ability to edit an existing definition via an invocation, rather than going back to the definition itself. The meaning of this will become clearer in the next section.

extension of the current language. In their capacity for independent activity, processes also resemble actors (Hewitt, 1979, Lieberman, 1987).

Allowing processes to be manipulated as data objects is just one of many expansions that would be needed to complete Flogo II. To facilitate message passing between processes, I would also add event streams, analogous to Flogo I's pulse wires, as a complement to the current data streams (Flogo II's events would be more than pure pulses, carrying information with them that could be interpreted as messages). Data structures are missing as well. Graphics and interface construction tools are less developed in Flogo II than in Flogo I.

Setting aside the addition of new features and facilities, Flogo II would need, in order to progress farther, more rigorous formalization at many levels. As is appropriate in an experimental language, data types, syntax, and execution model are all defined provisionally, to the degree of precision necessary to realize and test the core innovation, which is Flogo II's unification of procedural and declarative process description.

Related Work

Partial precedent for Flogo II's paradigm unification can be found in two languages, Occam (SGS-Thompson Microelectronics, 1990) and Eden (Breitinger et al., 1997). Occam was designed to leverage the power of networks of dynamically allocated processors—the "transputer" computer architecture that was expected to flourish in the 1990's, but did not. Occam programs allow two kinds of blocks, labeled SEQ and PAR, which contain statements to be executed in sequence or in parallel, respectively. These block structures are mutually and recursively nestable, and thus unified in the same sense that Flogo II unifies procedural and declarative statements. In Occam, however, all statements are procedural. Occam's PAR blocks achieve roughly the same computational capabilities as Flogo II's process blocks, but in a more low-level form.

Like Flogo II, Occam offers a straightforward method of encapsulating parallel computation. A process is defined with a parameter list and launched through a simple call, the way one defines and calls a procedure in other languages. As in Flogo II, calling a process is a form of instantiation, and a program can launch multiple instances of the same process definition, as needed, at run time[26]. This is more flexible than the statically arranged processes one finds in visual dataflow languages such as LabView and Flogo I.

Occam processes communicate through "channels," which are somewhat unwieldy low-level entities requiring lockstep coordination between processes. Flogo II's "streams" perform much the same function, but, like Flogo I's wires, they eschew the tracking of individual values, instead simply updating the stream as data becomes available. By de-emphasizing low-level predictability, Flogo II provides a simpler and more open model of data sharing in a multiprocess program.

Overall, Occam and Flogo II generate similar functionality spaces (assuming, now, a more complete Flogo II!). Because both offer unlimited mutual subsumption of parallel and sequential processing, proficient programmers in each language would be likely to carve up many problems similarly. However, the computational ideas that the two languages share are presented by Flogo II at a higher level, in a form that is often more convenient, and likely to be more approachable by learners.

[26] A similar encapsulation model is found in the research language Eden (Breitinger et al., 1997).

FLOGO II AS A LIVE PROGRAMMING ENVIRONMENT

Flogo II's unified paradigm provides the flexible expression of control that was lacking in Flogo I. Equally important is the fact that this was achieved without sacrificing the liveness that proved to be a strength of Flogo I. This section describes the "live text" environment of Flogo II, together with some of the simple animation tools that were used by children to build games, during the first phase of Flogo II piloting. Flogo II's liveness is characterized in terms of several "abilities" of the environment: editability, controllability, visibility, browsability and growability.

Editability and controllability

Editability refers to the essential property that Flogo II programs can be edited as they run. To edit a portion of the program, one simply selects it and begins editing it in place. The part to be edited may be any syntactically coherent program segment, ranging from a single constant or token to a long series of statements. The portion being edited is temporarily enclosed in a box, to show that it is provisional. If the user has chosen to leave the edited construct running while editing it, it will run on the basis of its original text, until the editing operation is completed. When the user presses <enter> or clicks elsewhere on the screen, the text is parsed to ensure that it conforms to the syntactic requirements of the selected program segment. If it doesn't, an error message is displayed and the user must either continue editing or cancel the operation. If the new text is acceptable, it replaces the original selection and becomes part of the running program.

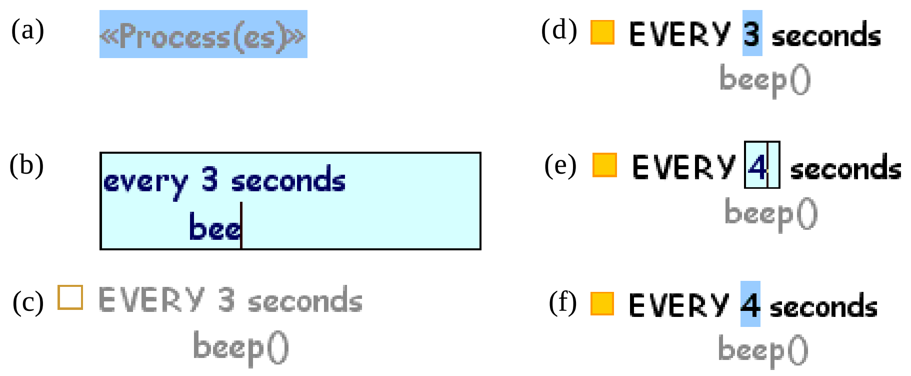
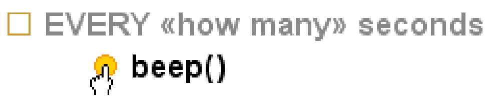


Fig. 1. Examples of Flogo II editing. (a) Click to insert placeholder statement in program; (b) edit placeholder; (c) Once editing is committed, edit boundary disappears and a statement control appears; (d) statement has been turned on and a portion has been selected for further editing; (e) during editing, statement continues to function in original form (in this case, beeping every 3 seconds); (f) new code takes effect once committed.

New statements or other syntactic units are easily inserted as well. Chunks of code can be moved around in drag-and-drop fashion. In each editing operation, the editor will accept only well-formed instances of the required syntactic unit. Where a syntactic unit has not been filled in the, system provides a "dummy" construct that displays an appropriate prompt (e.g. <<time unit>>). In some contexts, one can select from a menu of allowable options (expansion of this recently added feature would probably be helpful to beginners).

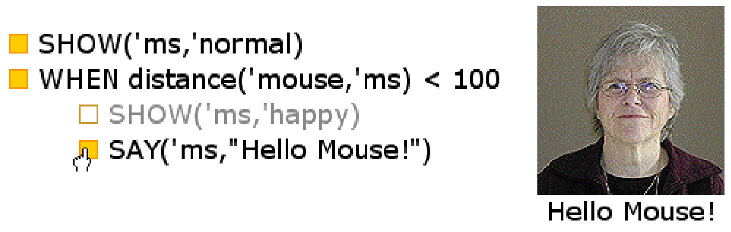
DiSessa (1997b) offered the term "pokability" to describe the ability to run any program fragment selectively, as one is reading or writing a program. Pokability of various flavors can now be found in many educational programming environments (e.g. Boxer, Squeak / eToys, Visual AgenTalk, Alice), as well as some professional ones (many LISP environments, for example). Flogo II programs are pokable, but also include additional kinds of controls enabling the programmer to determine which parts of the program should be “live” at any time.



```
☐ EVERY «how many» seconds
      beep()
```

Fig. 2: a procedural statement can be “poked,” despite the fact that the enclosing construct is incomplete.

Procedural statements in Flogo II appear with a circular control next to them; clicking the control executes the statement immediately (Fig. 2). (Shift-clicking executes the statement followed by all remaining statements in the same block). Once accepted by the editor, process statements appear with a square control, allowing the process statement to be turned on or off. These controls have two different behaviors. For top-level statements, which are free to be on or not, the controls function like switches: clicking once turns the statement on, clicking again turns it off. Process statements that are nested within other constructs, on the other hand, have a proper state that is dictated by the state of the enclosing construct. For these statements, the control box allows the user to make a temporary exception—turning on a statement that should be off, or vice versa—but only for as long as the mouse is held down. In Fig. 3 this feature is used to test the SAY command while the condition that normally turns it on is not met.



```
■ SHOW('ms,'normal)
■ WHEN distance('mouse,'ms) < 100
    ☐ SHOW('ms,'happy)
    ■ SAY('ms,"Hello Mouse!")
```

Fig. 3: Temporarily activating a process statement by pressing its control square.

Visibility

Flogo II illustrates a number of ways that program state can be reflected in the body of the program. The guiding principle is to associate each aspect of program state as directly as possible with the unit of code with which it is associated. If one follows this principle far enough, a qualitative change in the nature of a textual program is achieved. Rather than a set of instructions to be read and acted on by some other entity (a computer), the program feels and acts more like a self-contained device—a machine made of text.

The following elements of program state can be rendered live in the Flogo II environment:

1. Execution status. Process statements are dimmed when inactive, but shown in high contrast when running. The execution control next to each process statement also lights up while it runs. Step statements and their controls also light up while they are executed. Highlighting of very fast steps is prolonged for a moment to ensure visibility.
2. Variable values. Two kinds of statements establish variables in Flogo II: variable declaration with colon syntax, and the less-used let statement. Each displays the variable’s current value directly within the statement.

3. Expression values. The current value of any expression can be displayed beneath it in a thermometer display. (In the context of a step or an inactive process, “current value” means the value returned when the expression was last evaluated). In Flogo II terminology, a “thermometer” is any long, thin, horizontal information display that appears beneath a program clause. The actual display may be a varying-length bar of the kind usually called a “thermometer,” or some other graphic rendition, or small text extended by a simple bar to show what clause it applies to. What thermometers have in common is their shape, which minimizes disruption of textual structure, and a scalable format allowing them to line up precisely with the text above them. Thermometers for nested expressions also stack up readably.
4. Control construct state. Control construct progress can also be monitored using thermometers. In time-based constructs such as for 6 tenths, or every 5 seconds, a thermometer shows the passing of the specified time period (Fig. 4). The repeat statement shows its progress by filling in a series of dots, one per iteration (for large numbers of iterations, the representation switches to a continuous line). This can be an important visual cue when loops iterate very quickly, because rapidly repeated execution is not discriminable in the highlighting of steps.. When and while constructs do not require custom thermometers: if desired one can simply monitor the condition as a normal expression. A custom thermometer for BECOMES clauses, showing the test value varying in relation to the threshold value, has been designed but not implemented.

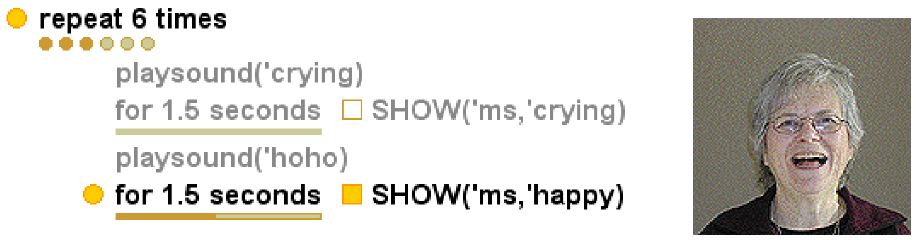
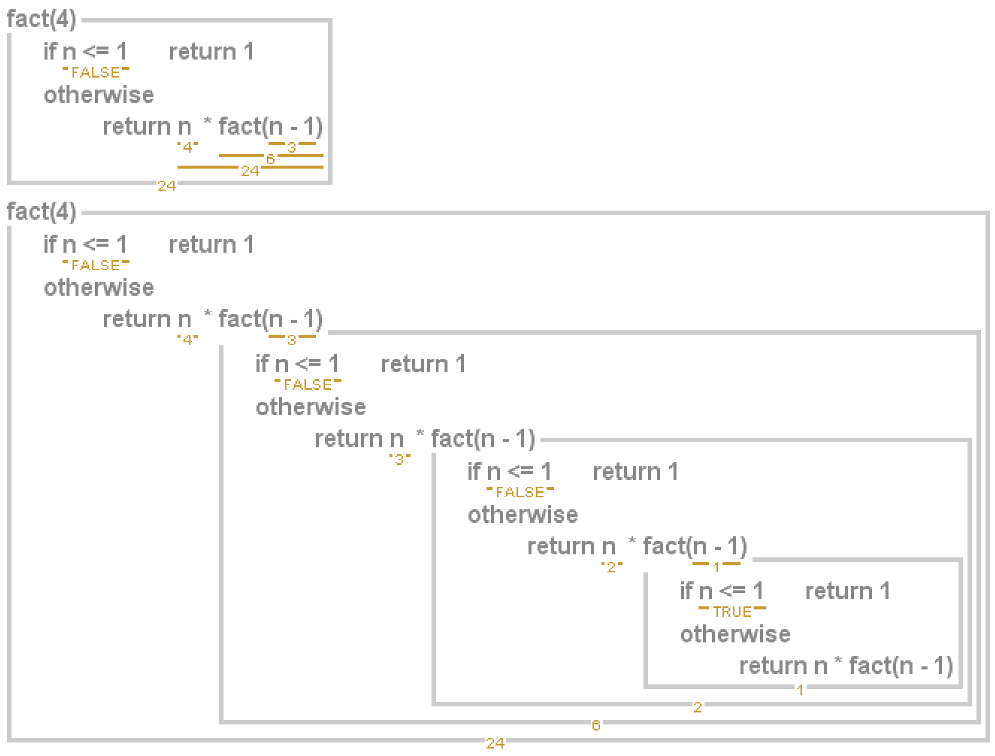

```
repeat 6 times
        playsound('crying)
        for 1.5 seconds  SHOW('ms,'crying)
        playsound('hoho)
        for 1.5 seconds  SHOW('ms,'happy)
```

Fig. 4: Control construct monitoring. This repeat statement is in its third iteration. The lower for statement is halfway through its 1.5-second interval.

Browsability and Growability

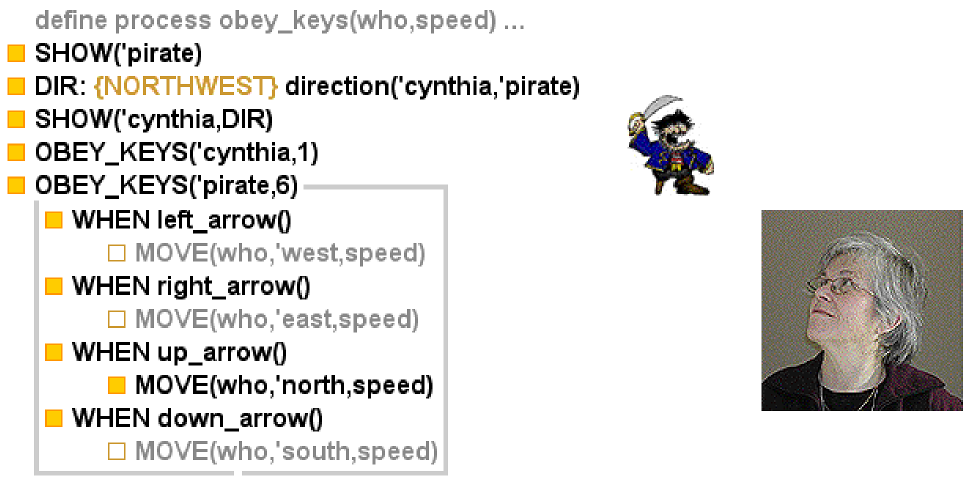
A live text program has a lot to show. To help the programmer find and focus on information of interest, the program must be browsable. A simple browsability feature in Flogo II is the control of thermometer displays. Thermometers can be revealed or concealed with a single click on the corresponding program clause. The other is the ability to open or close subroutine invocations (whether of the step or process variety) at their point of invocation in the code. Because Flogo II subroutine calls are distinct, allocated instances (and not, for example, passing phenomena on a stack), they can be examined not only during their execution but before or after. This can be useful with steps; for example, Flogo II’s subroutine browsing facility yields an unusually clear trace of the activity of a recursive function (Fig. 5).

Subroutines are a good way to keep programs readable, but the usual tradeoff, when reading code, is the extra effort it takes to look up a subroutine definition. In Flogo II the invocation is available right where it happens. Not only that, the subroutine code is live, and one can watch it run just like any other part of the program (Fig. 6). Moreover, the invocation is editable: any change in the invocation is immediately propagated back to the definition, and to all other invocations in the program.



```
fact(4)
    if n <= 1      return 1
      FALSE
    otherwise
        return n  * fact(n - 1)
                4           3
                        6
                   24
             24
fact(4)
    if n <= 1      return 1
      FALSE
    otherwise
        return n  * fact(n - 1)
                4           3
            if n <= 1      return 1
              FALSE
            otherwise
                return n  * fact(n - 1)
                        3
                    if n <= 1      return 1
                      FALSE
                    otherwise
                        return n  * fact(n - 1)
                                2           1
                            if n <= 1      return 1
                              TRUE
                            otherwise
                                return n * fact(n - 1)
                                                  1
                                          2
                                6
                    24
```

Fig. 5: A recursive function call opened up one level deep (above), and to its full depth (below).



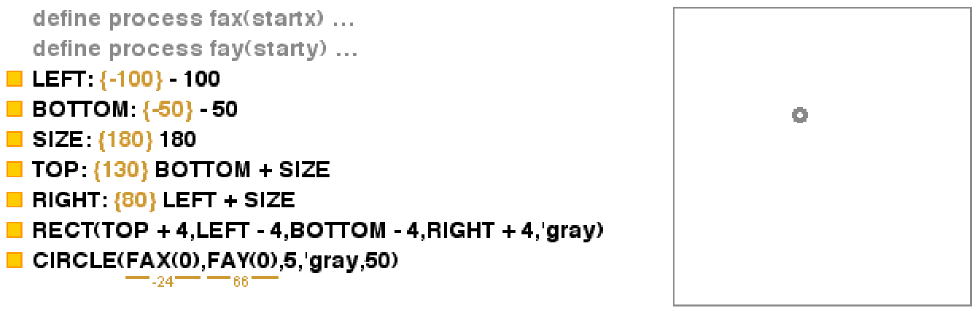
```
  define process obey_keys(who,speed) ...
■ SHOW('pirate)
■ DIR: {NORTHWEST} direction('cynthia,'pirate)
■ SHOW('cynthia,DIR)
■ OBEY_KEYS('cynthia,1)
■ OBEY_KEYS('pirate,6)
    ■ WHEN left_arrow()
          □ MOVE(who,'west,speed)
    ■ WHEN right_arrow()
          □ MOVE(who,'east,speed)
    ■ WHEN up_arrow()
          ■ MOVE(who,'north,speed)
    ■ WHEN down_arrow()
          □ MOVE(who,'south,speed)
```

Fig. 6: This program has two separate instances of the defined process obey_keys. (The definition, at top, is collapsed). The second instance, controlling the character 'pirate, is shown open. As the up arrow is pressed, the pirate moves up quickly while the 'cynthia character moves up slowly. The line SHOW('cynthia,DIR) has the effect of making Cynthia always look toward the pirate. This works because the image repertoire includes eight face images, each named for a compass direction. The value of DIR can thus be used to specify the appearance of the character.

The reader may wonder whether Flogo's system of allocated instances is inherently inefficient. Certainly it takes more memory than the standard practice of reusing stack space—a trade-off that is not that costly in age of abundant RAM. However, from the point of view of time-efficiency, Flogo II does as well as, if not better than, a stack-based language. The key is *growability*, that is, the fact that a Flogo II program's memory footprint never shrinks, but grows as needed to accommodate computation. It is important to clarify that Flogo II allocates a distinct instance not for every *time* that a subroutine is invoked, but rather for every *place* it is invoked. Should the program ever re-execute that piece of code, the previously allocated instance will be available for re-use. For example, if the recursive factorial program were embedded in a loop, each iteration would re-use the subroutine frames left over from the previous time around. Any time the argument exceeded the previous maximum, the chain of instances would be extended to accommodate it. User browsing can also trigger Instantiation, since the user is allowed to inspect any subroutine call even before it has executed. Creating an instance does take some time (on the scale of micro- to milliseconds), but it is a comparatively rare event.[27]

Sample project: BoxTower

To illustrate some of the power of Flogo II's dual ontology of steps and processes, and some of the possibilities of process encapsulation, a more elaborate programming project is briefly presented here. The "boxtower" project is not a complete computer game but provides an interesting and colorful dynamic environment on which a number of interesting games might be built. The program is far more advanced than anything written by the pilot group. Moreover, it uses capabilities (declarative shape graphics) added to the language after piloting ended. The example may be of help to the reader in gaining a qualitative sense of how processes work, and what they can express. In particular, having seen how Flogo II handles a familiar recursive step-based algorithm (factorial), it is instructive to ask: what would a recursive process look like, and what might it accomplish?



```
  define process fax(startx) ...
  define process fay(starty) ...
■ LEFT: {-100} - 100
■ BOTTOM: {-50} - 50
■ SIZE: {180} 180
■ TOP: {130} BOTTOM + SIZE
■ RIGHT: {80} LEFT + SIZE
■ RECT(TOP + 4,LEFT - 4,BOTTOM - 4,RIGHT + 4,'gray)
■ CIRCLE(FAX(0),FAY(0),5,'gray,50)
            —-24——  —66——
```

Fig. 7: BOXTOWER project beginnings: a "doughnut" driven by arrow keys.

We begin with a small program in which a small gray doughnut shape can be moved around within a bounded square area, under the control of the arrow keys (Fig. 7). Arrow key control is expressed differently here than in previous examples. Two processes, named "fax" (follow arrows X) and "fay" (follow arrows Y), each monitor a pair of arrow keys and output a stream of numbers based on arrow key activity. The code for "fax" is shown below. Note that this approach, unlike the OBEY_KEYS process illustrated earlier, permits diagonal movement, because X and Y coordinates are maintained separately.

[27] A full-featured descendant of Flogo II should probably allow advanced users to specify stack-based computation in some cases, e.g. deep recursive calls that are infrequently performed. But it may be appropriate for stack-based subroutine invocation to be a considered a somewhat specialized efficiency measure, as tail-recursion is now.

```
DEFINE PROCESS FAX(startx)
    OUTPUT: on start startx
    WHEN left_arrow()
        EVERY 1 hundredth
            set OUTPUT to OUTPUT - 2
            if OUTPUT < LEFT set OUTPUT to RIGHT  ; wrap-around motion
    WHEN right_arrow()
        EVERY 1 hundredth
            set OUTPUT to OUTPUT + 2
            if OUTPUT > RIGHT set OUTPUT to LEFT  ; wrap-around motion
```

Having established movement within a game frame, we can now add an interesting responsive display, generated by this recursive process:

```
DEFINE PROCESS BOXTOWER(BOTTOM,LEFT,SIZE)
    TOP: BOTTOM + SIZE
    RIGHT: LEFT + SIZE
    FILL: on resume choose('red,'blue,'green,'yellow) ; random color selection
    RECT(TOP,LEFT,BOTTOM,RIGHT,'gray,PCT,FILL) ; as PCT increases, color fills
in
    WHEN CY > TOP and SIZE > 8
        MIDDLE: (LEFT + RIGHT) div 2
        WHEN CX < MIDDLE
            BOXTOWER(TOP,LEFT,MIDDLE - LEFT)
        OTHERWISE
            BOXTOWER(TOP,MIDDLE,RIGHT - MIDDLE)
    START_TIME: on resume tenths_now()  ⎫
    AGE: tenths_now() - START_TIME       ⎬ timing code for fill-in effect
    PCT: min(100,AGE % (SIZE div 7))     ⎭
```

This process definition has three sections. The first draws a square of size and position specified by the process arguments. Each time the process resumes, the color of the square is randomly reset. The middle section will make a recursive call if the user-controlled doughnut shape is positioned somewhere above the drawn square. In that case (and if boxes have not yet become too small), a half-size box will be drawn above either the left or the right half of the current box, depending on the position of the doughnut. The recursive calls will themselves make recursive calls, so the effect will be a tower of ever-small boxes, rising to the level of the doughnut. As the user moves the doughnut about, the tower of boxes follows along, always finding a way to rise up to the doughnut's position.

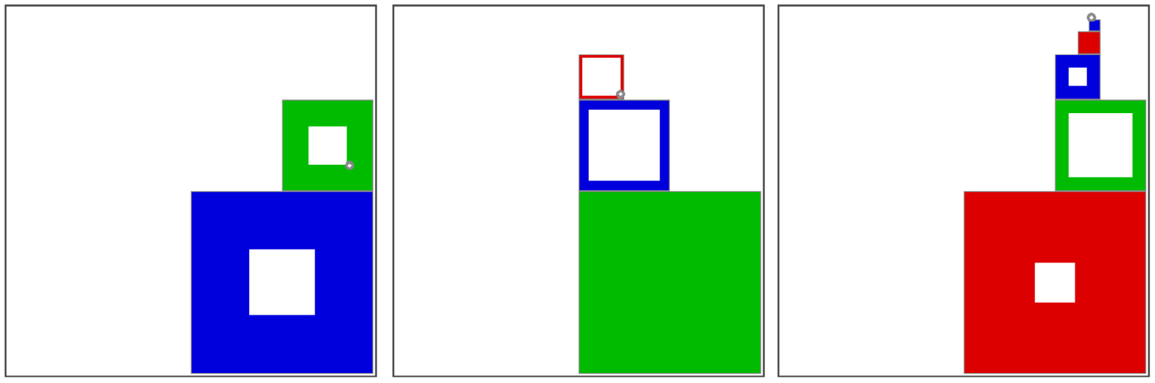
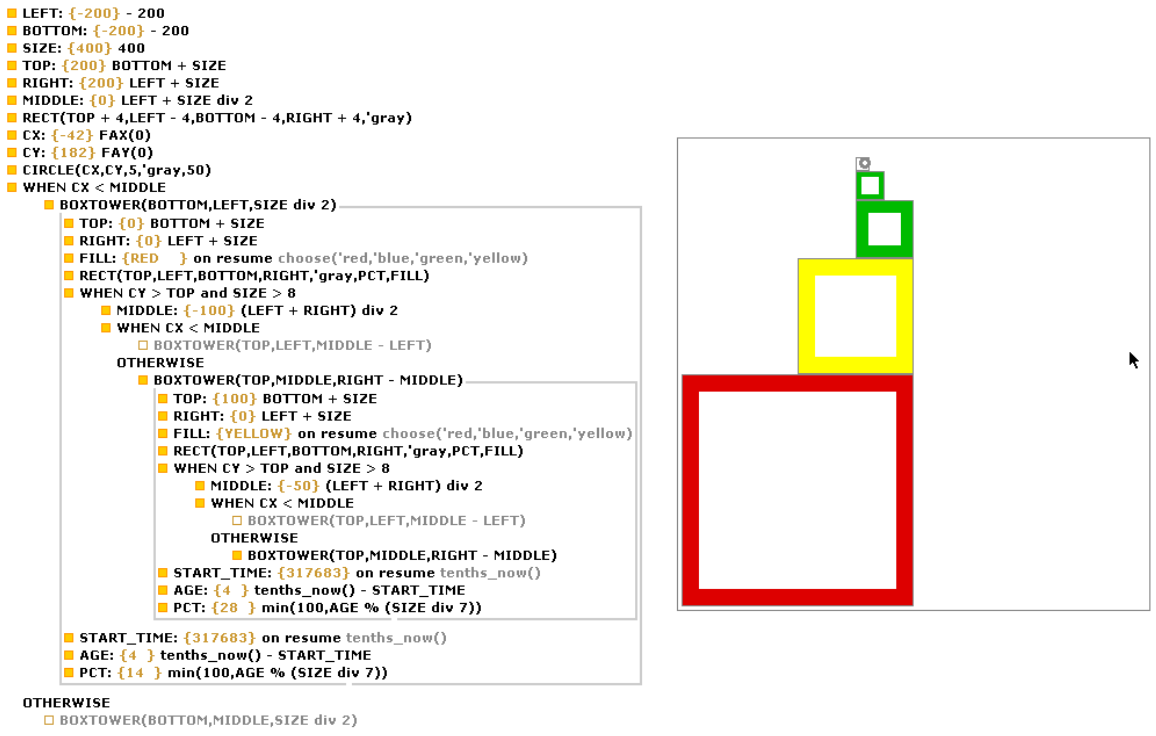


Fig. 8. Three snapshots of BOXTOWER interaction.

The third part of the BOXTOWER subroutine adds additional dynamism to the display. Each time a box reappears, it begins by being empty of color. As time passes, the variable PCT increases, and the box gradually fills in. PCT is computed in a way that makes smaller boxes fill in faster. (The % operator is like division, but returns a whole-numbered percentage. For example, 4 % 8 returns 50.) The filling in of a box might be made into a significant event in a game. Penalties or rewards depending on box color would provide incentives for the player to stay over some boxes to give them time to fill in, while moving off others before they can fill in.



```
■ LEFT: {-200} - 200
■ BOTTOM: {-200} - 200
■ SIZE: {400} 400
■ TOP: {200} BOTTOM + SIZE
■ RIGHT: {200} LEFT + SIZE
■ MIDDLE: {0} LEFT + SIZE div 2
■ RECT(TOP + 4,LEFT - 4,BOTTOM - 4,RIGHT + 4,'gray)
■ CX: {-42} FAX(0)
■ CY: {182} FAY(0)
■ CIRCLE(CX,CY,5,'gray,50)
■ WHEN CX < MIDDLE
    ■ BOXTOWER(BOTTOM,LEFT,SIZE div 2)
      ■ TOP: {0} BOTTOM + SIZE
      ■ RIGHT: {0} LEFT + SIZE
      ■ FILL: {RED    } on resume choose('red,'blue,'green,'yellow)
      ■ RECT(TOP,LEFT,BOTTOM,RIGHT,'gray,PCT,FILL)
      ■ WHEN CY > TOP and SIZE > 8
          ■ MIDDLE: {-100} (LEFT + RIGHT) div 2
          ■ WHEN CX < MIDDLE
              □ BOXTOWER(TOP,LEFT,MIDDLE - LEFT)
            OTHERWISE
              ■ BOXTOWER(TOP,MIDDLE,RIGHT - MIDDLE)
                ■ TOP: {100} BOTTOM + SIZE
                ■ RIGHT: {0} LEFT + SIZE
                ■ FILL: {YELLOW} on resume choose('red,'blue,'green,'yellow)
                ■ RECT(TOP,LEFT,BOTTOM,RIGHT,'gray,PCT,FILL)
                ■ WHEN CY > TOP and SIZE > 8
                    ■ MIDDLE: {-50} (LEFT + RIGHT) div 2
                    ■ WHEN CX < MIDDLE
                        □ BOXTOWER(TOP,LEFT,MIDDLE - LEFT)
                      OTHERWISE
                        ■ BOXTOWER(TOP,MIDDLE,RIGHT - MIDDLE)
                ■ START_TIME: {317683} on resume tenths_now()
                ■ AGE: {4  } tenths_now() - START_TIME
                ■ PCT: {28  } min(100,AGE % (SIZE div 7))
      ■ START_TIME: {317683} on resume tenths_now()
      ■ AGE: {4  } tenths_now() - START_TIME
      ■ PCT: {14  } min(100,AGE % (SIZE div 7))
  OTHERWISE
    □ BOXTOWER(BOTTOM,MIDDLE,SIZE div 2)
```

Fig. 9. Snapshot of live program execution of BOXTOWER program. Note live nested display of recursive calls.

As usual, the Flogo II programming environment provides live access to the running program. Recursive process calls can be opened up, monitored, and intervened in. For example, inactive BOXTOWER calls can be poked with the mouse, temporarily turning on an additional square. Opening up such nested invocations becomes visually unwieldy past about three levels deep, but one can imagine improvements to subroutine browsing (optional visual stacking of nested calls, for example) that could alleviate this.

Liveness and paradigm

In Chapter 4 we noted that conventional procedural languages make dubious candidates for a live programming environment. Yet half of Flogo II is procedural: how live, then, can Flogo II be? Procedural code in Flogo II is more live than in other environments, because of two factors: the visual integration of program state with program text, and Flogo II's growable memory system. During program execution, locations in a traditional procedural stack are constantly re-used for different purposes. Seen from outside, their contents are a chaotic jumble. Flogo II's growable memory architecture assigns a single meaning to each memory location. Every variable (global or local), every expression result has its own memory location. This creates a steady frame for program execution, and, consequently, a meaningful dynamic display.

And yet, procedural code remains less live than declarative code. Much procedural code spends most of its time in an inert state, waiting to be executed. When it does run, its execution flashes by as part of a thread of execution that may jump around in the program. Flogo II's steady trace of procedural code is helpful for sorting out what the procedural code is doing or has done. But when procedural code can be re-expressed in declarative terms, a simpler real-time display often results. As we will see, the visual reward of converting steps to process is an important characteristic of Flogo II as a learning environment.

Chapter 6: Flogo II Piloting and the Step/Process Distinction

Flogo II's two major distinguishing features are (i) its commitment to a duality between step- and process-based descriptions of computation; and (ii) its live interface. This chapter explores some consequences of the step / process duality. The following chapter considers the educational implications of live programming. To begin with, however, I present a broad overview of how Flogo II was tested with children. Some broad-brush impressions and judgments are included here. This will set the context for the subsequent detailed discussion of the major educational themes of this dissertation.

PILOTING OF FLOGO II

Flogo II was piloted with children in three phases[28]:

1. Afterschool Sessions
   (a)A series of five three-hour afterschool sessions, attended by 12 fifth and sixth graders. Activities were focused on the construction of games and other interactive construction, using Flogo II's sprite animation capabilities.

   (b) Contemporaneously with (a), and similarly focused on game-making, a series of seven two-hour afterschool sessions, attended by five boys in grades 7-8.
2. A week-long workshop at MIT lasting five days, with about four hours per day devoted to robotics programming activities. The workshop was attended by nine children aged 10-15, seven of whom had taken part in afternoon sessions.
3. Follow-up programming sessions at MIT in which participants alternated between research interviews with me, individually or in pairs, and "lab time" supervised by an undergraduate assistant. Six children came at least once. I gradually eliminated the "lab time" and narrowed the focus to three boys, two of whom came for five sessions, and one for nine. All had previous Flogo II experience, having attended either the afterschool sessions or the weeklong workshop, or both.

Subjects

In all, 19 different children, aged 10-15, worked with Flogo II for an average of about 20 hours each. All participants were confident computer users, experienced with word processing and media tools, and having some background in Microworlds Logo. Three had in fact participated in Flogo I workshops two years earlier. None, however, would be considered proficient in programming (by age-appropriate standards) at the outset of these trials.

Several limitations of this participant group should be acknowledged. First, the group was, at each stage, self-selected: We advertised the opportunity to participate, and

[28] Cynthia Solomon collaborated with me in organizing and running the Phase 1 afterschool sessions and Phase 2 workshop. Katie Epmeier collaborated in Phase 1 teaching. Through all three phases, undergraduate assistants Warren Chia, Sandra Galdamez, and Robert Kwok handled videotaping and robot construction, and provided general support.

interested children (or families) applied. Second, and perhaps because of this, boys outnumbered girls 4-2 in the interview group, and 14-5 overall. Racially, 5 out of 6 in the interview group and 17 out of 19 overall were Caucasian. Though not necessarily from wealthy families themselves, all participants were students at a wealthy and prestigious private school in the Boston area. Moreover, many participants were characterized by their teacher as “bright” in comparison to their classmates.

Since Flogo is ultimately intended for a much more diverse audience, diversity of the testing pool is clearly high on the agenda for future work. However, the current research question is not about broad trends in learning or performance, but about important landmarks in an emerging intellectual domain. Thus it is reasonable to expect that the observations reported here would not be invalidated, but rather extended, or at most reframed, by work with a wider variety of children, as well as in different contexts.

Data Collection

Video data was collected (by undergraduate assistants) during all three phases of piloting. In the first two phases, with many children working at the same time, a single camera (occasionally two) yields only a sample of the activity. Video segments include such things as whole group discussions, children’s demonstrations and explanations of their work, and selected episodes of children programming either on their own or with the help of an adult. For Phase 3, the video record is fairly complete.

In working with children, my role varied somewhat along the teacher/interviewer axis. In the first two phases I tended to contribute more teaching and advice, and in the third phase less. But in all cases, I attempted to elicit children’s own knowledge and problem-solving capabilities, through questions, encouragement, and careful modulation of my assistance. Ideas and advice were contributed in small increments, to see what use the student might make of each new piece. Where I could see the student confronting a multifaceted problem, I would sometimes offer a solution to one part of the problem, in order to clear away complications and sharpen the student’s engagement with what I judged to be the more interesting part, from the point of view of this study. For example, when students floundered with syntax, I would wait just long enough to get a sense of the syntactic difficulty, and then help them come quickly to a legal formulation of their intended computational actions—allowing them to work out for themselves whether these actions achieved the effect they desired.

The video records are supplemented by saved student files (all phases) and screen snapshots (Phase 3), and, for Phase 3, by notes taken during each session and fleshed out immediately afterward, averaging two pages per session, describing the overall flow of activity and highlighting points of interest.

These records provide a compendium of student work and activity within which certain strands of thinking, learning, and media use can be traced. Discussions in sections to follow will draw on this compendium.

Overview of Piloting Activities

To help set the stage for later in-depth discussion of issues and themes in the piloting data, this section provides a brief overview of the three phases: what children did, some broad impressions of what was learned, and how the software evolved. It should be noted that in the foregoing sections Flogo II has been presented in the form that it reached toward the end of the study. At the beginning of piloting the language had only a subset of the current features, and a large superset of the current bugs.

Phase 1: Afterschool sessions

One of the discoveries of Phase 1 was the strong potential of Flogo II as a game programming environment for children. Flogo I had been used exclusively for robotics, and Flogo II had been designed and developed with robotics and related behavior-construction applications foremost in mind (and with some sense of an eventual impact on programming-in-general). I included a few graphic effects, e.g. the ability to program a stationary face to look around, for the sake of a few warm-up activities prior to getting into robotics. Initially out of necessity (the robotics component was behind schedule), but then out of a sense of emerging opportunity, these onscreen projects grew to encompass all of Phase I. As game construction possibilities emerged, and often in response to children's requests, I gradually added more game-related features to the language:

- sprite movement
- sound effects
- a structured priority system for resolving contradictory animation commands (discussed below)
- Step-based alternatives to process-based animation commands (discussed below)
- random functions
- a "nearest sprite" primitive
- a "warp" command for moving sprites instantly to a new location
- a coordinate system, with xcor, ycor, setx and sety primitives
- an option for sprites to "wrap around" when reaching an edge of the play area
- arrow key input primitives

As these capabilities came online, children, working in twos or threes, used them first to make game fragments, and later (for some children) more complete games. By "game fragment" I mean some sort of interesting animated effect or response. For example, more than one group used the repertoire of images of their teacher to program the teacher to make a series of silly faces and comments, or to react amusingly to the approach of a monkey or a banana. Another child programmed a sort of dance scene, in which four sprites began at the corners of the screen and then gracefully converged in the center. Simple following behaviors, for example one sprite following another sprite which follows the mouse, were also common. What helped these sorts of effects crystallize into more game-like constructions was the notion of a score variable. Once introduced to a few groups, this idea spread quickly. A few lines of code around a score variable can easily become the nucleus of a game. For example:

```
SCORE: {10}
Set SCORE to 0 ; used manually when restarting game
EVERY TIME distance('hero,'target) becomes < 25
    Playsound('applause)
    Set score to score + 3
```

Adapting this idea, some groups also added a "lives" variable to their games; in these games, play would end when the player ran out of lives.

Given that the available game primitives related mostly to movable sprites and their spatial relationships, it is not surprising that most games were about chasing or being chased. Nonetheless there was some interesting variety. For example:

- a game in which the player must repeatedly guide a lemon to a picture of the teacher, while avoiding two other pursuing sprites. The target picture warps to a new random position on the screen every five seconds. (Figure X).
- a two-player "takeaway" game in which a pirate (controlled by arrow keys) and a monkey (controlled via the mouse) vie for possession of a bunch of bananas.
- A large number of stationary objects are scattered about the screen. Every couple seconds, a randomly selected one begins to move. The player earns points by noticing which one is moving, and clicking it before it stops.
- A game in which one tries to maneuver a character around a central object while being pursued by a swarm of monkeys. Points are awarded for each complete circumnavigation. (Considerable work in this project went into keeping the monkeys from clustering too tightly).

Another point in this creative space was demonstrated in a program written by undergraduate assistants Warren Chia and Robert Kwok. It is a slot machine game in which the player sets three sprites to cycling through a set of appearances. If two sprites end up looking the same, the game's host character responds encouragingly. If all three are the same, the player wins.

The relative ease of adapting a language designed for robotics into a game construction language helped to highlight the commonality of the programming challenges in the two domains, and to clarify that the essential category is not robotics programming so much as real-time programming.

Mitigating Flogo II's success in Phase 1 was the roughness of this early prototype. Bugs, absent features, and unhelpful feedback, frequently held students back. Not all students were equally willing or able to persist in the face of these obstacles, without substantial expert support. The language's creator, being the only expert in the room, was thus kept quite busy. A common theme in many student difficulties was Flogo II's formal distinction between steps and processes. Students were often unclear on what control construct would meet their needs, and what kinds of statements were allowed in different contexts. Flogo II provided inadequate support for experimenting in this realm. Its handling of misplaced steps and processes was not very clear. The editor would allow steps to placed in a process context, and vice versa. Misplaced steps would simply never execute, and misplaced processes would, confusingly, simply stay on. Meanwhile, no menu support was available to help children find and experiment with the set of allowable constructs in each context. (We handed out a paper-based language summary, but paper documentation is a very abstract resource, especially at this age). Given the excellent use that children made of menu facilities in Flogo I, this is a clearly a feature that should be added. Learning and design challenges of the step / process distinction are discussed further in the next chapter.

Phase I also revealed a difference between the younger (age 10-12) and older (age 13-15) groups of children. Broadly speaking, children in the younger group approached programming more naïvely. When confronted with the challenge of producing some behavior, their available options were either (i) to apply a familiar programming pattern, with some small adaptation, (for example, to maintain a "lives" variable by adapting code for a "score" variable) or (ii) to ask an adult for the "right way" to express it. An important third option, (iii) experimenting with available primitives to see what develops (i.e. tinkering) was not fully available, due to the incompleteness of the prototype. The older group, meanwhile, added a fourth approach: (iv) strategically combining known techniques in order to engineer the desired behavior[29]. This problem-solving orientation and ability put the older group in a much better position to achieve interesting new effects with the language, and gave them a different kind of ownership of their work. As a result, our sense was that the afterschool sessions for the older kids were, in general, more successful than those for the younger children, and that the prototype, in its current form, is a better fit for the older group. There remain, however, ample grounds for optimism that Flogo II programming could be brought solidly within reach of younger children through further development of the environment, and especially of features, such as menu-based editing, that support a tinkering approach.

Phase 2: Robotics workshop

Phase 2 brought nine children aged 10-15—seven afterschool participants and two new recruits—to MIT for five half-days of robotics programming. To help keep the focus on programming, we built robotic vehicles (from LEGO bricks) in advance, and provided them to participants. In place of the cricket controllers used with Flogo I, we now used the Media Lab's Tower technology (Mikhak, Lyon, & Gorton, 2002), which allows easier communications with the desktop via a direct cable link, faster data updates, and a larger number of sensors and motors.

Following the model of the second Flogo I workshop two years earlier, I offered the children a series of robotics programming challenges, initially very simple, intended to introduce techniques and build skills that children could then apply to more imaginative projects. As it turned out, however, children were highly motivated simply to work on the robotics challenges. In a whole-group discussion I floated the idea of working toward some kind of art exhibit, or a set of theme-based constructions, but these attempts to suggest a humanistic elaboration of pure robotics were rebuffed: the children simply wanted to get their robots to behave effectively. Moreover, they liked working from challenges rather than devising behaviors of their own. This motivation remained strong all week, and the children, girls as well as boys, not uncommonly turned down invitations to take a break and see a demonstration of Media Lab technology, or play outside, in favor of continuing work on their current robotics challenge.

In response to participant demand, I wrote more challenges on the wall each day, so the final set was quite large. Each challenge was actually a small series of subchallenges, designed to build up an interesting behavior by steps (Fig. 7). Working mostly in pairs, participants would select a challenge and work on it until they were ready to move on. Only the first two challenges were required of everyone. After that, participants were free to select the ones that interested them. However, we worked to recruit at least one group for each challenge, with the stated goal of building up expertise of the group as a whole. At two or three points during the day, when groups were ready, we would stop work for a demo session, in which teams would demonstrate and explain their working solutions to the group. There were usually three or four adults in the room, so adult support was considerably more available than in the afterschool sessions.

[29] For an example, see the head-shaking program in Chapter 7

A. Program your robot to…

- move in a square
- move in a circle
- move in a bigger circle
- move in a spiral
- spell “MIT”

B. Program your robot so that…

- a switch makes it stop or start
- another switch controls forward/backward
- touching a light sensor on the left side makes it veer to the left
- touching a light sensor on the right side makes it veer right
- one switch opens and closes the grabber
- finally: Using the sensors, control your robot manually and make it go to a ball, pick it up, and bring it back.

C. Program your robot to…

- drive to the edge of the table and stop before falling off.
- drive along the edge of the table
- drive around the table without falling off.

D. Make your robot…

- turn until it detects something near it (i.e., within about 8”), using a distance sensor.
- keep turning toward the near thing (e.g., your hand) as the thing moves.
- follow your hand all around the table

E. Program your robot to…

- follow a black line
- search for a line and then follow it
- When there is a break in the line, get your robot to search for where the line continues.

F. Program your robot to…

- scan for the brightest light in any direction
- find the brightest light and turn toward it

G. Program your robot to…

- avoid crashing through walls while moving
- navigate through a maze

H. Program your robot to…

- push a ball forward and not lose it
- Steer a ball left (or right) and not lose it.

I. Program your robot to…

- turn until it finds a graspable object
- approach the object and grab it.

Fig. 7: Phase 2 robotics challenges. Almost all were achieved by at least one of the 4-5 groups.

The robotics challenges progressed through four levels:

1. *Fixed behaviors*, such as drawing a shape or spelling a word. These involved no use of sensors, but nonetheless posed some interesting algorithmic challenges. For example, drawing a spiral requires iteration with a constantly increasing line length (or decreasing turn angle, but no-one did it that way). A program to spell a word is most helpfully organized into separate procedures for each letter.
2. *User-controlled behaviors*. The challenges use sensors, but only as user inputs, not as environmental clues. Several groups wrote programs so that, under manual control, their vehicles could be steered to a target object, grasp the object with its front “grabber” assembly, carry it away and deposit it in another place.
3. *Single autonomous behaviors*, such as following a line or avoiding the edge of a table. The most dramatic success at this level was a robot that nosed its way through a maze.
4. *Integrated autonomous behavior*, in which different behavioral capabilities are deployed at different times as needs arise. For example, some line-following robots were able to detect when they had lost track of the line, and then perform a distinct search behavior in order to find it again. Another robot performed a mission with several stages. First, it approached an object and grabbed it; then it turned in a 360° circle, keeping track of where it detected the brightest light. Then it turned back to face that light source. Finally it carried its cargo toward the light source and dropped it after traveling a fixed distance.

About half the participants eventually accomplished or made headway with some kind of level 4 challenge; the rest progressed as far as level 3. Complicating the challenge of level 4 was a weakness in the Flogo II prototype: that step-based code, once initiated, could not be interrupted. For example, if a line-following robot has lost its line, a good recovery behavior is to swing left and right in increasingly large arcs until the line is found again. Such a behavior is naturally expressed through procedural iteration. Once the line is found, no matter at what stage of a "swing," line-seeking should stop immediately and line-following should resume. Cases such as these made clear the importance of interrupting step-based execution, and facilities to do this were added immediately after the workshop.

For most of the participants, the workshop was clearly a success. The increased stability and fuller functionality of Flogo II (offset somewhat by a new source of erratic behavior, the controller interface), the higher adult-student ratio, and the participant age distribution were clearly factors. Perhaps most important, the satisfaction of robot programming for its own sake was revealed. This appeal naturally increases with the degree of behavioral sophistication attained. Here Flogo II succeeded, where Flogo I did not, in empowering children to construct substantially more advanced robot behaviors than one typically sees in children's robotics. At this stage of development, too many factors are in flux to attempt to quantify the expressive advantage of Flogo II. The language has already improved substantially since the workshop, with the addition of interruptible steps as well as live monitoring of expressions and control structures. Given that many further improvements can be envisioned, one can only say that indications at this point are very promising.

Phase 3: Interview sessions

In the months following the robotics workshop, several of the workshop participants returned, on a semi-regular basis, to undertake a number of robotics programming projects in a clinical interview context. Three students in particular returned often and undertook a series of projects:

Larry, aged 13 , had a strong interest in technology and programming. His actual programming experience was relatively modest however, consisting of occasional exposure to MicroWorlds LOGO at school, some Java programming at home—and about 20 hours of Flogo I experience two years earlier.

Tor, aged 13, had some slight prior exposure to LOGO.

Nick, aged 15, with a strong intellectual bent, had programmed his programmable calculator, but had virtually no other programming experience.

Although not as versed in programming as some children their age, these boys were all very bright, computer-literate, technophilic, mathematically strong, and highly interested in Flogo II and robotics. Working with these boys has the effect of deferring some important questions about how to support and attract a wider range of students—questions which will no doubt entail substantial further design work. Nonetheless, by following up the earlier phases with kids who are especially ready and eager to engage Flogo II as it currently exists, we are able to make maximum use of the current prototype in probing some of the basic questions about the intellectual content of real-time programming.

Among the projects these boys worked on, these figured most prominently:

1. A line-following challenge, which Nick and Tor undertook because their first effort during the robotics workshop had not worked satisfactorily.
2. The ball-push challenge. A robot equipped with an optical distance sensor must find every ball on a table, and push each one off the table, without falling off itself. Larry and Tor each solved this separately. Later, Nick and Tor undertook a variant of the problem.

3. A Bopit game. Returning to one of the problems that launched this work several years ago, I offered Tor the challenge of programming a version of this well-known game. He worked on this over three sessions; Nick helped out during the second of these.
4. The seesaw apparatus. Undergraduate assistant Robert Kwok developed this apparatus, through several stages of refinement. We posed the very difficult challenge of making a ball roll back and forth along the seesaw without running off either end. Larry worked on this challenge for one session; Nick and Tor did as well.

In taking on these challenges, the boys achieved some major successes, and worked through some difficult problems. Under the less chaotic conditions of the clinical interview, I was able to follow some evolving themes in the boys' approaches to programming—in particular, Larry's use of "permission variables," Nick and Tor's gradual integration of process-oriented code, and contrasts in the three boys' approach to meaning and readability. These provided some useful material for the concluding reflections of this dissertation.

STEPS AND PROCESSES IN THE LEARNING OF PROGRAMMING

We have seen that Flogo II's step / process duality brings expressive power to multithreaded programming. Beginners' relationship to this power is complex. On the one hand, as examples in this chapter will illustrate, Flogo II behavior descriptions seem to match up well with children's competence in thinking about process. Once children understand a particular piece of Flogo II code, they appear to be good at reasoning about its effects and interactions. With help, children have succeeded in using Flogo II to produce complex behaviors built from a combination of steps and processes. On the other hand, children are often tripped up by Flogo II's syntactic rules regarding steps and processes, which they find, by and large, mystifying. To begin to sort this out, I first examine steps and processes from a linguistic perspective, considering how it might be that children might be differently prepared for this duality at the syntactic and the semantic levels. Next, I look at the some of the landmarks in learning to use steps and processes as building blocks for behavior.

Steps and Processes as Linguistic Categories

Over the course of their engagement with Flogo II, all children became aware that the language has two kinds of statements called steps and processes, and there are rules about where they can go. But most were not able to describe the difference confidently, and they did not use the formal classifications of constructs as step or process to reason about what statements were allowable in what contexts. Robust, usable structural understanding of steps and process was, in general, not achieved in the time available.[30] How, then, did children manage to compose legal, functioning programs? By four means: semantic guidance, use of familiar patterns, trial and error, and help from others.

By "semantic guidance" I mean that there is some tendency for children's process descriptions to be well formed from Flogo II's point of view, simply by virtue of coherently describing the programmer's intentions. Suppose that, in the program's response to some event, the child wants an effect involving playing a trumpet sound, increasing the player's score, and moving a sprite east. Then there is at least a fair chance that she will correctly write:

[30] At least one subject, Larry, showed substantial progress toward this formal understanding. We return to his comments later.

```
Every time <some event>
    playsound(‘gasp)
    set score to score + 1
    for 2 seconds move(‘joe,’east)
```

Playing a “gasp!” sound effect is normally thought of as a discrete action, as is incrementing a score. The move command, on the other hand, describes a state of moving, which the child has experienced by controlling move commands directly. In thinking about what she wants, she may even have considered a long move versus a short move. In this case she will almost certainly write as above. Conversely, almost never will she write…

```
For 3 seconds set score to score + 1            ; illegal
```

…because it just doesn’t make sense to add 1 to a variable for 3 seconds. Similarly, since playing the gasp sound takes whatever brief amount of time it takes, not 3 (or any other externally imposed) number of seconds, one will almost never write

```
for 3 seconds playsound(‘gasp)                  ; illegal
```

The exception is the unusual case (which happened once during Flogo II piloting) that one actually wants the sound to repeat for the given time period, and one assumes such repetition to be the meaning of the above. This points up one of the main ways that semantic guidance can fail: when a believable English interpretation exists for an illegal language construct. In the case of our little example, this is most likely with the word “move.” In English, “move” has a couple possible interpretations as a discrete action: either (i) “make a movement” or (ii) “start moving.” These will account for the most likely malformation:

```
Every time <some event>
    playsound(‘gasp)
    set score to score + 1
    move(‘joe,’east)           ; illegal
```

When a language is used successfully by means of semantic guidance, we call it transparent (Hancock, 1995). However, the potential for Flogo II to be transparent in this sense is greatly reduced by the semantic overloading of English expressions relating to temporal process. That is, many English expressions have both a process and a step interpretation. English verbs (like “move”) are one major example. In normal usage the distinction between complete action vs. continuing process (called “aspect” in linguistics—e.g. Comrie, 1976) is made not by explicit markers, but by context. For example:

Photographer, to portrait subject: “Smile!” (i.e. once, or briefly, for a photo—step)

Dance director, to dancer: Smile!” (i.e. as you’re dancing—process)

Slavic languages, by contrast, require a different form depending on whether a completed or continuing action is meant. For example:

Photographer, to portrait subject: “Ulibnis!” (i.e. perform a smile)

Dance director, to dancer: “Ulibaysia!” (i.e. keep smiling)

If a version of Flogo II were built in a Slavic language, a significant class of user errors might be avoided. (This is a very researchable question.)

Talking about steps and processes in plain English can feel like trying to tie one’s shoes while wearing mittens. We try to disambiguate, but struggle to find adequate linguistic resources. One example of that struggle comes when there are step-oriented and process-oriented ways to control the same facility. For example, Flogo II has two ways of controlling the appearance and disappearance of screen sprites. First to be implemented were the process forms SHOW and HIDE. Hiding and showing, like a number of other facilities in Flogo II, obey a simple priority scheme: the most recently invoked process shadows the effects of any others. So, for example:

```
For 5 seconds
    SHOW(‘monkey)
    EVERY 5 TENTHS for 1 tenth HIDE(‘monkey)
```

…will allows us to see the sprite named ‘monkey for five seconds, briefly blinking off every half second. Children grasp this priority system remarkably well, and can predict the interaction effects of different HIDE and SHOW processes quite accurately. However, children had some difficulty using these commands to achieve certain effects in their games, such as the eradication of an enemy from the game. The problem is that if you HIDE a character, it remains hidden only as long as the HIDE process is in effect. Every invocation of SHOW or HIDE implies its own eventual undoing (with the exception of the top level of the program, where one can put a SHOW statement, turn it on and forget about it). In response to children’s requests, I added two step-oriented counterparts to HIDE and SHOW, to change the sprite’s visibility in one direction only. Casting about for some distinguishing name for these primitives, I named them “shownow” and “hidenow.” A sprite with no further role in a game could now be dismissed with hidenow. At the next afterschool session I introduced these new commands to the group. The children quickly grasped the utility of these new primitives, and were soon able to predict their interactions with the process versions as well. I asked if the names hidenow and shownow made sense, and if they help to explain how these commands differ from HIDE and SHOW. Heads nodded. One child said yes, “because it hides it, like, now.”

One can perhaps take heart from the fact that the children and I were able to understand each other, despite our inability to express clearly what it was that we understood. The name hidenow is in my opinion a misnomer, not denoting but, at best, alluding to its action via an instantaneous effect. The word “now” can mean “at this instant” but it can equally well mean “during this period,” so it really does not disambiguate steps from processes.

English does have one resource for distinguishing continuing activities from discrete acts, namely the progressive form –ing. One could imagine using -ING for all process names in Flogo II:

```
For 5 seconds
    SHOWING(‘monkey)
    EVERY 5 TENTHS for 1 tenth HIDING(‘monkey)
```

Or perhaps

```
For 5 seconds
    BE SHOWING(‘monkey)
    EVERY 5 TENTHS for 1 tenth BE HIDING(‘monkey)
```

Based this, one could make a standard form for step-based counterparts using “start” and “stop”:

```
    start hiding(‘monkey)

    stop hiding(‘monkey)
```

It is possible that such a terminology would be consistent and unambiguous enough to make up for its ugliness. But I doubt it. One of the difficulties is already evident in this example. We intended “hiding” to describe a period during which an object is hidden. However, another reasonable interpretation of “hiding” might be as a *transition* from visible to hidden, e.g. by gradually fading the icon out. That is, although the word "hiding" clearly indicates a process and not a step, it remains semantically overloaded, being able to describe either a state ("the clouds are hiding the sun"), or a transition ("Igor, have you finished hiding the specimens?").

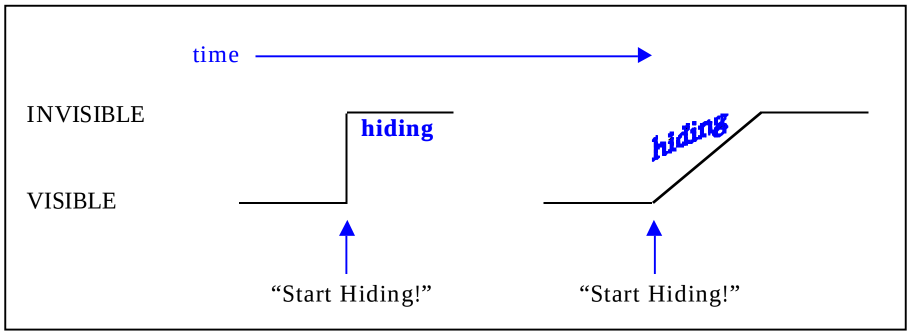


Fig. 1: Hiding as state vs. hiding as transition, leading to different meanings of "start hiding."

Coincidentally, around the same time that I was struggle with names like hidenow and shownow, my one-year-old son was confronting, I believe, a variant of this problem. He could say "up" to be picked up, and "down" to be put down. However, he often reversed the words. There is a natural tendency for confusion because we have only two words, "up" and "down," for talking about four things, namely:

The transition from down to up

The state of being up

The transition from up to down

The state of being down

Which states and transitions get the same names is really just a convention. Usually we name transitions along with the states that they go *to*—but not always: the west wind comes *from* the west. Meanwhile, the empirical evidence can be confusing. Try singing "the door on the bus goes open and shut" while manipulating the bus door in a pop-up book. Thinking of "open" as a transition, you may choose to sing "open" as you begin to open the door. But that means that at the moment you sing "open," the door is actually shut (and vice versa). On the other hand, you might wait and sing "open" once the door is completely open, and "shut" once it is shut. In that case your audience may observe that you sing "open" immediately before shutting the door. In either case, a one-year-old, or indeed any newcomer to the English language, may be excused for drawing the wrong conclusion.

What all of this helps to show is that one cannot resolve the ambiguities of processes and steps by describing them as states and transitions, respectively, because transitions themselves can sometimes be processes. In fact, steps and processes are categories that each subsume a large variety of "eventualities," as the linguists call them. A frequently cited paper by Bach (1986) partitions the space of eventualities as in Fig. 2.

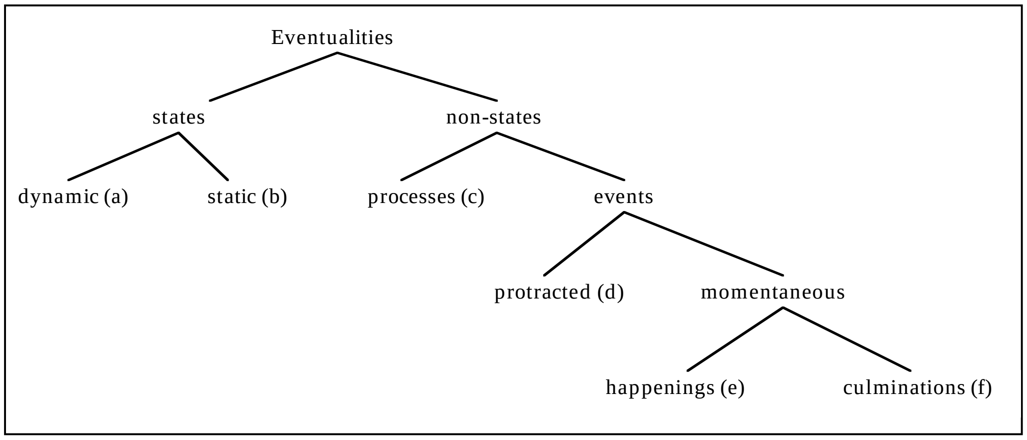


Fig. 2: Kinds of eventualities (Bach, 1996). Bach's examples are:
(a) sit, stand, lie + LOC
(b) be drunk, be in New York, own x, resemble x, love x
(c) walk, push a cart, be mean
(d) build x, walk to Boston
(e) recognize, notice, flash once
(f) die, reach the top

The important lesson from linguistics, however, is that despite this variety, the distinction between discrete, self-completing steps (or "events," as Bach calls them), versus continuing, temporally unbounded processes (including "processes and "states" in the diagram), has proven fundamental in linguistic analysis, helping to account for a wide range of linguistic phenomena. Moreover, this is true whether the language in question includes explicit aspect markers or not. For example, aspect explains why the sentences on the left seem normal, while those on the right sound "wrong," or require extra work to interpret (examples Bach's):

John slept a lot last night	John found a unicorn a lot last night
John fell asleep three times during the night	John slept three times during the night.

Bach's paper develops a formal "algebra of events," in which the key principle is that events can be understood as time-bounded "portions" of continuous processes, very much the way Flogo II forms steps from processes.

Bach's discussion and formal algebra take their inspiration from an oft-noted comparison between aspect and another linguistic distinction, that between mass nouns (e.g. beer) and count nouns (e.g. dog). The constitution of events as time-bounded process parallels that of count nouns from bounded masses, as in "a glass of beer." In a manner that parallels what we have already seen with steps and processes (in the case of playing "gasp" for three seconds, for example), English speakers are often able to use mass nouns as count nouns and vice versa. For example, although mass nouns such as "beer" do not, qua mass nouns, form plurals, nonetheless, "beers" will be interpreted to mean "servings of beer" or "kinds of beer." Conversely, the word "dog," ordinarily a count noun, is interpreted, in a sentence like "there was dog all over the road," to refer to the material that normally constitutes dogs. These tropes do not discredit the importance of count vs. mass as a fundamental semantic distinction. On the contrary: the distinction is essential to understanding how their meanings operate.

Event vs. duration in English temporal conjunctions

Discussion to this point has focused on English verbs and the reasons why they do not always help us distinguish step statements from process statements. But the semantic overloading we find in verbs also extends to many other parts of the language, including conjunctions such as “if,” “when,” and “while,” typically used for naming control constructs. This is another area where Flogo II beginners often had difficulty.

A thorny example is the word “when.” In Flogo II, when is the conditional construct for processes:

`when sensor(5) > 60`
`    motor1(‘on)`
	Meaning: continue to switch motor1 on and off in tandem with the truth of the condition.

But some illegal uses of when seem equally reasonable in English:

`when sensor(5) becomes > 60`
`    beep()`
	Meaning: (in a process context) monitor for threshold crossing event, and beep whenever it happens.

In Flogo II, the correct way to say this is with EVERY TIME. But this usage of “when” is not uncommon in other programming languages as a way to set up a “demon” process. Here is another one:

`when sensor(5) > 60`
`    beep()`
	Meaning: (in a step context) check condition once and beep if it was found to be true.

As it happens, this is exactly what “when” means in LISP. But in Flogo II the correct word would be “if.”

“When” is certainly one of the more slippery time-words, and should arguably have been avoided altogether in Flogo II. Among the meanings for “when” listed in my Encarta dictionary are these:

1. at or during the time that
2. as soon as somebody does something or something happens
3. at some point during an activity, event, or circumstance
4. each time something happens

Flogo II Construct	Meaning	Construct Type	Inner Type	English equivalents
WHEN	Turn inner process on and off as conditions change	Process	Process	When, whenever, while, as long as, if
AS SOON AS EVERY TIME *(alt.)*	Launch inner steps each time some sort of event occurs.	Process	Step	When, whenever, if, as soon as, every time
AS LONG AS	Enable inner process for a period of time ending when condition ceases to hold.	Step	Process	While, as long as
IF	Check condition once and, if true, execute inner steps	Step	Step	When, whenever, if, every time, as long as

Table 1: Possible names for four related control constructs in Flogo II

Other time phrases, while more specific than “when,” still have some ambiguity. Fred Martin once told me that he considered using “whenever” for an event monitor (second row of table), in order to contrast with “when” as a conditional (fourth row). The “ever” seems to highlight the sense of the statement applying over an expanse of time. But plain old “when” can mean the same thing. On reflection, one has to agree with the dictionary, which says defines “whenever” as an intensive form of “when” and nothing more. Much as in the case of my “hidenow,” adding intensity suggests a punctate event, but does not clearly denote it. While these ambiguities must be confronted in any language, Flogo’s repertoire of control constructs is larger than most, and thus stretches the lexicon especially thin.

Design implications of the linguistic analysis

What can we learn from all this concerning programming language design? The distinction between steps and processes is found to have strong cognates in the linguistic category of aspect and, beyond that, in the distinction between mass and count. Flogo’s distinction between steps and processes parallels precisely the distinctions between discrete and continuous process, as well as discrete and continuous matter, which underlies our language and by extension, our cognition. I take this as a kind of corroborating evidence in favor of the primary importance that Flogo II gives to the distinction. There are two kinds of corroboration. First, the linguistic evidence suggests that, however hard it is to talk about, the distinction between steps and processes is something that, at some level, we *already know*. If a bridge to that knowledge can be built (a question I will come back to in a moment), then users of Flogo II will find that their prior knowledge gives them a head start in using the language, and that their intentions are relatively easily translated into the language. The second kind of corroboration concerns generativity. If Flogo’s fundamental category system is equivalent to one that has emerged from hundreds of thousands of years of speaking and thinking about processes, then that bodes well for its effectiveness in generating a useful range of process descriptions.

This good news comes with some important qualifications, however. First, opinions vary concerning the accessibility of our knowledge of linguistic categories. If one holds a Chomskyan image of human language and cognitive capabilities locked inside specialized mental devices, beyond the reach of introspection (Jackendoff, 1994), then modeling a computer language after grammatical aspect may not make much more sense than modeling it after the anatomy of the gall bladder. Another Chomskyan might reason that by better conforming to innate expectations, Flogo II might indeed come more within the purview of innate linguistic abilities, but caution that these abilities would be of little avail in understanding the step / process distinction at a structural level.

On the other hand, adherents of cognitive linguistics (Janda, 2000) view grammatical knowledge not separate from, but rather tightly interwoven with, semantic and other cognitive capabilities. If they are right, the prospects for building on children’s linguistic knowledge in learning about steps and processes may seem stronger. The difference is not so clear-cut, however, since the cognitive linguistic view is less likely to see grammatical knowledge as taking the form of abstract rules. Abstract grammatical formulations that Chomskians see as locked up in a specialized facility, cognitive linguists will see as post-hoc descriptions of emergent patterns of thought and action. Either way, then, there is no free lunch. Flogo II’s formal distinction between steps and processes will be a new idea to learners, and one that they will have to construct gradually.

However—and this is what I am suggesting—if the step / process is deeply, albeit implicitly, woven into our ways of talking and thinking about situations and happenings, then we have a large base of intuitions on which to build, gradually, a formal understanding of steps and processes. Moreover, such understanding will be amply rewarded by a expanded ability to describe computations in terms that make intuitive sense.

Given that we have limited linguistic and cognitive resources for distinguishing steps and processes explicitly, some might argue that Flogo II would rock the boat less if it behaved like a human interlocutor, using contextual cues to infer the programmer's intentions regarding discrete vs. continuous computation. In the context of the broad design agenda I laid out in Chapters 1 and 2, I do not think this would be appropriate. Trying to spare learners the explicit distinction between discrete and continuous activity might make sense in a tool meant for short-term use. Such a design approach would be, in DiSessa's (2001) terminology, more functional than structural. But over the longer term, the ability of structure to take us beyond what we knew before, multiplying our creative powers, gives a structure-oriented design the decided advantage.

Moreover, discussions from previous chapters strongly suggest that even if one wanted to, it might not be possible to protect learners from the eventual need to distinguish discrete and continuous time. Having seen variants of it surface in a variety of learner difficulties—in C programming in undergraduate robotics, in MultiLogo programming, in Flogo I's pulse / data distinction, and in invented musical notations—one gains the impression that avoidance of the issue may do more harm than good.

The question remains, however, of how to deal with the difficulty of the step / process distinction, recognizing that it may take young programmers a long time to master it. In particular, what might the programming medium do to help? I suggest improvements in editing, representation, and terminology.

Editing improvements. Menus should be given a larger role in program construction in Flogo II. In a process context, menus would offer process statements, and in a step context, they would offer step statements. Like Flogo I's strongly typed pulse wires, this feature would head off illegal constructions. But its presentation of available constructs would constitute a more positive kind of help, as well.

The editor can also help by turning illegal statements into a scaffold for legal ones. For example, if the user types

```
Every time <some event>
    playsound('gasp)
    set score to score + 1
    move('joe,'east)            ; illegal
```

The system can automatically transform this into:

```
Every time <some event>
    playsound('gasp)
    set score to score + 1
    <<for how long>>
        move('joe,'east)
```

<<for how long>> is a dummy construct, readily transformable, with menu assistance, into a meaningful construct such as "for 1 second" or "as long as mousedown()." Conversely, steps placed in a process context would be automatically wrapped in an "<at what point>" dummy construct. In addition to sometimes (not always!) helping the programmer express her intentions, this feature provides some implicit explanation of why a misplaced statement doesn't belong where the programmer has put it.

Representation. Process statements and step statements are visually distinguished by the controls beside them—square for processes, round for steps. (They are also distinguished by uppercase vs. lowercase, but this is less salient to learners). There is, however, no visible difference between process and step blocks. Since Flogo's process / step rules concern the allowability of certain statements in certain blocks, the goodness of fit between statement and block ought to be visible. Attempts to date have not yielded a visual distinction that is clear, uncluttered, attractive, and meaningful; however, I believe one is possible.

Construct Names. Given the looseness of the English language with respect to events and durations, it seems clear that no set of names will perfectly elucidate the meanings of the different Flogo constructs. However, it also seems likely that the current set can be improved upon.

In combination, these changes in editing, representation, and nomenclature should go some way toward easing learners' engagement with steps and processes in Flogo II, but just how far remains to be seen. How best to introduce learners to the idea of steps and processes, and how both pedagogy and further media developments might best support them in programming successfully as they gradually learn and master the distinction, are important questions that this dissertation aims only to raise, not to resolve.

Steps and Processes as Computational Building Blocks

Even as Flogo II's syntax rules regarding steps and processes continued to puzzle them, children moved forward in their ability to use steps and processes in creating behaviors. There being no obvious limit to the potential complexity of process coordination strategies in advanced programming, the path of mastery of steps and processes is clearly a very long one. However, even some of the early steps along the way can be quite empowering, and children's project work with Flogo II provides some glimpses of these.

Interestingly, many of the examples involve the use of variables. Why should this be so? A variable is a kind of common ground between steps and processes. A variable is an enduring structure that can be accessed in both process-like and step-like ways. A variable might be maintained by steps and used by processes; or it might be maintained by a process and used by steps. In such cases, the variable serves as an articulation point in the program, making it possible for the programmer to choose to use a process to solve one part of the problem, and steps to deal with another. More generally, a variable can be used as a "steady frame" for communication by program components, whatever their combination of steps and processes, that are proceeding on independent schedules.

Tor and Nick and the use of Process-based Variables

Quite early in his programming experience, Tor took on this challenge: make Ms. Solomon's head shake faster and faster as you bring the mouse closer, and slower and slower as you move the mouse away. A concise solution might look like this:

```
Dist: distance('mouse,'ms)
Shaketime: dist / 10
Repeatedly
    For shaketime tenths show('ms,'noleft)
    For shaketime tenths show('ms,'noright)
```

Note that this problem has an inherent requirement for coordination of steps and processes, in that mouse distance is a continuously varying quantity, while head shaking is a sequential activity. The sample solution uses process statements to convert distance to an appropriate time interval (shaketime), which is then accessed by the head-shaking code. The variable shaketime is thus the articulation point between steps and processes in the program. Conceiving head-shake duration as a continuous function of mouse distance is, however, a relatively sophisticated maneuver, and it does not occur to Tor. Instead, over about an hour's labor, Tor developed this very large program (here edited slightly for readability):

```
D: distance('person,'monkey)
F:{10}
set F to 20
EVERY F TENTHS
    For F/2 tenths show('person,'noleft)
    For F/2 tenths show('person,'noright)
EVERY TIME D BECOMES < 300 set F to F - 1
EVERY TIME D BECOMES < 280 set F to F - 1
EVERY TIME D BECOMES < 260 set F to F - 1
EVERY TIME D BECOMES < 300 set F to F - 2
EVERY TIME D BECOMES < 240 set F to F - 2
EVERY TIME D BECOMES < 220 set F to F - 2
EVERY TIME D BECOMES < 200 set F to F - 3
EVERY TIME D BECOMES < 180 set F to F - 3
EVERY TIME D BECOMES < 160 set F to F - 3
EVERY TIME D BECOMES < 140 set F to F - 4
EVERY TIME D BECOMES < 120 set F to F - 4
EVERY TIME D BECOMES < 100 set F to F - 4
EVERY TIME D BECOMES < 80 set F to F - 5
EVERY TIME D BECOMES < 60 set F to F - 6
EVERY TIME D BECOMES < 40 set F to F - 6
EVERY TIME D BECOMES > 300 set F to F + 1
EVERY TIME D BECOMES > 280 set F to F + 1
EVERY TIME D BECOMES > 260 set F to F + 1
EVERY TIME D BECOMES > 240 set F to F + 1
EVERY TIME D BECOMES > 220 set F to F + 1
EVERY TIME D BECOMES > 200 set F to F + 1
EVERY TIME D BECOMES > 180 set F to F + 1
EVERY TIME D BECOMES > 160 set F to F + 1
EVERY TIME D BECOMES > 140 set F to F + 1
EVERY TIME D BECOMES > 120 set F to F + 1
EVERY TIME D BECOMES > 100 set F to F + 1
EVERY TIME D BECOMES > 80 set F to F + 1
EVERY TIME D BECOMES > 60 set F to F + 1
EVERY TIME D BECOMES > 40 set F to F + 1
```

(An aside: A solution like this, horribly cumbersome though it may appear, may be taken as a welcome sign by the teacher of programming. The good news is that Tor has devised a way to use the language's mechanisms to achieve his goals. The mechanism-devising orientation is a great advance over the more naïve expectation that for every desired behavior there is a "proper way to say it" in the programming language. Until he distinguishes knowing the language from using it to solve problems, the beginner is not yet fully "in the game" of programming. Tor, although not an experienced programmer[31], is clearly in the game.)

This program may be thought of as setting up a series of concentric circles around the 'ms sprite. As the mouse moves in across each circle, a decrementing of the variable is

[31] This was Tor's fourth Flogo II session. He had very little prior programming experience.

triggered. Likewise, crossing circles on the way out causes the speed to increase. Because the increments and decrements do not match up exactly, the program exhibited some peculiarities of behavior. When treated gently, however, it was clearly capable of generating the intended effect.

In his solution I believe Tor was adapting previous knowledge of how to use steps to generate cumulative effects in a variable. The prototype for this may well have been a programming technique that was in circulation in the group, allowing players of chase games to gain or lose points when a character comes in contact with a quarry or an enemy, respectively, e.g.:

```
When distance(<me>,<thing-I'm-chasing>) becomes < 50
    Set score to score + 1
```

Or

```
When distance(<pursuer>,<me>) becomes < 50
    Set score to score - 3
```

Tor's innovation was to replicate this effect at a series of distances from the central character. By treating distance in terms of a series of discrete segments, he created a framework in which his knowledge of managing a variable through discrete transactions could be applied.

Appropriating process-based variables

I suspect that in many instances learners find the step-oriented management of variables, of the kind that Tor developed, initially simpler to construct than process-oriented definitions of the sort found in my suggested solution (SHAKETIME: dist/20). A process-based definition abstracts over time, defining a formula or other process that will hold in all applicable circumstances. Tor's solution, by contrast, associated each EVERY TIME statement with a specific situation, making it easier to imagine the computation that one is specifying. Even when step-based formulations are more general, they may still be cognitively less challenging, simply by virtue of being expressed in terms of singular moments that can be easier to imagine.

However, process-based variables can sometimes confer advantages that learners find compelling. In the following examples we can see these advantages very gradually winning two young programmers over to the beginnings of a process-based approach.

In piloting Phase 2, Tor and Nick began working as partners on a series of robotics projects. Under Nick's influence, this team initially programmed in a strongly step-oriented style, not just in their management of variables, but in their program control structures. For example, rather than create a robot forward procedure as many other participants learned to do:

```
define process forward()
    motor1('on,'thisway)
    motor2('on,'thisway)
```

…they would instead write:

```
define step forward(time)
    for time seconds
        motor1('on,'thisway)
        motor2('on,'thisway)
```

Using routines such as this, their programs would proceed by repeatedly moving in tiny increments, fd(0.1). Flogo II's "on step" construct, which repeats a series of steps indefinitely, was a lesser-known feature among the group at large, but for Nick and Tor it was a staple. So was "do / while," which allows a series of steps to be interrupted and stopped whenever a condition changes.

On two separate occasions, Nick and Tor worked on a line-following challenge. The first time was in the robotics workshop involving eight children. When I checked in on them, they were having trouble understanding why their robot was not behaving correctly. The robot had two forward reflectance sensors positioned over the floor. Various parts of their program involved testing to see whether these sensors readings indicated the robot was over the tape. I showed them a helpful technique; to make two derived variables to represent whether each sensor is above the line. I called these RON (right sensor on the line) and LON (left sensor on the line).

```
RON: {TRUE} RIGHT > 20
LON: {FALSE} LEFT > 20
```

Nick and Tor liked this contribution their program. They appreciated that by watching each variable's constantly updated value, which Flogo II displays automatically, one can see immediately whether the program "thinks" either sensor is over the line. This is useful during normal program execution, and also when manually holding the robot in various positions, to test the veracity of RON and LON. It also provides a single point for editing threshold values, and guarantees that all uses of the sensors are consistent.

Nevertheless, the line follower did not work out as well as some of their other projects, and so when they were reunited a couple months later, they decided to give it another try. They worked on their own for a while, starting from scratch. When I joined them, I saw that they were back to their step-oriented ways. The "main" portion of their program read as follows:

```
ON STEP[32]
    do
        rt(0.1)
    while L < 15 and R > 15
    do
        lt(0.1)
    while R < 15 and L > 15
    do
        scan()
    while R < 15 and L < 15
```

They were having considerable trouble with this program, due in part to unreliable sensor readings because of uneven lighting. As they tinkered with the condition expressions governing when to do what, they would occasionally use less than (<) where they meant greater than (>), and vice versa. Conditions for the different actions were not always changed in parallel ways, so the robot sometimes had too many things to do, and sometimes not enough. As the robot jerked left and right erratically, and the ON STEP loop flickered round and round, it was very hard to follow what the program was doing.

To help them get control of the situation, I made two suggestions. First, a lighting change, to improve sensor readings. Second, I reminded them of the RON and LON variables I had shown them some weeks earlier, which they had liked at the time but had not brought over to their new project.

```
LON: L > 50
RON: R > 50
```

[32] ON STEP means the same as REPEATEDLY. It is a process statement containing steps that are repeated over and over as along as the process block is active.

Once these process statements were turned on, each variable was labeled with a constantly updated TRUE or FALSE. One could manually position the robot in different ways to see the effect on the variables, and while the program was running, it was easier to check what the robot was "seeing." Moreover, the main program code became easier to write. Nick and Tor quickly adapted their program, which came to look like this (also adding in previously missing code to let the robot go forward):

```
ON STEP
    do
        rt(1)
    while not LON and RON
    do
        lt(1)
    while not RON and LON
    do
            scan()
    while not RON and not LON
    do
        fd(1)
    while RON and LON
```

This time, it appears, the role of RON and LON in bringing needed manageability to their program was dramatic enough to make a more lasting impression. The following week, in his work on a ball-pushing robot, Tor spontaneously adapted the RON&LON approach. The challenge is to repeatedly find a ball on the table and push it off the edge, until no balls are left. While pushing a ball, the robot needs to continue to monitor the ball's presence. If the ball is no longer in front of the robot, it may mean that either the robot has lost control of the ball and must search for it again, or that the ball has fallen off the edge of the table, and the robot will fall next unless it stops moving forward. Tor created a process variable to continually monitor the ball's presence:

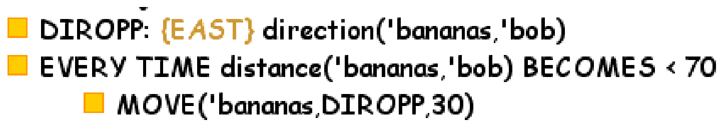
```
HAVEBALL: WHITE_DISTANCE() > 40
```

In subsequent projects as well, the declaratively defined variable became a standard part of both Tor and Nick's repertoire, and we see it again in their creation of a SPEED variable in the seesaw episode described in the Prologue.

Ben's Bananas Game and the Construction of Game State

Ben was the creator of the bananas takeaway game mentioned earlier. In its final form, the game allows one player to control a pirate by means of the arrow keys, while a second player controls a monkey by means of the mouse. These contestants are well matched: the pirate moves faster, but must always travel left, right, up, or down. The monkey moves more slowly but in any direction. At any point one of the characters is in possession of the bunch of the bananas, which it can retain as long as it avoids the other player. If the other player gets close enough, the bananas will stay with him instead.

Ben worked on this project for a couple sessions, during which time he experimented with various uses of sound and text and various patterns of movement. Initially the pirate and monkey pursued the bananas automatically rather than under player control. When I checked in with him, he was struggling to get a proper back-and-forth between the two contestants. A key part of his program looked like this:



```
■ DIROPP: {EAST} direction('bananas,'bob)
■ EVERY TIME distance('bananas,'bob) BECOMES < 70
    ■ MOVE('bananas,DIROPP,30)
```

Fig. 5: An early portion of Ben's takeaway game, intended to allow bob (the monkey) to take away the bananas.

The intent of these lines was to effect the "taking away" of the bananas by the monkey, named Bob. When Bob gets close enough to the bananas, the MOVE statement is meant to come into effect, causing the bananas to stay close to the monkey by following it at high speed. However, the program did not work. Since EVERY TIME is a construct used to trigger steps when an event occurs, Ben's use of the process MOVE inside the EVERY TIME statement was illegal. Flogo II's response, at the time, was simply to leave the MOVE statement running, so at this point the bananas were always following the monkey.

Why did Ben write the program this way? Clearly there is an important relationship between the *event* of the monkey getting close to the bananas, and the *process* of the bananas following the monkey. By putting the two together using EVERY TIME, Ben was perhaps envisioning EVERY TIME as meaning something along the lines of FROM THE MOMENT THAT—that is, a construct that responds to an event by initiating a process. This is quite a reasonable guess. Although Flogo II has no FROM THE MOMENT THAT construct, there are ways to achieve what Ben was envisioning, e.g.

```
EVERY TIME DISTANCE ('bananas,'bob) becomes < 70
    forever
        MOVE('bananas,DIROPP, 30)
```

This formulation, however, points up an important weakness in Ben's plan: there is no allowance for the possibility of the bananas ceasing to follow the monkey when taken away by the pirate. This might be remedied as follows

```
EVERY TIME DISTANCE ('bananas,'bob) becomes < 70
    until distance('bananas,'joe) becomes < 70
        MOVE('bananas,DIROPP, 30)
```

(… in combination with a corresponding clause for Joe, the pirate). Had forever and until clauses existed at the time of this episode, it might have been appropriate to show them to Ben. They did not exist, however, so I instead suggested a different approach: to create a variable called OWNER (i.e. current owner of the bananas), which would at all times hold either 'joe or 'bob. Then the game mechanism can work like this:

```
OWNER: {bob}
OWNER_DIR: DIRECTION('bananas,OWNER)
MOVE('bananas, OWNER_DIR, 30)
```

The variable OWNER puts the possession information in one place, where the banana moving code can easily access it. Because the variable OWNER attaches a constant meaning to a specific location, we can call it a steady frame representation. It accomplishes this by moving up one level of abstraction, from asking if Bob has the bananas or Joe has the bananas, to asking who has the bananas. Ben was not ready to make this leap himself, but when I made it for him, he took on the task of fitting it into his program. He knew that he needed to update the OWNER variable each time that possession should change hands. He wrote

```
WHEN DISTANCE('bananas,'bob) BECOMES < 70
    set OWNER to 'bob
```

This is almost right, except that WHEN, Flogo II's process-oriented switching construct, has been used as though it were an event handler. This is a prime instance of the linguistic difficulty mentioned earler: it makes perfect sense to say "when the distance becomes less than 70, do X." I suggested he use EVERY TIME, which he did. He then added the converse statement for the other character. He now had all the code necessary to maintain the OWNER variable:

```
EVERY TIME DISTANCE ('bananas,'bob) becomes < 70
    set OWNER to 'bob
EVERY TIME DISTANCE ('bananas,'joe) becomes < 70
    set OWNER to 'joe
```

Like other real-time variables, OWNER not only simplifies programming, but also helps to make the program easier to follow during program execution. In this case, owner is maintained not by a process expression, but by event-triggered steps. Both can be ways of maintaining useful real-time variables.

In all of the examples described so far, we see the definition of appropriate variables as a key task in programming. In each case, moreover, we can see that the variable-based representation requires a small but significant step in abstraction. The variables RON, LON, and HAVEBALL, for example, all use Boolean data to describe whether a certain condition is true. this is, I suggest, a little more abstract than using the condition to make an immediate decision, because the truth-or-falsity of the condition is reified as boolean data. Liewise, in the takeaway game, it is more direct to try to say "when the pirate has the bananas, do X, and when the monkey has the bananas, do Y." But the OWNER variable requires one take one step back and maintain an answer to the question "who has the bananas?" The reward for doing this is the ability to write a generalized statement that makes the bananas follow the owner, whoever the owner may be.

Sometimes making a program variable requires us to use or invent an abstraction that we are unfamiliar with. This was somewhat true of Nick and Tor 's experience with HAVEBALL, because neither had prior experience programming with formal Boolean data. But more often, the needed abstraction step uses concepts that are very familiar. The problem is not to comprehend some exotic new abstraction, but simply to think of generalizing in the right direction. In this sense it is not surprising that Ben quickly understood the OWNER variable once I had supplied it for him. The question is, what additional experiences will Ben require in order to be able to make such moves for himself? I think we see a clue to the answer in Tor's invention of a HAVEBALL variable. HAVEBALL is a very close parallel to the RON and LON variables that I twice provided to Nick and Tor. Based on these examples, one can postulate two components of development in the use of variables in programming: first, the accumulation of a wider range of experiences with variables, increasing the range of problems that one can solve through a very near parallel to a known class of cases; and second, the gradual loosening of the requirement that new abstraction steps parallel prior ones so closely. How these developmental components can grow and interact is an important question for further research. A third component waits a little farther down the road: the ability to make abstraction steps into previously uncharted conceptual terrain. We saw the beginnings of this in the Prologue to this document, a Nick and Tor groped toward a definition of instantaneous speed. The next chapter further considers the educational meaning of these sorts of abstraction steps, and the potential role of live programming environments in fostering them.

Chapter 7: Live Programming and the Development of Conceptual Systems

At first glance, a live interface seems like a simple gift from language designer to programmer. Looking more closely, we find that liveness is most helpful when used and managed strategically. I will briefly describe some examples of children's active management of program visualization, and of liveness as a factor in children's programming decisions. We consider programming-for-liveness as an intellectual endeavor and connect it, via the concept of the steady frame, to much more general processes in learning, development, and intellectual life.

In the second half of the chapter I will discuss what these considerations imply for the educational significance of real-time programming, and conclude with a review of the outcomes of this study as a whole.

PROGRAMMING FOR LIVENESS

The field of software visualization explores ways of revealing the execution of computer programs at a variety of levels, ranging from the detailed rendition of every step of a computation, to higher-level portrayal of the progress and meaning of algorithms (Price, Baecker, & Small, 1998). In the latter category especially, the recent trend has been away from fully automated visualization and toward the provision of toolkits that allow programmers to decide which aspects of execution will be tracked, and what form they will take in the visualization display (e.g. Hundhausen & Douglas, 2000). Still, constructing a program visualization remains a separate activity from programming itself. The up-front effort and context shift entailed in beginning a visualization project for one's program is a serious practical barrier to the routine use of visualization systems by programmers.

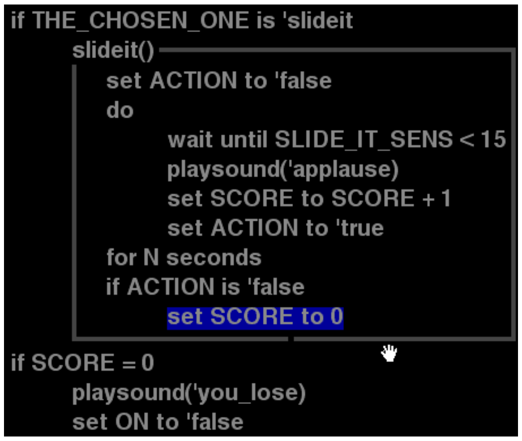
In a live programming environment that allows incremental extension of its visual interface, this situation is changed. Visualizations that support understanding of the program can be added gradually, as needed, within the program itself. There is no global up-front cost and no context shift. The tools for doing this are currently more developed in Flogo I than in Flogo II, but one can readily imagine similar interface construction tools in Flogo II, which could be used to intersperse graphical displays and controls throughout a program. Another simple extension to Flogo II would be to allow customization of "thermometer" displays for defined steps and processes.

Even without these additions, children's use of Flogo II has begun to show what it means for programming and program visualization to join together as a unified activity. In a way, it is a natural extension of things that programmers have always done—indenting, spacing, organizing, naming—to make their programs more readable as they write them. But readability of a live program is qualitatively different, because it concerns readability not just of the code but of its execution.

We have seen this already in Nick and Tor's ready adoption of the RON and LON variables in their line-following program, and their subsequent use of the same idea in variables such as HAVE_BALL and SPEED. These variables were useful for organizing their code, but what struck the boys most immediately, I believe, eliciting comments like "cool," was the capacity of these variables to provide a real-time view of an important part of the program's execution.

As the use of such variables gradually increased, I noticed that children were taking measures to ensure visibility of these variables. For example, when Tor's ball-pushing program grew too big to fit on the screen, TOR made a duplicate HAVE_BALL line near the bottom of the program, just so he could keep it visible as he worked on different parts of the program.[33] In subsequent programs he often kept real-time variables together in one section of the program, so that they could be monitored as a group. In his Bopit project, he had a section near the bottom reserved for variables that he wanted to monitor. When a variable's behavior became well understood and ceased to be problematic, he would move it up to another section of the program.

Developing a variable that is visually and computationally effective can be a struggle. In his Bopit project, Tor had a variable called SCORE. As the name suggests, this variable was used to track the player's score during a game. Each time the player made a correct response, SCORE would be incremented. However, the variable had another use as well. If the player failed to respond in time, SCORE would be set to zero. Another chunk of code, on detecting this, would play a "you lose" sound and set another flag variable causing the entire game loop to stop (See Fig. 1). This dual use led to some problems. For one thing, one could see one's score while playing the game, but as soon as the game ended, the score had been zeroed out. Also, Tor had a HIGH_SCORE variable that he attempted to update after each game. He soon realized that the zeroing of SCORE was preventing the high score from being updated.



```
if THE_CHOSEN_ONE is 'slideit
    slideit()
        set ACTION to 'false
        do
            wait until SLIDE_IT_SENS < 15
            playsound('applause)
            set SCORE to SCORE + 1
            set ACTION to 'true
        for N seconds
        if ACTION is 'false
            set SCORE to 0
if SCORE = 0
    playsound('you_lose)
    set ON to 'false
```

Fig. 1: Bopit program fragment showing use of SCORE variable to flag player timeout (see statement SET SCORE TO 0).

Tor considered two remedies for this problem. First, he made another variable called LATEST, into which he attempted to save the latest value of SCORE prior to the end of the game. This could have worked, but failed when he put the copying statement at a point after SCORE had already been zeroed. His next idea was to put high score code into each of the five separate Bopit modules (one each for "slide it," "light it," etc.), in order to make sure the high score was updated before SCORE became, in his word, "contaminated." I take his use of this word as evidence that he perceived a problem in the variable's inconsistent meaning. Before he could proceed with this, I suggested an alternative: suppose you never contaminate the score variable, but instead use other another variable to flag the timeout condition. Tor was very positive about this idea,

[33] Such a measure might become less necessary as Flogo II adds better capabilities for keeping different parts of a program in view—what Green & Petre (1986) call "juxtaposability." The point here, however, is that this a clear case of a programming action undertaken for the sake of real-time program visualization.

saying that it would be a lot simpler, and he implemented it correctly, following through the various related changes that needed to be made throughout the program.

Programming for steadiness

Recall that in Chapter 4 we defined a steady frame as…

> … a way of organizing and representing a system or activity, such that:
>
> (i) relevant variables can be seen and / or manipulated at specific locations within the scene, and
>
> (ii) these variables are defined and presented so as to be constantly present and constantly meaningful.

The mapping of this definition onto live real-time programs is straightforward. By eliminating contamination of the variable SCORE, Tor made it a real-time variable with a single consistent meaning, and thereby enhanced his program's ability to function as steady frame. The new steadiness immediately relaxed the demands on other activities to coordinate with the vicissitudes of SCORE maintenance. High score updating could happen once at the end, and players could check their score after the pressures of game play were past. In other words, the addition of a new steady frame within the program facilitated both program organization and program visualization.

Looking back, we can make the same interpretation of variables we have seen in other projects: RON and LON for line following, HAVE_BALL for the ball-pushing robot, OWNER in the banana takeaway game. All help to establish a steady frame for an important aspect of program execution. By so doing, all simultaneously serve program organization and program visualization. In the RON&LON case especially, we can also see the value of these variables for action-based exploration of dynamic relationships.

How is it that the interests of program organization and visualization are aligned? Why should a steady frame serve both at once? Cybernetically, one can think of the constantly meaningful variables within a steady frame as devices for *temporal decoupling* of processes, whether computational or human. For example, the code that updates the OWNER of the bananas will do so at its own appropriate times. Meanwhile, the animation code that moves the bananas about proceeds on its own schedule, trusting that OWNER is providing usable information whenever it is accessed. It is the steadiness of OWNER that makes this possible. As for program visualization, consider that a programmer is herself, among other things, a process (or, if you prefer, she is engaged in various processes). Without a steady frame, the requirements on the programmer to be looking in the right place at the right time can be burdensome or impossible. A steady frame allows the programmer to consult the display on her own schedule, at moments when the information is most useful to her. This is a highly desirable property in a real-time representation.

Consistency and steadiness in program representations is by no means a new value. Long-established software engineering practices, such as cultivating and documenting invariant assertions, or funneling data operations through a transaction layer, help to maintain steady meanings for variables and other data structures. Live real-time programming raises the stakes by exposing unsteadiness more plainly, and rewarding steadiness more immediately and perceptibly. As they gain experience in building live real-time programs, children have a new kind of opportunity to appreciate steady frames and gradually learn to use or make them. Farther along this developmental trajectory, I believe that the explicit notion of a steady frame (appropriately renamed and repackaged) could become a powerful idea that learners could come to use in support of both programming and learning—vocabulary to enhance what diSessa and colleagues have called children's "meta-representational competence" (diSessa, 2000; diSessa & Sherin, 2000).

The Steady Frame, Learning, and Theoretical Progress

In Chapter 4 I introduced the steady frame as an interface design concept. Now, application of the concept has shifted to children's programming maneuvers. What are we to make of this shift? Are we to infer that programming, and especially live, real-time programming, makes interface designers of all children? And if so, does this weigh for or against the universality of the intellectual substance of programming? The answers are, I believe, "yes" and "for." But to understand this, we need see how essential steady frames are, not just in our tools and notations, but also in the concepts that mediate our activity and understanding in the world. Our conceptual apparatus can be understood as an interface through which we act and perceive. When we look at it this way, conceptual growth and interface design begin to seem very similar indeed.

To ground this analysis, specific examples are needed. I offer brief examples of conceptual steady frames in athletics, mathematics learning, and science. This choice of examples also supports a subsequent suggestion I will want to make: that the shift to real-time makes programming more like all of these things than it was before—and, through real-time programming, these three things all become more like each other.

The athletic steady frame

If you are familiar with racquet sports, you know that good players are distinguished by their coverage of the court, that is, their ability to get to wherever their opponent sends the ball. In squash, these skills ascend through many levels. My squash teacher, himself internationally ranked, told me about watching world champion Jahangir Khan, of the great Khan squash dynasty, play in competition, and marveling at his ability to be wherever he needed to be in the court, without even seeming to break into a run. I don't know all that goes into that exalted level of court coverage, but an important early learning target in squash is the simple but powerful idea of "position." This idea has a pervasive influence on everything you do in squash. As soon as you have hit a ball, your immediate goal is to get back to a central position in the court, in preparation for the next shot. Footwork and racquet preparation also affect your readiness. Your opponent may be in the way of getting to the best position, so you aim your shots with the goal of dislodging him. At every moment in the game, there is something you can be doing to improve your position or challenge your opponent's position. In this sense, position is a constantly meaningful construct: "spatial and bodily factors favoring your ability to get to future balls," a steady frame, a stabilizing construct. Contrast this with the novice's experience: chase the ball, hit it, see what happens, chase the ball, hit it. Playing an experienced opponent, the novice is mystified at why he is the only one who keeps running desperately around the court. For the novice, each episode of getting to the next ball is an independent experience with its own beginning, middle, and end. This fragmented construction of events severely limits the novice's ability to improve his court coverage. He can see sees causal connections within an individual chasing episode (e.g. running faster now will get me to this ball sooner), but lacks a framework to see, and therefore to influence, all the other factors that go into the setting up of that chasing episode.

Essential to positional understanding in squash is the spatial coordinate system in which the player perceives himself and the game action. If a squash is a dynamic system, the obvious variables of interest include the positions and trajectories of the two players and the ball. Since these are all in motion, fixed localization of the garden hose variety is not possible; however, localization of these variables may be considerably more or less orderly depending on the coordinate system within which they are viewed. The experienced player monitors play and plans actions with respect to the court as a whole, and especially with respect to the central, commanding position (the “T”) that both players attempt to control. This spatial reference system provides a true steady frame for squash play, decoupling the player’s strategic and tactical thinking from wherever the player may happen to be on the way to or from a ball. By contrast, as the novice moves about the court, his coordinate system moves with him. In this decidedly unsteady frame, the movements of ball and opponent are harder to track and predict, and plans can be formulated only as far ahead as one’s current perspective will last.

A variant of this idea figures in the cinematography of the *Matrix* films. Here the “sport” is not squash but martial arts combat. In a celebrated scene from the first film, the character Trinity is seen leaping up in front of a hapless policeman. At the apex of her jump, the action suddenly freezes. With Trinity suspended in midair, poised to strike, the camera circles around her and her victim. Just as suddenly, action resumes. Trinity disables the policeman with one kick and then, by means of a series of moves of extraordinary speed and precision, overwhelms his several armed colleagues and makes good her escape.

There are at least two ways to interpret the segment in which action is frozen. To the extent that the effect suggests a pseudo-magical ability on Trinity’s part to step outside the flow of time—an interpretation corroborated by other scenes of bullet-dodging and other miracles—the scene is simply a fantasy. But an alternate interpretation connects the fantasy to an important truth. The circling camera suggests Trinity’s ability to see herself in just the kind of decentered coordinate system that would be needed in order to anticipate the interacting trajectories of an array of opponents. Moreover, such a steady frame would give Trinity the ability to compute those interactions and trajectories through a process that is decoupled from her own movements—a decoupling that can feel almost as empowering as stopping time. I am not especially accomplished in squash, but I have experienced the difference that a combination of concentration and positional sense can make in the reachability of balls. The experience is quite amazing. At times, the ball almost seems to hang in midair, waiting for you to get to it.

Before we leave the sports arena, I want to note that, in addition to coordinate systems, athletic technique includes many rather more literal instances of a steady frame. Swimming strokes and golf swings, for example, work better when there is a steady rotational axis through the back and head. Watch soccer players, basketball players, and jugglers practicing their ball-control skills, and note the steadiness (not rigidity!) of body orientation. These postural steady frames may seem more physical than mental, but they are cognitive nonetheless, requiring active maintenance by the body-mind. They are on a smooth continuum with the steadiness of Flogo II variables. I suspect, though I cannot prove it, that Tor and Nick’s pleasure in their early encounters with variables such as RON and LON resonates with prior experiences of physical poise. Less speculative is the assertion that, by immediately simplifying the action challenges of testing and monitoring a running program, RON and LON themselves produce a kind of athletic advance for the programmer.

Mathematics learning

Most of the case for the steady frame in mathematics learning has already been made, in different theoretical terms, by others. A fundamental and seminal example comes from Piaget: the conservation of number (Piaget, 1941/65). (Indeed, all Piagetian conservations represent real-time variables that help to steady the cognitive frame.) Conserving number means possessing a concept—the cardinality of a set—that is detached from the sequential process of counting. For the conserver, even when one is not counting, a set's cardinality has meaning and can be known. Steadiness of number is facilitated by knowledge of the transformations of a set (such as rearrangement) under which number is invariant, and further strengthened by knowledge of the incremental effects of operations such as adding or removing elements. We can still invoke counting to "update" our number, but the steadiness of the number frame means that such counting can be decoupled from other activities that use the number. Thus unencumbered, such activities become new opportunities to explore and develop more advanced arithmetical ideas building on the base concept of cardinality.

Since I have defined a steady frame as a representation that localizes consistent meanings, one can ask: where is the cardinality of a set localized? In a case such as this, one must take location in a virtual sense. The number of cookies on the plate is a memory indexed by, among other things, the phrase "how many cookies on the plate." This functions similarly to physical location in the sense that, until the memory fades, we are free at any time to turn back to it via such indices, much as we can turn back to a location on a page or screen.

Closely related to the idea of conservation is the notion of reification. In recent years a number of writers have noted that mathematical learning often shows a progression in which mathematical ideas first understood as actions or procedures come later to be understood as objects (e.g. Sfard, 1991; Dubinsky, 2001; Tall et al., 2000). Sfard (1991) proposes three steps in concept formation. *Interiorization* is the stage of getting acquainted with a set of processes or actions—counting, for example, or computing functions by evaluating algebraic formulas. *Condensation* is a gradual process of developing increasing abbreviated to carry out or refer to these actions. The learner's understanding is increasingly holistic, but it is still "tightly connected to a certain process." In the final step, *reification*, the learner's perspective makes a "ontological shift" whereby the concept is now seen as an object in its own right (e.g., to continue the previous examples, a number or a function)

> The new entity is soon detached from the process which produced it and begins to draw its meaning from the fact of its being a member of a certain category. At some point, this category rather than any kind of concrete construction becomes the ultimate base for claims on the new object's existence. A person can investigate general properties of such category and various relations between its representatives. He or she can solve problems finding all the instances of a category which fulfill a given condition. Processes can be performed in which the new-born object is an input. New mathematical objects may now be constructed out of the present one.

What advantage does the reified conception confer? Sfard makes some effort to elucidate this. She notes that procedurally conceived ideas "must be processed in a piecemeal, cumbersome manner, which may lead to a great cognitive strain and to a disturbing feeling of only local—thus insufficient—understanding." By contrast, the "static object-like representation … squeezes the operational information into a compact whole and turns the cognitive schema into a more convenient structure." These characterizations suggest qualitative process distinctions of the kind that I have explained in terms of the notion of a steady frame. For example, the reified conception of function provides a steady frame for varying functions in order to investigate, say, the relationship between formula and graph, or function and derivative—much as a hose nozzle's reification of aim aids exploration of the relationship between aim and trajectory. Sfard's paper doesn't quite see this through, and ends up emphasizing the value of objects as a "chunking" device that helps by reducing strain on short-term memory. From the perspective of the current analysis, that is a subsidiary point. The essential function of reification is not so much chunking as temporal decoupling of thinking processes.

In short, there is ample precedent for the analysis of mathematical learning and development in terms of mental representations that can establish steadier frames for thinking. The term “steady frame” merely adds a cybernetic perspective on how these representations help. In doing so, it also highlights the common function of mental representations and external media.

Scientific theories as steady frames

A scientific theory—especially a theory of dynamic phenomena—can be viewed as a kind of steady frame, in the sense that it defines variables of interest and relationships among them. Scientific advances sometimes come in the form of the definition of new constructs – e.g. momentum, energy – that allow us to monitor and interpret dynamic situations in steadier ways. This aspect of scientific discovery was highlighted in an AI project of Herbert Simon and colleagues in the 1980’s (Langley, Bradshaw, & Simon, 1981; Langley et al., 1987). Called “BACON,” it simulated the scientific process by looking for concise expressions of relationships between data variables. Given data sets from episodes in the history of science, BACON’s output included not just relationships between the input variables, but specifications for new intermediate or derived variables that either tended to remain constant throughout experimental variation, or allowed for simplified description of relationships—that is, they helped to provide a steadier frame for the experimental phenomena. BACON’s developers reported that these variables tended to have useful interpretations, and in fact often reproduced constructs (e.g. specific heat) that have since become standard scientific vocabulary. What this helps to show is that the criteria for improvement of a steady frame—meaningful variables with intelligible relationships—are also criteria for improvement of a scientific theory. This is not surprising if we view a scientific theory as an aid to thinking and action, not unlike the mathematical constructs discussed above.

CONCLUSIONS

Programming is Modeling

When Soloway (1986) famously wrote: “Programming = mechanisms + explanations,” he was highlighting the central role of programming “plans,” or reusable patterns, in accomplishing programming tasks. The formulation expresses a deep truth, but only a partial one. What is missing is any sense of a connection between programming and the wider world. The omission is perhaps not surprising given that the tasks that most students of programming worked on at the time, especially at the college level, were essentially data processing tasks. What programming “=” certainly depends heavily on its cultural embedding.

Running alongside the data-processing oriented programming there has been another educational programming culture, often using languages like LOGO, Boxer or (at more advanced levels) LISP—that made a habit of venturing beyond the conventional concerns of computer science, using programming to pursue creative and expressive goals, and to undertake projects in diverse intellectual domains such as geometry, mathematics, linguistics, physics, and music. This is a different kind of programming, and researchers studying it have naturally seen something different. In particular, they have seen programming as a medium for modeling, theory-making, and “situated abstraction”:

> Thus, the reason that a child interested in (say) music might well want to learn programming is that it provides a rich musical medium allowing her to express musical ideas to which she might otherwise never have given voice. Programming, on this view, is not an exceptionally valuable skill in the abstract … but is rather a skill whose

> definition and importance evolves, and becomes revealed, within particular applications and in accordance with the interests of the child. (Eisenberg, 1995)

> Many years of research … have generated a fair number of cases where programming has provided pupils with a means of expressing and exploring domain-specific ideas. … Programming, in these case, “obliges” pupils to formalize intuitive ideas by expressing them with the use of symbols, executing them on the computer and immediately observing their effect… It enables them to group a set of ideas and then either use them as a named object at a higher level of abstraction or reflect on a specific idea within that object. In this way, they can interchangeably use an idea as a tool and reflect on it as an object (to use Douady’s terminology, 1985). … Programming thus allows a merging of intuitive and reflective thinking. (Kynigos, 1995)

> Our central tenet, therefore, is that a computational environment can be a particularly fruitful domain of situated abstraction, a setting where we may see the externalised face of mathematical objects and relationships through the windows of the software and its accompanying linguistic and notational structures. … in an autoexpressive computational medium, the expression of the relational invariants is not a matter of choice: *it is the simplest way to get things done*. In this case the action / linguistic framework of the medium supports the creation of meaning: and it is here that computational media differ substantially from inert environments. (Noss & Hoyles, 1996, p.125)

These statements all anticipate and corroborate what we have seen in students’ Flogo II programming projects. In the emergence and refinement of variables such as OWNER (bananas takeaway game), HAVEBALL (ball pushing project), SCORE (Bopit project) and SPEED (seesaw project), we have witnessed situated abstraction at work, albeit on a small scale. These small-scale examples may be taken as suggestive of more elaborate and intellectually rich kinds of modeling to come with longer experience and further evolution of the learning environment. In addition, these examples show something of the role and quality of modeling in the context of real-time programming. Children have constructed, or eagerly adopted, the “relational invariants” expressed in the definitions of these variables, in part because they do indeed make it simpler to get things done. But in live real-time programming, there is a second payoff: these conceptual advances (situated abstractions, relational invariants, theoretical constructs, formalized ideas) also make aspects of the problem domain easier to see and control. They make the program a better real-time *interface* to the domain. The sense of empowerment accompanying each such step is palpable.

Live real-time programming is live real-time modeling

If programming is modeling, then live, real-time programming is modeling of a special kind, namely live, real-time modeling. A real-time model is a model in direct engagement with the situation it is modeling. A *live* real-time model is, in addition, a real-time model that one can observe in process, and in which one can intervene—at once a mirror and a lever. By engaging with a live real-time model we experience theory as a medium for perception and action. As we *construct* a real-time model, its increasingly live mediation of the domain enhances our appreciation of the step-by-step empowerment that effective abstraction can bring.

All the same, it is important to note that encouraging or rewarding model-making is not the same as requiring or “obliging” it. The young programmer is free to ignore the attractions of a meaningful interface, as Larry did in his use of permission variables. Tor, it seems to me, had an unusually strong affinity for preserving explicit meaning in his programs. He was constantly on the lookout for opportunities to make things less confusing, and he had no qualms about typing long variable names such as TWO_OR_LESS. He also had a gift for verbal play, making up names such as THE_CHOSEN_ONE and SAVE_YOURSELVES. Having vividly described one section of code as an “antibody” to another, he went on to call a flag variable, used to suppress the activity of the first section, a “de-antibody-izer.” Thus, in terms of both skills and proclivities, Tor was an especially strong modeler. By contrast, Larry tended to be confident in his ability to understand his own cryptic formulations, and uninterested in making them clearer, while Nick tended to see unfathomable program behavior as something to be amused by and to tinker with, but not necessarily to reorganize. Nonetheless, both Larry and Nick did find themselves in situations where they appreciated a clearer program. In a learning environment where reorganizing programs for clarity and simplicity was exemplified and discussed, I think both would, at their own pace, partake of the process.

Lehrer and Schauble (2000) sort mathematical and scientific models into four categories, each developmentally more advanced than the ones before:

Physical Microcosms that refer by means of resemblance to the modeled object.

Representational systems such as maps and diagrams, that bear some resemblance to their subjects, but are rendered in highly conventionalized forms.

Syntactic models, such as graphs or algebraic formulas, that can “summarize essential functioning of a system,” with no surface resemblance to the system.

Hypothetical-deductive models, that permit modeling and exploration of unseen or imagined domains, such as gases.

The basis for this developmental progression is the importance of similarity as the initial ground for the modeling relationship. Only gradually, as they begin to gain interest in the functional and predictive abilities of models, do students let go of the requirement of similarity. But real-time interaction provides another way to experience a link between model and subject, just as compelling as resemblance. As we watch our own motion transduced into graphs and variables, for example, and, conversely, as we watch actions in the model translated into physical effects, the modeling relationship becomes highly salient, not through resemblance but through mediation. Whether the availability of real-time models disrupts, accelerates, or simply facilitates the progression proposed by Lehrer and Schauble remains to be seen. But one way or another its impact will be strong. Lehrer and Schauble trace modeling activities back to the first grade. At that age, tools for constructing dynamic models might look quite different from the Flogo II programs we have seen. Nevertheless, one can imagine designing such tools, and perhaps embedding them in an environment like Flogo II, as an open toolset.

Real-time programming bridges our bodies and our ideas

Real-time programming (live real-time programming, specifically) brings our bodies, and the kinds of action knowledge that we have developed through bodily pursuits, in closer contact with theory-making. It does this in two ways: first by extending the range of programming applications into more physical and action-oriented realms. For example, the MIT Media Laboratory recently witnessed a production of children’s dancing called “Roballet,” in which children’s dance was enhanced by media effects programmed by the children. Some of these effects were triggered by sensor-detected actions of the dancers on stage. This sort of body / media hybrid art form will no doubt continue to flower in the coming years, and be a rich domain for real-time programming. One can also imagine similar media enhancements of sports and physical games.

Body-interactive games such as Bopit, as well as the dance games that are currently among the most popular stations in most video arcades,[34] bring the body slightly further into the digital realm, in the sense that these games emphasize the person reacting to the program, rather than the other way around. Video games represent a further step in the same direction. While opinions vary as to how much these games involve the body, there is no question that they are action games.

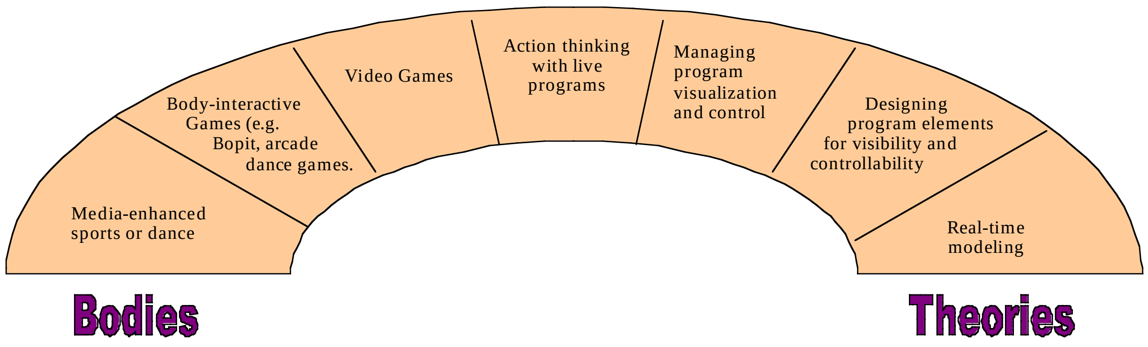

Media-enhanced sports or dance
Body-interactive Games (e.g. Bopit, arcade dance games.
Video Games
Action thinking with live programs
Managing program visualization and control
Designing program elements for visibility and controllability
Real-time modeling
Bodies
Theories

Fig. 2: Real-time applications and programming as a bridge from bodies to theories

The second way that real-time programming connects bodies to theories is through the expanded role of action and interaction in the programming process itself. In Chapter 4 we considered the importance of action thinking in children's investigation of programs—treating the program itself as a kind of video game. And in Chapter 6 we have seen children taking the initiative to shape their programs in order to facilitate perception and action. We have seen how the perception / action advantages of steady frame variables can provide an incentive for situated abstraction in the programming process, to the point where we can begin to see modeling as a powerful strand in real-time programming.

The principle of the steady frame suggests a continuity between intellectual and physical empowerment. In this continuity may be found educational opportunities that this document can only begin to suggest. Children have a consuming interest in physical action and the development of physical and action skills. Instead of asking them to check these interests at the classroom door, we can build increasing connections between action learning and intellectual learning. In the present work, the notion of a steady frame helps to explain one important such connection: that a good theoretical idea, when embodied in a real-time program, is often also a good interface element for real-time interaction.

The step/process duality should be fundamental in computational literacy

The steady frame concept also helps to explain how process-oriented constructs help to make programs more live during program execution. As the conceptual elements of a real-time program advance, we often see a shift from steps toward processes, as

[34] While dance music plays, a screen shows a scrolling notation of dance steps to be performed to the music. Sensors in the platform detect the player's accuracy in performing the dance. By dancing more precisely, the player can earn more points and play longer.

“relational invariants” become expressed in declarative form. At the same time, we have seen that steps are equally essential, for (i) providing part of the control framework needed to deploy processes flexibly (a major lesson of the Flogo I experience), (ii) offering a more concrete means of expressing ideas that the programmer is not ready to express declaratively, and (iii) expressing some things that just make more sense as steps.

Although steps and processes are both quite familiar from a variety of languages, a unified paradigm of the kind found in Flogo II is rare if not unprecedented in computer science, has never been rendered as straightforwardly as in Flogo II, and has never been embodied in a live programming environment. Thus, prior to the current study, beginning programmers have never been asked to distinguish and coordinate these two ways of describing computation. On balance, the considerations presented in this document weigh strongly in favor of the step / process duality as a central Big Idea that can move educational programming forward.

The formal, syntactic distinction between steps and processes was not grasped quickly by children, although older students did show progress in understanding the distinction. Difficulty, however, is not incompatible with Big Idea status. Proportional reasoning, for example, is a perennial sticking point in elementary and middle school mathematics, and yet we do not doubt its importance. If the idea is important enough, we can invest effort in smoothing children’s access to it. And indeed, as a first step in this direction I have suggested a number of interface improvements with the potential to make the difference more understandable, and, equally importantly, to support children in programming effectively even before the difference is fully understood.

For the moment, the argument in favor of a central role for steps and processes rests on these points:

Generativity. Combinations of steps and processes have been shown to express many kinds of real-time behaviors succinctly and understandably, at both beginner and intermediate levels. Using Flogo II, novices’ robotics projects progressed rapidly and appear to compare very favorably with project produced using other languages

Intelligibility. Anecdotally, children’s ability to reason about the functions and interactions of steps and processes seemed strong, despite their difficulty with them as syntactic categories.

Cognates in other programming contexts. The step / process duality is a variant of distinctions seen elsewhere, including pulse vs. data in Flogo I, process coordination issues in MultiLogo (Resnick, 1991), and edge / level confusions encountered by college-level robotics students. That an issue recurs in so many forms suggests that it may be fundamental, not easily avoided, and better faced directly.

Linguistic significance. The distinction between steps and processes parallels the fundamental distinction in linguistics known as aspect. Aspect in verbs, in turn, maps directly onto the equally fundamental mass / count distinction in nouns. That both processes and materials have been categorized in the same way suggests that this distinction between bounded and unbounded forms is highly adaptive, and deeply embedded in our cognition. This is significant in its own right, because Big Ideas gain value through their connections across domains. Moreover, linguistic considerations help us to understand why children’s semantic understanding of Flogo II’s steps and processes might be far ahead of their syntactic understanding.

Capacity for intelligible live embodiment. While processes provide a steady frame for live execution, programming in terms of pure processes quickly becomes arcane. A flexible balance between steps and processes gives programmers the resources to maximize intelligibility of the live program.

Support for situated abstraction. A live language supporting both steps and processes allows room for incremental shifts from step-based to process-based representations, often a hallmark of conceptual advances.

In Sum

This project has produced a connected web of designs, examples, analyses and interpretations. In almost every instance, the claims I have made are provisional, and would benefit from further investigation and refinement. To pick just a few highlights:

- The claim of generativity of steps and processes should be tested through fleshing out of the Flogo II language and its application to a wide range of programming problems, including some much larger projects.
- Proposed features to support learners in building syntactically correct combinations of steps and processes should be built and evaluated.
- The relationship between children's linguistic competence and their ability to reason about steps and processes should be clarified and assessed through empirical research.
- Longitudinal teaching and design experiments in the learning and pedagogy of children's real-time programming are needed to map the conceptual terrain more reliably and in more detail, going well beyond the few landmarks that have been noted here.
- Programming-as-theory-development needs to be further cultivated and investigated in the context of live real-time programming.

In the meantime, tenuous though many elements of the argument may seem in isolation, it is my hope that readers will find, in the mutual support and illumination that holds this web of ideas together, a revealing early account of the nature and value of real-time programming by learners, a coherent basis for optimism about its prospects, a sense of its potential contribution to the larger cause of computational literacy, and a clear agenda for the important and fascinating work that remains to be done.

References

Abelson, H., & Goldenberg, P. (1977). Teacher's guide for computation models of animal behavior. Logo Memo No. 46, MIT AI Laboratory.

Ambler, A.L. (1998). Guest editor's introduction: Selected papers from the 1997 Visual Programming Challenge. *J. Visual Languages and Computing* 9(2), pp.119-126.

Auguston, M., & Delgado, A. (1997). Iterative constructs in the visual data flow language. 1997 IEEE Symposium on Visual Languages (VL '97), Capri, Italy.

Bamberger, J. (1991). *The mind behind the musical ear: How children develop musical intelligence*. Cambridge, MA: Harvard University Press.

Bateson, G. (1980). *Mind and nature: A necessary unity*. Bantam Books.

Baum, D. (2002). NQC Programmer's Guide, v.2.03r1. Available at http://www.enteract.com/~dbaum/nqc.

Begel, A. (1996). LogoBlocks: A graphical programming language for interacting with the world. Advanced Undergraduate Project report. Massachusetts Institute of Technology. (http://www.cs.berkeley.edu/~abegel/mit/begel-aup.pdf)

Blackwell, A. (2002). Cognitive dimensions resource web page. http://www.cl.cam.ac.uk/~afb21/CognitiveDimensions

Breitinger, S., Loogen, R., Ortega-Mallén, Y., & Peña, R. (1997). The Eden Coordination Model for Distributed Memory Systems. Proceedings, *High-leve Parallell Programming Models and Supporting Environments (HIPS)*. IEEE Press.

Brooks, R. A. (1991). Challenges for complete creature architectures. In J.-A. Meyer & S. W. Wilson (Eds.), *From animals to animats: Proceedings of the 1st intl. conference on simulation of adaptive behavior* (pp. 434-443). Cambridge, MA: MIT Press.

Brooks, R. A. (1994). Coherent behavior from many adaptive processes. In D. Cliff, P. Husbands, J.-A. Meyer, & S. W. Wilson (Eds.), *From animals to animats 3: Proceedings of the 3rd international conference on simulation of adaptive behavior* (pp. 22-29). Cambridge, MA: MIT Press.

Bruckman, A. (1998). Community support for constructionist learning. CSCW 7:47-86.

Bruner, J.S. (1973). *Beyond the information given*. New York: Norton.

Burnett, M.M., Atwood, J.W. Jr., & Welch, Z.T. (1998). Implementing level 4 liveness in declarative visual programming languages. *Proceedings* of VL'98, Halifax, Nova Scotia, Canada.

Clark, A. (1997). *Being there: putting brain, body and world together again*. Cambridge, MA: MIT Press.

Clements, D. H., & Sarama, J. (1997). Research on Logo: A decade of progress. *Computers in the Schools, 14*(1-2), 9-46.

Clements, D. (1999). The future of educational computing research: The case of computer programming. , *Information Technology in Childhood Education Annual* . Charlottesville, VA: Association for the Advancement for Computing in Education.

Cole, M., & Wertsch, J. (undated). Beyond the Individual-Social Antinomy in Discussions of Piaget and Vygotsky. Vygotsky Project website: http://www.massey.ac.nz/~alock//virtual/project2.htm

Comrie, B. (1976). *Aspect*. Cambridge University Press.

Cooper, S., Dann, W., & Pausch, R. (200). ALICE: A 3-D tool for introductory programming concepts. Proceedings, 5th Annual Conference on Innovation and Technology in Computer Science Education, Canterbury, England.

Cox, P.T., Risley, C.C., & Smedley, T.J. (1998). Toward concrete representation in visual languages for robot control. *J. Visual Languages and Computing, 9*, pp. 211-239.

Cox, P.T., & Smedley, T.J. (2000). Building environments for visual programming of robots by demonstration. *J. Visual Languages and Computing, 11*, pp. 549-571.

Cromwell, J. (1998). The dawning of the age of multithreading. Dr. Dobbs Journal, Sept. 1998.

Davis, A.L. and Keller, R.M. (1982). Data flow program graphs. *IEEE Computer,* 15:26--41.

diSessa, A.A. (1986). Models of computation. In D. A. Norman & S. W. Draper (Eds.) *User-centered system design: New perspectives on human-computer interaction*. Hillsdale, NJ: Erlbaum.

diSessa, A. A. (1997a). Open toolsets: New ends and new means in learning mathematics and science with computers. Paper presented at the 21st Conference of the International Group for the Psychology of Mathematics Education, Lahti, Finland.

diSessa, A. A. (1997b). Twenty reasons why you should use Boxer (instead of Logo). Paper presented at the Sixth European Logo Conference, Budapest, Hungary.

diSessa, A. A. (2000). *Changing minds: Computers, learning and literacy*. Cambridge, MA: MIT Press.

DiSessa, A.A., & Sherin, B. (2000). Meta-representation: An introduction. *Journal of Mathematical Behavior 19*(4), pp. 385-398.

Douady, R. (1985). The interplay between the different settings, tool-object dialectic in the extension of mathematical ability. *Proceedings of the Ninth International Conference for the Psychology of Mathematics Education*, *2*, 33-52, Montreal.

Druin, A., & Hendler, J., Eds. (2000). *Robots for kids: Exploring new technologies for learning*. San Francisco, CA: Academic Press.

Dubinsky, E., & McDonald, M.A. (2001) APOS: A Constructivist Theory of Learning in Undergraduate Mathematics Education Research. In D. Holton et al. (Eds.) *The teaching and learning of mathematics at university level: An ICMI study*. Kluwer.

DuBoulay, B. (1989). Some difficulties of learning to program. In E. Soloway & J. C. Spohrer (Eds.), *Studying the novice programmer* (pp. 283-299). Hillsdale, NJ: Erlbaum.

Eisenberg, M. (1995). Creating software applications for children: Some thoughts about design. In diSessa, A., Hoyles, C., and Noss, R. (Eds.) *Computers and Exploratory Learning*. NATO ASI Series F. Springer. Berlin.

Eisner, E. W. (1991). *The enlightened eye: Qualitative inquiry and the enhancement of educational practice*. New York: Macmillan.

Erwig, M., & Meyer, B. (1995). Heterogeneous visual languages: Integrating visual and textual programming. *11th IEEE Symposium on Visual Languages*. Darmstadt, 1995, pp. 318–325.

Ghittori, E., Mosconi, M., & Porta, M. (1998) Designing and Testing new Programming Constructs in a Data Flow VL. Technical Report, DIS University of Pavia, Italy. URL: http://iride.unipv.it/research/papers/98tr-dataflow.html.

Gibson, J.J. (1966). *The senses considered as perceptual systems*. London: George Allen & Unwin.

Gindling, J., Ioannidou, A., Loh, J., Lokkebo, O., Repenning, A. (1995). LegoSheets: A rule-based programming, simulation, and manipulation environment for the programmable brick. Proceedings, Visual Languages 1995, Darmstadt, Germany, pp. 172-179. IEEE Press.

Good, J. (1996) The 'right' tool for the task: An investigation of external representations, program abstractions and task requirements. In W. D. Gray & D. A. Boehm-Davis (Eds.) *Empirical studies of programmers: Sixth workshop.* Norwood, NJ: Ablex.

Gough, J. (1997). How far is far enough? What is Logo really for? *Computers in the Schools, 14,* pp. 171-182.

Granott, N. (1993). *Microdevelopment of coconstruction of knowledge during problem solving: Puzzled minds, weird creatures, and wuggles.* Unpublished doctoral dissertation, MIT Media Laboratory, Cambridge, MA.

Green, T. R. G., & Petre, M. (1992). When visual programs are harder to read than textual programs. Proc. Sixth European Conference on Cognitive Ergonomics (ECCE 6), pp. 167-180.

Green, T. R. G., & Petre, M. (1996). Usability analysis of visual programming environments: A 'cognitive dimensions' framework. *J. Visual Languages and Computing, 7*, 131-174.

Hancock, C. (1995). The medium and the curriculum: Reflections on transparent tools and tacit mathematics. In diSessa, A., Hoyles, C., and Noss, R. (Eds.) *Computers and Exploratory Learning*. NATO ASI Series F. Springer. Berlin.

Hancock, C. (2001a). Children's understanding of process in the construction of robot behaviors. Paper presented at the symposium "Varieties of Programming Experience," AERA, Seattle, 2001.

Hancock, C. (2001b). Computer criticism and the epistemology of the possible. Unpublished manuscript. (MIT Media Lab general exam qualifying paper).

Hancock, C. (2002a). Toward a unified paradigm for constructing and understanding robot processes. Paper presented at the IEEE Symposium on Human-Centric Computing (HCC-02), Arlington, VA.

Hancock, C. (2002b). The paradigm case and the vexing episode: particular and general in the epistemology of design. Presented at ICLS 2002, Seattle, WA.

Harel, I. (1991). *Children designers: Interdisciplinary construction for learning and knowing mathematics in a computer-rich school.* Norwood, NJ: Ablex.

Harvey, B. (1991). Symbolic programming vs. the AP curriculum. *The Computing Teacher*. (Available online at the author's home page and at the Logo Foundation website).

Heger, N., Cypher, A., & Smith, D.C. (1998). Cocoa at the visual programming challenge 1997. *J. Visual Languages and Computing, 9,* pp. 151-169.

Hewitt, C. (1979). Viewing control structures as patterns of passing messages. In P. Winston & R. Brown (Eds.). *Artificial intelligance: An MIT perspective*. Reading, MA: Addison-Wesley.

Hils, D.D. (1992). Visual languages and computing survey: Data flow visual programming languages. *J. Visual Languages and Computing 3*, pp. 69-101.

Hogg, D.W., Martin, F., & Resnick, M. (1991). Braitenberg creatures. E&L Memo No. 13. Epistemology and Learning Group, MIT Media Laboratory.

Hoyles, C. (1993). Microworlds/Schoolworlds: The transformation of an innovation. In W. Dörfler, C. Keitel, & K. Ruthven (Eds.), *Learning from computers: Mathematics education and technology*. (pp. 1-17). Berlin: Springer-Verlag.

Hoyles, C., & Noss, R. (1999). Playing with (and without) words. Playground Project working paper. [http://www.ioe.ac.uk/playground/research/papers.htm]

Hundhausen, C., & Douglas, S. (2000). SALSA and ALVIS: A language and system for constructing and presenting low-fidelity algorithm animations. *Proceedings*, 2000 IEEE Symposium on Visual Languages, Seattle, WA. IEEE.

Ibrahim, B. (1998). Diagrammatic representation of data types and data manipulations in a combined data- and control-flow language. 1998 IEEE Symposium on Visual Languages, Nova Scotia, Canada.

Jackendoff, R.S. (1994). *Patterns in the mind.* New York: Basic Books.

Jackson, S., Krajcik, J., & Soloway, E. (1999). Model-It: A design retrospective. In M. Jacobson & R. Kozma (Eds.), *Advanced designs for the technologies of learning: Innovations in science and mathematics education*. Hillsdale, NJ: Erlbaum.

Janda, L. (2000). Cognitive linguistics. http://www.indiana.edu/~slavconf/SLING2K/pospapers/janda.pdf.

Johnson-Laird, P.N. (1983). *Mental Models.* Cambridge, MA: Harvard University Press.

Kafai, Y.B. (1995). *Minds in play: Computer game design as a context for children's learning.* Hillsdale, NJ: Erlbaum.

Kahn, K. (1996). Toon Talk': An animated programming environment for children. *Journal of Visual Languages and Computing, 7*(2):197-217.

Kahn, K. (1999a). From Prolog and Zelda to ToonTalk. In D. De Schreye (Ed.), *Proceedings of the International Conference on Logic Programming 1999*. MIT Press.

Kahn, K. (1999b). Helping children learn hard things: Computer programming with familiar objects and actions. In A. Druin (Ed.), *The design of children's technology*. San Francisco: Morgan Kaufman.

Kahn, K. (2000). "Educational Technology." Interview transcript included in ToonTalk documentation files. (…/ToonTalk/Doc/English/edtech.htm)

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.* San Francisco: Morgan Kaufman.

Kahn, K. (2001b). ToonTalk and Logo: Is ToonTalk a colleague, competitor, successor, sibling, or child of Logo? Presented at EuroLogo 2001. (To appear in LogoUpdate).

Kay, A. (1991). Computers, networks, and education. *Scientific American, 265*(3) (September), pp. 100-107 or 138-148.

Kodosky, J., McCrisken, J., & Rymar, G. (1991). Visual programming using structured dataflow. Proceedings, IEEE Workshop on Visual Languages, Kobe, Japan.

Kolodner, J.L, Camp, P.J., Crismond, D., Fasse, B., Gray, J., Holbrook, J., & Ryan, M. (2001). Learning by design: Promoting deep science learning through a design approach. Paper presented at "Design: Connect, Create, Collaborate," Univ. of Georgia, Athens, GA.

Kuhn, T. (1970. *The structure of scientific revolutions* (2nd Ed.). Univ. of Chicago Press.

Kynigos, C. (1995). Programming as a means of expressing ideas: Three case studies situated in a directive educational system. In diSessa, A., Hoyles, C., and Noss, R. (Eds.) *Computers and Exploratory Learning.* NATO ASI Series F. Springer. Berlin.

Koutlis, M., Birbilis, G., Kynigos, C., & Tsironis, G. (1997). E-Slate: a kit of educational components. Poster session, AI-ED '99, LeMons, France. [http://e–slate.cti.gr/resources/aied99.pdf]

Lakoff, G., & Johnson, M. (1999). *Philosophy in the flesh: The embodied mind and its challenge to western thought,* New York: Basic Books, 1999.

Langley, P., Bradshaw, G.L., Simon, H.A. (1981). BACON.5: The discovery of conservation laws. *Proceedings,* 7th Intl. Joint Conference on Artificial Intelligence (IJCAI '81), pp. 121-126. Vancouver, Canada. William Kaufman.

Langley, P., Simon, H.A., & Zytkow, J.M (1987). *Scientific discovery: Computational explorations of the creative process*. Cambridge, MA: MIT Press.

Lehrer, R., & Schauble, L. (2000). Modeling in mathematics and science. In R. Glaser (Ed.) *Advances in instructional psychology, vol 5: Educational design and cognitive science*. Mahwah, NJ: Erlbaum.

Lévi-Strauss, C. (1966). *The savage mind.* University of Chicago Press.

Lieberman, H. (1987). Concurrent object-oriented programming in Act 1. In A. Yonezawa & M. Tokoro (Eds.), *Object-oriented concurrent programming.* Cambridge, MA: MIT Press.

Martin, F. (1996). Kids learning engineering science using LEGO and the Programmable Brick. Presented at AERA, New York, NY.

Martin, F. (1994). *Circuits to control: Learning engineering by designing LEGO robots*. Unpublished doctoral dissertation. MIT Media Laboratory, Cambridge, MA.

Martin, F., Mikhak, B., Resnick, M., Silverman, B., & Berg, R. (2000). To Mindstorms and beyond: Evolution of a construction kit for magical machines. In A. Druin (Ed.) *Robots for kids: Exploring new technologies for learning experiences*. San Francisco: Morgan Kaufman / Academic Press.

Mikhak, B., Lyon, C., & Gorton, T. (2002). The Tower system: A toolkit for prototyping tangible user interfaces. Submitted as a long paper to CHI 2003. See also http://gig.media.mit.edu/projects/tower/.

Miller, D.P., & Stein, C. (2000). "So that's what pi is for!" and other educational epiphanies from hands-on robotics. In A. Druin, & J. Hendler (Eds.), *Robots for kids: Exploring new technologies for learning*. San Francisco, CA: Academic Press.

Minsky, M. (1986). *The society of mind.* New York: Simon&Schuster.

Morgado, L. (2001). Computer Programming in Kindergarten Education Research Using ToonTalk. Presentation at "Children's Programming Oddysey" Symposium, HCC'01, Stresa, Italy. Abstract at http://www2.informatik.uni-erlangen.de/HCC01/proposals/Morgado-paper.pdf

Neisser, U. (1976). *Cognition and reality*. San Francisco: W. H. Freeman.

Noss, R., & Hoyles, C. (1996). *Windows on mathematical meanings: Learning cultures and computers*. Dordrecht, the Netherlands: Kluwer Academic Publishers.

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

Papert, S. (1991). Situating Constructionism. In I. Harel & S. Papert (Eds.), *Constructionism*. Norwood, NJ: Ablex. Pp. 1 – 11.

Papert, S. (1993). The children's machine: Rethinking school in the age of the computer. New York: Basic Books.

Papert, S. (2000). What's the big idea? Toward a pedagogy of idea power. *IBM Systems Journal*, *39*(3&4), 720-729.

Papert, S. (2002). The turtle's long slow trip: Macro-educological perspectives on microworlds. *J. Educational Computing Research*, *27*(1&2), pp. 7-27.

Pea, R., Soloway, E., & Spohrer, J. (1987). The buggy path to the development of programming expertise. *Focus on Learning Problems in Mathematics*, *9*(1).

Perkins, D.N., & Martin, F. (1986). Fragile knowledge and neglected strategies in novice programmers. In E. Soloway & S. Iyengar (Eds.), *Empirical studies of programmers*. Norwood, NJ: Ablex.

Piaget, J., & Szeminska, A. (1941/1965). *The child's conception of number*. London: Routledge and Kegan Paul.

Portsmore, M. (1999). Robolab: Intuitive robotic programming software to support lifelong learning. *Apple Learning Technology Review*, Spring-Summer 1999.

Price, B., Baecker, R., & Small, I. (1998). An introduction to software visualization. In J. Stasko, J. Domingue, M.H. Brown, & B.A. Price (Eds.) *Software Visualization*. Cambridge, MA: MIT Press.

Repenning, A. (2000). AgentSheets(r): An interactive simulation environment with end-user programmable agents. Paper presented at the Interaction 2000, Tokyo, Japan.

Repenning, A., & Ambach, J. (1996). Tactile programming: A unified manipulation paradigm supporting program comprehension, composition, and sharing. Paper presented at the 1996 IEEE Symposium on Visual Languages, Boulder, CA.

Resnick, M. (1990). MultiLogo: A study of children and concurrent programming. *Interactive Learning Environments*, *1*(*3*), 153-170.

Resnick, M. (1993). Behavior construction kits. *Comm. ACM 36*:7, pp. 64-71.

Resnick, M. (1994) *Turtles, termites, and traffic jams: Explorations in massively parallel microworlds*. Cambridge, MA: MIT Press.

Resnick, M., Berg, R., & Eisenberg, M. (2000). Beyond black boxes: Bringing transparency and aesthetics back to scientific investigation. *Journal of the Learning Sciences*, 9(1) pp.7-30.

Resnick, M., Bruckman, A., & Martin, F. (1996). Pianos not stereos: Creating computational construction kits. *Interactions*, *3*(6).

Resnick, M., and Ocko, S. (1991). LEGO/Logo: Learning through and about design. In I. Harel & S. Papert (Eds.) *Constructionism*. Norwood, NJ: Able.

Roschelle, J., Kaput, J., Stroup, W. and Kahn, T.M. (1998). Scaleable integration of educational software: Exploring the promise of component architectures. Journal of Interactive Media in Education, 98 (6). [http://www-jime.open.ac.uk/98/6].

Schön, D.A. (1983). *The reflective practitioner: How professionals think in action*. New York: Basic Books.

SGS-Thompson Microelectronics, Ltd. (1990). *OCCAM 2.1 Reference Manual*.

Sherin, B. (2001). A comparison of programming languages and algebraic notation as expressive languages for physics. *International Journal of Computers for Mathematical Learning 6*: 1–61.

Tall, D., Thomas, M., Davis, G., Gray, E., & Simpson, A. (2000). What is the object of the encapsulation of a process? *Journal of Mathematical Behavior*, *18*(2), pp. 223-241.

Tanimoto, S. (1990). VIVA: A Visual Language for Image Processing. *Journal of Visual Languages and Computing*, 1(2), 127-139.

Travers, M. (1996). *Programming with agents: New metaphors for thinking about computation*. Ph.D. Dissertation, MIT.

Turbak, F., & Berg, R. (2002). Robotic design studio: Exploring the big ideas of engineering in a liberal arts environment. *Journal of Science Education and Technology*, *11*(3), 237-253.

Turkle, S. (1984). The second self: Computers and the human spirit. New York: Simon&Schuster.

Turkle, S. (1995). *Life on the Screen: Identity in the Age of the Internet*. New York: Simon & Schuster.

Turkle, S., & Papert, S. (1992). Epistemological Pluralism and the Revaluation of the Concrete. *Journal of Mathematical Behavior*, Vol. 11, No.1, pp. 3-33.

Webb, B. (2002). Robots In Invertebrate Neuroscience. *Nature, 417* (May), pp. 359-363.

Wenger, E. (1990). *Toward a theory of cultural transparency*. Unpublished doctoral dissertation. Univ. of California, Irvine.

Whitley, K.N. (2000). Empirical research of visual programming languages: An experiment testing the comprehensibility of LabView. Unpublished doctoral dissertation. Computer Science dept., Vanderbilt University.

Wilensky, U. (1991). Abstract meditations on the concrete and concrete implications for mathematics education. In I. Harel & S. Papert (Eds.), *Constructionism* (pp 193-204). Norwood, NJ: Ablex.

Wilensky, U. & Reisman, K. (in press). Thinking like a wolf, a sheep or a firefly: Learning biology through constructing and testing computational theories. Cognition & Instruction.

Wilensky, U. & Resnick, M. (1999). Thinking in levels: A dynamic systems perspective to making sense of the world. Journal of Science Education and Technology. Vol. 8 No. 1. pp. 3 - 18. [http://www.ccl.sesp.northwestern.edu/cm/papers/levels/levels.html]

Wilensky, U., Stroup, W. (2000) Networked gridlock: Students enacting complex dynamic phenomena with the HubNet architecture. *Proceedings*, Fourth Annual International Conference of the Learning Sciences, Ann Arbor, MI, June 14 - 17, 2000. [http://www.ccl.sesp.northwestern.edu/cm/papers/gridlock/Wilensky-Stroup.html]