Little Languages for Little Robots

Matthew C. Jadud
University of Kent Canterbury
Canterbury, UK
mcj4@kent.ac.uk

Brooke N. Chenoweth
Indiana University
Bloomington
Bloomington, IN
bchenowe@cs.indiana.edu

Jacob Schleter
Gibson High School
Fort Branch, IN

ABSTRACT

With serendipity as our muse, we have created tools that allow students to author languages of their own design for robots of their own construction. In developing a Scheme compiler for the LEGO Mindstorm we realized that there is great educational potential in the design and creation of new languages for small robotics kits. As a side effect of bringing Scheme and the Mindstorm together in a creative context, we have begun an exploration of teaching language design that is fundamentally different from the treatment of the subject in traditional literature.

1. INTRODUCTION

Jacob Schleter, a rising senior at Gibson High School, Fort Branch, Indiana, took part in the Indiana University College of Arts and Sciences Summer Research Experience for six weeks during the summer of 2002. He was interested in programming and robotics, and ended up joining the continuing effort in working with Scorth, our Scheme compiler for the LEGO Mindstorm. Jacob's efforts resulted in the creation of Jackll, a new programming language for the Mindstorm intended to be suitable for beginner programmers.[1]

This summer experience is particularly interesting to us, as we applied Scheme in an educational context where it was used to *create* languages as opposed to analyzing them. Typical applications of Scheme build up to using it as a tool for exploring the interpretation and compilation of programming languages.[1] In moving from an analytical to a synthetic task—from language interpretation to language creation—we have opened a whole new realm of possibilities. We believe there is great value in exploring the application of our Scheme compiler to the LEGO Mindstorm in *constructionist* contexts—contexts where students learn by building personally meaningful artifacts in the world.

[1]JACKLL: **Jac**ob's **K**iller **L**ittle **L**anguage

2. JACKLL: A NEW LANGUAGE

In building up to the evolution of Jackll and the philosophies its creation embodies, we feel it is appropriate to first introduce the LEGO Mindstorm, the target for our Scheme compiler, and situate Jackll with respect to other languages intended for beginner programmers.

2.1 What is the LEGO Mindstorm?

The LEGO Mindstorm Robotics Invention System is a commercial product from the LEGO Group that provides an inexpensive, reconfigurable platform for exploring robotics. It comes standard with two motors, two touch sensors, one light sensor, and hundreds of pieces for assembling all sorts of creations. The Mindstorm is currently programmable in a variety of languages, although in most cases these are incomplete languages limited in power and expressivity. A notable exception to this rule is pbForth, a complete ANSI Forth for the LEGO Mindstorm, which we used as the target for our Scheme implementation.

2.2 What is Scorth?

The continuing evolution of *Introduction to LEGO Robotics*, a non-majors course in the Computer Science Department at Indiana University, led us to develop Scorth, a Scheme compiler for the LEGO Mindstorm.[2] Scorth's creation was intended to support the use of *How to Design Programs* in the classroom; instead, our first serious application of Scorth was to the task of creating Jackll.

Brooke Chenoweth is responsible for large parts of Scorth, which handles a subset of the Scheme programming langauge. While Scorth does not (currently) compile directly to hardware, it does cross-compile to Forth, and produces code executable by pbForth, an excellent Forth implementation for the Mindstorm by Ralph Hempel.[9] Currently, we are extending the compiler to provide support for `call/cc` (continuations) and garbage collection; when complete, Scorth will provide a reasonably complete Scheme for the LEGO Mindstorm.

[2]Scorth: Scheme→Forth

2.3 A Language Suitable for Beginners?

Being relatively new to programming, Jacob wanted Jackll to be easy for beginners to use. To this end, we decided that every Jackll program should have a visual representation as well as a textual representation. The visual representation allows programmers to sketch pictures of the program when discussing it with others (or perhaps someday build a visual programming tool), while the textual representation provides a concise way of writing programs in Jackll for the Mindstorm. Our visual representation ended up being based largely on state machines.

Our discussions regarding beginner languages were in keeping with the literature, despite "what makes a language suitable for beginners?" being a live and open question for more than 25 years. Lecarme discussed the considerations employed in choosing to use Pascal in a first course in 1973; reasons for the use of Pascal in the classroom included the facts that:

- A good, fast compiler (written in Pascal) was available, both short and maintainable on-site,
- The language is clean and consistent,
- Pascal naturally avoided the use of `goto`s, and
- The language "does not assume its users to be stupid."

Reasons against choosing Pascal included:

- It is not broadly used,
- "It is not (yet) broadly known,"
- It does not provide many "powerful tools" in other languages, and
- It does not allow for dynamic allocation of memory.[15]

These considerations can be categorized (roughly) as involving **tools**, **consistency**, **acceptance**, **expressivity**, and **power**. Kölling, et al. expressed many of the same sentiments 23 years later in building up a series of arguments for the BlueJ programming environment, now a widely used tool for introducing Java in a relatively consistent, objects-first approach.[12] After discussing these sorts of issues, we opted to keep Jackll simple, sacrificing expressive power at times in favor of consistency of syntax and execution.

Visual representations of code show up in the literature as well; Cunniff, Taylor, and Black developed a flowchart-like representation of Pascal running on terminals connected to a VAX 780. They provided an environment that allowed students to create compilable flowcharts, making explicit the control-flow structure of students' code and effectively eliminating syntax errors.[4] ROBOLAB, a programming environment available for the Mindstorm, similarly provides an iconographic, flowchart interface for programming.[5] BlueJ takes visual representation of code a step further, and provides students with *multiple representations* of their code. They provide a visual formalism (UML) at the object level, and also the full source fo their program; both remain in sync regardless of which is edited.[13] While we were not able to write a visual editor for our language in the short few weeks available, we were able to compile our code to *dot*, and produce graphical representations of our code from the same source that produced an executable.[3] This allowed us to go from sketch to code to image, comparing the initial drawings with the automatically generated diagrams for consistency.

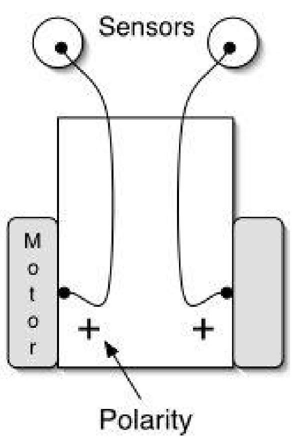


Figure 1: A simple Braitenberg vehicle.

2.4 The Evolution of Jackll

Jackll's visual notation is based on the common directed graph notation of state machines; we use circles for states, and arrows to indicate the transitions between states. Jackll's design began with a discussion of string-accepting finite-state automata; we then expanded on the notion of what an FSA could do by exploring behaviors common to programs for the LEGO Mindstorm: controlling motors, reading sensor values, etc.

To provide "test cases" for our new language, we explored the work of Valentino Braitenberg, and used his Vehicles as the motivation for our first programs.[3] We believed that if we could draw simple state machines (machines with a small number of nodes) that expressed the behavior of Braitenberg's first few vehicles, then we would accept that as a step towards having a language suitable for the LEGO Mindstorm. Our goal was to keep the language simple, and hence we began with what we believed to be a small, core functionality set that could grow as necessary.

Braitenberg Vehicles are characterized by a simple sensor and motor configuration (Figure 1). Each sensor is wired either with a positive influence on a given motor (an increase of stuff sensed increases the speed of the motor), or a negative influence (less stuff means less speed). Each Braitenberg Vehicle has an anthropomorphic name based

[3]Part of the AT&T Research Labs Graphviz library; http://www.research.att.com/sw/tools/graphviz/

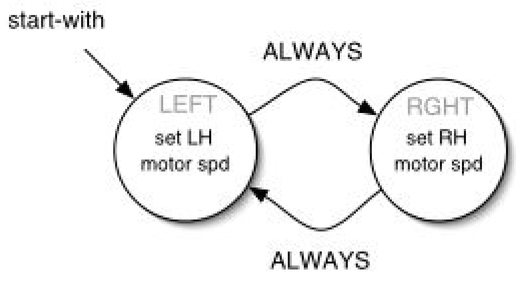


Figure 2: The Coward in Jackll's visual syntax.

```
(start-with LEFT)

(state LEFT
  (do (set-LH-motor-speed)
    (then-look-for
      [ALWAYS RGHT])))

(state RGHT
  (do (set-RH-motor-speed)
    (then-look-for
      [ALWAYS LEFT])))
```

Figure 3: The Coward in Jackll's textual syntax.

on the emotion it exemplifies; we thought the Coward was an interesting Vehicle, and used it as the basis for many of our test cases. Cowards are afraid of stuff, which in our case is light. Our Cowards had two light sensors, each wired to its respective motor with a positive influence; if more light is detected on one side, the Coward should accelerate on that side, thus steering away from the light source.

Jacob's first state machines for the Coward were, in hindsight, huge. With more than twenty states in the machine, it was difficult to tell what the program would actually *do* when executed. We agreed that this first program was not simple, but liked what we had so far: the nodes contained things the robot should do (like changing motor speed), and the arrows had conditions for moving between nodes (by checking light levels reported by the sensors). We discussed ways that we could improve this in general terms; eventually we solved the problem in two nodes, which seemed appropriate for the complexity of the Coward (Figure 2).

This was an exciting process; we worked as a language design team, and it was fun to try and find ways to simplify the language. Some truly excellent discussion came from attempting to develop a textual representation of the diagrams. Jacob took the lead and made a first attempt at representing the state diagrams in a 'Schemely' way (meaning an s-expression based prefix notation), because none of us really knew how the final language would look. Jacob's first syntax deviated somewhat from the notion of a prefix language, and depended heavily on naming; so much so, that we discovered it broke horribly with simple code omissions on the part of the programmer.

Despite the different backgrounds and levels of ability in our group, we had some excellent debates on the naming, structure, and the dynamics of Jackll. The collaborative evolution that led to a final textual syntax for Jackll (Figure 3) took place with the understanding that no idea could be challenged without justification, and no feature kept without defense. This dialogue was unlike most anything any of us had encountered in educational contexts before—the "student" had transitioned from being a *learner* to head language designer and *peer*.

3. DUELING PHILOSOPHIES

Jackll provided an example of the kinds of interactions that are possible when students have high expectations placed upon them with the support and freedom to meet and exceed those expectations. Scheme was particularly appropriate for the task of creating languages for little robots, but the beliefs and attitudes typically brought into the introductory classroom when teaching with Scheme are not. This is not immediately obvious; the following sections expose some of the more subtle tensions at work in combining Scheme and the LEGO Mindstorm.

3.1 Where we are coming from

We feel there is much to be learned from the constructionistic pedagogies and theories employed in the design and construction of the Mindstorm, a *toy to think with*.[19] Papert, Resnick, Martin, *et. al.* have long made the point that manipulatives like the LEGO do not force a particular style of learning on the student, but instead provide the student with the opportunity to play and explore dynamic systems in ways *natural to them as individuals*. We believe that constructionism—the theory that we construct new knowledge best by building personally significant artifacts—exemplifies the kinds of learning that took place in the design and creation of Jackll.[16]

The creation of Scorth grew out of a perceived need in *Introduction to LEGO Robotics* (*ItLR*) to tie Scheme more tightly to the LEGO Mindstorm. Matthew had been developing this course over the previous two years, in conjunction with the *TeamStorms* theory of instruction.[10].[4] TeamStorms provided a theoretical framework for developing introductory computing instruction using manipulatives like the LEGO Mindstorm. In keeping with the methodologies promoted by this theory of instruction, instructional values critical to the course were shared and discussed with the students in *ItLR*:

[4]A theory of instruction provides a philosophical grounding for instructors, indicating the values that must be implicit and explicit in a given classroom setting, as well as defining methods for dealing with instructional situations likely to arise in the classroom.[18]

- Active responsibility for learning on the part of participants
- Focus on process as well as product
- Exploration and discovery in learning
- Peer–peer interactions in learning
- Authentic, or real-world, learning situations
- Fun

Reflective discussion at the end of the spring 2002 term indicated that students felt a conflict between the notion of having fun in class and getting real work done. This was evident in attempting to integrate *How to Design Programs* into *Introduction to LEGO Robotics.* Over the course of several semesters, the dialogue invariably went like this:

Matthew: We'll be discussing design principles and problem solving while working with LEGO and Scheme this semester!

Student: Can we program the LEGO in Scheme?

Matthew: Erm. . . no.

Student: Oh. Why bother then?

An instructor's first instinct may be to dismiss this kind of response, but students are often more perceptive than we give them credit for. Sometimes, when moaning about how awful lecture is, students really mean *the lecture is awful.* Because of our experience in developing Jackll, we came to realize that the students in *ItLR* were not complaining about Scheme *per se*, but instead they were complaining about the philosophical conflict inherent in the approach used by *How to Design Programs* and the ways they worked most naturally with the LEGO.

3.2 LEGO and Scheme in the Wild

3.2.1 The Mindstorm in the Wild

In extreme contrast to our work we might situate the efforts of Barry Fagin *et. al.* at the Air Force Academy.[6, 7] Fagin's work is direct, almost drill-and-practice in its application of the Mindstorm to the computer science classroom. Attempting to apply TeamStorms in this setting might be exciting or (more likely) disastrous for instructor and student alike; the students at the Air Force Academy "do not suffer fools."[5] There is no judgment implied here—Fagin is addressing the fact that students at the Air Force Academy have *so much* to do, in so little time (and on so little sleep) that having an hour where they are asked to "just be creative" might represent a serious mismatch of student expectations and instructional values.

Much of the computer science education literature describes applications of the Mindstorm to learning contexts that fall somewhere between Fagin's highly constrained usage and our own. Typically, the LEGO is employed in an attempt to teach students a particular language or content area. Barnes discusses the suitablility of using the LEGO in an introductory Java curriculum; his work is less a discussion of the suitability of LEGO robotics in CS1, and more a discussion of the expressivity of the Java-esque languages available for the Mindstorm.[2] Kumar and Meeden (at Bryn Mawr and Swarthmore, respectively) as well as Klassner (Villanova) have all examined bringing robots into the Artificial Intelligence curriculum.[14, 11] Our work will likely be of interest to Klassner, as his efforts would be aided by the existence of a full LISP for the Mindstorm. The spirit of our work is probably most like that of Turbak and Berg at Wellesley College, where the LEGO Mindstorm is being applied in a liberal arts context in an attempt to broaden introductory students' horizons regarding the fundamental nature of engineering.[20]

[5]Barry Fagin, personal conversation, 2000

3.2.2 Scheme in the Wild

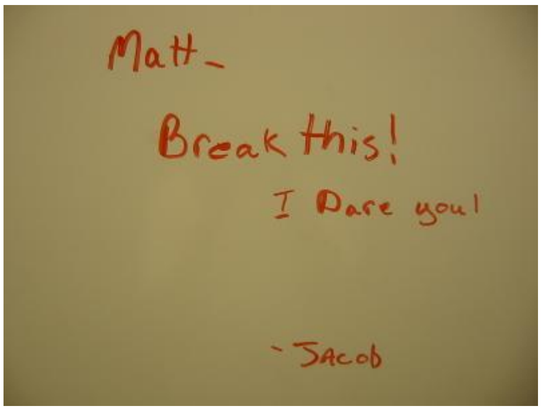
Scorth grew from the desire to bring Scheme to the Mindstorm, easing the integration of the new text *How to Design Programs* into the *Introduction to LEGO Robotics* classroom.[8] The most striking feature of this text is its pedagogic focus on the use of *design recipes.* These 'design recipes' define both how students should structure their code and (more importantly for our discussion) *how students should structure their thinking about programming.* Quoting from their preface,

> . . . We have . . . developed design guidelines that lead students from a problem statement to a computational solution in step-by-step fashion with well-defined intermediate products.

In contrast, Papert and Turkle found that creative students at MIT and Harvard who could not conform to similar rigid problem solving methodologies in programming classrooms typically became disenfranchised and discouraged from the educational process.[21] Students in *ItLR* expressed their displeasure with using Scheme in contrast to their writing about, building with, and programming the LEGO Mindstorm. We feel this is an example of the same kind of discouragement that Papert and Turkle discussed more than a decade ago—which is fitting, as the *How to Design Programs* design recipe is largely a restatement of pedagogical assertions that have been around for over twenty years.[17] While it does provide structure for students and instructors to lean on, the *How to Design Programs* approach does not address the motivational, creative, humanistic, and affective aspects of the classroom setting.

4. FUTURE WORK

Why does the creation of little languages for little robots interest us? Consider a small piece of the whiteboard we captured on film from this summer (Figure 4). We had been working hard at developing a textual syntax for Jackll, and had recently discovered a large problem with one of Jacob's early attempts. After a brief respite from


Matt –

Break this!

I Dare you!

~Jacob

Figure 4: The last word on Jackll.

debate and a few days to think, Jacob devised a new syntax based (roughly) on earlier attempts, taking into consideration recent discussion. This note on the whiteboard accompanied his final design.

We cannot understate the *pride* felt by all in the process of designing a new language for the Mindstorm—and we can safely say that Jacob had every right to put his name on Jackll. By engaging in the creative process, the *student–mentor* relationship faded away, and we were all debating our ideas based on their merit, nothing more. We intend to continue this work in several ways: first, to create additional tools to aid students in implementing their own languages; second, to research the nature of the learning taking place when engaging in the creative process with such tools; and last, to continue working at integrating constructivist and exploratory ideals into introductory computing contexts.

5. REFERENCES

[1] H. Abelson and G. Sussman. *Structure and Interpretation of Computer Programs.* MIT Press, Cambridge, Mass., USA, 1985.

[2] David J. Barnes. Teaching introductory java through lego mindstorms models. In *Proceedings of the 33rd SIGCSE technical symposium on Computer science education*, pages 147–151. ACM Press, 2002.

[3] Valentino Braitenberg. *Vehicles: Experiments in Synthetic Psychology.* MIT Press, 1984.

[4] N. Cunniff, R. P. Taylor, and J. B. Black. Does programming language affect the type of conceptual bugs in beginners' programs? a comparison of fpl and pascal. In *Conference proceedings on Human factors in computing systems*, pages 175–182. ACM Press, 1986.

[5] Martha N. Cyr. *ROBOLAB: Teacher's Guide for ROBOLAB Software.* Tufts University, 1998.

[6] Barry Fagin. Using ada-based robotics to teach computer science. In *Proceedings of the 5th annual SIGCSE/SIGCUE ITiCSEconference on Innovation and technology in computer science education*, pages 148–151. ACM Press, 2000.

[7] Barry S. Fagin, Laurence D. Merkle, and Thomas W. Eggers. Teaching computer science with robotics using ada/mindstorms 2.0. In *Proceedings of the annual conference on ACM SIGAda annual international conference (SIGAda 2001)*, pages 73–78. ACM Press, 2001.

[8] Matthias Felleisen, Robert Findler, Matthew Flatt, and Shriram Krishnamurthi. *How to Design Programs: an introduction to programming and computing.* MIT Press, 2001.

[9] Ralph Hempel. pbforth. Available from http://www.hempeldesigngroup.com/lego/pbFORTH/, November 2002.

[10] Matthew C. Jadud. TeamStorms as a theory of instruction. In *IEEE International Conference on Systems, Man, and Cybernetics.* IEEE, 2000.

[11] Frank Klassner. A case study of lego mindstorms' suitability for artificial intelligence and robotics courses at the college level. In *Proceedings of the 33rd SIGCSE technical symposium on Computer science education*, pages 8–12. ACM Press, 2002.

[12] Michael Kolling and John Rosenberg. Blue: a language for teaching object-oriented programming. In *Proceedings of the twenty-seventh SIGCSE technical symposium on Computer science education*, pages 190–194. ACM Press, 1996.

[13] Michael Kolling and John Rosenberg. Guidelines for teaching object orientation with java. In *Proceedings of the 6th annual conference on Innovation and technology in computer science education*, pages 33–36. ACM Press, 2001.

[14] Deepak Kumar and Lisa Meeden. A robot laboratory for teaching artificial intelligence. In *Proceedings of the twenty-ninth SIGCSE technical symposium on Computer science education*, pages 341–344. ACM Press, 1998.

[15] O. Lecarme. What programming language should we use for teaching programming. In W.M. Turski, editor, *Programming Teaching Techniques*, pages 61–74. North-Holland Publishing Company, 1973.

[16] S. Papert. *The Children's Machine: Rethinking School in the Age of the Computer.* Basic Books, New York, NY, 1993.

[17] Bryan Ratcliff. Algol 68 and structured programming for learner-programmers. In *Proceedings of the Strathclyde ALGOL 68 conference*, pages 157–160, 1977.

[18] Charles M. Reigeluth, editor. *Instructional-Design Theories and Models*, volume 2. Lawrence Erlbaum Associates, Inc., Mahwah, NJ, 1999.

[19] Mitchel Resnick. Behavior construction kits. *Communications of the ACM*, 36(7):64–71, 1993.

[20] Franklyn Turbak and Robert Berg. Robotic design studio: Exploring the big ideas of engineering in a liberal arts environment (extended abstract). In *Proc. of the AAAI Spring Symposium on Robotics in Education*, 2001.

[21] S. Turkle and S. Papert. Epistemological pluralism and the revaluation of the concrete. In I. Harel and S. Papert, editors, *Constructionism*, pages 161–192. Ablex, Norwood, NJ, 1991.