A Tool for Integrating Lisp and Robotics in AI Agents Courses

Frank Klassner
Department of Computing Sciences
Villanova University
Villanova, PA 19085
610-519-5671

Frank.Klassner@villanova.edu

ABSTRACT

This paper presents the RCXLisp library, an extension to Common Lisp that allows students to investigate a broad range of artificial intelligence and computer science topics using the LEGO MindStorms® platform. The library supports both remote control and on-board programming of MindStorms robots. It also supports targeted communication between multiple LEGO robots and command-center desktops. The package is the only one to be integrated into most popular Common Lisp programming environments. This paper also summarizes student experiences with the package over the years 2000-2003 in an Artificial Intelligence course.

1. INTRODUCTION

In Fall 1997 the Introduction to Artificial Intelligence (AI) course at Villanova University was reorganized around the concept of agent-oriented design [7]. This approach was augmented in the Fall 1999 offering with LEGO MindStorms [3] in order to have students explore the relationships among hardware, environment, and software organization in agent design. The course traditionally used Common Lisp in its programming projects, but at the time of MindStorms' adoption, there was no Common Lisp development environment for MindStorms. This situation led me to try Not Quite C (NQC) [1] for team-based robotics projects and Common Lisp for all individual projects in search, planning, and abstract machine learning.

Student surveys indicated that the burden of working with two different languages distracted them from learning the AI concepts of the course. My students and I also found that the NQC language was overly limited by the MindStorms firmware to which it was targeted. We also found it difficult to try to coordinate several robots in the course's semester-end contest in which teams of robots compete against each other in a 20'x9' arena to capture and defend colored ping-pong balls.

Based on these observations, I decided to develop a Common Lisp library, RCXLisp, for programming the MindStorms' platform. This paper describes how the RCXLisp library augments the MindStorms platform as a tool for agent pedagogy. Although it is targeted at symbolic-oriented AI, there is no inherent reason the library cannot support numeric-level AI projects such as neural networks. The second section of this paper describes the MindStorms platform's design insofar as it affected the design of RCXLisp. The paper's third section describes the RCXLisp library and support firmware. The fourth section presents experiences some projects that have been written in RCXLisp for the Villanova AI course. The fifth section discusses student experiences with the package over the four semesters (2000-2003) that it has been used in the AI course. The sixth section concludes with plans for future use and development for RCXLisp.

2. MINDSTORMS BACKGROUND

The RCX is the programmable brick at the heart of the LEGO MindStorms kit. Figure 1 shows an RCX in a simple robot.

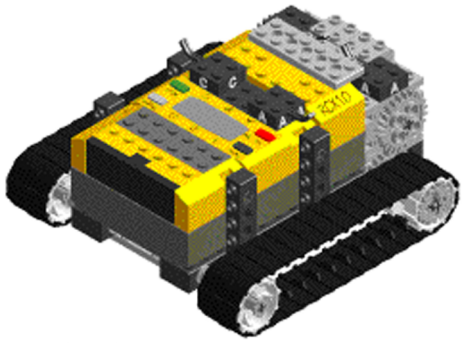

Figure 1. RCX unit. IR port is "in front," at lower left. Sensor ports are gray squares behind the IR port. Output ports are dark squares located beyond the LCD display.

The RCX has a 16MHz CPU (Hitachi H8/3292 microcontroller), 16KB RAM (and another 16K of ROM routines), and houses an infrared (IR) transmitter/receiver for sending and receiving data and commands from a desktop PC or from other RCXs. The IR transceiver has a range of 15-25 feet, depending on lighting conditions and reflectivity of walls. Version 1.0 of the platform used a serial-port IR Tower for broadcasting messages and software to the RCX from a desktop. Later versions (the current version is 2.0) replaced the serial device with a USB Tower whose IR receiving range is more limited: 8-12 feet. The RCX unit has 3 input ports, 3 output ports (labeled A, B, and C), and a 5-"digit" LED display. LEGO markets touch, light, temperature, and rotation sensors that can be connected to the RCX's input ports; third-party vendors offer magnetic compass sensors, ultrasonic distance sensors, and infrared distance sensors. Motors, LEDs, and infrared emitters can be attached to the output ports.

The RCX's replaceable firmware models a primitive virtual machine. It can be used in autonomous mode (the robot's behavior depends only on the program loaded into its memory) or in direct-control mode (a control program on a desktop computer broadcasts a series of instructions to the robot for on-board execution). Educators contemplating the use of Lego's Vision Command Camera system should note that the camera system is not connected directly to the RCX but relies on direct-control mode to control the RCX based on what the camera detects. The firmware supports 32 sixteen-bit global integer registers that can

be shared by up to 10 threads. Each thread can allocate up to 16 private registers. Only integer arithmetic is supported by the standard firmware.

From the hardware perspective, I do not believe that the available set of sensors for MindStorms is any more limiting for AI or robotics work at the collegiate level than that available at considerable greater cost for other robotics platforms. However, the RCX's 16KB of non-upgradeable onboard RAM does represent a problem for projects involving large-footprint real-time schedulers and planners.

LEGO firmware uses a broadcast protocol. It does not support targeted message-passing. If one has three RCXs in the same vicinity, two of them cannot exchange messages without the third inadvertently receiving the messages. Furthermore, it is not possible to have a desktop application coordinate several robots of a team without a mechanism to address the robots' RCXs individually. The firmware's lack of support for a call stack limits on-board programs' use of abstraction because nested function or procedure calls are not possible. LEGO firmware does not support dynamic memory allocation.

3. RCXLISP

3.1 Extended Firmware

Both as means of overcoming the problems cited earlier and as part of a larger project aimed at improving the MindStorms platform's usefulness in collegiate computer science curricula, I have developed the RCXLisp programming libraries. This package allows one to work with the RCX unit from the LEGO MindStorms® kit using Common Lisp.

The RCXLisp package is compatible with LEGO's firmware. It is, however, designed primarily to work with extended firmware my student Andrew Chang and I designed that supports wireless networking and most of the bytecodes from version 1.0 of LEGO's firmware. This extended firmware is called "Mnet firmware," and supports directed IR communication by adding source and target fields to the basic LEGO protocol and by allowing each RCX to set a 1-byte ID value for itself. Mnet firmware allows an RCX to restrict from what other RCXs it will accept messages.

As of version 1.3, Mnet firmware does not support dynamic memory allocation. Future versions are expected to include this capability (along with garbage collection) in order to support a larger subset of Common Lisp's functionality. I believe this lack of functionality is mitigated by RCXLisp's ability (discussed later) to integrate desktop remote control with on-board autonomous programs; problems requiring recursive or nested function calls can be solved on a desktop Lisp environment and the results can be communicated to the RCX's on-board control program.

3.2 Language Design

Moving from hardware-oriented issues, let us now discuss the design and organization of the RCXLisp language itself. Specifically, RCXLisp lets one

- remotely control the RCX from a Common Lisp program running on a desktop computer,
- write RCXLisp programs to run on the RCX,
- create and compile RCXLisp programs for downloading to RCXs "on the fly," from within Common Lisp desktop environments,
- simultaneously control more than one RCX from a single MindStorms infrared transceiver tower
- set up a network of RCX units that can communicate with each other in a targeted manner.

The RCX libraries support both the older serial-port infrared transceivers and the newer USB-port towers that LEGO is currently shipping. It is also possible to use the libraries to control more than one tower, opening up the possibility of extending the remote-control radius of a desktop through strategic placement of multiple IR towers in one room.

RCXLisp has two components. The first is "Remote RCXLisp," which is a collection of macros, variables, and functions for remotely controlling RCX units from a desktop. The second is "RCXLisp" proper, which is a subset of Common Lisp that can be cross-compiled to run on RCX firmware for autonomous control of the unit. The next two subsections provide details on the two parts of the language library, with the goals of showing the tight integration among Lisp environment, RCX, and desktop control processes the library makes possible, and the Common Lisp language features that the library encourages students to learn about as they set up robotics-inspired projects.

3.2.1 Remote RCXLisp

Since "Remote RCXLisp" is intended to run within a desktop Common Lisp environment, the design goal of this language was to adhere as closely as possible to the Common Lisp standard in Guy Steele's text, "Common Lisp: The Language, " 2nd edition (aka "CLTL2").

The "Remote RCXLisp" library provides users with the **with-open-com-port** and **with-open-rcx-stream** macros to set up communication environments for contacting the RCX units. These macros are modeled closely on Common Lisp's "with-open-stream" macro. **With-open-com-port** is used to specify the communication port (serial or USB) over which an RCX is to be contacted, and **with-open-rcx-stream** is used to define the RCX-unit-specific data stream that will use a port stream. The code in Figure 2 shows how the macros are used and provides an example of the functions that can be invoked to control an RCX from within the macros' communication environment. There are 45 functions defined for RCX control. Common Lisp functions are capitalized for easy readability.

The body of the "full-speed-ahead" function contains the following examples of "Remote RCXLisp" functions for controlling an RCX: **set-effector-state** (for initializing and controlling motors), **set-sensor-state** (for initializing how an input sensor port will be used to gather data), and **sensor** (for accessing the current value of a sensor port). The testing function uses **alivep** to determine if the RCX is in range and responding. It is useful to note for later discussion that the language includes a **var** function which asynchronously queries for the value stored in a given variable register.

All "Remote RCXLisp" functions take an optional final stream argument ("r" in full-speed-ahead, and "rcx10" in testing). One can forego repetitious typing of the stream argument, by using the

using-rcx macro as in figure 2 to define a default stream for enclosed RCXLisp functions. This macro is closely modeled on Common Lisp's "using-slots" macro for object-oriented programming. It also serves to define for the programmer a symbolic environment in which the desktop acts as a director telling some particular RCX what to do next.

```
(DEFUN full-speed-ahead  (r s dir)

"This will make the rcx in R go at speed S in direction DIR until touch sensor
on its '2' port returns 1."

(LET ((result 0))

    (set-effector-state '(:A :B :C) :power :off r)
            ;in case things are in an inconsistent state,
            ;turn everything off first
     (set-effector-state '(:A :C) :speed s  r)
     (set-effector-state '(:A :C) :direction dir  r)
              ; dir is eq  to  :forward, :backward, or :toggle
             ; no motion will occur until the
             ; next call to set-effector-state
     (set-sensor-state 2 :type :touch :mode :boolean  r)
     (set-effector-state '(:A :C) :power :on r)
     (LOOP ;this loop will repeat forever until sensor 2 returns a 1
         (SETF result (sensor 2 r))
         (WHEN (AND (NUMBERP result)
                          ;needed to keep = from causing error if
                          ;sensor function returns nil.
                         (= result 1))
            (RETURN)))
     (set-effector-state '(:A :C) :power :float r))))



(DEFUN testing ()

(with-open-com-port (port :LEGO-USB-TOWER)

  (with-open-rcx-stream (rcx10 port :timeout-interval 80 :rcx-unit 10)

   ; increase/decrease  serial timeout value of 80 ms depending on

   ;environmental factors like ambient  light.

   (WHEN (alivep rcx10)

       (full-speed-ahead rcx10  5 :forward)))))
```

Figure 2. Sample "Remote RCXLisp" code

```
(DEFUN full-speed-ahead  (r s dir)

"This will make the rcx in R go at speed S in direction DIR until touch sensor
on its '2' port returns 1."

(LET ((result 0))

   (using-rcx  r

    (set-effector-state '(:A :B :C) :power :off)
            ;in case things are in an inconsistent state,
            ;turn everything off first
     (set-effector-state '(:A :C) :speed s )
     (set-effector-state '(:A :C) :direction dir )
              ; dir is eq  to  :forward, :backward, or :toggle
             ; no motion will occur until the
             ; next call to set-effector-state
     (set-sensor-state 2 :type :touch :mode :boolean)
     (set-effector-state '(:A :C) :power :on )
     (LOOP ;this loop will repeat forever until sensor 2 returns a 1
         (SETF result (sensor 2 ))
         (WHEN (AND (NUMBERP result)
                          ;needed to keep = from causing error if
                          ;sensor function returns nil.
                         (= result 1))
            (RETURN)))
     (set-effector-state '(:A :C) :power :float ))))
```

Figure 3. Using 'using-rcx to clean up "Remote RCXLisp" code.

Programs that are intended to be executed on an RCX are first compiled within the RCXLisp desktop environment and then downloaded through the IR Tower to the RCX. Firmware is also loaded from the desktop Lisp environment. These two actions are accomplished with the **download-firmware** and **rcx-compile-and-download** functions. It is important to note that these functions *are native to the Lisp environment*. That is, no non-Lisp mechanism is needed for these actions. Both LeJOS [6] (the MindStorms Java virtual machine developed originally by Jose Solarzano) and Wick et al.'s Lego/Scheme compiler [8] require a separate command-line system program to download compiled code or firmware into the RCX. The recursive-descent parser+compiler in **rcx-compile-and-download** is implemented in Lisp.

3.2.2 "RCXLisp" Proper

RCXLisp is the subset of Common Lisp (with a few non-standard additions) that can be compiled and downloaded to run on an RCX unit autonomously. As with "Remote RCXLisp," the design goal was to follow CLTL2's standard as closely as possible, and to maintain compatibility with Lego firmware as well as Mnet extended firmware. However, because even the extended firmware does not yet support indirect addressing or call stacks, some of the "functional" nature of Common Lisp is still missing. For example, some RCXLisp functions cannot accept variable values for some arguments; they can only accept constants.

For consistency with "Remote RCXLisp," and to make it more straightforward to transfer desktop Lisp code to the RCX, RCXLisp implements all of the RCX-control functions in "Remote RCXLisp." In RCXLisp however, control functions like **set-sensor-state** do not have an optional final stream argument since it is assumed that the Lisp code will only be executed on the RCX unit itself. If an RCX needs to control the behavior of another RCX unit, it does not download programs into the other RCX. Instead, it sends integer-valued messages that the other RCX must interpret to determine what action to take.

RCXLisp supports analogs to the following subset of Common Lisp control expressions, along with their standard semantics defined in CLTL2: DOTIMES, COND, IF, LOOP, PROGN, RETURN, and WHEN. The RCXLisp language supports 16-bit signed integer arithmetic with the following operators: +, -, *, and / (integer division). RCXLisp provides the >, >=, <, <=, =, /=, and EQUAL Common Lisp comparison operators in their full functionality. Just as Common Lisp allows one to use comparison invocations such as "(< 2 x 6)" to test for when the value of x is between 2 and 6, so too does RCXLisp. It also provides a limited version of the Common Lisp RANDOM function.

RCXLisp also supports the **not**, **and**, and **or** boolean operators, along with their CLTL2 semantics (including the "boolean short-circuit"). RCXLisp does not support floating point arithmetic, but it does support the boolean data type (i.e. T and NIL) and certain keyword constants (although new keywords cannot be defined yet).

Constants are declared in RCXLisp programs with **defconstant**, which follows the semantics of the Common Lisp DEFCONSTANT form, and global variables can be declared with **defvar**, whose semantics are only partially the same as those of the Common Lisp DEFVAR form. Values (signed 16-bit integers

and T and NIL) can be stored into variables using **setq**, which is similar to the Common Lisp SETQ form. Currently there is no analog in RCXLisp to the Common Lisp SETF macro.

Since general function calls are not supported, RCXLisp does not have an analog to the Common Lisp DEFUN form. In an effort to support *some* kind of code abstraction, the language design borrows inspiration from NQC's emphasis on macros for code abstraction and includes a **defmacro** form that follows the complete semantics of Common Lisp's DEFMACRO form. RCXLisp also borrows from Rodney Brooks' much earlier (and proprietary) "L" language [2] a desire for simplicity (many advanced Common Lisp functions are not available in that language) and memory-conservation that is necessary for squeezing as much programming as possible into the small memories available on most robot platforms even today.

The language also provides two special-purpose forms that are neither in the "Remote RCXLisp" language nor the Common Lisp language. The first form is **defregister**, which is used to bind symbolic variable names to particular RCX variable registers. **Defregister** allows a programmer to tie a symbolic variable name to a given register so that a "Remote RCXLisp" program on a desktop using **var** to query a register can be guaranteed to access the intended variable value.

The second non-standard form is **defthread**, which is used to define RCX threads to run on an RCX unit. Calling this form "non-standard," however, is less of an indictment of RCXLisp than of the Common Lisp spec itself since as of 2003 no progress has been made in formalizing threading in the language!

```
(defconstant *receiver* 1)
(defregister 4 *LIMIT* 16)

(defthread (signaller) ()
  (loop
   (send-message 78)
   (sleep 15) ;; this is to leave the IR port silent for a
          ;; short time in case a desktop is sending a message.
  ))

(defthread  (alpha :primary t)  ( )
  (let  ((diff1 0)
       (diff2 0))
   (set-sensor-state *receiver* :type :light :mode :raw)
   (setq diff1 (abs (- (sensor *receiver* :raw)
                       (sensor *receiver* :raw))))
   (setq diff2 (abs (- (sensor *receiver* :raw)
                       (sensor *receiver* :raw))))
   (start-rcx-thread signaller)
   (loop
     (when (>= (abs (- diff1 diff2)) *LIMIT*)
        (play-tone 500 1))
     (setq diff1 diff2)
     (setq diff2 (abs (- (sensor *receiver* :raw)
                         (sensor *receiver* :raw)))))))
```

Figure 4. Multi-threaded RCXLisp Sample Code

Figure 4 shows a sample RCXLisp program that illustrates many of the features described above. The program will beep whenever the RCX is carried too close to a reflective object. This code makes the IR port on an RCX work together with a light sensor on sensor port 1 to implement a simple proximity sensor. The "signaler" thread repeatedly sends out an arbitrary integer message through the RCX's IR port. When the front of the RCX gets too close to a tall obstacle, the IR signal from the IR port will reflect back, and the light sensor will pick this echo up. As the reflections increase in intensity, the light sensor's value will jump more wildly. The value of *LIMIT* may need to be experimented with. It is declared as a register variable because this allows it to be modified dynamically by a "Remote RCXLisp" program on a desktop, by using **var** to access and set register 4.

3.3 Platform Support

The RCXLisp libraries are supported on the Allegro (Franz), MCL (Digitool), and Xanalys Common Lisp environments, on both Windows (98, 2000, and XP) and Mac OS X.

4. PROJECTS WITH RCXLISP

4.1 MindStorms Equipment

The AI course at Villanova uses RCXLisp and MindStorms in team-based active-learning projects. Each team's kit for constructing LEGO robots contained the following hardware:

(a) 3 Mindstorms Robotic Invention Systems packages

(b) 3 more light sensors beyond the three in (a)

(c) 3 more touch sensors beyond the six in (a)

(d) 3 more motors beyond the six in (a)

(e) 2 LEGO rotation sensors

(f) 2 HITECHNIC magnetic compass sensors

(g) 1 HITECHNIC infrared distance sensor

(h) 2 HI-TECH STUFF limit-switch adapters for motor ports

(i) 24 rechargeable batteries

(j) 1 large lockable toolbox to hold all of the above as well as a partially-completed robot.

The robotics laboratory also has a few extra sensors such as two HITECHNIC ultrasound distance sensors and two LEGO Vision Command cameras for special projects as they arise. Hitechnic (www.hitechnic.com) has ceased operations, but the third party manufacturer hi-techstuff (www.hitechstuff.com) offers many similar sensors except unfortunately for the compass sensor. We are in negotiations to get a new source of compass sensors, since we have found them to be very useful in navigation problems.

4.2 Project Descriptions

The following RCXLisp-based projects have been developed for the course:

I. Simple-Reflex Robot Design (and RCXLisp Orientation). This 10-day project's goal was to show students how robots with simple stimulus-response rules and no model of the environment could achieve effective behaviors. This project asked students to design a robot based on a tread-wheeled "Pathfinder" model described in LEGO's user manual. Students were required to start with this basic design in order to reduce time spent on distracting mechanical engineering issues, but they were encouraged to mount sensors as needed.

Students first built a robot that used a compass sensor to maintain a bearing (team 1 goes North, team 2 goes South, etc.)
They next added code to monitor either (a) whether a robot is too close to a wall, or (b) whether, via feedback from mounted light or touch sensors that the robot was about to roll over a dark tile on the floor. In both cases the robot had to back up and/or turn to avoid the obstacle for a brief time, then resume moving ahead on

the bearing. This was implemented twice using RCXLisp and "Remote RCXLisp."

II. Robot Odometry. This 2-week project's goal was to help students understand the major factors that can introduce error into a robot's internal representation of where it believes it currently located in the world – an important issue in any navigation process. It also introduced them to the importance of maintaining a representation of state (the robot's position).

Each team was required to design and build a robot that would measure the perimeter of a convex black shape on a light background on the floor. The reported measurement (over 190 cm) had to be accurate to within +/- 3 cm, and had to be obtained within 1 minute from the time the robot was started. The project allowed use of dead-reckoning and landmark-based navigation techniques. Although all teams succeeded in this project, all were surprised at how short the 1-minute time limit soon appeared in light of the accuracy constraint. If students elected to use a compass sensor to record orientation, then the shape was allowed to be either convex or concave.

III. Robotic 8-Puzzle Solver. This 2-week project had the goal of showing students that knowledge representations (data abstractions) that speed up search-based problem solvers can produce solution representations that are not easily translated into control programs for hardware.

The project had two stages. In the first stage students had to develop a knowledge representation and Lisp search program to solve the 8-Puzzle. The team developed a set of four operators that involved conceptually moving the "space" up, down, left, or right, rather than 32 operators for moving each of the numbered tiles up, down, left, or right. The students observed that this design decision dramatically reduced the branch factor of the search tree (4 vs. 32), leading to a faster execution time for the game-solver.

The second stage required students to write a "Remote RCXLisp" program that sent remote-control messages to a Mindstorms robotic arm mechanism to move pieces in an 8-Puzzle according to the solution developed by stage 1's programming. It was in this stage that students discovered that the search space reformulation trick ultimately cost them in the complexity of the translation their second program had to perform on the "move space" operator list to "move tile at (2,2) to (2,1)" types of commands.

Figure 5. Capture-the-Balls Competition. Later competitions allow dangling of RCX IR Towers over the playing field for better remote-control approaches.

IV. Capture-the-Balls Contest. This final project's goal was to help students tie the skills they developed in projects I-III. Each team was required to design and build a team of two or three robots each no larger than 1 cubic foot. The robots' task was to play in a 20-minute contest against other teams' robots. Contestants had to play in a 20x9 sq. ft. walled playing area in which each team had a 1 sq. ft. nest area, colored dark purple. All other portions of the playing field were light-yellow colored. Scattered throughout the field were black, white, and yellow ping pong balls. For each ball that was in a team's nest at the end of the contest, the following points were awarded: white +1, yellow +5, black –1. The playing field was marked with a black 1x1-foot grid whose lines were 1 cm wide. Figure 5 shows a view of one such contest layout.

Teams were encouraged to try a wide variety of game strategies, some of which required landmark-based navigation via the grid, others of which required state-space hill-climbing, and still others of which relied on probabilistic observations about the environment. Teams were also encouraged to make their use of strategies time-dependent: as the contest progressed, robots could switch strategies based on their current state (e.g. estimated score, position on field). Since robots were permitted to "attack" other nests and scatter or steal balls, there was a very wide variety of approaches that a team could explore, minimizing the risk of teams unintentionally duplicating their efforts.

Students were also encouraged to mix RCXLisp and "Remote RCXLisp" usage.

5. STUDENT EXPERIENCE WITH RCXLISP

5.1 Student Background

The elective AI course at Villanova has no formal programming prerequisites. Computer science majors typically take the course in their fourth year, by which time most majors have taken a Programming Languages (PL) course that briefly introduces them to Lisp or Scheme.

The course is also open to cognitive science minors and computer engineers, who generally have no programming experience in Lisp and at most one semester of introductory programming in Java.

5.2 Student Experience Reports

RCXLisp make extensive use of keyword and optional arguments, as well as streams and macros – concepts not often explored in depth in Common Lisp in courses like Programming Languages (PL) and almost never in Lisp-based AI courses simply because of a lack of motivating material. Students who I have taught in both courses have commented on how the RCXLisp environment's use of extended function argument capabilities helped them understand the benefits and pitfalls of these features better than when they briefly encountered them in the PL course. Several cognitive science minors have commented on how they felt they

could get past coding details faster in RCXLisp (and Common Lisp) than in Java because of the Lisps' lack of typing.

Both my students and I noticed the reduced overhead in learning how to program the RCXs. Since RCXLisp is just an extension of the Common Lisp they were already using the AI and PL courses, they could spend more time on the application problem rather than on learning yet another language.

The library's support for "on-the-fly" program generation and download also helped students appreciate the power of Common Lisp's lack of differentiation between code and data. Since Common Lisp function declarations are themselves merely formatted linked lists, students could generate plans as linked lists within a planner and then download the same data to the RCX as an immediately executable form.

Students have also commented on the immediacy of working with RCXs via the Common Lisp Listener: simply by invoking a function in the Listener, an RCX can be made to respond. This has helped them understand the differences and similarities between compiled and interpreted code.

The library's integration with low-level system functions such as infrared USB communication helps students get past the uninteresting details of port communication and instead concentrate on the AI-oriented problems of environmental noise, sensor sensitivity, and environmental nondeterminism.

6. CONCLUSIONS AND FUTURE WORK

RCXLisp is the first open-source Lisp approach for programming physical robots that supports both remote control and on-board programming of robots as well as targeted communication between multiple robots and command-center desktops. Coupled with the low cost and adaptability of MindStorms, RCXLisp should help make it easier for cost-conscious schools to add robotics-inspired projects to AI and courses, without having to turn computer science students into mechanical or computer engineers.

The library has separate standalone API functions for accessing serial ports generically and USB ports with MindStorms IR Towers attached, making it a useful basis for designing Lisp solutions for interfacing with other serial devices, but not for and USB devices. I have (ambitious?) plans to extend the API to cover USB generically.

Improvements remain. One important goal is to add call stack support and memory management support to the Mnet firmware. This would extend the Lisp functionality of the on-board language for RCXs (e.g add DEFUN and list-manipulation functions). Lest the lack of garbage collection seem too limiting for RCXLisp right now, it should be noted that leJOS [6] also does not currently implement a garbage collector in an effort to keep the JVM footprint as small as possible. A related possibility would be to eliminate firmware and target the H8 processor directly, as LegOS does [5].

Another goal is to integrate the LEGO Vision Command Camera into the RCXLisp library. This would give students a low-cost yet powerful tool for exploring machine vision problems at the undergraduate level.

From the standpoint of courseware improvements, I am working on a formal integration of RCXLisp with one or more open-source planners. The goal here would be to define several primitive action forms using RCXLisp. At issue would be deciding how low-level the plan operators should be. Alternatively, one could just supply RCXLisp with a vetted planner and leave the combination up to the students. This would most likely make the project too complicated for an undergraduate AI course, though.

7. ACKNOWLEDGMENTS

LEGO MindStorms and RCX are trademarks of the LEGO Group, which does not sponsor, authorize, or endorse any of the third-party work cited in this article. The author of this article has no financial relationship with the LEGO Group except for a discount purchase plan for MindStorms equipment for seminars run under NSF Grant No. 0306096.

I am grateful to Andrew Chang for his graduate independent study work that led to extending the MindStorms' firmware.

This material is based upon work supported by the National Science Foundation under Grant No. 0088884 and Grant No. 0306096. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation.

8. REFERENCES

[1] Baum, D. Not Quite C (NQC), Sept. 2003, http://www.baumfamily.com/nqc/

[2] Brooks, R. A. (1993), L: A Subset of Common Lisp, Technical report, Massachusetts Institute of TechnologyArtificial Intelligence Lab.

[3] Klassner, F., *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* (2002*).*

[4] Martin, F., The MIT HandyBoard Project, September 2003. http://lcs.www.media.mit.edu/groups/el/Projects/handy-board

[5] Noga, M. LegOS, September 2003, http://www.noga.de/legOS

[6] Solorzano, J. LejOS, Sep 2003, http://lejos.sourceforge.com

[7] Russell, S. and Norvig, P. *Artificial Intelligence: A Modern Approach.* 2nd edition, Prentice Hall, 2003.

[8] Wick, A., Klipsch, K., and Wagner, M. LEGO/Scheme compiler, http://www.cs.indiana.edu/~mtwagner/legoscheme