XS: Lisp on Lego MindStorms *

Taiichi Yuasa
Graduate School of Informatics, Kyoto University
Kyoto 606-8501, Japan
yuasa@kuis.kyoto-u.ac.jp

ABSTRACT

We present a Lisp system XS which is designed to control RCX blocks of the Lego MindStorms Robotics Invention System (RIS). Unlike previous Lisp/Scheme implementations for the MindStorms, the evaluator of XS runs autonomously on the RCX, with its own runtime stacks and garbage-collected heap. It communicates with the front-end subsystem on a PC, to provide an interactive programming environment with features such as backtrace, function trace, and terminal interrupt. The evaluator supports a language based on Lisp/Scheme, extended with functionality for interfacing with RIS devices such as motors and various kinds of sensors. It also supports mechanisms such as event waiting and asynchronous event handlers for controlling robots built with RIS.

1. INTRODUCTION

The Lego company, well-known worldwide by their toy blocks, is supplying a robotics system, called Lego MindStorms Robotics Invention System (RIS) [6]. The central component of the system is a programmable block called RCX, with an 8-bit CPU. By attaching motors, sensors, and other component blocks, one can build robots that are controlled by user-supplied RCX programs. Since even children can write simple programs to control their own robots, the RIS system is being used for education in programming and mechanical engineering.

In the programming environment supplied by the Lego company, RCX programs are described in a simple visual language [1]. Because of the simplicity, even small children can use the language. On the other hand, the visual language seems not suitable for learning realistic programming, because it is quite different from ordinary programming languages and its expressive power is limited. For instance, the language does not support parameterized subprograms nor user-defined variables. A C-like language NQC (Not Quite C) [2] is available, which is essentially a text-based variation of the visual language. Because of the limitation of the RCX firmware [8], the expressive power of NQC is also limited. For instance, although NQC has the concept of functions, no user-defined function can be invoked from another user-defined function.

An open-source operating system brickOS (formerly called legOS) [3] has been developed for RCX. By replacing the RCX firmware with brickOS, one can write control programs in the C language. One difficulty in using this OS, not only for children but also for expert programmers, is that its memory area is not protected against user programs and the OS easily crashes while executing a buggy program. In addition, because C is a compiler-oriented language, even for testing a simple functionality, one has to take a tedious procedure: create a source file, compile the program, download the binary, and run.

This paper presents a language system XS, which has been developed for the following objectives.

- Even children or novice programmers can easily use it.
- It can be used for developing advanced programs.
- It provides an interactive programming environment for efficient program development.

LegoScheme [11] and LegoLisp [5] have been developed for similar objectives. They support programming in extended languages in the Lisp family. In these systems, programs run in the front-end PC and send control instructions to the RCX. Because the front-end PC and the RCX communicate via infra-red (IR) signals, the RCX sometimes fails to receive control instructions, in particular when it is used in a moving robot. In addition, because the IR communication is slow, the RCX cannot receive control instructions that respond to sensor input, within a reasonable period of time. Our system XS also supports programming in a Lisp language in order to provide interactive programming environment. In XS, however, programs are downloaded to the RCX and are executed by an autonomous evaluator in the RCX. This solves the above problems in the two Lisp-based language systems.

*This project is sponsored by the Information-technology Promotion Agency (IPA) as an Exploratory Software Project.

In order to provide an interactive and satisfactory programming environment, XS supports the following features.

1. read-eval-print loop
2. interactive definition and re-definition of functions
3. appropriate error messages with backtraces
4. function trace and untrace
5. dynamic object allocation and garbage collection
6. robustness against program errors and stack/buffer overflow
7. terminal interrupts
8. truly tail-recursive interpreter
9. event/timer waiting and asynchronous event watchers
10. interface to RIS devices such as motors, sensors, lights, and sounds

These features other than the last three are commonly found in ordinary Lisp systems. Feature 8 is found in ordinary Scheme interpreters and feature 9 or its variation is found in multi-threaded Lisp systems. Therefore, from the user's point of view, XS looks as an ordinary Lisp system with extensions for controlling RIS devices.

The next section introduces some features of RCX that are necessary to understand the rest of the paper. Section 3 gives an introduction to the XS system, which is intended to provide a realistic image of the system. Section 4 presents the language that XS supports, followed by a simple program example in Section 5. Section 6 provides implementation details for important features of XS. Finally, in Section 7, we report the current status of the project.

2. RCX

The CPU in RCX is a 16 MHz Hitachi H8 Microprocessor with 16-bit addressing space. Each RCX has 32K (or 16K) bytes of ROM and 32K bytes of RAM. The contents of the ROM memory cannot be modified by the user and some memory locations in the RAM are reserved for use by the ROM program. The remaining memory area in the RAM is used for the RCX firmware (or another OS) and user programs. No secondary storage is available. Since memory constraint is severe, space is more important than speed in developing a language system for RCX.

At the front of an RCX block (see Figure 1) is an IR port, which provides the only communication means with the front-end PC. The PC uses a Lego-supplied "IR tower" for IR communication. The IR tower is connected with the PC through a serial port (RCX models 1.0 and 1.5) or through a USB port (RCX model 2.0). Since the communication range of IR is relatively small, the PC and the RCX sometimes fail to receive messages, in particular when the RCX is embedded in a moving robot.

On the surface of an RCX block are three effector ports (ports A to C) and three sensor ports (ports 1 to 3). At the

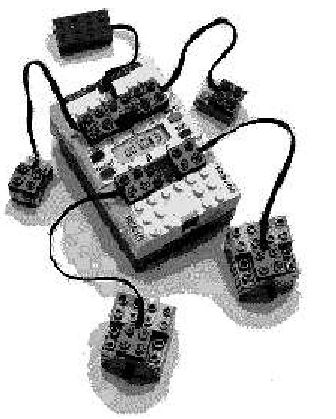

Figure 1: RCX attached with two motors (ports A and C), two touch sensors (ports 1 and 3), and a light sensor (port 2).

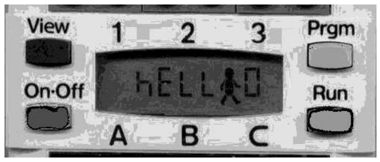

Figure 2: LCD, four buttons, and port labels.

center of the block is an LCD display (see Figure 2) which can display up to five characters. Around the LCD display are four buttons *On-Off*, *Prgm*, *Run*, and *View*. *On-Off* is used to turn on/off the RCX, *Prgm* for selecting a user program stored in the RCX, and *Run* for starting the selected program. *View* is intended for monitoring the running program, but this button is rarely used.

The RCX is powered by six AA batteries stored in the bottom of the block. Attached motors and active sensors are also powered by these batteries. Even after the RCX is turned off, contents of the RAM memory are maintained. Thus when the RCX is turned on, previously downloaded user programs are ready to run.

3. AN OVERVIEW OF XS

The system of XS consists of the front-end subsystem on PC and the subsystem on RCX. These subsystems cooperate with each other to provide an interactive programming environment which looks like an ordinary Lisp system.

Figure 3 illustrates a sample session with XS. When the front-end subsystem is invoked (line 1), it displays a prompt "`>`" and starts interaction with the user. Following the prompt, the user inputs a function definition (line 4) and tests it (line 7). Since the function definition refers an undefined variable `nil`, an error message is printed out (line 8) followed by a backtrace (line 9). Then the user defines the variable `nil` with the empty list as its value (line 10), and tests the function again (line 12). This time the function returns a correct answer (line 13) and the satisfied user ends the XS session (line 15). The user sees nothing special with this sample session. Internally, however, things are quite

```
% xs
Welcome to XS: Lisp on Lego MindStorms

>(define (ints n)
   (if (= n 0) nil (cons n (ints (- n 1)))))
ints
>(ints 3)
Error: undefined variable -- nil
Backtrace: ints > ints > ints
>(define nil ())
nil
>(ints 3)
(3 2 1)
>(bye)
sayonara
%
```

Figure 3: A sample XS session (line numbers are added for explanation).

different from ordinary Lisp systems, because the evaluator is located in an RCX.

When the front-end subsystem is invoked, it first checks whether the RCX subsystem is ready. If not ready (e.g., the user forgot to turn-on the RCX), then the front-end gives up interaction with the user.

```
% xs
RCX is not responding.
Make sure RCX is running, and try again.
%
```

When an S-expression is input from the PC keyboard, the front-end preprocesses the S-expression and sends the result to the RCX subsystem. The evaluator then evaluates the S-expression and sends back the value, which is displayed on the PC display. In case of a function definition, the evaluator installs the definition and returns the symbol that names the function.

The user can interrupt a running program by pressing Control-C. In the following example of interaction, the user pressed Control-C while the `let` expression is executing an infinite loop.

```
>(let loop () (loop))
Error: terminal interrupt
Backtrace: let > #<function>
>
```

A request for terminal interrupt from the front-end is passed to the RCX through IR communication. If the RCX is out of the IR range, it fails to receive an interrupt request. For such a case, XS provides another means for terminal interrupt: pressing the *View* button. If the RCX is within the IR range, terminal interrupt by the *View* button behaves in exactly the same way as in the case of terminal interrupt by Control-C. Otherwise, the front-end keeps waiting for a return value from the RCX, because there is no way for the front-end to recognize the interrupt caused by the *View* button. In this case, the user should move the RCX into the IR range after interrupting the program, and then press Control-C.

4. THE LANGUAGE

As will be clear from the sample session in the previous section, the language of XS is based on Scheme [4] rather than Common Lisp [9]. This is because Scheme allows more compact implementation for the following reasons.

- Since functions and variables share a single name space in Scheme, a symbol object needs to have only one slot to store its value.
- Since loops are realized by tail-recursive calls in Scheme, we do not need separate loop constructs. This reduces the code size of the evaluator.

4.1 Data types

The language of XS supports the following data objects.

- 14-bit signed integers
- conses (or pairs)
- functions
  - built-in functions
  - lambda closures (user-defined functions)
- symbols
  - predefined symbols for built-in functions
  - user-defined symbols
- miscellaneous
  - the empty list
  - truth values `#f` and `#t`

Among these, conses, lambda-closures, and user-defined symbols are represented as pointers to heap-allocated cells, whereas the others are represented as immediate data. Because of this design, the heap of the RCX subsystem starts with no cells allocated in it. We will discuss more details of data representation in a later section.

It is a pity that integers are 14-bit long. The largest positive integer is 8,191 ($= 2^{13} - 1$). This is because a word in RCX is 16-bit long and XS uses two bits as an object tag. Besides, the small memory size of RCX requires integers be immediate data. Heap-allocated integers would consume the heap space and operations on them would make the evaluator much larger.

In addition to the above data objects, XS defines *pseudo data objects* to facilitate program coding. Each such object is converted to an actual data object by the front-end reader.

- characters, converted to ASCII code integers.
  ex. `#\a` ⇒ `97`

- strings, converted to lists of ASCII code integers. ex. `"abc"` $\Rightarrow$ `(97 98 99)`
- reader constants, used to specify integers whose actual values are not important. Names of reader constants begin with a colon "`:`". Some of the reader constants are:
  `:a` (used to specify the effector port A)
  `:forward` (used to move forward a motor)
  `:white` (the brightest value for light sensors)
  `:La0` (the lowest pitch of the RCX sound system)

4.2 Built-in functions

The Appendix lists all functions (top-level forms, special forms, and ordinary functions) that XS supports. These functions are classified into two categories: those supported commonly by many conventional Lisp/Scheme systems (listed in Appendix A.1) and those specific to Lego MindStorms (listed in Appendix A.2).

Most of the common functions behave as specified in IEEE Scheme [4]. Only a few comment would suffice for understanding the set of common functions. XS does not support first-class continuations of Scheme. Instead, it supports `catch` and `throw` of Common Lisp for dynamic nonlocal exit. The function `gc` forces garbage collection and returns the number of free cells. The function `write` in XS corresponds to `display` in Scheme. Since strings in XS are actually lists of integers, the function `write` cannot display strings appropriately as intended by the user. The function `write-string` is added to solve this problem.

The Lego-specific top-level forms `last-value` and `ping` cover the unreliable IR communication. `last-value` literally returns the value of the last S-expression. This form is used when the front-end failed to receive the last value. `ping` is used to see if the RCX is alive and within the IR range. It immediately returns a certain value if the RCX is ready.

The function `sleep` suspends execution of the current program for a specified period of time. The time to suspend is given in 1/10 seconds. Although RCX can be controlled in the order of milliseconds, it seems that 1/10 second is short enough and the maximum suspension time of 13 minutes ($\approx (2^{13} - 1) \div 10 \div 60$) is long enough to control XS applications. For the same reason, the `time` function returns the "current time" in 1/10 seconds. The system clock of XS is initialized to zero when the RCX subsystem is started up. Because of the short length of XS integers, the value of `time` overflows in about 13 minutes. Therefore, when the user wants to measure a time span, it is recommended to reset the system clock with the function `reset-time`, before obtaining the starting time.

The special form `(wait-until` *expr*`)` waits until the event specified by *expr* happens. *expr* may be any expression, which is periodically evaluated until it returns true. For example,

```
(wait-until (pressed?))
```

waits until the Prgm button on the RCX brick is pressed (see below for the predicate `pressed?`).

The special form `with-watcher` provides another means for event waiting. Its general form is:

$$\texttt{(with-watcher ((}\mathit{event}_1\ \texttt{.}\ \mathit{handler}_1\texttt{)}\ \ldots\ \texttt{(}\mathit{event}_n\ \texttt{.}\ \mathit{handler}_n\texttt{))}\ \texttt{.}\ \mathit{body}\texttt{)}$$

During execution of the *body*, the evaluator periodically checks the specified *events*. If some $event_i$ evaluates to true, then execution of the *body* will be suspended and the corresponding $handler_i$ will be executed. Even during the execution of $handler_i$, the evaluator keeps checking $event_{i+1}$ to $event_n$ and if some $event_j$ ($j > i$) evaluates to true, then the evaluator will suspend the running $handler_i$ and executes $handler_j$. When execution of $handler_j$ is finished, the suspended execution of $handler_i$ will be resumed. That is, `with-watcher` allows nested execution of the *handlers* with the priority of the *events* in the reverse order they appear in the `with-watcher` form.

XS supports four kinds of sensors: light sensors, rotation sensors, temperature sensors, and touch sensors. Among these, light sensors and rotation sensors are active sensors. They must be turned on with `light-on` or `rotation-on` before use and are recommended to be turned off with `light-off` or `rotation-off` after use. The sensor values are obtained by `light`, `rotation`, `temperature`, and `touched?`. The arguments to these functions specify the sensor ports to which the sensors are connected.

The function `motor` sets up the direction of the motor that is connected to the specified effector port. The motor speed is set up by the function `speed`. These functions return the second argument so that multiple motors can be controlled with nested calls such as

```
(speed :a (speed :c :max-speed))
(motor :a (motor :c :forward))
```

The function `play` initiates playing a tune as specified by an association list of pitches and lengths. Pitches are integers but are usually expressed by reader constants `:La0`, `:La#0`, `:Si0`, `:Do1`, ..., `:So8`, `:So#8`, `:La8`, and `:pause`. The predicate `playing?` is used to check if the RCX has finished playing the tune. The predicate `pressed?` returns true if the *Prgm* button is being pressed, and returns false otherwise.

The function `puts` displays the given string (up to the first five characters) on the LCD display. `putc` displays the given character at the specified position on the LCD display and `cls` clears the entire LCD display. Finally, `battery` is used to check the battery level of the RCX.

5. A PROGRAM EXAMPLE

Figure 4 gives a simple program that controls a "land rover" which can bypass obstacles. Initially, the rover moves forward playing a tune. When the rover hits an obstacle, it backs up, turns right or left randomly, and then moves forward again. The rover repeats this action until the *Prgm*

```
(define (forward)
  (motor :a (motor :c :forward))
  (play '((:Re4 . 2) (:Do4 . 1) ...)))

(begin
  (speed :a (speed :c :max-speed))
  (forward)
  (with-watcher
    (((touched? 2)
      (motor :a (motor :c :back))
      (sleep 5)
      (motor (if (= (random 2) 0) :a :c)
             :forward)
      (sleep 5)
      (forward)))
    (wait-until (pressed?))
    (motor :a (motor :c :off))
    ))
```

Figure 4: A sample program: Land Rover.

button is pressed. The function `forward` defines the action to go forward. It directs the motors to move forward, initiates playing the tune, and returns. The `begin` expression gives the main action of the rover. It sets the speed of the motors, invokes `forward`, waits until the *Prgm* button is pressed, and then stops the motors. While waiting, each time the touch sensor hits something, the program moves back the motors for half a second, moves either motor (chosen randomly) forward for half a second while keeping the other motor moving backward, and moves the rover forward again.

6. IMPLEMENTATION

The XS system is built on top of a tiny operating system legOS [10], which is now called brickOS [3]. The version of legOS we are using is 0.2.4. legOS provides a **kernel** that replaces the Lego-supplied firmware, and a set of utility programs to download the kernel and user programs. The utility programs use the Lego Network Protocol (LNP) for IR communication. The LNP protocol does not guarantee message arrival but does guarantee the validity of a message if it arrives at the destination. User programs are written in C, which are then compiled by the GNU cross compiler for the H8 Microprocessors and downloaded by a utility program. Although the GNU cross compiler accepts a full set C language, run-time libraries are limited to those that are supplied by legOS. legOS itself is written mostly in C and partly in an assembly language.

The XS system is entirely written in C. The RCX subsystem is regarded as a user program for legOS. Thus it is compiled by the GNU cross compiler and downloaded by a utility program of legOS. The front-end subsystem is compiled by an ordinary C compiler available on each PC. These subsystems communicate with each other with the LNP protocol.

The front-end subsystem includes a reader that reads S-expressions and a printer that displays S-expressions. These components are similar to those in conventional Lisp systems. Each S-expression is preprocessed in the front-end before passed to the evaluator in the RCX. The preprocessor does many things. Indeed, it performs whatever it can do to reduce the load of the evaluator, such as syntax checking of special forms and some optimizations. We will mention more about the work of the preprocessor later in this section.

The RCX subsystem is essentially a "receive-eval-return" loop. It waits for an S-expression at the beginning of each iteration. When an S-expression arrives, it will be evaluated by the evaluator, and the value will be sent back to the front-end. The RCX subsystem repeats this process until it receives a command to stop, which is sent by the front-end when the user inputs "`(bye)`".

The evaluator is a truly tail-recursive interpreter for preprocessed S-expressions. The evaluator itself is defined as a recursive function in C but nevertheless it does not require the so-called "trampoline" mechanism to be tail-recursive. Details of the evaluator are beyond the scope of this paper and are left to another paper. Tail-recursive interpreter is mandatory for systems like XS that runs on a small memory space. It saves the heap space as well as the stack space, because unnecessary function frames do not remain in the stack.

The legOS kernel occupies about half of the RAM memory. XS uses the remaining half as follows.

- code of RCX subsystem: 11K bytes[1]
- I/O buffer: 256 bytes
- C stack: 1K bytes, i.e., 512 words
- value stack: 0.5K bytes, i.e., 256 words
- heap: 3K bytes for 768 cells
- bit table: 96 bytes for 768 bits

The C stack is used implicitly for executing the RCX subsystem. The RCX subsystem never accesses the C stack directly. The value stack is used to pass arguments to XS functions, to allocate local variables, and to protect objects from being garbage collected.

The heap is used to construct dynamic data structures. Only a single kind of cells can be allocated in the heap to allow efficient use of the small heap space. Every cell occupies two words (i.e., four bytes) and is aligned to a double-word boundary. Complex data structures are constructed by combining several cells. All unused cells are linked together to form a free-list. When a new cell is requested by the user program, one cell will be removed from the free-list and returned. If the free-list is empty, the user program will be suspended and a garbage collection will take place. The garbage collection uses a simple mark and sweep algorithm, which is very efficient for systems with a single kind of cells and no support for virtual memory. We use a separate table, the bit table, to remember the set of non-garbage cells.

[1]We have not measured the precise size of the on-memory RCX subsystem. The above number is actually the size of the binary file.

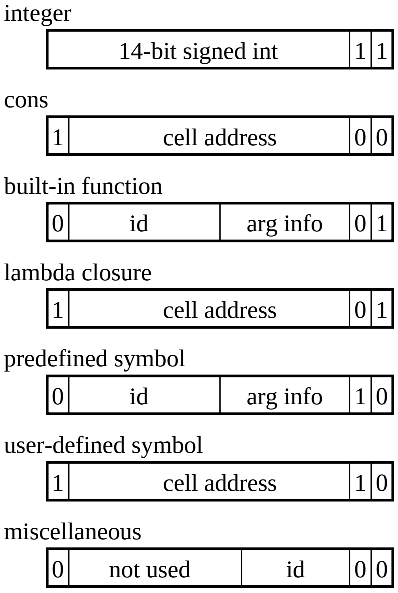

Figure 5: Object representation.

Each bit in the bit table corresponds to a cell and the total size of the table is only 96 bytes for the current configuration with 3K bytes of heap. It may be expected to have a real-time or incremental garbage collection in order to control robots. However, because of the small heap size and the relatively fast CPU of RCX, we have never experienced a trouble caused by the stop-the-world garbage collection.

6.1 Object representation

Figure 5 illustrates how objects are represented in XS. Every object is represented by a 16-bit word and the two least significant bits (LSBs) are used as tags to distinguish data types. The most significant bit (MSB) of an object other than integers is also used as a tag. The MSB tag is 1 for heap-allocated objects and is 0 for immediate data. The LSB tag values are arranged so that all functions have the same tag 01 and all symbols have the same tag 10.

The RAM memory of RCX is placed in the higher part of the 16-bit address space and thus the MSB of a cell address is always 1. Since cells are aligned to double-word boundaries, the LSBs of a cell address is always 00. Thus the representation of a cons object is nothing more than the raw pointer to a cell. ("cell address" in the Figure means a cell address without the MSB and the two LSBs.) This simplifies operations for list processing. For instance, the function `car` regards its argument as a cell pointer and will simply return the object that is pointed to by the pointer (see Figure 6 (a)). In contrast, for lambda closures and user-defined symbols, the pointer to a cell is obtained by clearing the two LSBs.

The representation of a built-in function contains an *id* and an *arg info*. The *id* is a number that uniquely identifies the function. The 6-bit *arg info* consists of the number of required arguments (2 bits), the maximum number of acceptable arguments (3 bits), and a flag indicating whether the function can accept arbitrarily large number of arguments (1

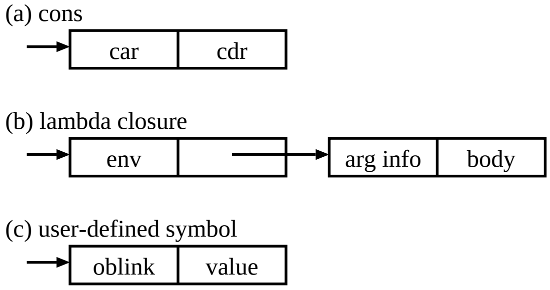

Figure 6: Interpretation of cells.

bit). When the evaluator invokes a built-in function, it first checks the number of arguments according to the *arg info*, and then jumps to the implementation code of the function by dispatching according to the *id*.

Each predefined symbol has an associated built-in function as the value of the symbol. As in Scheme, there is no predefined symbol whose symbol value is undefined or whose value is an object that is not a built-in functions. In addition, predefined symbols in XS are immutable, i.e., assignment is not allowed for the global variables named by predefined symbols. Because of the one-to-one correspondence between predefined symbols and built-in functions, a predefined symbol has the same representation as the associated built-in function except for the LSB tag. The value of a predefined symbol can be obtained simply by converting the LSB tag from 10 to 01.

6.2 Symbols

A user-defined symbol is represented by a cell pointer with the LSBs tagged with 10. The pointed cell is interpreted as illustrated in Figure 6 (c). The *oblink* field is used to link together all user-defined symbols in order to protect them from garbage collection. In the current implementation, a user-defined symbol once created will never be garbage-collected. The *value* field contains the symbol value, i.e., the value of the global variable that is named by the symbol. If the value is not defined, the *value* field contains an "undefined object", which is represented as a miscellaneous object. Since a symbol object is tagged with 10 in its LSBs, it can be regarded as a pointer to the second field of the cell. Thus it can be used, without modification, to access the *value* field.

Symbol names are maintained in the front-end. The front-end has a hash table, called **symbol table**, for mapping symbol names to symbol objects (both predefined and user-defined). When the reader of the front-end encounters a symbol token, it looks up the symbol table to convert the token into a symbol object. If no symbol is registered under the symbol name, the reader sends a request to the RCX subsystem to create a new symbol object. The RCX subsystem allocates a new cell, links it to the oblink, initializes the *value* field with the undefined object, and sends back the symbol object. The reader then registers the symbol object into the symbol table. This process of "interning" is not necessary for symbol tokens that are used only for local variables, because references to local variables are converted to references to locations in the value stack by the preprocessor of the front-end (see below).

In order to print out a symbol, the front-end has a hash table that maps user-defined symbols to symbol names. Names of predefined symbols are stored in another table indexed with function *ids*.

6.3 Lambda closures

The front-end preprocessor converts a lambda expression

> `(lambda` *parameters* `.` *original-body*`)`

into

> `(lambda` *arg-info* `.` *body*`)`

where *arg-info* is similar to that stored in build-in function objects and *body* is the result of preprocessing *original-body*. When the evaluator creates a new closure for the lambda expression, it simply replaces the first cell with a new cell, and stores the lexical environment in the car part of the new cell (see Figure 6 (b)). The rest of the lambda expression can be shared by all closures made from the lambda expression. Therefore, creation of a new closure consumes only one cell, though construction of the lexical environment may consume some more cells as explained below.

Local variables (including parameters) are allocated on the value stack. The preprocessor replaces local variable references with references to value stack locations relative to the base address of a function frame. When the evaluator invokes a lambda closure, it first pushes the lambda closure onto the value stack and then pushes the arguments. Then the evaluator starts executing the body. The other local variables (those that are specified by binding forms such as `let`) will be also allocated in the value stack. When a binding form is evaluated, the initial values of the local variables are pushed onto the stack. They are popped off the stack when the binding form is exited. Local variables are referenced by their offsets from the base address of the function frame. At the base address is the lambda closure and the i-th parameter is at the word offset i.

Lexical environments are constructed by linking heap cells. When a lambda closure is invoked, the initial lexical environment is the one stored in the closure. When a local variable is to be allocated on the stack, if it may be enclosed in some closures, it will be "lifted" into the heap. That is, a new cell is allocated and the car part of the cell is used to store the value of the local variable. The initial value of the local variable is moved to the car part of the cell and a pointer to the cell is stored at the stack location of the variable for quick reference. The cdr part of the cell is linked to the lexical environment at that time.

The preprocessor of the front-end performs all static analysis necessary to implement the above scheme, and passes appropriate information to the RCX evaluator by embedding in S-expressions. First, the preprocessor inserts lifting commands at appropriate places in S-expressions. The operands of a lifting command are the location of the variable to lift and the location of the lexical environment at the time of lifting. Second, the preprocessor replaces each reference to a local variable with one of the three commands: to reference a stack location, to reference the car part of the cell that is pointed to from a stack location, and to reference an element in the list that is stored in the closure as the lexical environment. The operand of the first two commands is a stack location and the operand of the last command is an index to the list. Third, the preprocessor embeds the location of the lexical environment to be saved in lambda closures, in the symbol `lambda` of each lambda expression. Since all static information is prepared by the preprocessor, only simple operations are left to be done at run time by the evaluator,

6.4 Communication

The front-end sends each (preprocessed) S-expression to the RCX, as a byte sequence. The receiver in RCX restores the S-expression from the byte sequence, by using cons cells in the RCX heap. XS uses post-fix notations for byte sequences. For instance, the list "`(1 2)`" will be transmitted as the following sequence of five bytes.

> low(`1`), high(`1`), low(`2`), high(`2`), `list2`

where low(x) and high(x) are the lower byte and higher byte, respectively, of the object representation of x. `list2` is a command to construct a list with the preceding two objects. By looking at the first byte, the receiver recognizes it is the lower byte of an integer because its LSB tag is 11. So, the receiver assumes the second byte is the higher byte of an integer, restores the integer `1`, and pushes it onto the value stack. Similarly, the receiver restores the integer `2` from the third and fourth bytes and pushes it onto the stack. Since the fifth byte is the command `list2`, the receiver pops out two objects from the stack, constructs a list with these objects, and then pushes the list onto the stack for further processing.

Commands in byte sequences are represented as miscellaneous objects (recall Figure 5). The higher bytes of miscellaneous objects are not used except for the MSB, which is used to distinguish from cons objects. Cons objects are never sent in byte sequences, because they are to be created by the receiver. Thus, by sending only the lower byte, the receiver can recognize it is a miscellaneous object. Depending on the *id*, the object may be an ordinary object such as the empty list and the boolean values, or it may be a command.

Some commands direct the receiver to invoke the evaluator. For example, the definition "`(define bar (foo 1))`" will be passed as the following 9-byte sequence.

> low(`foo`), high(`foo`), low(`1`), high(`1`), `list2`, `eval`,
> low(`bar`), high(`bar`), `define`

where low(`foo`) is the lower byte of the symbol object for the symbol `foo` (remember that symbols are interned in advance by the front-end reader). The receiver first constructs a list "`(foo 1)`" from the first five bytes and pushes it onto the stack. By the `eval` command, the receiver pops out the list, invokes the evaluator to evaluate the list as an S-expression, and then pushes the result onto the stack. The receiver then pushes the symbol object for `bar`. By the command `define`, the receiver pops out the variable symbol and then the initial value, and defines the variable.

Since the capacity of the I/O buffer in RCX is limited (256 bytes in the current configuration), a complex S-expression may not be sent at once. If this is the case, the S-expression will be sent in multiple byte sequences, one sequence at a time. Each such sequence ends with a special command `cont` which indicates more sequences follow.

The RCX subsystem returns values to the front-end in a similar way. The only difference is that returned values may contain cycles, in which case simple encoding would cause infinite communication. To avoid this situation, the RCX subsystem stops transmitting a value after sufficiently large number of bytes have been transmitted. The upper limit is computed from the total number of cells in the heap.

7. CURRENT STATUS

We have finished the preliminary implementation of XS. The front-end subsystem of XS is running on Linux (Redhat and Debian) and Cygwin platforms. The RCX subsystem is running on RCX models 1.0 and 1.5. Some students at Kyoto University have been using XS for studying Lisp/Scheme while enjoying robot construction. Their comments have been reflected to the design of XS. For instance, the `time` function and the `with-watcher` construct were added in response to their requests.

One known problem of XS is that user programs are not maintained in the RAM after XS is exited. RCX has the ability to maintain user programs in the RAM even after the power is turned off. In the case of XS, the RCX subsystem is a user program for RCX, and is maintained after power off. The user need not download the RCX subsystem each time he turns on the RCX. However, for RCX, user programs of XS are data and RCX does not maintain them. Thus the user has to download his XS program each time he turns-on the RCX. This means he always has to carry a PC to run his program. This problem seems hard to solve. With the current configuration of the XS system, the only possible solution would be to modify the operating system.

We would like to have the XS system run on Windows platforms and support RCX model 2.0. Unfortunately, legOS version 0.2.4, on which the current implementation is based, does not support the LNP protocol for Windows and for USB communication. Although the latest version of legOS supports USB communication on Windows platforms, it occupies much more RAM space than version 0.2.4. The latest version may be small enough for small applications, but seems too large to run the RCX subsystem of XS. We are trying to solve this problem by removing those components that are not necessary to run XS.

The XS system is not yet publicly available. We plan to distribute it as an open-source software.

8. REFERENCES

[1] Dave Baum: *Extreme Mindstorms: an Advanced Guide to Lego Mindstorms*, Apress (2000).

[2] Dave Baum: *Definitive Guide to LEGO Mindstorms Second Edition*, Apress (2003).

[3] brickOS at SourceForge, `http://brickos.sourceforge.net/`.

[4] IEEE Standard for the Scheme Programming Language, IEEE (1991).

[5] Frank Klassner, Scott Anderson: Lego Mindstorms: Not Just for K-12 Anymore, IEEE Robotics and Automation Magazine (2003).

[6] LEGO.com Mindstorms Home, `http://mindstorms.lego.com/`.

[7] Stig Nielsson: Introduction to the legOS kernel, `http://legos.sourceforge.net/docs/kerneldoc.pdf` (2000).

[8] Kekoa Proudfoot: RCX Internals, `http://graphics.stanford.edu/~kekoa/rcx/` (1998).

[9] Guy Steele: *Common Lisp the Language, Second Edition*, Digital Press (1990).

[10] Luis Villa: legOS HOWTO, `http://legos.sourceforge.net/HOWTO/` (2000).

[11] Adam Wick, Kasey Klipsch, Mitchell Wagner: Lego/Scheme Compiler V0.5.2, `http://www.cs.indiana.edu/~mtwagner/legoscheme/`.

[12] Dario Laverde, Giulio Ferrari, Jurgen Stuber: *Programming Lego Mindstorms with Java*, Syngress (2002).

APPENDIX

A. LIST OF XS FUNCTIONS

This appendix gives a list of all functions that XS supports, together with their parameter profiles. We use the following notations.

X*: zero or more X's
X^+: one or more X's
$\{X_1|\cdots|X_n\}$: one of $X_1, \ldots, X_n$
[X]: optional X

When the parameter profiles of functions $X_1, X_2, \ldots, X_n$ are the same except for their function names, we sometimes write the profile of X_1, followed by the names of $X_2, \ldots, X_n$. For instance, the profile of `or` is `(or` *expr**`)`.

A.1 Common functions

- top-level forms
  `(define` *sym expr*`)`
  `(define (`*sym sym** `[.` *sym*`])` *expr**`)`
  `(load` *file-name*`)`
  `(trace` *function-name*`)` and `untrace`
  `(bye)`

- basic forms
  (quote object)
  (set! sym expr)
  (lambda (sym* [. sym]) expr*)
- control
  (begin expr*)
  (apply function object* list)
  (if expr expr [expr])
  (catch expr expr*)
  (throw object object)
- conditional
  (and expr*) and or
  (not object)
- binding
  (let [sym] ((sym expr)*) expr*)
  (let* ((sym expr)*) expr*) and letrec
- type predicates
  (boolean? object) and integer?, null?, pair?, symbol?, function?
- comparison
  (eq? object object)
  (< int+) and >, =, >=, <=
- arithmetic
  (+ int*)
  (- int int*)
  (* int*)
  (/ int int) and remainder
  (logand int int) and logior, logxor, logshl, logshr
  (random int)
- list processing
  (car pair) and cdr
  (cons object object)
  (set-car! pair object) and set-cdr!
  (list object*)
  (list* object* object)
  (list-ref list int)
  (append [list* object])
  (assoc object a-list)
  (member object list)
  (length list) and reverse
- I/O
  (read) and read-char, read-line
  (write object)
  (write-char char)
  (write-string string)
- garbage collection
  (gc)

A.2 Lego-specific functions

- top-level forms
  (last-value)
  (ping)
- control
  (sleep int) ; in 1/10 seconds
  (time) ; in 1/10 seconds
  (reset-time) (wait-until expr)
  (with-watcher ((expr expr*)*) expr*)
- light sensors
  (light-on {1|2|3}) and light-off
  (light {1|2|3}) ; 0 (black) to 98 (white)
- rotation sensors
  (rotation-on {1|2|3}) and rotation-off
  (rotation {1|2|3}) ; in 360/16 degrees
- temperature sensors
  (temperature {1|2|3}) ; in Celsius
- touch sensors
  (touched? {1|2|3})
- motors
  (motor {:a|:b|:c}
    {:off|:forward|:back|:brake})
  (speed {:a|:b|:c} speed) ; 0 to 255
- sounds
  (play ((pitch . length)*))
  (playing?)
- Prgm button
  (pressed?)
- LCD display
  (puts string)
  (putc char column)
  (cls)
- battery level
  (battery) ; in 1/10 volts