XS Reference Manual

Taiichi Yuasa

November 2, 2006

Contents

1 Introduction

XS is a programming system for the Lego MindStorms Robotics Invention System (RIS). The central component of RIS is a programmable block called RCX, with an 8-bit CPU. RCX programs are prepared on the front-end PC and downloaded to the RCX through IR communication. By attaching motors, sensors, and other component blocks, one can build robots that are controlled by user-supplied RCX programs.

XS has been developed to provide an interactive and satisfactory programming environment for RIS. It supports programming in a Lisp language with extensions for controlling RCX devices. Specifically it 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 RCX 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 just like an ordinary Lisp system with extensions for controlling RCX devices.

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

Figure 1 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` to be the empty list (line 10), and tests the function again (line 12). This time the function returns a correct answer (line 13) and

```
 1  % xs
 2  Welcome to XS: Lisp on Lego MindStorms
 3
 4  >(define (ints n)
 5     (if (= n 0) nil (cons n (ints (- n 1)))))
 6  ints
 7  >(ints 3)
 8  Error: undefined variable -- nil
 9  Backtrace: ints > ints > ints
10  >(define nil ())
11  nil
12  >(ints 3)
13  (3 2 1)
14  >(bye)
15  sayonara
16  %
```

Figure 1: A sample XS session (line numbers are added for explanation).

the satisfied user ends the XS session (line 14). The user sees nothing special with this sample session. Internally, however, things are quite 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 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 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 name symbol of 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. In case 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 on the RCX brick. 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 means 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.

2 Notations and Terminology

The description of each language element of XS starts with a "header" which is in one of the following formats.

syntax	[Top-level Form]
syntax	[Special Form]
(*function-name argument-specification*)	[Function]
constant-name	[Reader Constant]

The first two formats are used for top-level forms and special forms, respectively. These forms provide syntactic constructs of XS such as those for conditional branches and variable bindings, and the full syntax of each construct is given in the header. An error will be signaled if the user gives a top-level form or a special form that does not obey its syntax.

The third format is used for built-in functions. Each header in this format starts with the name of the built-in function, followed by the specification of acceptable number and classes of arguments (see below). The general rule is that the function behaves as described, only when the supplied arguments satisfy the specification in the header. Otherwise, an error will be signaled. Exceptions to this rule, if any, are explicitly mentioned in the description of the function. The last format is used to start descriptions of reader constants. Each header in this format gives only the name of the constant.

The following syntax notations are used throughout this document.

Syntactic variables (i.e., non-terminal symbols) are written in *italic*.
Terminal symbols are written in `type-face`.
"*thing**" denotes zero or more occurrences of *thing*.
"[*thing*]" denotes at most one occurrence of *thing*.

where *thing* is a syntactic variable or a syntactic pattern that begins with "`(`" and end with the matching "`)`".

The following notations are used in argument specification in the headers of the built-in functions.

1. Argument classes are written in *italic*. They specify acceptable classes of arguments. In addition, they are used to denote arguments in the function descriptions. Only the followings, sometimes with subscripts, are used as argument classes.

   obj for arbitrary XS objects
   int for integers
   sym for symbols
   cons for conses
   list for proper lists
   fun for functions

2. "$class_1$ ... $class_n$" denotes n arguments of the *class*. The acceptable number of such arguments is given in terms of a condition on n (either "$n \geq 0$" or "$n \geq 1$") in the header.

3 Objects and Lexical Structure

XS supports the following six classes of objects. Each object in XS belongs to exactly one of the classes.

booleans
integers
symbols
the empty list
conses
functions

In addition, XS supports the following "pseudo object" to facilitate program coding.

characters
strings

3.1 Booleans

Boolean objects are `#t` and `#f`. These are used when a function returns a truth value: `#t` for true and `#f` for false.

3.2 Integers

Integers are written in binary, in octal, in decimal, or in hexadecimal, respectively in the following formats.

`#b`[*sign*]*binary-digits*
`#o`[*sign*]*octal-digits*
[`#d`][*sign*]*decimal-digits*
`#x`[*sign*]*hexadecimal-digits*

where

a *sign* is either "`+`" or "`-`",
a *binary-digit* is either `0` or `1`,
an *octal-digit* is one of `0`, `1`, ..., `7`,
a *decimal-digit* is one of `0`, `1`, ..., `9`, and
a *hexadecimal-digit* is one of `0`, `1`, ..., `9`, `a`, ..., `f`.

For each format, at least one digit is necessary.

Integer objects are signed and their precision is 14 bits. Thus XS supports integers in the range from -2^{13} $(= -8192)$ to $2^{13} - 1$ $(= 8191)$.

`:most-positive-integer` [Reader Constant]
`:most-negative-integer` [Reader Constant]

The values of these reader constants are `8191` $(= 2^{13} - 1)$ and `-8192` $(= -2^{13})$, respectively.

3.3 Symbols

Symbols are objects that can be uniquely identified by their *names* (or *symbol names*). The name of a symbol is a sequence of one or more characters.

A Symbol is denoted by its name if the symbol name satisfies all the following conditions.

1. The symbol name consists of the following characters.

```
! # $ % & * + - . / 0 1 2 3 4 5 6 7 8 9 : < = > ? @
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z [ ] ^ _
a b c d e f g h i j k l m n o p q r s t u v w x y z { } ~
```

2. The symbol name does not begin with "`:`" nor "`#`".

3. The symbol name is not the one that consists of a single dot "`.`".

4. The symbol name does not match the syntactic format of integers.

If the symbol name does not satisfy one or more conditions above, then denote the symbol by enclosing the symbol name with vertical bars "`|`". If the symbol name contains a vertical bar, then precede it with a backslash "`\`". If the symbol name contains a backslash, then precede it with another backslash.

Symbol names are case-sensitive. For instance, `baz` and `BAZ` are different symbols.

Names of built-in functions are *built-in symbols*. Other symbols are *user-defined symbols*. When the XS system is started, only built-in symbols exist in the system. User-defined symbols will be created later. For example, when the user defines a new global variable with the top-level form `define`, a new symbol that names the variable will be created.

Unlike Common Lisp, the symbol name of "`NIL`" does not automatically represent the empty list nor the false value.

3.4 Conses and Lists

Conses are objects mainly used to construct data structures such as lists and trees. A cons object has two objects called the `car` and the `cdr` of the cons object. A cons object is written as

`(`x_1 `.` x_2`)`

where x_1 and x_2 denote the `car` and `cdr`, respectively, of the cons object.

The empty list represents a list with no elements and is written as `()`. There is only one empty list in the XS system and thus `()` always denotes the same object.

Data structures whose `cdr` link ends with the empty list are called *lists*. More precisely, lists are defined recursively as follows.

1. The empty list is a list.
2. A cons object is a list if the object in the `cdr` part is a list.
3. These and only these objects are lists.

Lists are written as

`(`x_1 x_2 `...` x_n`)`

which is equivalent to

`(`x_1 `. (`x_2 `. ( ... . (`x_n `. ()) ... )))`

Here, n is the *length* of the list, and each x_i $(1 \leq i \leq n)$ is called the i^{th} *element* of the list. If n is zero, then the list is the empty list `()`.

Data structures whose `cdr` link ends with an object other than the empty list are called *dotted lists*. Dotted lists are written as

`(`x_1 x_2 `...` x_{n-1} `.` x_n`)`

which is equivalent to

`(`x_1 `. (`x_2 `. ( ... . (`x_{n-1} `.` x_n`) ... )))`

3.5 Functions

Function objects can *receive* some objects as *arguments* and *returns* an object as the *value*. An operation to send objects to a function and to get its value is called *call*. Only function objects can be called. Symbols and lambda lists (i.e., lists whose first element is the symbol `lambda`) are not function objects, and thus cannot be called.

XS provides various *built-in functions*. All built-in functions are described in this manual. Some built-in functions are *installation-time options*. These functions can be defined by using other built-in functions. Therefore, they may not be installed in your XS system[1], to save the memory space in the RCX brick. If not installed, load the file `lib.lsp` before you use them. This file contains definitions of all built-in functions that are installation-time options, and can be found in the directory where your XS system is stored.

You can define your own functions. Such *user-defined functions* can be created only by `lambda` forms.

3.6 Characters

Character objects represent those characters that the computer can handle. In XS, characters are not first-class objects. When the XS reader encounters a textual representation of a character, the character is converted to its ASCII code.

Ordinary character objects are written as

`#\`*char*

where *char* is the character that the character object represents. XS supports the following characters.

```
! " # $ % & ' ( ) * + , - . /
0 1 2 3 4 5 6 7 8 9 : ; < = > ? @
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
[ \ ] ^ _ `
a b c d e f g h i j k l m n o p q r s t u v w x y z
{ | } ~
```

Some "special characters" are written as

`#\`*name*

where *name* is the "name" of the special character. XS supports the following special characters.

[1] In the current distribution package, installation-options are not defined in the RCX subsystem. Therefore, you have to load the file `lib.lsp` before you use them.

```
#\tab
#\space
#\newline
#\return
#\page
```

3.7 Strings

Strings are ordered sequences of characters. Any character that the implementation supports can be a string element. Strings are written by listing all the element characters in order, enclosed with double quotes "`"`". If the string has a double quote as its element, the double quote must be preceded by a backslash "\". If the string has a backslash as its element, the backslash must be preceded by another backslash.

In XS, strings are not first-class objects. When the XS reader encounters a textual representation of a string, the string is converted to a list of ASCII codes.

3.8 Reader Constants

Reader constants look like symbols but they are converted to integers when they are read by the XS reader. Reader constants are built-in in the XS system, and the user cannot define his or her own reader constants. Each reader constant has a name that begins with a colon "`:`", and is converted to a fixed value.

3.9 Comments

If an input line contains a semi-colon "`;`" which is not part of a textual representation of an object, then the semi-colon and the following characters in the line are regarded as a comment.

4 Evaluation

4.1 Forms

Forms are objects that can be evaluated. Some forms can appear only at the top-level of XS programs. These forms are called *top-level forms*. Only those lists that begin with one of the following symbols are top-level forms [2].

```
define      load     fork
trace       untrace
last-value  bye
```

[2]Currently, the top-level form `fork` is available only on the Linux version of XS.

In the rest of this document, forms other than top-level forms are called *ordinary forms* or, if there is no possibility of confusion, simply called *forms*.

The followings are (ordinary) forms in XS.

1. special forms
2. function-call forms
3. variables
4. literals

A form, when evaluated, returns only one object as the value, if it ever returns.

Special Forms

Special forms are those lists whose first element is one of the following symbols.

```
and         or
quote       lambda         set!
let         let*           letrec
begin       if             catch
wait-until  with-watcher
```

These symbols are called *special form names*. Evaluation of special forms depends on the symbol.

Function-call Forms

Function-call forms are those lists whose first element is not a special form name. When a function-call form is evaluated, the first element of the form is evaluated first. The value must be a function object. Then the elements of the list form except the first are evaluated one after another, from left to right. The values of these forms are then passed to the function as the arguments. The value that the function returns is returned as the value of the function-call form.

The XS system is made *proper tail-recursive*. A function call is called *tail-recursive* if the caller simply returns the value of the callee, without doing anything else. For example,

```
(define (ints n)
  (if (= n 0) () (cons n (ints (- n 1)))))
```

the call of `ints` in the body is not tail-recursive because the caller (which happens to be the same as the callee in this example) has to invoke `cons` after the return of `ints`. On the other hand, the call of `ints` in the following definition is tail-recursive.

```
(define (ints n sofar)
  (if (= n 0) sofar (ints (- n 1) (cons n sofar))))
```

A proper tail-recursive system is a system which replaces all tail-recursive calls with appropriate jumps so that tail-recursive calls do not consume the stack space.

Variables

Variables are denoted (or named) by symbols. Every variable in XS has exactly one object as its *value*, at any time during the life time.

When a symbol is evaluated as a form, it returns the value of the variable it names. An error will be signaled if the symbol names no variable.

Like Scheme, XS uses a single name space for variables and functions. Each named function is stored as the value of the variable with the same name as the function name. For instance, the function object of the built-in function `cons` is stored as the value of the variable named `cons`. The form `(cons 1 2)` invokes the built-in function `cons` because the system first evaluates the variable `cons` and obtains the built-in function object. You can see this by typing the name of the built-in function.

```
>cons
#<function cons>
```

The value of a user-defined variable can be changed by a `set!` form. However, variables for built-in functions are *immutable*, i.e., their values cannot be changed.

Literals

Objects other than conses (including lists) and symbols are literals. When evaluated, a literal returns itself as the value.

As mentioned before, strings in XS are not first-class objects and are converted to lists of integers by the reader. Since strings are literals in ordinary Lisp systems which support first-class strings, we would like to handle strings as literals in XS programs as well. For instance, we would like to write

```
(define x "abc")
```

rather than

```
(define x '"abc")
```

which looks strange to most Lisp programmers. For this purpose, the reader encloses a string into a `quote` form when the string appears at a position of an expression. Thus, both `define` forms above are converted to:

```
(define x '(97 98 99))
```

4.2 Environments

Evaluation of a form depends on the *evaluation environment* (or an environment, for short) at the time of evaluation. The environment consists of three kinds of bindings.

- variable bindings
  A variable binding is a pair consisting of a variable and its name. The name of the variable is a symbol.
- catcher bindings
  A catcher binding is a pair consisting of a catcher and its name. The name of a catcher can be any object. A catcher is created by the `catch` form and is used for dynamic non-local exit.
- watcher bindings
  A watcher binding is a pair consisting of a condition and a handler. The condition and the handler are forms. A watcher is created by the `with-watcher` form and is used for asynchronous event handling.

There may exist several evaluation environments at the same time during execution of an XS program, but only one of them, called the *current environment*, is used to evaluate a form.

Initially, the current environment consists only of the bindings for built-in variables. A new current environment is generated by adding (or *establishing*) a binding to the current environment. The old environment becomes the current environment by removing (or *unestablishing*) the binding from the new current environment. When an ordinary form (a form that is not a top-level form) establishes a binding, the binding is automatically unestablished on exit from the form. On the other hand, when the top-level form `define` establishes a binding, that binding will remain established during the current XS session. The bindings for built-in variables and those established by the `define` form are collectively called *global variable bindings.* There are no global bindings for catchers and watchers.

When a function object is created, part of the current environment is saved in the function object. That part of the environment is called the *lexical environment*, and the rest of the environment is called the *dynamic environment.* When that function is invoked, an environment is formed consisting of the current dynamic environment (i.e., the dynamic environment of the current environment) and the lexical environment in the function object. This environment becomes the current environment for the function call. When the function returns, the current environment before the call becomes the current environment again. A lexical environment consists of *lexical bindings*, which are variable bindings established by ordinary (i.e., non-top-level) forms. Other bindings are called *dynamic binding.*

The following rules summarize whether a binding is lexical or dynamic.

- Bindings for built-in variables are global and thus dynamic.
- Bindings that are established by the `define` form are global and thus dynamic.
- Other Variable bindings are lexical.
- Catcher bindings and watcher bindings are dynamic.

The current environment is used to evaluate a form in the following ways.

- When a symbol s is used as a variable name, the current lexical environment is searched for a variable binding whose name is the same as s. If there are more than one such bindings, the one that was established last is used. If there is not such binding, the dynamic environment is searched. An error will be signaled if there is no such binding. The variable of the selected variable binding is used as the variable that the symbol s names.
- When an object x is specified as a catcher name in the `throw` form, a catcher binding whose name is the same (in the sense of `eq?`) as x will be retrieved from the current environment. If there are more than one such catcher bindings, the one that was established last is retrieved. The `throw` form returns from that catcher. An error will be signaled if there is no such binding.
- The XS system periodically (approximately every 100 msec) checks the watcher environment to see if there is a watcher binding whose condition is satisfied. If there is, the system suspends the current program execution and starts executing the handler of the watcher. If there are more than one such watcher bindings, the one that was established last will be selected. When execution of the handler is finished, the suspended execution will be resumed.

4.3 Lambda Expressions

Functions are defined by *lambda expressions.* A lambda expression is a list that begins with the symbol `lambda`.

> `(lambda` *lambda-list form**`)`

where the sequence of the *forms* are called the *body* of the function. Lambda expressions are special forms and thus can be evaluated. When evaluated, a lambda expression creates a new function object, in which the current environment is saved.

The *lambda-list* of a lambda expression specifies the parameters that the function can receive. The general format of a *lambda-list* is the following.

> `(`*sym** [. *sym*] `)`

When the function is invoked, the lambda-list establishes a variable binding for each *sym*. These bindings are unestablished on return from the function. Variables that are created by a lambda-list are called *parameters* according to the tradition of programming languages. The initial values of the parameters are determined by the arguments that are supplied to the function.

1. The symbols except the one after the dot "`.`" are names of *required parameters*. When the function is invoked, arguments to these parameters must be given. That is, when n required parameters are specified in the lambda list, then at least n arguments must be supplied. The i^{th} argument ($1 \leq i \leq n$) becomes the value of the i^{th} required parameter. An error will be signaled if too few arguments are supplied and there is no argument that corresponds to a required parameter.

2. The symbol after the dot "`.`" if any is the name of the *rest parameter*. All non-required arguments to the function are linked together as a list, and the list becomes the value of the rest parameter. The $(n+i)^{th}$ argument becomes the i^{th} element of the list, where n is the number of the required parameters. An error will be signaled if there remains an argument that does not correspond to a required parameter, when the rest parameter is not specified in the lambda list.

Each parameter name must not be the name of a top-level form or a special form, but can be the name of a built-in function.

The body of a function is a sequence of zero or more forms. When the function is invoked, the forms in the body are evaluated from left to right, after the parameter bindings are established. Then the value of the last form is returned as the value of the function. If the body has no forms, then the function returns the empty list `()`. In this document, we sometimes use the notation "`.` *body*" in place of "*form**", and say "evaluates the *body* and returns the value" to mean:

> "Evaluates the *forms* from left to right. Discards all the values of the *forms* except for the value of the last *form* (if any), which is then returned as the value of the *body*. If no *form* is supplied, the *body* simply returns `()`."

With this notation, a lambda expression is written as a list in the following format.

(`lambda` *lambda-list* . *body*)

5 Top-level forms

(`define` *sym* *form*) [Top-level Form]

Evaluates *form* and establishes a global variable binding named *sym* whose initial value is identical to the value of *form*. The value of the named variable may be modified by `set!`. The symbol *sym* must not be a top-level form name, a special form name, nor a built-in function name.

`(define (`*sym* `.` *lambda-list*`)` `.` *body*`)` [Top-level Form]

Creates a function defined by the lambda expression

> `(lambda` *lambda-list* `.` *body*`)`

and establishes a global binding named *sym* for the created function. This form is equivalent to:

> `(define` *sym* `(lambda` *lambda-list* `.` *body*`))`

The symbol *sym* must not be a top-level form name, a special form name, nor a built-in function name.

`(load` *string*`)` [Top-level Form]

Loads a program from the file named *string*. The contents of the file must be a sequence of top-level expressions. The effect of loading from the file is the same as supplying the top-level expressions in the file from the PC keyboard.

`(fork` sym_1 sym_2 $string_1$ $string_2$ $\ldots$ $string_n$`)` $(n \geq 1)$ [Top-level Form]

Creates a subprocess of the front-end subsystem and executes the program that is specified by the path $string_1$. The other strings (i.e., $string_2, \ldots, string_n$) are passed to the program as arguments. This top-level form establishes global bindings named sym_1 and sym_2 for the port numbers to write to the standard input and to read from the standard output, respectively, of the forked subprocess. These port numbers can be used to communicate with the subprocess by using the I/O functions of XS, such as `read` and `write`.

This top-level form is currently available only on the Linux version of XS.

`(trace` *sym*`)` [Top-level Form]

Starts tracing the function named *sym*. Only user-defined functions can be traced. When a traced function is invoked, its arguments and return value will be displayed together with the function name. This top-level form returns the function name *sym*. Tracing of a function can be canceled by the top-level form `untrace`.

`(untrace` *sym*`)` [Top-level Form]

Cancels tracing of the function named *sym*. Returns the function name *sym*, if the named function is a traced function. Otherwise, returns `#f`.

`(last-value)` [Top-level Form]

Returns the value of the last top-level expression. This top-level form is useful when the front-end failed to receive the last value from the RCX for some reasons. If the evaluation of the last top-level expression ends with an error, the error message and backtrace will be displayed on the PC display.

`(bye)` [Top-level Form]

Ends the current XS session. The evaluator in the RCX will be stopped. So, be sure to restart the evaluator by pressing the Run button when you start another XS session.

6 Predicates

6.1 Class Predicates

`(boolean?` *obj*`)` [Function]

Returns `#t` if *obj* is a boolean object. Otherwise, returns `#f`.

`(integer?` *obj*`)` [Function]

Returns `#t` if *obj* is an integer object. Otherwise, returns `#f`.

`(null?` *obj*`)` [Function]

Returns `#t` if *obj* is the empty list. Otherwise, returns `#f`.

`(pair?` *obj*`)` [Function]

Returns `#t` if *obj* is a cons object. Otherwise, returns `#f`.

`(symbol?` *obj*`)` [Function]

Returns `#t` if *obj* is a symbol object. Otherwise, returns `#f`.

`(function?` *obj*`)` [Function]

Returns `#t` if *obj* is a function object. Otherwise, returns `#f`.

6.2 Logical Operators

`(not` *obj*`)` [Function]

Returns `#t` if *obj* is `#f`. Otherwise, returns `#f`.

`(and` *form** `)` [Special Form]

Evaluates the *forms* from left to right until one of them evaluates to `#f`, in which case the `and` form returns `#f`. If none of the *forms* evaluates to `#f`, then returns the value of the last *form*. If no *form* is supplied, then the `and` form simply returns `#t`.

`(or` *form** `)` [Special Form]

Evaluates the *forms* from left to right until one of them evaluates to non-`#f`, in which case the `or` form returns the non-`#f` value. If all of the *forms* evaluates to `#f`, then returns `#f`. If no *form* is supplied, then the `or` form simply returns `#f`.

6.3 Equality

`(eq?` obj_1 obj_2`)` [Function]

Returns `#t` if obj_1 and obj_2 are the same (or identical) object. Otherwise, returns `#f`.

XS adopts the following rules, which are used to determine whether two objects are identical.

- Each of `()`, `#t`, and `#f` denotes a unique object.
- Integer objects are identical iff they represent the same integer value.
- Symbols are identical iff their names are the same.
- Built-in functions are identical iff their names are the same.
- When a user-defined function is created by a lambda expression, it is not identical to any already existing object.
- When a cons object is read (by the read-eval-print loop or by the function `read`) or created (by functions such as `cons`), it is not identical to any already existing object.

XS guarantees that once a variable binding is established with the initial value x, or is set to x by `set!`, the value of the variable remains identical to x until the value is changed by `set!`.

Similarly, the value of the car (or cdr) field of a cons remains identical until it is replaced by `set-car!` (or `set-cdr!`). Thus the function `car` (or `cdr`) applied to the same cons object always returns the same value until the car (or cdr) field of the cons object is replaced by `set-car!` (or `set-cdr!`).

7 Variables and Functions

`(quote` *obj*`)` [Special Form]

> Simply returns *obj*. `(quote` *obj*`)` can be written as `'`*obj*.

`(set!` *sym form*`)` [Special Form]

> Evaluates the *form* and replaces the value of the variable named by *sym*, with the value of the *form*. The value of the *form* is returned as the value of the `set!` form.

`(lambda` *lambda-list* `.` *body*`)` [Special Form]

> Creates and returns a function defined by the lambda expression (see Section 4.3).

`(let` [*name*] `((`*sym form*`)*` `)` `.` *body*`)` [Special Form]

> Evaluates all *forms* first from left to right, then establishes, for each *sym*, a variable binding named by *sym* whose initial value is identical to the value of the corresponding *form*. Then evaluates the *body* and returns the value. The variable bindings will be unestablished on exit from the `let` form.
>
> The `let` form
>
> `(let ((`x_1 *form*$_1$`)` ... `(`x_n *form*$_n$`))` `.` *body*`)`
>
> is equivalent to
>
> `((lambda (`x_1 ... x_n`)` `.` *body*`)` *form*$_1$ ... *form*$_n$`)`
>
> When the optional *name*, which must be a symbol, is supplied, the `let` form first establishes a variable binding named *name* whose initial value is the lambda expression:
>
> `(lambda (`x_1 ... x_n`)` `.` *body*`)`
>
> Thus the `let` form
>
> `(let` *name* `((`x_1 *form*$_1$`)` ... `(`x_n *form*$_n$`))` `.` *body*`)`
>
> is equivalent to
>
> `(let ((`*name* `(lambda (`x_1 ... x_n`)` `.` *body*`)))`
> `(`*name* *form*$_1$ ... *form*$_n$`))`

`(let* ((`*sym form*`)*` `)` `.` *body*`)` [Special Form]

Evaluates the first *form* and establishes a variable binding named by the first *sym* whose initial value is identical to the value of the first *form*. Does the same thing for each pair of *sym* and *form*, from left to right. Then evaluates the *body* and returns the value. The variable bindings will be unestablished on exit from the `let*` form.

If no pair of *sym* and *form* is supplied, the `let*` form

`(let* () .` *body*`)`

is equivalent to

`(begin .` *body*`)`

Otherwise, the `let*` form

`((let* ((`x_1 $form_1$`) ... (`x_n $form_n$`)) .` *body*`)`

is equivalent to

```
(let ((x₁ form₁))
  (let* ((x₂ form₂) ... (xₙ formₙ)) . body))
```

`(letrec ((`*sym form*`)* ) .` *body*`)` [Special Form]

Establishes, for each *sym*, a variable binding named by *sym* whose initial value is `()`. Then evaluates all *forms* from left to right, saves each value in the variable named by the corresponding *sym*, evaluates the *body*, and returns the value. The variable bindings will be unestablished on exit from the `letrec` form.

The `letrec` form

`(letrec ((`x_1 $form_1$`) ... (`x_n $form_n$`)) .` *body*`)`

is equivalent to

```
(let ((x₁ ()) ... (xₙ ()))
  (set! x₁ form₁) ... (set! xₙ formₙ) . body)
```

The primary purpose of this special form is to define local recursive (either self-recursive or mutual-recursive) functions. If the *forms* are lambda expressions, they can call each other because the established bindings can be accessed in the function bodies.

`(apply` *fun* obj_1 ... obj_n *list*`)` $(n \geq 0)$ [Function]

Calls the specified function *fun* and returns whatever the function returns. Each obj_i $(1 \leq i \leq n)$ becomes the i^{th} argument to the function, and the j^{th} element of the last argument *list* becomes the $(n+j)^{th}$ argument to the function.

`(trace-call` *sym fun list*`)` [Function]

This function is used internally to implement traced functions specified by the top-level form `trace`. The user is recommended not to use this function explicitly.

This function calls the function *fun* and returns whatever the function returns. Elements in the *list* are used as the arguments to the function, with the i_{th} element being the i_{th} argument. In addition, `trace-call` displays the trace information of the call as if the function named *sym* were traced.

8 Control Structure

8.1 Simple Sequencing

`(begin` *form** `)` [Special Form]

Evaluates the *forms* from left to right. Discards all the values of the *form*s except for the value of the last *form* (if any), which is then returned as the value of the `begin` form. If no *form* is supplied, `begin` simply returns `()`.

8.2 Conditions

`(if` $form_1$ $form_2$ [$form_3$]`)` [Special Form]

Evaluates $form_1$. If the value of $form_1$ is not `#f`, then evaluates $form_2$ and returns its value. Otherwise, evaluates $form_3$ (which defaults to `()`) and returns the value.

8.3 Dynamic Non-local Exits

`(catch` *form* . *body*`)` [Special Form]

Evaluates the *form* and establishes a catcher binding named by the value of the *form*. Then evaluates the *body* and returns the value. Unestablishes the catcher binding on exit from the form. When the function `throw` is invoked with the first argument being the name of the established catcher, the evaluation of the `catch` form terminates immediately and the `catch` form returns the value of the second argument to the `throw` call.

`(throw` obj_1 obj_2`)` [Function]

If there exists a catcher named by obj_1, then returns from the `catch` form that established the catcher binding, with the value obj_2. An error will be signaled if there exists no catcher named by the obj_1.

8.4 Timing

`(sleep` *int*`)` [Function]

> Suspends program execution for the specified period of time. The *int*, which must be positive, specifies the time for suspension in 1/10 seconds. For example, the argument `5` suspends for 0.5 seconds, and the argument `50` suspends for 5 seconds. This function returns the *int* after the specified period of time.

`(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.

`(reset-time)` [Function]

> Reset the system clock to zero and returns the "current time". Since the system clock is reset, the return value is usually zero.

`(wait-until` *form*`)` [Special Form]

> Waits until the event specified by *expr* happens. The *form* 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. The `wait-until` returns whatever the *form* returns (which cannot be `#f`).

`(with-watcher ((`*form* *form*`*)* ) .` *body*`)` [Special Form]

> Establishes "watchers" that wait for specified events to occur during execution of the *body*. When an event occurs, the watcher suspends execution of the *body* and execute the associated handler. The general form is:

```
(with-watcher ((event1 . handler1)
                ...
               (eventn . handlern))
   . body)
```

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. As execution of all handlers is completed, the execution of the body resumes.

9 Integer Operations

9.1 Comparison

`(=` int_1 ... int_n`)` $(n \geq 1)$ [Function]

Returns `#t` if all *ints* are equal. Otherwise, returns `#f`.

`(<` int_1 ... int_n`)` $(n \geq 1)$ [Function]

Returns `#t` if each int_i $(1 \leq i < n)$ is less than int_{i+1}. Otherwise, returns `#f`.

`(<=` int_1 ... int_n`)` $(n \geq 1)$ [Function]

Returns `#t` if each int_i $(1 \leq i < n)$ is less than or equal to int_{i+1}. Otherwise, returns `#f`.

`(>` int_1 ... int_n`)` $(n \geq 1)$ [Function]

Returns `#t` if each int_i $(1 \leq i < n)$ is greater than int_{i+1}. Otherwise, returns `#f`.

`(>=` int_1 ... int_n`)` $(n \geq 1)$ [Function]

Returns `#t` if each int_i $(1 \leq i < n)$ is greater than or equal to int_{i+1}. Otherwise, returns `#f`.

9.2 Arithmetic

`(+` int_1 ... int_n`)` $(n \geq 0)$ [Function]

Returns the sum of the *ints*. Returns `0` if no argument is supplied.

XS does not perform overflow checking. If the result of addition does not fit the 14-bit representation of integer objects, then this function will return an unexpected value.

(`-` int_1 ... int_n) ($n \geq 1$) [Function]

> If only one argument is supplied, returns the negative of the argument. If multiple arguments are supplied, this function returns the result of subtracting $int_2, \ldots, int_n$ from int_1.
>
> XS does not perform overflow checking. If the result of negation or subtraction does not fit the 14-bit representation of integer objects, then this function will return an unexpected value.

(`*` int_1 ... int_n) ($n \geq 0$) [Function]

> Returns the product of the *ints*. Returns `1` if no argument is supplied.
>
> XS does not perform overflow checking. If the result of multiplication does not fit the 14-bit representation of integer objects, then this function will return an unexpected value.

(`/` int_1 int_2) [Function]

> Divides int_1 by int_2 and returns the quotient in integer. The second argument int_2 must not be `0`. See the description of `remainder` below for the precise return value.

(`remainder` int_1 int_2) [Function]

> Divides int_1 by int_2 and returns the remainder. The divider int_2 must not be `0`.
>
> The result is an integer whose absolute value is less than the absolute value of int_2, and which satisfies the following equation.
>
> $$int_1 = \texttt{(+ (* (/ } int_1 \; int_2\texttt{) } int_2\texttt{) (remainder } int_1 \; int_2\texttt{))}$$
>
> The sign of the result value is the same as the sign of the dividend int_1.

9.3 Bit-wise Operations

(`logand` int_1 int_2) [Function]

> Returns the bit-wise "and" of int_1 and int_2.

(`logior` int_1 int_2) [Function]

> Returns the bit-wise inclusive "or" of int_1 and int_2.

(`logxor` int_1 int_2) [Function]

> Returns the bit-wise exclusive "or" of int_1 and int_2.

`(logshl` *int*$_1$ *int*$_2$`)` [Function]

Returns the integer obtained by shifting the bits that represents *int*$_1$ by *int*$_2$ bits to the left.

`(logshr` *int*$_1$ *int*$_2$`)` [Function]

Returns the integer obtained by shifting the bits that represents *int*$_1$ by *int*$_2$ bits to the right.

9.4 Random Numbers

`(random` *int*`)` [Function]

Returns a pseudo random number which is an integer between `0` (inclusive) and the *int* (exclusive).

10 List Processing

`(cons` *obj*$_1$ *obj*$_2$`)` [Function]

Creates a cons object, sets *obj*$_1$ to the `car` part and *obj*$_2$ to the `cdr` part, and returns the cons object.

`(car` *cons*`)` [Function]

Returns the value in the `car` part of the *cons*.

`(cdr` *cons*`)` [Function]

Returns the value in the `cdr` part of the *cons*.

`(set-car!` *cons obj*`)` [Function]

Replaces the value in the `car` part of the cons object with *obj*, and returns *obj*.

`(set-cdr!` *cons obj*`)` [Function]

Replaces the value in the `cdr` part of the cons object with *obj*, and returns *obj*.

`(length` *list*`)` [Function]

Returns the length of the *list* as an integer.

This function is an installation-time option (see Section 3.5).

`(list-ref` *list int*`)` [Function]

Returns the $(int - 1)^{th}$ element of the *list*. The argument *int* must be a non-negative integer. If *int* is greater than or equal to the length of the *list*, then this function returns `()`.

This function is an installation-time option (see Section 3.5).

`(list` $obj_1 \ \ldots \ obj_n$`)` $(n \geq 0)$ [Function]

Creates and returns a list of length n whose i^{th} element $(1 \leq i \leq n)$ is identical to obj_i.

`(list*` $obj_1 \ \ldots \ obj_n$`)` $(n \geq 1)$ [Function]

Creates and returns a dotted list consisting of $n - 1$ new conses $c_1, \ldots, c_{n-1}$. The car of each c_i will contain obj_i and the cdr of each c_i will contain c_{i+1} except for the last c_{n-1} whose cdr will contain obj_n.

This function is an installation-time option (see Section 3.5).

`(append` $list_1 \ \ldots \ list_n \ \ obj$`)` $(n \geq 0)$ [Function]

Returns the object with all the elements in the *lists* `cons`'ed in front of *obj*.

This function is an installation-time option (see Section 3.5).

`(member` *obj list*`)` [Function]

Searches the list for an element that is identical to *obj*. If such an element is found, returns the sublist starting with the element. Otherwise, returns `#f`.

This function is an installation-time option (see Section 3.5).

`(assoc` *obj list*`)` [Function]

Searches the list for an element which is a cons object and whose value in the `car` part is identical to the *obj*. If such an element is found, returns it. Otherwise, returns `#f`. Each element of the *list* must be a cons object.

This function is an installation-time option (see Section 3.5).

`(reverse` *list*`)` [Function]

Creates and returns a list containing the same elements as the *list* but in the reverse order.

This function is an installation-time option (see Section 3.5).

11 I/O

`:stdin` [Reader Constant]
`:stdout` [Reader Constant]
`:stderr` [Reader Constant]

These reader constants are used to explicitly specify a standard port to the I/O functions of XS such as `read` and `write`. `:stdin` represents the standard input of the front-end subsystem and the value is `0`. `:stdout` represents the standard output of the front-end subsystem and the value is `1`. `:stderr` represents the error output of the front-end subsystem and the value is `2`.

`(read` [*int*]`)` [Function]

Reads an object from the PC keyboard and returns the object. An error will be signaled if the eos (end of stream) is reached before having read an object.

If the optional *int* is supplied and is different from the value of the reader constant `:stdin`, then this function reads from the specified port, instead of the PC keyboard.

`(read-char` [*int*]`)` [Function]

Reads one character from the PC keyboard and returns the character code as an integer. An error will be signaled if the eos (end of stream) is already reached.

If the optional *int* is supplied and is different from the value of the reader constant `:stdin`, then this function reads from the specified port, instead of the PC keyboard.

`(read-line` [*int*]`)` [Function]

Reads one line of characters from the PC keyboard and returns them as a list of character codes. The newline character at the end of the line will be discarded. An error will be signaled if the eos (end of stream) is reached before reading a line.

If the optional *int* is supplied and is different from the value of the reader constant `:stdin`, then this function reads from the specified port, instead of the PC keyboard.

`(write` *obj* [*int*]`)` [Function]

Writes the *obj* to the PC display. Returns the *obj*.

The following output formats are used by the `write` function. Here, $\overline{x}$ denotes the output of the object x by `write`.

- `()`, `#t`, and `#f` are output as "`()`", "`#t`", and "`#f`", respectively.
- An integer is output in decimal, optionally preceded by a minus sign.
- For a symbol, the symbol name is output without escape characters such as vertical bars that enclose symbols.
- A list `(`x_1 ... x_n`)` is output as "`(`$\overline{x_1}$ ... $\overline{x_n}$`)`". A dotted list `(`x_1 ... x_{n-1} `.` x_n`)` is output as "`(`$\overline{x_1}$ ... $\overline{x_{n-1}}$ `.` $\overline{x_n}$`)`". In either case, one space character is output between two $\overline{x}s$ and between an $\overline{x}$ and a dot '`.`'.
- A function is output as "`#<function` *name*`>`" or "`#<function>`", where *name* is the name of the function.

If the optional *int* is supplied and is different from the value of the reader constant `:stdout`, then this function writes to the specified port, instead of the PC display.

`(write-char` *int*$_1$ [*int*$_2$]`)` [Function]

Outputs the character whose character code is the *int*$_1$. Returns the *int*$_1$. For example,

```
(write-char #\a)
```

outputs "`a`" whereas

```
(write #\a)
```

outputs "`97`".

If the optional *int*$_2$ is supplied and is different from the value of the reader constant `:stdout`, then this function writes to the specified port, instead of the PC display.

`(write-string` *list* [*int*]`)` [Function]

Outputs the *list* as a string. Each element of the *list* must be an integer. These elements are output as characters in the order they appear in the *list*. Returns the *list*. For example,

```
(write-string "abc")
```

outputs "`abc`" whereas

```
(write "abc")
```

outputs "`(97 98 99)`".

If the optional *int* is supplied and is different from the value of the reader constant `:stdout`, then this function writes to the specified port, instead of the PC display.

12 RCX Control

12.1 Sound System

`(play` *list*`)` [Function]

> Starts playing the tune specified by the *list*. Each element of the *list* must be a pair `(`*pitch* `.` *length*`)`. The *pitch* must be an integer between `0` (the value of the reader constants `:A0` and `:La0`) and `96` (the value of `:A8` and `:La8`) or the integer `97` (the value of `:pause`). The *length* must be a positive integer. The RCX plays the *pitches* in the *list* from left to right, with each *pitch* for the *length* units of time. If a *pitch* is the `:pause`, then the tune will be paused for the *length* units of time.
>
> This function returns immediately after having initiated the tune, without waiting for the end of the tune. The return value is usually `()`. If the *list* is too long to play, this function returns the sublist of the *list* that will not be played.
>
> If this function is invoked while the RCX is already playing a tune, then the RCX stops playing that tune and starts playing the new tune.

`(playing?)` [Function]

> Returns `#t` if the RCX is currently playing a tune. Otherwise, returns `#f`. This function is mainly used for checking whether the RCX has finished playing the last tune.

`:A0`, `:Am0`, ..., `:Gm8`, `:A8` [Reader Constants]

> These reader constants are used to specify pitches to the function `play`. Each constant name is in the following format.
>
> `:`*basic-note*[`m`]*octave*
>
> The *basic-note* is one of the seven basic notes:
>
> ```
> A  H  C  D  E  F  G
> ```
>
> and the optional "`m`" is the half-note modifier. The *octave* is a digit specifying the number of octaves to transpose the pitch. It should be between `0` (the lowest) and `8` (the highest). Figure 2 lists all the reader constants in this format. The value of the first constant `:A0` is `0`, the value of `:Am0` is `1`, and so on. The value of the last constant `:A8` is `96`.

`:La0`, `:La#0`, ..., `:So#8`, `:La8` [Reader Constants]

									`:A0`	`:Am0`	`:H0`
`:C1`	`:Cm1`	`:D1`	`:Dm1`	`:E1`	`:F1`	`:Fm1`	`:G1`	`:Gm1`	`:A1`	`:Am1`	`:H1`
`:C2`	`:Cm2`	`:D2`	`:Dm2`	`:E2`	`:F2`	`:Fm2`	`:G2`	`:Gm2`	`:A2`	`:Am2`	`:H2`
`:C3`	`:Cm3`	`:D3`	`:Dm3`	`:E3`	`:F3`	`:Fm3`	`:G3`	`:Gm3`	`:A3`	`:Am3`	`:H3`
`:C4`	`:Cm4`	`:D4`	`:Dm4`	`:E4`	`:F4`	`:Fm4`	`:G4`	`:Gm4`	`:A4`	`:Am4`	`:H4`
`:C5`	`:Cm5`	`:D5`	`:Dm5`	`:E5`	`:F5`	`:Fm5`	`:G5`	`:Gm5`	`:A5`	`:Am5`	`:H5`
`:C6`	`:Cm6`	`:D6`	`:Dm6`	`:E6`	`:F6`	`:Fm6`	`:G6`	`:Gm6`	`:A6`	`:Am6`	`:H6`
`:C7`	`:Cm7`	`:D7`	`:Dm7`	`:E7`	`:F7`	`:Fm7`	`:G7`	`:Gm7`	`:A7`	`:Am7`	`:H7`
`:C8`	`:Cm8`	`:D8`	`:Dm8`	`:E8`	`:F8`	`:Fm8`	`:G8`	`:Gm8`	`:A8`		

Figure 2: List of pitches (1).

These reader constants are also used to specify pitches to the function `play`. Each constant name is in the following format.

:*basic-note*[`#`]*octave*

The *basic-note* is one of:

`Do  Re  Mi  Fa  So  La  Si`

and the optional sharp-sign is the half-note modifier. The *octave* is a digit specifying the number of octaves to transpose the pitch. It should be between `0` (the lowest) and `8` (the highest). Figure 3 lists all the reader constants in this format. Each constant has the same value as the constant at the same position in Figure 2.

`:pause` [Reader Constant]

This reader constant is used to specify a pause to the function `play`. The value of this constant is `97`.

12.2 Buttons

The RCX brick has four buttons View, Prgm, On-Off, and Run. Among these, the View button can be used to cause a terminal interrupt and the Prgm button can be used to send a signal to the running program.

`(pressed?)` [Function]

Returns `#t` if the Prgm button has been pressed recently. Otherwise, returns `#f`. For example, the following expression can be used to wait until the Prgm button is pressed.

									:La0	:La#0	:Si0
:Do1	:Do#1	:Re1	:Re#1	:Mi1	:Fa1	:Fa#1	:So1	:So#1	:La1	:La#1	:Si1
:Do2	:Do#2	:Re2	:Re#2	:Mi2	:Fa2	:Fa#2	:So2	:So#2	:La2	:La#2	:Si2
:Do3	:Do#3	:Re3	:Re#3	:Mi3	:Fa3	:Fa#3	:So3	:So#3	:La3	:La#3	:Si3
:Do4	:Do#4	:Re4	:Re#4	:Mi4	:Fa4	:Fa#4	:So4	:So#4	:La4	:La#4	:Si4
:Do5	:Do#5	:Re5	:Re#5	:Mi5	:Fa5	:Fa#5	:So5	:So#5	:La5	:La#5	:Si5
:Do6	:Do#6	:Re6	:Re#6	:Mi6	:Fa6	:Fa#6	:So6	:So#6	:La6	:La#6	:Si6
:Do7	:Do#7	:Re7	:Re#7	:Mi7	:Fa7	:Fa#7	:So7	:So#7	:La7	:La#7	:Si7
:Do8	:Do#8	:Re8	:Re#8	:Mi8	:Fa8	:Fa#8	:So8	:So#8	:La8		

Figure 3: List of pitches (2).

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

Do not confuse this function with `touched?`, which is to check if a touch sensor is touching something.

12.3 LCD Display

`(puts` *list*`)` [Function]

Outputs the *list* to the LCD display. Each element of the *list* must be an integer. These elements are output as characters in the order they appear in the *list*. The return value is usually `()`. Up to five characters can be displayed on the LCD. If the *list* is too long, only the first five characters will be displayed, and `puts` returns the sublist of the *list* that will not be displayed.

`(putc` int_1 int_2`)` [Function]

Outputs the int_1 as a character at the specified column of the LCD display. The column is specified by the int_2, which must be an integer between `0` (the right-most column) and `4` (the left-most column). This function returns the int_1.

`(cls)` [Function]

Clears the LCD display and returns `()`.

12.4 IR Communication Status

`(linked?)` [Function]

Returns `#t` if the RCX is currently able to communicate with the front-end. Otherwise, returns `#f`.

12.5 Battery Level

`(battery)` [Function]

> Returns the current battery level. The value is an integer in 1/10 volts. For example, if the battery level is 8.5 volts, this function returns `85`. Since the RCX is powered by six AA batteries, the maximum return value is about `90`.

13 Lego Devices

13.1 Motors

Motors are attached to the effector ports `A` to `C` of the RCX. In XS programs, the effector ports are specified with integers: `1` for port `A`, `2` for port `B`, and `3` for port `C`. However, it is convenient to use the following reader constants.

`:a` [Reader Constant]
`:b` [Reader Constant]
`:c` [Reader Constant]

> These reader constants are used to specify effector ports: `:a` for port `A`, `:b` for port `B`, and `:c` for port `C`. Their values are `1`, `2`, and `3`, respectively.

The state of a motor is characterized by a direction and a speed. There are four possible directions, which are specified with the following reader constants.

`:off` [Reader Constant]
`:forward` [Reader Constant]
`:back` [Reader Constant]
`:brake` [Reader Constant]

> These reader constants are used to specify motor directions. With `:forward` and `:back`, the motor moves forward and backward. Actually, the meanings of "forward" and "backward" depend on how you connect the motor to the effector port. If the motor does not move in the direction you intend, connect the motor in a different way and retry. With `:off` and `:brake`, the motor does not move. The difference is that `:brake` hinders rotation of the motor, whereas `:off` allows the motor to rotate freely. The values of these constants are: `0` (`:off`), `1` (`:forward`), `2` (`:back`), and `3` (`:brake`).

The speed of a motor is an integer between `0` and `255`. When the speed is `0`, the motor does not move even if its current direction is either `:forward` or

`:back`. When the motor direction is either `:off` or `:brake`, the motor does not move even if its speed is not zero.

`:max-speed` [Reader Constant]

This is the maximum speed of a motor. Its value is `255`.

The following functions are used to change direction and speed of a motor.

`(motor` int_1 int_2`)` [Function]

Changes the direction of the motor that is attached to the effector port int_1 to the direction int_2. Returns the second argument int_2, so that multiple motors can be set up with nested calls such as

```
(motor :a (motor :c :off))
```

The port number int_1 must be one of `1`, `2`, and `3`.

`(speed` int_1 int_2`)` [Function]

Changes the speed of the motor that is attached to the effector port int_1 to the speed int_2. Returns the second argument int_2, so that multiple motors can be set up with nested calls such as

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

The port number int_1 must be one of `1`, `2`, and `3`.

13.2 Light Sensors

Light sensors are active sensors. You have to activate light sensors before using them and inactivate them after use.

`(light-on` *int*`)` [Function]

Activates the light sensor at port *int*, which must be one of `1`, `2`, and `3`. Returns the *int*. When activated, the red lamp on the light sensor will turn on.

`(light-off` *int*`)` [Function]

Inactivates the light sensor at port *int*, which must be one of `1`, `2`, and `3`. Returns the *int*. When inactivated, the red lamp on the light sensor will turn off.

Light sensors return light levels in terms of integers between `0` (black) and `98`[3] (white).

`:white` [Reader Constant]
`:black` [Reader Constant]

> These reader constants `:white` and `:black` represent the maximum value and the minimum value, respectively, that light sensors return.

`(light` *int*`)` [Function]

> Obtains the current light level from the light sensor at port *int* and returns the value as an integer. The port number *int* must be one of `1`, `2`, and `3`.

13.3 Rotation Sensors

Rotation sensors are active sensors. You have to activate rotation sensors before using them and inactivate them after use.

`(rotation-on` *int*`)` [Function]

> Activates the rotation sensor at port *int*, which must be one of `1`, `2`, and `3`. Returns the *int*. When activated, the current angle position of the rotation sensor will be used as the origin. The rotation values obtained later will be measured relative to the origin. The origin will be reset by another call of `rotation-on`.

`(rotation-off` *int*`)` [Function]

> Inactivates the rotation sensor at port *int*, which must be one of `1`, `2`, and `3`. Returns the *int*.

`(rotation` *int*`)` [Function]

> Obtains the current rotation value from the rotation sensor at port *int*, which must be one of `1`, `2`, and `3`, and returns the value as an integer. The value is in 360/16 degrees relative to the origin that has been set when the rotation sensor was activated. The value may be positive or negative depending on the direction of rotation. For instance, when the rotation sensor has been turned 270 degrees after activation, then this function will return `12` ($= 270/360 \times 16$) or `-12`. If the sign of the value is not what you expect, then turn upside down the rotation sensor.

[3]This value comes from the underlying brickOS. I do not know why it is not `99` or `100`.

13.4 Temperature Sensors

Temperature sensors are passive sensors. You do not have to activate/inactivate temperature sensors.

`(temperature` *int*`)` [Function]

> Obtains the current temperature from the temperature sensor that is attached to the port *int*, which must be one of `1`, `2`, and `3`, and returns the value as an integer in Celsius. For those who are not familiar with the Celsius system, 0°C is the freezing point and 100°C is the boiling point. Our body temperature is about 37°C.

13.5 Touch Sensors

Touch sensors are passive sensors. You do not have to activate/inactivate touch sensors.

`(touched?` *int*`)` [Function]

> Returns `#t` if the touch sensor at port *int* (which must be one of `1`, `2`, and `3`) is touching something. Otherwise, returns `#f`. For example, the following expression can be used to wait until the touch sensor at port 1 touches something.
>
> ```
> (wait-until (touched? 1))
> ```
>
> Do not confuse this function with `pressed?`, which is to check if the Prgm button has been pressed.

13.6 Lamps

Lamps can be attached to effector ports, and controlled by the functions `motor` and `speed` above. To turn on the lamp that is attached to port *port*, use

`(motor` *port* `:forward)` or `(motor` *port* `:back)`.

To turn off, use

`(motor` *port* `:off)` or `(motor` *port* `:brake)`.

The `speed` function can be used to change the brightness of the lamp. The higher the speed, the brighter the lamp. The maximum brightness is `:max-speed`.

Note that if a lamp and a motor are attached to the same effector port, then the lamp will be lit only when the motor is moving and the brightness will indicate the motor speed.

14 Memory Management

Some XS objects are internally represented with *cells*, which are allocated in the memory area, called the *heap*, in the RCX RAM memory. Each cell is a data structure that is capable of storing two XS objects in it. Thus cells are similar to cons objects. In fact, each cons object is represented with a cell. However, cells are used also for representing some other objects, such as user-defined symbols and user-defined functions.

Cells that are not used for representing objects are called *free cells*. Initially, when an XS session is started up, all cells in the heap are free. Each time the XS system needs cells to represent an object, it finds an appropriate number of free cells in the heap. Eventually, all available cells will be used up and there is no free cell in the heap. In order to continue the computation, the XS system recycles those cells that were once used to represent objects but that are not in use any more. This process of recycling is called *garbage collection*. Usually, garbage collection takes place when there is no more free cell, but the user can force a garbage collection by using the function `gc` (see below).

The number of cells allocated in the heap is determined at installation time of your RCX system. If XS cannot find a free cell after automatic garbage collection, then XS cannot continue the current computation. In that case, XS aborts the current computation with an error message "`heap full`". Even then, there remains a chance that the user can continue interacting with the XS system because some cells may be freed up by aborting the computation.

`(gc)` [Function]

Forces a garbage collection, and returns the number of free cells.

Acknowledgement

This project is sponsored by the Information-technology Promotion Agency (IPA) as an Exploratory Software Project. I would like to thank Dr. Takashi Chikayama, the Project Manager, for his valuable advises. I would also like to thank Dr. Sheng-Chuan Wu who carefully reviewed a draft of this manual and gave valuable comments to improve it.

Index

XS Handy Reference

This Reference lists all XS functions together with their parameter profiles, and all XS reader constants. 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

Special form names (including top-level form names) are underlined. Functions marked with ' † ' are installation-time options.

Common Functions

- top-level forms
  (define *sym form*)
  (define (*sym sym** [. *sym*]) *form**)
  (load *string*)
  (trace *sym*)
  (untrace *sym*)
  (bye)
- basic forms
  (quote *obj*)
  (set! *sym form*)
  (lambda (*sym** [. *sym*]) *form**)
- control
  (begin *form**)
  (apply *fun obj** *list*)
  (trace-call *sym fun list*)
  (if *form form* [*form*])
  (catch *form form**)
  (throw *obj obj*)
- conditional
  (and *form**)
  (or *form**)
  (not *obj*)
- binding
  (let [*sym*] ((*sym form*)*) *form**)
  (let* ((*sym form*)*) *form**)
  (letrec ((*sym form*)*) *form**)
- type predicates
  (boolean? *obj*)
  (integer? *obj*)
  (null? *obj*)
  (pair? *obj*)
  (symbol? *obj*)
  (function? *obj*)
- comparison
  (eq? *obj obj*)
  (< int^+)
  (> int^+)
  (= int^+)
  (>= int^+)
  (<= int^+)
- arithmetic
  (+ *int**)
  (- *int int**)
  (* *int**)
  (/ *int int*)
  (remainder *int int*)
  (logand *int int*)
  (logior *int int*)
  (logxor *int int*)
  (logshl *int int*)
  (logshr *int int*)
  (random *int*)
- list processing
  (car *cons*)
  (cdr *cons*)
  (cons *obj obj*)
  (set-car! *cons obj*)
  (set-cdr! *cons obj*)
  (list *obj**)
  (list* *obj** *obj*) †
  (list-ref *list int*) †
  (append *list** *obj*) †
  (assoc *obj a-list*) †

(member *obj list*) †
(length *list*) †
(reverse *list*) †

- I/O
  (read [*int*])
  (read-char [*int*])
  (read-line [*int*])
  (write *obj* [*int*])
  (write-char *char* [*int*])
  (write-string *string* [*int*])

- garbage collection
  (gc)

Lego-specific Functions

- top-level forms
  (last-value)
  (fork *sym sym string*$^+$)
  ; Linux version only

- control
  (sleep *int*) ; in 1/10 seconds
  (wait-until *form*)
  (with-watcher ((*form form**)*) *form**)

- system clock
  (time) ; value in 1/10 seconds
  (reset-time)

- IR communication test
  (linked?)

- light sensors
  (light-on {1|2|3})
  (light-off {1|2|3})
  (light {1|2|3})
  ; value 0 (:black) to 98 (:white)

- rotation sensors
  (rotation-on {1|2|3})
  (rotation-off {1|2|3})
  (rotation {1|2|3})
  ; value in 360/16 degrees

- temperature sensors
  (temperature {1|2|3})
  ; value in Celsius

- touch sensors
  (touched? {1|2|3})

- motors
  (motor {:a|:b|:c} {:off|:forward|:back|:brake})
  (speed {:a|:b|:c} *speed*)
  ; $0 \leq$ *speed* ≤ 255 (:max-speed)

- sounds
  (play ((*pitch* . *length*)*))
  ; see below for pitches
  (playing?)

- Prgm button
  (pressed?)

- LCD display
  (puts *string*)
  (putc *char column*)
  ; $0 \leq$ *column* ≤ 4 (left-most)
  (cls)

- battery level
  (battery) ; value in 1/10 volts

Other Reader Constants

- standard ports
  :stdin, :stdout, :stderr

- pitches
  :A0, :Am0, :H0, :C1, :Cm1, :D1, :Dm1, :E1, :F1, :Fm1, :G1, :Gm1, :A1, . . ., :A8

  :La0, :La#0, :Si0, :Do1, :Do#1, :Re1, :Re#1, :Mi1, :Fa1, :Fa#1, :So1, :So#1, :La1, . . ., :La8

  :pause

- integers
  :most-positive-integer
  :most-negative-integer