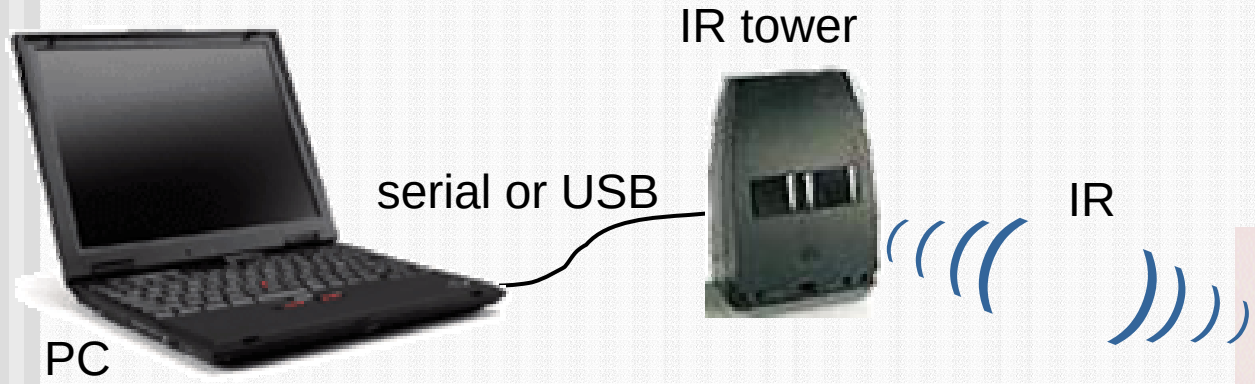
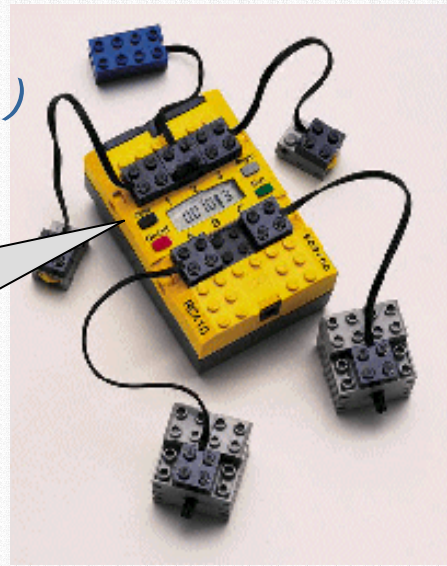
XS: Lisp on Lego™ Mindstorms

Taiichi Yuasa
Kyoto University

Hardware organization of Mindstorms



- 16 MHz Hitachi H8 MPU with 16-bit addressing space
- 32 KB ROM and 32 KB RAM
- LCD display & 4 buttons
- 3 effector ports & 3 sensor ports
- powered by 6 AA batteries

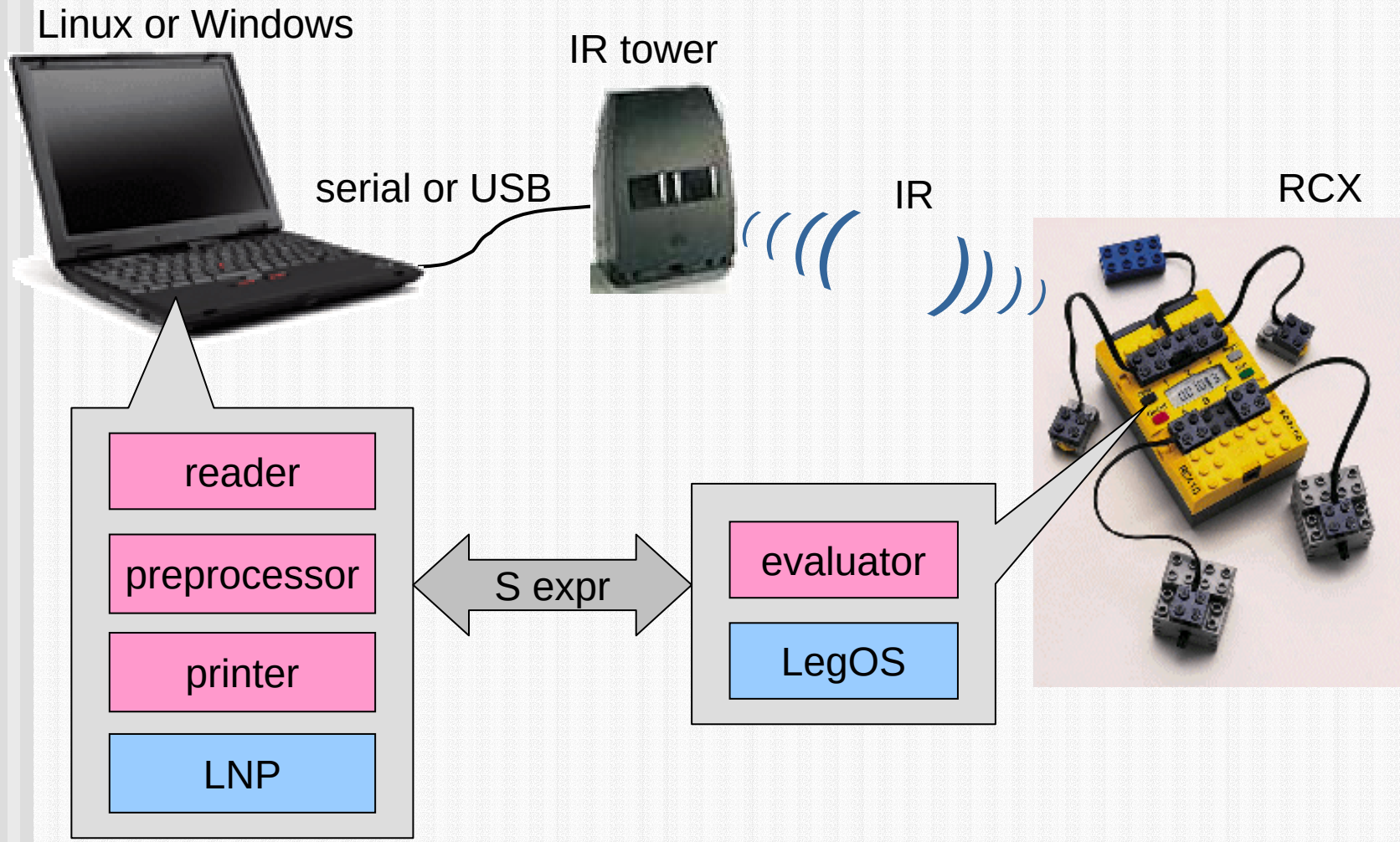


RCX 1.0, 1.5, or 2.0 attached with Lego devices and bricks

Features of XS

- Interactive program development
  - read-eval-print loop
  - interactive definition and re-definition of functions
  - appropriate error message with backtrace
  - trace and untrace functions
- Autonomous evaluator in RCX
  - dynamic object allocation and garbage collection
  - truly tail-recursive interpreter
  - robust against program errors and stack / buffer overflow
  - terminal interrupts
- Sufficient functionality to control robots
  - Scheme-like language with no first-class continuations
  - interface to Lego devices such as motors, sensors, lamps, sounds, ...
  - event / timer waiting and asynchronous event watchers

System overview of XS



Data types

- booleans: #f, #t
- integers: 14-bit signed
- empty list: ()
- conses
- functions
  - built-in functions
  - lambda closures (user-defined functions)
- symbols
  - built-in symbols (names of built-in functions)
  - user-defined symbols

Pseudo data types & reader constants

Converted by the Reader

- string ⇒ list of character code
  ex. “abc” ⇒ (97 98 99)
- character ⇒ ASCII code
  ex. #¥a ⇒ 97
- reader constants ⇒ integer
  :most-positive-integer, :most-negative-integer
  :a, :b, :c, :off, :forward, :back, :brake, :max-speed,
  :white, :black,
  :A0, :Am0, ..., Gm8, A8, La0, :La#0, ..., :So#8, :La8, :pause

Common functions

- top-level
  - (define sym expr)
  - (define (sym sym* [. sym]) expr*)
  - (load string) ; load from the named file
  - (trace sym)
  - (untrace sym)
  - (bye) ; sayonara
- basic
  - (quote object)
  - (set! sym expr)
  - (lambda (sym* [. sym]) expr*)

Common functions (cont.)

- control
  - (begin expr*)
  - (if expr expr [expr])
  - (apply function object* list)
  - (catch expr expr*)
  - (throw object object)
- condition
  - (and expr*)
  - (or expr*)
  - (not object)
- binding
  - (let [sym] ((sym expr)*) expr*)
  - (let* ((sym expr)*) expr*)
  - (letrec ((sym expr)*) expr*)

Common functions (cont.)

- type predicates
  - (boolean? object)
  - (integer? object)
  - (null? object)
  - (pair? object)
  - (symbol? object)
  - (function? object)
- comparison
  - (eq? object object)
  - (< 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)

Common functions (cont.)

- list processing
  - (car pair)
  - (cdr pair)
  - (cons object object)
  - (set-car! pair object)
  - (set-cdr! pair object)
  - (list object*)
  - (list* object* object)
  - (list-ref list int)
  - (append [list* object])
  - (assoc object a-list)
  - (member object list)
  - (length list)
  - (reverse list)
- I/O from/to front-end PC
  - (read)
  - (read-char)
  - (read-line)
  - (write object)
  - (write-char char)
  - (write-string string)
- garbage collection
  - (gc) ; returns # of free cells

Lego-specific functions

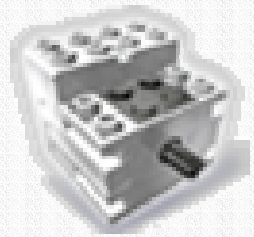
- top-level
  - (last-value) ; say that again?
  - (ping) ; are you alive?
- control
  - (sleep int) ; in 1/10 seconds
  - (wait-until cond)
  - (with-watcher ((cond . handler)*) . body)
    ; asynchronous event watchers
- system clock
  - (time) ; in 1/10 seconds (overflows in 13 min)
  - (reset-time)

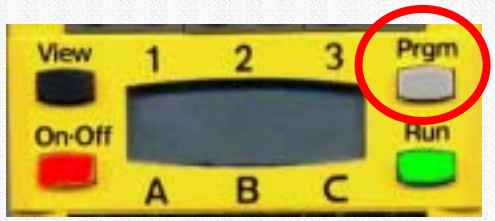
Lego-specific functions (cont.)

- light sensors
  - (light-on {1|2|3})
  - (light-off {1|2|3})
  - (light {1|2|3})
- rotation sensors
  - (rotation-on {1|2|3})
  - (rotation-off {1|2|3})
  - (rotation {1|2|3})
- temperature sensors
  - (temperature {1|2|3})
- touch sensors
  - (touched? {1|2|3})

Lego-specific functions (cont.)

- motors
  - (motor {:a|:b|:c} {:off|:forward|:back|:brake})
  - (speed {:a|:b|:c} int)
- sounds
  - (play ((pitch . length)*))
  - (playing?)
- Prgm button
  - (pressed?)
- LCD display
  - (puts string)
  - (putc char int)
  - (cls)
- battery level
  - (battery)

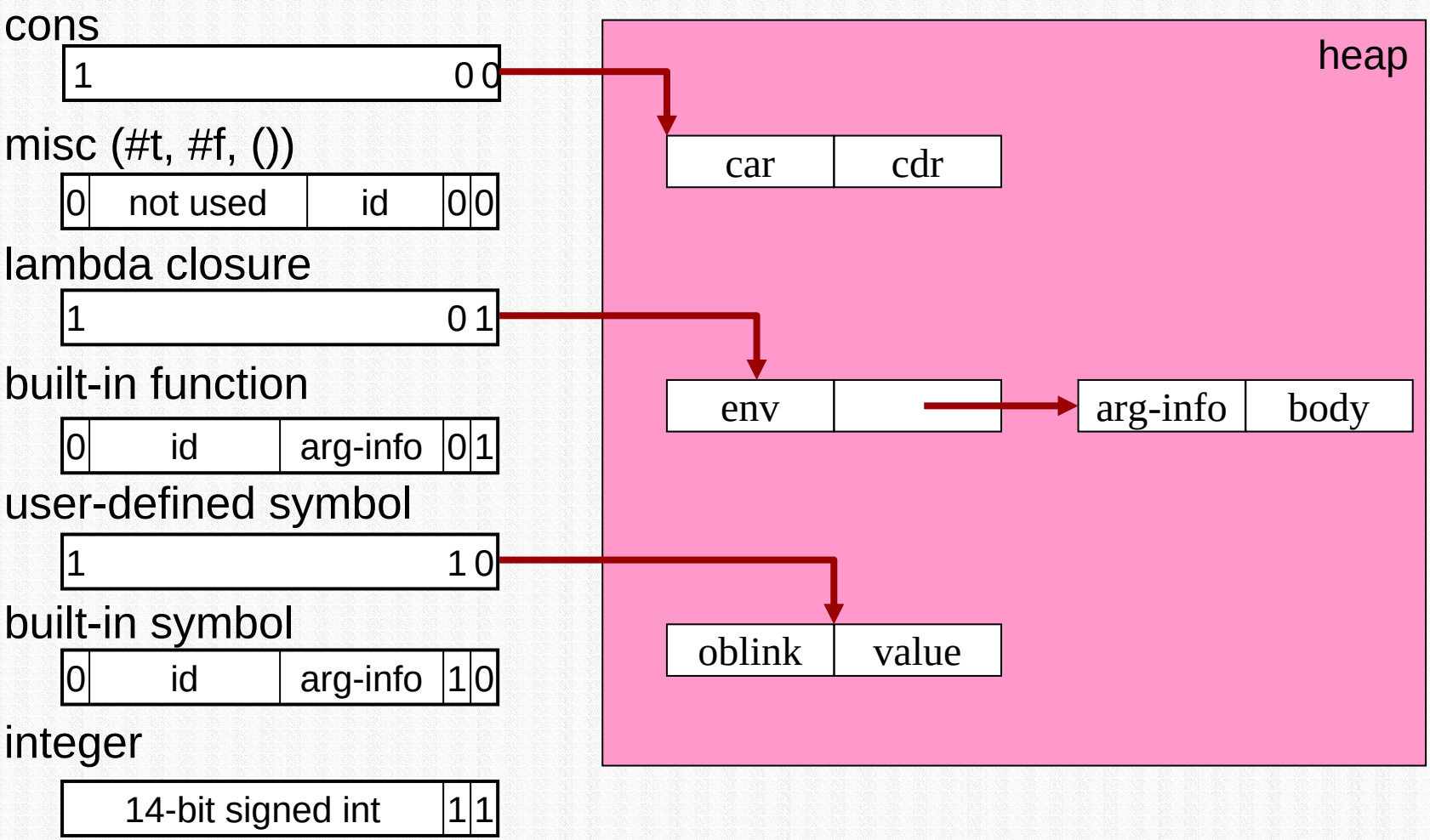




The Evaluator

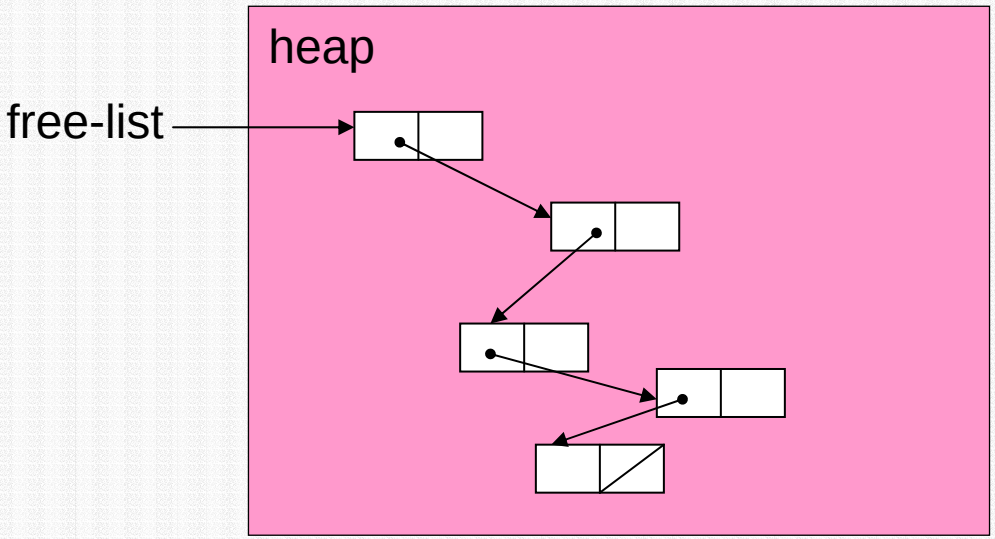
- written entirely in C
- compiled by GNU cross compiler
- sizes
  - LegOS: 14 KB
  - binary: 11 KB (including all built-in functions)
  - I/O buffer: 256 bytes
  - C stack: 512 words (= 1 KB)
  - variable stack: 256 words (= 0.5 KB)
  - heap: 768 cells (= 3 KB)

Object representation



Heap management

- every cell occupies two words (= 4 bytes)
- no need for compaction
- free cells are linked together to form a free-list
- mark & sweep, stop-the-world garbage collection

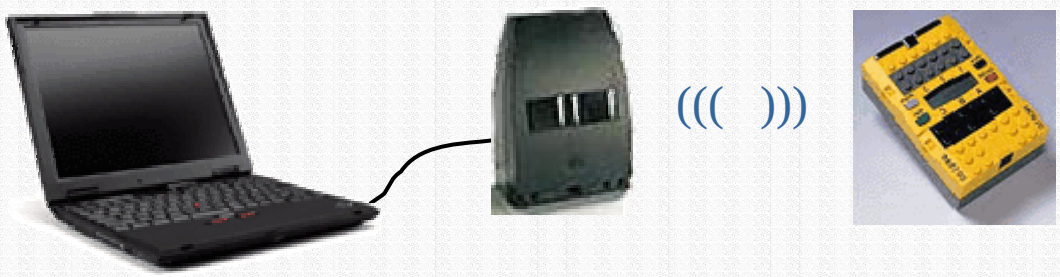


Current status of XS project

- Linux version for RCX 1.0 & 1.5 (serial) completed
- Windows version and support for RCX 2.0 (USB) almost finished by Franz Inc. (many thanks to John Foderaro)
- draft reference manual ready
- will soon start Web distribution as an open source

  will be linked from http://www.yuasa.kuis.kyoto-u.ac.jp/~yuasa

  and maybe from http://www.franz.com/



※ The project of XS is sponsored by the Information-technology Agency (IPA) of Japan as an Exploratory Software Project

show time

Using XS: Preparation

- Install GNU cross compiler for Hitachi H8 CPU, available at:

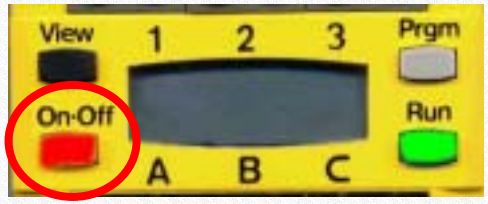
  http://legos.sourceforge.net/files/linux/
- Download legOS version 0.2.4 from:

  http://legOS.sourceforge.net/files/common/

  and “make” it.
- Connect the IR tower to your PC and turn on your RCX.
- Download legOS:

  % util/firmdl3 boot/legOS.srec
- Download the XS evaluator:

  % util/dll xs/eval.lx



- You may now turn off the RCX, since both legOS and the XS evaluator are kept in the RAM as long as the batteries are alive.

Using XS: Starting up

- Turn on your RCX and press the Run button.
- Start the XS front end:

```
% xs/xs
Welcome to XS: Lisp on Lego Mindstorms

>
```

- Following the prompt '>', enter a top-level form:

```
>(cons 1 2)
(1 . 2)
>
```

- To end the XS session, type (bye) or press Control-D:

```
>(bye)
sayonara
%
```

- Turn off your RCX.

Using XS: Error messages

- When an error is detected, you will see an error message, occasionally followed by a backtrace:

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

- Even then, the system is still alive. You may fix the bug online.

```
>(define nil ())
nil
>(ints 3)
(3 2 1)
>
```

Using XS: Trace and Untrace

- To see how some functions are invoked, use trace:

```
>(trace ints)
ints
>(ints 3)
0>(ints 3)
   1>(ints 2)
      2>(ints 1)
         3>(ints 0)
         3<(ints ())
      2<(ints (1))
   1<(ints (2 1))
0<(ints (3 2 1))
(3 2 1)
>
```

- To cancel the tracing, use untrace:

```
>(untrace ints)
ints
>(ints 3)
(3 2 1)
```

Using XS: Terminal interrupt

- If your program enters into an infinite loop, press Control-C to abort the current evaluation:

  >(let loop () (loop))

  ---- you press Control-C here ----

  Error: terminal interrupt

  Backtrace: let > #<function>

  >

- You may also press the View button of your RCX to abort the evaluation:

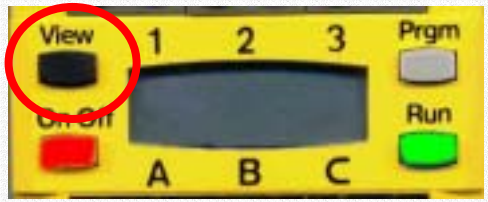
  >(let loop () (loop))

  ---- you press the View button here ----

  Error: terminal interrupt

  Backtrace: let > #<function>

  >



Programming XS: Tail recursion

- Because of the small size of RCX memory, nested function calls sometimes cause stack overflow:

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

  ints

  >(ints 20)

  Error: RCX C stack overflow -- 2

  Backtrace: ints > ints > ints > ints > ints > ints > ints > ints > ints > ints > ints > ints > ints > ints > ints > ints > ints > ints > ints

- Tail recursion is a programming technique, effective to avoid stack overflow:

  >(define (ints n x) (if (= n 0) x (ints (- n 1) (cons n x))))

  ints

  >(ints 20 ())

  (1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20)

Programming XS: Loops

- You may have noticed XS has no loop constructs such as while, for, do-while in C.
- This is because you can easily realize loop constructs by using tail recursion
  - while loop:

    (let loop () (if condition (begin body (loop))))
  - do-while loop:

    (let loop () body (if condition (loop)))

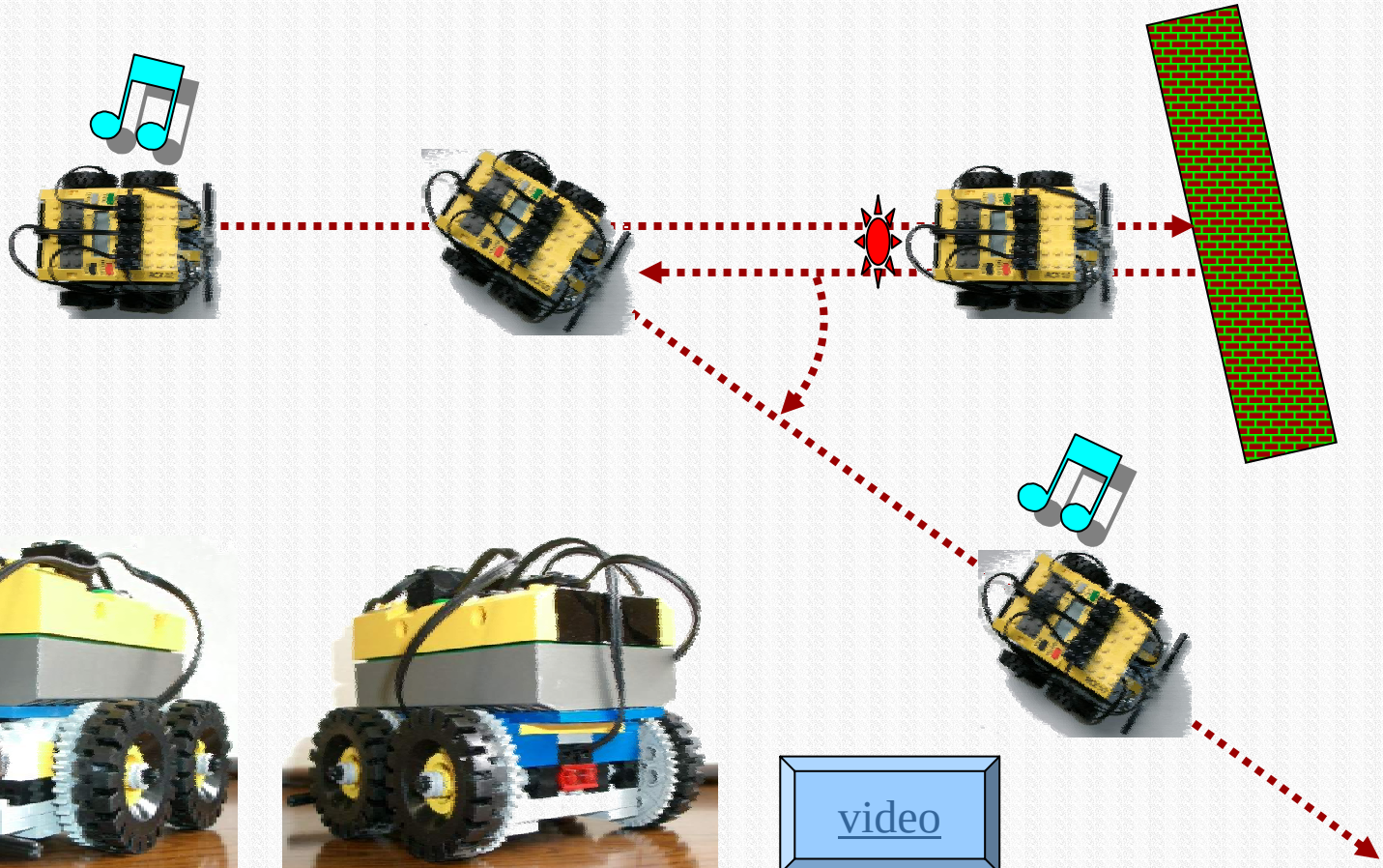
Programming XS: Event watchers

- a watcher is an asynchronous event-driven handler
- watchers are established by with-watcher

  (with-watcher (($event_1$. $handler_1$) ... ($event_n$. $handler_n$))
  . body)

  - $watcher_1$ ... $watcher_n$ are activated in this order and remain active during execution of body
  - new watcher is given a priority higher than any active watcher
  - only the watcher with the highest priority whose event evaluates to true is triggered at a time
  - when a handler is running, only watchers with higher priority may be triggered
  - when a watcher is triggered, the currently running handler is suspended during execution of the handler of the triggered watcher
  - no watcher is triggered while events are being evaluated

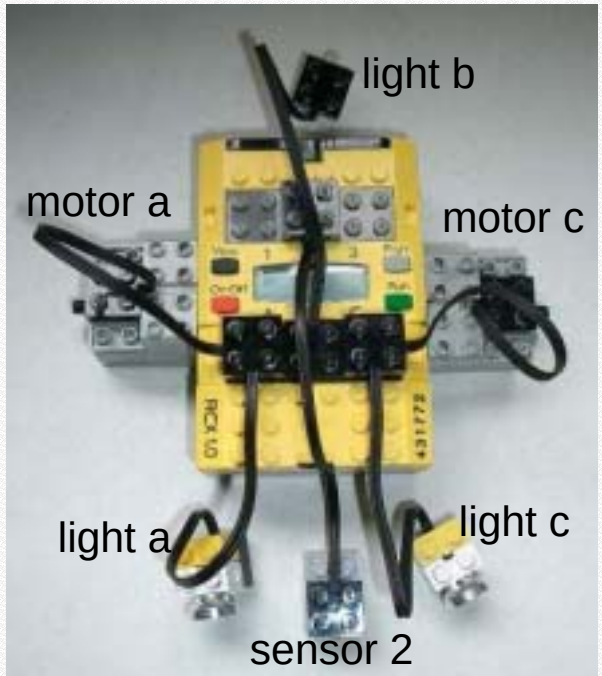
Sample program: Land Rover



video

Sample program: Land Rover

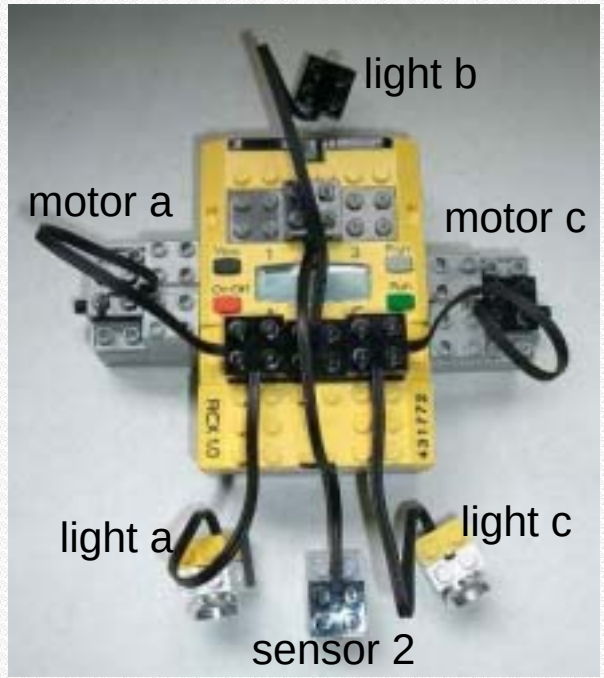
```
(begin
  (speed :a (speed :c (speed :b :max-speed)))
  (let loop ()
    (motor :a (motor :c :forward))
    (motor :b :off)
    (play '((:Re4 . 2) (:Do4 . 1) (:Re4 . 1) (:Fa4 . 1) (:Re4 . 1) (:Re4 . 2) (:Fa4 . 2)
            (:So4 . 1) (:Do5 . 1) (:La4 . 2) (:Re4 . 2)))
    (wait-until (or (touched? 2) (pressed?)))
    (if (pressed?)
        (motor :a (motor :c :off))
        (begin
          (motor :a (motor :c (motor :b :back)))
          (sleep 5)
          (motor (if (= (random 2) 0) :a :c) :forward)
          (sleep 5)
          (loop))
      )))
```



Sample program: Land Rover II

```
(define (forward)
  (motor :a (motor :c :forward))
  (motor :b :off)
  (play '((:Re4 . 2) (:Do4 . 1) (:Re4 . 1) (:Fa4 . 1) (:Re4 . 1) (:Re4 . 2) (:Fa4 . 2)
          (:So4 . 1) (:Do5 . 1) (:La4 . 2) (:Re4 . 2))))

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



Tracing Rover

1. Tracks a line, while recording the movement as a list
2. Draws the line on a white paper, by replaying the recorded movement