Lego MindStorms用のLisp処理系XS

湯淺 太一

ロボット開発キットである Lego MindStorms は，8 ビット CPU を搭載した RCX と呼ばれるブロックにさまざまなセンサやモータを接続することによって，自作ロボットの制御を可能にしている．本論文では，RCX 上で動作し，ロボットを制御するために設計された Lisp 処理系である XS を紹介する．XS の評価器は RCX 上で自律的に動作し，独自の実行時スタックやごみ集めの対象となるヒープを備えている．フロントエンド PC と交信することによって，バックトレース，関数トレース，端末機割込みなどの機能を提供し，対話的なプログラミング環境を実現している．処理系がサポートするプログラミング言語は Scheme をベースにし，モータやセンサなどとのインターフェイスを備えている．さらに，イベント待ちや非同期イベント割込みといったロボット制御に不可欠な機能も備えている．

Lego MindStorms is a robot development kit which makes it possible to control one's own robot by attaching various sensors and motors to a programmable block, called RCX, with an 8-bit CPU. In this paper, we present a Lisp system XS which runs on an RCX block to control robots. Unlike previous Lisp/Scheme implementations for the MindStorms, the evaluator of XS runs autonomously on the RCX, with its own runtime stack 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 programming language based on Scheme, extended with functionality for interfacing with attached devices such as motors and sensors. It also supports mechanisms such as event waiting and asynchronous event interrupts that are necessary for controlling robots.

1 はじめに

ブロック玩具で世界的に知られる Lego 社が，Lego MindStorms Robotics Invention System (RIS) [8] と呼ばれるロボット自作キットを提供している．このシステムの中心になるのは，8 ビットの CPU を備えてプログラミング可能な RCX と呼ばれるブロックである．これにモータやセンサ，その他のブロックを接続することによって，ユーザのプログラムで制御できるロボットを組み立てることができる．子供でも自分のロボットを制御するプログラムが書けることから，RIS システムはプログラミングやロボティクスの教育に使われている．

Lego 社から提供されているプログラミング環境を使用する場合，簡単なビジュアル言語を使って RCX のプログラムを記述する [1]．単純な言語なので，小さな子供でも使用することができる．その反面，このビジュアル言語は，通常のプログラミング言語では欠かせないパラメータ化されたサブプログラム (関数) やユーザ定義の変数 (大域変数および局所変数) といった概念をサポートしていないために，実際的なプログラミングの学習のためには適さないようである．文法的には C 言語に似た NQC(Not Quite C) [2] という言語も利用できるが，基本的にビジュアル言語のテキスト版である．RCX のファームウェアの制限 [11] のために，NQC の記述能力にも限界がある．例えば，NQC には関数という概念があるが，ユーザが定義した関数どうしが呼び出し合うことさえ許されない．

オープンソースの OS である brickOS [3] (以前は

XS: Lisp on Lego MindStorms.

Taiichi Yuasa, 京都大学情報学研究科, Graduate School of Informatics, Kyoto University.

コンピュータソフトウェア, Vol.24, No.4 (2007), pp.51–65.
[ソフトウェア論文] 2007 年 1 月 21 日受付.

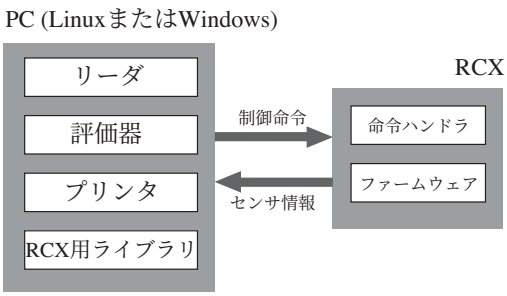

図 1　LegoScheme と LegoLisp のシステム構成．

legOS [9] [10] [13] と呼ばれていた) が RCX 用に開発されている．RCX のファームウェアを brickOS で置き換えることによって，C 言語で制御プログラムが書けるようになった．この OS は，OS のメモリ領域をユーザプログラムから保護していないために，バグのあるプログラムを実行すると，簡単に OS がクラッシュしてしまう．しかも，C はコンパイラ指向の言語であり，簡単な機能をテストするにも，ソースファイルを準備してそれをコンパイルし，バイナリをダウンロードして実行するという長いステップを必要とする．

本論文で紹介する言語システム XS は，次の目的のために開発された．

- 子供や初心者プログラマでも容易に使用できるように，インタープリタを基本とする言語システムとする．
- 高水準のプログラミング言語機能を利用したプログラム開発を可能とする．
- 効率的なプログラム開発のための対話的なプログラミング環境を提供する．

同様の目的のために開発されたシステムに，LegoScheme [14] と LegoLisp [6] がある．いずれも，Lisp ファミリーの拡張言語をサポートしている．これらのシステムでは，プログラムはフロントエンドの PC で動作し，RCX 用の拡張ライブラリを介して制御命令を RCX に送信する (図 1 参照)．RCX は制御命令に従ってモータを制御したり，センサからの入力を取得してフロントエンドに送信したりする．フロントエンド PC と RCX とは赤外線を使って交信するために，RCX が制御命令を受け損なってしまうことがある．特に，移動するロボットを制御する場合には，ロボットが赤外線の通信範囲を出てしまったり，赤外線通信ポートが正しい方向を向いていない場合に信号をとりこぼす．その上，赤外線通信は低速であるため，センサ入力に対応するためにフロントエンドから送られる制御命令を，RCX が妥当な時間内に受信できないことがある．我々の XS システムも，対話的なプログラミング環境を提供するために Lisp ベースの言語をサポートしている．しかし XS では，プログラムは RCX にダウンロードされ，RCX 内で自律的に動作する評価器 (evaluator) によって実行される．これによって，上記二つの Lisp ベースの言語システムの問題を解決している．

対話的で満足のいくプログラミング環境を提供するために，XS には次の機能が備わっている．

1. read-eval-print ループ
2. 関数を対話的に定義および再定義する機能
3. 適切なエラーメッセージ表示とバックトレース機能
4. 関数のトレース機能
5. 動的なオブジェクトの生成とごみ集め機能
6. プログラムエラーやスタック/バッファオーバーフローに対する頑健性
7. 端末機割込み
8. 真に末尾再帰的なインタープリタ
9. イベント待ち，タイマー待ち，非同期のイベント割込み機能
10. モータやセンサなどのデバイスへのインターフェイス

これらの機能のうち，最後の三つ以外は通常の Lisp システムにも備わっている．また，通常の Scheme システムには，末尾再帰的なインタープリタが備わっている．マルチスレッド対応の Lisp システムなら，イベント待ち等の (あるいはそれらに匹敵する) 機能を備えている．したがってユーザの視点からは，XS は通常の Lisp システムにモータやセンサを制御するための拡張をほどこしたものとみなすことができる．

次節では，本論文を理解するために必要な RCX の機能を紹介する．3 節では，XS システムの概要を紹介し，システムの具体的なイメージを与える．4 節で

は，XS がサポートする言語を紹介し，5 節で簡単なプログラム例をあげる．6 節では XS の主要な機能の実装の詳細について述べ，最後に 7 節でプロジェクトの現況を報告する．

2 RCX

RCX の CPU は，16 MHz の Hitachi H8 マイクロプロセッサであり，16 ビットのアドレス空間を持っている．1 台の RCX には 32K バイトの ROM と，32K バイトの RAM が搭載されている．ROM の内容はユーザが変更することができず，RAM のいくつかの領域は ROM プログラムのために予約されている．RAM のその他の領域が RCX のファームウェア (あるいは brickOS などの専用 OS) とユーザプログラムのために利用可能である．補助記憶はない．メモリの制約が厳しいために，RCX 用の言語システムを開発するときは，実行速度よりもメモリ効率を優先することになる．

RCX ブロックの前面 (図 2 参照) には赤外線ポートがあり，これがフロントエンド PC との唯一の交信手段である．PC 側は，Lego 社が提供する「赤外線タワー」を使って赤外線通信を行う．赤外線タワーは，PC のシリアルポート (RCX のモデル 1.0 と 1.5 の場合) あるいは USB ポート (モデル 2.0 の場合) に接続する．赤外線の交信範囲は比較的狭いので，移動するロボット等に RCX を組み込んでいる場合は，PC と RCX が交信に失敗することがある．

RCX ブロックの上面には三つの出力ポート (A, B, C) と三つのセンサポート (1, 2, 3) がある．上面の中心 (図 3 参照) には LCD ディスプレイがあり，最大 5 文字まで表示することができる．LCD ディスプレイの周りには四つのボタン On-Off, Prgm, Run, View がある．On-Off は RCX の電源ボタンであり，Prgm は RCX 内に格納されたユーザプログラムを選択するために使い，Run は選択したプログラムの実行を開始する．View は実行中のプログラムをモニターするために用意されているが，実際には滅多に使用しない．

RCX は，ブロックの底に収納された 6 本の単三乾電池で動作する．RCX に接続されるモータやアクティブセンサもこれらの電池を使って駆動する．RCX

図 2 RCX(出力ポート A と C にモータを，センサポート 1 番と 3 番にタッチセンサを，2 番に光センサを接続している)．

図 3 LCD, 四つのボタン，ポートラベル．

の電源がオフになっても，RAM メモリの内容は維持されるので，次に RCX の電源をオンにすると以前にダウンロードしたユーザプログラムを直ちに実行することができる．

3 XS の概要

XS のシステムは，PC 上で動作するフロントエンド処理系と，RCX 上で動作する評価器 (evaluator) から構成されている．これら二つのサブシステムが協調することによって，通常の Lisp システムのように見える対話的プログラミング環境を提供している．

XS のシステム構成を図 4 に示す．ユーザは，フロントエンドである PC(Linux または Windows) を使ってプログラムを開発する．ユーザが Lisp の S 式を一つ入力すると，フロントエンドのリーダがこれを読み込み，簡単な前処理を行ったあと，RCX 側の評価器に送信する．評価器はこの S 式を評価して結果をフロントエンドに返送する．フロントエンドのプリンタがこの結果を PC のディスプレイに表示する．この一連のステップを繰り返すことによって，ユー

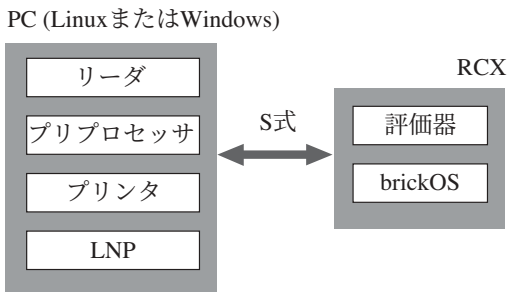

図 4　XS のシステム構成．

ザは MindStorms 用のプログラムを開発するのである．この繰り返しは，通常の Lisp 処理系における read-eval-print ループに相当する．通常の処理系と本質的に異なるのは，評価器が別プロセッサ上で動作し，赤外線通信を使って S 式を授受する点である．また，図 1 と比較すると明らかなように，評価器がフロントエンドではなく RCX に配置されている点が，前述の LegoScheme や LegoLisp との基本的な相違である．

図 5 に XS を使った対話例をあげる．フロントエンド処理系が起動される (1 行目) と，プロンプト “>” を表示し，ユーザとの対話を開始する．プロンプトに続けて，ユーザは関数定義を入力 (3〜5 行目) し，それをテストする (7 行目)．この関数は未定義の変数 `nil` を参照しているために，エラーメッセージ (8 行目) とバックトレース (9 行目) が表示される．ユーザは空リストを初期値として変数 `nil` を定義 (10 行目) し，再度テストを行う (12 行目)．今回は関数は正しい値を返し (13 行目)，ユーザは満足して XS のセッションを終了する (15 行目)．この例では，ユーザにとって特別なことは何もないが，内部的には，通常の Lisp 処理系とはまったく異なった処理が行われている．それは，評価器が RXC 上で動作しているためである．

フロントエンド処理系が起動されると，まず RCX 側の準備ができているかどうかを調べる．もし準備できていなければ (例えば，ユーザが RCX の電源を入れ忘れた場合)，フロントエンドはユーザとの対話をあきらめる．

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

```
 1  % xs
 2  Welcome to XS: Lisp on Lego MindStorms
 3  >(define (ints n)
 4     (if (= n 0) nil
 5         (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  %
```

図 5　XS との対話例 (説明のために行番号をつける)．

RCX の準備ができていれば，フロントエンドはプロンプトを表示してユーザとの対話を開始する．ユーザが PC のキーボードから S 式を一つ入力すると，それを前処理し，結果を RCX に送信する．評価器がこの (前処理済みの)S 式を評価し，結果を返信し，これが PC のディスプレイに表示される．関数定義式を入力した場合は，評価器が定義をインストールし，関数の名前である記号を返信する．

ユーザはコントロール C を入力することによって，実行中のプログラムを中断することができる．次の例では，`let` 式が無限ループを実行中に，ユーザがコントロール C を入力している．

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

このような端末機割込みの要求もフロントエンドから赤外線通信を使って RCX に受け渡される．もし RCX が赤外線通信のエリア外であれば，割込み要求を受け取ることができない．そのような場合に備えて，RCX の View ボタンを使った端末機割込みの方法を用意している．RCX が赤外線通信のエリア内にあれば，View ボタンを使った端末機割込みは，コント

ロール C を使った割込みとまったく同様に動作する．しかしエリア外の場合は，フロントエンドは View ボタンによって割込みが起きたことを知る方法がないので，RCX からの返答を待ち続ける．この場合，ユーザは割込みをかけた後で RCX を赤外線エリア内に移動し，コントロール C を押すことによって，フロントエンドと RCX との同期をとる．

4 言語

前節の対話例から明らかなように，XS の言語は，Common Lisp [12] ではなく，Scheme [5] をベースにしている．次の理由によって，Scheme はよりコンパクトな実装が可能だからである．

- Scheme では，関数と変数の名前空間が同一なので，記号オブジェクトには値を格納するスロットを一つだけ用意すればよい．
- Scheme では繰り返しは末尾再帰呼び出しによって実現されるので，繰り返しのための特別な構文を用意する必要がない．これによって，評価器のコード量を削減することができる．

4.1 データ型

XS の言語は，次のデータ型をサポートしている．

- 14 ビット符号つき整数
- コンス
- 関数
  - 組込み関数
  - 関数閉包 (ユーザ定義関数)
- 記号
  - 組込み関数の名前である記号
  - ユーザ定義の記号
- その他
  - 空リスト
  - 真偽値 (`#f` と`#t`)

これらのうち，コンス，関数閉包，ユーザ定義の記号はヒープに生成されるセルによって表現され，他は即値 (immediate data) として表現されている．この設計のために，起動時の RCX ヒープには，オブジェクトが一つも割り当てられていない．データ表現については 6.1 節で詳しく述べる．

整数が 14 ビット長なので，最大で 8,191 ($= 2^{13} - 1$) までの整数しか表現できない．これは，RCX の 1 ワードが 16 ビット長であり，そのうちの 2 ビットを XS がオブジェクトタグとして利用しているためである．さらに，RCX のメモリが小さいので，整数は即値として表現せざるを得なかった．整数を即値ではなく，ヒープに割り当てると，メモリを消費する上に，整数演算のための評価器内のコード量も増加する．

上記のデータに加えて，XS ではプログラム記述に便利なように次の**擬似オブジェクト**を提供している．これらは，フロントエンドのリーダが実際のデータに変換する．

- 文字: ASCII コードの整数に変換される．
  例．`#\a` ⇒ `97`
- 文字列: ASCII コードのリストに変換される．
  例．`"abc"` ⇒ `(97 98 99)`
- 入力時定数: 実際の値があまり重要でない整数を指定する場合に利用する．入力時定数の名前はコロン “:” で始まる．入力時定数の例を次にあげる．
  `:a` (出力ポートの A を指定するために使う)
  `:forward` (モータを前進させるために使う)
  `:white` (光センサが返すもっとも明るい値)
  `:La0` (RCX のサウンドシステムの最も低い音)

4.2 組込み関数

XS が提供している組込み関数 (最上位形式と特殊形式を含む) の一覧を付録にあげる．これらの関数には，大多数の Lisp あるいは Scheme 処理系でもサポートされている一般的なもの (「共通の関数」) と，Lego 用の XS に固有のもの (「固有の関数」) とに分類できる．共通の関数のほとんどは，IEEE Scheme 仕様 [5] に準拠しており，関数のプロファイルを見れば機能は自明であろう．ここでは，若干の補足説明をしておくにとどめる．

XS は，Scheme の一級継続をサポートしていない．その代わり，非局所的脱出を記述するために，Common Lisp の `catch` と `throw` をサポートしている．一級継続の生成は，基本的にスタックの内容をヒープに退避することによって実現される．後述の

ように，XS は 1K バイトの C 言語スタックと，512 バイトの独自スタックを使用しており，合わせると 1.5K バイトになる．これに対して，ヒープは 5K バイトしかない．一級継続はスタックの使用中の部分だけを退避すればよいとはいえ，ヒープにはユーザプログラムや実行時データも格納されるので，一級継続を生成していくと，簡単にヒープが満杯になってしまう．一方，`catch` と `throw` による非局所的脱出は，`catch` 式を実行した時点のスタック位置を記憶するだけで実装でき，ヒープに負担をかけない．

XS の関数 `write` は，Scheme の `display` に相当する．文字列は XS では実際には整数リストなので，関数 `write` は，ユーザが意図したとおりの適切な表示を行えない．この問題を解決するために，XS では `write-string` という関数が追加されていて，

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

が “`(97 98 99)`” と表示するのに対して，

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

は “`"abc"`” と表示する．

Lego 固有の最上位形式 (top-level form) である `last-value` と関数の `linked?`は，信頼性の低い赤外線通信の問題点に対処するためのものである．`last-value` は直前の S 式の値を返す．フロントエンドが直前の S 式をとりこぼした場合に使用する．`linked?`は，RCX がフロントエンドと交信可能かどうかを調べる述語であり，一定時間以内にフロントエンドから返事がなければ偽を返す．

関数 `sleep` は，実行中のプログラムを，指定した時間だけ中断する．中断時間は整数で指定し，単位は 1/10 秒である．RCX はミリ秒単位で制御することが可能であるが，実際に XS のアプリケーションを制御するためには，1/10 秒単位の指定ができれば十分であり，中断時間の上限である 13 分 ($\approx (2^{13}-1)\div 10\div 60$) は，実際的には十分に長いと考えられる．同じ理由で，関数 `time` も「現在の時刻」を 1/10 秒単位の整数として返す．XS のシステムクロックは，RCX 側処理系の起動時にゼロに初期化される．XS の整数長が短いために，`time` の返す値は，約 13 分でオーバーフローする．ユーザが時間間隔を測定する必要がある場合は，開始時間を得る前に，組込み関数の `reset-time` を使ってシステムクロックを初期化することが望ましい．

特殊形式の (`wait-until` *expr*) は，*expr* によって指定されたイベントが発生するまでプログラムの実行を中断する．*expr* は任意の式であり，その値が真になるまで，一定間隔で繰り返し評価される．例えば，

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

とすると，RCX ブロックの Prgm ボタンが押されるまで待つことになる (述語 `pressed?`については下記参照)．

特殊形式の `with-watcher` も，イベントに対処するための機構である．その一般的な書式は次のとおりである．

(`with-watcher` ((*event*$_1$. *handler*$_1$) ⋯ (*event*$_n$. *handler*$_n$)) . *body*)

body の評価中に，指定された *event* が発生しているかどうかを評価器がときどきチェックする (この間隔については後述する)．もしいずれかの *event*$_i$ の値が真になれば，対応する *handler*$_i$ を実行する間，*body* の実行が中断される．*handler*$_i$ の実行中にも，評価器は *event*$_{i+1}$ から *event*$_n$ までのチェックを行い，いずれかの *event*$_j$ ($j > i$) の値が真になると，実行中であった *handler*$_i$ を中断して，*handler*$_j$ の実行を始める．*handler*$_j$ の実行が終了すれば，中断されていた *handler*$_i$ の実行が再開される．つまり，`with-watcher` は，*event* に対して `with-watcher` 式に現れたのと逆の優先度でもって，*handler* の入れ子構造の実行を行う．

XS は，光センサ，角度センサ，温度センサ，タッチセンサの 4 種類のセンサをサポートしている．このうち，光センサと角度センサはアクティブセンサであり，使用前に `light-on` または `rotation-on` を使ってそれぞれ活性化する必要がある．使用が終われば `light-off` または `rotation-off` でオフにする．センサの値は，`light`, `rotation`, `temperature`, `touched?` によって取得する．これらの関数への引数は，センサが接続されているセンサポートの番号である．

先に，with-watcher で指定したイベントが発生したかどうかを，評価器が「ときどき」チェックすると書いたが，具体的には，約 1/10 秒おきにチェックしている．チェックの頻度が高くなると，評価器の実行効率が低下する．逆に間隔が長すぎると，発生したイベントをとりこぼす可能性が高くなる．XS がサポートする 4 種類のセンサのうち，イベントのとりこぼしが最も問題になるのはタッチセンサである．実際に計測してみると，タッチセンサが物に触れると，約 0.25 秒間はセンサ入力がオンになったままである．したがって，1/10 秒おきにセンサ入力をチェックすれば，物に触れたというイベントをとりこぼすことはない．

関数 motor は，指定された出力ポートに接続されているモータの動作を設定する．モータの速度は関数 speed で設定する．これらの関数は第 2 引数を値として返すので，次のような関数呼び出しの入れ子によって複数のモータを制御できる．

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

関数 play は，音の高さと長さを指定した連想リストを受け取って演奏を開始する．音の高さは整数で指定するが，通常は入力時定数の:La0, :La#0, :Si0, :Do1, ..., :So8, :So#8, :La8, :pause を使用する．playing?は，RCX が演奏を終了したかどうかを調べる述語である．述語の pressed?は，Prgm ボタンが押されていれば真を，そうでなければ偽を返す．

関数 puts は，与えられた文字列を最大 5 文字まで LCD ディスプレイに表示する．putc は，指定された文字を，LCD ディスプレイの指定された位置に表示し，cls は LCD ディスプレイ全体をクリアする．最後に，battery は RCX の電池の残量を調べるのに使用する．

5 プログラム例

障害物を回避する「ランドローバー」を制御する簡単なプログラムを図 6 に示す．この関数 rover が呼び出されると，まず PC のディスプレイにメッセージを表示し，ユーザが Prgm ボタンを押してから放すと，名前つき let 式 (named-let 式) によるループに入る．最初，ローバーは音楽を演奏しながら前進し，

```
(define (rover times)
  (write-string "Press Prgm button to start")
  (wait-until (pressed?))
  (wait-until (not (pressed?)))
  (let loop ((n times))
    (motor :a (motor :c :forward))
    (play '((:Re4 . 2) (:Do4 . 1) ...))
    (wait-until (or (touched? 2) (pressed?)))
    (if (touched? 2)
        (begin
          (motor :a (motor :c :back))
          (sleep 5)
          (motor (if (= (random 2) 0) :a :c)
                 :forward)
          (sleep 5)
          (if (> n 0) (loop (- n 1)))))
  )
  (motor :a (motor :c :off))
  )
```

図 6 サンプルプログラム (1).

Prgm ボタンが押されるか，あるいはタッチセンサが障害物に触れるまで待つ．障害物に触れた場合は少し後退してから右あるいは左に旋回し，再び前進する．具体的には，両方のモータを 0.5 秒間後退させ，任意に選択した左右いずれかのモータを 0.5 秒間前進させる．その間，もう一方のモータは後退し続けるので，ローバーはその場で 0.5 秒間旋回することになる．0.5 秒というと短すぎるように思えるが，実際のローバーの動きを見ると，もっと長いように感じる．動作変更の指令を出してからモータが反応するまでの遅延のためであろう．以上の動作を，rover への引数 times で指定された回数だけ繰り返す．ただし途中で Prgm ボタンが押された場合は，直ちに繰り返しを終える．最後に，モータを停止してから関数の実行は終了する．

ローバーのような移動ロボットの制御プログラムをいくつか書いてみると，上のプログラム中の次の機能が，他のプログラムでも共通して使えることに気がつく．

1. RXC のボタン (具体的には Prgm ボタン) を押すことによってローバーの動作を開始する．
2. ローバーの状態にかかわらず．Prgm ボタンを押すことによってローバーを停止させ，プログラムの実行を終了する．

```
(define (start f . args)
  (write-string "Press Prgm button when ready")
  (wait-until (pressed?))
  (wait-until (not (pressed?)))
  (catch 'end
    (with-watcher (((pressed?) (throw 'end 0)))
      (apply f args)))
  (motor :a (motor :c :off))
  )

(define (rover1 times)
  (define (simple-rover n)
    (motor :a (motor :c :forward))
    (play '((:Re4 . 2) (:Do4 . 1) ...))
    (wait-until (touched? 2))
    (motor :a (motor :c :back))
    (sleep 5)
    (motor (if (= (random 2) 0) :a :c)
           :forward)
    (sleep 5)
    (if (> n 0) (simple-rover (- n 1)))
    )
  (start simple-rover times)
  )
```

図 7　サンプルプログラム (2).

3. 制御プログラムからもローバーの停止とプログラムの終了を指示する.

これらの共通機能を汎用の関数として抽出し，図 6 のプログラムを書き直したものを図 7 に示す．関数 `start` が，汎用化した関数であり，引数として与えられた関数 `f` を，`args` を引数リストとして `apply` 式を使って呼び出す．`f` の実行中は，いつでも `throw` 関数を使って `catch` 式から脱出できるし，Prgm ボタンを押すことによっても脱出できる．最後に，モータを停止してから関数 `start` は終了する．二つ目の関数 `rover1` は，図 6 の `rover` と同じ働きを，内部的に定義された関数 `simple-rover` を `start` に受け渡すことによって実現している．共通的な機能を関数 `start` にまかせることにより，`simple-rover` 関数には，ローバーの基本的な制御のみが記述されており，図 6 の `rover` のループよりもすっきりしたものとなっている．この例には，Lisp(と Scheme) をベースとした XS の，高度な制御機能が盛り込まれている．末尾再帰呼び出しの最適化，内部関数の再帰的定義，第一級の関数オブジェクト，可変個引数の受け渡し，非局所的脱出などがそうである．XS は RCX の小さなメモリで動作するシステムであるが，小さいながらも，記述力のきわめて高い言語システムであるといえる．

6　実装

XS システムは，超小型の OS である brickOS [3] の上に構築されている．この OS は，Lego 社が提供するファームウェアを置き換える「カーネル」と，カーネルとユーザプログラムを RCX にダウンロードするためのユーティリティプログラムから構成されている (図 4 参照)．ユーティリティプログラムは，Lego Network Protocol (LNP) を使って赤外線通信を行う．この LNP というプロトコルは送信されたメッセージが必ず受信されることを保証するものではないが，もし受信されればメッセージの正しさは保証する．ユーザプログラムは C で記述し，H8 マイクロプロセッサ用の GNU クロスコンパイラを使ってコンパイルし，ユーティリティプログラムを使ってダウンロードする．GNU クロスコンパイラはフルセットの C 言語で記述したプログラムを処理できるが，実行時ライブラリは brickOS が提供するものに限定される．brickOS 自身は大部分が C 言語で記述され，一部はアセンブリ言語で記述されている．

XS システムは C 言語だけで記述されている．RCX 側の評価器は，brickOS にとってはユーザプログラムである．したがって，GNU クロスコンパイラでコンパイルして brickOS のユーティリティでダウンロードする．フロントエンド処理系は PC 上の通常の C コンパイラでコンパイルする．

フロントエンド処理系には，S 式を読み込むリーダと，S 式をディスプレイに表示するプリンタが含まれる．これらは，通常の Lisp 処理系のものと同様である．入力された S 式は，RCX の評価器に送られる前にプリプロセッサによって前処理される．この前処理では，評価器の負荷をできる限り軽減するためのさまざまな処理を行い，特殊形式の構文チェックやある程度の最適化処理も行う．前処理については，本節で再度言及する．

RCX 側の処理系は，本質的には「receive-eval-return」ループである．ループの最初に，S 式が一つ

到着するのを待つ．到着すると，評価器がこれを評価し，その値をフロントエンドに送り返す．RCX 側の処理系はこの処理を繰り返す．ユーザが “(bye)” と入力するとフロントエンドから終了命令が届き，これによって RCX 側の処理系はループをぬけて実行を終了する．

評価器は，真に末尾再帰的なインタープリタで，前処理された S 式を評価する．評価器自身は C 言語の再帰的関数によって定義されているが，末尾再帰のためのいわゆる「トランポリン」は使用していない．評価器の詳細は本論文の範囲外であり，他の論文にまわすことにする．末尾再帰的なインタープリタは，不要な関数フレームをスタックに残さずに関数呼び出しができるので，小さなメモリ空間で動作する XS のようなシステムには不可欠である．

brickOS のカーネルは，RAM メモリ 32K バイトの約半分を占有している．残り半分を XS が次のように使っている．

- RCX 側処理系のコード: 9.5K バイト†1
- メッセージバッファ: 256 バイト
- C 言語スタック: 1K バイト (= 512 ワード)
- 値スタック: 512 バイト (= 256 ワード)
- ヒープ: 5K バイト (= 1280 セル)
- ビットテーブル: 160 バイト (= 1280 ビット)

C 言語スタックは，RCX 側処理系を実行するために暗黙に使われ，RCX 側処理系が C スタックを直接アクセスすることはない．値スタックは XS の関数に引数を受け渡したり，局所変数を割り当てたり，オブジェクトをごみ集め処理から保護するために使用する．

ヒープは動的なデータ構造を構築するために使われる．ヒープに配置することのできるセルはすべて同じ構造であり，複雑なデータ構造は何個かのセルを結合することによって表現する．これによって小さいヒープ空間を有効に利用している．すべてのセルは 2 ワード (= 4 バイト) の大きさであり，2 ワードバウンダリに align されている．

†1 RCX 側処理系が占めるメモリ領域の大きさを測定する手段がないので，他の情報から推測した値である．バイナリファイルの大きさとほぼ一致する．

すべての未使用セルは 1 本のフリーリストにリンクされている．ユーザプログラムが新しいセルを要求すると，フリーリストからセルが 1 個取り出されて使われる．フリーリストが空であれば，ユーザプログラムを中断し，ごみ集め処理を開始する．ごみ集めは単純なマーク・スイープ法を採用しているが，セルが 1 種類だけであることと，仮想記憶をサポートしていないために，単純だがきわめて効率が良い．ごみでないセルをマークするために，ビットテーブルを使用している．このテーブルにはセルごとに 1 ビット使用しており，現在の 5K バイトのヒープ構成に対しては，テーブルのサイズはわずか 160 バイトである．ロボット制御のためには，実時間，あるいはインクリメンタルなごみ集めが期待されるが，ヒープが小さいことと RCX の CPU が比較的高速であることから，これまでのところ，上記の一括型ごみ集めでも特に問題は生じていない．ごみ集めの実時間性については，6.5 節で改めて議論する．

6.1 オブジェクト表現

XS のオブジェクトがどのように表現されているかを図 8 に示す．すべてのオブジェクトは 16 ビットの

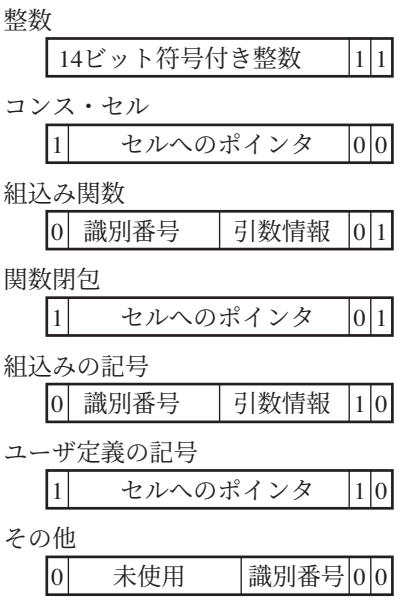

図 8 オブジェクト表現.

ワードで表現されており，下位 (LSB) の 2 ビットはデータ型を判別するためのタグとして使われる．オブジェクトの最上位ビット (MSB) も，整数以外はタグとして使われており，ヒープに割り当てられるオブジェクトは 1，その他の即値データは 0 である．LSB タグは，関数なら 01，記号なら 10 である．

RCX の RAM メモリは，16 ビットのアドレス空間の上位に配置されているので，セルのアドレスの MSB は常に 1 である．セルは 2 ワードバウンダリに align されているので，セルのアドレスの下位 2 ビットは常に 00 である．したがって，コンスオブジェクトの表現は，セルへのポインタそのものである．(図中の「セルへのポインタ」は，MSB と 2 ビットの LSB を除いたものである.) これによって，リスト処理が簡単化されている．例えば，関数 `car` は受け取った引数をセルポインタとみなし，そのポインタが指すオブジェクトを単純に返すだけである (図 9 (a) 参照)．一方，関数閉包とユーザ定義の記号に対しては，セルへのポインタは LSB 2 ビットをクリアすることによってポインタを得る．

組込み関数の表現には，識別番号と引数情報が含まれている．識別番号は，組込み関数を特定するための数値である．6 ビットの引数情報は，最低限必要な引数の個数 (2 ビット) と，引数の個数の最大値 (3 ビット)，関数が任意個の引数を受け取れるかどうかのフラッグ (1 ビット) から構成されている．評価器が組込み関数を呼び出す際に，まず引数情報に基づいて引数の個数をチェックし，識別番号にしたがって関数の実行コードにジャンプする．

組込みの記号は，その記号を名前とする組込み関数を値として格納している．Scheme と同様，値が未定義あるいは値が組込み関数以外である組込みの記号は存在しない．さらに，XS では組込みの記号は値を変更できないこととした．つまり，組込みの記号を名前とする大域変数への代入は許されない．組込みの記号と組込み関数の 1 対 1 に関係するので，組込みの記号は LSB タグを除けば，対応する組込み関数と同じ表現となっている．組込みの記号の値 (組込みの記号を名前とする組込み関数オブジェクト) を得るには，単に LSB タグを 10 から 01 に置き換えるだけでよい．

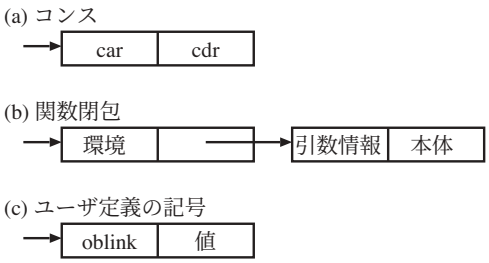

図 9　セルの解釈.

6.2　記号

ユーザ定義の記号は LSB タグが 10 であるセルポインタによって表現される．ポインタが指すセルは，図 9 (c) に図示するように解釈される．oblink フィールドはすべてのユーザ定義の記号をリンクするためのものであり，ごみ集めから記号を保護するために使われる．現在の実装では，ユーザ定義の記号はいったん生成されると決してごみとして回収されることはない．値フィールドは記号の値，つまりその記号を名前とする大域変数の値を格納する．値が未定義の場合は，値フィールドには「未定義」という特殊なデータが格納されている．記号オブジェクトは LSB タグが 10 なので，セルの第 2 フィールドへのポインタとみなすことができる．したがって，記号オブジェクトはそのままで値フィールドをアクセスするために使用できる．

記号名はフロントエンドが記憶している．フロントエンドは**記号表**というハッシュ表を持っており，これによって記号名から記号オブジェクトを取り出す．フロントエンドのリーダが記号トークンに出会うと，記号表を使ってトークンを記号オブジェクトに変換する．もし記号が未登録であれば，リーダは新しい記号オブジェクトを生成するよう RCX 側処理系に指示し，生成された記号オブジェクトを記号表に登録する．局所変数のためだけに使われる記号トークンに対しては，この「インターン」の処理は不要である．局所変数への参照は，前処理によって，変数が割り当てられる値スタック上の位置への参照に置き換えられ

るためである．

記号を表示するためにも，ユーザ定義の記号から記号名を得るためのハッシュ表を使う．組込み記号の名前は，対応する組込み関数の識別番号をインデックスとする別の表に格納されている．

6.3 関数閉包

前処理によって，ラムダ式

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

は，

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

に変換される．ここで *arg-info* は，組込み関数オブジェクトに収納されている引数情報と同様であり，*body* は *original-body* を前処理した結果である．評価器がこのラムダ式に対する新しい閉包を生成するときは，最初のセルを新しいセルで置き換え，その新しいセルの car 部に関数閉包生成時の環境を格納する (図 9 (b) 参照)．ラムダ式の残りの部分は同じラムダ式から生成されるすべての閉包で共有する．したがって，新しい閉包の生成はセルを 1 個消費するだけである．もちろん，閉包に保存される環境を構成するためにはさらに何個かのセルを必要とする．

局所変数 (関数へのパラメータを含む) は値スタックに配置される．前処理によって，局所変数への参照は，関数フレームのベースアドレスから相対的な位置への参照に置き換えられる．評価器が関数閉包を呼び出すときは，まず関数閉包を値スタックにプッシュし，次に引数を順にプッシュする．その後，評価器が関数本体の実行を開始する．その他の局所変数 (`let` 式などによって束縛される変数) も値スタックに配置される．変数を束縛する式が評価されると，局所変数の初期値が値スタックにプッシュされ，束縛する式 (タイプフェイス式など) の実行が終了するときにポップされる．

環境はヒープ中のセルをリンクすることによって構成する．関数閉包が呼び出された時点では，関数閉包に保存された環境が環境の初期値として使われる．局所変数がスタックに配置されるとき，もし閉包に保存される可能性があればヒープに「リフト」される．つまり，新しいセルが割り当てられて，その car 部が局所変数の値を格納するために使われ，cdr 部はその時点の環境にリンクされる．リフトされた変数値を高速にアクセスできるように，セルへのポインタがスタック上の変数位置に格納される．

フロントエンドは，上記の実装に必要なすべての静的解析を前処理として行い，適切な情報を S 式に埋め込んで RCX に受け渡す．まず，前処理によってリフトを行う命令が S 式の適切な箇所に挿入される．リフト命令のオペランドは，リフトすべき変数の場所 (ベースポインタ相対) と，リフト時の環境が格納されている場所 (同) である．次に，前処理によって局所変数への参照が次のいずれかの命令に置き換えられる．

- スタック位置への参照
- スタック位置から参照されているセルの car 部
- 環境として閉包に格納されているリストの要素への参照

最初の二つの命令に対するオペランドはスタック位置であり，三つめの命令に対してはリストへのインデックスがオペランドとなる．さらに，前処理によって，関数閉包に保存する環境の場所を，ラムダ式の記号 `lambda` の中に埋め込む．このようにすべての静的な情報は前処理によって準備されるので，実行時に評価器が行うのは単純な処理だけである．

6.4 通信

フロントエンドは，前処理済みの S 式をバイト列として RCX に送信する．受け手の RCX は，RCX ヒープのセルを使って受け取ったバイト列から S 式を復元する．XS は，バイト列として，一種の後置記法を採用している．例えば，リスト “`(1 2)`” は次の 5 バイトの列として送信される．

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

ここで，low(x) と high(x) は，オブジェクト x の下位バイトと上位バイトをそれぞれ表す．`list2` は，先行する二つのオブジェクトを要素とするリストを生成するためのコマンドである．受け手はまず上記の第 1 バイトの LSB タグを見て，整数の下位バイトであることが分かる．そこで，第 2 バイトを整数の上位バイトと解釈し，整数 (この例では `1`) を再構成して値

スタックにプッシュする．同様に，第 3 バイトと第 4 バイトから整数の 2 をプッシュする．五つめのバイトはコマンド `list2` なので，スタックから二つのオブジェクトをポップし，これら二つのオブジェクトを要素とするリストを生成し，結果をスタックにプッシュしなおして次の処理に進む．

バイト列中のコマンドは，図 8 の「その他のオブジェクト」として表現されている．「その他のオブジェクト」の上位バイトは MSB 以外は使用されず，ここを使ってコンスオブジェクトと区別する．コンスオブジェクトは受け手が生成するものであり，それ自体がバイト列として送信されることはない．そこで，下位バイトだけを送るだけで，受け手はそれが「その他のオブジェクト」であることが分かる．その識別番号フィールドを見ることで，それが空リストや真偽値といった通常のオブジェクトを表現するものなのか，あるいはコマンドなのかを判別することができる．

コマンドの中には，受け手に評価を依頼するものがある．例えば，定義 “`(define bar (foo 1))`” は次の 9 バイトの列として渡される．

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

ここで，low(`foo`) は，記号 `foo` のオブジェクト表現の下位バイトである (記号は前もってフロントエンドのリーダがインターン済みである)．上のメッセージを受け取ると，まず最初の 5 バイトからリスト “`(foo 1)`” を構築し，それをスタックにプッシュする．`eval` コマンドによって，受け手はこのリストをポップし，それを S 式として評価するために評価器を呼び出し，結果をスタックにプッシュする．次に記号オブジェクト `bar` をプッシュする．最後にコマンド `define` によって，変数名を表す記号と初期値をポップして変数を定義する．

RCX のメッセージバッファの容量は限られている (現在の実装では 256 バイト) ので，複雑な S 式を 1 回の通信で送信できないことがある．その場合は，S 式を複数のバイト列に分割して送信する．そのようなバイト列は `cont` というコマンドで終わり，バイト列がさらに続くことを表す．

RCX 側処理系がフロントエンドに値を返すときも同様の方法をとる．唯一の相違は，返す値が循環構造を持つ可能性がある点である．循環構造をフロントエンドで再構成できる機構を実装することもできたが，コード量にみあうだけの価値があるかどうか疑問だったのでサポートしないことにした．そこで，循環構造をエンコードしてしまうと無限に通信を繰り返すことになる．これを避けるために，RCX 側処理系は十分に多いバイトを送信すると自動的に送信を停止するようにした．単純に送信バイト数をカウントし，ヒープ中のセルの総数に基づいて計算された上限値を超えないかどうかをチェックするだけの簡単なコードである．

6.5 性能

以上のように実装した XS について，簡単な性能測定を行ったので報告する．まず実行性能については，Lisp ベンチマークの定番である tak [4] の実行時間を測定した．結果は，228.1 秒であった．XS はインタープリタでプログラムを実行するが，局所変数はスタックに配置するので tak の場合はセルを消費せず，実行中にごみ集めは起動されない．通常の PC 上であれば，どんなに遅い処理系でも数秒程度で tak の実行が完了するので，XS の実行速度は相当遅いといえる．にもかかわらず，実際にプログラムを起動してロボットを制御させると，それほど遅いという印象が無い．これは，プログラムが CPU バウンドではなく，CPU と比較すると反応がさらに遅いセンサやモータを制御するためのものだからである．高度に知的な制御を行えば，当然プログラムは CPU バウンドになっていくが，もともと小さいメモリと 8 ビットの CPU しか備えていない RCX でそのようなプログラムを期待するのは無理がある．

XS が提供するヒープ領域は 5K バイト，セル数にして 1280 個分である．この小さいヒープは，`cons` や `list` が生成するコンスオブジェクトの他に，ユーザプログラムと，実行時に生成される関数閉包のための環境を表現するのにも使われる．関数閉包の環境を構成するセルは一般にはあまり多くないので，ユーザプログラムを表現するために必要なセル数が問題となる．しかし，Lisp が高水準言語であることと，フ

ロントエンドによる前処理やヒープ内での表現を工夫したことによって，ユーザプログラムの表現に消費するセル数は，少なく抑えられている．例えば，図 6 のプログラムは 133 セル，図 7 のプログラムは 145 セルで表現されている．実行時にコンスオブジェクトや関数閉包を生成しないアプリケーションであれば，これらのプログラムの約 10 倍程度のサイズであっても実行できることになる．XS 配布のための Web ページ [16] には，赤外線タワーを追尾するプログラム `range.lsp` や，床に書いた曲線に沿って移動するプログラム `trace.lsp` をサンプルとして掲載しているが，これらを表現するセル数も，それぞれ 315 個および 154 個と，機能のわりには少なく抑えられている．

XS は，マーク・スイープ法によってごみ集めを行っているので，ごみ集めに要する時間は，使用中の (ごみでない) セルの個数に依存する．実際に時間を計測してみると，使用中セルがまったく無い場合は 0.177 秒，ほぼ全セルが使用中の場合は 0.421 秒であった．この差が，約 1200 個のセルをマークするために要する時間であり，マークのための時間は，使用中のセルの個数にほぼ比例する．組込み関数の `with-watcher` の説明のところで述べたように，タッチセンサの情報は，約 0.25 秒でクリアされる．したがって，ごみ集めが起動されるタイミングによっては，タッチセンサからのイベントをとりこぼす可能性がある．また，モータの駆動がごみ集めによって遅れる可能性も考えられる．現状でごみ集めによる停止時間が問題になっていないのは，ごみ集めの起動頻度が少ないためだと考えられる．実時間ごみ集めの導入が望ましいが，そのためには処理系のコード量の増加が避けられない．ごみ集め処理がアプリケーションの実行と平行して進行するので，使用中のセルを誤ってごみとして回収しないための機構を処理系の随所に埋め込む必要があるためである [15]．実時間ごみ集めの採用は，この処理系サイズの増加を考慮しながら，慎重に検討していきたい．

7 おわりに

XS のプロジェクトは，まず本稿の筆者が，シリアルタワー (PC のシリアルポートに接続する赤外線タワー) を利用する RCX モデル 1.0 および 1.5 に対して，Linux 版プロトタイプを legOS 上に開発することから始まった．その時点ですでに legOS の後継である brickOS が存在したが，legOS と比較して brickOS は規模が大きいために，より多くのメモリ領域を利用できる legOS を採用した．筆者には Windows のプログラミング経験がとぼしいので，Linux 版プロトタイプが完成した時点で，Windows 上への移植を米国 Franz 社に委託した．当時，Cygwin 対応の legOS が存在したが，Windows 上では直接動作していなかった．このために Franz 社では，Windows に対応していた brickOS を使用し，そこから XS に不要な機能 (マルチスレッド機能など) を削除することによって，当初の legOS よりもコンパクトな brickOS を構築し，適宜修正を加えることによって XS を動作させることに成功した．

USB タワーを利用するモデル 2.0 に対しては，Windows 用には USB タワーのドライバが Lego 社から提供されていたので，比較的早期に XS を稼動させることができた．Linux 用には当初ドライバがなかったが，XS の配布を開始して間もなく，最新バージョンの Linux には MindStorms 用の USB タワードライバが標準で装備していることが判明した．この USB タワーは Lego MindStorms 以外にはおそらく使われていないので，MindStorms がいかに広く普及しているかが分かる．

現在 XS は，RCX の三つすべてのモデルに対して，Windows 版と Linux 版が完成しており，Redhat Linux，Debian Linux，Windows XP などの上で動作が確認されている．平成 18 年度からは，京都大学工学部情報学科の演習用教材として利用されており [7]，他のいくつかの大学でも学部生向けの実験・演習の教材として検討を進めている．

XS は，オープンソースのソフトウェアとして公開している [16]．マニュアルや発表資料などのドキュメント類も，この Web ページからダウンロード可能である．

謝辞

XS の開発は，情報処理振興事業協会 (IPA) の未踏

プロジェクトとして実施された．未踏プロジェクトマネジャーの近山隆氏からは有益な助言をいただいた．Franz 社での移植作業は，John Foderaro 氏が中心となって行った．また，同社の Sheng-Chuan Wu 氏には，XS の英語版マニュアルの草稿をチェックしていただいた．Franz 社の Fritz Kunze 社長には，このプロジェクトを進めるにあたってさまざまな情報提供と助言をしていただいた．これらの方々に心より感謝の意を伝えたい．

参 考 文 献

[1] Baum, D.: *Extreme Mindstorms: an Advanced Guide to Lego Mindstorms*, Apress, 2000.
[2] Baum, D.: *Definitive Guide to LEGO Mindstorms*, Second Edition, Apress, 2003.
[3] brickOS at SourceForge, http://brickos.sourceforge.net/.
[4] Gabriel, R. P.: *Performance and Evaluation of Lisp System*, MIT Press Series in Computer Science, MIT Press, Cambridge, MA, 1985.
[5] IEEE Standard for the Scheme Programming Language, IEEE, 1991.
[6] Klassner, F. and Anderson, S.: Lego Mindstorms: Not Just for K-12 Anymore, IEEE Robotics and Automation Magazine, 2003.
[7] 京都大学工学部情報学科計算機科学コース計算機室: 計算機科学実験及演習 4 ―ロボットプログラミング, http://www.kuis.kyoto-u.ac.jp/isle/.
[8] LEGO.com Mindstorms Home, http://mindstorms.lego.com/.
[9] legOS Alternative Lego OS, http://sourceforge.net/projects/legos.
[10] Nielsson, S.: Introduction to the legOS kernel, http://legos.sourceforge.net/docs/kerneldoc.pdf, 2000.
[11] Proudfoot, K.: RCX Internals, http://graphics.stanford.edu/~kekoa/rcx/, 1998.
[12] Steele, G.: *Common Lisp the Language*, Second Edition, Digital Press, 1990.
[13] Villa, L.: legOS HOWTO, http://legos.sourceforge.net/HOWTO/, 2000.
[14] Wick, A., Klipsch, K. and Wagner, M.: Lego/Scheme Compiler V0.5.2, http://www.cs.indiana.edu/~mtwagner/legoscheme/.
[15] Yuasa, T.: Real-time Garbage Collection on General-purpose Machines, *Journal of Systems and Software*, Vol.11, No.3 (1990), pp.181–198.
[16] Yuasa, T.: XS: Lisp on Lego MindStorms, http://www.xslisp.com/.

A XS 関数一覧

XS が提供する全関数（特殊形式と最上位形式を含む）を，それらの関数プロファイルとともに以下に列挙する．次の記法を用いる．

X*: 0 個以上の X
X^+: 1 個以上の X
$\{X_1|\cdots|X_n\}$: X_1〜X_n のいずれか
[X]: 省略可能な X

特殊形式と最上位形式の名前には下線をつける．関数プロファイル $X_1, X_2, \ldots, X_n$ が関数名を除いて同一のときは，まず X_1 を書いて，そのあとに $X_2, \ldots, X_n$ の関数名を書くことがある．例えば，`or` の関数プロファイルは (`or` *expr*) である．セミコロン（;）以降はコメントである．なお，`list*`から `reverse` までの七つのリスト処理関数は，インストール時のオプションである．

Lisp に共通の関数

- 最上位形式
  (`define` *sym form*)
  (`define` (*sym sym** [. *sym*]) *form**)
  (`load` *file-name*)
  (`trace` *function-name*) と `untrace`
  (`bye`)
- 基本的な特殊形式
  (`quote` *obj*)
  (`set!` *sym form*)
  (`lambda` (*sym** [. *sym*]) *form**)
- 制御
  (`begin` *form**)
  (`apply` *fun obj** *list*)
  (`if` *form form* [*form*])
  (`catch` *form form**)
  (`throw` *obj obj*)
- 条件
  (`and` *form**) と `or`
  (`not` *obj*)
- 束縛
  (`let` [*sym*] ((*sym form*)*) *form**)

(let* ((*sym form*)*) *form**) と letrec

- 型述語

(boolean? *obj*) と integer?, null?, pair?, symbol?, function?

- 比較

(eq? *obj obj*)

(< *int*$^+$) と>, =, >=, <=

- 算術演算

(+ *int**)

(- *int int**)

(* *int**)

(/ *int int*) と remainder

(logand *int int*) と logior, logxor, logshl, logshr

(random *int*)

- リスト処理

(car *cons*) と cdr

(cons *obj obj*)

(set-car! *cons obj*) と set-cdr!

(list *obj**)

(list* *obj** *obj*)

(list-ref *list int*)

(append *list** *obj*)

(assoc *obj a-list*)

(member *obj list*)

(length *list*) と reverse

- 入出力

(read [*int*]) と read-char, read-line

(write *obj* [*int*])

(write-char *char* [*int*])

(write-string *string* [*int*])

- ごみ集め

(gc)

XS 固有の関数

- 最上位形式

(last-value)

(fork *sym sym string*$^+$) ; Linux 版のみ

- 制御

(sleep *int*) ; 単位は 1/10 秒

(wait-until *form*) ; イベント待ち

(with-watcher ((*form form**)*) *form**)
; 非同期イベント割り込み

- システムクロック

(time) ; 値は 1/10 秒単位

(reset-time)

- 赤外線通信のテスト

(linked?)

- 光センサ（引数の数値はポート番号）

(light-on {1|2|3}) と light-off

(light {1|2|3}) ; 値は 0(黒)〜98(白)

- 角度センサ（引数の数値はポート番号）

(rotation-on {1|2|3}) と rotation-off

(rotation {1|2|3}) ; 単位は 360/16 度

- 温度センサ（引数の数値はポート番号）

(temperature {1|2|3}) ; 単位は摂氏

- タッチセンサ（引数の数値はポート番号）

(touched? {1|2|3})

- モータ（引数の:a, :b, :c はポート名）

(motor {:a|:b|:c}
{:off|:forward|:back|:brake})

(speed {:a|:b|:c} *speed*)
; $0 \leq speed \leq 255$ (:max-speed)

- サウンド

(play ((*pitch* . *length*)*))

(playing?)

- Prgm ボタン

(pressed?)

- LCD ディスプレイ

(puts *string*)

(putc *char column*)

(cls)

- 電池残量

(battery) ; 単位は 1/10 ボルト