XSのマニュアル

湯淺太一

平成18年 11月 2日

目 次

1 はじめに

XS は，Lego MindStorms Robotics Invention System (RIS) のためのプログラミングシステムです．RIS の中心となるのは，8 ビット CPU を搭載した RCX というプログラム可能なブロックです．RCX で動作するプログラムは，フロントエンド PC で準備し，赤外線通信によって，RCX にダウンロードします．モータやセンサなどのデバイスを取り付けることによって，自分でロボットをデザインして組み立て，それを自分で作った RCX プログラムによって制御することができます． XS は，RIS のための対話型プログラミング環境を提供するために開発されました．RCX やそれに接続されたデバイスを制御することができる Lisp 言語が利用できます．この言語は，次の特徴をもっています．

1. read-eval-print ループ
2. 関数の対話的定義および再定義
3. 適切なエラーメッセージとバックトレース機能
4. 関数呼び出しの trace と untrace 機能
5. オブジェクトの動的生成とごみ集め
6. プログラムエラーやスタックオーバーフローなどに対して頑健
7. 端末機割込み
8. 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，非同期のイベント監視機能
10. モータ，センサ，ランプ，サウンドなどの RCX 機能へのインターフェイス

最後の三つ以外の特徴は，通常の Lisp 処理系に共通するものです．8 番目の特徴は通常の Scheme インタープリタに共通するものであり，9 番目の特徴は，同等の機能がマルチスレッド対応の Lisp 処理系にみられます．したがって，ユーザにとっては，XS は RCX の諸機能を制御するための拡張を行った通常の Lisp とみなせます．

XS の処理系は，PC で動作するフロントエンド部分と，RCX で動作する評価器から構成されています．これら二つの部分処理系が協調することによって，通常の Lisp 処理系と同様の対話型プログラミング環境を提供しています．

図 1に，XS の簡単な使用例を示します．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します．プロンプトに続けてユーザが関数定義を入力し（4 行目），テストします（7 行目）．この関数定義は未定義の変数 `nil` を参照しているので，エラーメッセージ（8 行目）

```
 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  %
```

図 1: XS の使用例（説明のために行番号をつけています）.

とバックトレース（9 行目）が表示されます．そこでユーザは，変数 nil を定義してその値を空リストに設定し（10 行目），関数をもう一度テストします（12 行目）．今回は，関数は正しい値を返す（13 行目）ので，ユーザは満足して XS の利用を終了します（14 行目）．この例には，通常の Lisp 処理系と異なることは特にありません．しかし内部的には，通常の Lisp 処理系とはかなり異なった処理が行われています．評価器が PC にではなく，RCX にあるからです．

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ます．まだ準備ができていなければ（例えば，RCX の電源を入れ忘れていたりすると），フロントエンドはユーザとの対話をあきらめます．

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

PC のキーボードから式が入力されると，フロントエンドはその式を前処理し，結果を RCX に送ります．評価器はそれを評価し，値をフロントエンドに送り返し，それが PC のディスプレイに表示されます．関数定義を行う式の場合は，評価器は関数をインストールし，関数の名前である記号を返します．

実行中のプログラムを中断するには，Control-C を入力（Ctrl キーを押しながら C のキーを押す）します．次の対話例では，let 式が無限ループを実行しているときに，ユーザが Control-C を入力しています．

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

フロントエンドからの端末機割込みの要求も，赤外線通信を使って RCX に送信されます．もし RCX が赤外線の有効範囲の外に出ていれば，割込み要求を受け取ることができません．そのような場合に備えて，XS は端末機割込みの方法をもう一つ用意しています．RCX ブロックの View ボタンを押せばよいのです．もし RCX が赤外線の有効範囲に入っていれば，View ボタンを押すことは，Control-C を入力することと効果はまったく同じです．しかし範囲外であれば，フロントエンドは RCX からの戻り値を待ち続けます．View ボタンによって割込みが起きたことを赤外線通信によって知ることができないからです．このような場合は，プログラムを中断した後で，RCX を赤外線の有効範囲に移動し，Control-C を入力してください．

2 表記法と用語

以下の章で，XS 言語の構成要素（関数など）を一つずつ説明していきます．それぞれの言語要素の記述は次のいずれかの「ヘッダ」で始まります．

syntax	[最上位形式]
syntax	[特殊形式]
(*function-name argument-specification*)	[関数]
constant-name	[入力時定数]

最初の二つは，それぞれ**最上位形式**（top-level form）および**特殊形式**（special form）の場合に使用します．これらの形式は，条件分岐や変数束縛といった XS の構文要素を与えるもので，それぞれの構文要素の文法がヘッダに記載されています．もし文法に従わない最上位形式あるいは特殊形式が入力された場合は，エラーが発生します．

三つ目は，組込み関数のためのものです．この形のヘッダは，組込み関数の名前で始まり，関数が受け取ることができる引数の個数と型（下記参照）が続きます．原則として，関数はヘッダに記された条件を満たす引数を受け取った場合に限り，説明文のとおりに動作します．そうでなければ，エラーが発生します．この原則に従わない例外的な場合もありますが，その場合は，関数の説明の中に

明記します．最後の形は，入力時定数（3.8節参照）のためのもので，入力時定数の名前だけからなります．

このマニュアルでは，次の文法記法を用います．

構文変数（非終端記号）は，イタリック体（*italic*）で表します．
終端記号はタイプフェイス（`type-face`）で表します．
“*thing**” は，*thing* が 0 個以上現れることを意味します．
“[*thing*]” は，*thing* が多くとも 1 個現れることを意味します．

ここで，*thing* は構文変数であるか，または “(” で始まり “)” で終わる構文パターンです．

組込み関数のヘッダでは，引数の仕様を次の記法を使って表します．

1. 引数の型は，イタリック体（*italic*）で表します．受け取ることのできる引数の型を表すと同時に，関数の説明の中で，受け取った引数の値を表すためにも使用します．引数の型としては，次のものだけ（場合によって，添え字がつくことがあります）を使います．

   obj：任意の XS オブジェクト
   int：整数
   sym：記号
   cons：コンス
   list：真のリスト（つまり，空リスト以外のリスト）
   fun：関数

2. “$class_1 \ldots class_n$” は，*class* 型の n 個の引数を表します．そのような引数として受け取ることができる個数 n は，“$n \geq 0$” または “$n \geq 1$” の形で，ヘッダに与えます．

3 オブジェクトと字句構造

XS には，次の六つの型のオブジェクトがあります．XS の各オブジェクトは，これらいずれか一つの型に属します．

真偽値
整数
記号
空リスト
コンス
関数

これらに加えて，プログラミングに便利なように次の「擬似オブジェクト」を利用することができます．

文字
文字列

3.1 真偽値

真偽値オブジェクトには，`#t` と `#f` があります． これらは，真か偽かを関数が返すときに値として使われ，`#t` は真を，`#f` は偽を表します．

3.2 整数

整数は 2 進，8 進，10 進，16 進のいずれかで入力し，それぞれ次のように表記します．

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

ここで，

sign は “`+`” または “`-`”，
binary-digit は `0` または `1`，
octal-digit は `0`, `1`, . . ., `7` のいずれか，
decimal-digit は `0`, `1`, . . ., `9` のいずれか，
hexadecimal-digit は `0`, `1`, . . ., `9`, `a`, . . ., `f` のいずれかです．

いずれの形でも，少なくとも 1 個の digit が必要です．

整数オブジェクトは符号付き 14 ビット整数を表現します．したがって，XS では -2^{13} $(= -8192)$ 以上 $2^{13} - 1$ $(= 8191)$ 以下の整数を扱うことができます．

`:most-positive-integer` [入力時定数]
`:most-negative-integer` [入力時定数]

これらの入力時定数は，XS が扱える最大の整数 `8191` $(= 2^{13} - 1)$ と，
最小の整数 `-8192` $(= -2^{13})$ をそれぞれ表します．

3.3 記号

記号は，**名前** (**記号名**あるいは**印字名**ともいいます) によって一意的に指定できるオブジェクトです．記号名は，1 個以上の文字からなる文字列です．

記号の名前が次の条件をすべて満たす場合，記号はその記号名によって指定することができます．

1. 記号名が次の文字のみから構成されている．

```
! # $ % & * + - . / 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. 記号名が “:” または “#” で始まらない.

3. 記号名が 1 個のドット “.” だけからなる文字列でない.

4. 記号名が整数の構文にマッチしない.

記号名がこれらの条件のいずれかを満たさない場合は，記号名を縦棒 “|” で囲んで記号を表します.

もし記号名が縦棒を含んでいれば，縦棒の直前にバックスラッシュ “\” をつけます．もし記号名がバックスラッシュを含んでいれば，その直前にもう一つのバックスラッシュをつけます.

記号名における大文字と小文字は区別されます．例えば，baz と BAZ は異なる記号を表します.

組込み関数の名前である記号を，**組込みの記号**と呼びます．その他の記号は**ユーザ定義の記号**です．XS システムが起動した直後は，システムの中には組込みの記号しか存在しません．ユーザ定義の記号は後で生成されます．例えば，最上位形式 define を使って新しい大域変数を定義すると，その変数の名前となる新しい記号が生成されます.

Common Lisp をはじめとするいくつかの Lisp では，“NIL” という名前の記号は空リストや偽値を表すことがありますが，XS では “NIL” という名前の記号は，組込みの記号ではありません.

3.4 コンスとリスト

コンスは，リストや木構造といったデータ構造を構成するために主に使われます．それぞれのコンスは，car および cdr と呼ばれる二つのオブジェクトを持っています．コンスオブジェクトは次のように表記します.

$(x_1 \ . \ x_2)$

ここで，x_1 と x_2 は，それぞれ car および cdr を表します.

空リストとは，要素を持たないリストのことで，() と表記します．XS システムには，空リストはただ一つしか存在しません．したがって，() と書くと，常に同じオブジェクトを表すことになります.

cdr をたどっていくと空リストで終わるデータ構造を**リスト**と呼びます．より正確には，リストは次のように再帰的に定義されます.

1. 空リストはリストである.

2. cdr がリストであるコンスはリストである.

3. 上記以外はリストでない.

リストは次のように表記します.

$$(x_1\ x_2\ \ldots\ x_n)$$

これは,

$$(x_1\ .\ (x_2\ .\ (\ \ldots\ .\ (x_n\ .\ ())\ \ldots\ )))$$

と等価です. ここで, n はリストの**長さ**であり, 各 x_i $(1 \leq i \leq n)$ は, リストの i 番目の**要素**と呼ばれます.

cdr をたどっていくと空リスト以外のオブジェクトで終わるデータ構造を**ドット・リスト**と呼びます. ドット・リストは次のように表記します.

$$(x_1\ x_2\ \ldots\ x_{n-1}\ .\ x_n)$$

これは,

$$(x_1\ .\ (x_2\ .\ (\ \ldots\ .\ (x_{n-1}\ .\ x_n)\ \ldots\ )))$$

と等価です.

3.5 関数

関数オブジェクトは, いくつかのオブジェクトを**引数**として**受け取り**, 何らかのオブジェクトを**値**として**返し**ます. 関数に引数を受け渡し, 関数から値を受け取る一連の操作を, **関数呼び出し**といいます. 呼び出すことができるオブジェクトは関数オブジェクトだけです. 記号やラムダリスト（lambda という記号で始まるリスト）は呼び出すことができません.

XS にはさまざまな**組込み関数**が用意されています. このマニュアルでは, そのすべてを説明しています. 組込み関数のいくつかは, **インストール時オプション**です. そのような関数は, 他の組込み関数を使って定義可能です. したがって, RCX のメモリを節約したい場合は, これらの関数をあなたの XS システムにインストールしておかなくても構いません. インストールしていない場合は, これらの関数を使用する前に, lib.lsp をロードしてください. このファイルには, インストール時オプションであるすべての組込み関数の定義が格納されていて, あなたが使う XS システムと同じディレクトリに置かれています.

自分で関数を定義することもできます. そのような**ユーザ定義関数**は, lambda 式を評価することによって生成されます.

3.6 文字

文字オブジェクトは，計算機が操作できる文字を表します．XS では，文字を表現するための特別なデータを用意していません．XS のリーダが文字オブジェクトを読み込むと，対応する ASCII コードに変換します．

通常の文字オブジェクトは

#\$char$

と表記します．ここで，$char$ は，その文字オブジェクトが表現する文字です．XS では，次の文字を利用可能です．

```
! " # $ % & ' ( ) * + , - . /
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
{ | } ~
```

いくつかの「特殊文字」は

#\$name$

と表記します．ここで，$name$ は，特殊文字の「名前」です．XS では，次の特殊文字を使うことができます．

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

3.7 文字列

文字列は，文字の並びです．XS が扱える任意の文字を文字列の要素とすることができます．文字列は，要素となる文字列を順に並べ，ダブルクオート “"” で囲むことによって表します．文字列がダブルクオートを要素とする場合は，ダブルクオートの直前にバックスラッシュ “\” をつけます．文字列がバックスラッシュを要素とする場合は，そのバックスラッシュの直前に，もう一つのバックスラッシュをつけます．

XS では，文字列を表すための特別なデータを用意していません．XS のリーダが文字列データを読み込むと，ASCII コードのリストに変換します．

3.8 入力時定数

入力時定数は記号のように見えますが，XS のリーダはこれらを整数に変換します．入力時定数は XS システムに組込みのもので，ユーザが自分の入力時定数を定義することはできません．個々の入力時定数には，“:” で始まる名前がついていて，あらかじめ決められた値に変換されます．

3.9 コメント

入力行がセミコロン “;” を含んでおり，それが何らかのオブジェクトのテキスト表現の一部でない場合，そのセミコロンとその行の以降の文字はコメントとみなされます．

4 評価

4.1 式

式は，評価することのできるオブジェクトです．式によっては，XS プログラムのトップレベルだけに許されるものがあります．そのような式を**最上位形式**と呼びます．最上位形式は，次のいずれかの記号で始まります[1]．

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

このマニュアルでは，最上位形式でない式のことを，**通常の式**と呼ぶことにしますが，混乱のおそれがない場合は，単に**式**と呼ぶこともあります．

XS の（通常の）式には，次のものがあります．

1. 特殊形式
2. 関数呼び出し式
3. 変数
4. リテラル

式は，それが評価されると，一つのオブジェクトを値として返します．

特殊形式

次のいずれかの記号で始まるリストは特殊形式です．

[1] 現在，最上位形式の `fork` は，Linux 版の XS のみで利用可能です．

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

これらの記号は，**特殊形式の名前**と呼ばれます．特殊形式の評価の方法は，先頭の記号によって決まります．

関数呼び出し式

関数呼び出し式は特殊形式の名前以外の記号で始まるリストです．関数呼び出し式が評価されると，第一要素がまず評価されます．その値は関数オブジェクトでなければなりません．次に，関数呼び出し式の第一要素を除いた残りの要素が一つずつ，左から右へ評価されます．そして，これらの値が関数へ引数として渡されます．関数が返す値が，関数呼び出し式の値となります．

XS の評価器は，**真に末尾再帰的**なインタープリタです．ある関数が別の関数を呼び出すときに，呼び出される関数が返す値を，呼び出し側の関数が他の処理をしないでそのまま自分自身の値として返す場合，その関数呼び出しは，**末尾再帰的**と呼ばれます．例えば，

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

において，関数本体の `ints` の呼び出しは末尾再帰的ではありません．呼び出し側（この例では，呼び出される関数とたまたま同じですが）が，`ints` がリターンした後に，`cons` を呼び出すからです．一方，次の定義における `ints` の呼び出しは末尾再帰的です．

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

処理系が真に末尾再帰的であるというのは，すべての末尾再帰的な呼び出しを適切なジャンプに置き換えることによって，末尾再帰的な呼び出しによるスタックの消費を抑えることを意味します．

変数

変数は記号によって表されます．XS のすべての変数は，いつでも**値**を一つだけ持っています．

記号が式として評価されると，その記号が表す変数の値が返ります．その記号を名前とする変数が存在しない場合はエラーが生じます．

Scheme と同様に，XS では変数と関数のための名前空間は一つだけです．名前のついた関数は，関数名と同じ名前の変数の値として格納されています．例えば，組込み関数の cons は，cons という名前の変数の値として格納されています．(cons 1 2) という式は，組込み関数の cons を呼び出しますが，それは，まず変数 cons を評価すると，組込み関数 cons の関数オブジェクトが得られるからです．

実際，組込み関数の名前を入力すると，関数オブジェクトが返ります．

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

ユーザ定義の変数は，set!式で値を変更することができます．しかし，組込み関数に対応する変数は，値を変更できません．

リテラル

コンス（リストを含む）でも記号でもないオブジェクトは，リテラルです．リテラルが評価されると，それ自身が値として返されます．

前述のように，XS における文字列は，入力時に整数のリストに変換されます．大多数の Lisp では，文字列はリテラルなので，XS においても文字列をリテラルとして扱うことが期待されます．例えば，次のように書けることが望ましいです．

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

これを，

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

と書くと，Lisp プログラマにとっては奇妙に見えます．このために，XS のリーダは，文字列が式とみなされる場所に現れた場合，その文字列を quote 式に埋め込みます．したがって，上の二つの define 式は，いずれも次のように変換されます．

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

4.2 環境

式の評価は，評価時点の**環境**に依存します．環境は 3 種類の束縛からなります．

- 変数束縛
  変数束縛は，変数とその名前の組です．変数の名前は記号です．

- キャッチャー束縛
  キャッチャー束縛は，キャッチャーとその名前の組です．キャッチャーの名前は任意のオブジェクトです．キャッチャーは `catch` 式で生成され，非局所的脱出（8.3節参照）に利用します．

- ウォッチャー束縛
  ウォッチャーは条件とハンドラの組です．条件とハンドラはどちらも式です．ウォッチャーは `with-watcher` 式で生成され，非同期のイベント処理（下記参照）に用います．

XS プログラムの実行中に複数の評価環境が同時に存在することがあります．しかし，式の評価に使われるのはそのなかの一つだけで，それを**現在の環境**と呼んでいます．

最初,「現在の環境」は組込み関数の束縛（組込み関数を値とする変数の束縛）だけからなります.「現在の環境」に束縛が一つ追加されると，新しい環境が生成され，それが「現在の環境」として使われます．追加された束縛の有効範囲が終了すると，その束縛は解除され，古い環境が再び「現在の環境」になります．通常の式（最上位形式でない式）が束縛を追加すると，その式の実行が終了するときに自動的に束縛が解除されます．一方，最上位形式の `define` 式が束縛を追加すると，その束縛は現在の XS セッションが終了するまで残ります．組込み変数の束縛と `define` 式で追加された束縛を総称して，**大域変数束縛**と呼びます．キャッチャーとウォッチャーには大域束縛はありません．

関数オブジェクトが生成されるときに,「現在の環境」の一部が関数オブジェクトの中に保存されます．環境のうちで，関数オブジェクトに保存される部分は**静的環境**と呼び，それ以外を**動的環境**と呼びます．関数が呼び出されると，現在の動的環境（つまり,「現在の環境」のうちの動的環境）と関数オブジェクトに保存されている静的環境を組み合わせて環境を構成します．この環境が，関数呼び出し時点における「現在の環境」として使われます．関数がリターンするとき,「現在の環境」は，呼び出し前の環境に戻ります．静的環境は，通常の（最上位形式でない）式によって追加された変数束縛（**静的束縛**）からなります．他の束縛は**動的束縛**です．

束縛が静的か動的かについての規則をまとめておきます．

- 組込み関数の束縛は大域的で，動的です．
- `define` 式による変数束縛は大域的で，動的です．
- 上記以外の変数束縛は静的です．
- キャッチャーとウォッチャーの束縛は動的です．

「現在の環境」は，式の評価に次のように使用されます．

- 記号 s が変数名として使用されるとき，s と同じ名前の変数束縛を現在の環境の中に探します．もしそのような束縛が二つ以上存在したら，最後に追加された束縛が使われます．もしそのような束縛が見つからなかったら，エラーが発生します．選択された変数束縛の変数が，記号 s の特定する変数として使われます．

- あるオブジェクト x が，`throw` 式の中でキャッチャーとして指定されているとき，x と名前が（`eq?`の意味で）同じキャッチャー束縛が「現在の環境」から取り出されます．そのようなキャッチャー束縛が複数存在すれば，最後に追加された束縛が取り出されます．`throw` 式の実行によって，取り出されたキャッチャーからリターンします．もしそのような束縛が存在しなければ，エラーが発生します．

- XS システムは時々（概ね 100 ミリ秒ごとに）ウォッチャー環境をチェックし，条件が満たされるウォッチャー束縛が存在するかどうか調べます．もし存在すれば，システムは現在のプログラム実行を一時的に中断し，ウォッチャーのハンドラを実行し始めます．そのようなウォッチャーが複数存在する場合は，最後に追加されたものが選択されます．ハンドラの実行が終了すると，中断されていたプログラム実行が再開されます．

4.3 ラムダ式

関数は**ラムダ式**によって定義されます．ラムダ式とは，記号 `lambda` で始まるリストです．

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

ここで，*form* の並びは，関数の**本体**です．ラムダ式は特殊形式なので，評価することができます．評価されると，そのラムダ式は，現在の静的環境を保持する新しい関数オブジェクトを生成します．

ラムダ式の中の *lambda-list* は，関数が受け取れるパラメータを指定します．*lambda-list* の一般形は次のとおりです．

(*sym** [. *sym*])

関数が呼び出されると，それぞれの *sym* に対応する変数が束縛されます．これらの束縛は，関数がリターンするときに解除されます．ラムダリストによって生成される変数は，**パラメータ**と呼ばれます．パラメータの初期値は，関数に渡された実引数によって決定します．

1. ドット “.” の後ろにある記号以外の記号は，**必須パラメータ**の名前です．関数が呼び出されるときに，これらのパラメータに対応する実引数が必ず与えられなければなりません．つまり，ラムダリストの中に指定されてい

る必須パラメータが n 個のとき，少なくとも n 個の実引数が与えられなければなりません．i 番目（$1 \leq i \leq n$）の実引数が，i 番目の必須パラメータの値となります．実引数の個数が少なすぎて，必須パラメータに対応する実引数が無いときは，エラーが発生します．

2. ドット “.” とその直後の記号が与えられたとき，記号は**残余パラメータ**を表します．必須の実引数以外のすべての実引数が 1 本のリストになり，そのリストが残余パラメータの値となります．必須パラメータの個数が n であるとき，$n+i$ 番目の実引数がリストの i 番目の要素になります．残余パラメータが指定されていない場合，必須パラメータに対応しない実引数があると，エラーが発生します．

最上位形式あるいは特殊形式と同じ名前のパラメータを指定してはいけませんが，組込み関数と同じ名前のパラメータを指定しても構いません．

関数の本体は，0 個以上の式の並びです．関数が呼び出されると，パラメータの束縛が追加された後に，本体の式が左から右へ順に評価されます．そして，最後の式の値が，関数の値として返されます．もし本体に式が一つもなければ，関数は空リスト () を返します．このマニュアルでは，“*form**” と書く代わりに，“. *body*” と書き,「*body* を評価してその値を返す」と表現することがあります．これは次のことを意味します．

> *form* の並びを左から右へ順に評価します．最後の *form*（がもしあれば）以外の *form* の値は破棄して，最後の *form* の値を *body* の値として返します．もし *form* が一つも与えられなければ，*body* は単に () を返します．

この記法を使えば，ラムダ式の構文は次のように表せます．

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

5 最上位形式

`(define` *sym form*`)` [最上位形式]

> *form* を評価し，*sym* という名前の大域変数束縛を追加します．*form* の値が，その変数の初期値となります．変数の値は，`set!`式を使って変更することができます．記号 *sym* は，最上位形式，特殊形式，あるいは組込み関数の名前と一致してはいけません．

`(define (`*sym* . *lambda-list*`)` . *body*`)` [最上位形式]

> 次のラムダ式で定義される関数を生成します．

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

そして，*sym* という名前の大域的束縛を追加し，生成された関数がその値となります．この最上位形式は，次の式と等価です．

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

記号 *sym* は，最上位形式，特殊形式，あるいは組込み関数の名前と一致してはいけません．

(load *string*) [最上位形式]

string という名前のファイルからプログラムをロードします．ファイルの中身は，式の並びでなければなりません．ファイルからロードするということは，ファイル中の各式を PC のキーボードから順に入力するのと同じ結果となります．

(fork sym_1 sym_2 $string_1$ $string_2$ ... $string_n$) $(n \geq 1)$ [最上位形式]

フロントエンド処理系のサブプロセスを一つ生成し，パス $string_1$ によって指定されるプログラムを実行します．その他の文字列（つまり，$string_2, \ldots, string_n$）は，プログラムへ引数として受け渡されます．この最上位形式は，sym_1 と sym_2 という二つの大域的束縛を追加し，生成されるサブプロセスの標準入力へ書き出すためのポート番号および標準出力から読み込むためのポート番号を，それぞれの初期値とします．これらのポート番号は，XS の入出力関数（read や write）を使ってサブプロセスと交信するために利用します．

この最上位形式は，現在，Linux 版の XS だけで利用可能です．

(trace *sym*) [最上位形式]

sym という関数のトレースを開始します．ユーザ定義関数だけがトレース可能です．トレースが指定されている関数が呼び出されると，関数名，実引数，戻り値が表示されます．この最上位形式は，関数名 *sym* を返します．関数のトレースは，最上位形式の untrace を使って解除することができます．

(untrace *sym*) [最上位形式]

sym という関数のトレースを終了します．指定された関数が実際にトレースされていれば関数名 *sym* を返し，そうでなければ，#f を返します．

(last-value) [最上位形式]

最後に入力された式の値を返します．この最上位形式は，フロントエンドが何らかの理由によってRCXからの値を受け取れなかった場合に役に立ちます．最後に入力された式の評価がエラーで終わっていれば，エラーメッセージとバックトレースがPCのディスプレイに表示されます．

(bye) [最上位形式]

XSの実行を終了します．RCXの評価器も停止しますから，XSを再起動するときは，必ずRunボタンを押してください．

6 述語

6.1 型述語

(boolean? *obj*) [関数]

obj が真偽値オブジェクトであれば#tを返し，そうでなければ#fを返します．

(integer? *obj*) [関数]

obj が整数オブジェクトであれば#tを返し，そうでなければ#fを返します．

(null? *obj*) [関数]

obj が空リストであれば#tを返し，そうでなければ#fを返します．

(pair? *obj*) [関数]

obj がコンスオブジェクトであれば#tを返し，そうでなければ#fを返します．

(symbol? *obj*) [関数]

obj が記号オブジェクトであれば#tを返し，そうでなければ#fを返します．

(function? *obj*) [関数]

obj が関数オブジェクトであれば#tを返し，そうでなければ#fを返します．

6.2 論理演算

(not *obj*) [関数]

obj が#f であれば#t を返し，それ以外なら#f を返します.

(and *form**) [特殊形式]

各 *form* を左から右へ順に評価し，いずれかの値が#f であれば，and 式は#f を返します．どの *form* の値も#f 以外であれば，最後の *form* の値を返します．*form* が一つも与えられていなければ，and 式は#t を返します.

(or *form**) [特殊形式]

各 *form* を左から右へ順に評価し，いずれかの値が#f 以外であれば，or 式はその（#f 以外の）値を返します．すべての *form* の値が#f であれば，#f を返します．*form* が一つも与えられていなければ，or 式は#f を返します.

6.3 等号

(eq? obj_1 obj_2) [関数]

obj_1 と obj_2 が同一のオブジェクトであれば#t を返し，そうでなければ#f を返します.

オブジェクトが同一であるかどうかを判定するために，XS は次の規則を使います.

- (), #t, #f は，それぞれ一つずつしか存在しません.
- ある整数を表す整数オブジェクトは，ただ一つしか存在しません.
- ある名前の記号は，ただ一つしか存在しません.
- ある名前の組込み関数は，ただ一つしか存在しません.
- ラムダ式は，それが評価されるたびに，それまでに生成されたどの関数オブジェクトとも異なる新しい関数オブジェクトを生成します.
- read-eval-print ループや関数 read が読み込んだコンスオブジェクトは，それまでに生成されたどのコンスオブジェクトとも異なります．また，cons などの関数によって生成されるコンスオブジェクトも，それまでに生成されたどのコンスオブジェクトとも異なります.

ある値 x を初期値とした変数束縛が追加されると，set!式によって変更されるまで，その変数の値は x と同一です．set!式によって変数にある値 y が代入されると，次に set!式によって変更されるまで，その変数の値は y と同一です．

同様に，コンスオブジェクトの car（あるいは cdr）フィールドの値は，set-car!（あるいは set-cdr!）によって置き換えられるまで，同一です．つまり，一つのコンスオブジェクトに car（あるいは cdr）を適用すると，その結果は，set-car!（あるいは set-cdr!）によって変更されるまでは，同一のオブジェクトを返します．

7 変数と関数

(quote *obj*) [特殊形式]

単に *obj* を返します．(quote *obj*) は，'*obj* と書くこともできます．

(set! *sym form*) [特殊形式]

form を評価し，その値を *sym* という名前の変数に代入します．*form* の値を，set!式の値として返します．

(lambda *lambda-list* . *body*) [特殊形式]

ラムダ式によって定義される関数（4.3節参照）を生成して返します．

(let [*name*] ((*sym form*)*) . *body*) [特殊形式]

すべての *form* を左から右へ順に評価し，各 *sym* ごとに，対応する *form* の値を初期値とする変数束縛を追加します．そして，*body* を評価してその値を返します．追加した変数束縛は，let 式を抜けるときに解除されます．

let 式

(let ((*x*$_1$ *form*$_1$) ... (*x*$_n$ *form*$_n$)) . *body*)

は，次の式と等価です．

((lambda (*x*$_1$ ... *x*$_n$) . *body*) *form*$_1$ ... *form*$_n$)

オプションの *name*（これは記号です）が指定されると，let 式はまず *name* という名前の変数束縛を追加します．その初期値は，次のラムダ式によって定義される関数です．

(lambda (*x*$_1$ ... *x*$_n$) . *body*)

つまり，let 式

(let $name$ ((x_1 $form_1$) ... (x_n $form_n$)) . $body$)

は，次の式と等価です.

(let (($name$ (lambda (x_1 ... x_n) . $body$)))
  ($name$ $form_1$ ... $form_n$))

(let* ((sym $form$)*) . $body$) [特殊形式]

最初の $form$ を評価し，その値を初期値とし，最初の sym を名前とする変数束縛を追加します．同様の処理を，sym と $form$ のそれぞれの組に対して左から右へ順に行います．そして，$body$ を評価してその値を返します．設定した変数束縛は，let*式の終了時に解除されます．sym と $form$ の組が一つも与えられなかった場合，let*式

(let* () . $body$)

は，次の式と等価です.

(begin . $body$)

その他の場合，let*式

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

は，次の式と等価です.

(let ((x_1 $form_1$))
  (let* ((x_2 $form_2$) ... (x_n $form_n$)) . $body$))

(letrec ((sym $form$)*) . $body$) [特殊形式]

各 sym に対して，sym を名前とし，初期値が () である変数束縛を追加します．そして，すべての $form$ を左から右へ順に評価し，それぞれの値を，対応する sym を名前とする変数に代入します．これらの変数束縛は，letrec 式から抜けるときに解除されます.

letrec 式

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

は次の式と等価です.

(let ((x_1 ()) ... (x_n ()))
  (set! x_1 $form_1$) ... (set! x_n $form_n$) . $body$)

この特殊形式は，局所的な再帰的（自己再帰でも相互再帰でも）関数を定義するために用います．*form* がラムダ式のとき，それらのラムダ式が定義する関数は，相互に呼び出すことが可能です．これは，追加された束縛を関数本体で参照できるからです．

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

指定された関数 *fun* を呼び出し，関数の返す値を返します．各 obj_i $(1 \leq i \leq n)$ が関数への i 番目の引数となり，最後の引数 *list* の j 番目の要素が，$n + j$ 番目の引数となります．

(trace-call *sym fun list*) [関数]

この関数は，関数トレース機能を実装するために，処理系が内部的に使用するものです．ユーザはこの関数を直接的に使用しないほうがよいです．

この関数は，関数 *fun* を呼び出し，それが返す値を返します．*list* の要素が関数への引数として使われます．i 番目の要素が i 番目の引数となります．さらに trace-call は，その関数の名前が *sym* であると仮定して，呼び出しに関する情報を表示します．

8 制御構造

8.1 逐次実行

(begin *form**) [特殊形式]

form を左から右へ順に評価し，最後の *form* の値を begin 式の値として返します．*form* が一つも与えられていないとき，begin は単に () を返します．

8.2 条件分岐

(if $form_1$ $form_2$ [$form_3$]) [特殊形式]

$form_1$ を評価し，その値が#f 以外であれば，$form_2$ を評価してその値を返します．$form_1$ の値が#f なら，$form_3$（デフォルトは ()）を評価してその値を返します．

8.3 非局所的脱出

(catch *form* . *body*) [特殊形式]

form を評価し，その値を名前とするキャッチャー束縛を追加します．そして *body* を評価してその値を返します．catch 式から抜けるときに，設定した束縛を解除します．catch 式の実行中に，設定されたキャッチャーの名前を第 1 引数として throw 式が呼び出されると，catch 式の実行は直ちに終了し，throw への第 2 引数が catch 式の値として使われます．

(throw obj_1 obj_2) [関数]

名前が obj_1 であるキャッチャーがあれば，そのキャッチャー束縛を追加した catch 式の実行を直ちに終了し，obj_2 を catch 式の値として返します．もし obj_1 を名前とするキャッチャーが存在しなければ，エラーが発生します．

8.4 タイミング

(sleep *int*) [関数]

プログラムの実行を，指定した時間だけ中断します．*int* は正の整数で，中断時間を 1/10 秒単位で指定します．例えば，引数が 5 なら 0.5 秒間，50 なら 5 秒間中断します．この関数は，指定した時間が経過すると，引数の *int* を値としてリターンします．

(time) [関数]

現在の時刻を，1/10 秒単位で返します．XS のシステムクロックは，RCX 側のサブシステムが起動されたときにゼロに初期化されます．XS は大きな整数を表現できないので，time の値は，およそ 13 分でオーバーフローします．したがって，この関数を使って経過時間を計測する場合，開始時刻を取得する前に，関数 reset-time を使ってシステムクロックをゼロに戻しておくことが望ましいです．

(reset-time) [関数]

システムクロックをゼロに再設定し，その上で現在の時刻を返します．システムクロックがリセットされるので，通常，この関数はゼロを返します．

(wait-until *form*) [特殊形式]

expr で指定した事象が発生するまで待ちます．*form* は任意の式であり，その値が真となるまで，繰り返し評価されます．例えば，

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

とすると，RCXのPrgmボタンが押されるまで待ちます．`wait-until` 式は，*form* が返す値（これは`#f` 以外の値）を返します．

(`with-watcher` ((*form* *form**)*) . *body*) [特殊形式]

指定された事象が発生するのを待つウォッチャーを設定します．*body* の実行中に事象が発生すると，ウォッチャーは *body* の実行を一時的に中断し，対応するハンドラを実行します．一般形は次のとおりです．

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

body を評価している間に，指定された *event* を評価器が定期的に評価します．もしいずれかの *event*$_i$ の値が真になれば，*body* の実行が中断され，対応する *handler*$_i$ が実行されます．*handler*$_i$ の実行中であっても，評価器は *event*$_{i+1}$ 以降 *event*$_n$ までを定期的にチェックし，いずれかの *event*$_j$ $(j > i)$ が真になれば，実行中の *handler*$_i$ を中断し，*handler*$_j$ を実行します．*handler*$_j$ の実行が終了すると，中断されていた *handler*$_i$ の実行が再開されます．つまり，*event* の優先順位は左から右へ高くなり，この優先順位に従って，`with-watcher` は，*handler* を入れ子的に呼び出すことができます．すべてのハンドラの実行が終了すると，中断されていた本体の実行が再開されます．

9 整数演算

9.1 比較

(`=` *int*$_1$ ... *int*$_n$) $(n \geq 1)$ [関数]

すべての *int* が等しければ`#t` を返し，そうでなければ`#f` を返します．

(`<` *int*$_1$ ... *int*$_n$) $(n \geq 1)$ [関数]

すべての *int*$_i$ $(1 \leq i < n)$ が *int*$_{i+1}$ よりも小さければ`#t` を返し，そうでなければ`#f` を返します．

(`<=` *int*$_1$ ... *int*$_n$) $(n \geq 1)$ [関数]

すべての *int*$_i$ $(1 \leq i < n)$ が *int*$_{i+1}$ 以下であれば`#t` を返し，そうでなければ`#f` を返します．

(`>` *int*$_1$ ... *int*$_n$) $(n \geq 1)$ [関数]

すべての int_i $(1 \leq i < n)$ が int_{i+1} より大きければ`#t`を返し，そうでなければ`#f`を返します．

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

すべての int_i $(1 \leq i < n)$ が int_{i+1} 以上であれば`#t`を返し，そうでなければ`#f`を返します．

9.2 算術演算

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

すべての int の和を返します．無引数であれば`0`を返します．

XS はオーバーフローのチェックをしません．もし計算結果が整数オブジェクトの 14 ビット表現の範囲を超えてしまったら，予期しない値を返すことになります．

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

1 引数の場合は，その値の正負を反転したものを返します．複数の引数が与えられた場合は，第 1 引数の int_1 から，$int_2, \ldots, int_n$ を引いた値を返します．

XS はオーバーフローのチェックをしません．もし計算結果が整数オブジェクトの 14 ビット表現の範囲を超えてしまったら，予期しない値を返すことになります．

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

すべての int の積を返します．無引数であれば`1`を返します．

XS はオーバーフローのチェックをしません．もし計算結果が整数オブジェクトの 14 ビット表現の範囲を超えてしまったら，予期しない値を返すことになります．

(`/` int_1 int_2) [関数]

int_1 を int_2 で割り，その商を整数として返します．結果は整数です．第 2 引数の int_2 は`0`であってはなりません．戻り値の正確な定義については，`remainder`の説明を参照してください．

(`remainder` int_1 int_2) [関数]

int_1 を int_2 で割り，その余りを返します．第 2 引数の int_2 は`0`であってはなりません．

結果は整数であり，その絶対値は int_2 の絶対値より小さく，次の条件を満たします．

$$int_1 = \texttt{(+ (* (/ } int_1 \ int_2\texttt{) } int_2\texttt{) (remainder } int_1 \ int_2\texttt{))}$$

結果の符号は，割られる数 int_1 と同じです.

9.3 ビット演算

(logand int_1 int_2) [関数]

int_1 と int_2 のビットごとの論理積を返します.

(logior int_1 int_2) [関数]

int_1 と int_2 のビットごとの論理和を返します.

(logxor int_1 int_2) [関数]

int_1 と int_2 のビットごとの排他的論理和を返します.

(logshl int_1 int_2) [関数]

int_1 を int_2 ビットだけ左へシフトした結果を返します.

(logshr int_1 int_2) [関数]

int_1 を int_2 ビットだけ右へシフトした結果を返します.

9.4 乱数

(random int) [関数]

0 以上 int 未満の擬似乱数を返します．結果は整数です.

10 リスト処理

(cons obj_1 obj_2) [関数]

car 部が obj_1，cdr 部が obj_2 であるコンスオブジェクトを生成して返します.

(car $cons$) [関数]

$cons$ の car 部の値を返します.

(cdr $cons$) [関数]

$cons$ の cdr 部の値を返します.

(set-car! $cons$ obj) [関数]

cons の car 部の値を *obj* で置き換えます．戻り値は *obj* です．

(set-cdr! *cons* *obj*) [関数]

cons の cdr 部の値を *obj* で置き換えます．戻り値は *obj* です．

(length *list*) [関数]

list の長さを整数として返します．

(list-ref *list* *int*) [関数]

list の $int-1$ 番目の要素を返します．引数の *int* は，非負整数でなければなりません．もし *int* が *list* の長さ以上であれば，この関数は，() を返します．

(list obj_1 ... obj_n) $(n \geq 0)$ [関数]

長さが n，i 番目 $(1 \leq i \leq n)$ の要素が obj_i であるリストを生成して返します．

(list* obj_1 ... obj_n) $(n \geq 1)$ [関数]

$n-1$ 個のコンス $c_1, \ldots, c_{n-1}$ を使ったドットリストを生成して返します．各 c_i の car 部は obj_i，cdr 部は c_{i+1} です．ただし，最後の c_{n-1} については，その cdr 部は obj_n です．

この関数はインストール時オプション（3.5節参照）です．

(append $list_1$ ... $list_n$ *obj*) $(n \geq 0)$ [関数]

最後の引数 *obj* の前に，各 *list* の全要素をコンスしたオブジェクトを生成して返します．

この関数はインストール時オプション（3.5節参照）です．

(member *obj* *list*) [関数]

obj がリストの要素であるかどうかを調べます．もし要素であれば，それで始まる部分リストを返します．そうでなければ，#f を返します．

この関数はインストール時オプション（3.5節参照）です．

(assoc *obj* *list*) [関数]

リストの要素のなかに，car 部が *obj* であるコンスが含まれるかどうかを調べます．もしそのようなコンスがあればそれを返し，なければ#f を返します．*list* の各要素はコンスオブジェクトでなければなりません．

この関数はインストール時オプション（3.5節参照）です．

(reverse *list*) [関数]

list の要素を反転したリストを生成して返します.

この関数はインストール時オプション（3.5節参照）です.

11 入出力

:stdin [入力時定数]

:stdout [入力時定数]

:stderr [入力時定数]

これらの入力時定数は，XS の入出力関数（read や write）を呼び出す際に，標準ポートを明示的に指定するために使用します．:stdin はフロントエンド処理系の標準入力を表し，その値は 0 です．:stdout はフロントエンド処理系の標準出力を表し，その値は 1 です．:stdin はフロントエンド処理系のエラー出力を表し，その値は 2 です.

(read [*int*]) [関数]

PC のキーボードからオブジェクトを一つ読み込んでそれを返します. もしオブジェクトを読み込むまでに入力の終わりに達したらエラーを発生します.

オプションの *int* が指定され，その値が:stdin と異なる場合，この関数は PC キーボードの代わりに，指定されたポートから読み込みます.

(read-char [*int*]) [関数]

PC のキーボードから文字を一つ読み込み，その文字コードを整数として返します．すでに入力の終わりに達していたらエラーを発生します.

オプションの *int* が指定され，その値が:stdin と異なる場合，この関数は PC キーボードの代わりに，指定されたポートから読み込みます.

(read-line [*int*]) [関数]

PC のキーボードから 1 行分の文字列を読み込み，文字コードのリストとして返します．行の最後の改行文字は捨てられます．1 行読み込んでいる間に入力の終わりに達したら，それまでに読み込んだ文字列を返します.

オプションの *int* が指定され，その値が:stdin と異なる場合，この関数は PC キーボードの代わりに，指定されたポートから読み込みます．

(write *obj* [*int*]) [関数]

obj を PC のディスプレイに表示します．*obj* を返します．

writeは，オブジェクトを次のように表示します．ここで，$\overline{x}$は，writeによる x の表示を意味します．

- (), #t, #f はそれぞれ “()”, “#t”, “#f” と表示します．
- 整数は，10 進で表示します．
- 記号は，縦棒などのエスケープなしに表示します．
- リスト $(x_1\ \ldots\ x_n)$ は，“$(\overline{x_1}\ \ldots\ \overline{x_n})$” と表示します．ドットリスト $(x_1\ \ldots\ x_{n-1}\ .\ x_n)$ は，“$(\overline{x_1}\ \ldots\ \overline{x_{n-1}}\ .\ \overline{x_n})$” と表示します．いずれの場合も，連続する二つの $\overline{x}$ の間および $\overline{x}$ とドット ‘.’ の間には空白を出力します．
- 関数は，“#<function *name*>” または “#<function>” と表示します．ここで，*name* は関数の名前です．

オプションの *int* が指定され，その値が:stdout と異なる場合，この関数は PC ディスプレイの代わりに，指定されたポートへ書き出します．

(write-char int_1 $[int]_2$) [関数]

int_1 を文字コードとする文字を表示します．int_1 を返します．例えば，

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

は “a” と表示し，

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

は “97” と表示します．

オプションの int_2 が指定され，その値が:stdout と異なる場合，この関数は PC ディスプレイの代わりに，指定されたポートへ書き出します．

(write-string *list* [*int*]) [関数]

list を文字列とみなして表示します．*list* の要素はすべて整数でなければなりません．これらの要素が，順に文字として表示されます．この関数は，*list* を値として返します．例えば，

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

は “abc” と表示し，

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

は “(97 98 99)” と表示します．

オプションの *int* が指定され，その値が:stdout と異なる場合，この関数は PC ディスプレイの代わりに，指定されたポートへ書き出します．

12 RCX の制御

12.1 サウンド

(play *list*) [関数]

list で表現された曲を演奏します．*list* の各要素は，音の高さと長さの組（*pitch* . *length*）です．ここで *pitch* は，0（入力時定数:La0 および:A0 の値）以上 96（:La8 および:A8 の値）以下の整数であるか，あるいは整数 97（入力時定数:pause の値）でなければなりません．*length* は音の長さを表す正の整数です．RCX は，*list* の要素が指定する音を，左から右へ順に鳴らします．

この関数は，曲の演奏が始まると，その終了を待たずに直ちにリターンします．戻り値は通常は () です．しかし，*list* が長すぎて全部を演奏できない場合は，演奏できない部分を *list* の部分リストとして返します．

RCX が曲を演奏している間にこの関数が呼び出されると，演奏中の曲を中断し，新しい曲の演奏を始めます．

(playing?) [関数]

RCX が曲の演奏中であれば#t を，そうでなければ#f を返します．この関数は，RCX が直前の曲の演奏を終了したかどうかを判定するために使用します．

:La0, :La#0, ..., :So#8, :La8 [入力時定数]

これらの入力時定数は，関数 play に受け渡す音の高さを指定するために使用します．それぞれの定数名は，次の形です．

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

```
                                                        :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
```

図 2: 音の高さを表す入力時定数（1）

basic-note は

```
Do  Re  Mi  Fa  So  La  Si
```

のいずれかで，順にド，レ，ミ，ファ，ソ，ラ，シを表します．シャープ記号は半音上がることを意味します．

octave は，何オクターブ目であるかを表し，0（最も低い）から 8（最も高い）までの数字です．図 2に，この形の入力時定数の一覧をあげます．最初の定数:La0 の値は 0 であり，次の:La0#の値は 1，等々であり，最後の:La8 の値は 96 です．

:A0, :Am0, ..., :Gm8, :A8 [入力時定数]

これらの入力時定数も，関数 play に受け渡す音の高さを指定するために使用します．それぞれの定数名は，次の形です．

:*basic-note*[m]*octave*

basic-note は，

```
A  H  C  D  E  F  G
```

のいずれかで，順にラ，シ，ド，レ，ミ，ファ，ソに対応します．オプションの “m” は，半音上がることを意味します．

octave は，何オクターブ目であるかを表し，0（最も低い）から 8（最も高い）までの数字です．図 3に，この形式の入力時定数の一覧をあげます．図中の各定数の値は，図 2の同じ位置にある定数と同じです．

:pause [入力時定数]

```
                                                :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
```

図 3: 音の高さを表す入力時定数（2）

この入力時定は，関数 play に休止（ポーズ）を指定するために使用します．この定数の値は 97 です．

12.2 ボタン

RCX には，四つのボタン View, Prgm, On-Off, Run がついています．そのうち，View ボタンは端末機割込み（第 1章参照）のために使用し，Prgm ボタンは実行中のプログラムに信号を伝えるために使用します．

(pressed?) [関数]

Prgm ボタンが押された直後であれば#t を返し，そうでなければ#f を返します．例えば，次の式を評価すると，Prgm ボタンが押されるまで待つことになります．

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

この関数と，タッチセンサが何かに触れたかどうかを判定するための touched?とを混同しないように注意しましょう．

12.3 LCD ディスプレイ

(puts *list*) [関数]

list を LCD ディスプレイに表示します．*list* の要素は整数で，これらが順に文字として表示されます．この関数の返す値は，通常は () です．LCD には，最大 5 文字を表示することができます．もし *list* が長すぎれば，最初の 5 文字だけが表示され，puts は () の代わりに，表示されなかった *list* の部分文字列を返します．

(putc int_1 int_2) [関数]

LCD ディスプレイの指定されたカラムに int_1 を文字として表示します．カラムは int_2 によって指定し，これは 0（右端）から 4（左端）までの整数でなければなりません．この関数は，int_1 を値として返します．

(cls) [関数]

LCD ディスプレイの表示をクリアします．() を返します．

12.4 赤外線通信の状態

(linked?) [関数]

RCX が現在フロントエンドと通信可能であれが#t を返し，そうでなければ#f を返します．

12.5 バッテリ残量

(battery) [関数]

現在のバッテリ残量を，1/10 ボルト単位の整数として返します．例えば，バッテリ残量が 8.5 ボルトであれば，この関数は 85 を返します．RCX は 6 本の単三乾電池で動作するので，この関数が返す最大の値は約 90 です．

13 デバイス

13.1 モータ

モータは，RCX の出力ポート A〜C に接続することができます．XS プログラムでは，出力ポートは整数で指定します．A ポートは 1，B ポートは 2，C ポートは 3 です．次の入力時定数を使うと便利です．

:a [入力時定数]
:b [入力時定数]
:c [入力時定数]

これらの入力時定数は，出力ポートを指定するために使用します．:a は A ポート，:b は B ポート，:c は C ポートを意味します．実際の値は，それぞれ 1，2，3 です．

モータの動作は，方向と速度の組で決定されます．方向には 4 種類あり，次の入力時定数で指定できます．

`:off` [入力時定数]

`:forward` [入力時定数]

`:back` [入力時定数]

`:brake` [入力時定数]

これらの入力時定数は，モータの方向を指定するために使います．`:forward` は前進，`:back` は後退です．「前進」と「後退」の意味は，モータがどのように接続されているかに依存します．もしモータが思った方向に進まない場合は，モータを逆向きに取り付けなおして，再度試してみてください．`:off` と `:brake` はどちらも停止を意味しますが，`:brake` はモータにブレーキをかけるのに対して，`:off` はモータをアイドル状態にします．これらの定数の値は，`0` (`:off`)，`1` (`:forward`)，`2` (`:back`)，`3` (`:brake`) です．

モータの速度は，`0` から `255` までの整数です．速度を `0` にすると，モータの方向が `:forward` または `:back` であっても，モータは回転しません．また，モータの方向が `:off` または `:brake` のときは，モータ速度がゼロ以外であっても，モータは回転しません．

`:max-speed` [入力時定数]

これはモータの最大速度を表し，値は `255` です．

次の関数は，モータの方向と速度を変更するものです．

(`motor` int_1 int_2) [関数]

出力ポート int_1 に接続されたモータの方向を int_2 にします．第 2 引数の int_2 を値とするので，複数のモータを設定するには，次のような入れ子の式を使うとよいです．

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

ポート番号 int_1 は，`1`，`2`，`3` のいずれかでなければなりません．

(`speed` int_1 int_2) [関数]

出力ポート int_1 に接続されたモータの速度を int_2 にします．第 2 引数の int_2 を値とするので，複数のモータを設定するには，次のような入れ子の式を使うとよいです．

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

ポート番号 int_1 は，`1`，`2`，`3` のいずれかでなければなりません．

13.2 光センサ

光センサは能動的センサなので，使用する前に活性化し，使用が終わったら不活性化する必要があります．

(light-on *int*) [関数]

センサポート *int* に接続された光センサを活性化します．ポート番号は 1, 2, 3 のいずれかです．活性化されると，光センサの赤い LED が点灯します．この関数は，*int* を返します．

(light-off *int*) [関数]

センサポート *int* に接続された光センサを不活性化します．ポート番号は 1, 2, 3 のいずれかです．不活性化されると，光センサの赤い LED が消えます．この関数は，*int* を返します．

光センサは，感知している明るさを，0（最も暗い）から 98 [2]（最も明るい）までの整数として返します．

:white [入力時定数]
:black [入力時定数]

入力時定数の:white と:black は，それぞれ光センサが返す値の最大値と最小値です．

(light *int*) [関数]

センサポート *int* に接続された光センサが感知している明るさを，整数として返します．ポート番号は，1, 2, 3 のいずれかです．

13.3 角度センサ

角度センサは能動的センサです．使用する前に活性化し，使用が終わったら不活性化する必要があります．

(rotation-on *int*) [関数]

センサポート *int* に接続された角度センサを活性化します．ポート番号は，1, 2, 3 のいずれかです．角度センサは，活性化された時点の回転軸の位置を，以後の原点として使います．あとで取得される角度の値は，この原点からの相対的な値となります．もう一度 rotation-on を呼び出すと，原点が再設定されます．この関数は，*int* を返します．

[2] この値は，ベースとなっている brickOS によるものです．どうして 100 や 99 ではなく 98 なのか不明です．

(rotation-off *int*) [関数]

センサポート *int* に接続された角度センサを不活性化します．ポート番号は，1, 2, 3 のいずれかです．この関数は，*int* を返します．

(rotation *int*) [関数]

センサポート *int* に接続された角度センサの現在の角度を取得し，その値を整数として返します．ポート番号は，1, 2, 3 のいずれかです．角度センサの返す値は，活性化されたときに記憶した原点からの相対的な値で，単位は 360/16 度です．値の正負は，回転方向を表します．例えば，角度センサが活性化されてから軸が 270 度回転すると，この関数は，12（$= 270/360 \times 16$）または-12 を返します．もし符号が期待したものと異なっている場合は，角度センサの上下を逆にしてください．

13.4 温度センサ

温度センサは，受動的なセンサです．したがって，温度センサを活性化したり不活性化する必要はありません．

(temperature *int*) [関数]

センサポート *int* に接続された温度センサから現在の温度を取得し，その値を整数（単位は摂氏）として返します．ポート番号は，1, 2, 3 のいずれかです．

13.5 タッチセンサ

タッチセンサは，受動的なセンサです．したがって，タッチセンサを活性化したり不活性化する必要はありません．

(touched? *int*) [関数]

センサポート *int* に接続されたタッチセンサが何かに触れていれば #t を返し，そうでなければ#f を返します．ポート番号は，1, 2, 3 のいずれかです．例えば，次の式を評価すると，1 番のポートに接続されたタッチセンサが何かに触れるまで待機します．

```
(wait-until (touched? 1))
```

この関数と pressed?を混同しないように注意しましょう．pressed? は，Prgm ボタンが押されたかどうかを判定するものです．

13.6 ランプ

ランプは出力ポートに接続し，モータ用の関数 motor と speed で制御します．ポート番号 *port* に接続されたランプを点灯するには，

(motor *port* :forward) または (motor *port* :back).

とします．消灯するには，

(motor *port* :off) または (motor *port* :brake).

とします．speed 関数を使うと，ランプの明るさを変えることができます．速度を早くするほど，ランプは明るく点灯します．明るさの最大値は，:max-speed です．

ランプとモータを同じ出力ポートに接続すると，モータが動いている間だけランプは点灯し，その明るさは，モータの速度を反映することになります．

14 メモリ管理

XS のデータオブジェクトには，**セル**を使って表現されているものがあります．セルは，RCX 内の RAM メモリに確保された**ヒープ**と呼ばれるメモリ領域に生成されます．個々のセルは，二つの XS オブジェクトを格納することができる構造体で，コンスオブジェクトとよく似ています．実際，1 個のコンスオブジェクトは，1 個のセルで表現されています．しかしセルは，ユーザ定義の記号やユーザ定義関数といった，コンス以外のオブジェクトを表現するためにも使用されています．

オブジェクトを表現するために使用されていないセルを**フリーセル**と呼びます．最初，XS が起動されたときは，ヒープ中のすべてのセルはフリーセルです．XS システムは，オブジェクトを表現するためにセルが必要になると，ヒープの中から必要な個数のフリーセルを探しだして使用します．このようにセルを使っていくと，いつかは，すべてのセルを使い切って，ヒープにはフリーセルがなくなってしまいます．計算を続けるために，XS システムは一度使ったセルのうちで，もう使われていないものを再利用します．この再利用の処理を**ごみ集め**と呼びます．通常，ごみ集め処理は，フリーセルがなくなったときに自動的に起動します．しかし，ユーザが後述の関数 gc を使って強制的にごみ集めを起動することもできます．

ヒープに生成されるセルの個数は，RCX のインストール時に決定します．もし自動的にごみ集めを行った後で十分なフリーセルが確保できなったら，XS は計算を続行できません．その場合，XS は “heap full” というエラーメッセージを表示して現在の計算を中止します．計算が中止された後でも，XS システムとの対話を続けられる可能性があります．それは，計算中止によってセルのいくつかがフリーになることがあるからです．

(gc) [関数]

ごみ集めを起動します．ごみ集めが終わると，その時点のフリーセルの個数を返します．

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ました．未踏 PM（プロジェクトマネジャー）の近山隆先生からは有益な助言をいただきました．Franz 社の Sheng-Chuan Wu 博士には，本マニュアルの英語版草稿に目を通していただき，改善のための有益なコメントをいただきました．これらの方々に心より感謝の意を伝えたいと思います．

索 引

XS 関数一覧

XS のすべての関数と入力時定数の一覧です．関数については，次の表記を使って引数情報を記載します．

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ます．‘†’は，関数がインストール時オプションであること表します．

Lisp に共通の関数

- 最上位形式
  (define *sym form*)
  (define (*sym sym** [. *sym*]) *form**)
  (load *string*)
  (trace *sym*)
  (untrace *sym*)
  (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*)
  (trace-call *sym fun list*)
  (if *form form* [*form*])
  (catch *form form**)
  (throw *obj obj*)
- 条件
  (and *form**)
  (or *form**)
  (not *obj*)
- 束縛
  (let [*sym*] ((*sym form*)*) *form**)
  (let* ((*sym form*)*) *form**)
  (letrec ((*sym form*)*) *form**)
- 型述語
  (boolean? *obj*)
  (integer? *obj*)
  (null? *obj*)
  (pair? *obj*)
  (symbol? *obj*)
  (function? *obj*)
- 比較
  (eq? *obj obj*)
  (< int^+)
  (> int^+)
  (= int^+)
  (>= int^+)
  (<= int^+)
- 算術演算
  (+ *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*)
- リスト処理
  (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*) †

- 入出力
  (read [*int*])
  (read-char [*int*])
  (read-line [*int*])
  (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*)
  (with-watcher ((*form form**)*)
  *form**)

- 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
  (light-on {1|2|3})
  (light-off {1|2|3})
  (light {1|2|3})
  ; 値は 0(:black)〜98(:white)

- 角度センサ
  (rotation-on {1|2|3})
  (rotation-off {1|2|3})
  (rotation {1|2|3})
  ; 単位は 360/16 度

- 温度センサ
  (temperature {1|2|3})
  ; 単位は摂氏

- タッチセンサ
  (touched? {1|2|3})

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

- サウンド
  (play ((*pitch* . *length*)*))
  ; pitch については下記参照
  (playing?)

- Prgm ボタン
  (pressed?)

- LCD ディスプレイ
  (puts *string*)
  (putc *char column*)
  ; $0 \leq$ *column* ≤ 4 (左端)
  (cls)

- 電池残量
  (battery) ; 単位は 1/10 ボルト

その他の入力時定数

- 標準ポート

  `:stdin`, `:stdout`, `:stderr`

- 音の高さ（pitch）

  `: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`

- 整数

  `:most-positive-integer`
  `:most-negative-integer`