はじめてのSchemeインタプリタ

情報工学課程 1 回生　湯浅信吾

平成 18 年 11 月 21 日

prologue

　終わりまで残り 914 ページ。
そこで立ち尽くす。
「はぁ」
ため息と共に空を仰ぐ。
その先に CommonLisp 第 2 版の終わりがあった。
誰が好んで、こんな分厚い本を書いたのか。
長い文字列が、悪魔のように伸びていた。
「はぁ…」
別のため息。俺のよりかは小さく、短かった。
隣を見てみる。
そこに同じように立ち尽くす女生徒が居た。
「Lisp は、好きですか」
いや、俺に訊いているのではなかった。
「わたしはとってもとっても好きです。
　でも、Common Lisp は…大きすぎるんです。
　言語の仕様とか、インタプリタの仕様とか、ぜんぶ。
　…ぜんぶ、大きすぎるんです。」
たどたどしく、ひとり言を続ける。
「それでも、この言語が好きでいられますか」
「わたしは…」
「作ればいいだろ」
「えっ…？」
少女が驚いて、俺の顔を見る。
「言語の仕様とか、インタプリタの仕様を
　作ればいいだけだろ。あんたにとっての Lisp は
　Common Lisp だけなのか？　違うだろ」
そう。
何も知らなかった無垢な頃。
誰にでもある。
「ほら、作ろうぜ」

俺たちは作り始める。
　　へぼい、へぼいインタプリタを

1 はじめに

1.1 おことわり

これは「坂道の下で出会った少年が少女が、逆境を乗り越えてインタプリタを完成させる、発売日がやたらと伸びる物語」ではありません。タイトルに『はじめての』がついているからといって、アレな趣味 の人向けな文章でもありません。

これはろくに勉強せずに Lisp インタプリタを作り始め、気がつけば Scheme インタプリタに変り、自分に使いやすいようにという大義名分の下、自分しか使えないであろうインタプリタを作った記録を元に Scheme インタプリタの作り方を書いたものです。本物の Lisp や Scheme を勉強したい人は内容を信じないように。また、これを読んで Lisp に失望したりしないように。

Lisp を知らない人にはわかりにくく、知ってる人にはまどろっこしいのは仕様です。**最低限 S 式を読めないとまったく分からないと思います。**この文章を鵜呑みにして人前で恥をかいても当方は一切責任を負いません。

1.2 ここで作るもの

先に完成品について少し触れておきます。言語仕様は Schemeっぽくしましたが、マクロは使えません。代わりに引数を評価しない関数 (FSUBR と FEXPR) に呼び出し元の環境を送ることによって、代用しています。末尾再帰、GC、コンティニュエーションは実装してます。用意している関数や特殊オペレータは限られていますが、自分で関数を定義することによって大抵のものは作れます。最後に、ここで作成するインタプリタ[1]の名前ですが、”**murasaki**” です。

1.3 3 分で分かる (?)Scheme 入門

Scheme の説明が全く無しではあんまりかと思ったのでちょっとだけ説明を書いておきます。Scheme とは Lisp という言語から派生した関数型言語[2]です。Scheme では多くのデータをリストで表します。リストは **(要素 1 要素 2 ... 要素 n)** という風に括弧を使って書きます。Scheme インタプリタは入力された式 (=プログラム) を評価 (=実行) し、その結果を返します。1, 2, 3 といった即値を入力すると、そのままの値が返され、x, y, z といった変数が入力されると、それの中身の値が返されます。(関数 引数 1 ... 引数 n) と書くと、関数の呼び出しとして扱われます。このように、式も内部的にはリストで表現されます。

2 データの設計

2.1 データ

murasaki の扱うデータは全て 64bit(またはその倍数) の固定長で、データ自身にデータの種類を判別するためのタグ (5bit) をつけた、いわゆる「オブジェクト・タグ方式」というものを採用しました。[3]また、GC のマーク付けのために 1bit を使用し、以上 6bit は固定なので、残り 58bit をデータの種類に応じて使います。このような、64bit から成るデータを総称し、**オブジェクト**と名づけ、実装においては構造体 LObject というもの[4]に値を突っ込みました。

[1] Scheme の仕様を無視しているので、一つの言語ともいえるかもしれません。
[2] 副作用があるから正確には…って話は無しで (笑)
[3] データを指すポインタの方にタグを付ける方式を「ポインタ・タグ方式」というそうです。
[4] 実体は単なる 1byte × 8 要素の配列です。

2.2 コンス

リストの元となるデータ型であるコンスは murasaki では、

- GC のための領域 (1bit)
- データ型を現す領域 (5bit)
- CAR 部 (29bit)
- CDR 部 (29bit)

からなる 64bit の構造体で構成されています。データ長は 8bytes 固定なので、オブジェクトのアドレスは必ず 8 の倍数となるため[5]、3bit 右シフトできるので[6]、29bit で 32bit のアドレス空間を表現できます。

2.2.1 数値

数値は必ず独立した一つのオブジェクトであり、コンスの CAR 部や CDR 部に即値として入れられることはありません。理由は**面倒だったから**です (笑)

2.2.2 シンボル

シンボルは、オブジェクトの後ろから 56bit(7bytes) をつかって 7 文字を格納し、7 文字以上の場合は、後ろに『シンボルの”続き”型』のオブジェクト[7]を作って、同様に 7 文字格納し、まだ足りない場合は同様にします。終端にはヌル文字 (0x00) を置いています。例えば、”long-symbol-name”という名前のシンボルを格納する場合下のようになります。

```
   64bit     64bit     64bit
[long-sy] [mbol-na] [me     ]
```

この 3 つのデータは必ず連続して配置される必要があります。

2.3 メモリ管理

murasaki では、オブジェクトを保持するために、連続した領域を起動時にまとめて取得します。あとは、オブジェクトを生成するたびにその領域を端から使っていきます。評価の過程で生じるごみは、後述する GC が回収します。

2.4 環境

変数の名前と値を対応付けるものを環境といいます。環境は単純に連想リスト[8]を利用しました。とりあえず、大域環境 (グローバル変数用の環境) として以下のようなリストを設定しておきます。

```
((#t . #t) (#f . #f) (nil . nil))
```

ここまで作れば評価器 (プログラムを実際に走らす部分) が作れます。

---

[5]始点が 8 の倍数とは限らないので、実際には始点のアドレスをオフセットとして保持する必要があります。

[6]8 の倍数なら xxx...xxx000 となるので、右端の 0 の部分を省けるという仕組みです。

[7]シンボル型とまったく同じ構造で、タグだけが違うというものです。

[8](名前 . 値) の対がつながったリストのことです。

2.5 評価器

評価器には引数として**式**と**環境**を渡し、以下のように動かします。

- 式が数値ならそれを返す
- 式がシンボルの場合は環境から対応する値を取り出して返す
- 式がコンスの場合は関数呼び出し (後述)

2.6 トップレベル

- 式の入力を待つ
- 入力された式と大域環境を評価器に渡す
- 評価器の返した値を表示

これをループさせれば、インタプリタの出来上がり。入力された S 式は、そのままの構造を持つリストに変換し、表示処理ではその逆をやるだけです。ここは**根性**で書くだけだと思うので、特に何も書きません。実際に作ってみようと思う方は頑張ってくださいね。

3 関数

3.1 SUBR/FSUBR

murasaki で扱う関数は大きく分けて、インタプリタ側に組み込む関数と、Scheme で定義した関数に分かれます。組み込み関数のうち、引数を評価するものを **SUBR**、引数を評価しないものを **FSUBR** と呼びます。SUBR には引数のリストが渡しますが、FSUBR には引数のリストと呼び出し元の環境も送ります。SUBR である car[9], cons、FSUBR である quote[10]は以下のように定義できます。

```
LObject *car(LObject *s)
{
    s = CAR(s); /* 第 1 引数を取得 */
    return CAR(s);
}

LObject *cons(LObject *s)
{
    LObject *ret = CreateCons();
    LObject *p = CAR(s); /* 第 1 引数を取得 */
    SETCAR(ret, p);
    p = CDR(s);
    p = CAR(p); /* 第 2 引数を取得 */
    SETCDR(ret, p);
    return ret;
}
```

[9]car、cdr は実際は nil に対して nil を返す必要があります。

[10]quote は eval の内部で処理させる例をよく見かけますが、murasaki では FSUBR として処理させてます。cond も同様です。

```
LObject *quote(LObject *s, LObject *env)
{
    return CAR(s);
}
```

3.2 環境

car や cdr を関数として認識できるように、大域環境をいじる必要があります。まず、SUBR 型/FSUBR 型のオブジェクトというものを作ります。中身は後ろから 32bit に SUBR/FSUBR の関数のアドレスを入れただけのものです。あとは、それのオブジェクトで car や cdr という名前のシンボルを束縛すれば大域環境の出来上がり。

3.3 SUBR/FSUBR の呼び出し

関数を呼び出せるようにするために、評価器を **Eval** と **Apply**[11]の二つの関数に分けます。二つの定義は以下の通りです。

Eval(form, env)

- form が数値ならそれを返す
- form がシンボルの場合は env から対応する値を取り出して返す
- form がコンスの場合は Apply(Eval(CAR(form)), CDR(form), env) を呼び出しその値を返す

ただし、CAR(s), CDR(s) はコンスの CAR 部、CDR 部を取り出す関数。

Apply(fn, args, env)

- fn が SUBR の場合は args の値を evlis(args, env) に書き換える
- fn から呼び出す関数のアドレスを取り出す
- SUBR なら args を引数に、FSUBR なら args と env を引数につけ関数を呼ぶ

ただし、evlis はリストを受け取り、各要素を評価したリストを返す関数。

ここまで作ると (car '(a b c)) とか、 (+ 1 2) とかが評価できるようになります。[12]入れ子構造を持った (car (cdr '((a b c) (d e f)))) とかも評価できます。感動ものですね (笑)

3.4 EXPR/FEXPR

さて、これで自由に組み込み関数を作れるようになったので、今度は Scheme 側で関数を作れるようにしましょう。Scheme で書かれ、引数が固定個であり、呼び出し前に引数が評価される関数を **EXPR** といいます。同様に、引数が不定個であり、呼び出し前に引数が評価されない関数を **FEXPR** といいます。今度はこの二つを定義/呼び出しできるようにします。

[11] これは Scheme 側から呼び出せる組み込み関数ではないので先頭を大文字にして区別しました。組み込み関数としての eval/apply は先頭を小文字で表します。

[12] ただし 'hoge を (quote hoge) に変換するのが案外めんどくさいのですが。

3.5 begin

EXPR を実行するための足がかりとして、begin という FSUBR をつくります。begin は引数として、1) 複数の式、2)begin を呼び出した環境 を受け取ります。そして、式を順番に評価し、最後の式の評価の結果を返します。これ自身は単純な関数ですが、EXPR/FEXPR を動かす上では重要な役割を果たします。

3.6 lambda

EXPR を定義するために lambda という FSUBR をつくります。lambda は引数として、定義する関数の式のリスト[13](forms)、仮引数のリスト (args)、lambda を呼び出した環境 (env) を受け取ります。そして、これを以下のようなリストにして返します

```
(forms args env)
```

ただし、普通のリストと区別するために先頭のコンスのタグを別のものに書き換え、それを EXPR オブジェクトと呼ぶことにします。

3.7 EXPR の呼び出し

引数のことを無視すれば、EXPR オブジェクトの forms をとりだし、begin の引数として送れば EXPR が呼び出せそうですね。[14] さて、その引数ですが、evlis で実引数を評価したリストを得た後に、EXPR オブジェクトの args を取り出し、両方のリストから要素を一つずつ取り出し、対を作ったものをリストにします。

```
[例 : ((lambda (x y z) (+ x (* y z))) 1 2 3)]
仮引数 (x y z)
実引数 (1 2 3)
生成物 ((x . 1) (y. 2) (z . 3))
```

そして、このリストの後ろに EXPR オブジェクトの env を取り出したものをつなげ、それを env2 とします。最後に begin(forms, env2) を呼び出して、その値を返せば EXPR の呼び出しは終わりです。これらの処理は全て Apply の中に書けばいいでしょう。

環境から値を探すときは先頭から探すので、もし、大域環境などで x, y, z といった仮引数と同じ名前の変数が定義されていても、引数の値の方が先に見つかり、こちらの値が使われます。

これにより、引数を正しく扱うと同時にレキシカルスコープも実現できました。クロージャも作り放題です。一応うまく動くんですが、思いつきでこんな設計にしたので本当にいいのかどうかよくわかりませんが (笑)

3.8 FEXPR

FEXPR を作る手順は EXPR を作るのとほぼ同じです。FEXPR を定義するために nlambda という FSUBR をつくります。nlambda も引数として、定義する関数の式のリスト (forms)、仮引数のリスト (args)、lambda を呼び出した環境 (env) を受け取り、それをリストにして返します。先頭のタグは EXPR オブジェクトとは別のものを付け、それを FEXPR オブジェクトと呼ぶことにします。ただし、仮引数の数は 2 個と固定です。[15]

[13] リストなのは複数の式を定義できるようにするためです。
[14] まあ、そのように設計したから当たり前なんですが (笑)
[15] 第一引数に実引数のリスト、第二引数に呼び出し元の環境が束縛されます。

3.9 FEXPR の呼び出し

これもまた、EXPR とほぼ同じですが、引数の扱いが若干異なります。まず、実引数は評価せず、そのままのリストの形で第一引数を束縛します。次に、現在の環境 (= 呼び出し元の環境) で第二引数を束縛します。

```
[例 : ((nlambda (s a) s) hoge foo)]
仮引数 (s a)
実引数 (hoge foo)
生成物 ((s . (hoge foo)) (a . 呼び出し元の環境))
```

で、これの後ろに FEXPR オブジェクトの env をつなげ、それを env2 と名付け、begin(forms, env2) を呼び出せば終わりです。これも EXPR と同じく Apply の中に書けばいいだけです。

「呼び出しもとの環境」とは、FEXPR を呼び出したときの環境であり、nlambda を呼び出した環境とは別ものなので、そこに注意してください。[16]

ここで、Eval や Apply を組み込み関数として Scheme 側から呼び出せるようにすると、[17]マクロのようなものが書けるようになります。もう…マクロなんてなくてもいいよね？

3.10 束縛を作る

さて、せっかく関数を作れるようになったので、それを束縛できるようにした方が便利でしょう。という訳で、define と set! を作ります。

define は FSUBR で、引数としてシンボルと任意の値を受け取り、それを対にしたものを現在の環境リストの先頭に付け加えます。第一引数であるシンボルは評価しませんが、第二引数は評価するということをお忘れなく。こうして関数を束縛できて、何がうれしいかというと、これだけで再帰が書けるようになるんです。関数定義のところで面倒なものを書いただけの価値があったということですね (笑)

さて、束縛を変更する set!ですが、これも FSUBR にしてもいいんですが、実は FEXPR でもかけたりします。[18]FEXPR の価値を感じれる瞬間です。だから……マクロは作らないって事でいいですよね (笑)

```
;set!の定義の例 (エラー処理なし)
(define set!
  (nlambda (s env)
    (set-cdr! (assoc (car s) env)
              (eval (cadr s) env))))
```

3.11 ここまででできること

条件分岐を行う cond も FSUBR として定義すれば比較的簡単に書けます。さらには、cond があれば FEXPR として if を定義することも可能ですし、lambda を使うことによって let を FEXPR として定義することさえも可能です

つまり、ここまで作ればほとんど完成なんです。

……『**継続**』というものさえなければ……

---

[16]FEXPR を束縛せずにその場で呼び出した場合は二つの環境は同一ですが、FEXPR を作成後束縛し、作成時とは別の環境で FEXPR を呼び出した場合、二つの環境は別物となります。

[17]組み込み関数としての Eval/Apply は小文字で示します。なお、処理内容は Eval/Apply を呼び出すだけです。

[18]ただし、コンスの参照先を書き換える set-car!, set-cdr!を用意しておく必要があります。

4 ごみ集め

4.1 Garbage Collection

メモリ管理の方式については最初に少々述べましたが、ここでは実装した GC について、簡単に書きます。

murasaki ではせっかくだから GC と一緒にコンパクションもやろうと思って、Stop and Copy 方式の GC を使うことにしました。[19]

この GC は下準備、マーク付け、コピーの三つから構成されています。マーク付けとコピーはメモリが不足したときのみ行います。

下準備とは

- オブジェクトが使用するメモリ空間を A と B の二つに分け、始め A を使用する
- インタプリタ側の変数[20]がオブジェクトを参照するたびに、その変数のアドレスをスタック[21]S に積む
- オブジェクトを参照した変数の寿命が尽きると、S からそのアドレスを取り出す

マーク付けとは

- S に入っているアドレスの中身が参照しているオブジェクトにマークをつける
- そのオブジェクトが参照している[22]オブジェクトにもマークをつける

コピーとは

- A にあるオブジェクトを全てたどり、マークが付いているもののみ B にコピーする
- コピー元の内容をコピー先のアドレス (”引越し先”) に書き換える
- S に入っているアドレスの中身の参照先を”引越し先”に書き換える
- A と B の役割を交換する

以上です。はっきりいって結構面倒です。素直に Mark and Sweep を使って置けばよかったと後で後悔しました (笑)

5 継続

5.1 評価器を射ち堕とした日

> 主よ、私は評価器を殺めました。
> 私は、この手で大切な評価器を殺めました。

Scheme には「継続をファーストクラスオブジェクトとして扱える」という大きな特徴があります。これだけでは何のことやらさっぱりわかりませんが、簡単に言うと、関数を越えた大域的な goto を使えるということです。

これは C の setjmp/longjmp と非常に似ています。しかし、longjmp が使えるのはスタックの状態が setjmp を呼び出したときから保たれているとき—即ち、スタックが浅くなる時だけです。それに対し、継続を利用したジャンプはスタックが深くなる時でもできます。

---

[19] けど、マーク付けをするところは完全に Mark and Sweep なんで正確には Stop and Copy ではないです。Mark and Copy とでも言うのでしょうか？

[20] インタプリタ内部で使われるローカル変数やグローバル変数のことです。

[21] ポップせずに中身が見れるようにする必要があるので、これも正確にはスタックではありません。

[22] 例えばコンスの CAR 部と CDR 部。それに加えて EXPR/FEXPR オブジェクトなども参照を持っているので注意してください。

例えば、A, B を関数とすると、
A → B → (setjmp)
という順番に関数を呼んでいくと、関数 B から抜けてからは longjmp は使えません。なぜなら、 A → B という順番に呼び出されたという情報は、どこにも保存されていないため再現できないからです。しかし、Scheme の継続を利用すると失われたスタックを再現し、より汎用的なジャンプが実現できます。

非常に分かりにくい説明でしたが、継続の概念を実装するには、関数呼び出しのスタックを完全に記憶できる必要があります。現在、関数呼び出しは Eval と Apply によって制御され、このままではスタックを触るどころか、スタックの状態を取得することすらできません。つまり、**現在の仕様ではどうやっても実装できない**ということです。

「避けられぬ終焉は　せめて作ったこの手で…」

5.2 ラベル

さて、Eval/Apply という二つの関数の存在を抹消したら、新たな評価器を作らなければなりません。とりあえず、Evaluator という名前にでもしておきましょう。

そこに、過去の Eval と Apply を**ラベルとして**定義します。お互いの呼び出しは goto で行います。詳しくは下のソースをご覧ください。

```
#define EVAL_LABEL   100
#define EVAL_LABEL_1 101
/* 中略 */

LObject *Evaluator(LObject *s, LObject *env) {
  /* 中略 */
eval_label: /* a1=s, a2=env と割り当てられる */
  /* 中略 */
  PushVar(a1); /* a1 の値を保存 */
  a1 = CAR(a1);
  PushLabel(EVAL_LABEL_1); /* goto の後の戻り先を設定 */
  goto eval_label;
eval_label_1:
  a1 = PopVar(); /* 保存した a1 の値を取り出す */
  a3 = a2;
  a2 = CDR(a1);
  a1 = ret_val; /* "上の "goto eval_label"によって得られた値 */
  PushLabel(EVAL_LABEL_2);
  goto apply_label;
eval_label_2:
  goto return_label;
  /* 中略 */
apply_label: /* a1=fn, a2=args, a3=env と割り当てられる */
  /* 中略 */
return_label:
  /* スタックから値を取り出し、対応するラベルに goto で飛ぶ。 */
  /* スタックが空になった場合、関数を抜ける。 */
}
```

長い上に分かりにくいソースで申し訳ございません。重要なのは二つ。一つは、ローカル変数、関数の呼び出しによるスタックの管理を自分でやるということです。もう

一つは、上のソースからは分かりませんが、このスタックの管理をリスト (コンスオブジェクト) を用いてやるということです。[23]このリストを返すと、スタックの状態を Scheme 側でただのオブジェクトと同様に取り扱えるようになります。例によってタグを付け替え、それをコンティニュエーションオブジェクトと呼ぶことにします。

5.3 call-with-current-continuation

言葉の説明だけでは分かりにくいので、実際に継続を用いた Scheme のプログラムを見てみましょう。

```
(define cont #f) ; 適当な初期値を代入
(+ 1 (call/cc (lambda (k)
                (set! cont k)
                1))) ; 結果=>2
(cont 2)             ; 結果=>3
```

まず、 + が関数と解釈され、引数の評価が始まります。call/cc を評価した時点で現在のスタックの状態 (=コンティニュエーションオブジェクト) が生成され、call/cc は、それを引数として lambda により定義された無名関数を呼び出します。そこで、コンティニュエーションオブジェクトでグローバル変数 cont を束縛し、1 を返します。これで + の引数の評価が終わったので、 + が実行され、結果として 2 が返ります。

問題は次の行です。束縛したコンティニュエーションオブジェクトに引数を付けて呼び出しています。こうすると、現在のスタックの状態が破棄されて、[24]スタックの状態はコンティニュエーションオブジェクトが生成されたときのものになります。(つまり、 + の引数を評価**しようとしていた**状態です。) そして、call/cc の戻り値が引数である 2 であるかのように振る舞い、結果として + の引数の評価後の値が (1 2) となるため、3 が返るというわけです。

これを実装するために評価器に追加するのは、call/cc が評価[25]された時点でコンティニュエーションオブジェクトを作成するという処理、call/cc のために goto した apply_label にて、そのコンティニュエーションオブジェクトを引数として、call/cc の引数である関数を呼び出すという処理。そして、コンティニュエーションオブジェクトに引数を付けて呼び出した場合、スタックの状態を書き換え、引数を返り値にするという処理です。

面倒なようですが、案外簡単にかけます。どちらかというと、今までの Eval/Apply を壊してラベルをたくさん付ける方がよっぽど大変です。

5.4 修正は何処までも続いてゆく → 世界の果てを目指して

さてさて、評価器を書き換えたことによる修正はまだ続きます。FSUBR の中には cond、begin のように内部から Eval を呼んでいたものがあります。これらをそのまま使用することはできないので、内部から Eval を呼んでいた FSUBR は**全て** Evaluator の内部のラベルとして書き換える必要があります。物凄く面倒です。異常なまでに面倒で途中で泣きたくなりました。しかし、これさえ作れば一応簡単な Scheme インタプリタとしては一応完成です。

---

[23]リストの先頭に新たなものを積むようにスタックを作るのがコツです。この先頭へのポインタは適宜書き換えますが、スタックを成すコンスの参照先を書き換えてはいけません。保存したスタックが書き換えられてしまうからです。また、ラベルは適当に割り当てた数値として格納しておくといいでしょう。

[24]Scheme 側のオブジェクトなので放って置いたら GC が勝手に回収してくれます。

[25]apply_label でやってもいいのですが、コンティニュエーションオブジェクトに引数を付けて呼び出す処理のことを考えると、eval_label でやった方が楽です。

5.5　継続まとめ？

「Scheme の継続の実装は多くの人が苦労し、各々の実装をしていった。」と後に聞きましたが、murasaki での継続の実装ははっきり言って**非常に面倒**なものです。先人が私と同じ実装をしたのか、全く違う実装をしたのかは定かではありません。[26] しかし、この方式ではコンティニュエーションオブジェクトを作るのにコストが全くかかりません。また、それを元にスタックを書き換えるのもコストが全くかかりません。ここだけが自慢です。……ここだけが。

5.6　末尾再帰

これまで末尾再帰に触れませんでしたが、関数呼び出しのスタックが自由に触れるようになった今、末尾再帰を goto に書き換えるのはそれほど難しくはないでしょう。ご自由にお作りください (笑)

murasaki では begin の最後でごにょごにょやってるだけです。

epilogue

「もし…
　Scheme に人と同じように、意志や心があるとして…
　そして、それを使う人たちインタプリタを作らせようって…
　そんな思いで、いるとしたら…
　murasaki が完成したという奇跡も、そんな Scheme のしわざかもしれないです」
いや…奇跡はこれからたくさん起こるのだろう。
そんな気がしていた。
「でも、それは奇跡じゃないですよね
　Scheme を大好きな人が、インタプリタを作り…
　人を好きな Scheme が、人にインタプリタを作らす…
　そんな、誰にでもある感情から生まれるものです
　murasaki だけじゃないです
　どんな Scheme だって、そうです
　わたしたちは Scheme を愛して、Scheme に育まれてるんです
　そう思います」
「なぁ…
　Scheme は、大きな家族か
「はい。Scheme も人も、みんな家族です」
「そっか…」
「だんご大家族です」
ああ…そうか。
そうだったのか。

ずっとずっと歌っていよう。
今日からの思い出を…
このインタプリタと…
Scheme のために。

　—完—

[26] 色々調べたところ、いくつか実装方法を見ましたが、私と同じ方法をとっているものは見つかりませんでした。