プログラミング言語入門

2012/06/28
watanabe

目次

はじめに

この文書はプログラミングの方法ではなくプログラミング言語についての文書です。プログラムがどのように実行されるかを理解していきます。では、なぜ、プログラミング言語なのでしょう。

著者が働いている職場には優れた技術者はたくさんいます。ただしそれぞれ受けた教育など背景が異なるため、プログラムは書けてもその基礎となっている計算機科学 (コンピュータサイエンス) の理解があやふやな人を、著者は多く見てきました。プログラミングに自信があるという人がもう一歩先に進

める道を示したいというのがこの文書を書き始めた動機です。

この文書を読むことで次の効果が得られることを期待しています。

- プログラミング言語とは何かを深く理解することで、プログラミングのレベルが上がる。
- “この言語が良い!”と言う時に、シンタックス、ライブラリ、プログラミング言語固有の機能など、どのレベルの機能に言及しているのか区別できる。
- プログラムがどのように実行されるか理解できる。
- 関数型言語の中心となる概念を理解できる。
- λ式とクロージャの違いが説明ができる。
- 計算機科学の良書である通称 SICP[*1] という本を読む準備ができる。
- 友達に“関数型言語の処理系を作ったことがある”と言える ;-)

どれか一つにでも魅力を感じれば、この文書の読者の対象と

[*1] Structure and Interpretaion of Computer Programs 2nd ed.: 訳本 計算機プログラムの構造と解釈 第 2 版

言えるでしょう。

プログラミング言語を理解するために、これからμSchemeR という独自のプログラミング言語を作成していきます。ものごとの本質を理解するにはその内部がどうなっているのかを理解することが最も大切だと信じているからです。理解するためには実際に作ることが一番です。作成する言語は小さな関数型言語を選びました。作成が簡単にも関わらず強力な機能を持っていること、通常の人にはなじみが少ないパラダイムである関数型のプログラミング言語を理解することにより、プログラミングの知識に幅を持てるようになるという理由からです。

内容が難しすぎそうと不安に思いますか。決してそんなことはありません。この文書の知識の源になっている通称 SICP という本は、MIT の計算機科学での入門レベルの講義に使われていました。著者も学部時代に研究室に配属されてまず読まされた本です。そのくらい、計算機科学に携わる人には基本であり、だからこそみなさんに知っていただきたい内容なのです。たしかに、読者として、ある程度のプログラミングをしたことのある人を想定しています。ある程度がどの程度なのかの線引きは出来ませんが、なるべく、興味を持ってくれた

人に全員に理解してもらえるよう書いたつもりです。

μSchemeR を実現するためのプログラミング言語には Ruby を選びました。Ruby は多くの人が親しんでいる手続き型言語であり、機能も強力なため、本質的なことを簡潔に説明するのに役立ちます。Ruby に精通していなくとも、Ruby の簡単な機能しか使いませんし、文章でも説明していきますので気負わず読み進めてみて下さい。

本文書では、なぜそれが必要なのかをできるだけ記載するようにしました。それが本質を理解するための近道だと思うからです。正解だけを示すのは簡単ですが、それがなぜ正解なのか、どうやって正解に行き着いたのか、問題は何だったのかを読み取るのは容易ではありません。そこで、まずうまく動作しない例を挙げ、問題を理解し、それを解決する手段を説明していきます。

本文書で出てくるプログラムをコピー & ペーストしていけば、実行できるようになっています。動作することを確認するくらいには役立つでしょう。ただし、本当に理解したいのであれば、なるべく自分でプログラミングしてみてください。それも本文書のプログラムを見て理解した後は、そのプログラムを見ずに。どこが理解できないでいるかが明確になるかと

思います。

また、本文書は末尾に示すとおり、クリエイティブ・コモンズ 表示 - 非営利- 継承 3.0 非移植 ライセンスの下に公開しています。改変、再配布をライセンスに従う範囲内で認めていますので、後輩の教育に有効活用していただければこんなにうれしいことはありません。人に教えることが自分で学ぶことの一番の近道であることを著者はこの文書を書きながらつくづく実感しました。

前振りが長くなってしまいました。それでは、はじめましょう。

第1章 プログラムと評価

プログラミング言語とは

プログラミング言語とはプログラムを書くための言語です。プログラミング言語の処理系とは、与えられたプログラムを計算するもので、コンパイラと実行系の組み合わせで計算したり、インタープリタで計算したりします。本書ではプログラムを逐次解釈し実行していくインタープリタを作って行きます。これは、与えられたプログラムの動作を規定するもので

すので、見方を変えれば、処理系であるインタープリタによりプログラミング言語を定義しているとも言えます[*1]。

では、そもそもプログラムとは何でしょう。計算するための手順とも言えそうですが、本文書では概念的なものではなく、実際に簡単なプログラムの例を示しながらそれが動くように少しずつ機能を追加していきます。

はじめてのプログラムの評価

最初のプログラムとして次のプログラムを考えます。

```
[:+, 1, 2]
```

今までのプログラムと全く異なる記述方法に戸惑うことでしょう。当然です。これは著者が勝手に考えた μSchemeR(名前も勝手に考えました) というプログラミング言語だからです。気にくわない書き方だと思いますが、この文書を読み終わるころには自分の好きな書き方へ直せる力が身についていると思いますので少しお付き合い下さい。

[*1] これをプログラミング言語を操作的意味論 (operetional semantics) で定義すると言います。

ただ少し想像力を働かせれば、多くの人は 1 と 2 を足し合わせるプログラムなのだと想像できるのではないでしょうか。そのとおり、正解です。では、“ このプログラムを与えられた時にこの結果を求める処理を考えて下さい ” と言われたとき、どう答えるでしょうか。“1 と 2 を足し合わせた結果を求める ” もしくは “ :+ が先頭にあった時、続く 2 つの引数の値を足し合わせる ” とも言えるでしょう。

では、次のプログラムの結果を求める処理はどうでしょう。

```
[:+, [:+, 1, 2], 3]
```

“ まず [:+, 1, 2] を計算してその結果と 3 とを足し合わせる ” と言えるでしょう。ここで前回の説明に “ 計算して ” という言葉が加わったことに気づいたでしょうか。すなわち、上 2 つのプログラムの計算結果を求める処理を考えたとき、その処理は :+ に続く引数を “ 計算 ” した後に足し合わせる必要があるのです。通常、我々はこの計算のことを “ 評価 (evaluation)” と言います。したがって、“ :+ が先頭にあった時、続く 2 つの引数を評価して、その値を足し合わせる ” が求める処理です。

コラム: μSchemeR のシンタックス

今回作成するプログラミング言語 μSchemeR では、`[:+, 1, 2]` などのように最初に関数を、その後引数を記述します。なぜ、このようなスタイルなのでしょう。他言語のように `x + y * z` と書かかれたものを計算するためには `(x + y) * z` なのか `x + (y * z)` なのかを演算子の優先度を考慮して決める必要があります。さらに言えば、後に出てくる if 文も `[:if, :true, 1, 0]` などのように必ず `[` の後に`:if` のようなキーワードが出てくるので、この文字列を見れば通常の関数適用なのか、特殊な構文なのかを簡単に解釈できます。一方で、`if (true) then 1 else 2;` や `x = y + 1;` などのように色々な形の構文を許すとそれがどのような構文なのかを解釈するために多くの計算 (字句解析/構文解析) が必要になってきます。計算機がこの処理をせずに人間がこの作業をすることで、計算機側は簡単な処理でそれを解釈できるようになっていると言うわけです

それでは、このプログラムの結果を求める処理を Ruby で記述してみましょう。

_eval[*2]は、与えられた式 exp を評価し、その結果を返します。

式がリストであった場合、最初の要素を関数として、残りを引数として、それぞれを評価してその値を求めます。求めた関数に求めた引数の値を適用 (apply) してその結果を_eval の結果とします[*3]。一方、リストでない場合、数字であれば即値として扱い数字そのものを返します。例えば 2 は 2 を返します。そうでなければ、組み込み関数とみなし、それに関連付けられた (Ruby 上での) 関数を返します。

```
def _eval(exp)
  if not list?(exp)
```

[*2] eval としないで_eval としているのは、Ruby の組み込み関数として eval が定義されているためです。気になる人は、Ruby の module 機能を使って名前空間を分け、_eval を eval として定義しなおして下さい。またその際は、p. 59 の parse で使われる eval は Kernel::eval に置き換えて下さい。

[*3] Ruby では、関数の最後に評価した式の値が関数の返り値になります。一般的に関数型言語では return <値>と書かずに、このような書き方をします。本文書では、関数型言語の考え方に近づくようにこの記述を利用していきます。

```
    if immediate_val?(exp)
      exp
    else
      lookup_primitive_fun(exp)
    end
  else
    fun = _eval(car(exp))
    args = eval_list(cdr(exp))
    apply(fun, args)
  end
end
```

コラム: プログラムコードと値の区別

“2 は 2 を返します”という文章で数字のフォントが違っていることに気づきましたか? 2 は (現在考えている μSchemeR) プログラムの、2 は (Ruby 上での) 数値の 2 を表します。こう記載することによって、プログラムとその評価された値とを区別します。

以降、上の Ruby プログラムで呼ばれている関数を説明していきます。

リストかどうかは配列のインスタンスかどうかで判断しています。

```
def list?(exp)
  exp.is_a?(Array)
end
```

組み込み関数は、関数名をキーに、関数本体をその値としたハッシュで保有します。関連付けられている関数は Ruby 上での関数です。組み込み関数の評価は、関数名に関連付けられた関数を値として返します。

```
def lookup_primitive_fun(exp)
  $primitive_fun_env[exp]
end

$primitive_fun_env = {
  :+ => [:prim, lambda{|x, y| x + y}],
  :- => [:prim, lambda{|x, y| x - y}],
  :* => [:prim, lambda{|x, y| x * y}],
}
```

car はリストの先頭の要素を、cdr は先頭の要素以降のリストを取得する関数です。この名前は奇妙[*4] ですが、Scheme で使われている名前ですので我慢してください。そのうち慣れてくるでしょう。

```
def car(list)
  list[0]
end

def cdr(list)
  list[1...list.length]
end
```

引数を評価する eval_list は、リストの要素それぞれを評価したものをリストにしたものです。

```
def eval_list(exp)
  exp.map{|e| _eval(e)}
end
```

そのままの値を返す即値として数字を定義しています。

[*4] car は Contents of the Address part of Register、cdr は Contents of the Decrement part of Register、後で出てくる cons は CONStruct から来ています。これらは Lisp が開発された IBM の計算機の機械語に由来しています。

```
def immediate_val?(exp)
  num?(exp)
end

def num?(exp)
  exp.is_a?(Numeric)
end
```

関数適用は、引数の評価値 (Ruby 上の値になります) を (Ruby 上の) 関数へ適用しています。fun_val.call(*args) は、fun_val という Ruby 上の関数を引数 args で呼び出します。*は、可変長引数に対応しており、args が [1, 2] の場合 fun_val.call(1, 2) と展開され、args が [1, 2, 3] の場合 fun_val.call(1, 2, 3) と展開されます。

```
def apply(fun, args)
  apply_primitive_fun(fun, args)
end

def apply_primitive_fun(fun, args)
  fun_val = fun[1]
  fun_val.call(*args)
end

def primitive_fun?(exp)
```

```
  exp[0] == :prim
end
```

実際に、[:+, 1, 2] を評価するときの動きを追ってみましょう。与えるプログラムはリストですのでまず先頭の要素 :+ を評価し、$lambda\{|x, y|x + y\}$ を値として得ます。これは、二つの引数を足す (Ruby 上の) 関数です。次に 1, 2 を評価し、それぞれ (Ruby 上での)1, 2 を得ます。これを (Ruby 上で) 適用することで、3 を得ます。この値は print など Ruby 上のプログラムで表示することができます[*5]。

```
puts _eval([:+, 1, 2])
```

を実行して 3 が表示されましたか。おめでとうございます。おそらく、あなたははじめてプログラミング言語のインタープリタを作ったのではないでしょうか。足し算程度しか出来ないプログラミング言語なので実感は無いかもしれませんが、正真正銘のプログラミング言語の処理系です。

[:+, [:+, 1, 2], 3] が評価される流れも自分で追ってみて下さい。_eval が再帰的に呼ばれている点が役立っている

[*5] 今回作成している μSchemeR 上でも、もちろん表示出来ますが、それは後のお楽しみとします

ことに気づけましたか。

まとめ

この章では次のことを学びました。

- 簡単なプログラムの計算方法 (また、我々はこの計算を“評価”と呼びました)
- 関数適用の評価方法、すなわち、関数と引数を評価して、得られた関数の評価値に引数の評価値を関数適用するということ
- プログラムが評価されると (プログラムの実行結果は Ruby という) 他の世界の値として得られること

普段何気なく書いている `x = y;` というプログラムは、実際は右辺をまず評価してその値を左辺の変数のアドレスに格納する、ということを行っています。漠然とは理解していたと思いますが、実際は本章で学んだ評価という考え方などに基づいてプログラムは実行されています。その内部を少し垣間見ることが出来たのではないでしょうか。

第2章 関数適用の評価

この章では関数型言語で大きな位置を占める関数適用の評価方法を学びます。関数適用とは関数に引数を渡して評価することを言います。前章では組み込み関数である:+ に 2 つの引数を与え適用した結果として、それらの和を評価値とする関数適用を見てきました。この章では、自分で作った関数についてその適用を考えてみます。

具体的なターゲットは、次のプログラムです。

```
[[:lambda, [:x, :y], [:+, :x, :y]],
        3, 2]
```

これは何でしょうか。次の Ruby のプログラムはどうでしょう。

```
def add(x, y)
    x + y
end
add(3, 2)
```

いずれも、引数を二つとりその和を値とする関数を用意し、その関数に引数 3 と 2 を与えて関数適用しているプログラムです。異なる点は、最初のプログラムの関数は名前を持っていません。言わば、関数の中身そのものです。もう少し詳しく説明すると、`[:lambda, [`$< parameters >$`], `$< body >$`]` は仮引数 $< parameters >$ でボディが $< body >$ の関数です。これを関数として、引数を与えることで、前章で学んだ組み込み関数`:+` と同様に関数適用することができます。

このプログラムを実行するためにはどうすれば良いでしょうか。“ 引数の 3, 2 を仮引数の x, y にそれぞれ代入して、中のプログラムを評価する ” と考える人が多いのではないでしょうか。おおよそ正しいのですが、考慮すべきポイントがあります。それを見ていきましょう。

環境

次のプログラムを考えます。

```
[[:lambda, [:x],
  [:+,
   [[:lambda, [:x], :x], 2],
   :x]],
 1]
```

この時、上の説明でうまくいくか考えて見ましょう。一番外側の:lambda の関数適用から考えていきます。:x を 1 に束縛して、:+ から始まるカッコの中の式を評価します。最初に関数:+ を評価して、次に 3 行目の:lambda を評価します。この関数は引数をそのまま返しますので、:x を 2 に束縛して関数適用すると 2 が返ります。問題は次に評価する 4 行目の:x です。この:x は 1 を返すべきですが、:x は先ほど 2 に束縛しています。その結果、2+2 すなわち答えが 4 となってしまうのです (図 2.1)。

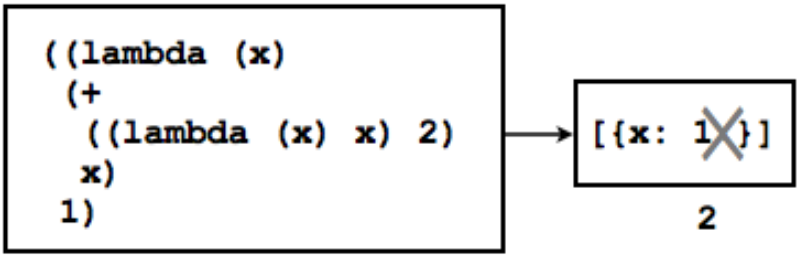

図 2.1　変数の値を上書きするモデルでの評価のようす。x の値が上書きされるため欲しい値が得られない。

コラム: 束縛とは

“ :x を 1 に束縛して ”という文章が出てきましたが、“ 束縛 ”とは何でしょう。変数:x と値 1 とを関連付けるという意味で代入と同じ意味を持ちますが、関数型言語で代入は副作用を引き起こすもの (今は分からないかもしれませんが環境を破壊することとも言います) のことを指すのでそれとは区別して、このように呼びます。また、変数を値に束縛する (bind variable to value) という表現に注意して

ところで先ほど、“:x は 1 を返すべき”と言いましたが、なぜでしょう。次のプログラムは上のプログラムの内側の λ 式の:x を:y に変えたものです。

```
[[:lambda, [:x],
  [:+,
   [[:lambda, [:y], :y], 2],
   :x]], 1]
```

このプログラムでは明かに求める答えは 3 です。プログラミング言語にはスコープという考え方があり変数の有効な範囲が決まっています。多くのプログラミング言語では、今回同様、変数の名前を変えただけでプログラムの結果が変わってほしくないので、:x は 1 を返すような言語仕様になっています。今回作成するプログラミング言語もそのような仕様とします。

話を元に戻しましょう。さて、先ほどの答えが 4 になってしまう問題を解決するためにはどうすれば良いのでしょうか。すでにお気づきかもしれませんが、内側の:lambda の:x と外側の:lambda の:x を区別すれば良いのです。内側の:lambda の関数適用を評価しているときは:x を 2 に束縛し、その外側では:x を 1 に束縛するようにします。具体的には、外側の

:lambda の関数適用を評価し始めるときに:x を 1 に束縛します。内側の:lambda の関数適用を評価し始めるときに:x を 2 に束縛する環境を新たに用意してそちらを優先し、その評価が終われればそれを破棄して元の環境に戻すことで、引き続き:x を 1 に束縛した環境を使うことが出来ます (図 2.2)。

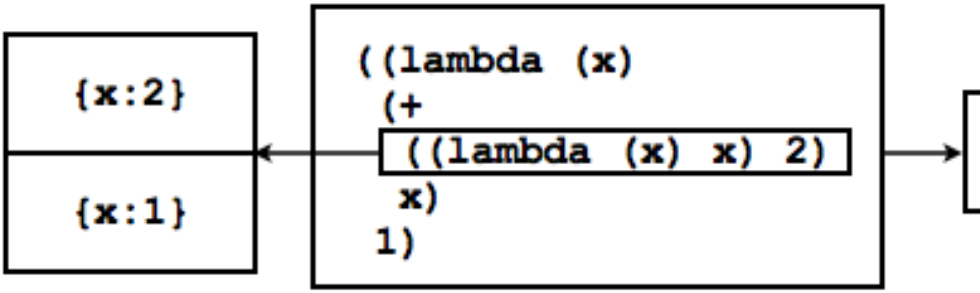

図 2.2　環境モデル。関数適用時に環境を拡張することで、スコープに応じて、変数が束縛している値を得ることができる。

ここで新しく環境という言葉を使いました。環境とは、変数とそれに束縛されている値の組のリストのことです。ここでは、{x:1, y:2} などと表記して x を 1 に y を 2 に束縛し

ていることを表し、その組のリスト [{x:1, y:2}, {x:3}] で環境を表現することとします。この表現方法を使えば、最初の :lambda を評価しはじめるときは [{x:1}] の環境で評価し、内側の:lambda を評価するときは [{x:2}, {x:1}] という環境で評価します。ただし同じ変数があった場合、先頭から見ていき最初にマッチした変数に束縛された値を採用するものとしてします。内側の:lambda を評価した後は [{x:1}] という環境に戻すことで、欲しい値が得られることになります。関数呼び出し時に仮引数と引数の組を環境のスタックに積み、呼び出し終了時にスタックから取り出すいうイメージです。

以降、環境に関する Ruby プログラムを定義していきます。

lookup_var は与えられた環境の中で、指定した変数が束縛している値を見つける関数です。

```
def lookup_var(var, env)
  alist = env.find{|alist| alist.key?(var)}
  if alist == nil
    raise "couldn't find value to variables:'#{
        var}'"
  end
  alist[var]
end
```

環境の拡張は、与えられた変数をキーに値を格納したハッシュを作り、それを環境の先頭に追加することで実現します。

```
def extend_env(parameters, args, env)
  alist = parameters.zip(args)
  h = Hash.new
  alist.each { |k, v| h[k] = v }
  [h] + env
end
```

let 式

少し遠回りになりますが、ここでプログラムを見やすくするために let 式というものを導入することにします。

```
[:let, [[:x, 3], [:y, 2]],
 [:+, :x, :y]]
```

このプログラムは:x を 3 に束縛し、:y を 2 に束縛した環境で、[:+, :x, :y] を評価し、その結果を let 式の評価値とする、と解釈します。λ 式とどこが違うんだと思うかもしれませんが、そのとおり同じものです。上の let 式は、下の式と同じ意味です。

```
[[:lambda, [:x, :y], [:+, :x, :y]], 3, 2]
```

単にプログラムの見やすさから導入した構文ですので、その評価方法も単純です。let は式から仮引数、引数、評価する式を取り出し、λ式に書き換え、評価した値を返します。

```
def eval_let(exp, env)
  parameters, args, body =
      let_to_parameters_args_body(exp)
  new_exp = [[:lambda, parameters, body]] + args
  _eval(new_exp, env)
end
```

let から仮引数、引数ならびに評価する本体の式を抜き出します。

```
def let_to_parameters_args_body(exp)
  [exp[1].map{|e| e[0]}, exp[1].map{|e| e[1]},
      exp[2]]
end
```

let 式かを判定する関数も用意しておきます。

```
def let?(exp)
  exp[0] == :let
end
```

クロージャ

それでは覚えたての let 式を使って次のプログラムを考えてみます。

```
[:let, [[:x, 2]],
 [:let, [[:fun, [:lambda, [], :x]]],
  [:let, [[:x, 1]],
   [:fun]]]]
```

このプログラムの結果、どのような答えが返ってきて欲しいですか。まだ let に慣れていない人のために読み方を補足すると、:x を 2 に束縛した環境で、:fun を:x を返す関数に束縛した環境で、:x を 1 に束縛した環境で、:fun 関数を適用した値を求める、というプログラムです。要は:x が fun:の呼び出し時の値をとるのか (この場合:x は 1 です)、評価されたときの値をとるのか (この場合:x は 2 です) です。正解は、“ プログラミング言語を作る人 (すなわち、著者!) が決める” です。そして、その答えは、2 です[*1]。

[*1] このようにプログラムの文脈だけで値を決められるスコープをレキシカルスコープと言い、一方で 1 が返るようなプログラム実行時の環境を利用する方法をダイナミックスコープと言います。

では 2 を得るためには、どう実現すれば良いでしょう。

λ 式を評価するときに、評価時の環境も合わせて持っておくことで、これを実現出来ます。λ 式を評価した時にその結果として、λ 式とその時の環境をペア $([(lambda()x), x : 2])$ で持ち、fun はこれを値として束縛します。その後:x が 1 を束縛すると、環境は [{x:1}, {x:2}] となります。ただし、fun を関数適用する際には、先ほどのペアで作成した環境 [{x:2}] を利用して λ 式を評価することで x の値は 2 となります。

まとめると、λ 式の評価値は λ 式とその評価時の環境のペアです。その λ 式を関数適用する際には、もう一方のペアである環境で (引数を仮引数に束縛して拡張して) 評価します。この λ 式と環境のペアのことをクロージャと呼びます。

実際の Ruby のプログラムで実現していきます。

λ 式は、λ 式と環境でクロージャを作りその評価値とします。

```
def eval_lambda(exp, env)
  make_closure(exp, env)
end

def make_closure(exp, env)
  parameters, body = exp[1], exp[2]
```

```
  [:closure, parameters, body, env]
end
```

λ 式の関数適用は、クロージャから λ 式と仮引数および環境を取り出し、取り出した環境を引数と仮引数で拡張して、λ 式を評価します。

```
def lambda_apply(closure, args)
  parameters, body, env =
      closure_to_parameters_body_env(closure)
  new_env = extend_env(parameters, args, env)
  _eval(body, new_env)
end

def closure_to_parameters_body_env(closure)
  [closure[1], closure[2], closure[3]]
end
```

最後に、_eval を変更します。式に加えて環境も引数とします。その他、導入した λ 式や let 式を扱えるよう変更します。apply も λ 式の関数適用を扱えるように変更します。

```
def _eval(exp, env)
  if not list?(exp)
    if immediate_val?(exp)
      exp
```

```
    else
      lookup_var(exp, env)
    end
  else
    if special_form?(exp)
      eval_special_form(exp, env)
    else
      fun = _eval(car(exp), env)
      args = eval_list(cdr(exp), env)
      apply(fun, args)
    end
  end
end

def special_form?(exp)
  lambda?(exp) or
    let?(exp)
end

def lambda?(exp)
  exp[0] == :lambda
end

def eval_special_form(exp, env)
  if lambda?(exp)
    eval_lambda(exp, env)
  else
```

```
    let?(exp)
      eval_let(exp, env)
  end
end

def eval_list(exp, env)
  exp.map{|e| _eval(e, env)}
end

def apply(fun, args)
  if primitive_fun?(fun)
    apply_primitive_fun(fun, args)
  else
    lambda_apply(fun, args)
  end
end
```

ユーザプログラムの評価に使う大域環境を用意します。大域環境には組み込み関数を設定します。これにより、組み込み関数を変数と同じように lookup_var で扱うことが出来ます。

```
$global_env = [$primitive_fun_env]
```

以降、プログラムを評価して下さいと言われた時は、次のようにプログラムと、この環境を引数として評価して下さい。

```
exp = [[:lambda, [:x, :y], [:+, :x, :y]], 3, 2]
```

```
puts _eval(exp, $global_env)
```

クロージャは強力です。次のプログラムを見てください。

```
[:let, [[:x, 3]],
 [:let, [[:fun, [:lambda, [:y], [:+, :x, :y]]]],
   [:+, [:fun, 1], [:fun, 2]]]]
```

:fun を束縛したλ式中の:x はその中で値を束縛していないにも関わらず利用できる点に注意して下さい[*2]。これはクロージャが環境を持っているからです。

評価 (eval) と関数適用 (apply)

プログラムの評価方法はこれでほぼ全て学びました。流れをおさらいしてみましょう。プログラムが与えられると、関数ならびに引数の部分に分けられそれぞれを評価します。その後、引数をその関数に適用します。すなわち、仮引数を引数に束縛して、関数のボディを評価します。次は、このボディの中に含まれるプログラムについて、これら一連の処理を繰り返すこ

[*2] 最後に定義する define などを利用することで $\{x : 3\}$ の外で fun を使うことが出来るようになります

とになります。このように評価 (eval) と関数適用 (apply) を再帰的に繰り返しながらプログラムは実行されていくのです。

λ式とクロージャの違い

ここまで読んできた皆さんなら、λ式とクロージャの違いをよく理解しているでしょう。λ式は単なるプログラムのコードであり、クロージャはλ式とそれを評価した時の環境のペアです。λ式を評価するとクロージャがその評価値となります。このクロージャを関数適用するときは、クロージャ中の環境 (仮引数を引数に束縛し拡張した環境) でλ式を評価します。

λ式扱うことができるプログラミング言語は、関数を値として扱う関数、すなわち高階関数として FORTRAN など古くから存在してきました。値として扱うとは、引数や返り値などで使うことが出来ることを言います[*3]。高階関数は処理を抽象化できるため強力な機能となります。例えば、関数の積分値を数値計算で求めたいとき、求める関数を引数とする

[*3] 専門用語では、関数がファーストクラスのオブジェクトであるとも言います。

ことで、各関数毎に同じ処理を書かずにすみますし、記載した処理がわかりやすくなります。

クロージャは高階関数よりもさらに強力です。ソースコードと環境のペアを値として扱うことで、内部状態を隠蔽することが可能です。最近のプログラミング言語ではクロージャを扱えるものが増えています。ぜひ、その可能性を最大限に利用してプログラミングをより楽しんでください。

まとめ

この章では次のことを学びました。

- 関数適用の評価方法
- クロージャの関数適用は、クロージャ中の環境を仮引数を引数に束縛して拡張した上で、λ式を評価する
- 環境とは、変数とそれに束縛された値の組のリスト
- クロージャはλ式と評価時の環境のペア
- プログラムは評価と関数適用が再帰的に呼ばれながら実行される

第3章 再帰

この章では再帰について学びます。ターゲットとなるプログラムは次のものです。

```
[:letrec,
 [[:fact,
   [:lambda, [:n], [:if, [:<, :n, 1], 1, [:*, :n,
        [:fact, [:-, :n, 1]]]]]]],
 [:fact, 3]]
```

まずは準備として、μSchemeR の機能を少し拡張しましょう。

条件式

次のような if 式で条件を扱えるようにします。

```
[:if, [:>, 3, 2], 1, 0]
```

if 式の評価は if 式から条件、真節、偽節を取得し、条件の評価値が真であれば真節を評価し、偽であれば偽節を評価し、その値を返します。

if 式を評価するプログラムを上のとおり書いていきましょう (後で使う letrec も合わせて一緒に定義しています)。

```
def special_form?(exp)
  lambda?(exp) or
     let?(exp) or
     letrec?(exp) or
     if?(exp)
end

def eval_special_form(exp, env)
  if lambda?(exp)
    eval_lambda(exp, env)
  elsif let?(exp)
    eval_let(exp, env)
  elsif letrec?(exp)
    eval_letrec(exp, env)
```

```
  elsif if?(exp)
    eval_if(exp, env)
  end
end

def eval_if(exp, env)
  cond, true_clause, false_clause =
      if_to_cond_true_false(exp)
  if _eval(cond, env)
    _eval(true_clause, env)
  else
    _eval(false_clause, env)
  end
end

def if_to_cond_true_false(exp)
  [exp[1], exp[2], exp[3]]
end

def if?(exp)
  exp[0] == :if
end
```

if 式で分岐するために論理値のリテラルを導入します。:true, :false は (Ruby の)true, false として解釈するよう大域環境に加えます。

```
$boolean_env =
    {:true => true, :false => false}
$global_env = [$primitive_fun_env, $boolean_env]
```

条件式で扱えるよう、組み込み関数に不等号、等号の演算子を加えます。

```
$primitive_fun_env = {
  :+ => [:prim, lambda{|x, y| x + y}],
  :- => [:prim, lambda{|x, y| x - y}],
  :* => [:prim, lambda{|x, y| x * y}],
  :> => [:prim, lambda{|x, y| x > y}],
  :>= => [:prim, lambda{|x, y| x >= y}],
  :< => [:prim, lambda{|x, y| x < y}],
  :<= => [:prim, lambda{|x, y| x <= y}],
  :== => [:prim, lambda{|x, y| x == y}],
}
$global_env = [$primitive_fun_env, $boolean_env]
```

再帰

これで準備が出来ました。いよいよ再帰を見ていきます。次のプログラムを実行してみましょう。

```
[:let,
```

```
 [[:fact,
   [:lambda, [:n], [:if, [:<, :n, 1], 1, [:*, :n,
        [:fact, [:-, :n, 1]]]]]],
 [:fact, 0]]
```

1 が表示されましたか。では次のプログラムはどうでしょう。

```
[:let,
 [[:fact,
   [:lambda, [:n], [:if, [:<, :n, 1], 1, [:*, :n,
        [:fact, [:-, :n, 1]]]]]],
 [:fact, 1]]
```

エラーになりました。この違いは何でしょう。:let を評価すると:fact をλ式の値であるクロージャに束縛します。このクロージャの環境には:fact は含まれていない点に注意しましょう。[:fact 1] として関数適用するとクロージャ中の環境を用いて評価します。評価を進め、if 式で偽となると:fact を評価しますが、先に述べたようにこの環境には:fact は含まれていないためエラーとなるのです (図 3.1)。すなわち、関数として定義しようとした式の中でその関数の名前を使うには let 式では不十分であることが分かります。

これを解決するためにはどうすれば良いでしょう。問題は、先に述べたように、:lambda を評価してクロージャを作ると

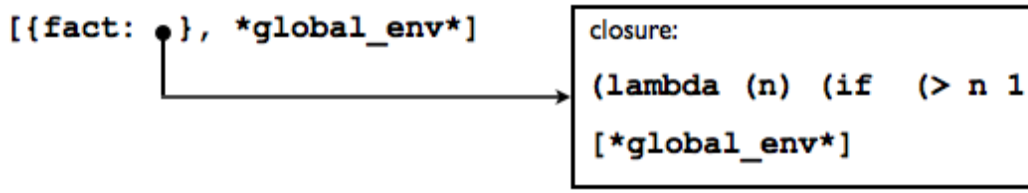

図 3.1　[:fact 1] 評価時の環境のようす。λ 式評価時の環境に fact がないため、クロージャ内の環境には fact は存在せず、[:fact 1] でクロージャ中の fact を参照しようとするとエラーになる。

きに:fact が束縛されていないため、返されたクロージャを関数適用に用いると、その中の:fact が評価できない点にあります。すなわち、作られるクロージャ中の環境に、それが出来た後で束縛しようとしている:fact が含まれている必要があるのです。

これを解決するために、少しトリッキーなことを行います。アイデアは、λ 式を評価しクロージャを作成するときに環境として予め、パラメータ (ここでは:fact) の領域を確保しておくことです。ただし、それを束縛する値はまだ定まっていな

いので、ダミーの値を入れておきます (図 3.2(a))。このとき、パラメータは評価されないためダミーの値でも問題にはなりません。λ式を評価してもλ式の中は評価されずにクロージャとして返されるためです。λ式の評価値であるクロージャが得られたら、先ほどのパラメータをダミーの値からその値に束縛するように変更します。(図 3.2(b)) これでパラメータを評価すると、そのパラメータを環境として含むクロージャを得ることが出来ます。得られた環境を使って letrec 式のボディ `[:fact, 1]` を評価すると (図 3.2(c))、λ式内の`:fact` を所望どおり参照することが出来ます。

以上の機構を実装した再帰を扱う `letrec` を導入します。

```
def eval_letrec(exp, env)
  parameters, args, body =
      letrec_to_parameters_args_body(exp)
  tmp_env = Hash.new
  parameters.each do |parameter|
    tmp_env[parameter] = :dummy
  end
  ext_env = extend_env(tmp_env.keys(), tmp_env.
      values(), env)
  args_val = eval_list(args, ext_env)
  set_extend_env!(parameters, args_val, ext_env)
  new_exp = [[:lambda, parameters, body]] + args
```

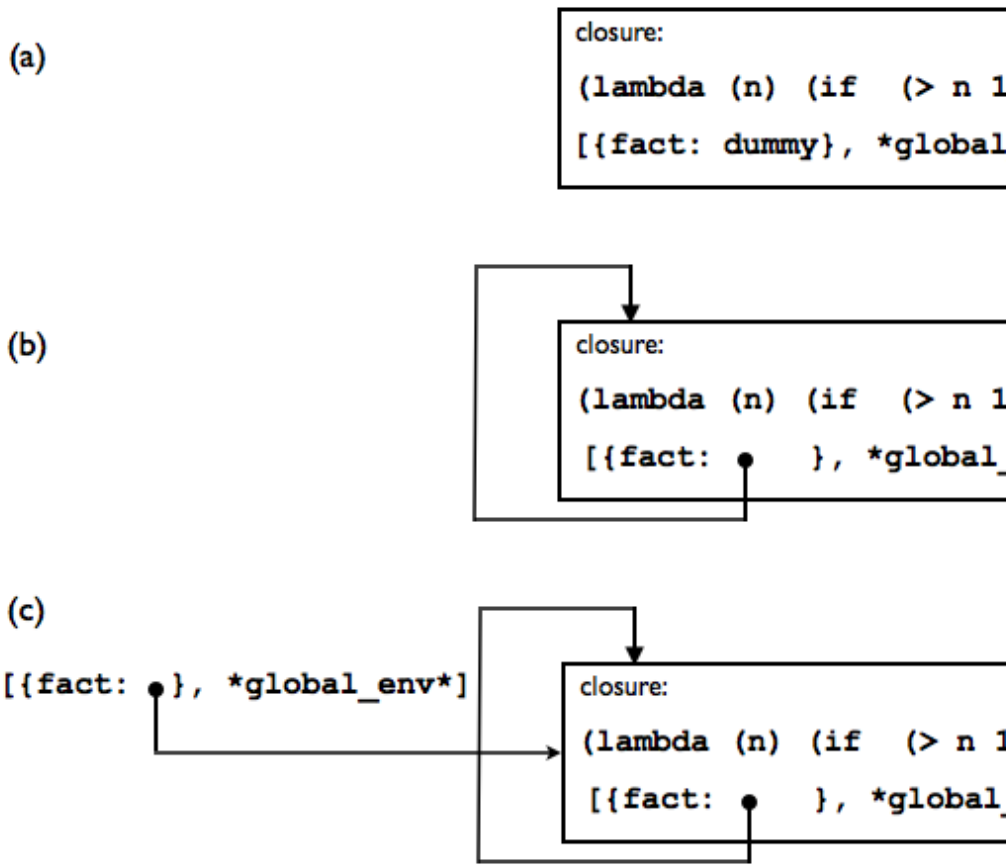

図 3.2　(a) λ 式の評価時の環境に `fact` をダミー値 `dummy` に束縛しておき、(b) 得られたクロージャの環境の `fact` をクロージャ自身に束縛することで、(c)`[:fact 1]` でクロージャの環境内の `fact` が参照可能となる

```
  _eval(new_exp, ext_env)
end

def set_extend_env!(parameters, args_val,
    ext_env)
  parameters.zip(args_val).each do |parameter,
      arg_val|
    ext_env[0][parameter] = arg_val
  end
end

def letrec_to_parameters_args_body(exp)
  let_to_parameters_args_body(exp)
end

def letrec?(exp)
  exp[0] == :letrec
end
```

それでは、実際に試してみましょう。

```
exp =
  [:letrec,
   [[:fact,
     [:lambda, [:n], [:if, [:<, :n, 1], 1, [:*, :
         n, [:fact, [:-, :n, 1]]]]]]],
   [:fact, 3]]
puts _eval(exp, $global_env)
```

正しく、6 が表示されましたね。

純粋関数型言語 – 代入はどこへ?

おめでとうございます。あなたは、この時点で関数型言語で必要な全ての要素を学んだと言えます。ただし“最小限の範囲内で”、という限定付きです。今までに学んできたプログラミング言語は、純粋関数型言語 (pure functional language) と言われ、代入と言った副作用の無い言語です。状態が変わらない、そもそも状態すら持たないために、変数を、それを束縛している値で入れ替えても問題ありません。最適化ではインライン展開と言われる手法です。関数呼び出しを減らすことが出来るためそれに関わるコストを削減できます。関数呼び出し時に、環境のスタックを積み上げたことを思い出して下さい。それが不要になります。もしくは、どの順序で評価してもその値は変わらないため、それぞれを並列に処理することができます。最近、関数型言語が見直されるようになってきたのは、このような背景が多分にあります。

そうは言っても手続き型言語で代入を多用している方にとっては、本当に代入なしで複雑な処理を記述できるのか疑問に思われることでしょう。大丈夫です。実は、今まで書いてきた Ruby のプログラムも簡単に代入なしで書くことが出来ます。本質的に代入が必要なのは、たった一ヶ所、set_extend_env! で使われているハッシュへの代入のみです[*1]。他の箇所では引数で渡された配列の中を代入などにより変更していません。したがってこれ以外の代入は、let 式相当の機能で置き換えても問題ありません (Ruby には let 式相等の機能がありませんが...)。

本文書で考えるプログラミング言語にはあえて代入は導入しません。下手に代入があるとそれに頼ってしまうことがあるからです。新しい関数型言語の考え方に慣れるためにこの言語で色々と処理を書いて見てください。

[*1] 関数名の最後の!は、quote の略で、Scheme では一般的に副作用がある関数の最後にクオート文字!をつけてこれを区別しています。これを継承しました。

関数型言語と再帰 – for 文はどこへ?

再帰は関数型言語にとって大きな意味を持ちます。手続き型言語の for 文などループに相当する重要な制御構文です。関数型言語の特徴は、データ構造に着目して再帰的に処理をすることが多い点です。

実は今回の対象としているプログラミング言語も、数字など基本的な式を組み合わせて一つのプログラムとなるように設計されています。これを式の構成に関する帰納法と呼びます。`eval_let` などの中で、評価すべきプログラムを構成している要素に分け、それぞれの要素に対して再帰的に `eval` していることを確認してみて下さい。

また、先の例では階乗を求めるプログラム fact を使いました。これも見方を変えると再帰的に定義された整数に従った構成に関する帰納法とも言えます。整数は、0 という整数が存在すること、またある整数が存在したとき +1 したものが次に大きな整数として定義されます。この定義に従い、0 のときの解を示し、またある整数が与えられた時、それより一つ小さい整数の解を用いて求めるものを定義することで、全ての場合の解を求めることができます。

その他にもまだ扱っていませんが、リストは空リストであるか、リストに先頭の要素を加えたものである、と再帰的に定義されます。一般的なリストを扱う関数はこの定義に従い、与えられたリストが空リストであった場合の処理と、与えられたリストの最初の要素に対する処理を記載し、残りのリストは再帰を使って定義します。わかりづらいと思いますので、4 章の p. 51 のリストの項目 (特に `length` 関数) を見てからもう一度この文章を読んで見てください。

まとめ

この章では次のことを学びました。

- 再帰関数を実現するための方法
- 再帰関数の評価値であるクロージャは、その中の環境でクロージャ自身を参照する。
- これまでに作成してきた μSchemeR は代入のような副作用がない純粋関数型言語である
- 手続き言語のループに相当するものを関数型言語では再帰を用いる

第4章
少し言語を拡張して

前章までで本質的なことはすべて学んだと言いました。ただし、このままでは実際のプログラムが書きづらいのも確かです。この章では、我々の言語を書きやすくするため、いくつかの機能を追加していきます。重要な機能は前章で学び終わっていますので、気軽に読んでみて下さい。

リスト

ここではリストを扱えるようにします[*1]。ここで紹介する関数は、本来の Scheme ではリストに限らず使えるものですが、今回はリストを対象に機能を限定します。

リストは空リストもしくはリストに先頭の要素を加えたものとして構造的帰納法で定義されます。Ruby では配列で表現します。

null?は与えたリストが空リストか調べるものです。空リストは:nil で表されます。その他、後で説明するリスト用の組み込み関数も環境として定義しておきます。

```
def null?(list)
  list == []
end

$list_env = {
  :nil => [],
  :null? => [:prim, lambda{|list| null?(list)}],
  :cons => [:prim, lambda{|a, b| cons(a, b)}],
```

[*1] 今回我々が作成しようとしている Scheme の元言語である Lisp という名は List Processing すなわちリスト処理から由来しています。それほど、リストを扱うのが得意な言語なのです。

```
  :car => [:prim, lambda{|list| car(list)}],
  :cdr => [:prim, lambda{|list| cdr(list)}],
  :list => [:prim, lambda{|*list| list(*list)}],
}

$global_env = [$list_env, $primitive_fun_env,
    $boolean_env]
```

cons は、リストに先頭要素を加えます。リスト以外のものに要素を加えようとすると、我々の不十分な処理系はエラーを返します。

```
def cons(a, b)
  if not list?(b)
    raise "sorry, we haven't implemented yet..."
  else
    [a] + b
  end
end
```

car, cdr はそれぞれリストの先頭の要素、および先頭の要素を除いたリストを返します。

```
def car(list)
  list[0]
end
```

```
def cdr(list)
  list[1...list.length]
end
```

list は、与えられたリストをそのまま返します。Ruby で可変長引数は配列で渡されますので、配列をリストとして用いているため、そのままの値を使うことができるためです。

```
def list(*list)
  list
end
```

定義

ここでは定義を扱います。次の式を考えてみましょう。

```
[:define, :id, [:lambda, [:x], :x]]
```

これは、:id を引き数で与えられたものをそのまま返す関数として定義するものです。したがって、その後、下の式を評価すると、

```
[:id, 3]
```

3 が返されることを期待します。

もう一つ、異なる記述方法の定義を導入します。このプログラムは上の定義と同じ意味を持ちます。

```
[:define, [:id, :x], :x]
```

みなさんには、こちらの方がなじみがあるかと思います。関数名に続き仮引数と、それに続く関数のボディから成ります。

これを実装するためにはどうすれば良いでしょうか。変数に定義する値を束縛した環境を付け加えます。ポイントは、その後の評価でもその定義が使えるように環境を書き換える必要があることです。また、すでに変数が束縛されている場合には値を書き換えるようにします。これらは環境を代入により上書きします (関数名が!を使う関数を呼んでいる点に注意しましょう)。

```
def eval_define(exp, env)
  if define_with_parameter?(exp)
    var, val = define_with_parameter_var_val(exp)
  else
    var, val = define_var_val(exp)
  end
  var_ref = lookup_var_ref(var, env)
  if var_ref != nil
```

```
    var_ref[var] = _eval(val, env)
  else
    extend_env!([var], [_eval(val, env)], env)
  end
  nil
end

def extend_env!(parameters, args, env)
  alist = parameters.zip(args)
  h = Hash.new
  alist.each { |k, v| h[k] = v }
  env.unshift(h)
end

def define_with_parameter?(exp)
  list?(exp[1])
end

def define_with_parameter_var_val(exp)
  var = car(exp[1])
  parameters, body = cdr(exp[1]), exp[2]
  val = [:lambda, parameters, body]
  [var, val]
end

def define_var_val(exp)
  [exp[1], exp[2]]
```

```
end

def lookup_var_ref(var, env)
  env.find{|alist| alist.key?(var)}
end

def define?(exp)
  exp[0] == :define
end
```

準備ができましたので下のようなリストを扱うプログラムをいろいろと実行してみましょう[*2]。

```
[:define, [:length, :list],
 [:if, [:null?, :list],
  0,
  [:+, [:length, [:cdr, :list]], 1]]]
[:length, [:list, 1, 2]]
```

[*2] 実行する前に p.60 で示す special_form?と eval_special_form を追記する必要があります。

cond 式

条件分岐は if 式で記述出来ますが、条件が多くなると、if のネストが深くなり、プログラムが見づらいものになっていきます。そこで次のような式を実行できる cond を導入します。

```
[:cond,
 [[:>, 1, 1], 1],
 [[:>, 2, 1], 2],
 [[:>, 3, 1], 3],
 [:else, -1]]
```

この式は上から、リストの左の条件式を順に評価し、真になればその右の式を評価値を cond 式の値とします。偽であれば、その下のリストに対して同様のことを行います。:else があった場合は、その右の式を値とします。リストはいくつあっても構いません。この場合は 2 が返り値となります。

この実装は if 式に書き換え、それを評価するだけです。

```
def eval_cond(exp, env)
  if_exp = cond_to_if(cdr(exp))
  eval_if(if_exp, env)
end

def cond_to_if(cond_exp)
```

```
  if cond_exp == []
    ''
  else
    e = car(cond_exp)
    p, c = e[0], e[1]
    if p == :else
      p = :true
    end
    [:if, p, c, cond_to_if(cdr(cond_exp))]
  end
end

def cond?(exp)
  exp[0] == :cond
end
```

パーサー

ここまでプログラムを書いてきて、プログラムが書きづらかったことでしょう。プログラムを Ruby で評価しやすいように、Ruby の配列を用いてパーサーを省略するとともに、Ruby のシンボルを用いてシンボルテーブルを省略していたためです。Lisp や Scheme はよくカッコのお化けと言われます

が、今まさにその表記法に移る時が来ました。

次のように“ () ”を使う本来の Scheme の記述方法プログラムを Ruby の文字列として入力すると、今までと同じように“ [] ”や“ , ”を使った Ruby のデータ型に変換するものを作ります。変換後のデータを評価させれば今までどおりの結果が得られますので、ユーザは“ () ”を使う普通の Scheme のプログラムを入力できるようになります。

```
_eval(parse('(define (length list) (if (null?,
    list) 0 (+ (length (cdr list)) 1)))'),
      $global_env)
puts _eval(parse('(length (list 1 2 3))'),
    $global_env)
```

これは、“ [”, “] ”を “ (”, “) ”に、変数を Ruby のシンボルに置き換え、“ , ”を削除することにより実現します。

```
def parse(exp)
  program = exp.strip().
    gsub(/[a-zA-Z\+\-\*><=][0-9a-zA-Z\+\-=*]*/, '
        :\\0').
    gsub(/\s+/, ', ').
    gsub(/\(/, '[').
    gsub(/\)/, ']')
  log(program)
```

```
  eval(program)
end
```

quote

次に追加する機能は quote です。次のようにリストを引数として与えるときなどで便利です。

```
puts _eval(parse('(length (quote (1 2 3)))'),
    $global_env)
```

quote の引数は評価せずに引数をそのまま評価値として返します[*3]。

```
def eval_quote(exp, env)
  car(cdr(exp))
end

def quote?(exp)
```

[*3] quote は通常、' を使って簡易に記載できます。すなわち、(quote 1 2 3) は'(1 2 3) と同じものです。これは処理系が' を読み込んだ時に、quote に展開することで実現されています。腕に自信のある方はぜひこの機能の実装にチャレンジしてみて下さい。Ruby 1.9 から正規表現でカッコの対応付けが可能になっています。

```
  exp[0] == :quote
end
```

それでは今まで、拡張してきた機能が動作するようにしましょう。

```
def special_form?(exp)
  lambda?(exp) or
    let?(exp) or
    letrec?(exp) or
    if?(exp) or
    cond?(exp) or
    define?(exp) or
    quote?(exp)
end

def eval_special_form(exp, env)
  if lambda?(exp)
    eval_lambda(exp, env)
  elsif let?(exp)
    eval_let(exp, env)
  elsif letrec?(exp)
    eval_letrec(exp, env)
  elsif if?(exp)
    eval_if(exp, env)
  elsif cond?(exp)
    eval_cond(exp, env)
```

```
  elsif define?(exp)
    eval_define(exp, env)
  elsif quote?(exp)
    eval_quote(exp, env)
  end
end
```

REPL

最後にインタープリタと呼ばれるにふさわしい処理を付け加えます。インタープリタはユーザと対話しながらプログラムを作成することができる点に特徴があります。この機能、すなわち、ユーザから入力を読み取り (Read)、その結果を評価し (Eval)、その結果を表示する (Print) ことを繰り返す (Loop) 機能です。これは頭文字をとって、REPL とも呼ばれます。

実現は上の機能をそのまま単純に実装します。ここで、pp[*4] という式を整形する処理を新たに定義しています。

```
def repl
  prompt = '>>> '
```

[*4] pretty print の略です。

```
  second_prompt = '> '
  while true
    print prompt
    line = gets or return
    while line.count('(') > line.count(')')
      print second_prompt
      next_line = gets or return
      line += next_line
    end
    redo if line =~ /\A\s*\z/m
    begin
      val = _eval(parse(line), $global_env)
    rescue Exception => e
      puts e.to_s
      redo
    end
    puts pp(val)
  end
end

def pp(exp)
  if exp.is_a?(Symbol) or num?(exp)
    exp.to_s
  elsif exp == nil
    'nil'
  elsif exp.is_a?(Array) and (exp[0] == :closure
      )
```

```
    parameter, body, env = exp[1], exp[2], exp[3]
    "(closure #{pp(parameter)} #{pp(body)})"
  elsif lambda?(exp)
    parameters, body = exp[1], exp[2]
    "(lambda #{pp(parameters)} #{pp(body)})"
  elsif exp.is_a?(Hash)
    if exp == $primitive_fun_env
      '*prinmitive_fun_env*'
    elsif exp == $boolean_env
      '*boolean_env*'
    elsif exp == $list_env
      '*list_env*'
    else
      '{' + exp.map{|k, v| pp(k) + ':' + pp(v)}.
          join(', ') + '}'
    end
  elsif exp.is_a?(Array)
    '(' + exp.map{|e| pp(e)}.join(', ') + ')'
  else
    exp.to_s
  end
end
```

それでは実行してみましょう。

```
>> repl
>>> (define (fib n) (if (< n 2) n (+ (fib (- n
    1)) (fib (- n 2)))))
```

```
nil
>>> (fib 10)
55
```

実行出来ました。

今までよりはずいぶん楽になるのではないでしょうか。

その他

他に不便なところはありませんか。まだまだあるでしょう。実現していない機能は多々あります。named let や let*などの機能を調べそ実装にトライしてみて下さい。友達に速度が遅いと言われたら、コンパイラを作りましょう。Haskell のように遅延評価でないとと言われたら、遅延評価にしてしまいましょう。他の言語のこの機能がない、と言われたら、自分で追加してしまいましょう。自分でプログラミング言語を作ったからこそ味わえるおもしろさです。大いに使い倒して下さい。

まとめ

この章では、プログラミングを便利にするような次の機能を実現しました。

- define による定義
- リスト
- cond 式
- パーサー
- quote
- REPL

第5章 次のステップ

Scheme in μSchemeR にチャレンジ

既にみなさんは関数型言語の本質は理解していますが、まだ関数型言語のプログラミングに慣れているとは言えません。プログラミングは書いて慣れるものですので、プログラミング言語の原理を理解しているだけでは不十分です。様々なプログラムを書いてみて、その考え方を学んでください。

その教材を一つご紹介します。今回は Ruby を用いて

Scheme のサブセットを実現しました。この Ruby のプログラムを Scheme で書いて見ましょう。しかも、ただ Scheme で書くのはつまらないので、今回開発した μSchemeR で動作させて下さい。これが実現すれば、Ruby 上で μSchemeR が動き、その上で今回作成する (今回 Ruby で作成したのと同等の)Scheme 処理系が動き、その上でユーザが与えた Scheme プログラムが解釈、実行されることになります。想像しただけでエキサイティングになりませんか。

足りない機能があれば、μSchemeR の処理系を Ruby で書き足しながら実現してみてください。

実現する上でのヒントです。2ヵ所で代入が必要になります。`letrec` と `define` です。代入を実現する`:set!`は次のとおり実現できます。与えられた変数を与えられた値に束縛するよう環境を書き換えるものです。`define` と異なる点は、代入する変数がまだ使われていない場合エラーとする所です。`special_form?`、`eval_special_form` も書き換えるのを忘れないで下さい。

```
def eval_set!(exp, env)
  var, val = setq_to_var_val(exp)
  var_ref = lookup_var_ref(var, env)
```

```
  if var_ref != nil
    var_ref[var] = _eval(val, env)
  else
    raise "undefined variable:'#{var}'"
  end
  nil
end

def setq_to_var_val(exp)
  [exp[1], exp[2]]
end

def setq?(exp)
  exp[0] == :setq
end
```

```
(let ((x 1)) (let ((dummy (set! x 2))) x))
```

2ヵ所で代入が必要と言いましたが、逆に言えばそれ以外の場所で代入は使ってはいけません。関数型言語の醍醐味を存分楽しんで下さい。

SICP にチャレンジ

今のみなさんであれば、SICP と呼ばれる本、Structure and Interpretaion of Computer Programs 2nd ed.: 訳本 計算機プログラムの構造と解釈 第 2 版 [1] を十分読める実力を持っています。

むしろアドバンテージがありますので、使っている機能を μSchemeR に実装していきながら読み進めるくらいの余裕があることでしょう。

SICP の後半では、この文書で行ったように Scheme の処理系を実装します。それを使った言語機能の拡張は間違いなく、みなさんのためになります。ぜひトライしてみて下さい。

シンタックスの変更

せっかく自分が作ったプログラミング言語なのにシンタックスがカッコ悪い (もしくはカッコが多い) と友達にバカにされませんでしたか?すでに十分プログラミング言語について理解しているあなたは“そんなのは見掛け上の文法の話で中身は変わらない。機能を見てくれ。”と言うかもしれませんが、残

念ながら反応はあまり変わりません。そこでクールなシンタックスに変えて見返してみましょう。I-expression という記法があり[*1]、これは Python のようにインデントで文法を解釈するというものです。

例えば、階乗を求めるプログラムは次のように書けます。カッコがずいぶん少なく、モダンな感じのプログラミング言語に見えませんか?

```
define (fact x)
 if (= x 0) 1
  * x
   fact (- x 1)
```

これを次のようにして実現してみましょう。脚注の URL に I-Expression を解釈するプログラムが記載されています。このプログラムを実行できるように μSchemeR を拡張します。その後、拡張された μSchemeR で I-expression で書かれたプログラムを解釈し、得られたプログラムを、μScheme 実行します。

[*1] http://srfi.schemers.org/srfi-49/srfi-49.html

参考文献

[1] 計算機プログラムの構造と解釈. ピアソンエデュケーション, 2000.

この文書の最新版は、https://github.com/ichusrlocalbin から取得できます。