Schemeの始めの1歩

by T. Sakuragawa

- LISPインタプリタの動作
- LISPの基本的なデータ型
- LISPの複合データ型
- 変数
- 大域変数の操作
- 新しい函数の宣言
- 組み込み述語とブール函数
- 条件式
- 再帰呼び出し
- 再帰的函数を定義する際の考え方

LISPインタプリタの動作

1. プロンプト(入力促進記号)を出力
2. ユーザからの入力を待つ
3. ユーザが式を入力(複数行に渡っても可, read)
4. 式を内部表現(セル表現(未説明, 2分木))に直す
5. 内部表現で表された式の値を求める(評価, evaluation)
6. 求めた値を表示する(print)
7. 1に戻る。read-eval-print loopと呼んだりする

LISP(Scheme)の基本的なデータ型(の一部)

- シンボル: abc, x, define, 1+

  定数として使う場合は'(シングルクォート)を付ける。ないと変数として解釈
- 数値(整数, 有理数, 実数(浮動小数点数), 複素数)

  10, -19878767546434244 1, 2/3, 3.14, -2i
- 文字列: “abc“, ““
- 文字: #\a, #\U+14
- 真偽値: #t, #f
- シンボル以外は評価してもそれ自身が値となる

実行例1

- > 'abc ;ここで>はプロンプト

  abc
- > (quote abc) ;'abcは(quote abc)の略記法

  abc
- > (/ 2 3)

  2/3
- > “string“

  “string“
- > (* “a“ “b“) ; を評価するとエラーとなる

  Schemeは動的片付けの言語なので実際実行してみて(データ型の)エラーを検出する

LISP(Scheme)の複合データ型1

- リスト
  (a b c), (), (a b (c d) (e f g)), (a . b), (a b . c)
  基本は2つのデータのペア(二分木)
  型が異なっていてもよい
  連ねることで任意個のデータの列を表せる
  入れ籠にできる(再帰的データ構造)
  LISPの式はリストである(他の言語との最大の違いの一つ)
  プログラムをデータとして加工できる
  メタサーキュラーインタプリタ(LISPで記述したLISPインタプリタ)(のもどき)を容易に書ける
  二分木で他の木も表そうという考え方
  S表現で表す(二分木との対応に慣れが必要)

実行例2

- > '(a b c) ；リストも定数とするには「'」が必要
  (a b c)
- > '(a b c (d e f))
  (a b c (d e f))
- > (car '(a b c (d e f))) ；これ以下の実行例はそのうち解説
  a
- > (cdr '(a b c (d e f))) ; 2分木を分解する組み込み函数がcar(カー),
  (b c (d e f)) ; cdr(クッダー)
- > (cons 'a '(b c d)) ; 2分木を組み立てるのがcons(コンス)
  (a b c d)

LISP(Scheme)の複合データ型2

- ベクタ(1次元配列)

  #(a b c), #(a b 1 (a c)), #(a #(b c))

  データの、長さがある列

  型が異なっていてもよい

  入れ籠にできる

  Schemeの場合2次元配列はない

  授業では最後の方にしか出てこない

Schemeの変数

- 変数(variable)という概念は大抵のプログラミング言語にある(例外:コンビネータ論理)
- 変数とは、直感的にはデータが入る箱で、大抵の場合名前を付ける
- プログラムが同じでも、変数に入っているデータによって実行結果が異なる(つまり計算機の動作に柔軟性をもたらす根本概念の一つ)
- 名前の有効範囲(スコープ)がプログラム(あるいはファイル)全体(一部のこともあるが)である大域変数(global variable)
- 有効範囲がプログラムの一部(一つの函数など)となる局所変数(local variable)
- 言語によってはさらに種類がある
- 局所変数は、メモリ上の格納領域がスタック上などに一時的に確保される場合が多い
- 別の場所の異なる局所変数に同じ名前をつけることが可能(変数名を考えるのが楽)
- 変数名を組織的に変えてもプログラムの意味は同じ(λ 計算の α 変換)
- 動的型付けの言語(Schemeはこちら)では、変数には型が大抵ない
- 何度でも変数に値を代入できる言語(Schemeはこちら)と、(領域が確保されるごとに)1度しか代入できない言語がある
- Schemeの場合にはシンボルが変数を表す(シンボルそのものとして解釈してほしい場合には「'」を前に付ける)

Schemeの大域変数操作

- > (define x 10)

  大域変数xを初期値10として宣言(LISPの方言により書き方は異なる。後述のset!も)
- > x

  単にxと書けば変数xに入ったデータが値となる(変数xを参照する、などと言う)
- > (+ 20 4 (* x x) (*3 x))

  20+4+x^2+3xの値を計算する式
- > (set! x (+ 2 5))

  変数xに新しい値7を代入する(assign)
- > (set! x (+ x 1))

  xの値を1増やす(increment)
- set!は副作用(この場合にはやりたいことが変数への代入そのものなので通常の日本語では副作用というか本作用なのだが、他にも影響がある動作なので副作用というのである)のある操作であり、使用すると函数的なプログラミングではなくなる

Schemeにおける函数の宣言

- LISPにおけるプログラミング = 函数を次々に宣言していくこと
- あらかじめ定義されている函数が多数ある(組み込み函数)
- (define (函数名 仮引数1 仮引数2 …)

  (仮引数たち(の一部)を含む式(本体)))

  (基本形)

  仮引数(パラメータ)とは、函数呼び出しをしたときに実際の引数(実引数)の値が入る変数

  仮引数は局所変数である

  同じ名前の大域変数は本体の部分から参照できなくなる

  実は函数宣言は大域変数の宣言の一種(Schemeは大域変数と函数の区別がないLISP, ただし今はこれを意識しなくてもよい)
- > (define z 3) ; であったとして、

  > (define (fun1 x y) (+ (* x x) (* y y) z 2))

  など。(fun1 3 2)を評価するとx=3, y=2としてx^2+y^2+5を計算する

Schemeの述語

- 真偽値(#t, #f)を返す函数を「述語」と呼ぶ
- 組み込み述語は「?」で終わる。例外は=, <, >, <=, >=(数値の比較)
- and, or(ここまでn-ary), notを組み合わせて複雑な論理式を構成できる
- and, orは複数個の実引数を持つ場合、それぞれ左から評価して行って最初に#f, #tであることがわかるとそれより右側にある実引数の評価を行わない。これは通常の函数の場合と異なる動作である。副作用がある場合に注意が必要。

実行例3

- > (= 1 1)

  #t
- > (= 1 0)

  #f
- > (> 2 1)

  #t
- > (>= 1 1)

  #t
- > (and (= 1 1) #t)

  #t
- > (not (= 1 1))

  #f

Schemeの条件式1

- (if 条件式 式1 式2) ; が基本形
- 条件式の部分には述語を函数とする式か、それらをand, or, notで組み合わせた論理式を記述
- 条件式が#f以外だと式1を評価
- #fだと式2を評価
- もう片方は評価しない
- 評価した式の値がif文の値となる
- 式1に副作用がある場合には、式2を省略する場合がある
- この場合には#fの時には特に何もしない。if文の値は存在しない
- > (define (f x) (if (> x 0) (* x x) 0)); など

Schemeの条件式2

- if文を幾つも連ねて記述する場合、condを使うことでスッキリと記述できる場合がある
- (cond (条件式1 式1)

  (条件式2 式2)

  .

  (else 式n)) ;は、

(if 条件式1 式1

(if 条件式2 式2

.

(if 条件式n-1 式n-1 式n) .))

と同じ意味

再帰呼び出し(recursive call)

- 函数定義の中に現れる、その函数自身を用いた函数呼び出し
- プログラミングの基本的な技法の一つ
- LISPでは必須の考え方
- 再帰的データ構造と相性が良い
- 複数の函数がお互いに呼び出す相互再帰呼び出しも可能
- 繰り返しを記述できる
- 循環定義になっているとは限らない(なっていると停止しない函数となる)

再帰的函数を定義する際の考え方

- 再帰的函数とは、再帰呼び出しを行うように宣言された函数
- もっとも簡単(あるいは基本的)と考えられるデータの場合、再帰呼び出しを行わないように記述する
- 再帰呼び出しの部分を記述する際、呼び出された函数が仕様(specification, 函数が満たすべき性質)通り動作すると仮定して記述する
- ただし、再帰呼び出し時の実引数(実際に渡るデータ)が、ある意味少し簡単になっているように記述する
- そうなっていないと停止しない函数となる
- 末尾再帰呼び出し(tail recursion, 再帰呼び出し後返ってきた値をそのまま自身の値として返すような再帰呼び出し)の場合には効率的な実現(implementation, この場合にはインタプリタの方式あるいはコンパイラの吐くコード)が可能

再帰的定義の例

- (define (fact x)

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

一番簡単なのがx=0の場合で値が1

再帰呼び出しを行う際に引数が少し簡単になっている(x-1)

xが負の場合には停止しない

LISPの処理系の多くは任意多倍長整数演算(bignum)を行えるので、3000!なども問題なく求められる

末尾再帰呼び出しではない(最後にxを掛けるため)

再帰的定義の例(続き1)

- 同じ函数をcondを使って書くこともできる(この場合はむしろ長くなる)。
- (define (fact x)

  (cond ((= x 0) 1)

  (else (* x (fact (- x 1))))))
- この例の記述の際、以下のようにしている

a) condの行は何文字か字下げ(indent)している

b) elseの行の行頭の(を前の行の同じレベルの(の位置に合わせている

このように人間に読みやすく記述するのが普通

LISP用のエディタの場合、カッコ対応を取ったりインデントしたりする機能が付いている場合が多い

再帰的定義の例(続き2)

- (define (fact x) (fact1 x 1))

  (define (fact1 x y)

  (if (= x 0) y (fact1 (- x 1) (* x y))))

  としてもよい。こうすると末尾再帰呼び出しとなり、実行時の函数呼び出しについてのメモリ使用量(スタックなどの使用量)が線形(リニア)から定数に改善される
- ただし、プログラムの可読性はこの場合減少している