Common Lisp の スコープ と エクステント

株式会社数理システム 知識工学部
黒田 寿男

2008 年 09 月 17 日

概 要

Common Lisp を学ぶさいに壁となる一つが、スペシャル変数に代表される、プログラムの実行につれて動的に見え隠れする実体への理解です。理解を妨げる理由はいくつか考えられますが、中でも大きなものとして、他の多くのプログラミング言語は Common Lisp のような動的な側面を持っておらず、Common Lisp を第 2 プログラミング言語として学ぶ人にとって、今まで習ってきたことに類比させることが困難である、というのが挙げられます。こういった類比を無理矢理やったような解説がネット上でいくつか見受けられますが、理解を困難にこそすれ助けているとは思えません。

Common Lisp を理解する上で壁にぶつかったさいには、ネットに散らばっているような解説を読む必要はなく、CLtL2 あるいは CLHS (Common Lisp Hyper Spec) をわかるまでじっくり読むしかないのですが、どちらも技術的厳密にしかも英語で書かれているため決っして楽に読めるものではありません。

本稿は CLtL2 の 3 章 Scope and Extent をほぼそのまま日本語訳し、理解を助けると思われるいくつかの解説を補いました。CLtL2 を読む際のガイドブックとなる目的で書かれたものです。最終的には原著 (ISBN-1-55558-041-6) にあたることで理解を深めてください。

スコープとエクステント

Common Lisp の様々な特徴を説明する際にスコープとエクステントという概念がしばしば有用となる。これらの概念は、あるオブジェクトや要素をプログラムの離れたところから参照しようとするときに必要になる。スコープ (scope) は参照できる空間的すなわち字面上の領域を指し、エクステント (extent) は参照が継続する時間を指す。

例として以下の簡単なプログラムを考えてみる。

```
(defun copy-cell (x) (cons (car x) (cdr x)))
```

パラメータ x のスコープは defun の ボディ部全体となる。この defun のボディ部以外からパラメータ x を参照する方法というものはない。同じように、パラメータ x のエクステントは (どの特定の copy-cell 呼び出しにおいても) この関数が呼ばれてから抜けるまでの間である。 (一般にはパラメータのエクステントは関数を抜けた後も続くことがあるが、この単純な例においてはそれはない。)

Common Lisp では、参照されうる実体 (entitiy) は言語上の構成要素[1]を実行することによって確立 (establish) され、実体のスコープとエクステントはその実体を確立するところの構成要素と (その構成要素が実行されている間の) 時間に相対的に決まる。ここでの議論において「実体」という語は、シンボルやコンスなどの Common Lisp データオブジェクトだけでなく、変数の束縛 (レキシカル、スペシャルの両方とも) や キャッチャー (catcher) 、飛び先 (go target) なども指す。実体と実体の名前を区別することは大切である。以下のような関数定義において、

[1]原文は construct。defun や let、catch や block などがその例。「構成要素」という語がわかりにくければ、「フォーム」や「文」に置き換えて読んで良い。本稿においても訳の都合で construct を 「文」とした箇所がある。

```
(defun foo (x y) (* x (+ y 1)))
```

この関数の一番目のパラメータを指すのは x というただ一つの名前だが、新たな束縛は関数が呼ばれるごとに確立される。ここで束縛とは特定のパラメータ代入のことを言う。名前 x を参照した際の値はそれが現れるスコープだけでなく (上記 foo の例だと、ボディ部で使われている x は関数定義時のパラメータスコープ内に現われている) 特定の束縛つまり代入にも依存する。 (この場合は、それぞれの関数呼び出しに応じてそれぞれの代入値が与えられる。) さらに詳しい例を後で見ることにする。

スコープとエクステントの種類

Common Lisp のスコープとエクステントには以下の種類がある。

- レキシカルスコープ。ここでは確立された実体への参照はレキシカルに (つまり字面上) その実体を確立するところの構成要素に含まれているプログラムのある部分だけから可能である。そういった構成要素は典型的にはボディ部と呼ばれる部分を持ち、確立された全ての実体のスコープは、そのボディとなる。

  例: 関数のパラメータ名は一般にレキシカルスコープをもつ。
- 無限スコープ。プログラムのどこからでも参照できる。
- 動的エクステント。実体が確立されてからそれが解かれるまでの期間参照できる。原則として実体は、それを確立する構成要素の実行が正常であれ異常であれ終了した時点で参照できなくなる。したがって動的エクステントを持つ実体はそれを確立する構成要素をネストさせるのに対応して、ちょうどスタックのような働きをする。

  例: with-open-file 文はファイルを開き、ファイルへの連結を表わすストリームオブジェクトを生成する。ストリームオブジェクトは無限エクステントを持つ。しかし、開いたファイルへの連結は動的エクステントを持つ。with-open-file の実行を抜けると、それが正常であれ異常であれストリームは自動的に閉じられる。解説 1 に例を補った。

  例: スペシャル変数への束縛は動的エクステントを持つ。
- 無限エクステント: 実体は参照が残っている限り存在し続ける。 (なお、Common Lisp の実装は既にもう参照することが無いと保証できる実体についてはそれを破棄することが許されている。ガーベッジ・コレクションと呼ばれるものが即ちこのことである。)

  例: Common Lisp のデータオブジェクトはほとんどが無限エクステントを持つ。

  例: レキシカルスコープな関数パラメータの束縛は無限エクステントを持つ。 (対照的に、Algol のレキシカルスコープなプロシージャパラメータは動的エクステントを持つ。) 以下の関数定義は、

  ```
  (defun compose (f g)
    #'(lambda (x)
        (funcall f (funcall g x))))
  ```

  二つの引数をとり、すぐに関数をその値として返す。このとき f と g のパラメータ束縛は消えない。なぜなら返された関数が次に呼ばれたとき、この束縛への参照をすることが未だ可能だからである。したがって、

  ```
  (funcall (compose #'sqrt #'abs) -9.0)
  ```

  は、値 3.0 を返す。 (この例は Algol あるいは他の Lisp 方言では必ずしもうまく動かない)

以上に加えて、無限スコープ + 動的エクステント の意味で 動的スコープ という言葉が使われることがある。その流れで、スペシャル変数は動的スコープを持つ、というふうに言うことがある。なぜならスペシャル変数は 無限スコープと 動的エクステントを持ち、その束縛が有効である限り、プログラムのどこからでも参照することができるからである。[2]

シャドゥイング

ここまでは シャドゥイング について考慮してこなかった。実体への離れたところからの参照は何らかの名前を使う。もし別の二つの実体が同じ名前を持っていると、最初のものが二つ目によってシャドゥされて、一つ目を参照することはできなくなる。

レキシカルスコープの場合、もし二つの構成要素がネストしていて別々の実体に同じ名前が割当てられていれば、内側の構成要素内で参照できる実体は内側で確立されたものとなる。つまり内側が外側をシャドゥする。内側の構成要素の外でかつ外側の構成要素の内では、外側の構成要素で確立された実体が参照される。例えば、

```
(defun test (x z)
  (let ((z (* x 2))) (print z))
  z)
```

let 文によってつくられた変数 z の束縛は関数 test のパラメータ束縛をシャドゥする。print が参照している z は let による束縛である。一方、関数の最後に出てくる z は パラメータ z を参照する。

動的エクステントの場合、二つの実体が時間的に重なって現れたとすると、一方が他方の中でネストしているはずである。これは Common Lisp の設計上こうなる。

動的エクステントでの名前による実体の参照は、常に、最後に確立されてまだ消えていないものについてなされる。例えば、

```
(defun fun1 (x)
  (catch 'trap (+ 3 (fun2 x))))

(defun fun2 (y)
  (catch 'trap (* 5 (fun3 y))))

(defun fun3 (z)
  (throw 'trap z))
```

(fun1 7) の呼び出しを考える。結果は 10 である。throw が実行されたとき、trap という名前で二つの別々のキャッチャーが存在する。一つは fun1 で確立されたもので、もう一つは fun2 で確立されたものである。後者がより最近に確立されているために、値 7 が fun2 の catch の値として戻される。fun3 から見ると fun2 の catch が fun1 のそれをシャドゥしている。仮に fun2 が次のように定義されていたとすると、

```
(defun fun2 (y)
  (catch 'snare (* 5 (fun3 y))))
```

[2]なお、動的スコープ という言葉は誤称であり、本稿では使わないことにする。

二つのキャッチャーは別の名前を持つので、fun1 のキャッチャーがシャドゥされることはない。したがって結果は 7 となる。

原則として、本稿ではスコープとエクステントについて述べるに留め、シャドゥイングの可能性については暗黙の了解があるものとする。

スコープとエクステントの規則

Common Lisp におけるスコープとエクステントについての重要な規則は以下。

- 変数束縛は一般にレキシカルスコープと無限エクステントを持つ。
- dynamic-extent 宣言がある変数束縛もまたレキシカルスコープと無限エクステントを持つが、束縛された値であるオブジェクトが動的エクステントとなることがある。(dynamic-extent 宣言はプログラマの責任において、通常の無限エクステントではなくたとえ動的エクステントしか持たないとしても正しく動くように書く必要がある。) 解説 2 に例を載せた。
- symbol-macrolet によるシンボルマクロ名への変数束縛はレキシカルスコープと無限エクステントを持つ。
- スペシャル宣言された変数束縛は無限スコープと動的エクステントを持つ。
- 関数名束縛、例えば flet や labels によるものは、レキシカルスコープと無限エクステントを持つ。
- dynamic-extent 宣言がある関数名束縛もまたレキシカルスコープと無限エクステントを持つが、束縛された値である関数オブジェクトが動的エクステントとなることがある。
- macrolet によって確立される関数名束縛 [3]はレキシカルスコープと無限エクステントを持つ。
- コンディションハンドラとリスタートは無限スコープと動的エクステントを持つ。解説 3 を見よ。
- catch や unwind-protect で確立されるキャッチャーは無限スコープと動的エクステントを持つ。解説 4 を見よ。
- block 文で確立される 飛び出し点 (exit point) はレキシカルスコープと動的エクステントを持つ。(このような飛び出し点は do, prog など他の繰り返し文でも確立される。)
- tagbody 中のタグによって名前を与えられ、go 文で参照される飛び先 (go target) は、レキシカルスコープと動的エクステントを持つ。(このような飛び先は do, prog など他の繰り返し文でも確立される。)
- nil や pi などの定数は無限スコープと無限エクステントを持つ。

レキシカルスコープの規則は、function 文に表われるλ式がそのλ式から見えている非スペシャル (non-special) な変数についての閉包 (closure) となることを示す。つまり、λ式で表わされた関数は、字面上有効などの非スペシャル変数も参照でき、そのことはそれらの変数束縛を確立させた構成要素を抜けた後でも有効だ、ということである。先に見た compose の例はこれを示すものの一つである。また、スペシャル変数については、いくつかの Lisp 方言がそうであるようにその束縛は閉包されない。

レキシカルスコープを扱う構成要素はそれが確立する実体に対して実行のたびに実質的に新たな名前を与えると考えられる。したがって動的なシャドゥイングは起こらない (レキシカルなシャドゥイングは起こりうる)。これは動的エクステントと組み合わさったときに特に重要である。例えば、

[3]マクロ名は関数名と同じネームスペースをつかう

```
(defun contorted-example (f g x)
  (if (= x 0)
      (funcall f)
      (block here
         (+ 5 (contorted-example g
                                 #'(lambda ()
                                     (return-from here 4))
                                 (- x 1))))))
```

(contorted-example nil nil 2) の呼び出しを考える。結果は 4 が返る。実行中に contorted-example への呼び出しが三回起こる。そのうちの二回がそれぞれ block に挟まれて呼び出される。

```
(contorted-example nil nil 2)

  (block here1 ...)

    (contorted-example nil #'(lambda () (return-from here1 4)) 1)

      (block here2 ...)

        (contorted-example #'(lambda () (return-from here1 4))
                           #'(lambda () (return-from here2 4))
                           0)
          (funcall f)
                where f => #'(lambda () (return-from here1 4))

            (return-from here 4)
```

funcall が実行されるときに二つの別々の block 飛び出し点が、見た目は同じ名前 here を持って存在する。上のトレースでは、これら飛び出し点は解説の都合で添字で区別している。funcall の結果 return-from が実行されるときには、内側の (here2) ではなくて、外側の飛び出し点 (here1) が参照される。これはレキシカルスコープの規則であって、つまり、function 文 (ここでは #' で省略されている) が実行されて関数オブジェクトを生成するときに字面上見えている飛び出し点が、あとからその関数が funcall によって呼ばれたときに参照される。

もし、この例で、(funcall f) のかわりに (funcall g) と置くと、(contorted-example nil nil 2) の呼び出し結果は 9 となる。なぜなら funcall の結果、(return-from here2 4) が実行されることになって、内側の飛び出し点 (here2) から戻ってくるからである。そうすると、4 が 真ん中の contorted-example から返り、それに 5 が加えられ 9 となり、それが外側の block と一番外側の contorted-example の戻り値となる。重要なのは、飛び出し点の選択は内側外側ではなく、function 文が実行されたときの λ 式にレキシカルスコープ規則に則って包みこまれた情報に依存することである。

この contorted-example がうまく動くのは f で表わされる関数が飛び出し点のエクステント (有効期限) 内に実行されたからにすぎない。block の飛び出し点は非スペシャルな変数束縛のようにレキシカルスコープを持つが、無限ではなく動的エクステントを持つ点で違っている。実行が一旦 block の外に

出てしまうと、飛び出し点は無効となる。例えば、

```
(defun illegal-example ()
  (let ((y (block here #'(lambda (z) (return-from here z)))))
    (if (numberp y) y (funcall y 5))))
```

(illegal-example) が 5 を返すように思えるかもしれないが、それは以下の間違った論法による。let 文が変数 y を block の返り値に束縛する。この値はλ式の結果である関数である。y は数値ではないので 引数に 5 を与えられて呼び出される。return-from がこの値を here と名付けられた飛び出し点から返すだろう。そして、block から再び戻って y に 5 を与え、それが数値なので、(illegal-example) の返り値として戻される。

この議論は Common Lisp の飛び出し点が動的エクステントを持つために成り立たない。return-from の手前までの論法は合っている。しかし return-from の実行はエラーとなる。しかしそれは飛び出し点を参照できないせいではなくて、正しく参照しているのだがその飛び出し点はすでに無効となっているからである。

解説 1

with-open-file 文はファイルを開き、ファイルへの連結を表わすストリームオブジェクトを生成する。ストリームオブジェクトは無限エクステントを持つ。しかし、開いたファイルへの連結は動的エクステントを持つ。

本文の最後のほうに出てきた contorted-example と illegal-example に倣って考えるとわかりやすい。

```
(defun contorted-open-file (f)
  (if (not (null f))
      (funcall f)
      (with-open-file (stream "foo")
        (contorted-open-file #'(lambda () (read stream))))))
```

(contorted-open-file nil) を呼び出すと、contorted-open-file が二回呼ばれることになる。最初の呼び出し時には f が nil なので with-open-file がファイル “foo” を開き、ストリームオブジェクトを生成してそれを stream に束縛する。次に二回目の contorted-open-file が関数オブジェクトを引数に呼ばれる。この関数が呼ばれると stream からファイルの内容を読み込む (read を発行する)。今度は f が non-nil なので funcall で f を呼び出して得られた結果、すなわち ファイル “foo” に書かれてある最初の S 式が (contorted-open-file nil) の結果として返される。

この contorted-open-file がうまく動くのは f で表わされる関数がストリームオブジェクトの有効期限すなわちエクステント内に実行されたからにすぎない。with-open-file で確立されるストリームオブジェクトは非スペシャルな変数束縛と同じくレキシカルスコープを持ち無限エクステントを持つが、ファイルへの連結そのものは動的エクステントを持つ。実行が一旦 with-open-file の外に出てしまうと、ファイルへの連結は無効となる。例えば、

```
(defun illegal-open-file ()
  (let ((y (with-open-file (stream "foo") #'(lambda () (read stream)))))
    (funcall y)))
```

(illegal-open-file) の実行はエラーとなる。しかしそれはストリームオブジェクトを参照できないせいではなくて、正しく参照しているのだがそのストリームオブジェクトがすでに閉じられてしまっている (ファイルへの連結が無効となっている) からである。なお、上の read を pathname に置き換えてやって (illegal-open-file) を実行すると、エラーにならない。ストリームオブジェクトがレキシカルスコープを持ちかつ無限エクステントを持つことがわかる。

解説 2

dynamic-extent 宣言がある変数束縛もまたレキシカルスコープと無限エクステントを持つが、束縛された値であるオブジェクトが動的エクステントとなることがある。

次のようなプログラムを考える。

```
(defun example-dynamic-extent ()
  (declare (optimize (speed 3) (safety 0) (debug 0)))
  (let ((x (make-array 8 :element-type 'fixnum :initial-element 1)))
    (declare (dynamic-extent x))
    #'(lambda ()
        (loop for i across x
            sum i))))
```

let により導入された変数 x は declare 文 を使って dynamic-extent 宣言がされている。dynamic-extent 宣言があろうがなかろうが x の束縛はレキシカルスコープと無限エクステントを持つ。しかし、x に束縛されたオブジェクト (この場合は大きさ 8 のアレイ) は処理系によっては動的エクステントしか持たされない場合がある[4]。したがって、`(funcall (example-dynamic-extent))` の結果は保証されない。つまりこの例は間違ったプログラムである。理由は funcall で呼ばれた関数内で x が参照できないためではなく、また x の束縛が無効になってしまっているわけでもなく、ただ、x に束縛されているオブジェクトが動的エクステントを持つために example-dynamic-extent の実行を抜けたあと消滅してしまっている可能性があるからである。

なお、上の例から dynamic-extent 宣言を抜けば `(funcall (example-dynamic-extent))` の結果はどの処理系においても 8 となる。

[4]具体的に言うと dynamic-extent 宣言された変数へ束縛されているオブジェクトは (処理系によっては) ヒープではなくスタックに置かれる。したがって一旦関数を抜けるとそのオブジェクトは消滅してしまう。

解説 3

コンディションハンドラとリスタートは無限スコープと動的エクステントを持つ。

以下でコンディションハンドラが動的エクステントを持つことを例示する。
ここでも本文の最後に出てきた contorted-example に倣う。

```
(defun contorted-handler (f g x)
  (if (= x 0)
      (funcall f)
      (handler-bind ((error #'(lambda (c)
                                (declare (ignore c))
                                (return-from contorted-handler 4))))
        (+ 5 (contorted-handler g
                                #'(lambda () (error "Foo."))
                                (- x 1))))))
```

`(contorted-handler nil nil 2)` の呼び出しを考える。結果は 9 が返る。実行中に contorted-hander への呼び出しが三回起こる。そのうちの二回はそれぞれ handler-bind でコンディションハンドラを確立した後に呼び出される。

```
(contorted-handler nil nil 2)

  (handler-bind ... (return-from contorted-handler 4)          [1]

    (contorted-handler nil #'(lambda () (error "Foo.")) 1)

      (handler-bind ... (return-from contorted-handler 4)      [2]

        (contorted-handler #'(lambda () (error "Foo."))
                           #'(lambda () (error "Foo."))
                           0)
          (funcall f)

            (error "Foo.")

            (return-from contorted-handler 4)                  [2]
```

funcall が実行されて error でシグナルが上がるときに二つの別々のハンドラが存在するが、外側で確立されたハンドラ [1] は内側で確立されたハンドラ [2] によってシャドゥされる。これは、最後に確立されてまだ消えていないものが参照される、という動的エクステントの規則による。ちなみにこの例で、`(funcall f)` のかわりに `(funcall g)` と置いても結果は同じである。

解説 4

catch や unwind-protect で確立されるキャッチャーは無限スコープと動的エクステントを持つ。

ここで注意すべきなのは「キャッチャー」は Common Lisp プログラマが直接操作できるオブジェクトではない点である。catch 文において確立されるキャッチャーは名前を与えられるが、その名前からプログラマがキャッチャーオブジェクトを得る手段はない。unwind-protect で確立されるキャッチャーは名前すら与えられていない。以下の例で、

```
(let ((stream (open "foo")))
  (unwind-protect
      (read stream)                    ; protected form
    (when (streamp stream)             ; cleanup form
      (close stream :abort t))))
```

キャッチャーとはクリーンアップフォーム (この例では when 以下) の直前に暗黙に置かれたものだと考える。名前が与えられないためにこのキャッチャーがシャドゥされてしまうことは無いので、unwind-protect を抜けるときにクリーンアップフォームが実行されることが保証できる。

$Id: scope-and-extent.tex,v 1.4 2008/09/25 02:18:02 kuroda Exp $
$Log: scope-and-extent.tex,v $
Revision 1.4 2008/09/25 02:18:02 kuroda
公開第 1 版