ゼミ: SICP

久野　靖*

2010.9.10

Ableson, Sussmann, Sussmann, 和田英一訳, 計算機プログラムの構造と解釈 第2版 (Structure and Interpretation of Computer Programs, 2nd ed.), ピアソン, 2000.

0　序文

□ 「教育者、将軍、栄養士、心理学者、親はプログラムする。軍隊、学生、一部の社会はプログラムされる。」

- 知的活動としてのプログラム→計算機プログラミング (本書の主題)。
- プログラムそのもの、プログラムの記述方法について扱う (本書の範囲)。

□ 本書の主題に関わる 3 つの現象:　「ひとの心」「プログラムの集積」「計算機」

- プログラムは「心に生まれた物理的・心理的プロセスのモデル」
- 巨大、複雑、部分的理解。満足できるモデル化はほとんどない
- 「このために」(認知の広まり、一般化に伴い) 進化。
- 「陽気な気分の源泉」の由来:プログラムとして表現した機構の (心の中と計算機上での) 絶え間ない解明とそれによる認知の拡大
- 「技術が夢を解釈するなら、計算機はプログラムを装って夢を実行するであろう。」

□ 「計算機は苛酷な仕事師」

- プログラムは正確でなければならず、言わんと欲することを細部まで正確に言わなければならない
- (他の記号活動と同様に) 議論を通じてプログラムの正当性を確信する
- Lisp は意味論 (もう 1 つのモデル) を与えられている→「プログラムの機能が述語論理で記されていれば」論理の証明技法を使用できる
- しかし、巨大になると (プログラムは普通そう)、仕様自身の適切さ、一貫性、正当性が疑わしくなる→大きなプログラムで述語論理の証明はあまりない
- 別の案:　正当性の確信できる小さなプログラム群の貯蔵庫＋証明された価を組織化する技法 (本書で扱う)
- 「強力な組織化技法の発見と熟成が巨大で重要なプログラムを創造する能力を加速」
- 逆に「巨大プログラムを書くのは負担なので、関数の重みを減らす新しい技法の発明が必要」

□ (プログラムと違い) 計算機は物理法則に従う→高速実行には配線長の短さ、熱の除去等が必要 (ハード屋の問題)

- ハードウェアはわれわれがプログラムを考えるより下のレベルで動作
- Lisp を機械語に変換するプロセスもプログラム可能な抽象モデル→ その研究/創作は任意のモデルをプログラム化する組織化プログラムへの見通しを与える (計算機自身もモデル化できる)
- 例:　最小の物理的スイッチ要素を微分方程式で記述した量子力学でモデル化→その細部の振舞いはプログラムによる数値近似→その計算の実行は…?

□ 3 焦点 (「ひとの心」「プログラムの集積」「計算機」) を分けて考えるのは単なる戦術的便宜の問題ではない

- 論理的分離は焦点間での記号のやりとりの加速をひき起こす
- (プログラム集積の) 豊富さ、(心の) 活力、(計算機の) 能力→人間の経験を超え、相互の関係は準安定
- ハードの増強→より大きなプログラム、新しい組織化原理、豊かな抽象モデルをもたらす
- 読者は「何のため?」と問うべきだが、「ほろ苦い哲学による便秘のためにプログラムの楽しみを放棄するといけないから」時々ね。

□ (さほど多くないが) 整列、探索、素数判定など「厳密な数学関数を実行する」もの→そういうプログラムをアルゴリズムと言う

- アルゴリズムの最適な振舞い (とくに実行時間と記憶容量) に多くの知見

*筑波大学ビジネス科学研究科

- プログラマは優れたアルゴリズムと成句を獲得すべし
- 「厳密な記述に抵抗する」プログラムもあるが→その性能を見積り、常に改善に務めるのは「プログラマの責任である」

□ Lisp は 25 年間使われた長命の言語。先輩は Fortran だけ

- Fotran は科学と工学の計算のため、Lisp は人工知能のため
- これらは重要な領域→今後 25 年以上は活発に使われるだろうから、この 2 つの言語に専念 (?)

□ Lisp は変容する → 本書で使う Scheme の場合: 変数束縛が静的スコープ、関数が関数を返せる、など

- 「構文は Algol 60 にも似ている。Algol 60 は現役の言語としては 2 度と使われないだろうが、Scheme と Pascal の遺伝子の中に住んでいる」
- 両者は対照的: Pascal はピラミッド的静的構造、Lisp は組織が作り出す動的構造
- 両者の組織化の原則は同じだが、Lisp プログラマが自由裁量で輸出できる機能はずっと多い
- Lisp プログラムは「その効用がそれを生み出した応用を超越するような関数でライブラリを膨張」
- リストデータ構造がそのような効用の成長に大きく貢献 (極めて汎用的であるため)
- Pascal の過度の宣言性は関数の特殊化を引き起こし、偶発的協力を妨げる
- 「1 つのデータ構造に働く 100 の関数の方が、10 のデータ構造に働く 10 の関数よりありがたい」
- 「ピラミッドは千年もの間立っていなければならず、組織は進化するか消滅しなければならない」

□ Pascal による入門教科書と違いは「Lisp をプログラムする真面目な本」ということ。「MIT だからできる」わけではない

- 人工知能のために Lisp を使う準備の本ではない。あくまで Lisp プログラミングの教科書。システムが大きくなるとソフトウェア工学と人工知能のプログラミングにおける関心事は同じになってくる。だから Lisp は人工知能以外でも使われるようになってきている
- 人工知能の研究は重要なプログラミングの問題を多く生み出した。
- 他の文化では問題が現れるとそれに対する言語が作られたが、人間との対話が非常に多いシステムになると言語は重要でなくなる傾向がある。このため、そういうシステムはさまざまな言語機能を持った複雑なものになりがち
- Lisp は単純な構文と意味を持ち、構文解析が不要なので、言語処理系を構築するのが簡単
- この自由さを享受し、さまざまなものを発明すべし。Lisp ばんざい (笑)

0.1 第 2 版への前文

□ 本書の内容は 1980 年から MIT の計算機科学入門科目のもとになっていた。

- この版ではさまざまな改良をおこなった --- 汎用演算システム、解釈系、レジスタマシンシミュレータ+翻訳系など。また IEEE 1990 Scheme 準拠
- 新しい主題: 状態オブジェクト、並列、関数型、遅延評価、非決定性
- この版は 1 学期で終わらせるのは無理なので取捨選択が必要。教師によっては 3、4 章までで終わらせたいらしい

0.2 第 1 版への前文

□ MIT の入門科目で、毎年 600〜700 名に教えられている

- 関心事 1: 計算機言語は演算を実行させる方法だけでなく、方法に関する考えを表現する新しい形式的媒体である → プログラムは人が読むように書くべき
- 関心事 2: 教える題材は、特定の言語の構文でも、特定のアルゴリズムでも、アルゴリズムや計算の基礎でもなく、「巨大なソフトウェアシステムの知的複雑性の制御に使う技法」

□ これを学んだ学生は、プログラミングの各種の流儀と美学に健全な感覚を持つことを期待

- 巨大システムの複雑さを制御する主要な技法の力を持つべき
- 模範的な書き方なら 50 ページのプログラムが読めるべき
- 何を読まず、何を理解しないでよいかを知るべき
- 元々の作者の精神と文体を維持したまま、プログラムの修正に確信を持つべき

□ ここで扱う技法はプログラミングに限られたものでなく、工学設計のすべてに共通

- 複雑さを制御するために、細部を隠す抽象を構成し、標準的で分かりやすい部品を交換可能にし、公認インタフェースを用意し、ある面を強調し他の面を軽視するような設計記述言語を用意する

□ 「計算機科学」は科学ではなく、その重要さに計算機はあまり関わらないという確信

- 計算機革命は「われわれが物を考える方法、考えを表現する方法での革命」
- 変化の本質は「手続き的認識論」(宣言的観点より命令的観点からの知識の構造の研究) の出現
- 数学は「何である」の概念、計算機科学は「いかにして」の概念を精密に扱う枠組みを提供

□ 手段として Lisp の一方言を扱う

- 言語を形式的に扱う必要はないので形式的に教えることもないが、学生は使うだけで数日で慣れる (Lisp の利点)
- 形式的性質はチェスの規則のように、1 時間で話せる
- 少しすれば、構文はもう忘れて実質 (何が計算したい、問題をどう分解し、部品をどう扱うか) に注力
- Lisp の他の利点として部品分解の戦略でさまざまなものが使用可能 (ただしそれらを強制はしない)
- データの抽象化、高階関数、代入による局所状態、ストリームと遅延評価、組み込み言語、漸進的設計/構築/テスト/虫とりのある対話環境など (McCarthy をはじめとする Lisp の人たちに感謝)

□ Scheme は Lisp と Algol の能力と気品を統合しようとする試み

- Lisp から→単純な構文の力、プログラムをデータとして表現、ごみ集め
- Algol から→静的スコープ、ブロック構造
- そして、Church の λ 算法と言語の関係に着目した開拓者にも感謝

0.3 謝辞

(略)

1 手続きによる抽象の構築

□ John Locke: 単純な考えを超えて力を引き出す 3 つの働き:

- いくつかの単純な考えを合成して 1 つにする
- 2 つの考えをもちより、同時に見えるようにもう 1 つのもので繋ぐ
- 実在において付随している他の考えから分離する (抽象)

□ 「計算プロセス」をとりあげる

- 計算機の中にある抽象的な存在
- 進行しながらもう 1 つの抽象的な存在であるデータを操作
- プロセスの進行は規則のパターン、プログラムの指示に従う
- 「プロセスに指示しようとして、プログラムを作る」→ 呪文で魔法をかける

□ 計算プロセスと魔法の類似性

- 見ることも触ることもできず、物質でできてもいない
- にもかかわらず、実在し、知的作業を遂行し、実世界に影響
- 「プログラムはプログラム言語の記号式で注意深く構成」

□ 計算プロセスは「正しく動く計算機の中では」「プログラムを精細かつ正確に実行」

- 初心者は魔術の結果を理解し予測する技法を学ぶ必要
- 安全に閉じ込められているので魔法の学習よりはるかに危険度は低い
- しかし実世界を相手とするプログラムには用心、熟練、知識が必要

□ ソフトウェア技術の名人といわれる人たち → プロセスが期待通りの仕事を実行することが「相当確かであるように」プログラムを組み上げる能力を持つ

- システムの振舞いを前もって見ることができる
- 予想外の破滅的な結果にならないようにプログラムを構築する術
- 事態が生じたら虫とりができる
- 巧みに設計された計算システムは、巧みに設計された自動車や原子炉と同様「モジュラに作られていて、部品ごとに作り、交換し、個々に虫とり可能」

2 Lisp によるプログラム

□ プロセスを記述する言語として Lisp を使用

- 1950 年代に、再帰方程式という論理表現についての推論の形式化として、McCarthy が考案

- そのようなものの初めての数学的形式化にもかかわらず、Lisp は実用。名前は LISt Processing から来ているが、記号処理を前提に設計
- そのためアトム、リストなどのデータオブジェクト→著しい特徴

□ Lisp は共同開発の成果ではなく「利用者の要求と実装上の考察に応じて試作的、非公式的に進化」したもの

- Lisp 社会は公式定義という動きに伝統的に抵抗
- 初期のコンセプトが持つ柔軟さ/優雅さ+進化→最新のアイデアの取り込みが常に可能
- 現在では Lisp はさまざまな方言から成る一族。ここでは Scheme を使用

□ 試作的性格、記号処理向け → 数値計算は遅かったが、今ではコンパイラもあり、かなり速くなっている

- 速度メインでない応用には多く使われている。操作系のシェル言語、エディタや CAD の拡張言語など

□ なぜ Lisp か? → 主要なプログラムの構成やデータ構造を学び、言語の基礎となる言語学的機能に関係づけるのに優れた媒体となる特徴を持つ

- とりわけ、手続きの定義そのものがデータとして扱えること --- 能動的な手続き、受動的なデータという境界をぼやかすことに依存した手法が使える
- これにより、解釈系や翻訳系などを書くのにも適している
- その上、Lisp でプログラムを書くのは大いに楽しい

3 プログラムの要素

□ 強力なプログラム言語は計算機に仕事のやり方を指示する手段以上のもの。プロセスに関する考えをまとめる枠組として役立つ

- 単純な概念を統合して複雑な概念を構成する手段に注目。強力な言語では次の 3 つがある:
- 基本式 --- 言語が関わる最も単純なもの
- 組み合わせ法 --- より単純なものから合成物を作る
- 抽象化法 --- 合成物に名をつけ、単一のものとして扱う

□ プログラムするとき、手続きとデータという 2 つのものを扱う (実はそれほど違わない)

- 早くいえば (?)、データは処理したい「もの」、手続きは処理法の記述
- 強力な (?) プログラム言語は、基本的データと基本的手続きが記述でき、手続きとデータを組み合わせたり抽象化する手段を持つべき
- 以下本章では手続きの構築規則に着目するため、単純な数値データのみを扱う (が、同じ規則が合成データの手続きにも使える)

3.1 式

□ Scheme の解釈系と対話してみるとよい。式を入力するとそれを評価した結果を表示

```
> 486
486
```

- 基本手続き (+、*など) を表現する式と数を組み合わせた合成式で数に対する基本手続きの作用を表現

```
> (+ 137 349)
486
```

- 式のならびをかっこで囲んで手続きの作用を表現する式: 組み合わせ (combination)。左端の要素: 演算子、他の要素: 被演算子。組み合わせの値は「演算子が指定する手続きを、被演算子の値である引数に作用させて得たもの」

□ 演算子を前に置く記法→前置記法。利点は任意個の引数が可能なこと

```
> (+ 21 35 127 7)
75
```

- 第 2 の利点は組合せの入れ子ができること

```
> (+ (* 3 5) (- 10 6))
19
```

- 入れ子の深さや式の全体としての複雑さに制限はない

```
(+ (* 3 (+ (* 2 4) (+ 3 5))) (+ (- 10 7) 6))
```

- しかし、人間は混乱しやすいので、清書系を使う

```
(+ (* 3
      (+ (* 2 4)
         (+ 3 5)))
   (+ (- 10 7) 6))
```

- 解釈系の基本動作は式にかかわらず同一 (read-eval-print loop)

3.2 名前と環境

□ プログラム言語の重要な点: 名前を使って計算オブジェクトを指す手段を用意すること

- 「オブジェクトを値 (value) とする変数 (variable) を識別するものが名前」
- Scheme では define で名前をつける

```
> (define size 2)
> (* 5 size)
10
```

- define は合成演算の結果を表すこともできるので、最も単純な抽象化手段
- 毎回複雑なオブジェクトを打ち込んでいたら不便
- 複雑なプログラムは徐々に複雑さが増すオブジェクトを積み上げていく

□ 値と記号を対応づけ、後で取り出せるためには、「記憶」の保持が必要 → これを「環境」(正確にはこの場合は大域環境) と呼ぶ

3.3 組合せの評価

□ 組合せを評価するとき、解釈系自身も手続きに従っていると考える→ その手順:

- 1. 部分式を評価
- 2. 最左部分式の値である手続き (演算子) を、残りの部分式の値である引数 (被演算子) に作用させる
- 分かる重要なこと: 各要素の評価プロセスが必要 → 本質的に再帰的
- 深い入れ子を考えると、再帰は複雑なものを簡潔に表現
- 式を図 1.1 のような木構造で表すと、上に向かって値を計算 →「木構造のため込み」(tree accumulation)

□ 評価する必要のないものもある → 規則が必要:

- 数字の列は、その表す数値とする
- 基本演算子の値は、対応する演算を実行する機械命令の列
- それ以外の名前の値は、その環境で名前と対応づけられたオブジェクト

□ 「*や+のような記号は大域環境に含まれていて、その値は機械命令の列」と考えれば 2 番目の規則は 3 番目に含まれることになるかも

- 記号の意味を確定させるのには環境に言及せざるを得ない --- Lisp のような対話型言語の場合
- 「環境は評価がおこなわれる文脈を提供」(→ 3 章)

□ define は例外的。define を 2 つの引数に作用させるのではない。(define x 2) は x を 2 に対応づける (組合せではない)

- このような一般評価規則の例外→「特殊形式」
- 他の特殊形式もまもなく出て来る
- 構文は同じ。式の評価規則は単純な一般規則+少数の特殊形式の規則から成る

3.4 合成手続き

□ 強力なプログラム言語に備わっているべき要素がLisp にもある:

- 数と算術演算子は基本的データと手続き
- 組合せの入れ子は演算を組み合わせる手段
- 名前と値を対応づける定義は抽象の「そこそこの」手段

□ より強力な抽象化技法: 手続き定義

- 例: 2 乗の計算

```
(define (square x) (* x x))
```

- 一般形

```
(define (名前 仮パラメタ…) 本体)
```

- 使用例

```
> (square 21)
441
```

- さらにこれを利用して x*x+y*y など

```
(define (sum-of-squares x y)
  (+ (square x) (square y)))
(define (f a)
  (sum-of-squares (+ a 1) (* a 2)))
```

3.5 手続き作用の置換えモデル

□ 演算子が合成手続き名であっても評価の進め方はこれまでと同じ

- つまり、各要素を評価してから手続きを引数に作用させる
- 手順:「合成手続きを引数に作用させるには、各仮パラメタを対応する引数で取り替え、手続きの本体を評価する」

```
  (f 5)
→ (sum-of-squares (+ a 1) (* a 2))
→ (sum-of-squares (+ 5 1) (* 5 2))
→ (+ (square 6) (square 10))
→ (+ (* 6 6) (* 10 10))
→ (+ 36 100)
→ 136
```

□ これを「手続き作用の置換えモデル」という。注意点:

- 置換えの目的は手続き作用を考える補助であり、処理系の働きを述べるものではない (解釈系は仮パラメタに値を置換えて手続きの字面を操作しつつ作用を評価するのではない) → 3 章、4 章で扱う
- 本書では次々にさまざまな解釈系のモデルを示す (理工学では単純で不完全なモデルからはじめ、詳しく調べるうちにその不完全さが分かって洗練が必要となる)

3.6 作用的順序と正規順序

□ 「演算子と被演算子を評価し、次に結果の手続きを結果の引数に作用させる」以外の可能性

- 例: 値が必要になるまで被演算子を評価しない

```
  (f 5)
→ (sum-of-squares (+ 5 1) (* 5 2))
→ (+ (square (+ 5 1)) (square (* 5 2)))
→ (+ (* (+ 5 1) (+ 5 1)) (* (* 5 2) (* 5 2)))
→ (+ (* 6 6) (* 10 10))
→ (+ 36 100)
→ 136
```

- この場合、(+ 5 1) と (* 5 2) の評価は 2 度ずつ行われている
- この評価順序 (「完全に展開し、その後簡約」) → 正規順序の評価 (normal-order evaluation)
- 解釈系の評価順序 (「引数を評価してから簡約」) →作用的順序の評価 (applicative-order evaluation)」
- 置換えを使ってモデル化でき正しい値が得られる手続き作用については、両者は同じ結果をもたらす

□ Lisp が作用的順序順序を採用しているわけ:

- 式の多重評価を避けることによる効率
- 置換えでモデル化できる範囲外では正規順序の評価が複雑になる
- しかし正規順序の評価も有効な道具→3 章と 4 章で扱う

3.7 条件式と述語

□ ここまでで定義できる手続きの表現力は小さい←テストの結果で演算を切り替える方法がないため

- 例:

```
       | x  (x>0)
|x| = | 0  (x=0)
       | -x (x<0)
```

- Lisp の場合、cond 特殊形式を使う

```
(define (abs x)
  (cond ((> x 0) x)
        ((= x 0) 0)
        ((< x 0) (- x))))
```

□ cond の一般の形は:

```
(cond (述語 1 本体 1)
      (述語 2 本体 2)
      ...
      (述語 N 本体 N))
```

- 評価方法: 述語を先頭から順に評価し、最初に結果が真であるものに対応する本体を評価して返す
- すべての述語が真でないなら cond の結果は「定義されない」
- 述語とは真または偽を結果として取る式や手続き
- 別案: (else は cond の最後の節で述語の代わりに書けるもの)

```
(define (abs x)
  (cond ((< x 0) (- x))
        (else x)))
```

- 別案: (特殊形式 if は 2 つの式どちらかだけを選ぶ)

```
(define (abs x)
  (if (< x 0) (- x) x))
```

□ 複合条件のための論理合成演算

- (and 式 1 式 2 … 式 N) --- 左から順に式を評価し、結果が偽ならそこでやめて偽を返す。最後まで偽でなければ最後の式の結果を返す
- (or 式 1 式 2 … 式 N) ---左から順に式を評価し、結果が偽でないならそこでやめてその結果を返す。最後まで偽なら偽を返す

```
(define (>= x y)
  (or (> x y) (= x y)))
```

- (not 式) --- 式の真偽を反転

```
(define (>= x y)
  (not (< x y)))
```

3.8 問題 1.1〜1.5

3.9 例: Newton 法による平方根

□ これまでに出て来た手続きは数学の関数に類似 (パラメタ→値を規定)

- しかし手続きと数学の関数には重要な違い: 手続きは実効的
- 例: 平方根の定義

  √ x = y ≧ 0 かつ y**2 = x であるような x
- これは数学の関数を定義しているが手続きではない (どうやって具体的に x を求めたらいいか分からない)。以下も同様

```
(define (sqrt x)
  (the y (and (>= y 0)
              (= (square y) x))))
```

□ 両者の違い: 「もののあり様の記述」と「ことのなし方の記述」

- あるいは「平序文」と「命令文」。数学は前者、計算機科学は後者

□ Newton 方による平方根→予測値 y に基づき y と x/y の平均を取ることでよりよい予測値を得る

- 開きたい数と予測値からはじめて、予測値が十分正確なら終わり、そうでなければ予測値を改良

```
(define (sqrt-iter guss x)
  (if (good-enough? guess x)
      guess
      (sqrt-iter (improve guess x)
                 x)))
(define (improve guess x)
  (average guess (/ x guess)))
(define (average x y)
  (/ (+ x y) 2))
```

- 「十分よい」は自乗と開きたい数の差が許容値 (ここでは 0.001) 未満ということにする

```
(define (good-enough? guess x)
  (< (abs (- (square guess) x)) 0.001))
```

- どんな数の平方根も 1 として予測開始

```
(define (sqrt x)
  (sqrt-iter 1.0 x))
```

□ これで普通に計算できる

- このような単純な手続き言語でも C や Pascal と同様に数値計算が書ける
- 手続き呼び出しがあれば特別な構文がなくても反復ができる

3.10 問題 1.6〜1.8

3.11 ブラックボックス抽象としての手続き

□ sqrt → 「相互に定義する手続きの組みとして定義したプロセス」

- sqrt-iter → 「再帰的」(定義に自分自身を使ってある) → このような循環がうまく行くのかどうか? → この点は 1.2 節で扱う
- 別の論点: 全体問題を予測値が十分良好とは? 改良はどうするか? などの部分問題に自然に分割 → 手続きの束

□ 分割「戦略」は単に部分に分けることよりも大切

- 「最初の 10 行」「次の 10 行」みたいに分けてもしかたがない
- 「部品としてまとまった仕事ができるように」分けるのが大切
- good-enough? の定義に square を使うとき、square はブラックボックスとみることができる
- ブラックボックスの中でどうやって計算するかには関心がない
- 関心があるのは「2 乗を計算する」ということだけ
- 計算する方法は隠しておいて後で考える
- good-enough? に関する限り、square は「手続きではなく手続き抽象 (procedural abstraction)」
- このレベルでは 2 乗を計算する手続きならどれでも同じくらい有用
- 次の 2 つは返す値だけ考えれば区別できない

```
(define (square x) (* x x))

(define (square x)
  (exp (double (log x))))
(define (double x) (+ x x))
```

3.12 局所名

□ 手続き利用者が気にしないでよい手続き実装の細部の 1 つ → 仮パラメタの名前のつけ方

```
(define (square x) (* x x))
(define (square y) (* y y))
```

- 「手続きの意味はパラメタ名と無関係であるべき」という原則は自明に見えるが深い影響
- たとえば、パラメタ名は手続き本体に対して局所的でなければならない

```
(define (good-enough? guess x)
  (< (abs (- (square guess) x)) 0.001))
```

- square の x は good-enough の x とは別でなければならない。square が終わった後 good-enough が x を使うかも知れないから、square が good-enough の x に影響を及ぼしてはいけない
- パラメタが各手続きに局所的でないと、両方の x が混同され、good-enough の結果が square のどの版を使ったかによって変わることになる

□ 仮パラメタには「手続き定義中で仮パラメタがどんな名前を持っていても構わない」という特別な役割 → 束縛変数

- 「手続き定義は仮パラメタを束縛する」
- 束縛変数を統一的につけ替えても手続き定義の意味は変わらない
- 束縛されていない変数 → 自由変数
- 名前が束縛されている式の範囲 → 有効範囲 (scope)
- 手続き定義中では、仮パラメタとして宣言された束縛変数の有効範囲は、その手続きの本体

□ 上の good-enough?の定義では、guess と x は束縛変数、<、-、abs、square は自由

- guess と x は「<、-、abs、square と異なる限り」どんな名前を選んでも結果は変わらない
- guess を abs に変更すると、abs を「補足 (capture) した」ことに → abs が自由変数から束縛変数に

- 一方、good-enough? の意味は自由変数の意味と関係している (abs は絶対値を計算するもの。これを cos に取り替えたら good-enough? は別の関数に

3.13 内部定数とブロック構造

□ 前項で「名前を隔離する方法」の 1 つを学んだ (手続きの仮パラメタ) が、別の方法もある

□ 今のプログラムはバラバラにできていた

```
(define (sqrt x)
  (sqrt-iter 1.0 x))
(define (sqrt-iter guss x)
  (if (good-enough? guess x)
      guess
      (sqrt-iter (improve guess x) x)))
(define (good-enough? guess x)
  (< (abs (- (square guess) x)) 0.001))
(define (improve guess x)
  (average guess (/ x guess)))
```

□ 問題: この中で利用者にとって重要なのは sqrt で、それ以外は「利用者の心に虚しく響くだけ」

- sqrt と一緒に自分のプログラムを書くときに good-enough?という名前の別の手続きを定義できない
- 大きなプログラムを分担作成するときに問題になる
- たとえば大きな数値計算ライブラリであちこちで反復解法を使うとそこでも improve や good-enough? という補助手続きを持たせたいかも

□ 解決策→補助手続きを局所化

```
(define (sqrt x)
  (define (good-enough? guess x)
    (< (abs (- (square guess) x)) 0.001))
  (define (improve guess x)
    (average guess (/ x guess)))
  (define (sqrt-iter guss x)
    (if (good-enough? guess x)
        guess
        (sqrt-iter (improve guess x) x)))
  (sqrt-iter 1.0 x))
```

- 定義の入れ子→ブロック構造 (名前保護の基本的機構)
- さらに内部化に際して単純化が可能
- x は sqrt で束縛されているから、sqrt の中の手続きは x の有効範囲内

□ だから sqrt 中の手続きは x を自由変数としておけば sqrt の x を参照 (静的有効範囲、lexical scoping)

```
(define (sqrt x)
  (define (good-enough? guess)
    (< (abs (- (square guess) x)) 0.001))
  (define (improve guess)
    (average guess (/ x guess)))
  (define (sqrt-iter guess)
    (if (good-enough? guess)
        guess
        (sqrt-iter (improve guess))))
  (sqrt-iter 1.0))
```

- 出来るだけブロック構造を使って大きな問題を部品に分割していく
- 「ブロック構造は Algol60 で始まり、先進的プログラム言語の多くに見られ、巨大プログラムの構築作業を援助する重要な道具になっている」