情報システム科学実習II (2)

変数と関数による抽象化

1999年10月20日 増原 英彦

関数的プログラミングに限らず、プログラミングにおける基本的な要素は、

- 原始的な式 (primitive expression)
- 合成 (composition) のための手段
- 抽象化 (abstraction) の手段

の3つである。

1 原始的な式

数のように分割できない式を原始的な(primitive) 式という。原始的な式には以下のようなものがある。

- `345`, `3.14159265359` のような数
- `#t`(真), `#f`(偽) のような真偽値
- `#\a`, `#\Newline` のような文字
- `"primitive expression"`のような文字列

2 式の組み合わせ

演算子 (operator) を使って式を組み合わせることができる。Scheme では、式を組み合わせる場合に必ず括弧を使い、頭置記法(prefixed notation) を用いる。つまり、次のように全体を括弧で囲った場合に、先頭の ⟨式0⟩ が演算子で、続く ⟨式1⟩, ⟨式2⟩, ... が引数(operand) になる。

(⟨式0⟩ ⟨式1⟩ ⟨式2⟩...)

練習 2-1: (式)

以下の計算をするSchemeの式を書け。(組み込み演算子としてどのようなものが使えるかはR4RSを参照せよ。)

1. 13 を 3 で割った余り
2. 文字「a」の大文字
3. 文字列「not hungry」の5 ~ 10文字目から成る文字列
4. 文字列「internationalization」の長さ
5. $(12 \times 60 + 34) \times 60 + 56$
6. $2 \sin \frac{\pi}{4}$ (とりあえずπは3.14で代用する)
7. $\sqrt{3^2 + 4^2}$
8. 13が3で割り切れれば真、そうでない場合は偽となるような式
9. $\cos 1$が$1/2$より大きければ真、そうでなければ偽となるような式
10. 文字「a」に対応する文字コードに3を足した値に対応する文字
11. 文字列 `"abcd"` と `"efgh"` を結合した文字列の3 ~ 6文字目から成る文字列

∎

3 変数: 値に名前をつける

変数を使うことで計算された値に名前をつけることができる。変数を用いる効果としては、次のようなものがある。

- 意図が不明な値に名前を付けることができるため、プログラムの意味がより分かりやすくなる。(具体的な値を使っている場合よりも、より抽象化されたプログラムにすることができる)
- 同じ計算式が何度も使われる場合に、変数によって置き換えることで式が簡単になる
- ある値を変更する際に、変数の定義だけを変更すればよくなる

Schemeでは、

`(define ⟨名前⟩ ⟨式⟩)`

と書くことで、⟨名前⟩の変数に⟨式⟩の値を結びつけることができる。例えば、

```
(define size 2)
```

と書くと、`size`という名前の変数と 2 という値が結びつけられる。以降、2と書くかわりに`size`と書いても全く同じ値が得られる。

例えば、「半径2の円の円周」・「半径2の円の面積」を求める式を

```
(* 3.14 2 2)          (円周)
(* 3.14 (* 2 2))      (面積)
```

と書いたのに対して

```
(define pi 3.1415)
(define radius 2)
(* pi 2 radius)                 (円周)
(* pi (* radius radius))        (面積)
```

と書く場合では、意味の分かりやすさや変更のしやすさが大きく違う。

3.1 一時変数

同じ式が何度も登場する場合、一時変数 (temporary variable) を使うと式が簡単になる。Scheme では、以下のような形式によって一時変数が利用できる:

```
(let ((⟨変数1⟩ ⟨式1⟩)
      (⟨変数2⟩ ⟨式2⟩)
           ⋮
      (⟨変数n⟩ ⟨式n⟩))
  ⟨式⟩)
```

例えば次の式は、変数`x`が`(sin (/ pi 3))`の値、変数`y`が`(cos (/ pi 3))`の値であるとして、式`(sqrt (+ (* x x) (* y y)))`を評価する。

```
(let ((x (sin (/ pi 3)))
      (y (cos (/ pi 3))))
  (sqrt (+ (* x x) (* y y))))
```

このとき、変数`x`や`y`は式`(sqrt (+ (* x x) (* y y)))`の中のみで有効であることに注意。(詳しい仕組みについては次回以降に説明する。)

4 関数: 計算に名前をつける

計算の手順にも名前を付けることができ、これを関数 (function) と呼ぶ。関数は、

`(define (⟨名前⟩ ⟨仮引数⟩ ⟨仮引数⟩...) ⟨式⟩)`

のように定義して、

`(⟨名前⟩ ⟨引数⟩ ⟨引数⟩...)`

のようにして使う。このように定義された関数も、`*`のような原始的な演算子と同様に使える。

例えば、`(+ (* 3.82 3.82) (* size size) (* 10 10))`という式には、「何かを二乗する」計算が頻出しているので、「何かを二乗する」関数を定義してみる。

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

すると、先の式は`(+ (square 3.82) (square size) (square 10))`と書くことができる。

関数を定義すれば、その関数の名前と「何を計算するか」さえ覚えておけば、複雑な定義の詳細を忘れてしまうことができる。これは、計算手順の抽象化の第一歩である。

練習 2-2: (関数)
以下の計算をする関数を定義せよ。

1. 座標 (x, y) 上の点の原点からの距離`(distance x y)`
2. x が y で割り切れる場合真、そうでなければ偽を返す関数 `(divisible? x y)`
3. 文字「a」の文字コードに与えられた数 (offset) を加えた値に対応する文字`(alphabet-at offset)`
   例えば`(alphabet-at offset)`は「`#\c`」になる。
4. 与えられた文字列の (1) 先頭 1 文字、(2)(文字列全体の長さ) − 2(の文字列化したもの)、(3) 最後の 1 文字をつなげたものを返す`(abbriviate word)`。例えば、「internationalization」であれば「i18n」を返す。

■

4.1 計算の順序

複雑な式を評価するとき、実際にはどのようにして計算が行われているのだろうか?

例えば「`(+ (* 3 3) (* 4 4))` のような式からどうやって 25 が求められたり、`(square 8)` という式からどうやって 64 が求められているのだろうか。

原始的な演算子の場合: 式を評価する手順は、

- まず副式を全て評価する
- 演算子に副式を評価した値を適用 (apply) する。

  「適用する」とは、+だったら実際に加算を行うことであり、計算機の内部では「加算をする機械命令」が実行される。

のように定義できる。この定義は (後で述べるように) 再帰的なものである。

合成関数がある場合: 演算子部分が合成関数であるような式の場合、式全体を関数の定義部分で置き換えて、さらに評価を進める。その際に、仮引数の変数は実引数によって置き換える。

例えば、`(define (dobule x) (+ x x))` という定義があったときに、`(double 3)` という式を評価すると、「`(double 3)` を、`(+ x x)` で置き換える。その際、`x` は 3 で置き換える」ので、`(+ 3 3)` というものに置き換えて評価が続けられる。

こみいった式の場合は、どの式から置き換えをしてゆくかという問題が生じる。

正規順序 (normal order): まず、外側の式を関数定義で置き換える。次に、副式を順に置き換えてゆく。置き換えるものがなくなったら、原始的な演算子の式になっているので、それを評価する。

適用的順序 (applicative order): まず、副式を評価して、評価した値に置き換えておく。次に、全体の式を関数定義で置き換える。得られた式をさらに評価する。

例えば、`(square (double 3))` を評価する場合、正規順序では、

```
(* (double 3) (double 3))
(* (+ 3 3) (double 3))
(* (+ 3 3) (+ 3 3))
(* 6 6)
36
```

のように、また適用的順序では、

```
(square (+ 3 3))
(square 6)
(* 6 6)
36
```

のように計算が進む。

5　条件判断

5.1　条件分岐

条件によって計算の内容を変えることによって、より複雑な計算をさせることができる。`Cond` (conditional) 形式は、

```
  (cond (⟨条件式1⟩ ⟨式1⟩)
        (⟨条件式2⟩ ⟨式2⟩)
                 ⋮
        (⟨条件式n⟩ ⟨式n⟩))
```

のように書く。⟨条件式 i⟩ が成立した場合に ⟨式 i⟩ が評価される。一番最後の条件式に `else` と書くと、他の条件全てが成立しなかった場合の計算を書ける。

例えば、絶対値を求める関数は数学的には

$$|x| = \begin{cases} x & (x > 0) \\ 0 & (x = 0) \\ -x & (\text{otherwise}) \end{cases} \tag{1}$$

であるが、これが次のように書ける:

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

また、条件式が一つだけの場合は、

```
  (if ⟨条件式⟩
    ⟨式1⟩
    ⟨式2⟩)
```

のように書くことができる。この場合、⟨条件式⟩ が成り立った場合には、⟨式 1⟩ が、そうでない場合には ⟨式 2⟩ が評価される。

5.2 条件の組み合わせ

複雑な条件判断を行うために、`and`, `or`, `not` といった形式が用意されている。特に`and`, `or` は特別な評価の順序を持っている:

- `(and` ⟨式 1⟩ ⟨式 2⟩ `...)` は、副式を左から順に評価してゆき、全てが真であった場合に真となる。偽になる副式が 1 つでもあった場合、全体も偽となる。その際、(左から順に評価しているので、残る副式は評価されない。)

- `(or` ⟨式 1⟩ ⟨式 2⟩ `...)` は、副式を左から順に評価してゆき、1 つでも真となる式があった場合に全体が真となる。その際、残る副式は評価されない。全てが偽となった場合は全体も偽となる。

練習 **2-3:** (条件分岐)

以下の計算をする関数を定義せよ。適宜前で定義された関数を使用してよい。

1. 与えられた時刻が午前 (0 ~ 11) だった場合`"am"`という文字列を、午後 (12 ~ 23) だった場合`"pm"`という文字列を返す`(meridiem hour)` (それ以外の値には`"error"`という文字列を返す。)
2. 2 つの数が充分に近い場合に真、そうでない場合に偽を返す関数`(close-enough? x y)` (充分に近いとは、定数 δ を使って $|\frac{x-y}{y}| < \delta$ である場合とする。)
3. 数 x, y を受け取り、x が y の平方根になっていれば真、そうでなければ偽を返す`(sqrt? x y)` (ただし、数値誤差を考慮して、x^2 と y が充分に近いことを判定することにする。)
4. 与えられた数が 2 または 3 の倍数であるときに真、そうでないときに偽となる関数`(multiple-of-2-or-3? x)`
5. 与えられた数が正のときはその数の平方根を、負のときは自乗を返すような関数`(sqrt-or-square x)` (演算子の位置に条件式を書いてみよ)

∎

6 不純な計算

これまで見てきた計算は、四則演算のように何度計算しても結果が変わらないようなものであった。このようなものは関数的 (functional) な計算という。

現実世界のプログラムでは、計算をする度に結果が変わるものや、計算をした結果、外界に影響を与えるものがある。このようなものを命令的 (imperative) な計算と言う。例:

- キーボードから 1 行分の文字列を入力をする: `(read-line)`

- 画面へ文字列を出力する: `(display "To be or not to be ...")`

- 画面を改行する: `(newline)`

- Unix コマンドの実行: `(exec "ls")` (STk 独特の関数)

- 乱数の発生: `(random 100)` (STk 独特の関数)

このような計算を扱うためには、順番に評価を行う手段が必要になる。そのために、

```
  (begin
   ⟨式1⟩
   ⟨式2⟩
     ⋮
   ⟨式n⟩)
```

という形式がある。この形式では、⟨式 i⟩ を順番に評価する。

関数定義の本体や、`let` 文の本体では、`begin` を省略できる。つまり、

```
(define (display-and-newline message)
  (begin
    (display message)
    (newline)))
```

と

```
(define (display-and-newline message)
  (display message)
  (newline))
```

は同じことである。

6.1 おまけ: STk による GUI の例

STk では、ボタンやメニューなどのグラフィカルユーザインターフェイスを簡単に構築するための命令的な演算子が多数用意されている。ここではその一部を紹介する。(Web ページにコードあり)

```
;; hello-button.scm
;; GUI 関係の演算子を利用する場合に必要となるライブラリを
;; ロードする
(require "Tk-classes")
;; 画面にメッセージを表示する関数を定義する。
(define (say-hello) (display "Hello.") (newline))
;; ボタンを作る。ボタンの字面は :text で、ボタンが押されたときの
;; 動作は :command で指定する。
(define hello-button
 (make <Button> :text "Good day!" :command say-hello))
;; ボタンを表示する。(作るだけでは表示されないことに注意)
(pack hello-button)
;; ボタンが見えたはずである。押してみよう。
;; 最後にボタンを消す。
(destroy hello-button)
```

表示できる部品にはラベル・入力欄・メッセージ領域など多様なものがある。(Web ページにコードあり) 詳しくは STk の解説のページ `~masuhara/lib/stk/lib/stk/4.0.1/Help/STk-hlp.html` から参照できる。

```
;; interactive-greetings.scm
(require "Tk-classes")          ; おまじない
;; ラベルを作る
(define label-for-entry
  (make <Label> :text "Tell me your name:"))
;; 名前の入力欄を作る
(define entry-for-name (make <Entry>))
;; メッセージを表示する領域を作る :width, :height は表示領域の
;; 大きさを指定する
(define message-area (make <Message> :width 300 :height 100))
;; 挨拶を書く関数
(define (greetings)
  ;; 入力欄に書かれた文字列を読む
  (let ((name (value entry-for-name)))
    ;; 表示領域の内容を変更する
    (slot-set! message-area ’text
               ;; 挨拶文
               (string-append "How are you, " name "?"))))
;; OK と書かれたボタンを作る :command はボタンが押されたときに
;; 呼び出される関数を指定する
(define OK-button
  (make <Button> :text "OK" :command greetings))
;; ラベル、入力欄、ボタン、表示領域を画面上に表示する
(pack label-for-entry entry-for-name OK-button message-area)
```

課題 2-1: (関数)

以下の計算をする関数を定義せよ。

1. 与えられた西歴年 (year) が閏年の場合に真、そうでないときは偽を返す関数
   `(leap-year? year)`
   (閏年は、4 で割り切れ、かつ、100 で割り切れないまたは400 で割り切れるような年である。)
2. 現在時刻を調べ、午前中だったら「Good morning.」午後だったら「Good afternoon.」と挨拶を表示する関数。(現在時刻を調べるには、Unix コマンド `date +%H` を使え)
3. STk の GUI 機能を使ってボタンを作り、それ押すと適当な関数を実行し、その結果を表示する関数

▮

A 課題の提出方法等

授業連絡用メイリングリスト: `paradigm@graco.c.u-tokyo.ac.jp` 授業に関する連絡・質問・議論等々はこのアドレスへ自由に送ってよい。

形式: レポートは web ページを作成し、作成したことをメールで報告することで提出する。

Web ページは`u-tokyo.ac.jp` ドメインから読むことができるものであれば、どこに作ってもよい。また、先頭のページは各課題へのリンクとすること。

体裁は自由だが、タイトル・日付・氏名・課題の番号・問題文等のレポートとしての最低限の様式を満たしていること。また、単にプログラムを書くだけでなく、そ

の説明や実行例などをのせること。

登録: レポート用の先頭ページの作成した上で、以下の情報を `paradigm-report@graco.c.u-tokyo.ac.jp` へ送ること。

1. 氏名
2. 学生証番号
3. レポートページの URL
4. メールアドレス

登録された URL はまとめて全員が閲覧できるようにする予定である。

提出: 各回のレポートが完成したら、その旨を `paradigm-report@graco.c.u-tokyo.ac.jp` へ送ること。

B　Scheme の基本操作

Scheme プログラミングを効率よくおこなうために、エディタから Scheme を使う方法、デバッガの使い方、処理系依存の点などについて説明する。なお、ここでは、Scheme 処理系として **STk** を、エディタとして **GNU-Emacs** (Nemacs, Mule)[1] を使うことを前提としている。

B.1　教育用計算機システム Unix 環境における Scheme

教育用計算機システムの Unix 環境では、`~masuhara/lib/stk` の下にインストールされた STk が利用できる。利用のためには、ファイル `~/.cshrc` に以下の行を加える:

```
setenv STK_LIBRARY ~masuhara/lib/stk/lib/stk/4.0.1/
set path=(~masuhara/lib/stk/bin $path)
```

確認するには、`kterm` 中から以下のように操作する:

```
masuhara@as301> source .cshrc⟨Return⟩
masuhara@as301> stk⟨Return⟩
Welcome to the STk interpreter version 4.0.1 [SunOS-5.6-sun4]
Copyright © 1993-1999 Erick Gallesio - I3S - CNRS / ESSI <eg@unice.fr>
STk> (exit)⟨Return⟩
masuhara@as301>
```

また、Emacs の中から STk を利用するために、ファイル `~/.emacs` に以下の行を加える。ファイルがない場合は新たに作る:

```
(setq scheme-program-name "stk")
```

確認のためには、まず一度 Emacs を終了し再度起動する。その後、Emacs の中で「*M*-x run-scheme」[2]とタイプすると、Scheme と対話するための “`*scheme*`” という名前のバッファが作られ、そこにプロンプト (“`STk>`”) が表示されているはずである。

---

[1] この資料中では、Emacs, Nemacs, Mule の差異は重要でないので、単に Emacs と呼ぶことにする。

[2] 見慣れない表記かも知れないが、「`meta` というキーを押しながら `x` というキーを押し、その後『`run-scheme`』とタイプし、最後に `return` キーを押す」、という意味である。

マニュアル: ディレクトリ`~masuhara/lib/stk/doc/stk-4.0.1/`の中にSTkのreferenceマニュアルやFAQがある。Schemeに関する情報は実習webページの「関連情報」以下を参照のこと。

ソースコード: ディレクトリ`~masuhara/src/STk-4.0.1/`の中にSTk処理系のソースコードがある。また、`~masuhara/src/STkwin32-4.0.1.zip`はMS-Windows用の処理系である。

B.2 Emacs の中で Scheme と対話する

Emacsの中で「*M*-`x run-scheme`」とタイプすると、Schemeと対話するための“`*scheme*`”という名前のバッファが作られる。ここのプロンプト(“`STk>`”)へ向かって簡単なプログラムを入力したり、デバッグをしたりする。

実際に対話している例を示す。人間が入力したのは、下線の引かれている部分。また、“`;;`”で始まる行は、コメントであり実際には現れていないものである。

```
Welcome to the STk interpreter version 4.0.1 [SunOS-5.6-sun4]
Copyright © 1993-1999 Erick Gallesio - I3S - CNRS / ESSI <eg@unice.fr>
;; 式の入力は最後に ⟨Return⟩ を押す
STk> (+ 1 2 3 4 5)⟨Return⟩
15
STk> (sqrt (+ (* 10 10) (* 5 5)))⟨Return⟩
11.1803398874989
;; 関数の定義
STk> (define (twice x) (+ x x))⟨Return⟩
twice
STk> (twice 123)⟨Return⟩
246
;; 二行以上に渡る式や定義は、各行の最後に ⟨C-j⟩ を押し、[a]
;; 一番最後で ⟨Return⟩ を押す
STk> (define (delete-last-character str)⟨C-j⟩
       (substring str 0 (- (string-length str) 1)))⟨Return⟩
delete-last-character
STk> (delete-last-character (exec "date +%H"))⟨Return⟩
"17"
STk>
```

---

[a] これは「`control`キーを押しながら`j`キーを押す」という意味。

入力が間違ったら…… Emacsの中からSchemeを使っている場合、⟨*Return*⟩を押すまでは入力の内容を変更することができる。もし、式の入力の途中で間違いに気付いた場合は、文書を編集しているのと同じ要領で訂正すればよい。間違いを訂正した後は、ポイント(カーソル)を式の最後尾へ移動してから、⟨*Return*⟩を押す。

同じ式をもう一度入力するには…… 以前に入力した式をそのままもう一度入力したい、あるいは、少し修正して入力し直したい場合には、*M*-p, *M*-nによって呼び出すことができる。*M*-pを押し続けていると、どんどん昔の入力にさかのぼってゆき、*M*-nはその逆である。

B.3　Scheme プログラムの編集

大きなプログラムは、普通ファイルで編集する。その際、編集している関数の定義をその場で Scheme へ送る方法がある。Scheme プログラムを編集しているバッファにおいて、利用できるコマンドには以下のようなものがある。

なお、Scheme プログラムのファイル名には `.scm` という名前が付いていないと、これらの機能は使えない。

キー	働き
C-c *C*-e	カーソルのある関数定義を Scheme へ送る
C-c *C*-r	リージョンをまるごと Scheme へ送る
C-x *C*-e	直前の S 式を Scheme へ送る
C-c *C*-l	ファイルをロードさせる
C-c *C*-z	Scheme との対話バッファへ移動

B.4　その他

ファイルのロード: `(load "ファイル名")` Unix のパス名がそのまま使える。

無限ループに入ったら: Scheme との対話バッファから *C*-c *C*-c とタイプすると強制的に停止させることができる。

Scheme の終了: `(exit)` を実行する。

B.5　Emacs のカーソル移動

Emacs の基本的なキー操作として、コントロールキーを使ったキーボードの操作がある。

C-b	一文字戻る
C-f	一文字進む
C-n	次の行へ
C-b	前の行へ

もちろん、キーボードには矢印キー「← ↓ ↑ →」が付いているのでそれを使ってもよいのだが、コントロールキーを使う利点はいくつかある。

- タッチタイプしやすい
- 単語・S 式単位の移動ができる
- どんな環境でも使える
- どんなキーボードでも同じ場所にある

メタキー(「◇」と書いてあるキー)組み合わせて使えば、単語やS 式単位の移動が簡単におこなえる。

M-`b`, *M*-`f`	前後の単語へ移動
M-`t`	単語の入れ替え
C-*M*-`b`, *C*-*M*-`f`	前後のS式への移動
C-*M*-`t`	S式の入れ替え
C-*M*-`Space`	S式をマーク
	等々

詳しくは、

M-`x describe-bindings` ⟨*Return*⟩

で出てくるヘルプを見るか、Emacsの解説書でも読んで欲しい。

B.6　ファイルを使ったプログラムの編集

少し大きいプログラムを書くようになってくると、ファイル上でプログラムを編集しつつ、Schemeと対話するようなプログラミングをしたくなる。ここでは、典型的なプログラミングのサイクルを説明する。

B.6.1　準備、Schemeを起動しておく

Emacsを起動したら、STkを起動する。

M-`x run-scheme` ⟨*Return*⟩

B.6.2　最初は、おもむろにファイルを開いてプログラムを書いてゆく

例として「`ex1.scm`」というファイルを作ることにする。(ファイル名の最後が「`.scm`」となっていることに意味があるので注意。)

C-`x` *C*-`f`
`File: ~/hello.scm`⟨*Return*⟩

これで「`hello.scm`」という名前のバッファが作られる。このとき、status lineに「`Scheme mode`」と表示されていることを確認。

このバッファは、単に編集するだけであり、⟨*Return*⟩を押しても式が実行されるわけではないことに注意。試しに以下のようなプログラムを書いてみよう。

```
(require "Tk-classes")            ; おまじない

;; 文字列から最後の1文字を取り除いた文字列を作る
(define (delete-last-character str)
  (substring str 0 (- (string-length str) 1)))

;; ユーザのログイン名を得る
(define (get-user-name) (delete-last-character (exec "whoami")))

;; 挨拶文を書いたラベルを作る
(define (greetings name)
  (make <Label> :text (string-append "Hello " name)))

;; ユーザへの挨拶を表示する
(define (hello) (pack (greetings (get-user-name))))
```

B.6.3　Scheme にファイルをロードさせる

このように沢山の定義を一度に書いてしまうと、一つ一つ定義を Scheme へ送るのは面倒になる。そこで、ファイル中の定義まとめて Scheme に送る (通例 load と呼ばれる)。

```
C-c C-l
Load Scheme file: (default hello.scm): ~/⟨Return⟩                  *1
Save hello.scm (y/n)? y                                             *2
```

のようにする。*1 のところで別のファイル名を入力すれば、そのファイルがロードされる。また、ファイルがすでにセーブ(*C*-x *C*-s) されていた場合は、*2 の質問は出て来ない。

Stk の場合、ロードが完了しても特別なメッセージは表示されず、プロンプトが一個新たに表示されるだけである。

B.6.4　Scheme 対話バッファへの移動

C-c *C*-z

とすると、Scheme の対話バッファに切り替わる。ここで

```
STk> (hello)⟨Return⟩
```

とするとウィンドウが表示されるはずである。

B.6.5　ファイルを保存する

プログラムを書いたら、後日のためにセーブしておく。「ex-1.scm」というバッファで

C-x *C*-s

とするとセーブは完了する。

B.6.6　保存したファイルを編集する

「後日」になったら、保存したファイルを使ってプログラミングを再開する。まず、Emacs を起動して Scheme を走らせる。

M-x run-scheme⟨*Return*⟩

次に、前に保存したファイルを Emacs に読み込む。

```
C-x C-f
Find: ~/hello.scm⟨Return⟩
```

このファイルを、Scheme にロードさせる。

```
C-c C-l
Load Scheme file: (default hello.scm) ~/⟨Return⟩
```

これで、ファイルのロードは完了したので、Scheme 対話バッファで式を入力すれば実行できる。

```
C-c C-z
Stk> (hello)⟨Return⟩
;; ウィンドウが表示される
```

■