超特急: 一時間でわかる ML 超入門

住井 英二郎
eijiro.sumii@gmail.com

2005 年 3 月 5 日

ML って何？

ここでいう ML とは、Mailing List のことではなく、HTML や XML のような Markup Language でもなく、プログラミング言語の名前です。元々は「定理証明」という基礎研究の「メタ言語」(Meta Language)として 1970 年代に開発されたのですが、そんなことは気にしないで単に ML と呼ぶことが多いようです。

「そんな言語は聞いたこともない」という人のほうが多いと思いますが、ML を最初に開発した Robin Milner という研究者は、コンピュータ科学のノーベル賞といわれるチューリング賞を受賞しました。また、ML は世界の多くの大学・大学院で教えられ（東大、京大の情報系を含む)、プログラミング言語の研究などアカデミックな世界では常識になっています。様々なアプリケーションやライブラリが ML で開発され、その一部は

http://caml.inria.fr/cgi-bin/hump.en.cgi

や

http://www.npc.de/ocaml/linkdb/

などに紹介されています。

ML の特長

ML は Java や C#と違って、特定の企業が強力に宣伝しているわけではないので、どうしてもマーケティングでは負けてしまいます。が、様々な言語をベンチマークした"The Computer Language Shootout Benchmarks" (http://shootout.alioth.debian.org/)でもわかるように、ML は性能と簡単さの**両方**を兼備した、ほぼ唯一の言語です。実行速度では C の次で C++より上、プログラムの行数では Haskell, Perl, Ruby などと肩を並べています（Haskell は ML の親戚みたいなものですが)。

また、言語としては簡単であるにもかかわらず、Perl や Ruby と違ってコンパイル時にきちんとした「型チェック」が行われるので、ベンチマークだけでなく複雑なソフトウェアを開発するときも簡単&強力です。ちょっと誇張された極論ですが「ML の型チェックを

通ったプログラムはバグがない」といわれるぐらいです。ICFP プログラミングコンテスト(http://icfpcontest.org/)という、使用言語や参加資格に制限のない国際プログラミングコンテストでは、C や Haskell を抜いて ML がもっとも多く優勝しています（コンテストの課題はレイトレーシングや対戦ゲームなどで、参加数は圧倒的に C や C++が上回っているにも関わらず)。

問題: 次の文は合っているか間違っているか、○×をつけてください。

1. ML では実用のためのアプリケーションやライブラリがほとんど開発されていない。
2. ML は C や C++よりも遥かに遅い。
3. ML は学習や開発が難しく生産効率が低い。

解答: すべて×

能書きは良いからインストールしてみる

ML には様々な実装や変種がありますが、現在の主流は Objective Caml、省略して OCaml（オーキャムル）です。英語なら http://caml.inria.fr/index.en.html、日本語がよければ http://ocaml.jp/に情報があります。

ソースコードや Windows, RedHat Linux, Mac OS X のパッケージは

http://caml.inria.fr/download.en.html

からダウンロードできます。また、Debian GNU/Linux では OCaml のパッケージが標準で配布されています。もしソースコードからインストールする必要があっても、ほとんどの UNIX ならダウンロード・展開して「**./configure**」「**make world.opt**」「**make install**」だけで OK です。

インストールができたら、**ocaml** というコマンドを実行すると

```
> ocaml
        Objective Caml version 3.08.2

#
```

のようなメッセージが表示されるはずです（バージョン番号などは環境によります)。そこで、おもむろに「**123 + 456 ;;**」と打ってリターンキーを押すと

```
# 123 + 456 ;;
```

```
- : int = 579
#
```

のように答え **579** が出てきます。これで OCaml のインストールはとりあえず成功です。

ちなみに、ダイナミックリンクやグラフィックスなど、やや特殊な拡張機能もチェックしておきたければ

```
# #load "graphics.cma" ;;
# Graphics.open_graph "" ;;
- : unit = ()
# Graphics.draw_circle 100 100 50 ;;
- : unit = ()
#
```

と実行してみてください（**#load** の**#**も打つのを忘れないでください)。別のウィンドウが開いて、左下に小さな円が出てくれば OK です。インストールの方法によっては駄目かもしれませんが、すぐに差し支えはないので、あまり気にしなくて構いません。

整数と整数演算

ML では、プログラムは**式**として表現され、その**値**を計算することがプログラムの実行に相当します。…なんていうと難しく聞こえますが、要するにさっきの **123 + 456** みたいな式がプログラムで、それを実行して **579** という値を計算したわけです。このように式を計算して値を求めることを**評価**と呼び、「式 **123 + 456** を評価すると値 **579** が得られる」などと言います。

ML はただの電卓ではなくプログラミング言語なので、もっと複雑な処理もできますが、とりあえず簡単な式と値から紹介していきます。まず「**123**」「**456**」「**-789**」のような**整数**と、足し算、引き算、掛け算、割り算、余りなどの整数演算があります。

```
# 1 + 2 ;;
- : int = 3
# 3 - 4 ;;
- : int = -1
# 5 * -6 ;;
```

```
- : int = -30
# 7 / 3 ;;
- : int = 2
# 8 mod 3 ;;
- : int = 2
```

値の他に「**- : int =** 」という出力も表示されますが、とりあえず関係がないので無視してください。

もちろん、これらの式を組み合わせることもできます。括弧をつけることもできますし、もし括弧をつけなければ「掛け算や割り算は、足し算や引き算より先」という普通の順番になります。このような計算の順番のことを**優先順位**といい、「*****は**+**より優先順位が高い」等の言い方をします。

```
# 1 + 2 * 3 - 4 ;;
- : int = 3
# (1 + 2) * (3 - 4) ;;
- : int = -3
```

式は複数行にまたがって書くこともできます。一行でも複数行になっても、値は変わりません。

```
# (1 + 2) *
  (3 - 4) ;;
- : int = -3
```

ただし、変なところに改行を入れると意味がわかりにくくなることもあります。

```
# 1 + 2 *
  3 - 4 ;;
- : int = 3
```

ちなみに行末に入力する**;;**は、式の評価を開始する合図であり、式の一部ではありません。

問題: 次の式を評価したら、値はいくつになるでしょうか。

1. **3 + 7**

2. **1 mod 2 + 3 * 4 / 5**
3. **1 mod (2 + 3 * (4 / 5))**
4. **-3 - -7**

解答: 順番に **10**, **3**, **1**, **4**

いろいろな式と値、型

整数以外には、**浮動小数**や**文字列**などの値、浮動小数演算や文字列演算などの式があります。ただし ML では整数と浮動小数のように、異なる種類の値は厳しく区別されています。足し算や掛け算ですら、整数は**+**や*****、浮動小数は**+.**や***.**と異なります。

```
# 1.2 -. 3.4 ;;
- : float = -2.2
# 5.6 +. 7.8 *. 9.0 ;;
- : float = 75.8
# "hello" ^ "world" ;;
- : string = "helloworld"
```

先程から **int** とか **float** とか表示されているのは、このような値の種類のことです。このような値の種類のことを「型」と呼び、「値 **123** の型は **int** である」「式 **1.2 -. 3.4** の型は **float** である」等と言います。ML は型について厳しい言語で、型が合わないとエラーになります。

```
# "123" + 456 ;;
Characters 0-5:
  "123" + 456 ;;
  ^^^^^
This expression has type string but is here used with type int
# 3 * 7.0 ;;
Characters 4-7:
  3 * 7.0 ;;
      ^^^
This expression has type float but is here used with type int
```

これは一見すると面倒そうですが、複雑なソフトウェアの開発では非常に有効です。

問題: 次の中からエラーになる式を選んでください。

1. **"123" ^ "456"**
2. **"123" + "456"**
3. **123 +. 456.0**

解答: 2.と 3.

標準の関数を利用する

OCaml には、様々な関数やライブラリが標準で用意されています。たとえば、整数を浮動小数に変換する関数 **float_of_int** があります。関数を呼び出すには、その関数の名前と引数を並べて書けば OK です。

```
# float_of_int 123 ;;
- : float = 123.
# float_of_int (3 + 7) ;;
- : float = 10.
```

整数を画面に表示する関数 **print_int** のように、単に値を返すのではなく**副作用**を起こす関数もあります（このために ML は、副作用のない Haskell と違って、「純粋でない」関数型言語と言われます)。

```
# print_int 12345 ;;
12345- : unit = ()
```

print_int は**()**という値を返します。これはユニットと呼ばれる値で、型 **unit** を持ち、特に値がないときにダミーの値として使われます。

ユニットは返値だけでなく、引数として使われることもあります。たとえば、関数 **print_newline** は、引数としてユニットを受け取ると、副作用として改行を出力し、結果としてユニットを返します。引数も返値も特に要らないので、どちらもダミーの値としてユニットを使っているわけです。

```
# print_newline () ;;

- : unit = ()
```

このような標準の関数についてのマニュアルは、英語が

http://caml.inria.fr/pub/docs/manual-ocaml/libref/Pervasives.html

日本語訳が

http://ocaml.jp/archive/ocaml-manual-3.06-ja/libref/Pervasives.html

にあります。たとえば、平方根を計算する関数は **sqrt**、文字列を画面に表示する関数は **print_string** であることがわかります。exception（例外）とか reference（参照）とか、まだ説明していない言葉もありますが、とりあえず無視して結構です。

標準のライブラリを利用する

電卓みたいな計算だけではつまらないので、OCaml に標準で用意されているライブラリ・モジュールを利用して、もうちょっとプログラミング言語っぽいことをしてみましょう。ライブラリには、最初からロードされているものと、そうでないものがあります。たとえば、乱数生成のためのモジュール **Random** は、はじめからロードされていて

```
# Random.init 12345 ;;
- : unit = ()
# Random.int 10 ;;
- : int = 7
# Random.int 10 ;;
- : int = 1
# Random.int 10 ;;
- : int = 1
```

のように利用できます。**Random.init** は乱数生成器を初期化する関数、**Random.int** は与えられた正整数より小さい、ランダムな非負整数を返す関数です。別の例としては、カレントディレクトリの名前を取得する関数 **Sys.getcwd** や、カレントディレクトリを変更する関数 **Sys.chdir** などがあります。

```
# Sys.getcwd () ;;
- : string = "/home/sumii"
# Sys.chdir "/home/sumii/tmp" ;;
- : unit = ()
# Sys.getcwd () ;;
- : string = "/home/sumii/tmp"
# Sys.chdir ".." ;;
```

```
- : unit = ()
# Sys.getcwd () ;;
- : string = "/home/sumii"
```

一般に、モジュールで定義されている関数や定数は「モジュールの名前.関数や定数の名前」という形で使うことができます。このように最初からロードされているモジュールの一覧は

http://caml.inria.fr/pub/docs/manual-ocaml/manual034.html

和訳は

http://ocaml.jp/archive/ocaml-manual-3.06-ja/manual034.html

にあります（ただし和訳は一部未完のようです)。

最初からロードされていないライブラリとしては、

- UNIX のシステムコール機能を提供する **Unix** モジュール（Windows でも利用可能）
- 正規表現による文字列処理機能を提供する **Str** モジュール
- 簡単なグラフィックス機能を提供する **Graphics** モジュール

等々があります。たとえば、**Str** モジュールだったら、まず

```
# #load "str.cma" ;;
```

としてライブラリをロードし（**#load** の**#**も忘れずに打ってください）

```
# Str.global_replace (Str.regexp "f...") "xxxx" "abcdefghijkl" ;;
- : string = "abcdexxxxjkl"
```

のように文字列検索や置換などの機能を利用することができます。これらのモジュールのマニュアルは

http://caml.inria.fr/pub/docs/manual-ocaml/

ないし

http://ocaml.jp/archive/ocaml-manual-3.06-ja/

の「Part IV」にあります。

変数や関数を定義する

さて、ここで問題です。次の式を計算した結果はいくつになるでしょうか。

1. **2.0 *. 10.0 *. 3.141592653589793**
2. **10.0 *. 10.0 *. 3.141592653589793**
3. **20.0 *. 20.0 *. 3.141592653589793**

計算自体は簡単ですが、同じ数 **3.141592653589793** を何度も入力するのは、いかにも非生産的です。こういうときは、その数を表す**変数**を定義するのがプログラミングの常套手段です。そうすれば、たとえば円周率として用いる数を後から変えたくなっても、一回の変更で OK となり、プログラムの保守が簡単になります。OCaml では

```
# let pi = 3.141592653589793 ;;
val pi : float = 3.14159265358979312
```

のように、「**let** 変数の名前 **=** 式」という構文で変数を定義することができます。定義した変数は、式の中で普通に使用することができます。

```
# 2.0 *. 10.0 *. pi ;;
- : float = 62.8318530717958623
# 10.0 *. 10.0 *. pi ;;
- : float = 314.159265358979326
# 20.0 *. 20.0 *. pi ;;
- : float = 1256.6370614359173
```

しかし「**10.0 *. 10.0 *. pi**」と「**20.0 *. 20.0 *. pi**」のあたりは、まだ同じような計算をしていて無駄な感じがします。こういうときは、浮動小数 **r** を**引数**（ひきすう）とし、**r *. r *. pi** を**返値**（かえりち）とする**関数**を定義するのが鉄則です。OCaml では、やはり **let** 構文を使用して関数を定義することができます。

```
# let area_of_circle r = r *. r *. pi ;;
val area_of_circle : float -> float = <fun>
# area_of_circle 10.0 ;;
- : float = 314.159265358979326
# area_of_circle 20.0 ;;
- : float = 1256.6370614359173
```

関数 **area_of_circle** の型は **float -> float** になります。これは「浮動小数を受け取って浮動小数を返す関数」という意味です。

複数の引数

引数が複数の関数を定義したかったら、二通りのやり方があります。一つは、引数を **(x, y)** のような**組**にして受け取るやり方です。

```
# let distance (x, y) = sqrt (x *. x +. y *. y) ;;
val distance : float * float -> float = <fun>
# distance (3.0, 4.0) ;;
- : float = 5.
```

distance は **float * float -> float** という型を与えられます。これは「**float** と **float** の組を受け取って **float** を返す関数」という意味です。

もう一つは、引数を一つずつ順に受け取るやり方です。

```
# let distance x y = sqrt (x *. x +. y *. y) ;;
val distance : float -> float -> float = <fun>
# distance 3.0 4.0 ;;
- : float = 5.
```

この **float -> float -> float** という型は「二つの **float** を一つずつ順に受け取って、**float** を返す関数」という意味です。

OCaml では、どちらかといえば後者が普通のスタイルとされています（**カリー化**といいます)。前者と後者では型が違うので、混同できないことに注意してください。

```
# let distance x y = sqrt (x *. x +. y *. y) ;;
val distance : float -> float -> float = <fun>
# distance (3.0, 4.0) ;;
Characters 9-19:
  distance (3.0, 4.0) ;;
           ^^^^^^^^^^
This expression has type float * float but is here used with type
float
```

再帰関数

さて、またまた問題です。次の式の値を求めてください。

1. **1**
2. **1 * 2**
3. **1 * 2 * 3**
4. **1 * 2 * 3 * 4**
5. **1 * 2 * 3 * 4 * 5**
6. **1 * 2 * 3 * 4 * 5 * 6**

これも計算自体は簡単ですが、やはり同じような式の繰り返しで、手動で入力するのは無駄な印象です。

そこで、正の整数 **n** を受け取って、**1** から **n** までの整数を掛けた値を返す関数 **factorial** を考えてみます。まず、もし **n** が **1** だったら、**factorial 1** は「**1** から **1** までの整数を掛けた値」なので、明らかに **1** を返します。では、もし **n** が **1** より大きかったらどうでしょうか。さっきの問題を見ればすぐにわかると思いますが、**n > 1** のときは、一つ前の **factorial (n - 1)** に **n** を掛けると、**factorial n** が求まります。

以上のような関数を、OCaml では次のように書くことができます。

```
# let rec factorial n =
    if n = 1 then 1 else
    factorial (n - 1) * n ;;
val factorial : int -> int = <fun>
# factorial 1 ;;
- : int = 1
# factorial 2 ;;
- : int = 2
# factorial 3 ;;
- : int = 6
# factorial 10 ;;
- : int = 3628800
```

factorial は型 **int -> int** すなわち整数を受け取って整数を返す関数ですが、定義の

中で **factorial** 自身を呼び出しています。このように自分自身を用いて定義された関数を**再帰関数**と呼びます。OCaml では、再帰関数を定義するときは、**let** の後に **rec** というキーワードを入れることになっています。2 行目の **if** ... **then** ... **else** ...は、**if** の後が真であれば **then** の部分が評価され、偽であれば **else** の部分が評価される、という式です。

余談ですが、このように C や Perl などの命令型言語ではループを利用するような場合でも、OCaml や Haskell のような関数型言語では再帰を利用するほうが普通です。変数の中身が後から変わる**破壊的代入**が不要となり、プログラムの見通しが良くなってバグが減ると考えられているからです。いまだに Fortran 言語など 1960 年代以来の伝統で「再帰はループより非効率的」という人もいますが、現代のコンピュータで問題になるほどの違いはありませんし、「末尾再帰」という処理によりループとまったく同様にコンパイルできることも少なくありません。

問題: 次の再帰関数は何を計算するか、当ててください。

1.
```
let rec sum n =
  if n = 0 then 0 else
  sum (n - 1) + n
```
2.
```
let rec gcd m n =
  if m = 0 then n else
  if m > n then gcd (m mod n) n else
  gcd (n mod m) m
```
3.
```
let rec power m n =
  if n = 0 then 1 else
  if n mod 2 = 0 then power (m * m) (n / 2) else
  m * power m (n - 1)
```

解答:

1. 1 から n までの整数の総和
2. 非負整数 m と n の最大公約数
3. m の n 乗

2.は「m と n の最大公約数 = (m mod n)と n の最大公約数」という性質を利用したアルゴリズム、3.は $m^{2n} = (m^2)^n$ という性質を利用したアルゴリズムです。どちらも巧妙なアルゴリズムですが、ループで再帰よりわかりやすく書くのは難しいでしょう。

高階関数、無名関数

「同じような計算を繰り返すときは、関数を使うと簡単になる」という話をしましたが、次のような場合はどうでしょうか。

1. `(3 + 7) + (123 + 456)`
2. `(3 - 7) - (123 - 456)`
3. `(3 * 7) * (123 * 456)`
4.
```
let h x y = x * x + y in
h (h 3 7) (h 123 456)
```

今度は出てくる数字は同じなのですが、計算のほうが違っています。ただし **let … in …** というのは、**in** の中でだけ使えるローカル変数や関数を定義する OCaml の構文です。

ML では、こういうときも関数を利用することができます。

```
# let f g = g (g 3 7) (g 123 456) ;;
val f : (int -> int -> int) -> int = <fun>
```

f は「関数 **g** を受け取って、**g (g 3 7) (g 123 456)** を計算した結果を返す」という関数になります。**f** のように、関数を引数として受け取る（ないし関数を結果として返す）ような関数のことを**高階関数**といいます。高階関数を使うと、先のような処理の繰り返しも簡単になります。

```
# let g1 x y = x + y in f g1 ;;
- : int = 589
# let g2 x y = x - y in f g2 ;;
- : int = 329
# let g3 x y = x * y in f g3 ;;
- : int = 1177848
# let g4 x y = x * x + y in f g4 ;;
- : int = 15841
```

いちいち **g1**, **g2**, **g3**, **g4** などを定義するのが面倒だったら、

```
# f (fun x y -> x + y) ;;
```

```
- : int = 589
# f (fun x y -> x - y) ;;
- : int = 329
# f (fun x y -> x * y) ;;
- : int = 1177848
# f (fun x y -> x * x + y) ;;
- : int = 15841
```

のように書くこともできます。**fun x y ->** ...は「**x** と **y** を順に受け取って...を返す」という**無名関数**を表現する構文です。より一般に、引数の名前は **x** や **y** でなくても良いですし、引数は一個でも三個以上でも構いません。

関数としての二引数演算子

+や**-**のような二引数演算子に限っては、**fun x y -> x + y** などのかわりに**(+)**のように略すこともできます。

```
# f (+) ;;
- : int = 589
# f (-) ;;
- : int = 329
```

マニュアルで**+**や**-**などが

```
val (+) : int -> int -> int
val (-) : int -> int -> int
```

のような形で載っているのは、これと同じ書き方を使っているわけです。

ただし特別な場合として、*****に括弧をつけるときだけは前後に空白が必要です。 **(***はコメントの始まりを、***)**はコメントの終わりを表すためです。

```
# f ( * ) ;;
- : int = 1177848
```

このことを忘れると、よくわからないエラーが出てしまいます。

```
# f (*) ;;
Characters 2-5:
Warning: this is the start of a comment.
  f (*) ;;
    ^^^
* *) ;;
- : (int -> int -> int) -> int = <fun>
```

関数を定義するスタイル

ついでですが、**fun** を使うと同じ関数をいろいろなスタイルで定義することもできます。たとえば

```
# let distance x y = sqrt (x *. x +. y *. y) ;;
val distance : float -> float -> float = <fun>
```

のかわりに

```
# let distance = fun x y -> sqrt (x *. x +. y *. y) ;;
val distance : float -> float -> float = <fun>
```

や

```
# let distance = fun x -> fun y -> sqrt (x *. x +. y *. y) ;;
val distance : float -> float -> float = <fun>
```

のように書いても同じです。引数を一つずつ順に受け取るのではなく、組としてまとめて受け取るスタイルについても、

```
# let distance (x, y) = sqrt (x *. x +. y *. y) ;;
val distance : float * float -> float = <fun>
```

のかわりに

```
# let distance = fun (x, y) -> sqrt (x *. x +. y *. y) ;;
```

```
val distance : float * float -> float = <fun>
```

と定義することが可能です。ただし、あまりわかりにくくなってもしようがないので、普通は最初の **let distance x y =** ...という構文を使用します。

問題: 次の中に一つだけ型の違う式があります。どれでしょうか。

1. **fun x y -> abs (x - y)**
2. **fun x -> fun y -> abs (x - y)**
3. **let dist x y = abs (x - y) in**
   dist
4. **let dist (x, y) = abs (x - y) in**
   dist
5. **let dist (x, y) = abs (x - y) in**
   fun x y -> dist (x, y)

ただし **abs** は整数を受け取って、その絶対値を返す関数です。

解答: 4.

再帰データ型

整数や浮動小数だけの計算には限界があります。高度なプログラムを開発するには、高度なデータ構造が必要です。ML では、すでに出てきた **(x, y)** のような組の他に、「**x** または **y**」のような型を定義することができます。型の再帰も可能です。これらを用いると様々なデータ構造を表すことができます。

たとえば、整数を葉とする「木」は次のように定義できます。

```
# type int_tree = Leaf of int | Node of int_tree * int_tree ;;
type int_tree = Leaf of int | Node of int_tree * int_tree
```

これは「**int_tree** は **Leaf**（葉）または **Node**（枝）であり、**Leaf** は整数、**Node** は **int_tree** と **int_tree** の組からなる」という意味の定義です。

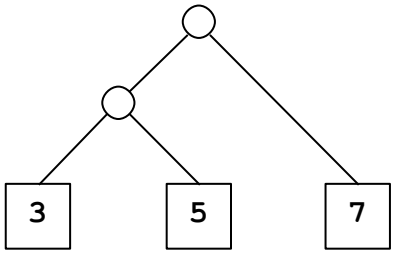
たとえば **Leaf 123** という式は、整数 **123** を持つ葉が一つだけの木を表します。

```
# Leaf 123 ;;
- : int_tree = Leaf 123
```

Node(Leaf 123, Leaf 456)だったら、枝が一つあって、整数 **123** を持つ葉が左に、整数 **456** を持つ葉が右にある、という木になります。

```
# Node(Leaf 123, Leaf 456) ;;
- : int_tree = Node (Leaf 123, Leaf 456)
```

Node(Node(Leaf 3, Leaf 5), Leaf 7)という式は、下のような木を表すことになります。ただし○が枝、□は葉を表すものとします。



もう少しややこしい例として、とにかくランダムに木を作る関数は、次のように書くことができます。

```
# let rec make_random_tree () =
    if Random.int 2 = 0 then Leaf(Random.int 10) else
    Node(make_random_tree (), make_random_tree ()) ;;
val make_random_tree : unit -> int_tree = <fun>
# make_random_tree () ;;
- : int_tree = Node (Leaf 8, Node (Node (Leaf 8, Leaf 3), Leaf
1))
# make_random_tree () ;;
- : int_tree = Leaf 3
# make_random_tree () ;;
- : int_tree = Node (Node (Leaf 3, Node (Leaf 4, Node (Leaf 0,
Leaf 2))), Leaf 8)
```

make_random_tree は引数が特に要らないので、ダミーの値**()**を受け取ることに気をつけてください。そもそも木というデータ構造自体が再帰により定義されているので、再帰関数によって非常に自然な記述が可能となっています。

パターンマッチング

さて、先のように木を作っても使えなくてはしようがないので、作った木に対して何か処理や操作をする手段が必要です。そのために ML では**パターンマッチング**という仕組みを用います。

たとえば、**int_tree** の葉の値を合計する関数 **int_tree_sum** は、以下のように定義できます。

```
# let rec int_tree_sum t =
    match t with
      Leaf i -> i
    | Node(l, r) -> int_tree_sum l + int_tree_sum r ;;
val int_tree_sum : int_tree -> int = <fun>
# int_tree_sum (Node(Node(Leaf 3, Leaf 5), Leaf 7)) ;;
- : int = 15
```

まず、関数 **int_tree_sum** は引数 **t** を受け取ります。その次の **match** … **with** …という式がパターンマッチングです。そもそも「**int_tree** は **Leaf** または **Node** である」と定義したので、**int_tree** について調べたかったら、まず **Leaf** なのか **Node** なのか場合わけして、もし **Leaf** だったら中身の整数を、**Node** だったら左の枝と右の枝を調べることになります。上の例では、**t** が **Leaf** だったら中身の整数 **i** を返し、**Node** だったら左の枝 **l** と右の枝 **r** のそれぞれについて、関数 **int_tree_sum** を呼び出して再帰を行っています。**make_random_tree** の場合と同様に、再帰データ構造である木に対する処理が、再帰関数によって非常に自然に記述できることがわかります。

なお、このような関数の定義では **let f x = match x with** …という形がよく出てくるので、省略することが可能になっています。たとえば

```
# let rec int_tree_sum = function
      Leaf i -> i
    | Node(l, r) -> int_tree_sum l + int_tree_sum r ;;
val int_tree_sum : int_tree -> int = <fun>
```

という具合です。一般に **fun x -> match x with** …というパターンマッチング関数を、

function ...のように省略することが可能です。

多相データ型と多相関数

さて、上の例では整数を葉とする木を定義しましたが、もし浮動小数を葉とする木、文字列を葉とする木、整数の組を葉とする木…等々も定義したくなったら、別々に定義するのはいかにも非効率的ですし、「同じようなことをしているコードは一つにまとめる」というプログラミングの鉄則にも違反します。

そこで ML では「任意の型αについて、αを葉とする木」のような**多相データ型**を定義することができます。このαのことを**型変数**といいます。ただし普通の環境でαのようなギリシャ文字を入力するのは面倒なので、**'a** のように表記するのが通常です。

```
# type 'a tree = Leaf of 'a | Node of 'a tree * 'a tree ;;
type 'a tree = Leaf of 'a | Node of 'a tree * 'a tree
# Leaf 123 ;;
- : int tree = Leaf 123
# Node(Leaf 4.56, Leaf 78.9) ;;
- : float tree = Node (Leaf 4.56, Leaf 78.9)
# Node(Leaf "abc", Node(Leaf "def", Leaf "ghi")) ;;
- : string tree = Node (Leaf "abc", Node (Leaf "def", Leaf
"ghi"))
```

このようなデータ型に対して、たとえば木の高さを求める関数 **height** は次のように書くことができます。

```
# let rec height = function
      Leaf _ -> 0
    | Node(l, r) -> 1 + max (height l) (height r) ;;
val height : 'a tree -> int = <fun>
```

Leaf _の**_**はダミーのパターンで、"don't care"と呼ばれます。とにかく **Leaf** であれば高さ **0** なので、中身は「気にしない」というわけです。**Node** の場合は、左の枝と右の枝の高さの大きいほう(**max**)に **1** を足して返します。この関数は、どんな値を葉に持つ木でも受け取れるので、**'a tree -> int** という**多相的**な関数型を与えられます。このような関数のことを**多相関数**といいます。

一般に木やリストといった、要素の集まりを表す**コンテナ**の定義では、このような多相関数・多相データ構造が特に役に立ちます。ちなみに C++の STL（標準テンプレートライブラリ）は、まさに多相関数や多相データ構造のライブラリなのですが、プログラマが複雑な型を書かないといけなかったり、マクロのようにインライン展開してからコンパイル時チェックが行われるので、エラーメッセージがとてもわかりにくかったりします。ML では、複雑な多相関数や高階関数の型も自動で決定されますし（**型推論**といいます)、きちんと別々に型チェックされるので、C++のような問題はありません。

リスト

ML では特別な多相データ型として、**リスト**が最初から定義されています。ML のリストは、空のリスト**[]**か、要素 **x** をリスト **y** に連結した **x :: y** という形（「cons セル」といいます）のどちらかです。先の **tree** と違って最初から定義されているので、普通の構文ではないのですが強いていえば

```
type 'a list = [] of unit | :: of 'a * 'a list
```

のように考えることもできなくはありません。

たとえば、とにかくランダムにリストを生成する関数 **make_random_list** は

```
# let rec make_random_list () =
    if Random.int 3 = 0 then [] else
    Random.int 10 :: make_random_list () ;;
val make_random_list : unit -> int list = <fun>
# make_random_list () ;;
- : int list = [4]
# make_random_list () ;;
- : int list = [3; 2; 3; 1; 7; 5; 9; 2; 1; 4; 9; 1]
# make_random_list () ;;
- : int list = []
```

のように定義できますし、リストの長さを返す関数 **length** は

```
# let rec length = function
```

```
    [] -> 0
  | _ :: list -> 1 + length list ;;
val length : 'a list -> int = <fun>
# length [1; 2; 3] ;;
- : int = 3
```

のように定義できます。ただし**[1; 2; 3]**という形の式は、**1 :: 2 :: 3 :: []**などのリストを略記した構文です。

命令型言語の機能: 参照

ML はいわゆる関数型言語とされており、破壊的代入などの副作用を無闇に使用することは一般に推奨されていません。が、必要な場合のために、命令型言語の機能も用意されています。たとえば、呼び出すたびに一つずつ増える整数を返す関数 **count** は次のように書くことができます。

```
# let counter = ref 0 ;;
val counter : int ref = {contents = 0}
# let count () =
    counter := !counter + 1;
    !counter ;;
val count : unit -> int = <fun>
# count () ;;
- : int = 1
# count () ;;
- : int = 2
# count () ;;
- : int = 3
```

はじめの **ref** という演算子は、新しい格納場所（**参照セル**）を作って、与えられた値で初期化します。演算子**!**は、参照セルの現在の値を返します。**:=**は参照セルへの破壊的代入です。なので、たとえば **counter := !counter + 1** は、参照セル **counter** の値を一つ増やすことになります。…**;**…というのは式を順に評価する構文です。ただし**;**の前の式の値は捨てられてしまうので、ダミー値（**unit** 型）以外だと警告されます。

例外処理

Java や C++のような**例外処理**の機能もあります（実は ML のほうが先ですが）。**exception** 構文で例外データ型を定義し、**raise** で例外を発生、**try** … **with** …で例外を捕獲します。たとえば次のプログラムは、与えられた整数リストの要素をすべて掛け算しつつ、もし **0** があったら例外を発生して、すぐに計算を中断します。

```
# exception Zero ;;
exception Zero
# let rec multiply_int_list = function
    [] -> 1
  | i :: l ->
      if i = 0 then raise Zero else
      i * multiply_int_list l ;;
val multiply_int_list : int list -> int = <fun>
# multiply_int_list [2; 3; 4] ;;
- : int = 24
# multiply_int_list [5; 0; 6] ;;
Exception: Zero.
# try
    multiply_int_list [5; 0; 6]
  with Zero -> 0 ;;
- : int = 0
```

モジュールと分割コンパイル

大きなソフトウェアを作るときは、プログラムを複数の**モジュール**に分割してコンパイルしたくなります。OCaml では、「モジュールの名前**.ml**」というファイルに実装を、「モジュールの名前**.mli**」というファイルに**インターフェース**を記述するだけで、簡単に分割コンパイルができます。インターフェースには関数や定数、データ型や例外の宣言を記述します。

たとえば **Test** という名前のモジュールを作成したかったら、まず **test.ml** というファイルに

```
let x = 123
```

```
let f y = x + y
let _ = print_int (f 456)
```

のように実装を定義しておき、**test.mli** というファイルに

```
val x : int
val f : int -> int
```

のようにインターフェースを宣言しておけば、他のモジュールから **Test.x** という整数や **Test.f** という関数が使えます。ただしインターフェースに記述されていない定義は、外部から参照できません。

もちろん、

```
> ocamlopt test.mli test.ml -o test
> ./test
579
```

のようにコンパイル実行も可能です。ファイルの数が多くなってきたら、OCamlMakefile

http://www.ocaml.info/home/ocaml_sources.html

という、コンパイルなどを自動でやってくれる便利なツールもあります。

おわりに

プログラミング言語 ML の特徴をものすごい勢いで述べてきましたが、細かい話になるので触れていない点なども少なくありません。というか、飛ばしたことのほうが多いです（タイトルからして「超特急」「超入門」ですし）。英語なら http://caml.inria.fr/index.en.html 、日本語では http://ocaml.jp/ や http://www.sato.kuis.kyoto-u.ac.jp/~igarashi/class/isle4-05w/mltext/ や http://www.i.kyushu-u.ac.jp/~bannai/ocaml-intro/に、チュートリアルやテキスト、メーリングリストなど様々な情報がありますので、もし時間があったらぜひとも参照してみてください。

そして貴方も ML の世界へ…