ソフトウェア技法: No.1 (概要、基礎)

亀山幸義 (筑波大学情報科学類)

1 関数プログラミング

関数プログラミング (Functional Programming) は、**関数** の概念を中心にすえたプログラミングである。関数型プログラム言語は、関数の概念を定式化した計算モデルである **ラムダ計算** に基づくプログラミング言語であり、関数プログラミングを行うのに適した様々な機能を持っている。

ラムダ計算 (lambda calculus) は、Alonzo Church らが創始した体系 (およびそれを発展・変形した体系の総称) であり、関数の概念のみをモデル化したシンプルなものでありながら、現代の計算機の 1 つの計算モデルである **チューリング機械**(Turing Machine) と同等の表現力 (計算可能性) を持つという著しい特徴を持つものである。

ここで出てきたラムダ計算、チューリング機械、計算可能性は、計算機科学における極めて重要な題材であるが、本授業ではカバーできないので、他の授業あるいは独学で勉強されたい。(たとえば、3-4 年次の「計算モデル論」)

1.1 関数プログラミングと関数の概念

なぜ、関数に基づいたプログラムを書くと、良いのだろうか？これは深いテーマであり、簡単に答えが出せる質問ではないが、いくつかの利点をあげることはできる。

- プログラム (あるいは計算) が、入力データをもらって出力データを返すと考えたとき、最も自然なモデルが、関数あるいは部分関数[*1]である。
- 計算モデルの 1 つであるラムダ計算は、長い研究の歴史を持ち、深く研究されていて理論的基礎付けがしっかりしている。これにより、プログラム言語の設計がしっかりした基礎のもとになされることになり、ひいては、プログラムの検証 (正しさの保証) やプログラムの解析 (性質の分析) がやりやすくなる。

ところで、関数型プログラム言語とは呼ばれていない他の多くのプログラム言語 (たとえば、C 言語など) も、関数と呼ばれる構文要素をもっている。それらの言語を、関数型プログラム言語と呼ばないということは、C 言語の関数と、ラムダ計算や関数型プログラム言語の「関数」の概念は、異なるのであろうか。これの答え (の一部) は次の表で与えられる。

	同じ引数を与えると、いつでも同じ答えか？	普通の変数の値を書換えられるか？
数学の部分関数	Yes	No
Haskell 言語の関数	Yes	No
OCaml 言語の関数	No	No
Lisp 言語や C 言語の関数	No	Yes

この表は、上に行くほど数学的な関数 (正確には部分関数) に近くなり、下に行くほど遠くなるということをあらわす。表にある通り、関数型プログラム言語の中でも Haskell 言語では、同じ関数に同じ入力 (実引数) を与えると、いつでも同じ答えが返るため、数学的な部分関数に非常に近い。

OCaml 言語では、その性質は厳密には成立しない。しかし、普通の変数の値を書換える機能 (C 言語等における x:=x+3 などの機能) は持たず、値の書換えをする場合は特別なデータを作る必要がある、など、「副作用を伴う計算」は、その他の部分 (副作用を伴わない計算) と分離されている。そして、プログラムの大部分では、副作用を持たない計算で書けるよう工夫されており、そのような部分は、数学的な部分関数と同じ性質をもつので、プログラムの性質の推論や検証は、やりやすい。実際に OCaml のプログラムの多くの部分は、そのような副作用を持たないプログラムとして表現できるので、プログラムの検証等がやりやすい言語であるといえる。

[*1] 部分関数 (partial function) は、関数に似ているが、入力データによっては、出力が未定義のことがあるものである。たとえば、実数上の割り算や三角関数の tan は、引数の値によっては対応する値が存在しないので、関数ではなく部分関数である。なお、全ての関数は部分関数でもある。

一方で、Lisp 言語は、関数型プログラム言語に分類されてはいるが、変数の値を書換えるなどの操作がどの変数に対してもできてしまうため、Lisp の「関数」は、どのような副作用をもつか予測が困難であり、数学的な関数・部分関数の概念からは遠いものである。すなわち、Lisp のプログラムの解析や検証は、C 言語等と同程度に困難であることが多い。なお、Lisp (およびその一種といえる Scheme) では、副作用を無闇に使わないようにプログラムを書くのが普通であり、そのようなプログラムだけを書くのであれば、「関数プログラミング」と見なすことができるので、「関数プログラミングをすることもできるプログラム言語」とは言える。

このように、ひとくちに「関数型プログラム言語」といっても、それがもつ関数の概念は様々である。しかし、多くの関数型プログラム言語が、(C 言語などよりは) 数学的な関数に近い「関数」概念をもち、プログラムの検証や解析がやりやすくなることが多い、という傾向にある。

1.2 関数型プログラム言語の種類

現在広く使われている関数型プログラム言語には、主に以下の 3 種類がある。

- Lisp; Common Lisp や Scheme; 動的に型付けられる。
- ML ファミリー; SML や OCaml や F#; 静的に型付けられる。値呼び計算方式。
- Haskell; 静的に型付けられる。必要呼び計算方式。

以上のものは、関数型言語としての機能が主たるものであるが、最近では、種々のプログラム言語に関数型の機能が取り込まれている。たとえば、筑波大学情報科学類の卒業生である、まつもとゆきひろさんが設計した Ruby のほか、twitter の記述言語として有名になった Scala や、ウェブブラウザでの標準的言語である JavaScript は、オブジェクト指向言語をベースとして関数型言語の機能を取り込んだものになっている。また、最近では C++ 言語に関数クロージャの概念が取りこまれている。

この授業では、代表的な関数型プログラム言語の 1 つとして、OCaml を取りあげる。OCaml は ML の 1 つである。ML は meta-langauge(メタ言語) の略であり、もともとは、証明システム PCF を記述するための言語[*2]として、Robin Milner[*3]が設計した言語である。ML は最初は SML (Standard ML) という言語だけであったが、その後、様々に拡張・変更された言語が提案されており、その全体を ML ファミリーと言う。OCaml は SML に対抗して、フランス INRIA で開発されたプログラム言語である。OCaml という名前は、もともとは Objective Caml の省略形だったが、今では省略形ではなく、本名が OCaml になっている。読み方は「オーキャメル」あるいは「オーキャムル」である。

OCaml は、現在、世界中の研究、教育、産業界で使われているプログラム言語である。たとえば、対話的定理証明系 Coq などの記号処理の分野のシステム、種々のコンパイラの記述のほか、証券会社のトレーディングシステムや飛行機の予約システムなど信頼性を要求されるもので使わているほか、オペレーティングシステムの世界でユニカーネルとして有名な Mirage システムも OCaml で書かれている。

1.3 OCaml の資料

OCaml のチュートリアル的なテキストはインターネット上にもいくつかあるが、日本語による入門的な書籍が 2007 年に 3 冊刊行された。これらのうち、自分にとって読みやすそうな 1 冊を買うことを勧める。

- 書籍: 「プログラミングの基礎」, 浅井健一著, サイエンス社, ISBN 978-4-7819-1160-1 (2007 年).
- 書籍: 「プログラミング in OCaml〜関数型プログラミングの基礎から GUI 構築まで」、五十嵐淳著、技術評論社, ISBN 978-4-7741-3264-8, (2007 年).
- 書籍: 「入門 OCaml」、OCaml-Nagoya 著, 毎日コミュニケーションズ、ISBN 978-4-8399-2311-2 (2007 年).

また、無料のチュートリアルテキストもインターネット上にいくつもある。そのいくつかをあげる。(この他にも多数あるので、自分で探してほしい。)

- OCaml チュートリアルの日本語訳: `ocaml.jp/`**チュートリアル**

[*2] 何らかの言語を記述するための言語を「メタ言語」と言う

[*3] Turing 賞受賞者

- ITpro というオンラインの雑誌における OCaml に関するわかりやすい解説
  `itpro.nikkeibp.co.jp/article/COLUMN/20060808/245371/`
- 名古屋大学のジャック・ガリグ先生の 2016 年 筑波大学集中講義の資料
  `www.math.nagoya-u.ac.jp/~garrigue/lecture/tsukuba16/`
- OCaml の開発元 (フランス) の資料: `http://caml.inria.fr`

2 OCaml の起動と終了、対話的環境

coins 計算機システム (筑波大学情報科学類計算機システム) には、OCaml がインストールされているので、特に何もセットアップしなくても OCaml を起動できる。起動コマンドは、すべて小文字で ocaml であり、これを，shell が動いている端末から起動すると、OCaml の対話モードにはいる。(なお，emacs から便利に使う方法は，後述する。)

```
% ocaml
        OCaml version 4.02.2

# 1 + 2 * 3;;
- : int = 7
# print_string "abc\n";;
abc
- : unit = ()
#
```

対話モードでは、ユーザの入力に応じて OCaml 処理系が計算を行い、結果の値とその型を表示する。計算結果だけでなく、型も表示するところが、OCaml らしいところである。

行頭の#はプロンプトである。上記の最初の例では、`1 + 2 * 3;;` がユーザの入力であり、7 が結果の値、そして、`int` がその型である。2 つ目の例では、`print_string "abc\n";;` が入力であり、その実行の際に abc という文字列が表示される。(実際には abc という文字列のあとに改行コードが出力されている。) この場合、計算結果は無意味な値 `()` であり、その型は `unit` である。

この 2 つの計算だけでも、「型」とは何か、-(マイナス) の記号は何だろう、等々の疑問がわいてきたと思うが、それらは後述することにして、ここでは、処理系の終了方法をチェックしておこう。

OCaml を終了したい時は、正式には `#quit;;` と入力するが、もっと簡単に、control-D (control キーを押しつつ "D" を押す) を入力してもよい。なお、shell から control-Z を入力することにより、OCaml 処理系を中断することもできるが、多数の OCaml 処理系プロセスを中断したままでいると、計算機に大きな負荷をかけてしまうことがある。

2.1 ファイルの作成と読み込み

上記のように、ちょっとしたテストの時には対話モードが良い。一方、少し長いプログラムを書いて走らせるときには、プログラムをファイルに書きこんで、そのファイルを OCaml から読み込む (読み込みながら実行する) という方法が良い。この読み込み操作も、対話モードから行う。OCaml プログラムを格納するファイルは、拡張子を .ml とすること。

```
# #use "myprogram.ml";;
```

が 2 つ重なっていて読みにくいが、1 つ目の # は OCaml プロンプトであり、そのあとがユーザの入力である。つまり、上記のものは、`#use "myprogram.ml"` というコマンドを入力したものである。このようにすると、現在の directory の下にある `"myprogram.ml"` というファイルを読み込み、上から順番に実行してくれる。

なお、絶対パスを記述することもできる。

```
# #use "/home/prof/kam/ocaml/myprogram.ml";;
```

このファイル myprogram.ml には、OCaml の式を何個でも記述してよい。ただし、計算がエラーとなる式が途中に 1 つでもあると、読み込みを終了してしまう。

区切りとしての ;; についての注意. 対話モードで入力するときは、1 つの式を入力し終わるごとに ;; (セミコロンを 2 つ連続) をタイプする必要があるのに対して、ファイルにたくさんの式を続けて書く時には、(もし、OCaml 処理系が区切りを自動的に認識できるなら) ;; で区切る必要はない。この理由により、ファイルとして提供するソースコードでは、;; で区切っていない式たちもときどき登場する。このようなコードを、対話モードで直接入力してテストしようと思ったら、;; で区切る必要があるので注意されたい。

2.2 emacs からの便利な使い方

emacs から OCaml プログラムの編集をするために便利なツールとして、tuareg-mode というものがある。

- 初期設定: 自分の `~/.emacs` ファイルの末尾に，`/home/prof/kam/ocaml/append-tuareg-coins.el` の中身を追加する。(Unix shell では `cat xxx >> yyy` とすると，ファイル yyy の末尾に ファイル xxx の内容が追加される．不等号を 2 つ続けて書くことに注意せよ．もともとあるファイルを壊す危険があるので，かならず，事前に `cp ~/.emacs ~/.emacs-backup` 等として，バックアップを取っておくこと．)
  そのあと，一度，emacs を抜けて，もう一度 emacs を起動する。上記のファイルの追加に失敗したら，起動時にエラーがでる．
- 使い方 1: emacs で，xxx.ml という名前のファイル (OCaml のプログラムがはいったファイル) を編集すると，キーワードがきれいに色分けされ，ファイルが見やすくなる。
  なお、xxx.ml というファイルを編集するとき、TAB キーを押すと、適切なインデントをしてくれる。
- 使い方 2: 式の上にカーソルを置いて，C-c C-e (control-c control-e のこと) とタイプすると，その式を OCaml で実行してくれる。(注意：なお、C-c C-e を押した初回だけは、emacs の一番下の行の mini buffer 行に「ocaml を起動するかどうか」聞かれる表示がでてくるので、単に、return (enter) キーを押して起動してほしい。2 回目からは聞かれない。もちろん、１度 emacs を抜けて、再起動したら、その初回はまた聞かれる。)

tuareg-mode の提供元は、`http://tuareg.forge.ocamlcore.org` である。

2.3 コンパイル

OCaml の対話モードでの実行速度は十分速いが、実行速度をさらに上げたいときは、コンパイラを使ってコンパイルすることもできる。

コンパイルするコマンドは ocamlc である。ネイティブコンパイラを使ってコンパイラする場合は ocamlopt である。これらの詳細については、参考文献を参照してほしい。

2.4 OCaml を自分の計算機で動かしたい人へ

OCaml の処理系は、Linux, MacOS などの Unix 系統の OS の上ではごく簡単に動くが、Windows でも動作させることができる。本実験は、情報科学類計算機室において遂行することが必要であるが、興味をもった人は、是非、自分の持っている PC 等にインストールして楽しんでほしい。(Debian 系 Linux であれば apt-get install ocaml だけでインストールできる可能性が高い。) 必要な情報は、OCaml の開発元のページ (http://caml.inria.fr) を見るか、あるいは、インターネットで、インストール方法のガイドが書いてあるページを探して欲しい。

3 変数と関数の定義

まず、コメント (注釈) と式をいくつか入れてみよう。

```
(* 例題ファイル 1 ex1.ml *)

(* This is a comment.
   ここに書いてあるものはコメントです。
   コメントは2行以上でも構いません。
 *)

(* 式のいろいろ; 式の終わりには ;; が必要 *)
1 + 2 * (3 / 2) -1 ;;

(* ;; を書くまでは、2行以上でも1つの式である *)
1 + 2 *
  (3 /

   2) -1 ;;

(* OCaml は、型については非常に厳しいルールをもっていて
   型が合わない式は実行せずにエラーとなる。たとえば、次の例は
   整数型と実数型をまぜたためエラーとなる *)
1 + 2.0 ;;  (* これはエラー *)
```

変数の定義には 2 種類の形式があり、変数を定義して使うのは、`let x = e1 in e2` という形の式を実行すればよい。

```
(* 変数は let で定義できる  *)
let x = 1 in x + 2 ;;

(* let は入れ子でもよい  *)
let x = (let y = 2 in y * 2) in
    let z = x + 3 in x + z ;;

(* let x = e1 in e2 の形の式でセットした x の値は
   「その場限り」で忘れられてしまう。
let y = x + 1 ;; (* これはエラー *)

(* let で導入する変数が同じ場合、内側が有効となる  *)
let x = 1 in
   let x = 2 in
     x ;;

(* ややこしい例  *)
let x = 1 in
   let x = (let x = x + 1 in x * 2) in
     x + 2 ;;
```

これらの場合すべてで、`let x = e1 in e2` が式であること、つまり、式が書ける場所に自由に出ていていることに注意されたい。

変数の定義のもう 1 つの形式は、in 以下がない let であり、`let x = e1 ;;` の形となる。これは、トップレベルでのみ使える形である。

```
(* in 以下を書かない let の例。
   let で宣言された変数はこの後ずっと有効である  *)
x ;;  (* これはエラー *)
let x = 1 ;;
x * 2 ;; (* in のない let で定義された変数の値は後でも使える *)
x + 5 ;; (* in のない let で定義された変数の値は後でも使える *)

(* in のない let はトップレベルでないと使えない *)
let x = (let y = 2) in 3 ;; (* これはエラー *)
```

次に、if-then-else (条文式) を使ってみよう。

```
(* if then else も使える  *)
let x = 1 in
  if x * 2 = x + x then "ok"
  else "ng" ;;

(* if then else も入れ子ネスト ()にできる  *)
let x = 1 in
  if x * 2 = x + x then
    if x + 3 > 3 then "ok"
    else "ng"
  else "ng" ;;

(* ただし、then と else で違う型の結果にはできない  *)
let x = 1 in
  if x * 2 > 1  then  3
  else 4.0 ;;

(* また、else 以下がないと else () が補われる *)
if 1 = 0 then () ;;

(* 従って、以下のものは型があわないのでエラー *)
if 1 = 0 then 5 ;;  (* 構文エラーでなく型エラー *)

(* ところで () というのは奇妙だが、これは unit 型の
  ただつの要素である 1 *)
let x = () ;;
```

関数の定義も let で行なう。これも、上記の 2 通りの形式のどちらでもよい。

```
(* 関数の定義 *)
let f x = x + 1 in
  f (f 3) ;;

(* 関数の別の定義, ラムダ式を使う *)
let f = fun x → x + 1 in
  f (f 3) ;;

(* in のない let で関数を定義する *)
let f x = x + 1 ;;
f 3 ;;

(* 関数だって、ネストできる *)
let f = fun x → fun y → x + y in
  (f 3) 4 ;;

(* 再帰関数は、今までの構文では定義できない *)
let g x =
   if x = 0 then 1 else x * (g (x − 1))
in
  g 3 ;;  (* これはエラー *)

(* 後で詳しく習うが、再帰関数の定義には、特別なキーワード rec が必要。
   次の関数 f は、階乗を計算する *)
let rec g x =
   if x = 0 then 1 else x * (g (x − 1))
in
  g 3 ;;
```

関数の定義の let を入れ子にしたり、関数の中で関数を定義してもよい。

```
(* 複数の関数の定義を続ける *)
let f x = x + 1 in
let g x = x + (f 3) in
    f (g 5) ;;

(* 関数を「後から」定義することはできない *)
let f x = g x + 1 in  (* エラー *)
let g x = x + 3 in
    f (g 5) ;;

(* 入れ子の関数定義 *)
let f x = let g y = x + y in g 5 in
    f 3 ;;

(* 入れ子の関数定義では、内側のものは外からは「見えない」 *)
let f x = let g y = x + y in g 5 in
    g 3 ;; (* エラー *)

(* 入れ子の関数定義では、外の変数は中から「見える」 *)
let f x = let g y = x + y in g 5 in
    f 3 ;;

(* 変数や関数のスコープは静的に決まる *)
let x = 3 in
let f y = x + y in
let x = 5 in
   f 7    (* 12 ではなく 10 が返る *)
```

複数の引数をもつ関数も同様に定義できる[*4]。

```
let f x y = x + y * 2 in
  f 10 20;;

let f x y = x + y * 2 in
  f (f 10 20) (f 30 40) ;;

let f x y z = if x=y then z else z + 1 in
  f 10 20 30 ;;
```

真理値についての補足：

[*4] 「複数の引数を持つ」というのは、厳密には、正しくないが、後に解説する。

```
(* ところで、真理値 (bool)を表すデータも使える *)
true ;;
1 = 2 ;;
let x = 1 in
  if x > 0 then "big" else "small" ;;
let x = 1 in
 (x > 3) || ((x < 2) && (x > 5)) ;;

(* = は、真理値 (bool)同士にも使える *)
true = (1 = 2);;
```

演習問題 1.

新しいプログラム言語を習うときに最初につまづくのは、表層構文に慣れてないことに起因する問題である。そこで、以下のことを調べよ。

- `a - b - c` といった式において、括弧付けがどのようにされるか調べなさい。
- 関数を引数に適用する式 (関数適用) は、OCaml では括弧をつけずに、`f x` のように書く。(なお、括弧をつけて `f(x)` と書いてもよい。) この (目に見えない) 関数適用と、四則演算 (足し算や引き算) の記号は、どちらが優先度が高いかを、`f 10 + 3` などの式で試しながら、調べなさい。
- `else` を省略した `if-then` 式を入れ子にすると、優先度が気になる。つまり、
- `if a then if b then c else d` といった式の else は前の if のものか後ろの if のものかわからない。どちらであるかを調べなさい。
  ただし、unit 型の値は `()` しかないため、2 つの値を使って区別することができない。この場合、OCaml のプリント式を使えばよい。たとえば、`print_string "abc"` という式は、文字列 abc を画面に印刷するが、返す値は `()` である。そこで、`if a then (print_string "abc") else ()` とすると、abc が印刷されるかどうかで、どちらのケースになったかが区別できる。
- (発展課題) OCaml の上記以外の記号の優先度について、実際のプログラムを走らせて調べなさい。