1 Objective Caml の基礎知識

§ 1.1 Objective Caml とは

系統的には Objective Caml は ML 系の関数型プログラミング言語である．

ML は 80 年代の初期に Edimburgh LCF という証明器のために開発された言語だが，最大の特徴はラムダ計算に基いた強い型システムである．特に注目したいのは，ML のプログラムの全ての型が自動的にチェックされ，そのチェックが成功すれば実行時に型エラーが起り得ないことだ．C のような Segmentation Fault は当然のこと，JAVA のような cast failure もない．

関数型言語は，Lisp や Smalltalk のように(自由変数を含んだ)関数が値として扱われ，引数として渡せることをいう．Haskell のような純粋関数型言語と違い ML では副作用のある命令は自由に使える．

Objective Caml はそういう ML を拡張して，オブジェクトシステムや省略可能引数を備えている．多くのライブラリも提供されている．

オンライン情報は次の URL にある．

- `http://www.ocaml.jp/`
- `http://caml.inria.fr/`
- `http://wwwfun.kurims.kyoto-u.ac.jp/soft/`

§1.2 入力ループとコンパイラの使い方

Objective Caml は C や Java にない入力ループを提供している，入力ループでは処理系と対話しながら，値や関数を定義したり，式を計算・実行したりしながらプログラムを作っていくことができる．プログラムを単位とするよりも，この方が容易に関数やアルゴリズムの動作確認ができる場合がある．

それとは別に，通常のバイトコードコンパイラとネイティヴコードコンパイラがある．

(a) 立ち上げと終了

入力ループは Unix シェル，emacs の中，ocamlbrowser の中という 3 つの環境で使える．

Unix シェルの場合は ocaml というコマンドを呼ぶだけでいい．

```
$ ocaml
        Objective Caml version 3.08.3

#
```

最後の「#」はプロンプトと言い，入力待ちを表している．この時点で既に入力ループに入っている．

入力ループから出たいとき，「#quit;;」と入力する．

```
# #quit;;
$
```

ocamlbrowser の中では File メニューの Shell を選ぶ．

emacs の中では「M-x run-caml」を実行する．

(b) 入力

入力ループで何かを入力する．たとえば「1+2;;<CR>」

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

「;;」 は Objective Caml に入力が終わったことを知らせる．それがないと，<CR>が 無視され，改行しても入力が続く．

「- : int = 3」が Objective Caml の出力した答である．「-」は，Objective Caml が入力された式の値を計算し，それをそのまま答えたことを表す．「int」は入力された式が整数型だったことを表す．「3」は実行の結果である．

(c) バッチ・コンパイラ

Objective Caml には二種類のバッチ・コンパイラがある．バイトコードを生成する ocamlc とネイティヴコードを生成する ocamlopt．

使い方は基本的に C のコンパイラと同じで，詳しい使い方は man ページを参照．

(d) 値と関数の定義

値を定義すると答えが少し違う．

```
# let x = 2*2;;
val x : int = 4
```

「let x」は，Objective Caml が名前の定義を行ったことを表す．x は定義された値の名前で，int はその型で，そして 4 は x の新しい値である．この定義の後 x を式の中に書くと，その値は 4 になる．

```
# x+1;;
- : int = 5
```

関数の定義は同じように行なう．

```
# let f x = x+1;;
val f : int -> int = <fun>
```

Objective Caml は関数型言語なので，関数は普通の値である．したがって，値の定義と同様に答は val で始まる．型は整数から整数への関数なので，「int -> int」．関数の値は表示できないので，値のところが「<fun>」 になっている．

(e) 変数の定義

値を定義の後で変更したい場合，定義のときにリファレンスを作らないといけない．

```
# let x = ref 1;;
val x : int ref = {contents=1}
```

変数の場合，型は単に int ではなく，int ref になる．値を取り出すのに「!」を使い，変更するのに「:=」を使う．

```
# !x;;
- : int = 1
# x := 5;;
- : unit = ()
# !x;;
- : int = 5
```

「x := 5」 は x を定義するのではなく，既に定義された整数の変数 x の中身を変える．その答の「- : unit = ()」 は特に意味がない．unit 型はただ一つの値「()」しか含まない型であり，Objective Caml では結果がない式の値はすべて「()」を返すように定義されている．

定義されていない変数に値を入れようとしたり，変数の型と違う型の値を入れようとしたりすると，エラーになる．

```
# y := 5;;
Unbound value y
# x := "hello";;
This expression has type int ref but is here used with type
string ref
```

エラーメッセージ「Unbound value y」 は y という名前の変数が存在しないことを，2 つ目のエラーメッセージは変数 x の型が int(整数)なのに，式"hello" の型が string(文字列)だったので，新しい値を入れることができなかったということを報告している．

どちらもよくある間違いで，特に「This expression has type ... but is here used with type ... 」という形のエラーメッセージは多くのプログラム

の間違いを指摘してくれる．Objective Caml のプログラミングでは，これを正しく理解することが重要だ．

(f) ファイルからプログラムを読み込む

いちいち入力ループから一つずつ式を入力していくのは面倒なので，あらかじめプログラムをファイルに保存しておき，それを一括して読み込むこともできる．そのとき，Objective Caml は，ファイルの内容があたかも入力ループで入力されたように動作する．

ファイル `test.ml` の中身は次の通りだとする．

```
let double x = x * 2;;          (* double は引数の 2 倍を計算する *)
let y = 10;;                    (* y を適当な値に *)
y + double y;;                  (* これで 3 倍だ！ *)
```

ファイルの中で `(*` と `*)` で囲まれた部分は註釈で，無視される．

`test.ml` を読み込む．

```
# #use "test.ml";;
val double : int -> int = <fun>
val y : int = 10
- : int = 30
```

このようにファイルからプログラムを読み込む場合は，入力ループで「`#use "ファイル名";;`」 のように入力すればよい．答は，読み込んだ入力によるものである．「`#use`」は結果を出さない．

§ 1.3 値，実行と制御構造

(a) 基本データ型

Objective Caml ではさまざまなデータが扱える．以下の式はデータをそのまま表す定数である．

整数 (`int`) `565` `-6327` など．
ただし，Objective Caml が負数を引き算と勘違いする場合は，
`(-23)` のように括弧を付けなければならない．

実数（float） 1． -5.3 2e4 など．

整数と区別するために必ず小数点「.」や「e」がなければならない．「e」は 10 のべき乗を表す．（2e4 $= 2 \times 10^4$）

文字（char） 'a' '\n' '\027' など．

最後の形は ASCII コードを三桁の十進数で表す．

文字列（string） "hello" "April 26, 1971" など．

文字と同様，文字コードも使える．

真偽値（bool） true（真）と false（偽）のみ．

単位（unit） () のみ．

意味のない型．何も返したくないときに使う．

(b) 定数式

定数だけでできた式．上記の基本データ型を元に，その組合せなどで値が作れる．組込の組合せ方法は三つある．

組（値$_1$, ... , 値$_n$）と書く．

値の型はばらばらでも構わない．前後の括弧は省略できる．その型は要素の型をすべて反映し，型$_1$ * ... * 型$_n$ になる．

```
# (1, 1.0, "one");;
- : int * float * string = (1, 1.0, "one")
```

配列（array） [| 値$_1$; ... ; 値$_n$ |] と書く．

値の型はすべて同じでなければならない．要素の型が t のとき，型は t array になる．

```
# [|1; 2; 3|];;
- : int array = [|1; 2; 3|]
```

リスト（list） [値$_1$; ... ; 値$_n$] と書く．

配列と同様，値の型はすべて同じである．要素の型が t のとき，リストの型は t list になる．

```
# [1; 2; 3];;
- : int list = [1; 2; 3]
```

ただし，配列と違って，リストの先頭にあとから自由に要素を加えることができる．

```
# 1 :: [2;3];;
- : int list = [1; 2; 3]
```

もっと正確に言うと，`[...]` を使った書き方は便宜的なもので，リストはすべて `::`（cons 構成子）と `[]`（空リスト）から作ることができる．

```
# 1 :: 2 :: 3 :: [];;
- : int list = [1; 2; 3]
```

(c) 計算式

関数や演算子でできた式．定数式も含む．

例

```
2 * (3 + 5)
4. +. sin 1.5
cos (1.3 +. 4.5)
sub array 3 5
```

四則演算については通常の結合規則が適用される．関数適用が最も優先順位が高い．したがって，2 行目では括弧を付けなくてよい．また，最後の行では，関数 `sub` を 3 つの引数 `array`, `3`, `5` に適用する．

Objective Caml は十分な数の演算子を用意している．以下にそれらをその型とともに示す．

数値演算子

```
+ - * / mod          int -> int -> int
+. -. *. /. **       float -> float -> float
-                    int -> int
-.                   float -> float
```

数値演算子は整数用と実数用が別々に定義されている．ただし，引数の型が実数だと入力時にわかれば，整数用のものは自動的に実数用に変換される．

比較演算子

```
=  <>  <  > <= >=    'a -> 'a -> bool
== !=                'a -> 'a -> bool
```

型の 'a はどの値でも使えるということを意味する．同じ型どうしの値ならどの値でも比較できる．2 行目は値のメモリ上の物理的な位置を比較する．

真偽演算子

```
&&  ||                bool -> bool -> bool
```

「式 1 && 式 2」は両方の式が真のときのみ真を返し，「式 1 || 式 2」は両方の式の少なくとも一方が真のときのみ真を返す．ただし，実際には次のように実行される．「式 1 && 式 2」は式 1 の結果が偽だったら式 2 を実行しない．「式 1 || 式 2」は式 1 の結果が真だったら式 2 を実行しない．

結合演算子

```
@            'a list -> 'a list -> 'a list
^            string -> string -> string
```

"^" は文字列にも使える．たとえば，"hello" ^ " world"は "hello world" になる．

配列演算子

```
配列.(添字)          'a array -> int -> 'a
配列.(添字) <- 値 'a array -> int -> 'a -> unit
```

「a.(i)」で配列 a の i 番目(最初の要素の番号を 0 とする)の要素を取り出す．「a.(i) <- 式」で配列 a の i 番目の要素を式の値に書き換える．

文字列演算子

```
文字列.[添字]          string -> int -> char
文字列.[添字] <- 値 string -> int -> char -> unit
```

配列演算子と同じ．

(d) 定義

既に定義をどのように書くか説明したが，もっと形式的に説明すると，次のようになる．

`let` 定数名 `=` 計算式
`let` 関数名 引数$_1$ 引数$_2$ ... 引数$_n$ `=` 計算式
`let rec` 関数名 引数$_1$ 引数$_2$ ... 引数$_n$ `=` 計算式

定数の場合，式の結果の型がそのまま値や変数の型になるが，関数だと引数の型を含んだ関数型が付けられる．上記の定義で引数の型が 型$_1$ ... 型$_n$ で結果の型が 型$_0$ だとすると，関数の型はこうなる

型$_1$ `->` 型$_2$ `->` ... `->` 型$_n$ `->` 型$_0$

数学ではこのような，矢印がたくさん入った型はあまり使わないが，要するに (型$_1$ × 型$_2$ × ... × 型$_n$) → 型$_0$ と同じようなものだと思えばよい．

`let` と `let rec` の違いは，前者では新しく定義された名前が右辺の計算式からは見えないが，`let rec` なら見える点にある．再帰的な定義をするときは，必ず `let rec` を使わなければならない．

`let`，`let rec` とも，同時に複数の定義ができる．そのとき，`and` を使う．

```
# let x = 3 and y = 2*2;;
val x : int = 3
val y : int = 4
```

新しく定義された名前は `let rec` でなければ各々の定義の後でしか見えない．すなわち，定義中の名前が計算式の中に出る場合，それ以前の定義が使われる．

```
# let x = 1 and y = x+2;;
val x : int = 1
val y : int = 5
```

`let rec` だと，すべての計算式からすべての名前が見える．

```
# let rec up x = down (x+10)
  and down x = if x <= 15 then x else up (x/3);;
val up : int -> int = <fun>
val down : int -> int = <fun>
```

(e) 制御

制御構造には選択とループがある．

選択

```
if 計算式1 then 計算式2 else 計算式3
if 計算式1 then 計算式2
```

計算式$_1$ を実行し，その結果が `true` ならば 計算式$_2$ を実行し，その結果を返す．そうでなければ 計算式$_3$ を実行し，その結果を返す．計算式$_3$ がない場合，計算式$_2$ は `()` を返さなければならない．

条件付きループ

```
while 計算式1 do 計算式2 done
```

計算式$_1$ が `true` を返す限り，計算式$_2$ を実行する．計算式$_2$ の返す値と関係なく，終わると `()` を返す．

添字付きループ

```
for 変数名 = 計算式1 to 計算式2 do 計算式3 done
for 変数名 = 計算式1 downto 計算式2 do 計算式3 done
```

計算式$_1$ の結果から 計算式$_2$ の結果までの整数の値を変数に入れ，計算式$_3$ を実行する．`to` のときはループごとに変数の値を一つ増やし， `downto` のときは減らす．

上記の三種類に加えて，(g)で説明するパターンマッチングでも制御が行える．

(f) ブロック構造

一つの計算式の中に定義をしたり，連続して複数の計算式を実行したいとき，ブロックを作らなければならない．各定義の後に「`in`」を書いて，計算式の間に「`;`」を書くと全体が一つのブロックになり，一つの計算式とみなされる．結果は最後の式の結果になる．

さらブロックを明確に区切りたい場合には， 括弧または begin と end で囲むことができる．

```
# let maxabs x y =
    let x' = if x < 0 then -x else x
    and y' = if y < 0 then -y else y in
    if x' < y' then y' else x'
val maxabs : int -> int -> int = <fun>
# let minmax array =
    let max = ref array.(0)
    and min = ref array.(0) in
    for i = 1 to Array.length array - 1 do
      if array.(i) > !max then max := array.(i);
      if array.(i) < !min then min := array.(i);
    done;
    (!min,!max) ;;
val minmax : 'a array -> 'a * 'a = <fun>
```

maxabs は x と y の絶対値を計算し，大きい方を返す．minmax は配列の中の最大値と最小値を計算し，その組を返す．

(g) パターン・マッチング

上記の minmax は組を返すが，組から値を出すのにはパターン・マッチングを使う．

```
# let (mn,mx) = minmax [|2;1;4;3|];;
val mn : int = 1
val mx : int = 4
```

let にこういう便利な機能が付いている．実は，let による名前の定義はもっと自由におこなうことができる．

```
let パターン = 計算式
```

パターンは計算式と同じ書き方をするが，

(i) 関数や命令，実行されるようなものは入らない．
(ii) 名前は右辺の計算式によって定義される．もし，その名前が既に定義されていた場合は，元の値は隠され，以後その名前は新しい値を表す．

値を含んだパターン・マッチングの例

```
# let (1,x) = (1,3);;
val x : int = 3
```

パターンと値が合わないときは，エラーが起こる．

```
# let (1,x) = (2,3);;
Uncaught exception: Match_failure ("", 5, 8).
```

パターンと match 文を組み合わせて使うと，場合分けができる．

```
# match (2,3) with (1,x) -> x | (2,x) -> x + 2;;
- : int = 5
```

match 文で場合わけするには，次のように，パターンにマッチさせたい値とパターンのリストと各場合に応じて実行される計算式の列を書く．match 文は，与えられた値がパターンとマッチするかどうかパターンをリストの前から順番にチェックしていく．最初にマッチした時点で，そのパターンによって実行されるべき計算式の列を計算する．

```
match 計算式 of
  パターン ->
    計算式
| パターン ->
    計算式
| ...
```

ここまででは，パターンとして使えるのは，変数，単純な値，そして組のパターンだけだが，新しいデータ型を定義して，それらをパターンとして使うこともできる．

(h) 値としての関数

定義文を使わなくても，新しい関数を作れる．

```
# fun x -> x+1;;
- : int -> int = <fun>
```

要するに，今まで let を使った関数定義はこの fun を使った値定義の省略でしかなかった．

```
# let succ = fun x -> x+1 ;;
succ : int -> int = <fun>
```

一般的には次の構文がある．

fun 引数$_1$ 引数$_2$ ... 引数$_n$ -> 計算式

パターン・マッチングと組み合わせた function 構文もある．

```
function パターン ->  計算式
       | パターン ->  計算式
       | ...
```

(i) 新しいデータ型の定義

1. 略称

データ構造を新しく定義するのではなく，既に定義されたデータ構造に名前を付ける．C 言語の typedef に当る．

```
# type complex = float * float;;
type complex = float * float
```

Objective Caml は，型を略称に自動的に置き換えたりはしないので，略称で型を表示してほしいときは人間が教えてやらないといけない．

```
# let add_complex ((x,y) : complex) ((x',y') : complex)
    : complex = (x+.x',y+.y');;
val add_complex : complex -> complex -> complex = <fun>
# add_complex (1.,2.) (3.,4.);;
- : complex = (4.0, 6.0)
```

2. 直和型

名前の付けられたいろいろな値のうち，いずれか一つの値を持つようなデータを表す型である．

たとえば，トランプを考えよう．まずスーツを定義する．

```
# type color = Diamond | Heart | Spike | Clove;;
type color = Diamond | Heart | Spike | Clove
```

直和型 color は，４つの名前「Diamond」,「Heart」,「Spike」,「Clove」でそれぞれ表される値のどれか一つを表す．これらの名前のことを構成子という．

```
# Heart;;
- : color = Heart
```

さらにカードを定義する．

```
# type card = Normal of color * int | Joker;;
type card = Normal of color * int | Joker
```

ここで「Normal」は引数をとる構成子である．of の後に書かれているのが引数の型である．

```
# Normal (Clove,11);;
- : card = Normal (Clove, 11)
# Joker;;
- : card = Joker
```

直和型 card は，２つの構成子「Normal」,「Joker」 のいずれかで表される値であり，特に構成子が「Normal」の場合は，その値として直和型 color の値(スーツ)と整数(トランプに書いてある数字)の組を持つ．

リストも直和型として定義されている．:: と [] は特別な記号なので，新しくリストを定義しようとする場合は，別の名前を使う．

```
# type 'a list = Cons of 'a * 'a list | Nil;;
type 'a list = Cons of 'a * 'a list | Nil
```

上の例と違い，リストは型自身が引数('a)をとる．これはどんな型の値のリストも許されることを意味する．たとえば整数のリストは引数 'a に int を与えたリストと考えることができる．Cons の 2 番目の引数もリストであることに注意しよう．リストは再帰的な型で，Cons がいくつかあって，最後に Nil で終わるという形になる．

```
# Cons(1, Cons(2, Cons (3, Nil)));;
- : int list = Cons (1, Cons (2, Cons (3, Nil)))
```

直和型はパターンマッチングにも使える．たとえば，次にあげるのはリストの長さを計算する関数である．

```
# let rec length = function
      Nil -> 0
    | Cons (a,l) -> 1 + length l ;;
val length : 'a list -> int = <fun>
```

ここで function 構文を使ったが，「let rec length l = match l with」の略称でしかない．

リストでは型引数を 'a と書いたが，もしも複数の引数が必要であれば，type ('a,'b,'c) t = ... という形でも書ける．直和型だけでなく，型引数は略称型や次の直積型でも使える．

3. 直積型

組の各要素を名前を付けて識別すると便利である．そのような値を表す型が直積型である．

複素数の例を直積型で書くとこうなる．

```
# type complex = {re : float; im : float};;
type complex = {
  re : float; im : float;
}
# let i = {re=0.0;im=1.0};;
val i : complex = {re=0.0; im=1.0}
# i.im;;
- : float = 1.0
# let add_complex a b =
    {re = a.re + b.re; im = a.im + b.im};;
val add_complex : complex -> complex -> complex = <fun>
# add_complex i i;;
- : complex = {re=0.0; im=2.0}
```

直積型では，指定した名前の要素を可変にできる．

```
# type person = {name : string; mutable age : int};;
type person defined.
# let me = {name = "Gariko"; age = 25};;
val me : person = {name="Gariko"; age=25}
```

```
# me.age <- me.age + 1;;
- : unit = ()
# me;;
- : person = {name="Gariko"; age=26}
```

可変な直積型の値は，変数と違い，後から定義された関数の中に使ったり，渡したりできる．

```
# let birthday pers = pers.age <- pers.age + 1;;
val birthday : person -> unit = <fun>
# birthday me;;
- : unit = ()
# me;;
- : person = {name="Gariko"; age=27}
```

ちなみに，変数に使われる `ref` 型は，可変な直積型でしかない．

```
type 'a ref = { mutable contents : 'a }
```