2021年度「プログラミング言語」配布資料 (3)

五十嵐 淳

2022年10月02日

1 OCaml

OCaml (旧名称: Objective Caml) は INRIA というフランスの国立の計算機科学の研究所で設計・開発された言語である．歴史的には，証明支援系と呼ばれる，計算機によって数学的証明を行うためのソフトウェアを操作するための言語の研究から発展してきたもので，当初は ML (Meta Language) と呼ばれていたが，研究が進展し，また方言が派生していく過程で OCaml ができあがった．Standard ML や F# などの方言をまとめて ML と呼ぶことが多い．(興味があったら Caml の歴史, Standard ML の歴史 などを見よ．)

ML は関数型プログラミング (functional programming) と呼ばれるプログラミングスタイルをサポートしている．ML は核となる部分が小さくシンプルであるため，プログラミング初心者向けの教育用に適した言語である．と同時に，大規模なアプリケーション開発のためのモジュール化機能が充実している．ML の核言語は型付きラムダ計算と呼ばれる，形式的な計算モデルに基づいている．このことは，言語仕様を形式的に (数学的な厳密な概念を用いて) 定義し，その性質を厳密に「証明」することを可能にしている．実際，Standard ML という言語仕様においては，(コンパイラの受理する) 正しいプログラムは決して未定義の動作をおこさない，といった性質が証明されている．

OCaml は Standard ML とは構文が異なるが，ほとんどの機能を共有している．また，OCaml では Standard ML には見られない，独自の拡張が多く施されており，関数型プログラミングだけでなく，オブジェクト指向プログラミングもサポートされている．また効率的なコードを生成する優れたコンパイラが開発されている．

1.1 OCaml 爆速入門 (その 1)

OCaml の概要をプログラム例を交えて紹介する．二分探索木がプログラムできるための最短ルートを通るため，より詳しいことはトップページで紹介している他の資料にもあたってもらいたい．

1.1.1 OCaml は電卓として使える

OCaml には入力を即座に実行して結果を表示する対話的処理系が用意されていて，インタラクティブにプログラムの実行を進めることができる[1]．OCaml では入力された式を値に評価することでプログラムの実行が進むので，対話的処理系は電卓のようなものである．

[1] Java や C のように，(プログラムが書かれた) ファイルをコンパイルするバッチコンパイラもある．対話的処理系はプログラムの実行結果を与えるという意味ではインタプリタだが，内部的にはプログラムを低水準コードにコンパイルしている．

```
# 1 + 2 * 3;;
- : int = 7
```

が処理系による入力プロンプトの記号で以降が実際の入力である．;; は電卓の=キーのようなもので，式の入力の終わりを告げるための記号である (Enter キーを押す必要がある)．次の行に入力された数式の計算結果 (値と呼ぶ) 7 が表示されている．int は入力された式 1 + 2 * 3 の型を示している．

```
# 5 < 8;;
- : bool = true
# 5 = 2 || 9 > -5;;
- : bool = true
```

このように比較演算の結果は true, false という bool 型の定数で表される．等号は == ではなく= である．また，不等号は <> である[2]．基本的なデータとしては，他にも浮動小数点数の float 型，文字列の string 型などがあるがここでは紹介しない．

1.1.2 OCaml はメモリ機能がついた電卓として使える

OCaml では let 宣言を使って式の計算結果に名前をつけることができる．OCaml ではこの名前のことを変数と呼ぶ．

```
# let x = 1 + 2 * 3;;
val x : int = 7
# x - 10;;
- : int = -3
```

val は値を表す名前が宣言されたことを表している．(型を定義すれば val ではなく type になったりする．)

これは Java であれば，変数宣言 (と初期化)

```
int x = 1 + 2 * 3;
```

に相当すると考えられるが，いくつか重要な違いがある．

- 式の計算結果がまずあって，それに名前を付ける—値を格納するためのメモリ領域に名前を付けているわけではない—ので，int x; のように変数の宣言だけをする，ということはできない．
- 変数の型を指定する必要はなく，処理系が右辺を見て変数の型を推論してくれる．これは ML の大きな特徴のひとつである**型推論** である．プログラマが型を指定したければ変数名と= の間に「:〈型〉」と書く．

```
# let x : int = 1 + 2 * 3;;
val x : int = 7
```

- 変数への (再) 代入はできない．ただし，同じ名前を再利用することはできる．

```
# let x = 4 < 5;;
val x : bool = true
```

「これは代入ではないのか？」と思うかもしれない．それに対する説明はもうすこし後で行う．

[2]比較演算子として == や != もあるのだが，ここでは説明しない．

1.1.3 OCaml は関数電卓として使える

OCaml の主要なプログラム構成要素は関数である．関数も let を使って定義する．関数の定義の構文は，

let〈関数名〉(〈パラメータ 1〉, ...,〈パラメータ n〉) =〈式〉

となる．〈パラメータ 1〉 ～ 〈パラメータ n〉の値を受け取って，〈式〉を計算してその値を返す．特に return などを使う必要はなく計算式のみ書けばよい．例を見てみよう．

```
# let average(x, y) = (x + y) / 2;;
val average : int * int -> int = <fun>
# average(5, 7);;
- : int = 6
# average(10, 3);;
- : int = 6
```

関数の型は〈引数 1 の型〉* ... *〈引数 n の型〉->〈返値型〉と表現される．関数についても，パラメータや返値の型を書く必要はなく，**型推論 (type inference)** が行われて関数 average は 2 つの整数を取って，整数を返すことがわかる．数式と違って計算結果 (=の右側) として単に<fun> と表示されているが，これは average が何らかの関数であることを示している．= 以下の式を関数の本体 (body) という．

パラメータの数がひとつの時は括弧を省略することができ，そして，ふつうは省略する (ただし，関数名との間には空白が必要である)．

```
# let double n = n + n;;
val double : int -> int = <fun>
# double 256;;
- : int = 512
```

1.1.4 条件分岐

条件によって異なる値を計算したい時には if 式を使う．if 式は

if〈条件〉then〈式 1〉else〈式 2〉

という形で，〈条件〉(これは bool 型の式でなければならない) が true ならば，〈式 1〉の値が，false ならば〈式 2〉の値が if 式全体の値になる，というものである．例えば，絶対値を計算する関数 abs は以下のように書ける．(対話的処理系への入力は，この abs の定義のように複数行にまたがってもよい．)

```
# let abs n =
    if n < 0 then -n else n;;
val abs : int -> int = <fun>
# abs(-100);;
- : int = 100
# abs 20;;
- : int = 20
```

-100 のまわりには括弧が必要である．これがないと，OCaml は abs -100 を「abs 引く 100」と解釈してしまい，「関数から引き算をするとは？」といって型についてのコンパイルエラーが出てくる．

```
# abs -100;;
  ^^^
Error: This expression has type int -> int
       but an expression was expected of type int
```

1.1.5 再帰関数

再帰関数を宣言する時には，let ではなく let rec と書く．(rec は「再帰」を表す recursion からとっている．) 例えば，階乗 (factorial) を計算する関数は

```
# let rec fact n =
    if n = 1 then 1 else n * fact (n - 1);;
val fact : int -> int = <fun>
# fact 5;;
- : int = 120
```

のように定義できる．

ここでも n - 1 のまわりの括弧が必要である．これがないと，OCaml はこの式を「(fact n) - 1」と解釈してしまう．(fact n-1 とマイナスの前後の空白を詰めて書いても無駄である．) OCaml では関数の引数を括弧で囲むが必要なく，関数呼び出し (fact n) が算術演算子よりも強く結合することになっている．

この fact 5 の実行過程を式の変形で示すと以下のようになる．

```
   fact 5
→ if 5 = 1 then 1 else 5 * fact (5 - 1)
→ 5 * fact (5 - 1)
→ 5 * fact 4
→ 5 * (if 4 = 1 then 1 else 4 * fact (4 - 1))
→... → 5 * (4 * fact 3)
→... → 5 * (4 * (3 * (2 * fact 1)))
→ 5 * (4 * (3 * (2 * (if 1 = 1 then 1 else ...))))
→ 5 * (4 * (3 * (2 * 1)))
→ 5 * (4 * (3 * 2))
→ 5 * (4 * 6)
→→ 120
```

fact に 0 (や負数) を与えた場合は，この計算は

```
   fact 0
→ 0 * fact (-1)
→ 0 * ((-1) * fact (-2))
→
```

となって止まらず，ついには (式が大きくなり過ぎて) メモリが足りなくなってスタック・オーバーフローというエラーになってしまう．

```
# fact 0;;
Stack overflow during evaluation (looping recursion?).
```

(スタック・オーバーフローについてはそのうち詳しく説明する．)

1.1.6 変数の有効範囲と静的スコーピング

関数の定義中で使ってよい変数はパラメータだけではない．それまでに宣言された変数を使うことができる．以下は，与えられた半径から円の面積を計算する関数 `circle_area` の定義である．(`*.` は浮動小数点数型 `float` のための乗算演算子である．)

```
# let pi = 3.14;;
val pi : float = 3.14
# let circle_area radius = pi *. radius *. radius;;
val circle_area : float -> float = <fun>
# circle_area 4.0;;
- : float = 50.24
```

`circle_area` の中で，パラメータ以外の変数 `pi` が使われている．

さて，上で，同じ変数名の定義が複数あってもそれは代入ではない，といったが，それは以下のような例から確認できる．

```
# let pi = 3.00;;
val pi : float = 3.
# circle_area 4.0;;
- : float = 50.24
```

関数定義中の `pi` の使用は，本体を計算する時点での `pi` の値をとってくるのではなく，**関数が定義された時点での** `pi` の値を使うことになる．一方で，以下のような計算をすると再定義された `pi` の値 `3.0` が引数になるし，

```
# circle_area pi;;
- : float = float = 28.259999999999998
```

新しい `pi` を別の関数定義で使えば，円周率が 3.0 となる面積計算になる．

```
# let circle_area2 radius = pi *. radius *. radius;;
val circle_area2 : float -> float = <fun>
# circle_area2 4.0;;
- : float = 48.
```

(`48.` の最後の小数点は整数定数の表記 `48` と区別するために表示される．`48.0` と同じ扱いである．)

このように OCaml では，変数の使用に対応する定義は，使用箇所から上に辿って一番近いものになる．このような，プログラムテキスト中での字面上の関係を使って使用と定義の対応を決める方式を，静的スコーピング (static scoping または lexical scoping)，という．

1.1.7 let 式による局所定義

let を式の中で使うことによって，式の計算中，値に一時的に (局所的に) 名前をつけることができる．これが「let 式」である．let 式の構文は

let〈変数〉=〈式 1〉in〈式 2〉

であり，直観的には「〈式 2〉の計算中 (のみ)，〈変数〉を〈式 1〉(の値) とする」という意味で，〈式 1〉を計算し，その値に〈変数〉という名前をつけ，〈式 2〉の値を計算する．〈式 2〉を let 式の本体ということがある．

```
# let n = 5 * 2 in n + n;;
- : int = 20
```

この〈変数〉の有効範囲は〈式 2〉だけで，〈式 2〉の外側では使用することができない．別の言い方をすると，n + n を計算する間だけの一時的な変数である．そのため，let 式の計算後に同じ名前を参照しても (既に n が定義されていない限り) エラーとなる．

```
# n;;
Characters 0-1:
  n;;
  ^
Error: Unbound value n
```

let 式もただの式の一種なので，式が書けるところでは自由に使うことができる．典型的には，〈式 2〉に let を使うことで複数の変数を順に定義していくように書ける．

```
# let x1 = 1 + 1 in
  let x2 = x1 + x1 in
  let x3 = x2 + x2 in
  x3 + x3;;
- : int = 16
```

x1 を定義している let の本体は let x2 以下全部である．

また，以下のように，関数定義の本体で let を使うこともできるし，さらには，関数を局所的に定義することもできる．

```
# let cylinder_vol(r, h) =
    let bottom = circle_area r in (* 底面積の計算をする局所関数 *)
    bottom *. h;;
val cylinder_vol : float * float -> float = <fun>
# let cube n = n * n * n in
  cube 1 + cube 2 + cube 3;;
- : int = 36
# cube;;
Characters 0-4:
  cube;;
```

```
  ^^^^
Error: Unbound value cube
```

二番目の例では，$1^3 + 2^3 + 3^3$ を計算するために，まず (局所的に)3 乗関数を定義している．この式の外側では `cube` は使用できない．(途中の `(* ... *)` は OCaml のコメント記法である．)

1.1.7.1 練習問題

- 与えられた正整数 n に対し $(1^2 + 2^2 + \cdots + n^2)$ を計算する再帰関数 `squaresum : int -> int` を定義せよ．引数 n がゼロや負である場合の動作はどう定義してもよい．
- 与えられた正整数 n に対し，n 番目のフィボナッチ数を計算する再帰関数 `fib : int -> int` を定義せよ．引数 n がゼロや負である場合の動作はどう定義してもよい．
- ユークリッドの互除法を使ってふたつの正整数 n, m の最大公約数を計算する再帰関数 `gcd : int * int -> int` を定義せよ．引数がゼロや負である場合の動作はどう定義してもよい．(ヒント: x を y で割った剰余は $x \mod y$ で表す．)

:::spoiler 解答例

```
let rec squaresum n =
  if n < 1 then 0
  else n * n + squaresum (n-1)

let rec fib n =
  if n <= 2 then 1
  else fib (n-1) + fib (n-2)

let rec gcd(n, m) =
  if n < m then gcd(m, n)
  else
        let n' = n mod m in
        if n' = 0 then m else gcd(m, n')
```

:::

1.2 OCaml 爆速入門 (その 2) ―データ構造の基本

OCaml では，`type` 宣言を使って様々な複合的なデータ型を定義することができる[3]．

1.2.1 レコード

レコード (record) は，いくつかの値を (それぞれに名前をつけて) まとめたような値である．例えば二次元座標 (座標は整数値) を表すレコード型 `point` は以下のように定義できる．

```
# type point = {x: int; y: int; };;
type point = { x : int; y : int; }
```

[3]組 (tuple) と呼ばれる，要素を並べて作るデータについては基本的なものだが省略する．

x, y をレコードのフィールド (field) 名と呼ぶ．レコード型の値を作るには，

```
# let origin = {x = 0; y = 0 };;
val origin : point = {x = 0; y = 0}
```

などとする．(OCaml の文法は，最後のフィールドに関してやや寛容で，型定義にしてもレコードを作るにしても，最後のセミコロンはあってもなくてもよい．表示の仕方も型定義とレコード値でぶれているのが面白い．)

レコードから各フィールドの値を取り出すには，ふたつの方法がある．ひとつは**ドット記法 (dot notation)** と呼ばれる方法で，<レコード式>.<フィールド名> と書く．

```
# origin.x;;
- : int = 0
# let middle(p1, p2) =
   {x = average(p1.x, p2.x); y = average(p1.y, p2.y) };;
val middle : point * point -> point = <fun>
# middle(origin, {x=4; y=5});;
- : point = {x = 2; y = 2}
```

もうひとつは **パターンマッチ (pattern match)** と呼ばれる機構を用いる方法である．パターンとは，構造を持つ値の形の骨組と情報を取りだしたい部分を指定する記法である．let などの変数を宣言する機構と組み合わせて使うことができる．

例:

```
# let {x = ox; y = oy} = origin;;
val ox : int = 0
val oy : int = 0
```

{x = ox; y = oy} がパターンで，ox, oy は取り出したフィールドの値につける名前である．origin の値の形と照合を行い変数に対応する部分が取り出されている．先程の middle は以下のように書くこともできる (これだと余りパターンマッチを使う恩恵がないが).

```
# let middle(p1, p2) =
    let {x = p1x; y = p1y} = p1 in
    let {x = p2x; y = p2y} = p2 in
    {x = average(p1x, p2x); y = average(p1y, p2y) };;
val middle : point * point -> point = <fun>
```

また，関数のパラメータに直接パターンを書くこともでき，

```
# let middle({x = p1x; y = p1y;}, {x = p2x; y = p2y;}) =
    {x = average(p1x, p2x); y = average(p1y, p2y) };;
val middle : point * point -> point = <fun>
```

と書いてもよい．フィールドが沢山あるレコードから，一部のフィールドの値を取り出したい場合には，不要なフィールドについては何も書かなくてよく，また，フィールド値の名前としてフィールド名そのものを使う場合には= 以下を省略することができるので point の x 座標を取り出す関数は

```
# let get_x {x} = x;;    (* equivalent to let get_x {x=x} = x;; *)
```

のように書ける．

1.2.1.1 練習問題

- ふたつの点を引数として，その二点を直径とする円の面積を返す関数 `area : point * point -> float` を定義せよ．整数 `n` は `float_of_int` という関数を呼ぶことで `float` に変換することができる．平方根は (`float` を引数とする) `sqrt` 関数で計算できる．
- 有理数を表現する型 `rational` を以下のように定義する．

```
type rational = {
  num : int; (* 分子 *)
  den : int; (* 分母 *)
};;
```

例えば「5 分の 4」は `{num=4; den=5}` と書くことになる．ふたつの有理数の和を求める関数 `sum : rational * rational -> rational` を定義せよ．(約分もせよ．)

:::spoiler 解答例

```
let sum r1 r2 =
  let r3_num = r1.num * r2.den + r2.num * r1.den in
  let r3_den = r1.den * r2.den in
  let g = gcd(r3_num, r3_den) in
  {num = r3_num / g; den = r3_den / g}
```

:::

1.2.2 ヴァリアント

ヴァリアント (variant) は，異なるデータを混ぜて使うための仕組みである．これは応用範囲が非常に広く，OCaml でデータを定義する，といったら 9 割 (当社比) がこの仕組みを使っている．

まず，一番単純な場合として，いくつか自前の定数を作ってそれらを混ぜて使うことを考える．ふりかけを表すデータ型 `furikake` を定義してみよう．ここではふりかけの種類として「さけ」「かつお」「のり」の 3 種類を考える．

```
# type furikake = Shake | Katsuo | Nori;;
type furikake = Shake | Katsuo | Nori;;
```

この定義により，型 `furikake` とそれに属する定数 `Shake`, `Katsuo`, `Nori` が使えるようになる．縦棒 `|` の直観的な意味は「または」である．

```
# Shake;;
- : furikake = Shake
```

OCaml ではこれらの定数をデータ・コンストラクタ (data constructor)，または単にコンストラクタと呼ぶ．OCaml ではコンストラクタの名前は英大文字で始める必要がある．

一般に，この furikake のような，定数を列挙することで構成されるデータ型を**列挙 (enum) 型** と呼び，Java や C では列挙型を定義するための専用構文がある．

さて，ヴァリアントについての処理の基本は場合分けである．

例えば，与えられたふりかけが菜食主義者でも食べられるかを判定する isVeggie 関数を考えてみよう．この関数は，ふりかけの種類によって 3 通りに場合分けを行うことになる．ヴァリアントについて条件分岐を行うには match 式と呼ばれる式を使う．以下が isVeggie 関数の定義である．

```
# let isVeggie f =
    match f with
      Shake -> false
    | Katsuo -> false
    | Nori -> true
    ;;
val isVeggie : furikake -> bool = <fun>
# isVeggie Shake;;
- : bool = false
```

match 式の構文は

match〈式 0〉with〈パターン 1〉->〈式 1〉| ... | 〈パターン n〉->〈式 n〉

で，〈式 0〉の値を〈パターン 1〉から順に照合していき，i 番目でマッチしたら (形があったら)，〈式 i〉を計算しその値が match 式全体の値となる．

今回の例では〈パターン i〉は単にコンストラクタの名前となっているが，もっと複雑なヴァリアントの場合には，パターンも複雑なものがでてくる．match 式には，パターンの exhaustiveness check といって，〈パターン 1〉から〈パターン n〉が〈式 0〉の値としてありうる全ての場合を尽くしているかをチェックしてくれる機能がある．上の isVeggie で f の型は furikake なので，3 つのパターンを用意しないとコンパイラが警告を出す．下のように，例えば Katsuo の場合を書かずに定義すると，定義はできるものの，「マッチ対象の値 (つまり f) が Katsuo だとマッチしないよ」という警告が出る．

```
# let isVeggie f =
    match f with
      Shake -> false
    | Nori -> true
    ;;
Characters 21-73:
  ....match f with
        Shake -> false
      | Nori -> true
Warning 8: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
Katsuo
val isVeggie : furikake -> bool = <fun>
```

また，パターンには，整数定数を書くこともでき，Java の switch 文のような処理も記述できる．例えば n が 20 以下の素数ならば true となり，その他の場合 false になるような式は

```
match n with
    2 -> true
  | 3 -> true
  | 5 -> true
  | 7 -> true
  | 11 -> true
  | 13 -> true
  | 17 -> true
  | 19 -> true
  | _ -> false
```

と書くことができる．最後の行の _ は**ワイルドカード・パターン (wildcard pattern)** と呼ばれ，何にでもマッチする．ワイルドカード・パターンは整数以外でも使えるので，isVeggie は以下のように書いてもよい．

```
# let isVeggie f =
    match f with
      Nori -> true
    | _ -> false
    ;;
```

(ワイルドカード・パターンの行を先に書くと何でも false を返すことになってしまう．上のパターンから順に照合することに注意．)

また，複数のパターンで同じ式を計算する場合には，パターンをまとめて記述することもできる．

```
match n with
    2 | 3 | 5 | 7 | 11 | 13 | 17 | 19 -> true
  | _ -> false
```

2 | 3 ... | 19 は **or パターン (or pattern)** と呼ばれ一般的には，<パターン 1> | <パターン 2> | ... | <パターン n> という形をして「いずれかのパターンにマッチするならば」という条件を表現することができる．isVeggie は以下のようにも書ける．

```
# let isVeggie f =
    match f with
      Shake | Katsuo -> false
    | Nori -> true
    ;;
```

列挙型は，ヴァリアントの使い方としては特殊である．一般的に，ヴァリアントのコンストラクタは値を「運ぶ」ことができる．例えば，一品料理の型を考えてみよう．一品料理は「トンカツ」「味噌汁」「ふりかけごはん」のいずれかとするが，味噌の種類は「赤」「白」「あわせ」，味噌汁の具は「わかめ」「とうふ」「大根」のうちからひとつが選べ，ごはんのふりかけも選べるようにする．このような型の定義として，トンカツと 9 種類の味噌汁，3 種類のふりかけごはんの計 13 種類のコンストラクタから成る列挙型も考えられるが，ここでは 3

種類のコンストラクタ PorkCutlet, Soup, Rice を用意し，Soup には味噌と具の種類の情報を，Rice にはふりかけの情報を付与できるようにする．

```
# type miso = Aka | Shiro | Awase;;
type miso = Aka | Shiro | Awase
# type gu = Wakame | Tofu | Radish;;
# type dish = PorkCutlet | Soup of {m: miso; g: gu} | Rice of furikake;;
type dish = PorkCutlet | Soup of { m : miso; g : gu; } | Rice of furikake
```

of 以下には型の名前を書いて (int や，この例のように，レコードやヴァリアント型でもよい[4])，各コンストラクタに何の情報が付与されるかを指定する．具体的な一品料理は，コンストラクタの後に (関数呼出しの引数のように) 付帯情報を書く．

```
PorkCutlet;;                              (* トンカツ *)
- : dish = PorkCutlet
# Soup {m = Aka; g = Tofu};;              (* 豆腐赤だし *)
- : dish = Soup {m = Aka; g = Tofu};;
# Rice Shake;;                            (* 鮭ふりかけごはん *)
- : dish = Rice Shake
```

どの式も型は dish になっていることに注意すること．

引数を取るコンストラクタを単体で使うことはできない．

```
Rice;;
Characters 0-4:
  Rice;;
  ^^^^
Error: The constructor Rice expects 1 argument(s),
       but is applied here to 0 argument(s)
```

ヴァリアント型は，このようなデータ付コンストラクタの集まりであるが，見方を変えると，異なる種類の値 (ふりかけや，味噌と具のレコード) を混ぜて使うための仕組みであると考えられる．

さて，与えられた dish が固形かどうかを判定する関数 isSolid を書いてみよう．これは，isVeggie と同じように match を使って記述するが，引数を取るコンストラクタについては引数につける名前を同時に指定する．

```
# let isSolid d =
    match d with
      PorkCutlet -> true
    | Soup m_and_g -> false
    | Rice f -> true
    ;;
val isSolid : dish -> bool = <fun>
```

この例では f などの付帯情報を表す変数は全く使っていない (ので，m_and_g や f の代わりにワイルドカード

[4]複数の情報を持たせたい場合，レコードではなく (記述量が少ない) 組を使うことが多いのだが，Java との対応関係が見て取りやすいレコードを使っている．

_ を書いてもよい．) が，対応する，->の右側の式で使うことができる．例として，料理の値段を計算する関数を定義してみよう．まず，以下が値段表である．

一品料理	値段 (円)	備考
トンカツ	350	
味噌汁	90	具にかかわらず同値段
ふりかけご飯	80 or 90	のりふりかけのみ 80 円

dish の値段を計算する関数 price_of_dish は以下のように定義できる．

```
# let price_of_dish d =
    match d with
      PorkCutlet -> 350
    | Soup _ -> 90
    | Rice f -> (match f with Shake -> 90 | Katsuo -> 90 | Nori -> 80)
  ;;
val price_of_dish : dish -> int = <fun>
```

ご飯の場合，ふりかけで値段が違うので，ふりかけを表す f についてさらに match で場合分けをして値段を計算している．

実は，コンストラクタパターンの引数部にさらにコンストラクタパターンを書いて入れ子にすることで，ふたつの match をひとつにまとめることができる．

```
# let price_of_dish d =
    match d with
      PorkCutlet -> 350
    | Soup _ -> 90
    | Rice Shake -> 90
    | Rice Katsuo -> 90
    | Rice Nori -> 80
  ;;
val price_of_dish : dish -> int = <fun>
```

Rice Katsuo は d が Rice Katsuo である場合を表すパターンとなっている．さらに，or パターンを使うと，鮭と鰹の場合をまとめることができる．

```
# let price_of_dish d =
    match d with
      PorkCutlet -> 350
    | Soup _ -> 90
    | Rice (Shake | Katsuo) -> 90
    | Rice Nori -> 80
  ;;
val price_of_dish : dish -> int = <fun>
```

味噌汁も同じ値段なので，さらに or パターンでまとめて書いてもよい (`Soup _ | Rice (Shake | Katsuo) -> 90` になる)．ただし，`Soup m_and_g | (Rice (Shake | Katsuo))` のように，Soup の味噌，具材に**名前をつけてはいけない**．これは，or パターンで列挙した可能性のうちどの場合だったとしても—Soup だろうが Rice だろうが—-> の右側で同じ変数が使えなければならない，ということからきている．個人的には値段改定があってどちらかだけ値上りした時に少し面倒なので，このあたりに留めたいと思っている．

このように場合の数が増えて，パターンが複雑化してくると exhaustiveness check のありがたみが段々とわかってくる[5]．

1.2.2.1 練習問題

- 一品が菜食主義者でも食べられるかどうかを判定する関数 `isVeggieDish : dish -> bool` を定義せよ．

:::spoiler 解答例

```
let isVeggieDish d =
  match d with
    PorkCutlet -> false
  | Soup m_and_g -> true
  | Rice f -> isVeggie f
```

:::

1.2.3 再帰ヴァリアント

次に，一品料理を並べて定食 (menu 型) を作ろう．定食は一品料理を好きな数だけ並べてよいものとする．好きな数だけ並べたいので，1 皿の場合，2 皿の場合，と皿の数に対応したコンストラクタを用意するのでは間に合わない．

そこで，定食を，

- 0 皿の定食を表す `Smile` [6]
- 既存の定食に，もう一皿先頭に加えたコース料理を表す `Add`

という二種類のコンストラクタで表現する．

[5] とはいえ，場合によっては，以下のようにふりかけの値段を計算する `price_of_rice` を定義し，それを使うことがベターなこともありえる．この定義が最善，といっているわけではない．

```
# let price_of_furikake f =
    match f with
      Shake -> 15
    | Katsuo -> 15
    | Nori -> 5
  ;;
val price_of_furikake : furikake -> int = <fun>
# let price_of_dish d =
    match d with
      PorkCutlet -> 350
    | Soup m_and_g -> 90
    | Rice f -> price_of_furikake f + 75
  ;;
val price_of_dish : dish -> int = <fun>
```

[6] https://toyokeizai.net/articles/-/70541

```
type menu = Smile | Add of {d: dish; next: menu};;
```

Add の of 以下に今定義しようとしている型の名前 menu が現れている **再帰的な型定義**となっている．しかし，ここまでのことが理解できていれば menu 型の値を作るのはそれほど難しくはない．

```
# let m1 = Smile;;                              (* 貧乏人は笑顔で我慢 *)
val m1 : menu = Smile
# let m2 = Add {d = PorkCutlet; next= m1};;     (* トンカツのみ *)
val m2 : menu = Add {d = PorkCutlet; next = Smile}
# let m3 = Add {d = Rice Nori; next= m2};;      (* のりふりかけご飯を追加 *)
val m3 : menu = Add {d = Rice Nori; next = Add {d = PorkCutlet; next = Smile}}
# let m4 = Add {d = Rice Shake; next= m3};;     (* ごはんのおかわりつき *)
val m4 : menu =
  Add
   {d = Rice Shake;
    next = Add {d = Rice Nori; next = Add {d = PorkCutlet; next = Smile}}}
```

さて，定食の値段を計算する関数 price_of_menu を考えてみよう．引数は menu なので自然に以下のような定義が思いつく．

```
# let price_of_menu m =
    match m with
      Smile -> 0
    | Add {d = d1; next = m'} -> ???
    ;;
```

(Add の引数はレコードなのでレコードパターンを使って，フィールドの値を let を使わずに取り出していることにも注目してもらいたい．)

さて，??? の部分はどうなればよいだろうか．要するに 1 品目 (d1) と残り (m') から構成されている定食なのだから，この値段は d1 の値段と残り m' の値段である．残りの値段は price_of_menu を **再帰的に**呼出すことによって計算できる．以下が完成版の定義である (rec を忘れずに)．

```
# let rec price_of_menu m =
    match m with
      Smile -> 0
    | Add {d = d1; next = m'} -> price_of_dish d1 + price_of_menu m'
    ;;
val price_of_menu : menu -> int = <fun>
```

データ構造が再帰的に定義されているので，それを処理する関数は自然と再帰関数になる．

1.2.3.1 練習問題

- 定食が菜食主義者でも食べられるかどうかを判定する再帰関数 isVeggieMenu : menu -> bool を定義せよ．

:::spoiler 解答例

```
let rec isVeggieMenu m =
  match m with
        Smile -> true
  | Add {d = d1; next = m'} -> isVeggieDish d1 && isVeggieMenu m'
```

:::

1.3 OCaml 爆速入門 (その 3) ー変更可能データ構造

1.3.1 変更可能レコード

さて，(ここまでの) レコードについては，レコードを一度作ったら，そのフィールドの値を変更することはできない．{〈式 0〉 with 〈フィールド名 1〉 = 〈式 1〉 ; ... ; 〈フィールド名 n〉 = 〈式 n〉} という形式で，既存のレコード〈式 0〉を使って，フィールドの新しい値が with 以下に従う (指定されていないフィールドは〈式 0〉の値を再利用) ような新しいレコードを作ることはできる (これを functional update ということがある) が，これはあくまでも新しいレコードを作るもので，〈式 0〉のレコードはそのままである．

```
# let p = {origin with y = 3};;
val p : point = {x = 0; y = 3}
# origin;;
- : point = {x = 0; y = 0}
```

変更可能レコード (mutable record) は，フィールド値を (破壊的に) 変更できるようなレコードのことである．変更が可能かどうかはレコード型の宣言を行う際に，フィールド名の前に mutable をつけることでフィールド毎に指定することができる．

```
# type mutable_point = {mutable x : int; mutable y : int; };;
type mutable_point = {mutable x : int; mutable y : int; }
# let m_origin = {x = 0; y = 0};;
val m_origin : mutable_point = {x = 0; y = 0}
```

フィールドの値を変更するためには〈式 1〉.〈フィールド名〉<-〈式 2〉という形式を使う．これは〈式 1〉のレコードの〈フィールド名〉のフィールドの値を〈式 2〉に書き換える，というものである．

```
# m_origin.x <- 2;;
- : unit = ()
# m_origin;;
- : mutable_point = {x = 2; y = 0}
```

変更を行った後で m_origin の値を見ると変わっているのがわかる．このようなレコードの更新を functional update に対し，destructive update, in-place update ということがある．

このフィールド値の変更を行っている m_origin.x <- 2 は Java であれば (代入) 文相当のものであるが，OCaml ではこれも式の一種である．式の型は unit 型と呼ばれ，その型の値は () という定数ひとつである．その他，画面に文字列や数を端末画面に表示する関数も unit を返す関数として定義されている．unit 型は

Java (や C) の void 型と似ているが，void 型の値は存在しないという点が違う[7].

```
# print_int;;
- : int -> unit = <fun>
# print_int (5+15);;
20- : unit = ()
```

表示されている 20 は式の値ではなく，print_int の呼出しの結果表示されたものであることに注意せよ．

unit 型の式は (その計算が止まれば) 値が常に () なので，値自体にはほとんど意味がなく，その式の計算中に引き起こされるレコードのフィールドの変更や画面への文字の表示などの方が重要である．こういった，式の計算中に起こる，「値が求まる以外のこと」一般を総称して計算効果 (computational effect) という．

1.3.1.1　変更可能レコードを理解する　変更可能なフィールドを持つレコードは，Java オブジェクトのように，実体はヒープと呼ばれるメモリ領域にあって値としてはそのレコードの ID・背番号が使われていると考えるのがよい．

```
# let p1 = {x = 0; y = 1};;
val p1 : mutable_point = {x = 0; y = 1}
```

のようにレコードを作る際には，1. ヒープ領域にふたつのフィールド分を持つレコード用の領域を確保し，1. x フィールドに 0，y フィールドに 1 を格納し，1. そのレコードの ID に p1 という名前をつける．という手順が踏まれていると考える．

```
# p1.x <- 2;;
- : unit = ()
```

という変更式は，p1 が指すレコードの x フィールドに 2 を書き込む，と考えられる．

```
# let p2 = p1;;
val p2 : mutable_point = {x = 2; y = 1}
```

のような定義は，単に p1 という名前がついたレコードの ID に p2 という別名をつけていると考える．(実は，mutable フィールドを持たないレコードも実装上はヒープ領域に確保していて，let で名前がつけられているのはレコードそのものではなくレコードの ID なのだが，フィールドの書き換えが起きない限り，p1 はレコードそのものにつけられた名前で let p2 = p1;; がレコードのコピーを行っているように考えても差し支えない．)

さて，p1 も p2 も同じレコードを指しているので，

```
# p2.y <- 3;;
- : unit = ()
```

としてから，p1.y の値を求めると 3 が得られることになる．

```
# p1.y;;
- : int = 3
```

一方，

[7] Java で返値型が void になっているメソッドを書いていると，この型の要素なんてないのに return; は一体何を返しているんだろう，という不思議な気持ちになります．

```
# let p3 = {p2 with x = 2};;
```

のような with を使った functional update の場合は，1. ヒープ領域に新たにレコードを確保し，1. x フィールドに 2，y フィールドには p2.y を格納したレコードを作り，1. そのレコードの ID に p3 という名前をつける．という手順が踏まれていると考える．新しいレコードをヒープ領域に確保し，(更新されたフィールド以外の) 各フィールドの値はコピーしている，というのがポイントである．

1.3.1.2　ref 型　OCaml では Java などの代入可能な「変数」—OCaml では**参照 (reference)** と呼ぶ—を mutable record の一種で表現することができる．参照は内部的には contents というフィールドをひとつだけ持つ mutable record だが，通常は，ref〈式〉という構文で作る．

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

型は int ref となっているが，これは「整数への参照型」である．実は，任意の型についての参照を作ることできる．

```
# let y = ref 3.14;;
val y : float ref = {contents = 3.14}
```

参照の型は〈型〉ref と複合的な型表現になっている．ref は単体では型ではなく，「何を格納しているか」の情報をつけてはじめて型になる．このような型を**パラメータ付き型 (parameterized type)** と呼ぶことがある．パラメータ付き型は参照特有のものではなく，プログラマが定義するデータ型でも作ることができる．これについては多相性についての章で学ぶ．

参照は mutable record なので，.contents や<- で読み書きすることができるが，参照のために特別な構文が用意されている．参照の中身の読み出しには前置演算子 !，更新には := という中置演算子を使う．

```
# x := 2;;  (* x.contents <- 2 と同じ *)
- : unit = ()
# !x;;  (* x.contents と同じ / boolean の否定ではない *)
- : int = 2
# x := !x + 1;;
- : unit = ()
# !x;;
- : int = 3
```

これらは Java で書くならば，

```
int x = 1;
x = 2;
x = x + 1;
```

と同じようなものだが，参照の内容の読み出しには明示的に ! が必要である．これは面倒なように思われるが[8]変数という値を格納する箱と，その中身を区別する，という意味で，概念の整理に役に立つと思う[9]．

[8]実際面倒である

[9]実際，Java や C で変数の説明には，変数のアドレスを表す **左辺値 (L-value)** と変数の中身を表す **右辺値 (R-value)** という概念が使われる．つまり x = x + 1; という文は，「x の左辺値のアドレスに x の右辺値プラス 1 を格納する」というように説明される．

1.3.1.3 構造等価性と物理等価性 (*)

(ここは少しアドバンスドなトピックです．)OCaml の比較演算子 = は，いろいろな型の値の「等しさ」を調べることができ，整数や真偽値だけではなく，(同じ型の) レコードやヴァリアント同士の比較をすることができる．レコード同士の比較の場合，全てのフィールドで格納された値同士が等しいことを以って，全体が「等しい」と扱われる．ヴァリアントの場合は，コンストラクタ同士が等しく，かつ，コンストラクタが運んでいる値同士が等しい場合に等しいと判断される．

```
# let p1 = {x = 0; y = 1};;
val p1 : mutable_point = {x = 0; y = 1}
# let p2 = {x = 0; y = 0};;
val p2 : mutable_point = {x = 0; y = 0}
# p2.y <- 1;;
- : unit = ()
# p1 = p2;;
- : bool = true
# Rice Nori = Rice Shake;;
- : bool = false
```

OCaml には == という比較演算子も用意されている．(否定形は!= である．) これもいろいろな型の値の「等しさ」を調べるものだが，「等しさ」の基準が= とは異なっていて，データの ID 同士を比較するものとなっている．例えば，下のプログラムで，p1 と p2 は各フィールドの値は等しいものの，別に作られたレコードなので ID が異なり，== の意味では等しいとは判断されない．一方，p3 は p1 と同じ ID(に別の名前をつけただけ) なので == で等しいと判断される．

```
# p1 == p2;;
- : bool = false
# let p3 = p1;;
val p3 : mutable_point = {x = 0; y = 1}
# p3 == p1;;
- : bool = true
# p3 = p1;;
- : bool = true
# p3 = p2;;
- : bool = true
```

= のような構造を辿って行う比較による等価性を**構造等価性 (structural equality)**，== のような ID の比較による等価性を **物理等価性 (physical equality)** と呼ぶ．Java の == もオブジェクトの ID を比較するものなので物理等価性による比較を行う演算子である．構造等価性による比較演算子は用意されていないが，equals というメソッドを構造等価での比較のために使うことが多い．

一般に，物理等価ならば構造等価でもある．また，(変更可能なデータについては) ふたつの式の値が物理等価であることと，一方に施された変更 (書き換え) がもう一方を通じても観察できることが___論理的に同値___である．

```
(* この時点では p1 と p2 の等価性を知らないと仮定しよう *)
# p1.x <- 0;;
- : unit = ()
```

```
# p2.x <- 0;;
- : unit = ()            ともに x を 0 にしておく
# p1.x <- 100;;
- : unit = ()
# p2.x = 100;;
- : bool = true          p2.x も変更されてる! つまり p1 == p2
- : bool = false         p2.x は変更されていない! つまり p1 != p2
```

また，構造等価性はデータの構造を辿って比較を行うので時間がかかる場合がある．構造に循環があると止まらないことすらある．一方，物理等価性は単なる ID(実はアドレス) 同士の比較なのでデータの大きさに関わらず定数時間で行えるが，内容的に区別する必要がなさそうなデータについても等しくないと判断する場合がある．特に，現在の OCaml の実装では，

```
# Shake == Shake;;
- : bool = true
# Rice Shake == Rice Shake;;
- : bool = false
```

と (OCaml 言語の内部実装を知らないと) よくわからない結果になってしまう．OCaml の言語仕様では，変更不可能なデータ型についての == の意味は実装依存となっている (バージョンがあがる前後で同じプログラムの挙動が変わるかもしれない!) ので使いどころに注意が必要である．

1.3.2 制御構造

1.3.2.1 逐次実行 式をセミコロンで区切って並べると，左から順に逐次実行 (sequential execution) をして，最後の式の値が全体の値となる．

```
# p1.x <- 0; p2.y <- 4; print_int p1.y; 100
4- : int = 100
```

Java ではセミコロンは文や宣言の終わりを示す記号 (**ターミネータ (terminator)** という) なので各文・宣言の後に必ずセミコロンが必要になるが，OCaml では式と式の間にはさむ記号 (**セパレータ (separator)** という) なので，最後の式の後にはセミコロンをつけてはいけない．

逐次実行において最後以外の式の値は単に捨てられてしまう．ので，unit 以外の型の式を書くと，「捨てていいの？」という意味で警告が発生する．また，計算効果がない式を書くのは，計算をして結果を捨てるだけなので無駄である．

```
# 1+1; 5;;
Characters 0-3:
  1+1; 5;;
  ^^^
Warning 10: this expression should have type unit.
- : int = 5
```

OCaml には ignore という引数に関わらず () を返すだけの関数があり，わざと値を捨てたい場合は，この関数を呼んで引数を捨てることを明示する慣習となっている．

```
# ignore(1+1); 5;;
- : int = 5
```

1.3.2.2　条件分岐　OCaml での，if 式 if〈条件式〉then〈式 1〉else〈式 2〉は「〈条件式〉の値が true ならば〈式 1〉の値を，false ならば〈式 2〉の値になる」というものだが，〈式 1〉や〈式 2〉が計算効果を持つ場合は，Java や C などの if 文に近い役割になる．

```
# let is_positive n =
    if n > 0 then print_string "n is positive\n"
    else print_string "n is not positive\n";;
# is_positive 100;;
n is positive
- : unit = ()
```

Java の if 文では else 部分を省略できるが，OCaml の if 式も unit 型の式に限って else を省略することができる．

```
# let is_positive' n =
    if n > 0 then print_string "n is positive\n";;
# is_positive' 100;;
n is positive
- : unit = ()
# is_positive' (-100);;
- : unit = ()
```

then 部が unit 型の式でない場合にはエラーになる．これは，条件が成立しなかった場合に，どんな値になればよいかわからないためであるとも考えられるし，if e0 then e1 は if e0 then e1 else () の略記だと考えてもよいだろう．

```
let f n = if n > 0 then 2;;
Characters 24-25:
  let f n = if n > 0 then 2;;
                          ^
Error: This expression has type int but an expression was expected of type
         unit
```

if と逐次実行の ; の優先度には注意が必要である．if の方が ; よりも強く結合するので，then と else の間にうかつに; を入れると，妙なエラーに遭遇することになる．

```
# let is_positive n =
    if n > 0 then print_int n; print_string " is positive"
    else print_int n; print_string " isn't positive";;
```

この定義の意図は，n の値に続けて“is positive”もしくは“isn’t positive”を表示させるというものだが，OCaml は ; が現れたところで一旦 if を切ってしまうので，else が現れたところで，突然 else を書くんじゃない，と文法エラーになってしまう．

```
Characters 86-90:
       else print_int n; print_string " isn't positive";;
       ^^^^
Error: Syntax error
```

これを修正するには，; でつながれた部分を begin, end で囲む[10].

```
# let is_positive n =
    if n > 0 then
      begin
        print_int n;
        print_string " is positive"
      end
    else print_int n; print_string " isn't positive";;
val is_positive : int -> unit = <fun>
```

しかし，else にある ; についても，; が現れたところで，if が終わったと判断するので，これでは n の値に関わらず print_string " isn't positive" が実行されてしまい

```
# is_positive 100;;
100 is positive isn't positive- : unit = ()
```

という結果になってしまう．正しくは，

```
# let is_positive n =
    if n > 0 then
      begin
        print_int n;
        print_string " is positive"
      end
    else
      begin
        print_int n;
        print_string " isn't positive"
      end;;
```

である．

この優先順位のつけ方は，奇妙に感じるかもしれないが，Java や C 言語の「then や else 部に複数の文を書く時には{} で囲まければいけない」という規則に合わせているような気もする．

また，begin, end は，実は (,) と同じで，何でも囲むことができるし，begin, end の代わりに (,) を使ってもよい．

```
# let is_positive n =
    if n > 0 then (
      print_int n;
```

[10] Ruby っぽい？ (歴史的には順序が逆ですが．)

```
      print_string " is positive"
    ) else (
      print_int n;
      print_string " isn't positive"
    );;
val is_positive : int -> unit = <fun>
```

ふつうの数式を begin end で囲ってもよい (ふつうはしないが).

```
# begin 1 + 2 end * 5;;
- : int = 15
```

ただし，さすがに begin を) で閉じたり，(を end で閉じたりしてはいけない.

```
# begin 2 + 3 ) * 10;;
Characters 0-5:
  begin 2 + 3 ) * 10;;
  ^^^^^
Syntax error: 'end' expected, the highlighted 'begin' might be unmatched
# (2 + 3 end * 10;;
Characters 0-1:
  (2 + 3 end * 10;;
  ^
Syntax error: ')' expected, the highlighted '(' might be unmatched
```

1.3.2.3 ループ OCaml にもループのための構文 while と for が用意されている．が，使用頻度が少ないせいか，特に for は Java や C のそれと比べてかなり制限されている.

- while〈条件式〉do〈ループ本体式〉done …〈条件式〉が true である限り〈ループ本体式〉を評価する.
- for〈変数〉=〈初期値〉to〈上限〉do〈ループ本体式〉done …〈変数〉を〈初期値〉(整数) から 1 ずつ増やし，〈上限〉以下である限り〈ループ本体式〉を評価する.
- for〈変数〉=〈初期値〉downto〈下限〉do〈ループ本体式〉done …〈変数〉を〈初期値〉(整数) から 1 ずつ減らし，〈下限〉以上である限り〈ループ本体式〉を評価する.

いずれの構文でも〈ループ本体式〉は unit 型であることが要請される.

```
# for i = 1 to 10 do print_int i; print_newline() done;;
1
2
3
4
5
6
7
8
9
```

```
10
- : unit = ()
```