情報システム科学実習II第7回
参照，例外処理，入出力

担当: 山口 和紀・五十嵐 淳

2001年11月28日

今週はOCamlに備わっている副作用を伴う機能を概観する。OCamlプログラムの実行は式を値に評価する過程であった．副作用とは、式の評価中に起こる「なにか」であり、代入、ファイルへの入出力などがその一例である。副作用を伴わない式は何度評価しても結果は不変であり，また，式をその評価結果の値で置換えてもプログラムの挙動は変化しない．一方，副作用を伴う式は，評価する度に違った結果が得られる可能性があり，評価結果でその式を置換えててしまうとプログラムの挙動が変わってしまう。そのため，副作用を伴うプログラムの挙動はしばしば推論しにくくなる。

副作用を伴う関数の一例は，端末への表示を行う関数 `print_string` である．

```
# print_string "Hello, World!\n";;
Hello, World!
- : unit = ()
```

ここでは，副作用として引数の文字列が標準出力 (端末画面) へ書き出されている．また，この関数は返り値として `()` を返している．このように，副作用自体に意味があり計算結果としては重要な値を伴わない関数は，典型的には **unit** 型を返す関数として表現される．(OCamlでは基本的に全てのプログラム中の式は何らかの値を返す．命令型言語におけるコマンドもこのような関数として表現されている．)

また，関数 `read_line` は，引数 `()` で呼ばれると，端末からの入力を待ち，その結果を文字列として返す.

```
# read_line ();;
foo             ⇐ キーボードからの入力
- : string = "foo"
```

キーボードからの入力がかわる度に同じ式 `read_line ()` の返す値は変っていく．そのため，副作用を伴う式については，いつ評価されるかを常に頭にいれてプログラムしなければならない.

1　参照、更新可能レコードと配列

ここまで見たプログラムでは一度作成したデータを更新する手段は与えられていない．OCaml にはいくつかの更新可能なデータ構造が用意されている．**参照**(*references*)，**更新可能レコード**(*mutable records*)，**配列**(*array*) である．

1.1　参照

OCaml における参照の概念はほぼメモリアドレスと考えることができ，C における変数，ポインタなどに対応する．参照の値はそのアドレスが指す先にデータを格納しており，代入操作によってその内容を書き換えることができる．t 型の値への参照値は `ref` t 型が与えられる．

参照の生成は `ref` 関数で行う．`ref` 関数は参照先に格納する初期値を引数としてとり，それを格納したアドレスを返す．(ただし，実際のアドレスの数値が観察できるわけではない．) この関数は通常の数学的な意味の関数とは違い，`read_line` のように呼び出す度に新しい値 (アドレス) を返す．参照から格納された値を取出すには前置オペレータ `!` を使用する．また，代入式

〈式$_1$〉 := 〈式$_2$〉

は〈式$_1$〉を評価した結果の参照に〈式$_2$〉の評価結果を代入するものである．右辺の型が t であるときには，左辺の型は `ref` t でなければならない．代入式自体は代入という副作用を起すだけで，その結果は常に `()` である．以下は簡単な参照を使った例である．

```
# let p = ref 5 and q = ref 2;;
val p : int ref = {contents=5}
val q : int ref = {contents=2}
# (!p, !q);;
- : int * int = 5, 2
# p := !p + !q;;
- : unit = ()
# (!p, !q);;
- : int * int = 7, 2
```

参照を理解する上で大事なのは，格納されている値とそれのアドレスを区別することである．代入式は左辺のアドレスの**内容**を右辺の値で書き換える．`!` が必要なのも，この区別をはっきりさせるものと思えば良い．C などの代入文では，`i = j` と書いたときに，左辺は暗黙のうちに変数 `i` のアドレスをさし，右辺の `j` は変数 `j` の内容を指し示している．

参照値はデータ構造に格納することもできる．

```
# let reflist = [p; q; p];;
val reflist : int ref list = [{contents=7}; {contents=2}; {contents=7}]
# p := 100;;
- : unit = ()
# reflist;;
- : int ref list = [{contents=100}; {contents=2}; {contents=100}]
```

代入操作が reflist の出力結果に変化を与えていることに注意．ただし，reflist の 3 要素の値 (アドレス) が変化したのではなく，その指し示す値が変っただけであることに留意せよ．このように同じ参照が複数個所で使用されることによって，ある場所での代入が別の場所に影響をおよぼすことを**エイリアシング**(*aliasing*) といい，命令型言語のプログラミングで頭を悩ませる種の一つである．

また，参照の参照を考えることもでき，C におけるポインタのように扱うことができる．

```
# let refp = ref p and refq = ref q;;
val refp : int ref ref = {contents={contents=100}}
val refq : int ref ref = {contents={contents=2}}
# !refq := !(!refp);;
- : unit = ()
# (!p, !q);;
- : int * int = 100, 100
```

1.2 更新可能レコード

これまでみたレコードではフィールドの値は更新できなかった．with を用いても，新しいフィールドを使った新しいレコードを生成していた．実は，OCaml のレコードはフィールド毎に更新可能かどうかを指定することができる．更新可能フィールドには型宣言時に mutable キーワードを用いる．

```
# type teacher = {name : string; mutable office : string};;
type teacher = { name : string; mutable office : string; }
# let t = {name = "Igarashi"; office = "604B"};;
val t : teacher = {name="Igarashi"; office="604B"}
```

フィールドの更新は,

⟨式$_1$⟩.⟨フィールド名⟩ <- ⟨式$_2$⟩

という形で,⟨式$_1$⟩の値であるレコードの⟨フィールド名⟩フィールドの内容を⟨式$_2$⟩で置換える．

```
# t.office <- "605B";;
- : unit = ()
# t;;
- : teacher = {name="Igarashi"; office="605B"}
```

Exercise 7.1 ref 型の値の表示を見て気づいている人もいるかもしれないが，ref 型は以下のように定義された 1 フィールドの更新可能なレコードである．

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

関数 ref, 前置オペレータ !，中置オペレータ := の定義を，レコードに関連した操作で書け．

1.3 配列

配列は同じ種類の値の集合を表すデータという意味ではリストと似通っているが，長さは生成時に固定であり，どの要素にも直接 (先頭から順に辿ることなく) アクセスできる．また，各要素を更新することができる．配列には，要素型を t として t array という型が与えられる．配列の生成，要素の参照，更新はそれぞれ，

[|⟨式$_1$⟩; ⟨式$_2$⟩; ...; ⟨式$_n$⟩|]
⟨式$_1$⟩.(⟨式$_2$⟩)
⟨式$_1$⟩.(⟨式$_2$⟩) <- ⟨式$_3$⟩

という形で行い，n 要素の配列で初期値がそれぞれ ⟨式$_i$⟩ であるもの，配列 ⟨式$_1$⟩ の ⟨式$_2$⟩ 番目の要素，配列 ⟨式$_1$⟩ の ⟨式$_2$⟩ 番目の要素を ⟨式$_3$⟩ で更新，という意味である．配列の大きさを越えた整数で要素にアクセスすると Invalid_argument という例外が発生する．

Exercise 7.2 与えられた参照の指す先の整数を 1 増やす関数 inc を定義せよ．

```
# let x = ref 3;;
val x : int ref = {contents=3}
# incr x;;
- : unit = ()
# !x;;
- : int = 4
```

Exercise 7.3 以下で定義する funny_fact は再帰を使わずに階乗を計算している．どのような仕組みで実現されているか説明せよ．

```
# let f = ref (fun y -> y + 1)
# let funny_fact x = if x = 1 then 1 else x * (!f (x - 1));;
val f : (int -> int) ref = {contents=<fun>}
val funny_fact : int -> int = <fun>
# f := funny_fact;;
- : unit = ()
# funny_fact 5;;
- : int = 120
```

2 制御構造

最初の説明でも触れたように，副作用を伴う計算は，副作用の起こる順番に気をつけてプログラミングしなければならない．例えば，以下のようなプログラムに見るように，

```
# let x = print_string "Hello, " in
# print_string "World!\n";;
Hello, World!
- : unit = ()
```

x が束縛される値の計算 (と，それに伴う副作用) が in 以下の計算よりも前に起こることを知っておかなければならない．特に注意が必要なのは，OCaml では，関数呼出しなどの複数の引数の評価順序である．

```
# let f x y = 2 in
# f (print_string "Hello, ") (print_string "World\n");;
World
Hello, - : int = 2
# (print_string "Hello, ", print_string "World\n");;
World
Hello, - : unit * unit = (), ()
```

現在の実装では，後ろの引数から順に評価が行われるが，OCaml の仕様としては未定義なので，引数の計算に副作用を伴うときには，let などを用いて計算順序をプログラム中に明示的にする必要がある．

OCaml では，命令型言語に見られるような制御構造がいろいろ用意されている．まず，上のプログラムのような「コマンド列」を表現するための機能として，

⟨ 式$_1$⟩; ⟨ 式$_2$⟩; ···; ⟨ 式$_n$⟩

という形の式が用意されている．この式全体は，⟨ 式$_1$⟩ から順番に評価を行い，⟨ 式$_n$⟩ の値を全体の値とする．また，途中の式の結果は捨てられてしまうので，⟨ 式$_{n-1}$⟩ までの式は通常は副作用を伴うものである．OCaml では，⟨ 式$_i$⟩ $(i < n)$ が **unit** 型でない場合には Warning を発行する．() でなく，意味のある値を返す式の値を捨ててしまうのはバグであることが多いのからである．プログラマが，値が要らないことを確信している場合は，ignore 関数をつかって，明示的にそのことを示すことが推奨される．

```
# print_string "Hello, "; print_string "World\n";;
Hello, World
- : unit = ()
```

また，ループのための式として，while 式，for 式が用意されている．while 式は

while ⟨ 式$_1$⟩ do ⟨ 式$_2$⟩ done

という形で，**bool** 型の式 ⟨ 式$_1$⟩ の評価結果が false になるまで ⟨ 式$_2$⟩ の評価を行う．式全体は () を返す．for 式は

for ⟨ 変数名 ⟩ = ⟨ 式$_1$⟩ to ⟨ 式$_2$⟩ do ⟨ 式$_3$⟩ done

もしくは

for ⟨ 変数名 ⟩ = ⟨ 式$_1$⟩ downto ⟨ 式$_2$⟩ do ⟨ 式$_3$⟩ done

という形で，まず ⟨ 式$_1$⟩, ⟨ 式$_2$⟩ を整数 n, p に評価する．その後，⟨ 変数名 ⟩ を $n, n+1, \ldots, p$ (downto の場合は $n, n-1, \ldots, p$) の順に束縛して ⟨ 式$_3$⟩ を評価する．⟨ 変数名 ⟩ の有効範囲は ⟨ 式$_3$⟩ である．式全体の値は () である．

これらのループを表現する式は，副作用を伴わないとまったく意味がないことに注意すること．副作用がなければ，while の繰り返し条件の式つねに同じ値を返すし，for 式も計算をいくらか繰り返して () を返すだけである．

以下はキーボードから入力された行を二つつなげて画面に出力し返す関数である．終了にはピリオドだけからなる行を入力する．

```
# let parrot () =
#   let s = ref "" in
#     while (s := read_line (); !s <> ".") do
#       print_string !s;
#       print_endline !s;
#     done;;
val parrot : unit -> unit = <fun>
```

print_endline 関数は引数の文字列に改行をつけて出力するための関数である．

Exercise 7.4 参照を使って階乗関数を定義する．... 部分を埋めよ．

```
let fact_imp n =
  let i = ref n and res = ref 1 in
    while (...) do
      ...;
      i := !i - 1
    done;
    ...;;
```

3 例外処理

例外(*exception*) は、計算を進めていく過程でなんらかの理由で計算を中断せざるをえない状況を表現するための仕組みである。なんらかの理由の例としては、0 での除算、パターンマッチの失敗、ファイルのオープンの失敗、など典型的には実行時エラーの発生した状況が多い。基本的には例外が発生すると残りの計算をせずに中断して値を返さず実行を終了してしまう。(インタラクティブコンパイラでは入力プロンプトに戻る。) 例外発生は OCaml 式がその型の値を返さない唯一の例である。

以下は例外を発生する式である。2 番目の式はファイルを開くための関数が呼び出されているが，ファイルがないという例外が発生している．最後の式では、4 / 0 の結果の出力だけでなく残りの計算である文字列の出力が行われずに実行が中断されていることがわかるだろう。

```
# hd [];;
Uncaught exception: Match_failure ("", 9, 19).
# open_in "";;
Uncaught exception: Sys_error ": No such file or directory".
# print_string (string_of_int (4 / 0)); print_string "not printed";;
Uncaught exception: Division_by_zero.
```

それぞれ、パターンマッチングが失敗したことを示すMatch, OSに対するシステムコール関連のエラーが発生したことを示すSys_error, 0での除算が発生したことを示すDivision_by_zero, が発生している。また、いくつかの例外では例外の名前のあとにOCamlの値が付加されて例外に関する詳しい情報を提供している。

OCamlでは、プログラマが新しい例外を宣言、その例外を発生させることができる。また、例外の発生をプログラム中で検知し、中断される前に処理を再開することができる。

3.1 exception 宣言と raise 式

新しい例外の宣言はexception宣言で行う。

```
# exception Foo;;
exception Foo
```

例外の名前は、ヴァリアント型のコンストラクタと同様、英大文字で始まらなければならない。例外を発生させるのはraise式で行う。

```
# raise Foo;;
Uncaught exception: Foo.
```

raise式は、値を返さずに例外を発生させるので、任意の場所で用いることができる。別の言い方をするとraise式には任意の型が与えられる。下の関数定義の例では、raise Foo式が真偽値や整数の必要な場所で使われていることがわかるだろう。

```
# let f () = if raise Foo then raise Foo else 3;;
val f : unit -> int = <fun>
```

先に見た、Sys_errorのような値を伴う例外は、宣言時に何の型の値を伴うかも宣言しなければならない。以下は整数を伴う例外の宣言とraiseの例である。

```
# exception Bar of int;;
exception Bar of int
# raise (Bar 2);;
Uncaught exception: Bar 2.
```

OCamlでは(raiseで発生させる前の)例外にはexn型が与えられ、第一級の値として関数に渡したり、データ構造に格納することができる。別の見方をすると，exception宣言により，新しいコンストラクタがexn型に追加されることになる．またヴァリアント形と同様にパターンマッチで、例外コンストラクタが適用された値を取り出すことができる。

```
# let exnlist = [Bar 3; Foo];;
val exnlist : exn list = [Bar 3; Foo]
# let f = function
#     Foo -> 0
#   | x -> raise x;;
val f : exn -> int = <fun>
```

```
# f Foo;;
- : int = 0
# f (Bar 4);;
Uncaught exception: Bar 4.
```

標準ライブラリの例外　その他の, いくつかの定義済の例外を紹介しておく. Invalid_argument, Failure は文字列を伴う例外で, いくつかのライブラリ関数で発生する. 前者は関数引数が意味をなさない場合, 後者は関数引数は正しいが何らかの理由で計算が失敗した場合に発生する. Not_found は探索を行う関数で, 探索対象が見つからなかった場合に発生させるものである. End_of_file はファイルからの入力関数がファイルの終端に到達した場合に発生する.

また例外を発生させるだけの関数, invalid_arg, failwith なども用意されている.

```
# failwith "foo";;
Uncaught exception: Failure "foo".
```

3.2　例外の検知

式の評価中に起こる例外は、その種類がわかる場合には、try 式で、検知して後処理を行うことができる。try 式の一般的な形は、match 式と似ていて、

try ⟨式⟩ with
    ⟨パターン$_1$⟩ -> ⟨式$_1$⟩
  | ...
  | ⟨パターン$_n$⟩ -> ⟨式$_n$⟩

となる。まず、⟨式⟩を評価し、例外が発生しなければその値を全体の値とする。評価途中で例外が raise された場合には、順にパターンマッチを行っていき、マッチした時点で⟨式$_i$⟩を評価し、try 式全体の値となる。何もマッチするものがなかった場合には、もともと発生した例外が再発生する。try 式の値は、⟨式⟩,⟨式$_1$⟩, ...,⟨式$_n$⟩、のいずれかになるので、これらの式の型は一致していなければならない。

```
# try 4 + 3 with Division_by_zero -> 9;;
- : int = 7
# try 1 + (3 / 0) with Division_by_zero -> 9;;
- : int = 9
# try 1 + (3 / 0) with Sys_error s -> int_of_string s;;
Uncaught exception: Division_by_zero.
```

Exercise 7.5 階乗関数 fact を負の引数に対して Invalid_argument を発生させるように改良せよ.

Exercise 7.6 以下の関数 change は, お金を「くずす」関数である.

```
# let rec change = function
#     (_, 0) -> []
#   | ((c :: rest) as coins, total) ->
#       if c > total then change (rest, total)
#       else c :: change (coins, total - c);;
Characters 18-168:
Warning: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
([], 1)
val change : int list * int -> int list = <fun>
```

与えられた (降順にならんだ) 通貨のリスト coins と合計金額 total からコインのリストを返す.

```
# let us_coins = [25; 10; 5; 1]
# and gb_coins = [50; 20; 10; 5; 2; 1];;
val us_coins : int list = [25; 10; 5; 1]
val gb_coins : int list = [50; 20; 10; 5; 2; 1]
# change (gb_coins, 43);;
- : int list = [20; 20; 2; 1]
# change (us_coins, 43);;
- : int list = [25; 10; 5; 1; 1; 1]
```

しかし，この定義は先頭にあるコインをできる限り試してしまうため，可能なコインの組合わせがあるときにでも失敗してしまうことがある.

```
# change ([5; 2], 16);;
Uncaught exception: Match_failure ("", 19, 169).
```

これを，例外処理を用いて解がある場合には出力するようにしたい．以下のプログラムの，2 個所の ... 部分を埋め，プログラムの説明を行え.

```
let rec change = function
    (_, 0) -> []
  | (c :: rest, total) ->
      if c > total then change (rest, total)
      else
        (try
          c :: change (coins, total - c)
         with Failure "change" -> ...)
  | _ -> ...;;
```

4 チャネルを使った入出力

OCaml には、最初に見た print_string 以外にも様々な入出力用の関数が用意されている.

標準入出力, 標準エラー出力関連の関数　print_char, print_string, print_int, print_float, print_endline, print_newline は標準出力 (通常は端末画面) に引数を書き出す行う関数である．それぞれ，引数の型が異なったり，末尾の改行の出力機能がついていたりするが，基本的な動作は同じである．

print を prerr に変えれば，標準エラー出力に書き出す関数になる．

また，標準入力 (通常はキーボード) から読み込みを行う関数に，read_line, read_int, read_float がある．

ファイル操作とチャネル　OCaml では入出力先を表現するのにチャネルという抽象的な概念を用いている．チャネルは通信路のことで，ここに向かって書き出したり，ここから読み込みを行うことで，その先につながった何か (ファイル，ディスプレイ) に書き込んだりすることができる．チャネルはさらに，入力用，出力用のものにわかれており，OCaml ではそれぞれ，in_channel 型，out_channel 型として定義されている．また，標準入力などは定義済の値として用意されている．

```
# (stdin, stdout, stderr);;
- : in_channel * out_channel * out_channel = <abstr>, <abstr>, <abstr>
```

チャネルに対する入出力には，input_char, input_line, output_char, output_string などの関数で読み書きを行う．

また，ファイル名から新たなチャネルを生成することもできる．出力用のチャネルは open_out, 入力用のチャネルは open_in 関数で得ることができる．どちらも，ファイル名の文字列を引数として受け取る．また，使い終わったチャネルは close_out, close_in 関数で閉じなければならない．

その他の入出力関数については，マニュアルの 18 章を参照されたい．

Exercise 7.7 print_int 関数を stdout, output_string などを用いて定義せよ．

Exercise 7.8 二つのファイル名を引数にとって，片方のファイルの内容をもう片方にコピーする関数 cp を定義せよ．とくに一度開いたファイルは最後に閉じることを忘れないように．

A　OCaml の文法について補足

いくつか，文法について補足しておく．以前に見たように演算子には優先順位がついていて強いものから結合し部分式を構成する．例えば ; は if よりも結合が弱いので，

```
if false then print_string "a"; print_string "b";;
```

は

```
(if false then print_string "a"); print_string "b";;
```

と同じ意味である．また，

```
if false then print_string "a"; print_string "b" else ();;
```

は文法エラーになってしまう．(; まで読んだ時点で if 式が終わったと判断してしまうためで先にある else はみてくれない．)

分かりにくいのは，if など複数のキーワードから構成され，同じ優先度を持つ式が入れ子になった場合である．基本的には OCaml の文法で終端となるキーワードがないもの let, if, match, try は，できる限り先まで含めて式の一部であるように読まれる．例えば，

```
if a then if b then c else d
```

は

```
if a then (if b then c else d)
```

のことである．内側の if が右にできる限り伸びようとしているのがわかるだろう．

また，match の入れ子，練習問題にある match と try の組合わせは要注意である．

```
match e with
  A -> match e' with B -> ... | C -> ...
| D -> ...
```

は D の分岐も内側の match の一部として考えられてしまうので，D が外側の match であるようにするには以下のように括弧が必要である．

```
match e with
  A -> (match e' with B -> ... | C -> ...)
| D -> ...
```

練習問題の try 式も括弧がないと最後の | _ -> 以降が try 式の一部と見なされてしまうのである．

第 4 週の優先順位の表と上のルールで，一見して明らかでない部分式の結合はわかるはずである．(プログラムが見にくくならない程度に括弧をつけるのもよい習慣である．)