Jacques Garrigue, 2005 年 12 月 7 日

8 入出力・コンパイル・デバッグ

8.1 出力

```
# Printf.printf "%s = %d\n" "x" 3;;                    (* C みたいな printf *)
x = 3
- : unit = ()
# let print_pair (s, n) = Printf.printf "%s = %d\n" s n;;
val print_pair : string * int -> unit = <fun>          (* 型が正しい! *)
# List.iter print_pair ["a", 1; "b", 2; "c", 3] ;;
a = 1
b = 2
c = 3
- : unit = ()
# let oc = open_out "test.txt" ;;                      (* ファイルを出力用に開く *)
val oc : out_channel = <abstr>
# List.iter (fun (s,n) -> Printf.fprintf oc "%s = %d\n" s n)
    ["a", 1; "b", 2; "c", 3] ;;
- : unit = ()
# close_out oc ;;                                      (* ファイルを閉じ忘れない *)
- : unit = ()
# Sys.command "cat test.txt" ;;                        (* シェルで命令を実行 *)
a = 1
b = 2
c = 3
- : int = 0
```

8.2 入力

```
# let s = read_line () ;;                              (* 端末から一行を入力 *)
Hello
val s : string = "Hello"
# let hello () =
    Printf.printf "What is your name? ";
    let s = read_line () in
    Printf.printf "Hello %s!\n" s ;;
val hello : unit -> unit = <fun>
# hello ();;
What is your name? Jacques
Hello Jacques!
- : unit = ()
# let square () =
    Printf.printf "Enter a number: ";
    let s = read_line () in
    let x = int_of_string s in                         (* 文字列を整数に変換 *)
    Printf.printf "(%d)^2 = %d\n" x (x*x) ;;
val square : unit -> unit = <fun>
# square ();;
Enter a number: 15
(15)^2 = 225
- : unit = ()
# let read_file name =
    let ic = open_in name in                           (* ファイルを入力用に開く *)
```

```
    let lines = ref [] in
    begin
      try while true do lines := input_line ic :: !lines done
      with End_of_file -> close_in ic          (* ファイルの最後に例外が起きる *)
    end;
    List.rev !lines ;;
val read_file : string -> string list = <fun>
# read_file "test.txt" ;;
- : string list = ["a = 1"; "b = 2"; "c = 3"]
```

8.3 文字列の解析

```
# #load "camlp4o.cma" ;;                          (* 解析用に言語拡張を読み込む *)
        Camlp4 Parsing version 3.08.3

# open Genlex ;;                                   (* 汎用字句解析の定義を輸入 *)
# let lexer = Genlex.make_lexer ["="] ;;   (* '=' をキーワードとする字句解析器 *)
val lexer : char Stream.t -> Genlex.token Stream.t = <fun>
# let s = lexer (Stream.of_string "a = 3") ;;
val s : Genlex.token Stream.t = <abstr>
# (parser [< ' x >] -> x) s ;;                    (* ストリームから一個を読む *)
- : Genlex.token = Ident "a"
# (parser [< ' Ident x >] -> x) s ;;
Exception: Stream.Failure.                  (* 読まれるとストリームから消える *)
# (parser [< ' Kwd x >] -> x) s ;;                  (* パターンマッチもできる *)
- : string = "="
# (parser [< ' Int 0 >] -> "zero" | [< ' Int 3 >] -> "three") s ;;
- : string = "three"                              (* 二つ目のパターンが選ばれた *)
# (parser [< ' x >] -> x) s ;;
Exception: Stream.Failure.                          (* もう読むものがない *)
# let parse_def =                                  (* 「定義」を読む解析器 *)
    parser [< ' Ident x; ' Kwd "="; ' Int n >] -> (x, n) ;;
val parse_def : Genlex.token Stream.t -> string * int = <fun>
# parse_def (lexer (Stream.of_string "a = 3")) ;;
- : string * int = ("a", 3)
# let rec repeat p l s =                                (* 再帰的な解析器 *)
    match s with parser
      [< r = p; s >] -> repeat p (r :: l) s
    | [< >] -> List.rev l ;; (* 次のトークンがない, または Ident でないとき *)
val repeat : ('a Stream.t -> 'b) -> 'b list -> 'a Stream.t -> 'b list = <fun>
# let read_pairs name =                                 (* ファイルの中身を解析 *)
    let ic = open_in name in
    let s = lexer (Stream.of_channel ic) in
    let l = repeat parse_def [] s in
    close_in ic;
    l ;;
val read_pairs : string -> (string * int) list = <fun>
# read_pairs "test.txt" ;;
- : (string * int) list = [("a", 1); ("b", 2); ("c", 3)]
```

参考のために, Genlex.token の定義をここに写す.

```
type token =
    Kwd of string                    (* make_lexer に渡された英数字列か記号列 *)
  | Ident of string                                  (* それ以外の英数字列 *)
  | Int of int
  | Float of float
```

```
  | String of string                                    (* ".." で囲まれた文字列 *)
  | Char of char                                        (* '.' で囲まれた文字 *)
```

8.4 コンパイルと実行

以下のプログラムをファイル hello.ml に書く.

```
let hello () =
  Printf.printf "What is your name? ";
  let s = read_line () in
  Printf.printf "Hello %s!\n" s ;;

hello ();;
```

シェルでそれをコンパイルする.

```
$ ocamlc -c hello.ml
$ ocamlc hello.cmo -o hello
```

そして，それを実行する.

```
$ ./hello
What is your name? Jacques
Hello Jacques!
```

プログラムを複数のファイルに分割した場合，他のファイルの中の値の名前の前にファイル名 (頭文字を大文字にして) を付けなければならない.

```
(* hello.ml *)
let hello () =
  Printf.printf "What is your name? ";
  let s = read_line () in
  Printf.printf "Hello %s!\n" s ;;


(* hello2.ml *)
Hello.hello ();;
```

正しい順番でコンパイルして，リンクしなければならない.

```
$ ocamlc -c hello.ml
$ ocamlc -c hello2.ml
$ ocamlc hello.cmo hello2.cmo -o hello2
```

注：C と同様，リンク命令の中にコンパイルが入ってもいい.

```
$ ocamlc hello.ml hello2.ml -o hello2
```

8.5 デバッグ

プログラムが期待どおりに動かないときには，途中状態を知りたいことが多い．C なら printf を使うことが多いが，ocaml ではデータ構造の受け渡しが多いので，何をどう見せればいいか分かりにくい．ここに二つの方法を説明する.

`#trace` 定義された関数の呼び出し状況を調べるために，`#trace` という命令が用意されている．関数がの引数と返り値が印刷される．

```
# let rec fact n =
    if n = 0 then 1 else n * fact (n-1) ;;
val fact : int -> int = <fun>
# #trace fact;;
fact is now traced.
# fact 2;;
fact <-- 2
fact <-- 1
fact <-- 0
fact --> 1
fact --> 1
fact --> 2
- : int = 2
```

ただし多相型は印刷できないので，可能なときは元の定義に多相でない型を手で書かないといけない．

```
# #trace List.rev;;
List.rev is now traced.
# List.rev [1;2];;
List.rev <-- [<poly>; <poly>]
List.rev --> [<poly>; <poly>]
- : int list = [2; 1]
```

多引数の関数の扱いも弱い．

```
# let t0' = subst ["a", int] t0 ;;
subst <-- [("a", Named ("int", []))]
subst --> <fun>
subst* <-- Funct (Var "a", Var "b")
subst <-- [("a", Named ("int", []))]
subst --> <fun>
subst* <-- Var "b"
subst* --> Var "b"
subst <-- [("a", Named ("int", []))]
subst --> <fun>
subst* <-- Var "a"
subst* --> Named ("int", [])
subst* --> Funct (Named ("int", []), Var "b")
val t0' : etype = Funct (Named ("int", []), Var "b")
```

記録 元のプログラムに手を入れてもいいなら，記録を取るようにそれを変更することができる．

```
let tr = ref []
let rec subst (bl : bindings) t =
  let r = match t with
      ...
  in tr := (bl, t, r) :: !tr; r
val subst : bindings -> etype -> etype = <fun>
# tr := [] ;;
# subst ["a", int] t0;;
# !tr;;
- : (bindings * etype * etype) list =
[([("a", Named ("int", []))], Funct (Var "a", Var "b"),
```

```
  Funct (Named ("int", []), Var "b"));
 ([("a", Named ("int", []))], Var "a", Named ("int", []));
 ([("a", Named ("int", []))], Var "b", Var "b")]
```

しかし，これでは計算の構造が見えない．多相ヴァリアントを使えば，計算の構造を保ったトレースが書ける．

```
let reset tr = tr := [[]]
let enter tr x =
  match !tr with
    tr1 :: rem -> tr := [] :: (x :: tr1) :: rem
  | _ -> failwith "enter"
val enter : 'a list list ref -> 'a -> unit
let leave tr f =
  match !tr with
    tr1 :: (_::tr2) :: rem -> tr := (f (List.rev tr1) :: tr2) :: rem
  | _ -> failwith "leave"
val leave : 'a list list ref -> ('a list -> 'a) -> unit
let tr = ref [[]]

let rec subst (bl : bindings) t =
  enter tr (`Subst'(bl,t));                          (* 例外のために引数を記録する *)
  let r = match t with
    ...
  in
  leave tr (fun tr1 -> `Subst(bl,t,r,tr1)); r                (* 結果を記録する *)
val subst : bindings -> etype -> etype
# let t0' = subst ["a", int] t0;;
val t0' : etype = Funct (Named ("int", []), Var "b")
# !tr;;
- : (_[> `Subst of bindings * etype * etype * 'a list
       | `Subst' of bindings * etype ] as 'a) list list
= [[`Subst
    ([("a", Named ("int", []))], Funct (Var "a", Var "b"),
     Funct (Named ("int", []), Var "b"),
     [`Subst ([("a", Named ("int", []))], Var "b", Var "b", []);
      `Subst ([("a", Named ("int", []))], Var "a", Named ("int", []), [])])]]
```

この方法を使うと，同じ tr を複数の関数で共有できる．

実習課題 (8 回目)

1. 入力する数値の合計を計算する関数を定義せよ．

```
# mysum () ;;
1
2
3
0
total = 6
- : unit = ()
```

2. 同じ関数で入力を一行で取るようにせよ．

```
# mysum2 () ;;
1 2 3
total = 6
- : unit = ()
```

ヒント：入力を read_line で読んだ後，Stream.of_string を使う．

3. mysum と mysum2 をコンパイルして，シェルで実行できるようにせよ．

4. 前回の expr 型の値 (プログラム) の自由変数を計算する関数を定義せよ．
   val free_ids : expr -> string list
   定義は free や free_env に倣っていいが，Fun(x, e) のような場合では，例え x が e の中の自由変数であっても，Fun(x, e) の自由変数にならないことに注意．Let や Letrec のときにも同様の規則が適用される．

5. unify や free_ids の動きを#trace や他の記録方法で調べよ．