Jacques Garrigue, 2012 年 11 月 14 日

GUI とグラフィックス

11 LablTk の利用

LablTk ライブラリーを使って，GUI を作る．
ocaml で LablTk を使うには以下のコマンドを入力すればいい．

```
# #directory "+labltk";;                          (* directory を path に追加 *)
# #load "labltk.cma";;                             (* ライブラリーを読み込む *)
# open Tk;;                                                  (* 定義を輸入 *)
```

ocamlc で使うには，

```
$ ocamlc -I +labltk labltk.cma ...
```

をコマンドに追加し，各ファイルの先頭に open Tk を追加する．(ocaml でも上記のオプションを渡すと#directory および#load の代わりになる)
また，ocamlbrowser で定義を見るために，ocamlbrowser -I +labltk で起動すればいい．

簡単な例

```
open Tk;;
let top = openTk ();;                                  (* この openTk は関数名 *)
val top : Widget.toplevel Widget.widget = <abstr>   (* top はウィンドウである *)
let b = Button.create top ~text:"Hello World!";;    (* top の中にボタンを作成 *)
val b : Widget.button Widget.widget = <abstr>
pack [b];;                                                (* ボタンを配置する *)
- : unit = ()
mainLoop();;                                                       (* 実行 *)
```

これでウィンドウが表われ，その中に Hello World! と書かれたボタンが表示される．mainLoop() は終わらない．このプログラムを ocamlc -I +labltk labltk.cma hello.ml -o hello でコンパイルしてもいい．
ocaml で操作している場合，以下の用に続けられる．

```
<Ctrl-c>Interrupted                                 (* Emacs では C-cC-c で止める *)
# Button.configure b ~command:(fun () -> print_endline "Hello!");;
- : unit = ()                              (* ボタンが押されると command が呼ばれる *)
# mainLoop();;
Hello!
Hello!
Hello!                                              (* ボタンを押す度に出力される *)
<Ctrl-c>Interrupted
# destroy top ;;                                    (* これでウィンドウが消える *)
- : unit = ()
```

省略可能引数　上記のプログラムで使われる Button.create は以下の型をもっている (Button.configure も返り値以外は同じ型である)

```
# Button.create ;;
- : ?name:string ->                    ?activebackground:Tk.color ->
    ?activeforeground:Tk.color ->      ?anchor:Tk.anchor ->
    ?background:Tk.color ->            ?bitmap:Tk.bitmap ->
    ?borderwidth:int ->                ?command:(unit -> unit) ->
    ?cursor:Tk.cursor ->               ?default:Tk.state ->
    ?disabledforeground:Tk.color ->    ?font:string ->
    ?foreground:Tk.color ->            ?height:int ->
    ?highlightbackground:Tk.color ->   ?highlightcolor:Tk.color ->
    ?highlightthickness:int ->         ?image:[< Tk.image ] ->
    ?justify:Tk.justification ->       ?padx:int ->
    ?pady:int ->                       ?relief:Tk.relief ->
    ?state:Tk.state ->                 ?takefocus:bool ->
    ?text:string ->   ?textvariable:Textvariable.textVariable ->
    ?underline:int ->                  ?width:int ->
    ?wraplength:int -> 'a Widget.widget -> Widget.button Widget.widget
```

しかし，適用するときには，全ての引数を渡していない．?で始まる引数は省略可能引数で，それを渡さない場合には既定値が使われる．それを渡す場合，引数の前に label:という形に引数の名前を書く．名前があるので，引数の順番は自由になる．

同様に，型の中に label:と書かれる引数もあるが，それは名前付きでありながら省略可能ではない．ただし，渡すときに名前を書くので，違う順番で渡してもいい．

亀グラフィックス

キャンバスの上にグラフィックスを描く．

```
open Tk ;;
let top = openTk() ;;
val top : Widget.toplevel Widget.widget = <abstr>
let c = Canvas.create top ~width:640 ~height:480 ~background:`White ;;
val c : Widget.canvas Widget.widget = <abstr>
pack [c];;

let clear () =                                  (* キャンバスの上のものを全て消す *)
  Canvas.delete c [`Tag "all"] ;;
val clear : unit -> unit

type turtle =                                   (* "亀"の型．値が変更できる *)
  {mutable x: float; mutable y: float; mutable angle: int; mutable pen: bool}
let turtle = {x = 320.; y = 240.; angle = 90; pen = true} ;;

let pi = acos (-1.) ;;                          (* πを自分で計算する *)
val pi : float = 3.14159265358979312
let to_rad deg = float deg *. pi /. 180. ;;
val to_rad : int -> float
let round f = truncate (f +. 0.5) ;;            (* 四捨五入の実装 *)
val round : float -> int
let round_point (x, y) = (round x, round y)
val round_point : float * float -> int * int

let rel_pos dx dy =                             (* 亀から見た位置の変換 *)
  let a = to_rad turtle.angle in
```

```
  (turtle.x +. float dx *. cos a +. float dy *. sin a,
   turtle.y -. float dx *. sin a +. float dy *. cos a)
val rel_pos : int -> int -> float * float

let draw_turtle () =                                          (* 亀を描く *)
  let to_screen dx dy = round_point (rel_pos dx dy) in
  Canvas.delete c [`Tag "turtle"];                          (* 前の亀を消す *)
  ignore (Canvas.create_polygon c ~tags:["turtle"] ~fill:`Blue
            ~xys:[to_screen 0 5; to_screen 0 (-5); to_screen 9 0]);
  update () ;;                                             (* すぐに描画を行う *)
val draw_turtle : unit -> unit

let forward d =                                             (* 前に d 歩進む *)
  let (x', y') = rel_pos d 0 in
  if turtle.pen then ignore begin       (* ignore は関数の返り値を unit にする *)
    Canvas.create_line c
      ~xys:[round_point (turtle.x, turtle.y); round_point (x', y')]
  end;
  turtle.x <- x'; turtle.y <- y';
  draw_turtle () ;;
val forward : int -> unit

let left a =                                               (* 左に a 度回転する *)
  turtle.angle <- (turtle.angle + a) mod 360;
  draw_turtle () ;;
val left : int -> unit
let right a = left (-a);;                                  (* 右に a 度回転する *)
val right : int -> unit
```

上の関数で実験してみよう.

```
# update () ;;                    (* mainLoop を待たずにウィンドウが表示される *)
# forward 100 ;;                                            (* 線が描かれる *)
# let square d = for i = 1 to 4 do forward d; right 90 done ;;
val square : int -> unit = <fun>
# square 100;;                                             (* 四角が描かれる *)
```

レコード型 type turtle = ... はレコード型を定義している．各要素に名前が付く組と同義である．直和型と同様，フィールド名が定義された型と関連付けられるので，他の型では使えない．さらに，各フィールドに対して，mutable を指定すると可変になる．そのフィールドの値に ref 型を使うのと同じ効果があるが，OCaml では mutable の方が基礎になる．

```
type 'a ref = {mutable contents: 'a} ;;
let (!) r = r.contents ;;
let (:=) r x = r.contents <- x ;;
```

多相ヴァリアント型 clear 関数で使われる `Tag は多相ヴァリアントである．多層ヴァリアントは通常の直和型と同じ使い方をするが，最大の特徴は型宣言が要らないことである．例えば `Tag "all"という値に [> `Tag of string] という，値の形を表した型が付く．

LablTk で多層ヴァリアントを使う理由は二つある．一つは，LablTk は大くのモジュールからできているが，多層ヴァリアントだと型がどこで定義されているかを気にする必要はない (そもそも定義されていなくてもいい)．もう一つは，同じ構成子 (場合名) を複数の異なる文脈で使いたいことがある．例えば，index 系の型は LablTk で 7 種あるが，ほとんどの場合では `End が使える．

GUI で亀グラフィックス

亀グラフィックスのプログラムを続ける.

```
let buttons = Frame.create top ;;
let entry = Entry.create buttons ~width:10 ;;
let reset_button = Button.create buttons ~text:"Argument:"
    ~command:(fun () -> Entry.delete_range entry ~start:(`At 0) ~stop:`End);;

let get_arg () = try int_of_string (Entry.get entry) with _ -> 0 ;;
val get_arg : unit -> int
let call (cmd : int -> unit) () = cmd (get_arg ()) ;;
val call : (int -> unit) -> unit -> unit
let fw_button = Button.create buttons ~text:"Forward" ~command:(call forward);;
let left_button = Button.create buttons ~text:"Left" ~command:(call left);;
let right_button = Button.create buttons ~text:"Right" ~command:(call right);;

(* widget を並べる. `Left では横に, `Bottom では縦に *)
pack [coe reset_button; coe entry]            (* coe で型が any widget になる *)
    ~side:`Left ;;
pack [fw_button; left_button; right_button] ~side:`Left ;;
pack [buttons] ~side:`Bottom ~anchor:`W ;;
mainLoop() ;;                                  (* 全てが pack されたら実行する *)
```

テキストフィールドに引数を入れて, 命令のボタンをクリックすると, 亀が動く.

様々な拡張ができる. 例えば, ペンを付けたり外したりする.

```
let pen_button = Checkbutton.create buttons ~text:"Pen"
    ~command:(fun () -> turtle.pen <- not turtle.pen) ;;
if turtle.pen then Checkbutton.select pen_button ;;
pack [pen_button] ~side:`Left;;
```

練習問題 11.1 *1.* 上記の亀グラフィックスのプログラムに線を消し, 亀を元の位置に戻すボタンを追加せよ.

2. 関数 call を次のように変更する.

```
let list : ((int -> unit) * int) list ref = ref []
let record = ref false
let call cmd () =
  let arg = get_arg () in
  if !record then list := (cmd, arg) :: !list;
  cmd arg ;;
```

これを利用し, 使った命令の列を記録し, それを後再実行できるようにせよ. 具体的には, *Record, Stop, Play* というボタンを追加し, *Play* を押すと, 前の *Record* と *Stop* の間に使った命令が再実行されるようにする.

3. Canvas.create_line の返り値の型と Canvas.delete の型を見て, 最後の命令をキャンセルする *Undo* ボタンを追加せよ.

4. Label.create と Label.configure を使って, 亀の現在の座標を表示させる.