Programmation impérative et fonctionnelle avec OCaml
TP 6

Objectifs :
— Types algébriques.
— Interprétation.
— Utilisation de la bibliothèque graphique.

Interpréteur de Logo

L'objectif de cet exercice est l'écriture d'un interpréteur pour un sous-ensemble (très) restreint du langage Logo.

Le Logo est un langage de programmation éducatif destiné à des enfants. Il a été conçu par Wally Feurzeig et Seymour Papert au laboratoire d'Intelligence Artificielle du MIT en 1967, en s'inspirant des théories de l'apprentissage constructiviste du psychologue suisse Jean Piaget.

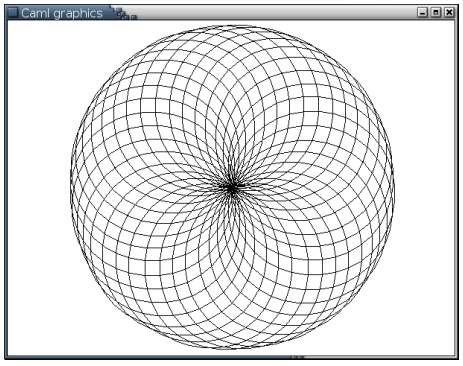
Il permet, en particulier, de dessiner dans le plan en décrivant un enchaînement de déplacements. Le « crayon » est habituellement désigné comme la « tortue ». À chaque instant, cette dernière a une position dans le plan et un cap. Le dessin s'effectue par une séquence d'ordres donnés à la tortue : « avance de 10 unités », « tourne à droite de 90 degrés », « répète 10 fois ce qui suit »... Par exemple, le dessin d'un carré (de côté 100) s'écrira en Logo :

```
repeat 4 [forward 100; right 90]
```

où `repeat` exécute une boucle bornée (ici, 4 itérations), `forward` avance et `right` tourne à droite (angle en degrés).

On veut ici pouvoir exécuter le programme `exemple` suivant (`ne` et `sz` sont des variables) qui produit le dessin de la figure 1 :

```
repeat ne [right (360 / ne);
           repeat ne [right (360 / ne); forward sz]]
```



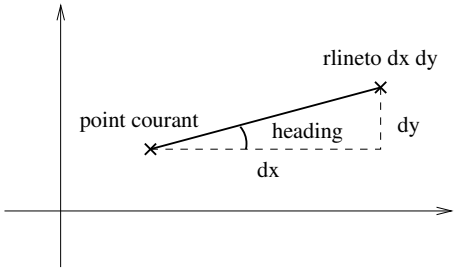


FIGURE 1 – À gauche, le dessin produit par l'exemple avec `ne=36` et `sz=20`. À droite, utilisation de `Graphics.rlineto`.

1. Les paramètres des commandes Logo sont des expressions qui peuvent être :
   — des entiers ;
   — des variables ;
   — des opérations arithmétiques binaires sur deux sous-expressions (on peut se limiter ici à la division, seule utile pour l'exemple).

   Écrire le type `expression` permettant de représenter ces expressions.
2. Représenter l'expression `(360 / ne)` avec ce type.
3. On va considérer uniquement les instructions Logo suivantes :
   — `right` qui prend en paramètre une expression désignant l'angle (en degrés) de la rotation vers la droite ;

— `forward` qui prend en paramètre une expression désignant la distance à parcourir dans la direction du cap courant ;
— `repeat` qui prend en paramètres une expression désignant le nombre d'itérations et l'instruction à répéter ;
— un `bloc` (ou séquence) d'instructions.

Écrire le type `instruction` permettant de représenter ces instructions.

4. Représenter le programme `exemple` donné ci-dessus avec ce type.
5. Pour prendre en compte les variables, on utilise un *environnement* associant à chaque variable une valeur entière. On pourra le représenter par une liste de couples `(variable, valeur)` et utiliser la fonction `List.assoc` pour y accéder. Écrire la fonction :

   ```
   val eval : (string * int) list -> expression -> int
   ```

   qui évalue une expression en traitant le cas des variables.
6. Pour dessiner, nous allons utiliser la bibliothèque `Graphics` d'OCaml. Les fonctions de cette bibliothèque utilise un *point courant* (la position de la tortue en Logo). Pour cette question, nous ne nous servirons que de la fonction :

   ```
   val rlineto : int -> int -> unit
   ```

   `Graphics.rlineto dx dy` dessine un segment depuis le point courant jusqu'au point courant translaté de `(dx, dy)` (voir la figure 1). Nous allons utiliser également un *cap courant*, implémenté par une référence locale (ici un cap en degrés entier) :

   ```
   let play = fun env prog ->
     let heading = ref 0 in
     let rec play_rec = fun instr ->
       ... in
     play_rec prog;;
   ```

   Écrire la fonction `play` qui interprète une instruction Logo :

   ```
   val play : (string * int) list -> instruction -> unit
   ```

   Remarques :
   — Il n'est pas nécessaire de définir de référence locale pour représenter la position du point courant, car le module `Graphics` le maintient à jour implicitement après chaque opération graphique.
   — Les fonctions trigonométriques (`sin` et `cos`) s'appliquent à des radians exprimés en nombre flottant, et non des degrés entiers.
   — On compilera avec la commande suivante pour pouvoir utiliser le module `Graphics` :
   `ocamlc -o tp6 graphics.cma tp6.ml`
   — Il est en général **fortement déconseillé** de mélanger les styles récursif et impératif (modification de références), mais c'est la démarche la plus simple sur cet exemple qui manipule explicitement un **état**, i.e. le cap courant et la position du point courant.
7. On testera les fonctions écrites avec le programme suivant :

   ```
   let () =
     Graphics.open_graph ""; (* ouverture de la fenêtre graphique *)
     Graphics.moveto 300 225; (* initialisation du point courant *)
     let env = [("ne", 36); ("sz", 20)] in
     play env exemple;
     ignore (Graphics.read_key ());; (* attend qu'une touche soit pressée *)
   ```

8. *Question subsidiaire : Dans certains contextes, l'utilisation de références n'est pas possible (e.g. pour gérer l'environnement lors de la compilation de sous-blocs d'instructions qui peuvent déclarer de nouvelles variables) et il faut que l'interpréteur* `play` *prenne un « état » (i.e. la direction courante) en paramètre et renvoie cet état (éventuellement modifié, en l'occurrence par l'instruction* `Right`*). Cet état sera ainsi passé d'appel récursif en appel récursif dans le style récursif terminal. Écrire une fonction* `play2` *sans utiliser de référence.*
   Si le module `Graphics` *ne gérait pas implicitement un point courant, ce point devrait également être géré dans l'état passé en paramètre. Écrire un type structure* `state` *pour représenter ce nouvel état (direction* et *point courant) et une nouvelle fonction* `play3` *qui utilise la fonction* `draw_poly_line` *du module Graphics plutôt que* `rlineto` *(donc sans utiliser le point courant de la fenêtre graphique).*