Interpréteur LOGO en OCaml

Maxence Ahlouche

Table des matières

1 Présentation du projet

Le but de ce projet était de programmer, en monôme, les analyseurs syntaxiques et sémantiques d'un interpréteur du langage LOGO, le reste des fonctions nécessaires étant fourni. Ce rapport présente la démarche de conception et de programmation qui a été suivie pour ce projet.

Ce programme a été réalisé, pour les parties non fournies par les professeurs, en programmation purement fonctionnelle, à l'aide du langage OCaml. Il a pour objectif de simuler le comportement d'une tortue LOGO, qui servait à initier à la programmation de manière ludique.

Le langage LOGO tel que nous l'avons étudié ne comporte que cinq instructions élémentaires :
- `JUMP`, qui permet de déplacer la tortue sans dessiner ;
- `MOVE`, qui déplace la tortue tout en dessinant ;
- `ROTATE`, qui permet de diriger la tortue ;
- `COLOR`, qui change la couleur du crayon ;
- `CALL`, qui appelle une fonction définie auparavant.

Il offre également plusieurs structures de contrôle :
- `IF (...) THEN (...) ELSE (...)`, qui permet d'exécuter certaines instructions seulement si une certaine condition est vérifiée ;
- `REPEAT (...)`, qui permet de répéter un bloc d'instructions autant de fois que souhaité ;
- `DEF (...)`, la définition de procédures, qui introduit notamment la possibilité de dessiner facilement des fractales.

Les fonctions développées dans le cadre du projet transforment la liste de mots du programme LOGO (déjà parsée) en une liste de commandes interprétables par la fonction d'affichage.

2 Le programme

Cette section couvre les différentes étapes de réalisation de ce projet, selon leur ordre chronologique.

2.1 Conception

L'étape de conception a été très courte, du fait que la structure du programme était donnée avec le sujet. Elle consistait principalement en la définition des types nécessaires au passage des informations entre les analyseurs syntaxique et sémantique. J'ai donc commencé par définir mon type `programme`, qui était composé d'une liste de `procedure` et d'un arbre de `sous_programme` correspondant au programme principal. Comme j'avais l'intention de produire un résultat, même basique, le plus tôt possible, je ne me suis pas occupé de définir le type `procedure`, et me suis contenté de le déclarer. Un `sous_programme` peut être soit une instruction élémentaire (Move, Jump, Rotate ou Color), soit une structure de contrôle :

- `If` : il est alors composé d'une condition, d'un premier bloc d'`instruction` qui sera exécuté si la condition est vérifiée, et un second bloc d'`instruction` qui sera exécuté dans le cas contraire.
- `Repeat` : celui-ci est formé d'une expression indiquant le nombre de fois que le bloc d'instructions devra être exécuté, et du bloc d'`instruction` en question.
- `Call` : le nom de la fonction appelée et la liste des valeurs des paramètres passés à cette fonction forment ce `sous_programme`.

Plus tard, j'ai défini le type `procedure`, qui est composé du nom de la procédure, de sa liste de paramètres, et de l'arbre d'instructions qui lui correspond.

2.2 Développement

2.2.1 L'analyseur syntaxique

Le rôle de l'analyseur syntaxique est de transformer la liste de mots générée par l'analyseur lexical en un programme interprétable par l'analyseur sémantique.

L'analyseur syntaxique de mon programme, lorsqu'il rencontre le mot-clé `DEF`, interprète la procédure (fonction `interprete_procedure`), et s'appelle récursivement avec la liste de mots privée de la définition de la procédure. Lorsqu'il rencontre le `BEGIN` du programme principal, il construit l'arbre de `sous_programme` correspondant (fonction `interprete_instructions`).

La fonction `interprete_procedure` s'appelle récursivement tant qu'elle trouve des paramètres, puis construit l'arbre de `sous_programme` de la définition (fonction `interprete_instructions`), et renvoie la procédure ainsi créée.

La fonction `interprete_instructions` interprète la liste de mots qu'on lui passe en paramètre, mais uniquement jusqu'au `END` de même niveau que la première instruction de la liste, c'est-à-dire le premier bloc d'instructions de la liste, y compris les sous-blocs. Cette particularité permet de générer très facilement les `sous_programme` `If`, `Repeat`, ainsi que les définitions de procédures. Cette fonction renvoie la liste des mots qui restent à interpréter, ainsi que l'arbre de `sous_programme` déjà interprété.

2.2.2 L'analyseur sémantique

En commençant à travailler sur l'analyseur sémantique, je me suis rendu compte que mes types `procedure` et `sous_programme` correspondaient très exactement aux types `definition` et `instruction` pré-déclarés. J'ai donc commencé par convertir les premiers en les seconds.

Le rôle de l'analyseur sémantique est de générer une liste de commandes (`cmd`) interprétables par la fonction de dessin, à partir d'un `programme`.

L'analyseur sémantique de mon programme fait appel à la fonction `eval_instructions` avec les paramètres initiaux, c'est-à-dire un `environnement` (liste de `procedure` et liste d'associations `parametre`/valeur) ne contenant que les procédures, et l'`etat` (coordonnées et angle de visée) initial du curseur.

La fonction `eval_instructions` est chargée de générer la liste des `cmd`. Pour ce faire, elle parcourt l'arbre d'`instruction` du `programme`, et utilise la fonction `execute_instruction`, qui calcule le nouvel état du curseur, à partir de son état courant et de l'`instruction` à exécuter. Elle se charge aussi de remplacer les `If`, `Repeat` et `Call` en le sous-arbre d'instructions approprié, et l'évalue également. Enfin, elle enrichit et dépile l'environnement au fur et à mesure qu'il est nécessaire de le faire.

2.3 Tests

2.3.1 La courbe du dragon

```
1  DEF dragonendroit (l, n)
2  BEGIN
3          IF (n <= 0) THEN
4          BEGIN
5                  MOVE (l)
6          END
7          ELSE
8          BEGIN
9                  ROTATE (45)
10                 CALL dragonendroit ((l*0.707), (n-1))
11                 ROTATE (-90)
12                 CALL dragonenvers ((l*0.707), (n-1))
13                 ROTATE (45)
14         END
15 END
16 DEF dragonenvers (l, n)
17 BEGIN
18         IF (n<=0) THEN
19         BEGIN
20                 MOVE (l)
21         END
22         ELSE
23         BEGIN
24                 ROTATE (-45)
25                 CALL dragonendroit ((l*0.707), (n-1))
26                 ROTATE (90)
27                 CALL dragonenvers (l*0.707, n-1)
28                 ROTATE (-45)
29         END
30 END
31 BEGIN
32         COLOR (255,100, 0) (* fancy color *)
33         ROTATE (90)
34         JUMP (100)
35         ROTATE (-90)
```

```
36          JUMP (100)
37          CALL dragonendroit (300, 17)
38  END
```

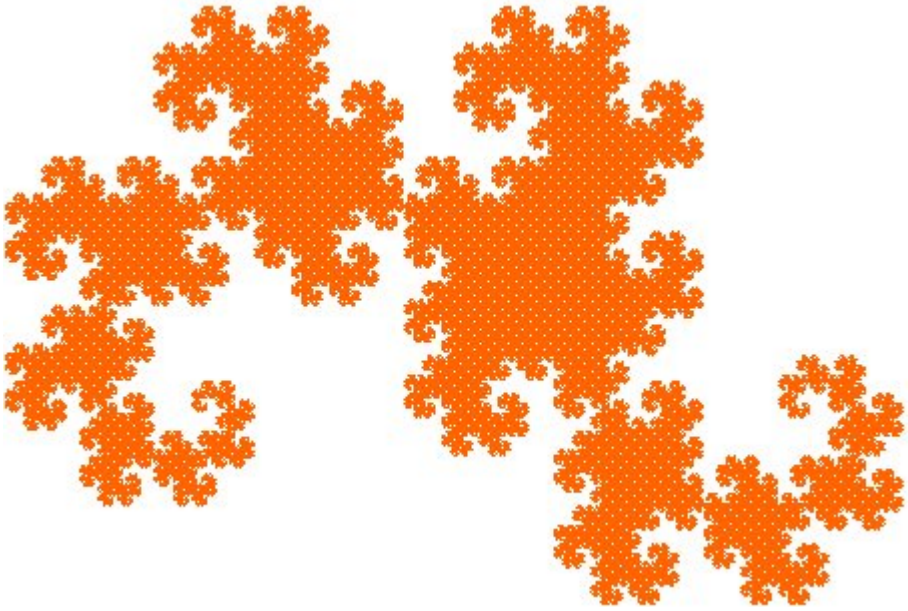

FIGURE 1 – Le résultat donné par l'exécution de dragon.logo

2.3.2 L'éponge de Menger

```
1   DEF menger (n, l, h, angle)
2   BEGIN
3           IF (n>0) THEN
4           BEGIN
5               (* affichage des 3 sous-carres du bas *)
6           REPEAT (3)
7               BEGIN
8               CALL menger ((n-1), (l/3), (h/3), (angle))
9                   JUMP (l/3)
10          END
11          (* retour au point d'origine, pour ne pas perturber les appels
                precedents a la fonction *)
12              ROTATE (180)
13          JUMP (l)
14
15          (* on monte d'une ligne *)
16              ROTATE (180+angle)
17          JUMP (h/3)
18              ROTATE (-angle)
19
20          (* on affiche le carre milieu-gauche *)
```

```
21              CALL menger ((n-1), (l/3), (h/3), angle)
22          (* on se place au niveau du carre milieu droit *)
23              JUMP ((2*l)/3)
24          (* on affiche le carre milieu droit *)
25              CALL menger ((n-1), (l/3), (h/3), angle)
26
27                  (* on retourne a l'origine *)
28                  ROTATE (180)
29                  JUMP ((2*l)/3)
30                  ROTATE (180+angle)
31                  JUMP (h/3)
32                  ROTATE (-angle)
33
34                  (* on affiche les 3 carres du haut *)
35                  REPEAT (3)
36                  BEGIN
37                          CALL menger ((n-1), (l/3), (h/3), angle)
38                          JUMP (l/3)
39                  END
40                  (* Retour a l'origine *)
41                  ROTATE (180)
42                  JUMP (l)
43                  ROTATE (angle)
44                  JUMP ((2*h)/3)
45                  ROTATE (180-angle)
46          END
47          ELSE
48          BEGIN
49          (* affichage du carre *)
50              REPEAT (2)
51              BEGIN
52                      MOVE (l)
53                      ROTATE (angle)
54                      MOVE (h)
55                      ROTATE (180-angle)
56              END
57          END
58  END
59  BEGIN
60          COLOR (20, 20, 255)
61          ROTATE (90)
62          JUMP (150)
63          ROTATE (-90-20)
64      JUMP (10)
65          CALL menger ((4), (400), (400), (100))
66          JUMP (400)
67          ROTATE (30+20)
68          CALL menger ((4), (100), (400), (50))
69          ROTATE (180-30)
70          JUMP (308)
71          ROTATE (-90)
72          JUMP (530)
73          ROTATE (-90-20)
```

```
74          CALL menger ((4), (400), (100), (50))
75  END
```

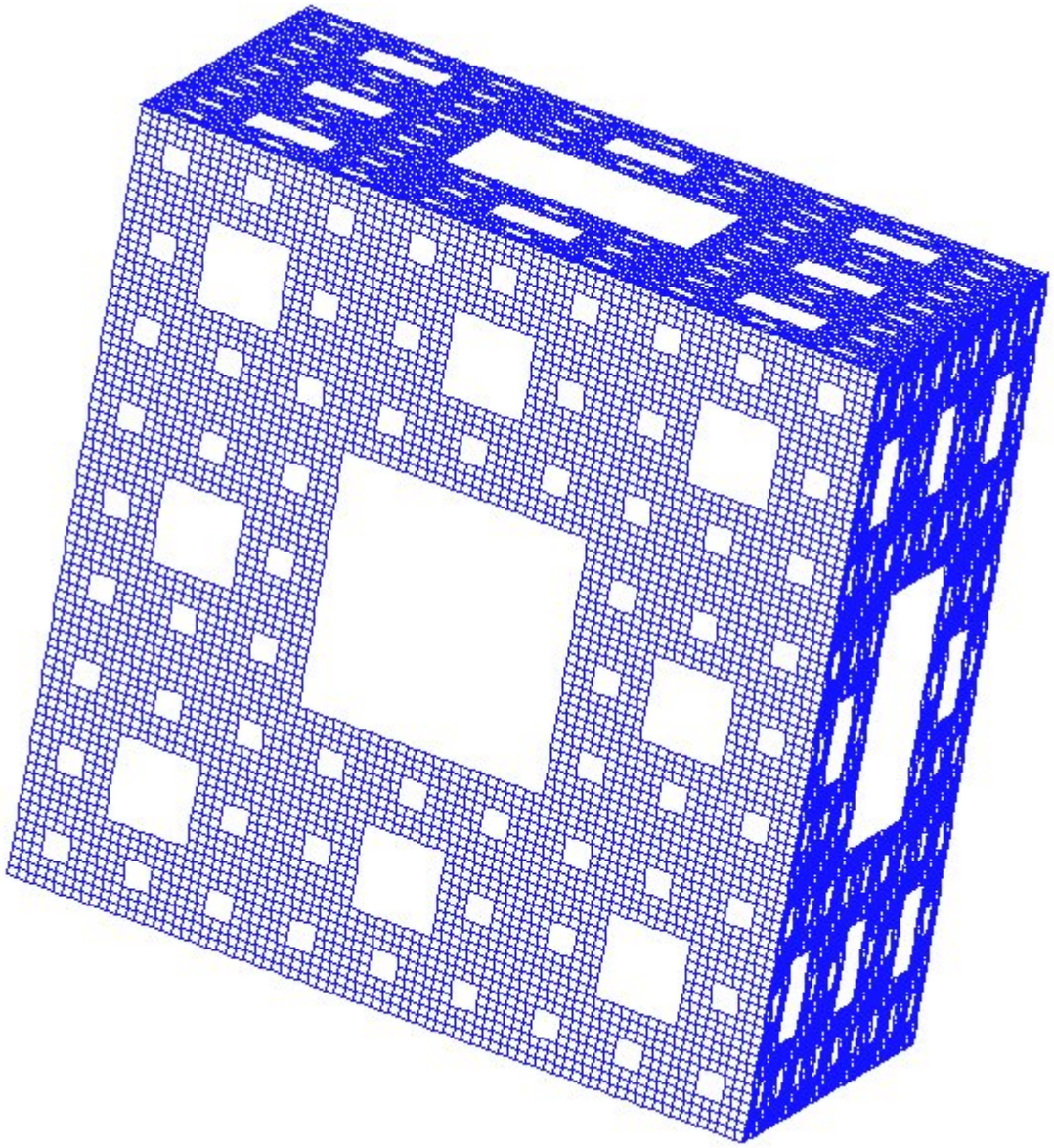

FIGURE 2 – Le résultat donné par l'exécution de menger.logo

3 Le code source

3.1 logo_types.ml

```
1  (* type des expressions arithmetiques *)
2  type expr =
3    Plus of expr * expr
4  | Moins of expr * expr
5  | Div of expr * expr
6  | Mult of expr* expr
7  | Const of float
8  | Var of string
9  | Cosinus of expr
10 | Sinus of expr
11 | Tangente of expr;;
12 (* type des expressions booleennes*)
13 type test =
14   Equal of expr * expr
15 | InfEq of expr * expr
16 | And   of test * test
17 | Or    of test * test
18 | Not   of test;;
19 (* type des mots cles du langage *)
20 type mot =
21   IF
22 | THEN
23 | ELSE
24 | BEGIN
25 | END
26 | REPEAT
27 | DEF
28 | CALL
29 | ROTATE
30 | MOVE
31 | JUMP
32 | COLOR
33 | FIN
34 | EXPR of expr
35 | TEST of test
36 | IDENT of string;;
37 (* type des commandes a generer pour afficher les figures *)
38 type cmd =
39         Change_color of Graphics.color
40         |Moveto of float*float
41         |Jumpto of float*float
42 ;;
43
44 (* type parametre *)
45 type parametre = string;;
46 (* type instruction elementaire sous forme d'arbre *)
47 type instruction =
48   | Move of expr
49   | Jump of expr
50   | Rotate of expr
51   | Color of expr * expr * expr
52   | If of (test * instruction list * instruction list)
53   | Repeat of (expr * instruction list)
```

```
54    | Call of (string * expr list);;
55
56  (* type procedure : nom de la procedure * liste des params * liste des sous
        -programmes *)
57  type definition = string * parametre list * instruction list;;
58  (* Type programme : liste des procedures * liste des instructions de
        premier niveau (d'indentation) *)
59  type programme = definition list * instruction list;;
60
61  (* type des environnements : une liste de variables et une liste de
        procedures *)
62  type environnement = (parametre * expr) list * definition list;;
63  (* type des etats du systeme : la position du curseur et son angle de visee
         en degres *)
64  type etat = (float*float) * float;;
```

3.2 logo_analyseur_syntaxique.ml

```
1   open Logo_types;;
2
3   (*
4    * fonction get_params
5    * But : recuperer les valeurs des parametres passes lors d'un call.
         Logiquement, devrait etre une fonction auxiliaire de
         interprete_instructions, mais je la mets la pour l'avoir une seule
         fois dans l'env
6    * Entree : la liste de mots
7    * Precondition : prog correct
8    * Sortie : la nouvelle liste de mots et la liste des params
9    * Postcondition : -
10   *)
11  let rec get_params lmot =
12    match lmot with
13    (* tant qu'on trouve des parametres, on les recupere *)
14    | EXPR(valeur_param)::lmot' -> let (new_lmot, valeurs_params) =
          get_params lmot'
15                                   in (new_lmot, valeur_param::
                                         valeurs_params)
16    (* fin des parametres *)
17    | _ -> (lmot, []);;
18
19  (*
20   * fonction interprete_instructions
21   * But : interpreter une liste de mots jusqu'au END de meme niveau que la
         premiere instruction
22   * Entree : la liste de mots
23   * Precondition : la liste contient au moins un END
24   * Sortie : la nouvelle liste de mots a interpreter et l'arbre d'
         instructions correspondant a la liste de mots jusqu'au END
25   * Postcondition : -
26   *)
27  let rec interprete_instructions lmot =
```

```
28    match lmot with
29    (* END: on remonte d'un niveau dans l'arbre *)
30    | END::lmot' -> (lmot', [])
31    (* BEGIN: on interprete la suite *)
32    | BEGIN::lmot' -> interprete_instructions lmot'
33    (* MOVE: on ajoute l'instruction Move et on interprete la suite *)
34    | MOVE::EXPR(e)::lmot' ->
35        let (new_lmot, instructions) = interprete_instructions lmot' in
36        (new_lmot, Move(e)::instructions)
37    (* JUMP: on ajoute l'instruction Jump et on interprete la suite *)
38    | JUMP::EXPR(e)::lmot' ->
39        let (new_lmot, instructions) = interprete_instructions lmot' in
40        (new_lmot, Jump(e)::instructions)
41    (* ROTATE: on ajoute l'instruction Rotate et on interprete la suite *)
42    | ROTATE::EXPR(e)::lmot' ->
43        let (new_lmot, instructions) = interprete_instructions lmot' in
44        (new_lmot, Rotate(e)::instructions)
45    (* COLOR: on ajoute l'instruction Color et on interprete la suite *)
46    | COLOR::EXPR(r)::EXPR(g)::EXPR(b)::lmot' ->
47        let (new_lmot, instructions) = interprete_instructions lmot' in
48        (new_lmot, Color(r, g, b)::instructions)
49    (* IF: on ajoute l'instruction If constituee du test et des 2 sous-blocs
          d'instructions *)
50    | IF::TEST(t)::THEN::BEGIN::lmot' ->
51        let (new_lmot_if, instructions_if) = interprete_instructions lmot' in
52        let (new_lmot_else, instructions_else) = interprete_instructions
              new_lmot_if in
53        let (new_lmot, instructions) = interprete_instructions new_lmot_else
              in
54        (new_lmot, If(t, instructions_if, instructions_else)::instructions)
55    (* ELSE: on interprete simplement le bloc, il sera traite dans le cas if
          *)
56    | ELSE::BEGIN::lmot' -> interprete_instructions lmot'
57    (* REPEAT: on ajoute l'instruction Repeat constituee du nombre de
          repetitions et du sous bloc d'instructions *)
58    | REPEAT::EXPR(e)::BEGIN::lmot' ->
59        let (new_lmot_rpt, instructions_rpt) = interprete_instructions lmot'
              in
60        let (new_lmot, instructions) = interprete_instructions new_lmot_rpt
              in
61        (new_lmot, Repeat(e, instructions_rpt)::instructions)
62    (* CALL: on recupere les valeurs des parametres passes a la fonction et
          on ajoute l'instruction correspondante *)
63    | CALL::IDENT(nom_proc)::lmot' ->
64        let (new_lmot_def, valeurs_params) = get_params lmot' in
65        let (new_lmot, instructions) = interprete_instructions new_lmot_def
              in
66        (new_lmot, Call(nom_proc, valeurs_params)::instructions)
67    | _ -> failwith "rtfm␣noob␣(interprete_instructions)";;
68
69  (*
70   * fonction interprete_procedure
71   * But : interpreter une liste de mots correspondant a une procedure
```

```
72  * Entree : la liste de mots commencant par la definition de la procedure
73  * Precondition : programme LOGO correct
74  * Sortie : la nouvelle liste de mots a interpreter, la liste de parametres
        et l'arbre d'instructions de la procedure
75  * Postcondition : -
76  *)
77  let rec interprete_procedure lmot =
78    match lmot with
79    (* on remonte d'un niveau dans l'arbre d'instructions *)
80    | END::lmot' -> (lmot', [], [])
81    (* on recupere la liste des parametres *)
82    | IDENT(param)::lmot' ->
83        let (new_lmot, params, instructions) = interprete_procedure lmot' in
84        (new_lmot, param::params, instructions)
85    (* on recupere l'arbre d'instructions de la fonction *)
86    | BEGIN::lmot' ->
87        let (new_lmot, instructions) = interprete_instructions lmot' in
88        (new_lmot, [], instructions)
89    | _ -> failwith "rtfm noob (interprete_procedure)";;
90
91    (*
92     * fonction analyseur syntaxique
93     * But : transformer la liste des mots du programme LOGO en un programme
           interpretable par l'analyseur semantique
94     * Entree : la liste des mots du prog LOGO
95     * Precondition : Pas d'erreurs dans le programme LOGO (syntaxe, sens,
           etc)
96     * Sortie : liste des procedures et liste des sous-programmes du prog
           principal, interpretable par l'analyseur semantique
97     * Postcondition : le programme en sortie correspond au programme en
           entree
98     *)
99  let rec analyseur_syntaxique lmot =
100   match lmot with
101   (* definition du prog principal; constitue d'une liste de sous-programmes
          *)
102   | BEGIN::lmot' ->
103       let (_, instructions) = interprete_instructions lmot' in
104       ([], instructions)
105   (* Definition d'une procedure : on recupere la procedure et la nouvelle
          liste de mots *)
106   | DEF::IDENT(nom_proc)::lmot' ->
107       let (new_lmot, params, instructions_proc) = interprete_procedure lmot
              ' in
108       let (procs, instructions) = analyseur_syntaxique new_lmot in
109       ((nom_proc, params, instructions_proc)::procs, instructions)
110   | _ -> failwith "rtfm noob! (analyseur_syntaxique)";;
```

3.3 logo_analyseur_semantique.ml

```
1  open Logo_types;;
2
```

```
3  let pi = 3.141592654;;
4  let deg2rad angle = pi *. angle /. 180.;;
5  let etat_initial = ((0., 0.), 0.);;
6
7  (*
8   * fonction get_procedure
9   * But : retourner la liste de params et l'arbre d'instructions d'une
        procedure a partir de son nom
10  * Entree : le nom de la procedure et les defs ed l'env courant
11  * Precondition : la fonction recherchee est bien dans l'env courant
12  * Sortie : la procedure correspondant au nom fourni
13  * Postcondition : -
14  *)
15 let rec get_procedure nom_proc defs =
16   match defs with
17   | (nom, params, instructions)::defs' -> if nom_proc = nom then (nom,
       params, instructions) else get_procedure nom_proc defs'
18   | _ -> failwith "rtfm noob (get_procedure)";;
19
20 (*
21  * fonction get_valeur_param
22  * But : trouver la valeur d'une Var
23  * Entree : le nom de la Var en question et la liste de parametres de l'env
       courant
24  * Precondition : la Var est bien declaree dans l'env
25  * Sortie : la valeur associee a la Var
26  * Postcondition : -
27  *)
28 let rec get_valeur_param var params =
29   match params with
30   | (param, valeur)::params' -> if var = param then valeur else
       get_valeur_param var params'
31   | _ -> failwith "rtfm noob (get_valeur_param)";;
32
33 (*
34  * fonction evalue_expression
35  * But : evaluer la valeur de l'expression passee en param
36  * Entree : l'expression a evaluer
37  * Precondition : -
38  * Sortie : la valeur flottante de l'expression
39  * Postcondition : -
40  *)
41 let rec evalue_expression env expr =
42   match expr with
43   | Const c -> c
44   | Plus(a, b) -> (evalue_expression env a) +. (evalue_expression env b)
45   | Moins(a, b) -> (evalue_expression env a) -. (evalue_expression env b)
46   | Div(a, b) -> (evalue_expression env a) /. (evalue_expression env b)
47   | Mult(a, b) -> (evalue_expression env a) *. (evalue_expression env b)
48   | Cosinus(t) -> cos (deg2rad(evalue_expression env t))
49   | Sinus(t) -> sin (deg2rad(evalue_expression env t))
50   | Tangente(t) -> tan (deg2rad(evalue_expression env t))
```

```
51    | Var(v) -> let (params, _) = env in evalue_expression env (
          get_valeur_param v params);;
52
53  (*
54   * fonction evalue_condition
55   * But : Evaluer la valeur booleenne d'une condition
56   * Entree : la condition
57   * Precondition : -
58   * Sortie : la valeur de la condition
59   * Postcondition : -
60   *)
61  let rec evalue_condition env test =
62    match test with
63    | Equal(a,b) -> ((evalue_expression env a) = (evalue_expression env b))
64    | InfEq(a,b) -> ((evalue_expression env a) <= (evalue_expression env b))
65    | And(a,b) -> ((evalue_condition env a) && (evalue_condition env b))
66    | Or(a,b) -> ((evalue_condition env a) || (evalue_condition env b))
67    | Not(a) -> not (evalue_condition env a);;
68
69  (*
70   * fonction execute_instruction
71   * But : calculer l'etat du curseur apres l'exec d'une instruction, a
           partir de l'etat courant
72   * Entree : l'environnement courant, l'instruction a executer et l'etat
           avant l'instruction
73   * Precondition : -
74   * Sortie : le nouvel etat
75   * Postcondition : -
76   *)
77  let execute_instruction env instruction ((x,y), angle) =
78    match instruction with
79    | Move(e) ->
80        let distance = evalue_expression env e in
81        let (new_x, new_y) = ((cos (deg2rad angle))*.distance +. x, (sin (
              deg2rad angle))*.distance +. y) in
82        ((new_x, new_y), angle)
83    | Jump(e) ->
84        let distance = evalue_expression env e in
85        let (new_x, new_y) = ((cos (deg2rad angle))*.distance +. x, (sin (
              deg2rad angle))*.distance +. y) in
86        ((new_x, new_y), angle)
87    | Rotate(t) ->
88        let rotation = evalue_expression env t in
89        ((x, y), angle+.rotation)
90    | _ -> failwith "rtfm noob (execute_instruction)";;
91
92  (*
93   * fonction enrichir_env
94   * But : associer les params a leurs valeurs dans l'env
95   * Entree : les params de l'env courant, la liste de params, et la liste de
           valeurs
96   * Precondition : -
97   * Sortie : l'env enrichi
```

```
98  | * Postcondition : -
99  | *)
100 | let rec enrichir_env params_env params_proc valeurs_params =
101 |   (List.map2 (fun p v -> (p,v)) params_proc (List.map (fun x -> Const(
        evalue_expression (params_env, []) x)) valeurs_params))@params_env;;
102 |
103 | (*
104 | * fonction execute_programme
105 | * But : transformer un arbre d'instructions en une liste de commandes
      interpretables par la fonction de dessin
106 | * Entree : l'arbre d'execution du programme
107 | * Precondition : -
108 | * Sortie : l'etat a la fin du bloc et la liste de cmd correspondant a l'
      arbre d'instructions
109 | * Postcondition : -
110 | *)
111 | let execute_programme (defs, instructions) =
112 |   let rec eval_instructions instructions env etat =
113 |     match instructions with
114 |     | [] -> (etat, [])
115 |     (* Move: on recupere le nouvel etat du curseur apres l'instruction Move
          , et on ajoute la cmd correspondante *)
116 |     | Move(e)::instructions' ->
117 |         let ((x, y), angle) = execute_instruction env (Move(e)) etat in
118 |         let (etat_suivant, cmd_list) = eval_instructions instructions' env
              ((x, y), angle) in
119 |         etat_suivant, Moveto(x, y)::cmd_list
120 |     (* Jump: idem *)
121 |     | Jump(e)::instructions' ->
122 |         let ((x, y), angle) = execute_instruction env (Jump(e)) etat in
123 |         let (etat_suivant, cmd_list) = eval_instructions instructions' env
              ((x, y), angle) in
124 |         etat_suivant, Jumpto(x, y)::cmd_list
125 |     (* Rotate: idem *)
126 |     | Rotate(t)::instructions' ->
127 |         let etat_courant = (execute_instruction env (Rotate(t)) etat) in
128 |         let (etat_suivant, cmd_list) = eval_instructions instructions' env
              etat_courant in
129 |         etat_suivant, cmd_list
130 |     (* Color: on ajoute la cmd correspondante *)
131 |     | Color(r, g, b)::instructions' ->
132 |         let (etat_suivant, cmd_list) = eval_instructions instructions' env
              etat in
133 |         etat_suivant, Change_color(Graphics.rgb (int_of_float (
              evalue_expression env r))
134 |                                                 (int_of_float (
                                                      evalue_expression env g
                                                      ))
135 |                                                 (int_of_float (
                                                      evalue_expression env b
                                                      ))
136 |                                   )::cmd_list
```

```
137     (* If: si la condition est vraie, on execute le premier bloc d'
            instructions, le deuxieme sinon *)
138     | If(test, instructions_if, instructions_else)::instructions' ->
139         if evalue_condition env test then
140           eval_instructions (instructions_if@instructions') env etat
141         else
142           eval_instructions (instructions_else@instructions') env etat
143     (* Repeat: on execute le bloc d'instructions tant que l'expression est
            > 0 *)
144     | Repeat(expr, instructions_rpt)::instructions' ->
145         let x = evalue_expression env expr in
146         if x > 0. then
147           eval_instructions ( (Repeat(Const(x-.1.), instructions_rpt))::(
                instructions_rpt@instructions') ) env etat
148         else
149           eval_instructions instructions' env etat
150     (* Call: on enrichit l'environnement et on execute le bloc de la
            fonction *)
151     | Call(nom_proc, valeurs_params)::instructions' ->
152         let (params, defs) = env in
153         let (_, params_proc, instructions_proc) = get_procedure nom_proc
                defs in
154         let new_env = (enrichir_env params params_proc valeurs_params, defs
                ) in
155         let (nouvel_etat_call, cmd_list_call) = (eval_instructions
                instructions_proc new_env etat) in
156         let (nouvel_etat, cmd_list) = (eval_instructions instructions' env
                (nouvel_etat_call)) in
157         nouvel_etat, cmd_list_call@cmd_list
158
159   (* la liste de procedures joue le role d'env initial *)
160   in let (_, cmd_list) = eval_instructions instructions ([], defs)
        etat_initial in cmd_list;;
```