Rapport de projet CAML
Mini-langage : LOGO

Adrian COURRÈGES
1A IN, goupe D

Janvier 2007

Table des matières

1 Introduction **3**
1.1 Présentation de LOGO . . . 3
1.2 Spécifications du sujet . . . 3

2 Choix des structures de données **4**
2.1 Instructions . . . 4
2.2 Définitions de procédures . . . 4
2.3 Programme LOGO . . . 4
2.4 Environnement . . . 4
2.5 Etat du système . . . 5

3 Réalisation et raffinage des fonctions de lecture **5**
3.1 Fonction lit_bloc . . . 5
3.2 Fonction lit_definition . . . 6
3.3 Fonction lit_programme . . . 6
3.4 Exemple et test . . . 6

4 Réalisation et raffinage des fonctions d'évaluation **7**
4.1 Fonction evalue_expression . . . 7
4.2 Fonction evalue_condition . . . 8
4.3 Fonction execute_instruction . . . 8
4.4 Procédure execute_programme . . . 10

5 Tests **10**
5.1 Programme utilisant le module . . . 10
5.2 Exemples de dessins obtenus . . . 11

6 Manuel d'utilisation et contraintes **13**

7 Améliorations ou extensions possibles **13**

8 Conclusion **13**

9 Code source **13**

1 Introduction

1.1 Présentation de LOGO

LOGO est un langage de programmation qui a été développé en 1970, dans le but de familiariser tout néophyte à certains concepts algorithmiques. Il permet, grâce à une syntaxe simple, de réaliser des dessins graphiques en donnant des instructions de la forme "avancer de 3" ou "tourner de $+30°$" à un curseur défini par ses coordonnées dans le plan et son orientation angulaire. Cette approche conviviale permet alors de mieux appréhender la notion de procédures paramétrées, de boucles, de tests, ou de récursivité.

On a ici affaire à une version simplifiée du langage LOGO utilisant uniquement les mot-clés :

`DEF, BEGIN, END, IF, THEN, ELSE, REPEAT, CALL, MOVE, JUMP, COLOR, FILL, NOFILL`

ainsi que les classiques opérateurs (identiques à ceux de CaML) :

`sin, cos, tan, *, +, -, /, =, <=, <, >=, >, &&, ||, not`

Ainsi pour dessiner un carré en LOGO, il suffit d'écrire :

```
BEGIN
REPEAT (4)
  BEGIN
  MOVE   (75)
  ROTATE (90)
  END
END
```

1.2 Spécifications du sujet

La rédaction d'un algorithme LOGO doit obéir à certaines règles syntaxiques :
- les arguments d'une fonction sont des chaînes ou des nombres, entre parenthèses, séparés par des virgules.
- `ROTATE`, `MOVE`, et `JUMP` ne prennent qu'un seul argument. (en degré pour `ROTATE`)
- `COLOR` prend 3 arguments (code RGB).
- `FILL` et `NOFILL` ne prennent aucun argument.
- un bloc d'instruction doit toujours être encadré par des balises `BEGIN` et `END`.
- `CALL` est suivi d'une chaîne de caractères et d'arguments.
- `IF` est suivi d'un test entre parenthèses puis d'un bloc précédé par `THEN` et d'un précédé par `ELSE`.
- `REPEAT` prend un argument et est suivi d'un bloc.
- `DEF` est suivi d'une chaîne de caractères, d'arguments et d'un bloc.
- les commentaires LOGO sont encadrés par `(*` et `*)` et ignorés à l'exécution.

La structure d'un programme LOGO comprend la définition de zéro ou plusieurs procédures, en tête du fichier, suivies d'un unique bloc `BEGIN`...`END` constituant le corps principal.

En outre, un analyseur lexical et syntaxique manipulant les flux CaML est fourni par le module `logo_base`. Il permet la lecture de fichiers source LOGO ainsi que le tracé de certaines fonctions graphiques de base. Il implémente entre autres les types `expr`, `test`, `mot` définis explicitement. (consulter le fichier `logo_base.mli` pour plus d'informations)

Objectif : Réaliser le module `logo_principal.ml`, en se basant sur le module `logo_base.ml`, permettant d'obtenir un dessin graphique à partir d'un fichier texte LOGO.
Les fonctions à fournir sont détaillées dans `logo_principal.mli`.

Note : le traitement des erreurs de syntaxe dans les programmes LOGO n'est pas exigé, mais dans la mesure du possible, les fonctions d'analyse lexicales renverront à l'utilisateur les erreurs les plus grossières et les plus immédiatement décelables.

2 Choix des structures de données

Dans cette section sont définis les types permettant de retranscrire un programme LOGO en syntaxe compréhensible par CaML. Des types issus du module `logo_base` sont utilisés.

2.1 Instructions

Le type d'une instruction LOGO a été défini en CaML de la sorte :

```
type instruction =
     ROTATION of expr
    |DEPLACE of expr
    |SAUT of expr
    |COULEUR of expr * expr * expr
    |REMPLIR
    |ARRETREMPLIR
    |APPEL of string * (expr list)
    |SI of test * (instruction list) * (instruction list)
    |REPETE of expr * (instruction list)
    |LISTE of (instruction list) ;;
```

Cette forme découle directement des mots-clés LOGO. Les 6 premières entités correspondent aux instructions élémentaires : `ROTATE`, `MOVE`, `JUMP`, `COLOR`, `FILL` et `NOFILL` avec leur(s) paramètre(s) de type expr. Les 3 suivantes sont des instructions composées, un bloc d'instruction étant assimilé à une `instruction list`. Enfin, on considère qu'un bloc d'instruction constitue une expression en soi, d'où la présence de `LISTE of (instruction list)`. Cette convention se révèle particulièrement utile pour la fonction `execute_instruction` définie par la suite : si on exécute l'instruction `SI(test, bloc1, bloc2)` il convient d'exécuter un des deux blocs, et donc une liste d'instruction.

2.2 Définitions de procédures

La définition d'une procédure est introduite par le mot `DEF`, son type est le suivant :

```
type definition = PROC of (string * (string list) * (instruction list)) ;;
```

`string` est la chaîne de caractères correspondant au nom de la procédure LOGO. La liste des noms des variables de la procédure est contenue dans `string list`. Le type `string` a été choisi car il correspond au type de sortie de fonctions de lecture de `logo_base`. Le bloc `instruction list` contient le code de la procédure.

2.3 Programme LOGO

Le type programme LOGO est :

```
type programme = PROG of (definition list) * (instruction list) ;;
```

Comme spécifié dans le sujet, un programme contient une liste de procédures, ainsi qu'un bloc constituant le corps principal du programme.

2.4 Environnement

Un environnement contient les définitions de procédures et de variables.

```
type environnement = ENV of ((definition list) * ((string * float) list)) ;;
```

La première liste contient des définitons de *toutes* les procédures du programme et la seconde est formée de couples (variable, valeur).
Un environnement est associé à une unique procédure. Cette convention permet de savoir que la valeur d'un paramètre `n` de la `(string * float) list` est bien celle du `n` de la procédure liée à l'environnement, et non pas du `n` d'une autre procédure de la `definition list`.
Pour le bloc principal d'un programme LOGO la `(string * float) list` est vide : tous les paramètres sont renseignés explicitement (sous forme de nombres).

2.5 Etat du système

L'état contient les informations relatives à la position, l'orientation du curseur, et le mode remplissage.

```
type etat = {
             abs : float ;  (* abscisse du curseur               *)
             ord : float ;  (* ordonnée du curseur               *)
             ang : float ;  (* orientation du curseur en radians *)
             remp : bool ;  (* mode remplissage                  *)
             ox : float ;   (* abscisse d'origine du remplissage *)
             oy : float     (* ordonnée d'origine du remplissage *)
             } ;;
```

Comme on n'a souvent besoin d'accéder qu'à une composante particulière de l'état, l'emploi de cette structure évite le filtrage complet qu'un type `ETAT of (float * float *...)` imposerait. On gagne également en lisibilité du code. De plus, cette structure est adaptée à l'*évolution* que le mini-langage pourrait connaître : on peut rajouter des variables (pour des styles de tracé : pointillé, trait épais...) sans avoir à modifier profondément le filtrage. La fonction `execute_instruction` étant la seule à travailler sur des états, il est aisé d'apporter des extensions au mini-langage en modifiant cette fonction.

3 Réalisation et raffinage des fonctions de lecture

On définit tout d'abord trois fonctions auxiliaires réalisant certaines lectures sécurisées, et déclenchant une exception en cas d'erreur de syntaxe au sein du programme LOGO.

La fonction `lire_unique_arguments : flux_lecture -> expr` s'utilise après la lecture de `ROTATE`, `MOVE`, `JUMP`, ou `REPEAT` afin d'obtenir un unique paramètre.

La fonction `lire_rgb : flux_lecture -> expr * expr * expr` s'utilise après la lecture de `COLOR` afin d'obtenir 3 composantes RGB.

La fonction `lire_chaine : flux_lecture -> string` s'utilise après la lecture de `CALL` pour obtenir la chaîne du nom de la procédure.

3.1 Fonction lit_bloc

```
(*-----------------------------------------------------------------
   fonction lit_bloc : flux_lecture -> instruction list

   semantique : s'utilise apres lecture d'un BEGIN
                retourne la liste d'instruction d'un bloc et lit
                le END final.
                declenche une exception si erreur de syntaxe logo.

   arguments : f flux de lecture

   resultat : liste des instructions
-----------------------------------------------------------------*)
```

Principe :
Le filtrage s'effectue avec la lecture du mot-clé suivant. Le cas de base est la lecture de `END`, il suffit alors de renvoyer la liste vide. Dans le cas d'instructions élémentaires, on lit les paramètres correspondant à l'instruction, et on concatène l'élément formé avec l'appel récursif sur le reste du flux. Pour les instructions composées, on lit d'abord le(s) bloc(s) auxquels l'instruction s'applique, avant de concaténer avec la liste récursive.

Le sujet impose que `BEGIN` soit déjà lu au moment de l'appel de `lit_bloc`. La fonction obtenue ici tolère cependant son appel sur un `BEGIN`.

3.2 Fonction lit_definition

```
(*-----------------------------------------------------------------
   fonction lit_definition : flux_lecture -> definition

   semantique : s'utilise apres lecture d'un DEF
                retourne la definition de la procedure avec son
                nom, sa liste d'arguments, et son bloc.
                declenche une exception si erreur de syntaxe logo.

   arguments : f flux de lecture

   resultat : definition
-----------------------------------------------------------------*)
```

Principe :
`DEF` étant déjà lu, on stocke le nom dans la chaîne `nom` grâce à `lire_chaine`. On met ensuite les noms des arguments dans une liste `l`. On lit le mot `BEGIN`, puis on met dans `bloc` la liste des instructions de la procédure grâce à la fonction `lit_bloc`. Il suffit de renvoyer `PROC(nom,l,bloc)`.

3.3 Fonction lit_programme

```
(*-----------------------------------------------------------------
   fonction lit_programme : flux_lecture -> programme

   semantique : lit un programme LOGO et retourne le couple forme
                de la liste de definitions de procedures et du
                bloc principal du programme.
                declenche une exception si erreur de syntaxe logo.

   arguments : f flux de lecture

   resultat : definition
-----------------------------------------------------------------*)
```

Principe :
Si l'appel se produit sur un `DEF` on utilise `lit_definition` pour ajouter une définition à la liste des procédures du reste du programme. On fait de même pour toutes les procédures suivantes. Lorsque `BEGIN` est lu, on sait qu'on a atteint le corps principal du programme, on emploie `lit_bloc` pour l'ajouter au membre de droite du couple résultat et on renvoie le résultat.

Un tel choix impose une contrainte pour la rédaction d'un programme LOGO : toutes les procédures doivent être définies au début du programme. Si une procédure est définie après le bloc principal `BEGIN...END`, elle sera **ignorée**.

3.4 Exemple et test

Voici l'exemple de la conversion du fichier LOGO `carre.logo` en syntaxe interprétable par CaML. Les fonctions définies par la suite ont pour rôle d'évaluer un tel programme.

```
#lit_programme(ouvre_fichier "carre.logo");;
- : programme =
 PROG
  ([PROC
     ("carre", ["n"],
      [REPETE (Const 4.0, [DEPLACE (Var "n"); ROTATION (Const 90.0)])])],
   [SAUT (Const 100.0); ROTATION (Const 90.0); SAUT (Const 100.0);
    ROTATION (Moins (Const 0.0, Const 90.0)); APPEL ("carre", [Const 50.0]);
    ROTATION (Const 180.0); APPEL ("carre", [Const 50.0])])
```

On peut également trouver des tests unitaires pour chaque fonction sous forme de commentaires dans le listing.

4 Réalisation et raffinage des fonctions d'évaluation

La syntaxe LOGO utilisant des mesures en degrés, et les fonctions trigonométriques de CaML en radians, on utilise par la suite une fonction utile pour convertir les degrés en radians.
`rad_of_deg : float -> float`

4.1 Fonction evalue_expression

On utilise une fontion de raffinage :

```
(*-----------------------------------------------------------------
   fonction valeur_de_var : ('a * 'b) list -> 'a -> 'b

   semantique : donne la valeur float associee a une chaine
                dans une liste de couple (chaine,valeur)

   arguments : (float * string) list
               un nom de variable de type string

   resultat : float correspondant a la variable
-----------------------------------------------------------------*)
```

Principe :
On parcourt récursivement la liste de couples (variable,valeur). On renvoie la valeur float lorsqu'une occurence de la variable a été trouvée, et une exception si la variable n'apparaît pas dans la liste.

```
(*-----------------------------------------------------------------
   fonction evalue_expression : environnement -> expr -> float

   semantique : donne la valeur float d'une expression pour
                un environnement donne.

   arguments : env environnement
               e expression

   resultat : float correspondant a l'expression evaluee
-----------------------------------------------------------------*)
```

Principe :
Le cas direct est `CONST(valeur)` car il suffit alors de renvoyer le float `valeur`. Pour `Var(nom)` on évalue la valeur de l'expression dans l'environnement grâce à `valeur_de_var` définie ci-dessus. Les autres cas sont constitués de constructeurs d'arité 2 (opérateurs arithmétiques) ou 1 (opérateurs trigonométriques), il suffit donc d'évaluer par récurrence leurs paramètres et d'exécuter explicitement l'opération correspondante. Les angles sont convertis en radians avant évaluation par un opérateur trigonométrique grâce à `rad_of_deg`.

4.2 Fonction evalue_condition

```
(*-----------------------------------------------------------------
   fonction evalue_condition : environnement -> test -> bool

   semantique : donne la valeur booleenne d'un test pour un
                environnement donne.

   arguments : env environnement
               t de type test

   resultat : booleen correspondant a la valeur du test
-----------------------------------------------------------------*)
```

Principe :
Si le test concerne une comparaison entre expressions, il suffit d'appliquer la comparaison aux expressions évaluées à l'aide de `evalue_expression` précédemment définie. Si le test est un constructeur logique d'arité 2 de deux tests, on applique l'opérateur du constructeur aux deux tests evalués par `evalue_condition`. Pour `Not(test)` on prend la négation du `test` évalué.

4.3 Fonction execute_instruction

Des raffinages sont définis par la suite pour la fonction `execute_instruction` :

```
(*-----------------------------------------------------------------
  execute_instruction : (environnement -> instruction -> etat)
                        -> etat

  semantique : execute une instruction et renvoie le nouvel
               etat apres execution.

  arguments : env : environnement
              inst : instruction
              etat : etat du systeme

  resultat : etat apres execution de l'instruction
-----------------------------------------------------------------*)
```

Principe :
Le filtrage s'effectue sur l'instruction entrée. Il faut donc envisager chaque cas possible pour l'instruction :

- `SI(test,alors,sinon)` : On évalue `test` dans l'environnement grâce à `evalue_condition`, s'il est positif on exécute recursivement le bloc `alors` et dans le cas contraire le bloc `sinon`.
- `REPETE(e,liste)` : On détermine la forme explicite float de `e`. Si `e`<1, il n'y a rien à faire, sinon on exécute la liste d'instructions une fois, puis on fait un appel récursif pour une répétition du même bloc, avec `e` décrémenté de 1.
- `ROTATION(angle)` : On évalue la valeur de `angle` puis on la convertit en radians. On incrémente l'angle de l'état actuel du système par la valeur obtenue.
- `SAUT(e)` : On évalue la valeur de `e`. A partir des coordonnées actuelles du curseur et de son orientation angulaire, on détermine les nouvelles coordonnées `(x,y)` du curseur. On déplace le curseur à cette position sans tracer de trait avec `fmoveto` puis on renvoie un état dont la nouvelle abscisse est `x` et l'ordonnée `y`.
- `DEPLACE(e)` : On évalue la valeur de `e`. A partir des coordonnées actuelles du curseur et de son orientation angulaire, on détermine les nouvelles coordonnées `(x,y)` du curseur. Si le mode `FILL` est désactivé, on déplace le curseur à cette position en traçant un trait avec `flineto`. Si le mode `FILL` est activé on vérifie que le point avant déplacement est différent de l'origine du remplissage, puis on dessine le triangle avec le point d'origine du remplissage,

le point avant instruction, et le point après instruction. Enfin, on renvoie un état dont la nouvelle abscisse est `x` et l'ordonnée `y`.

- `REMPLIR` : On renvoie un nouvel état dont les coordonnées du curseur et son orientation sont inchangées, mais dont le booléen `remp` vaut `true` et les coordonnées du point d'origine du remplissage égales aux coordonnées courantes du curseur.
- `ARRETREMPLIR` : On se contente de renvoyer le même état où le booléen `remp` a été mis à `false`.
- `LISTE(bloc)` : ce cas est indispensable à l'exécution d'une instruction conditionnelle ou de répétition ou d'appel de procédure. Si `bloc` est une liste vide, il n'y a rien à faire, sinon il suffit d'exécuter la tête, puis d'appeler récursivement la fonction sur la queue de la liste, avec le nouvel état obtenu après l'exécution de la tête.
- `APPEL(nom,liste_val)` : A partir du nom de la procédure, on extrait de l'environnement la liste des noms de ses variables et son bloc d'instruction. On crée la liste des couples (nom de variable,valeur) à partir des deux listes à disposition. On crée un nouvel environnement dans lequel la liste des procédures reste inchangée mais avec notre nouvelle liste de couples. Il suffit alors d'exécuter le bloc de la procédure avec le nouvel environnement.
  Les deux fonctions de raffinage nécessaires pour traiter le cas d'appel de procédures sont détaillées ci-dessous.

```
(*-----------------------------------------------------------------
  fonction extraire_proc : (definition list -> string )
                           -> (string list * instruction list)

  semantique : extraction de la liste d'arguments et du bloc
               d'une procedure a partir de son nom et d'une
               liste de procedures

  arguments : l : liste de defs de procedures
              nom : nom de la procedure

  resultat : couple(liste d'arguments,bloc d'instruction)
-----------------------------------------------------------------*)
```

Principe :
On parcourt la liste de définitions de procédures. Si un nom correspond à celui recherché, on renvoie la liste des variables et le bloc de la procédure en question. Si on atteint la liste vide, l'utilisateur a essayé de faire un `CALL` sur une procédure non définie, et on déclenche alors une exception.

```
(*-----------------------------------------------------------------
  nouvelle_varliste : (environnement -> 'a list -> expr list)
                      -> ('a * float) list

  semantique : liste de couples (variable,valeur) a partir de
               la liste des variables et celle des valeurs

  arguments : env : environnement
              lnom : liste des noms des variables
              lvar : liste des valeurs des variables

  resultat : liste de couples (variable,valeur)
-----------------------------------------------------------------*)
```

Principe :
On parcourt les deux listes simultanément, en formant des couples (variable,valeur). Si les listes n'ont pas le même cardinal, l'utilisateur a commis une erreur sur le nombre de paramètres, soit à la définiton de la procédure, soit lors de l'appel ; on déclenche alors une exception.

4.4 Procédure execute_programme

```
(*-----------------------------------------------------------------
  Procedure execute_programme : programme -> unit

  semantique : Produit un dessin a partir d'un programme LOGO

  arguments : prog : programme
-----------------------------------------------------------------*)
```

Principe :
Selon les conventions, on place le curseur en `(0,0)` avec une couleur noire. On crée un état initial avec une orientation nulle, en `(0,0)`, avec le mode de remplissage désactivé. L'environnement de départ pour le bloc principal du programme contient les définitions de procédures du programme entré et une liste vide pour les valeurs de variables (tous les arguments de fonction du bloc principal sont explicites). Il suffit alors d'appeler la fonction `execute_instruction` sur cet environnement vierge et la liste des instructions du bloc principal.

5 Tests

5.1 Programme utilisant le module

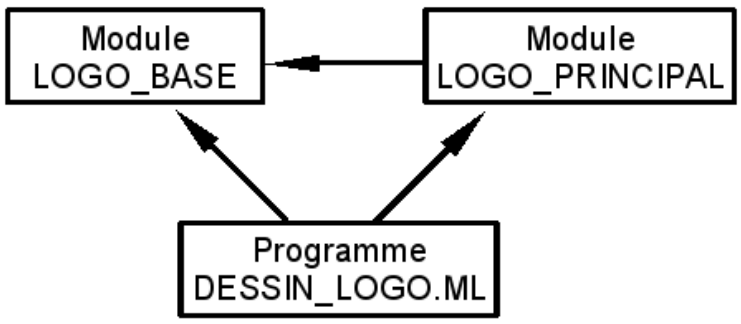
Un fichier `dessin_logo.ml` a été créé dans le but de réaliser simplement un dessin simplement à partir du chemin d'accès du fichier LOGO sur le disque. Ce programme charge en mémoire les deux modules `logo_base` et `logo_principal`, et implémente une fonction de dessin :

```
(*-----------------------------------------------------------------
  procedure dessin : string -> unit

  semantique : realise un dessin a partir d'un fichier LOGO.

  arguments : nom : chaine du chemin d'acces au fichier LOGO.
-----------------------------------------------------------------*)
```

Cette fonction se charge d'ouvrir une fenêtre graphique, initialise le flux, et appelle les fonctions du module `logo_principal`. Une fois le tracé accompli, la fenêtre graphique de CAML se ferme dès qu'une touche est entrée.

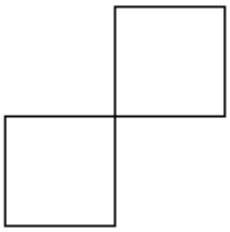
Pour exécuter le programme `penrose.logo` il suffit alors de rentrer :
`#dessin "penrose.logo" ;;`

On a le schéma de dépendance suivant entre les modules :

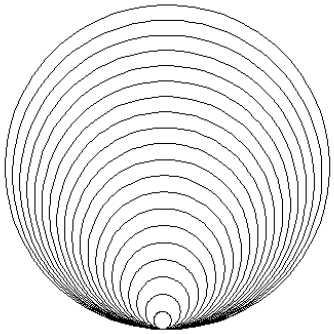


5.2 Exemples de dessins obtenus

Voici les dessins obtenus en exécutant les programmes LOGO fournis dans l'archive.

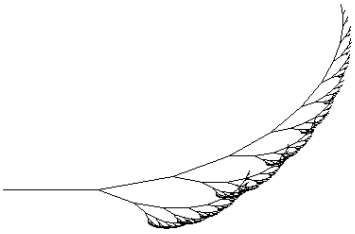


Programme `carre.logo`

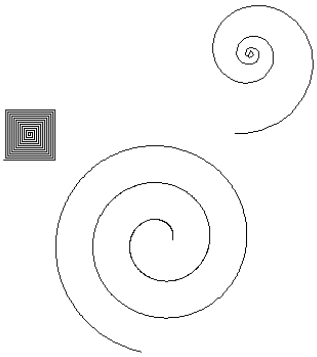


Programme `cercle.logo`

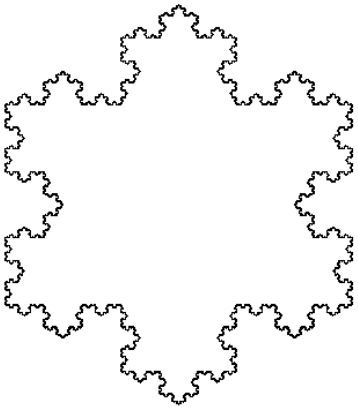
Programme `ecrou.logo`



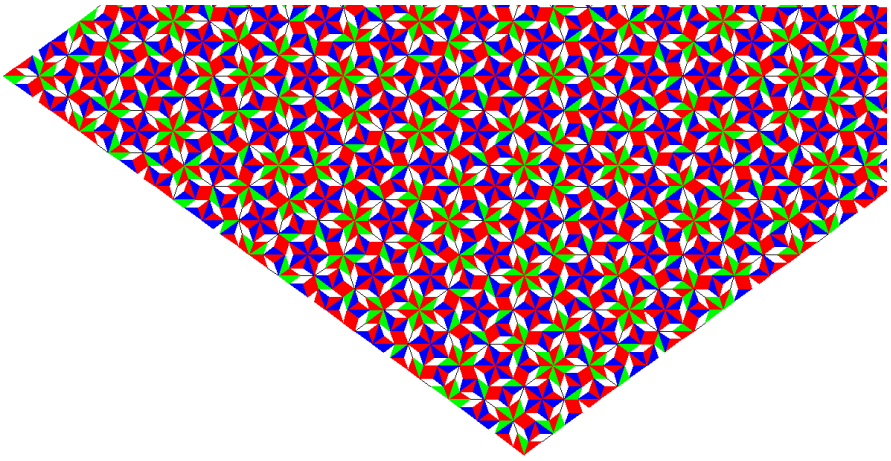
Programme `fougere.logo`



Programme `spirales.logo`



Programme `vonkoch.logo`

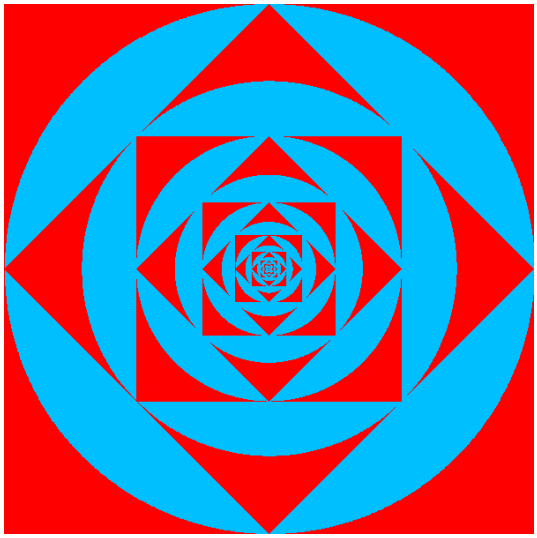


Programme `penrose.logo`

Commentaires sur la finalité des tests :

`carre` contient les instructions les plus basiques. On doit s'assurer qu'il est parfaitement exécuté avant de s'attaquer au debugage du mode de remplissage ou d'appel mutuel de procédures.
`ecrou` et `penrose` sont adaptés pour la vérification du bon fonctionnement du mode remplissage et du changement de couleur.
`cercle` et `spirales` sont exigeants au niveau de l'évaluation des paramètres. Ils ont besoin d'une précision élevée, et permettent donc de déceler des lacunes dans les fonctions d'évaluation trigonométriques.
`fougere` et `vonkoch` comprennent de nombreux appels de procédures. Ils permettent de vérifier que des appels mutuels sont bien gérés, et qu'il n'y a aucun problème au niveau de la génération d'environnements.

Voici un programme LOGO que j'ai créé dans le but de rassembler les critères de tests précédents : on utilise des couleurs, le remplissage, une procédure s'appelant elle-même, et des dessins de droites et de cercles pour vérifier la précision des évaluations.



Programme `mosaique.logo`

```
DEF CARRE (n)                 | DEF MOSAIQUE (l)
BEGIN                         | BEGIN
COLOR (255,0,0)               | IF (l<=1)
  FILL                        |  THEN
  REPEAT (4)                  |   BEGIN
  BEGIN                       |   END
   MOVE (n)                   |  ELSE
   ROTATE (90)                |  BEGIN
   END                        |   CALL CARRE (l)
  NOFILL                      |   JUMP (l/2)
END                           |   CALL cercle (l*3.14159265)
                              |   ROTATE (45)
DEF cercle (r)                |   CALL MOSAIQUE (l/1.414213562)
BEGIN                         |  END
COLOR (0,191,255)             | END
FILL                          |
REPEAT (1000)                 | BEGIN
  BEGIN                       | CALL MOSAIQUE (600)
  MOVE   (r/1000)             | END
  ROTATE (360/1000)           |
  END                         |
NOFILL                        |
END                           |
```

6 Manuel d'utilisation et contraintes

Les fichiers `.mli` fournissent les premières informations pour utiliser les modules. D'autre part, la finalité de l'interpréteur est, pour l'utilisateur lambda, le tracé de la figure à partir du nom du fichier, ce que fait la fonction `dessin` du fichier `dessin_logo.ml`
Toutefois, certaines interprétations ont dues être faites pour les instructions LOGO présentant une ambiguïté. Les conventions et règles d'utilisation sont résumées ci-dessous :

- La structure d'un programme LOGO est une liste de procédures suivie d'un unique bloc principal.
- Tout bloc d'une instruction composée est encadré par `BEGIN...END`.
- A l'activation du mode remplissage, la position du curseur est sauvegardée ; c'est depuis cette position `0` que seront tracés le(s) triangle(s) lorsque le curseur passe de `P` à `P'`. Ce mode reste activé tant qu'on ne le désactive pas explicitement par `NOFILL`.
- Un `REPEAT` d'un nombre <1 est ignoré.

7 Améliorations ou extensions possibles

La gestion des erreurs de syntaxe LOGO est minimale, mais l'interpréteur tente de renvoyer un message le plus explicite possible. Le système détectant de telles erreurs pourrait être amélioré, même si les efforts se sont davantage portés sur la réalisation d'un interpréteur opérationnel pour des programmes respectant les conventions LOGO.

Des fonctionnalités supplémentaires peuvent être implémentées : par exemple différents modes de tracé (pointillé, trait d'une certaine épaisseur). Il suffirait de modifier le module `logo_base` pour rajouter des mots-clés et `logo_principal` pour rajouter des champs à la structure `etat`.

8 Conclusion

L'ensemble des tests qui ont été menés à partir de différents fichier LOGO se sont montrés concluants. Des tests unitaires ont également été réalisés pour s'assurer du bon fonctionneent de chaque sous-fonction.
L'interpréteur LOGO n'a pas besoin de manuel en soi pour son utilisation car il suffit de connaître le nom du fichier pour lancer le tracé. Toutefois, il est nécessaire pour l'utilisateur de connaître la syntaxe du mini-langage ; celle-ci a été abordée au début de ce rapport, et les conventions qui ont été prises pour l'interpréteur ont été détaillées précédemment.

D'un point de vue pédagogique, LOGO se révèle bénéfique pour l'utilisateur s'initiant à l'algorithmique, mais également pour le programmeur réalisant l'interpréteur du mini-langage. Ce projet m'a permis de mieux appréhender les procédés d'analyse lexicale, qui interviennent avant tout processus de compilation.

9 Code source

Les pages suivantes contiennent le code source de `module_principal` ainsi que du programme permettant de le tester : `dessin_logo.ml`. Des commentaires ainsi que des tests unitaires y sont inclus.