Présentation

L'objectif de ce projet est la maîtrise de certains concept fondamentaux de la programmation, à travers la réalisation d'un interprète pour un sous-ensemble d'un célèbre langage de programmation : LOGO. Le langage LOGO est intimement lié à la tortue LOGO qui est simplement un curseur graphique manipulé par les instructions du langage.

Le but de ce projet est donc de comprendre la syntaxe du langage LOGO et ainsi de les interpréter afin de réaliser les opérations sur la fenêtre graphique de camllight.

Le langage LOGO manipule un curseur graphique définit par ses coordonnées cartésiennes dans le plan ainsi que son orientation en degrés. De plus, le langage LOGO devra permettre de définir des procédures avec des paramètres formels, et donc, de les appeler par la suite.

La syntaxe du langage LOGO ainsi que les instructions que devra comprendre notre interprète nous sont fournis. Ces instructions sont les suivantes :

- ✔ ROTATE (<*exp*>)
  L'orientation du curseur est modifiée de <*exp*> degrés (dans le sens direct).
- ✔ MOVE (<*exp*>)
  Déplace le curseur suivant son orientation en dessinant un segment de longueur <*exp*>.
- ✔ JUMP (<*exp*>)
  Déplace le curseur suivant son orientation d'une distance <*exp*>.
- ✔ COLOR (<*exp*>,<*exp*>,<*exp*>)
  Change la couleur du curseur (noir par défaut).
  Les arguments, compris entre 0 et 255, correspondent respectivement à la proportionde rouge, de vert et de bleu.
- ✔ FILL
  Mode remplissage. Mémorise la position courante O du curseur, et lorsque le curseur passe de P à P', trace le triangle plein (O,P,P').
- ✔ NOFILL
  Annule le mode précédent. C'est le mode par défaut.
- ✔ BEGIN <$inst_1$>...<$inst_n$> END
  Exécute les instructions du bloc de <$inst_1$> à <$inst_n$>.
- ✔ CALL <*nom*> (<exp_1>,...,<exp_n>)
  Appelle la procédure <*nom*> avec les paramètres <exp_1>,...,<exp_n>.
- ✔ IF <*exp*> THEN <$bloc_1$> ELSE <$bloc_2$>
  Exécute les instructions du bloc <$bloc_1$> ou <$bloc_2$> suivant la valeur de l'expression booléenne <*exp*>.
- ✔ REPEAT (<*exp*>) <*bloc*>
  Répète <*exp*> fois les instructions du bloc <*bloc*>.
- ✔ DEF <*nom*> (<var_1>,...,<var_n>) <*bloc*>
  Définit la procédure <*nom*> avec les paramètres formels <var_1>,...,<var_n> et les instructions du bloc <*bloc*>.

Les types et fonctions déjà définis

Le fichier logo_base.mli contient certaines définitions de type et les noms et types de certaines fonctions déjà implémentées dans le module logo_base.zo.

Les types que nous avons déjà à notre disposition sont les suivants :

- type expr
  Type représentant les expressions arithmétiques du langage.

- type test
  Type représentant les expressions booléennes du langage.

- type mot
  Type représentant les mots-clés et les identificateurs du langage.

- type flux_lecture
  Type des fichiers utilisés en lecture.

Les fonctions déjà implémentées sont les suivantes :

- value ouvre_fichier : string -> flux_lecture
  Ouverture d'un fichier en lecture a partir de son nom.

- value ferme_fichier : flux_lecture -> unit
  Fermeture d'un fichier

- value position_courante : flux_lecture -> string
  Information sur la position courante dans le fichier sous la forme d'une chaîne de caractères. Utile au debugage.

- value lit_condition : flux_lecture -> test
  Lecture d'une expression booléenne.

- value lit_paramètres : flux_lecture -> string list
  Lecture d'une liste de variables entre parenthèses, séparées par des virgules. Utile pour lire les paramètres formels d'une procédure.

- value lit_arguments : flux_lecture -> expr lit
  Lecture d'une liste d'expressions entre parenthèses, séparées par des virgules. Utile pour lire les paramètres réels d'une procédure appelée.

- value lit_mot_cle : flux_lecture -> mot
  Lecture d'un mot-clé ou d'un identificateur (variable).

Plus les fonctions graphiques :

- value fcolor : float -> float -> float -> unit

  Changement de couleur du mode graphique. Les trois arguments sont convertis en entier et utilisés modulo 256. Ils correspondent donc à trois octets codant respectivement la proportion de rouge, de vert et de bleu dans la nouvelle couleur du curseur.

- value fmoveto : float * float -> unit

  Déplacement sans trace à la position (x, y). Les deux arguments sont convertis à l'entier le plus proche. (x, y) devient la prochaine position courante.

- value flineto : float * float -> unit

  Déplacement avec trace à la position (x, y). Les deux arguments sont convertis à l'entier le plus proche. Une ligne est tracée entre la position courante et (x, y), prochaine position courante.

- value ftriangle : float * float -> float * float -> float * float -> unit

  ftriangle (ox, oy) (x, y) (x', y') : trace un triangle plein entre les points (ox, oy), (x, y) et (x', y'). Vérifie qu'on a bien la condition (ox, oy) <> (x, y) (le curseur a bougé). (x', y') devient la prochaine position courante. Utile en mode FILL.

Les types et fonctions à définir

Les types

Les types que nous devons utiliser ne nous sont pas définis mais simplement nommés afin de nous guider. Ces types que nous avons à définir sont les suivant :

- type instruction

  C'est le type des instructions qui correspondra avec les instructions précédentes.

```
type instruction =
            Rotate of expr
(* Rotation du curseur de <exp> degrés dans le sens direct *)
            |Move of expr
(* Déplacement du curseur de <exp> en traçant un segment *)
            |Jump of expr
(* Déplacement du curseur de <exp> sans tracé *)
            |Color of expr * expr * expr
(* Change la couleur du curseur *)
            |Fill
(* Mode remplissage *)
            |Nofill
(* Annule le mode précédent *)
            |If of test*(instruction list)*(instruction list)
(* Si le test est vrai, il faut exécuter la première liste d'instructions, sinon la seconde *)
            |Repeat of expr*(instruction list)
(* il faut exécuter n fois la liste d'instructions, n étant l'évaluation de l'expression *)
            |Call of mot*(expr list);;
(* appel de la fonction "mot" avec les arguments correspondant à la liste d'expressions *)
```

- type definition

  C'est le type des définitions de procédure. Il faut stocker son nom (type mot), les arguments (type expr list) qu'elle utilise et la liste d'instructions (type instruction list) qu'elle exécute.

```
type definition  =
            Def of mot*(expr list)*(instruction list) ;;
```

- type programme

  C'est le type des programmes LOGO : un programme est un ensemble de définitions de procédures (type definition list) suivi d'un bloc principal (type instruction list) où les fonctions seront très probablement appelées.

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

- type environnement

  C'est le type des environnements contenant les variables accompagnées de leur valeur associée dans cet environnement (type (expr*float) list) et les définitions de procédures (type definition list).

```
type environnement =
        En of ((expr*float) list)*(definition list);;
```

- type etat

  C'est le type des états du système. Il contient toute les informations concernant le curseur graphique ainsi que les modes employés (FILL et MOVE≠JUMP) et quelles instructions restent à exécuter, chaque instruction étant accompagnée de l'environnement dans lequel elle devra être exécutée.

  Ordre des composantes du type etat :

  - coordonnées cartésiennes : (float*float)
  - orientation en degré par rapport à l'axe des abscisses : float
  - proportion de rouge, vert et bleu pour la couleur : (float*float*float)
  - mode fill activé : bool
  - trace activé (MOVE) : bool
  - liste d'instructions restantes (accompagnées de leur environnement respectif) : (instruction*environnement) list

```
type etat =
    Etat of (float*float)*float*(float*float*float)*bool*bool*((instruction*environnement) list);;
```

Les fonctions

- ✔ value lit_bloc : flux_lecture -> instruction list

  Lecture d'un bloc BEGIN...END sachant que le mot-clé BEGIN est déjà lu. Le mot-clé END est à lire en dernier.
  On filtre la lecture d'un mot-clé (lit_mot_clé flux) avec tout les mots-clés définis. De plus, lorsqu'on lit un mot, le curseur de lecture est déplacé.

  Raffinage :

  - ✗ fonction extraction : 'a list -> 'a qui permet de renvoyer seulement le premier élément d'une liste. Utile avec la fonction lit_arguments pour les instructions rotate, jump, move et repeat. En effet, la fonction lit_arguments renvoie une liste et dans le cas des instructions rotate, jump et move, on sait que la liste se résume à un seul élément. De même pour le nombre de répétitions du Repeat.

  - ✗ Fonction triplet : 'a list -> 'a*'a*'a qui permet de renvoyer les 3 premiers éléments d'une liste sous la forme d'un triplet. Utile avec la fonction lit_arguments pour l'instruction color. En effet, de même que pour la fonction précédente, la fonction lit_arguments renvoyant une liste, on récupère les 3 arguments nécessaires à l'instruction color.

  Cas délicats :

  - ✗ Bien que le mot-clé BEGIN soit supposé déjà lu, pour éviter des erreurs inutiles, on continue de lire le bloc si jamais le mot-clé BEGIN est lu. De plus, cela sera utile pour lire les instructions dans un THEN, ELSE, REPEAT ou CALL.

  - ✗ Lors de la lecture d'un IF, on lit le test avec la fonction lit_condition puis le bloc d'instructions du THEN suivi de celui du ELSE.

  - ✗ Lors de la lecture d'un REPEAT, on lit le nombre de fois où le bloc d'instructions devra être exécuté, puis on lit ce bloc d'instructions.

  - ✗ Lors de la lecture d'un CALL, on lit le nom de la procédure (lu avec lit_mot_cle flux qui renvoie IDENT("nom")) puis ses arguments.

  - ✗ La lecture de DEF renvoie une erreur car un DEF ne pourra jamais se trouver à l'intérieur d'un bloc BEGIN...END.

Récursivité : la fonction lit_bloc est récursive, son cas terminal est la lecture du mot clé END. Dans le cas où le END aurait été oublié à la fin du bloc, s'il y a une définition après, la lecture d'un DEF renvoie l'erreur correspondante, s'il n'y a plus rien, une erreur est renvoyée :

```
#Uncaught exception: Failure "lexing: empty token"
```

Algorithme complet sur la page suivante :

```
Soit lit_bloc(flux) : filtrer  lit_mot_cle(flux) avec

                BEGIN       -> lit_bloc(flux)

        ou      ROTATE      -> Soit argument=extraction(lit_arguments(flux))
                                   dans Rotate(argument)::lit_bloc(flux)

        ou      MOVE        -> Soit argument=extraction(lit_arguments(flux))
                                   dans Move(argument)::lit_bloc(flux)

        ou      JUMP        -> Soit argument=extraction(lit_arguments(flux))
                                   dans Jump(argument)::lit_bloc(flux)

        ou      COLOR       -> Soit arguments=triplet(lit_arguments(flux))
                                   dans Color(arguments)::lit_bloc(flux)

        ou      FILL        -> Fill::lit_bloc(flux)

        ou      NOFILL      -> Nofill::lit_bloc(flux)

        ou      IF          -> Soit condition=lit_condition(flux)
                                   dans  Soit inst_then=lit_bloc(flux)
                                   dans  Soit inst_else=lit_bloc(flux)
                                   dans If(condition,inst_then,inst_else)::lit_bloc(flux)

        ou      THEN        -> lit_bloc(flux)

        ou      ELSE        -> lit_bloc(flux)

        ou      REPEAT      -> Soit nombre=extraction(lit_arguments(flux))
                                   dans  Soit inst_repeat=lit_bloc(flux)
                                         dans Repeat(nombre,inst_repeat)::lit_bloc(flux)

        ou      CALL        -> Soit nom=lit_mot_cle(flux)
                                   dans  Soit arguments=lit_arguments(flux)
                                         dans Call(nom,arguments)::lit_bloc(flux)

        ou      DEF         -> Erreur « Pas de DEF dans un BEGIN...END »

        ou      END         -> liste_vide

        ou      IDENT(string)-> Erreur « Pas de IDENT dans un fichier logo »
```

✔ value extraction : 'a list -> 'a

Extrait le premier élément d'une liste.

```
Soit  extraction(liste) : filtrer liste avec
                    liste_vide          -> Erreur « La liste est vide ! »
          ou        tête::liste_vide    -> tête
          ou        _                   -> Erreur « La liste contient plus d'un élément »
```

✔ value triplet : 'a list -> 'a*'a*'a

Extrait les 3 premiers éléments d'une liste sous la forme d'un triplet.

```
Soit  triplet(liste) : filtrer liste avec
                    liste_vide          -> Erreur « La liste est vide ! »
          ou        a::b::c::liste_vide -> (a,b,c)
          ou        _                   -> Erreur « La liste ne contient pas exactement 3 éléments »
```

✔ value lit_definition : flux_lecture -> definition

Lecture d'une définition de procédure : une définition comprend son nom, les paramètres et le bloc d'instructions associées. Le mot-clé DEF est supposé DEJA LU.
On lit donc d'abord le nom de la procédure puis ses paramètres formels et enfin le bloc d'instructions qu'elle exécute lors de son appel.

```
Soit  lit_definition(flux) : Soit nom=lit_mot_cle(flux)
                                dans  Soit arguments=lit_arguments(flux)
                                      dans Def(nom,arguments,lit_bloc(flux))
```

✔ value lit_programme : flux_lecture -> programme

Lecture d'un programme qui comprend une liste de définitions de procédures ainsi qu'un bloc principal d'instructions.
On utilise une fonction auxiliaire en faisant passer en paramètres la liste de définitions de procédures.

Récursivité : la fonction auxiliaire est récursive, son cas terminal correspond à la lecture du bloc principal, s'il n'y en a pas, l'erreur renvoyée est :

```
#Uncaught exception: Failure "lexing: empty token"
```

Algorithme complet :

```
Soit  lit_programme(flux) :
        Soit aux(flux,liste_definitions) : filtrer mot_clé avec
                        DEF    -> aux(flux,lit_definition(flux)::liste_definitions)
                ou      BEGIN -> Prog(liste_definitions,lit_bloc(flux))
                ou      _      -> Erreur « Erreur de syntaxe : oubli probable d'un BEGIN
                                                                    ou d'un DEF »
        dans aux(flux,liste_vide)
```

- value evalue_expression : environnement -> expr -> float

  Évaluation d'une expression (type expr) dans un environnement (type environnement). On filtre l'expression qu'on cherche à évaluer avec tout les cas du type expr.

  Pour les opérations arithmétiques, on évalue chacune des expressions puis on effectue l'opération correspondante.

  Pour le cas Var(string) il faut regarder dans l'environnement la valeur de cette variable si elle existe. Pour ce faire, on filtre l'environnement : quand on trouve la variable on renvoie sa valeur, sinon on cherche dans la suite des couples. Enfin si on ne la trouve pas, on renvoie un message d'erreur.

  Pour les fonctions trigonométriques, CaML fonctionne en radians, il faut donc d'abord évaluer l'expression à l'intérieur de la fonction puis la convertir en radians avant de lui appliquer la fonction CaML correspondante.

  Récursivité : la fonction est récursive et elle possède 2 cas terminaux, l'expression Const qui renvoie directement son argument, et l'expression Var qui renvoie la valeur de la variable dans l'environnement ou bien une erreur si cette variable n'appartient pas à l'environnement.

  Algorithme complet sur la page suivante :

```
Soit  evalue_expression(environnement,expression) : filtrer expression avec

            Plus(expr1, expr2)          ->
        evalue_expression(environnement,expr1)+evalue_expression(environnement,expr2)

    ou      Moins(expr1, expr2)         ->
        evalue_expression(environnement,expr1)-evalue_expression(environnement,expr2)

    ou      Div(expr1, expr2)           ->
        evalue_expression(environnement,expr1)/evalue_expression(environnement,expr2)

    ou      Mult(expr1, expr2)          ->
        evalue_expression(environnement,expr1)*evalue_expression(environnement,expr2)

    ou      Var(string)                 -> filtrer environnement avec
                                               En((str,val)::q,_) si str=string -> val
                                           ou  En((str,val)::q,liste_def)       ->
                                           evalue_expression(En(q,liste_def),expression)
                                           ou  _                                ->
                                           Erreur « La variable n'appartient pas à l'environnement »

    ou      Const(float)                -> float
    ou      Cosinus(expr)               -> cos(evalue_expression(environnement,expr)*π/180)
    ou      Sinus(expr)                 -> sin(evalue_expression(environnement,expr)*π/180)
    ou      Tangente(expr)              -> tan(evalue_expression(environnement,expr)*π/180)
```

✔ value evalue_condition : environnement -> test -> bool

Évaluation d'une condition (type test) dans l'environnement (type environnement).
On filtre la condition qu'on cherche à évaluer avec tout les cas du type test.

. Récursivité : la fonction est récursive et elle possède 2 cas terminaux, les tests Equal et InfEq qui ont besoin de evalue_expression et qui renvoie directement un booléen.

Algorithme complet sur la page suivante :

```
Soit  evaluer_condition(environnement,test) : filtrer test avec

                    Equal(expr1,expr2)          ->
Si evaluer_expression(environnement,expr1)=evaluer_expression(environnement,expr1) alors Vrai
                                                                                    sinon Faux

          ou        InfEq(expr1,expr2)          ->
Si evaluer_expression(environnement,expr1)<=evaluer_expression(environnement,expr1) alors Vrai
                                                                                    sinon Faux

          ou        And(test1,test2)            ->
Si evaluer_expression(environnement,expr1) et evaluer_expression(environnement,expr1) alors Vrai
                                                                                    sinon Faux

          ou        Ou(test1,test2)             ->
Si evaluer_condition(environnement,expr1) ou evaluer_condition(environnement,expr1) alors Vrai
                                                                                    sinon Faux

          ou        Not(test)                   -> Si evaluer_condition(environnement,test) alors Faux
                                                                                    sinon Vrai
```

✔ value execute_instruction : environnement -> instruction -> etat -> etat

Exécution d'une instruction (type instruction) dans un environnement (type environnement) depuis la position (type etat) et renvoie le nouvel état atteint.
On filtre l'instruction avec toutes les instructions déclarées. Seul le cas du Call n'est pas traité dans ce programme, en effet, j'ai préféré le traiter dans le execute_programme car une instruction Call ne modifie pas seulement l'état du système mais également l'environnement.

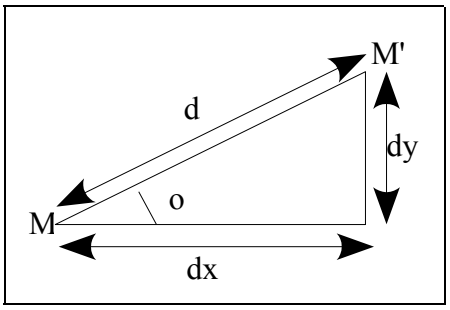
Raffinage :

✗ fonction add_env : instruction 'a list -> 'b -> ('a * 'b) list qui devra permettre de créer la dernière composante de « état » en associant à chaque instruction de la liste ('a list), l'environnement donné ('b).

Détail du traitement des instructions :

✗ Rotate : on modifie simplement l'orientation du curseur grâce à la fonction evalue_expression en n'oubliant pas de convertir le résultat en radians.

✗ Move : pour déplacer le curseur de M à M' d'une distance d, il faut connaître son orientation par rapport à l'axe des abscisses afin d'obtenir ses nouvelles coordonnées cartésiennes. Schéma :



dx = d*cos(o)
dy = d*sin(o)

On calcule donc les nouvelles coordonnées cartésiennes afin de modifier l'état, de plus on place le booléen trace à **Vrai** .

✗ Jump : idem que pour l'instruction Move sauf que l'on place le booléen trace à **Faux** .

✗ Color : on modifie seulement les 3 composantes gérant la couleur en évaluant chacune des expressions.

- Fill : on modifie le booléen fill en le mettant à **Vrai** .

- Nofill : on modifie le booléen fill en le mettant à **Faux** .

- If : on commence par évaluer la condition grâce à la fonction evalue_condition, puis, si celle-ci est **Vrai** on modifie la liste d'instructions du Then avec la fonction add_env afin d'obtenir une liste correspondant au type etat, puis on l'insère en tête de la liste de couples (instruction*environnement). Si la condition est évaluée **Faux**, on fait la même chose mais avec la liste d'instructions du Else.

- Repeat : si le nombre de répétitions est évaluée strictement inférieur à 1, on ne fait rien. Sinon on évalue ce nombre puis on appelle récursivement la fonction execute_instruction avec comme instruction le même Repeat dont le nombre de répétitions a été décrémenté et on ajoute à la liste de couples (instruction*environnement) la liste d'instructions du Repeat après l'avoir modifiée avec la fonction add_env.

Récursivité : La fonction est récursive seulement pour le cas du Repeat, où elle se rappelle en décrémentant le nombre de répétition, et dans le cas où ce nombre est strictement inférieur à 1, on renvoie simplement l'état.

Algorithme complet sur la page suivante :

```
Soit execute_instruction(environnement,instruction,Etat((x,y),o,(r,g,b),fill,trace,liste_instructions)) :
        filtrer instruction avec

            Rotate(expr)                ->
                Etat((x,y),o+evalue_expression(environnement,expr)*π/180,
                                                        (r,g,b),fill,trace,liste_instructions)

      ou    Move(expr)                  ->
                Etat((x+evalue_expression(environnement,expr)*cos(o),
             y+evalue_expression(environnement,expr)*sin(o),(r,g,b),fill,Vrai,liste_instructions)

      ou    Jump(expr)                  ->
                Etat((x+evalue_expression(environnement,expr)*cos(o),
           y+evalue_expression(environnement,expr)*sin(o),(r,g,b),fill,Faux,liste_instructions)

      ou    Color(expr1,expr2,expr3)  ->
                Etat((x,y),o,(evalue_expression(environnement,expr1),
                            evalue_expression(environnement,expr2),
                            evalue_expression(environnement,expr3),fill,trace,liste_instructions)

      ou    Fill                        ->
                Etat((x,y),o,(r,g,b),Vrai,trace,liste_instructions)

      ou    Nofill                      ->
                Etat((x,y),o,(r,g,b),Faux,trace,liste_instructions)

      ou    If(test,instructions_then,instructions_else)                    ->
                Si evalue_condition(environnement,test)       alors
      Etat((x,y),o,(r,g,b),fill,trace,add_env(instructions_then,environnement)@liste_instructions)
                                                              sinon
      Etat((x,y),o,(r,g,b),fill,trace,add_env(instructions_else,environnement)@liste_instructions)

      ou    Repeat(expr,_) si evalue_expression(environnement,expr)<1       ->
                Etat((x,y),o,(r,g,b),fill,trace,liste_instructions)

      ou    Repeat(expr,instructions_repeat)                                ->
                Soit nombre=Const(evalue_expression(environnement,expr)-1) dans
                execute_instruction environnement Repeat(nombre,instructions_repeat)
         Etat((x,y),o,(r,g,b),fill,trace,add_env(instructions_repeat,environnement)@liste_instructions)

      ou    Call(_,_)                   -> Erreur « Call traité dans le execute_programme »
```

✔ value add_env : 'a list -> 'b -> ('a * 'b) list

Associe à chaque instruction de la liste, l'environnement donné. Renvoie une liste de couples.

Récursivité : la fonction est récursive et son cas terminal est le cas de la liste vide qui sera atteint car la fonction est toujours rappelée avec seulement la queue de la liste.

```
Soit  add_env(liste_instructions,environnement) : filtrer liste_instructions avec
                    liste_vide     ->     liste_vide
          ou        tête::queue    ->     (tête,environnement)::add_env(queue,environnement)
```

- ✔ value execute_programme : programme -> unit

  Exécution d'un programme à partir de la position initiale (0,0) avec une orientation nulle, une couleur noire (0,0,0), pas de mode trace ni fill.

  Explications :

  - ✗ On utilise une fonction auxiliaire aux qui prend en paramètres l'environnement, l'état du système ainsi que les coordonnées cartésiennes de 2 points et le nombre de points mémorisés pour le mode Fill. La fonction aux est initialement appelée avec un environnement contenant seulement les définitions de procédure du programme, l'état correspondant à la position initiale (0,0) avec une orientation nulle, une couleur noire (0,0,0), pas de mode trace ni fill et les instructions du bloc principal, chacune accompagnées de l'environnement initial. Les coordonnées des points sont mises à (0,0) et le nombre de points mémorisés est mis à 0.

  - ✗ On exécute chaque instructions de la liste dans son propre environnement avec la fonction execute_instruction sauf pour l'instruction Call que l'on traite grâce à la fonction recherche_def.

  - ✗ Quand le mode Fill est actif on mémorise la position courante (même si l'on ne s'est pas encore déplacé) puis on mémorise chaque nouvelle position. Dès que l'on a 3 positions mémorisées, on trace le triangle correspondant et on garde en mémoire les 2 dernières positions pour continuer si le mode Fill reste actif.

  Raffinage :

  - ✗ recherche_def : environnement -> instruction -> environnement * instruction list

    Cette fonction sera utile pour l'exécution d'un Call permettre de récupérer un nouvel environnement avec les variables utiles à l'exécution de cette procédure. De plus, elle renverra la liste d'instructions de la procédure.

  Récursivité : la fonction auxiliaire est récursive et son cas terminal est le cas où la liste de couples d'instructions associées à leur environnement est vide. Ce cas est atteint dans tout les cas, en effet, lorsque l'on appelle la fonction execute_instruction, on enlève de cette liste l'instruction que l'on exécute. A part pour les instructions If, Repeat et Call, cette liste diminue donc strictement à chaque étape. Dans le cas de ces instructions particulières, la liste d'instructions augmente mais ceci ponctuellement.

Structure du programme :

On commence par filtrer l'état pour voir s'il reste des instructions à exécuter :

```
filtrer etat avec
                    Etat((x,y),o,(r,v,b),fill,trace,liste_vide)          ->
          ou        Etat((x,y),o,(r,v,b),fill,trace,(inst,inst_env)::queue)  ->
```

- Si la **liste est vide** on ne fait rien.

- Dans le cas où la liste de type (instruction*environnement) list **n'est pas vide**, on filtre en même temps cette instruction ainsi que son environnement associé afin de pouvoir traiter à part le cas du Call.

```
filtrer (inst,inst_env) avec
                Call(nom,liste_arguments),call_env      ->
          ou _,_                                        ->
```

  - **Si l'instruction est un Call**, on récupère le nouvel environnement avec les couples de variables des paramètres formels de la procédure associées à leur valeur dans cet appel. On récupère également la liste d'instructions de la procédure. Pour ce faire, on utilisera une fonction auxiliaire (recherche_def, cf. raffinage). En suite, on rappelle récursivement la fonction aux avec le nouvel environnement, le nouvel état auquel on a ajouté la liste des instructions de la procédure associées au nouvel environnement grâce à la fonction add_env et les même valeurs pour les coordonnées des 2 points ainsi que le nombre de points mémorisés.

```
Soit (nouv_env,liste_inst)=recherche_def(call_env,inst) dans
aux(nouv_env,Etat((x,y),o,(r,v,b),fill,trace,add_env(liste_inst,nouv_env)@queue),
(x0,y0),(x1,y1),mem)
```

  - **Sinon** on filtre le résultat de l'execute_instruction dans l'environnement de l'instruction et avec l'état de départ auquel on a retiré de la liste d'instructions celle que l'on exécute. Dans ce filtrage on se préoccupe essentiellement de l'état des booléens fill et trace. Il y a donc 4 cas distincts.

```
filtrer execute_instruction(inst_env,inst,Etat((x,y),o,(r,v,b),fill,trace,queue) avec
                                Etat((x',y'),o,(r,v,b),Faux,Faux,liste_inst)   ->
                        ou      Etat((x',y'),o,(r,v,b),Faux,Vrai,liste_inst)   ->
                        ou      Etat((x',y'),o,(r,v,b),Vrai,Faux,liste_inst)   ->
                        ou      Etat((x',y'),o,(r,v,b),Vrai,Vrai,liste_inst)   ->
```

- **fill=Faux, trace=Faux :** on se déplace jusqu'à la nouvelle position grâce à la fonction fmoveto (même si cette position est la même), on change de couleur avec la fonction fcolor (même si celle-ci est la même) puis on appelle récursivement la fonction aux avec l'environnement de l'instruction, l'état que l'on a filtrer et les mêmes valeurs pour les positions des points ainsi que pour le nombre de points mémorisés.

```
fmoveto(x',y') ; fcolor(r,v,b) ; aux(inst_env,Etat((x',y'),o,(r,v,b),Faux,Faux,liste_inst),(x0,y0),(x1,y1),0)
```

- **fill=Faux, trace=Vrai :** idem que précédent mais cette fois, on trace une ligne vers la nouvelle position avec la fonction flineto.

```
flineto(x',y') ; fcolor(r,v,b) ; aux(inst_env,Etat((x',y'),o,(r,v,b),Faux,Faux,liste_inst),(x0,y0),(x1,y1),0)
```

- **fill=Vrai, trace=Faux :** on effectue un test pour savoir si le curseur graphique a changé de position :

```
Si (x<>x') ou (y<>y')
```

  - **Si oui**, on filtre le nombre de points déjà mémorisés :

```
filtrer mem avec
                0 ->
            ou  1 ->
            ou  2 ->
            ou  _ ->
```

    - **0 point mémorisé :** on se déplace jusqu'à la nouvelle position avec la fonction fmoveto. Puis on appelle récursivement la fonction aux avec l'environnement de l'instruction, l'état que l'on a filtrer et on mémorise dans la première coordonnée la position courante. La deuxième coordonnée n'est pas modifiée et le nombre de points mémorisés passe à 1.

```
fmoveto(x',y') ; aux(inst_env,Etat((x',y'),o,(r,v,b),Vrai,Faux,liste_inst),(x',y'),(x1,y1),1)
```

- **1 point mémorisé :** idem sauf que la première coordonnée n'est pas modifiée car on mémorise la position courante la seconde coordonnées et que le nombre de points mémorisés passe à 2.

```
fmoveto(x',y') ; aux(inst_env,Etat((x',y'),o,(r,v,b),Vrai,Faux,liste_inst),(x0,y0),(x',y'),2)
```

- **2 points mémorisés :** on se déplace jusqu'à la nouvelle position avec la fonction fmoveto puis on trace le triangle plein ayant pour 3 sommets, les 2 coordonnées en paramètres de aux et la position courante avec la fonction ftriangle. En suite, on appelle récursivement la fonction aux avec l'environnement de l'instruction, l'état que l'on a filtrer et la seconde coordonnée devient la première, la position courante est mémorisée dans la seconde coordonnée et le nome de points mémorisés reste 2.

```
fmoveto(x',y') ;
ftriangle((x0,y0),(x1,y1),(x',y')) ; aux(inst_env,Etat((x',y'),o,(r,v,b),Vrai,Faux,liste_inst),(x1,y1),(x',y'),2)
```

- **autrement :** Erreur lors du mode Fill.

```
Erreur « Problème lors du FILL »
```

- **Si non**, on filtre le nombre de points déjà mémorisés :

```
sinon filtrer mem avec
        0 ->
    ou  _ ->
```

- **0 point mémorisé :** on change la couleur avec la fonction fcolor. Puis on appelle récursivement la fonction aux avec l'environnement de l'instruction, l'état que l'on a filtrer et on mémorise dans la première coordonnée la position courante. La deuxième coordonnée n'est pas modifiée et le nombre de points mémorisés passe à 1.

```
fcolor(r,v,b) ; aux(inst_env,Etat((x',y'),o,(r,v,b),Vrai,Faux,liste_inst),(x',y'),(x1,y1),1)
```

- **autrement :** idem sauf que les 2 coordonnées et le nombre de points mémorisés ne change pas.

```
fcolor(r,v,b)  ; aux(inst_env,Etat((x',y'),o,(r,v,b),Vrai,Faux,liste_inst),(x0,y0),(x1,y1),mem)
```

- **fill=Vrai, trace=Vrai** : idem en remplaçant les déplacement par des tracés de ligne avec la fonction flineto.

```
Si (x<>x') ou (y<>y')      alors  filtrer mem avec
                                        0 ->  flineto(x',y')
                                  aux(inst_env,Etat((x',y'),o,(r,v,b),Vrai,Vrai,liste_inst),
                                        (x',y'),(x1,y1),1)
                                  ou    1 ->  flineto(x',y')
                                  aux(inst_env,Etat((x',y'),o,(r,v,b),Vrai,Vrai,liste_inst),
                                        (x0,y0),(x',y'),2)
                                  ou    2 ->  flineto(x',y')
                                              ftriangle((x0,y0),(x1,y1),(x',y'))
                                  aux(inst_env,Etat((x',y'),o,(r,v,b),Vrai,Vrai,liste_inst),
                                        (x1,y1),(x',y'),2)
                                  ou    _ ->  Erreur « Problème lors du FILL »

                                                    sinon  filtrer mem avec
                                        0 ->  fcolor(r,v,b)
                                              aux(inst_env,Etat((x',y'),o,(r,v,b),
                                              Vrai,Vrai,liste_inst),(x',y'),(x1,y1),1)
                                  ou    _ ->  fcolor(r,v,b)
                                              aux(inst_env,Etat((x',y'),o,(r,v,b),
                                             Vrai,Vrai,liste_inst),(x0,y0),(x1,y1),mem)
```

Algorithme complet sur la page suivante :

```
Soit  execute_programme(Prog(liste_definitions,liste_instructions)) :
        Soit aux(environnement,etat,(x0,y0),(x1,y1),mem) : filtrer  etat avec

                Etat((x,y),o,(r,v,b),fill,trace,liste_vide)                -> ( )

        ou      Etat((x,y),o,(r,v,b),fill,trace,(inst,inst_env)::queue)    -> filtrer  (inst,inst_env) avec

                        Call(nom,liste_arguments),call_env              ->
                                Soit  (nouv_env,liste_inst)=recherche_def(call_env,inst)  dans
aux(nouv_env,Etat((x,y),o,(r,v,b),fill,trace,add_env(liste_inst,nouv_env)@queue),(x0,y0),(x1,y1),mem)

                        ou      _,_                                        ->
                filtrer  execute_instruction(inst_env,inst,Etat((x,y),o,(r,v,b),fill,trace,queue) avec

                                Etat((x',y'),o,(r,v,b),Faux,Faux,liste_inst)        ->
                                        fmoveto(x',y') ; fcolor(r,v,b)
                                        aux(inst_env,Etat((x',y'),o,(r,v,b),Faux,Faux,liste_inst),
                                                (x0,y0),(x1,y1),0)

                                ou      Etat((x',y'),o,(r,v,b),Faux,Vrai,liste_inst)        ->
                                        flineto(x',y') ; fcolor(r,v,b)
                                        aux(inst_env,Etat((x',y'),o,(r,v,b),Faux,Vrai,liste_inst),
                                                (x0,y0),(x1,y1),0)

                                ou      Etat((x',y'),o,(r,v,b),Vrai,Faux,liste_inst)        ->
                                        Si (x<>x') ou (y<>y')       alors  filtrer mem avec
                                                0 ->  fmoveto(x',y') ;
                                        aux(inst_env,Etat((x',y'),o,(r,v,b),Vrai,Faux,liste_inst),
                                                (x',y'),(x1,y1),1)
                                        ou      1 ->  fmoveto(x',y') ;
                                        aux(inst_env,Etat((x',y'),o,(r,v,b),Vrai,Faux,liste_inst),
                                                (x0,y0),(x',y'),2)
                                        ou      2 ->  fmoveto(x',y') ;
                                                      ftriangle((x0,y0),(x1,y1),(x',y'))
                                        aux(inst_env,Etat((x',y'),o,(r,v,b),Vrai,Faux,liste_inst),
                                                (x1,y1),(x',y'),2)
                                        ou      _ ->  Erreur « Problème lors du FILL »

                                                              sinon filtrer mem avec
                                                0 ->  fcolor(r,v,b)
                                                      aux(inst_env,Etat((x',y'),o,(r,v,b),
                                                      Vrai,Faux,liste_inst),(x',y'),(x1,y1),1)
```

```
                                ou      _ ->    fcolor(r,v,b)
                                                aux(inst_env,Etat((x',y'),o,(r,v,b),
                                                Vrai,Faux,liste_inst),(x0,y0),(x1,y1),mem)

                        ou      Etat((x',y'),o,(r,v,b),Vrai,Vrai,liste_inst)        ->
                                Si (x<>x') ou (y<>y')       alors  filtrer mem avec
                                        0 ->    flineto(x',y')
                                aux(inst_env,Etat((x',y'),o,(r,v,b),Vrai,Vrai,liste_inst),
                                        (x',y'),(x1,y1),1)
                                ou      1 ->    flineto(x',y')
                                aux(inst_env,Etat((x',y'),o,(r,v,b),Vrai,Vrai,liste_inst),
                                        (x0,y0),(x',y'),2)
                                ou      2 ->    flineto(x',y')
                                                ftriangle((x0,y0),(x1,y1),(x',y'))
                                aux(inst_env,Etat((x',y'),o,(r,v,b),Vrai,Vrai,liste_inst),
                                        (x1,y1),(x',y'),2)
                                ou      _ ->    Erreur « Problème lors du FILL »

                                                        sinon  filtrer mem avec
                                        0 ->    fcolor(r,v,b)
                                                aux(inst_env,Etat((x',y'),o,(r,v,b),
                                                Vrai,Vrai,liste_inst),(x',y'),(x1,y1),1)
                                ou      _ ->    fcolor(r,v,b)
                                                aux(inst_env,Etat((x',y'),o,(r,v,b),
                                                Vrai,Vrai,liste_inst),(x0,y0),(x1,y1),mem)

        dans aux(En(liste_vide,liste_definitions),
        Etat((0,0),0,(0,0,0),Faux,Faux,add_env(liste_instructions,En(liste_vide,liste_definitions))),
        (0,0),(0,0),0)
```

- ✔ value recherche_def : environnement -> instruction -> environnement * instruction list

  Si l'instruction est un Call, la fonction recherche dans la liste de définitions la procédure ayant le même nom que la première composante du Call et renvoie le nouvel environnement avec la première liste modifiée de telle sorte qu'elle contiennent les nouveaux couples de paramètres formels de la procédure chacun associés à sa valeur pour ce Call. Cette fonction renvoie également la liste d'instructions de la procédure.

  Raffinage :

  - ✗ fonction make_list : environnement -> 'a list -> expr list -> ('a*float) list qui va permettre de fabriquer la liste de couple pour la première composante de l'environnement.

  Explications :

  - ✗ On filtre d'abord l'instruction pour vérifier que l'on a bien appeler la fonction avec une instruction Call. Renvoie une erreur sinon.
  - ✗ On filtre ensuite l'environnement pour trouver la définition de la bonne procédure, et quand on la trouve, on fabrique la liste de couples pour le nouvel environnement et on remplace l'ancienne liste de couples par la nouvelle. En effet, la procédure aura uniquement besoin de ces paramètres.
  - ✗ On traite également dans ce filtrage le cas où la liste d'instructions de la procédure est vide.

  Récursivité : la fonction est récursive seulement dans le cas où la première définition de procédure de la liste de l'environnement n'est pas la bonne, la fonction est alors rappelée avec la queue de la liste des définitions de procédures. Si la définition recherchée appartient à cet environnement, le cas terminal sera le cas où la définition est trouvée, sinon une erreur est renvoyée.

  Algorithme complet sur la page suivante :

```
Soit recherche_def(environnement,instruction) : filtrer instruction avec

        Call(nom_proc,liste_arg)   -> filtrer environnement avec

              En(_,liste_vide)                                                    ->
                    Erreur « L'environnement ne contient pas la définition »

        ou    En(_,Def(nom,liste_expr,liste_vide)::queue) quand nom=nom_proc      ->
                    environnement,liste_vide

        ou    En(liste_var,Def(nom,liste_expr,liste_inst)::queue) quand nom=nom_proc   ->
                        En(make_list(environnement,liste_expr,liste_arg),
                              Def(nom,liste_expr,liste_inst)::queue),liste_inst

        ou    En(liste_var,tête::queue)                                           ->
                    recherche_def(En(liste_var,queue),instruction)

ou      -                          -> Erreur « La fonction recherche_def doit être
                                              appelé avec une instruction Call »
```

✔ value make_list : environnement -> 'a list -> expr list -> ('a*float) list

Fabrique une liste de couples à partir d'un liste quelconque et d'une liste d'expressions. Le $k^{ième}$ élément de cette liste est un couple dont le premier élément est le $k^{ième}$ de la première liste et le second l'évaluation du $k^{ième}$ élément de la seconde.
Utile pour créer la première liste d'un environnement.

Récursivité : la fonction est récursive et son cas terminal normal est celui où les 2 listes sont vides en même temps. Sinon, si une liste se vide avant l'autre, une erreur est renvoyée.

```
Soit make_list(environnement,liste_argu,liste_expr) : filtrer (liste_argu,liste_expr) avec
                liste_vide,liste_vide                  ->    liste_vide
        ou      tête1::queue1, tête2::queue2     ->
    (tête1,evalue_expression(environnement,tête2))::make_list(environnement,queue1,queue2)
        ou      _                                  ->    Erreur « Les 2 listes n'ont pas le même
                                                                nombre d'éléments »
```

Les tests de validation

Les tests unitaires

✔ lit_bloc

```
MOVE (n)
IF (p <= 0)
THEN
  BEGIN
  END
ELSE
  BEGIN
  ROTATE (10)
  CALL fougere (n*0.8, p-1)
  END
JUMP (-n)
REPEAT (4)
        BEGIN
        JUMP (10)
        ROTATE (30)
        END
END
```

✔ *Fonctionnement nominal :*

```
#lit_bloc (ouvre_fichier("lit_bloc.logo"));;
- : instruction list =
 [Move (Var "n");
  If
   (InfEq (Var "p", Const 0.0), [],
    [Rotate (Const 10.0);
     Call
      (IDENT "fougere",
       [Mult (Var "n", Const 0.8); Moins (Var "p", Const 1.0)])]);
  Jump (Moins (Const 0.0, Var "n"));
  Repeat (Const 4.0, [Jump (Const 10.0); Rotate (Const 30.0)])]
```

```
MOVE (n)
IF (p <= 0)
THEN
  BEGIN
  END
ELSE
  BEGIN
  ROTATE (10)
  CALL fougere (n*0.8, p-1)
  END

DEF defaut ()
BEGIN
COLOR (0,0,0)
END

END
```

✔ *Erreur dûe à un DEF dans le bloc BEGIN...END :*

```
#lit_bloc (ouvre_fichier("lit_bloc_def.logo"));;
Uncaught exception: Failure "Pas de DEF possible dans un BEGIN...END"
```

✔ extraction

✔ *Fonctionnement nominal :*

```
#extraction [1;2];;
- : int = 1
```

✔ *Erreurs :*

```
#extraction [];;
Uncaught exception: Failure "La liste est vide !"

#extraction [1;2];;
Uncaught exception: Failure "la liste ne contient pas qu'un seul élément"
```

✔ triplet

✔ *Fonctionnement nominal :*

```
#triplet [3;2;8];;
- : int * int * int = 3, 2, 8
```

✔ *Erreurs :*

```
#triplet [];;
Uncaught exception: Failure "La liste est vide !"

#triplet [1;2];;
Uncaught exception: Failure "La liste ne contient pas exactement 3 éléments !"

#triplet [1;2;4;8];;
Uncaught exception: Failure "La liste ne contient pas exactement 3 éléments !"
```

✔ lit_definition

```
retour (l)
BEGIN
        ROTATE (30)
        JUMP(-l)
        ROTATE (-90)
END
```

✔ *Fonctionnement nominal :*

```
#lit_definition (ouvre_fichier "lit_def.logo");;
- : definition =
 Def
  (IDENT "retour", [Var "l"],
   [Rotate (Const 30.0); Jump (Moins (Const 0.0, Var "l"));
    Rotate (Moins (Const 0.0, Const 90.0))])
```

```
DEF retour (l)
BEGIN
        ROTATE (30)
        JUMP(-l)
        ROTATE (-90)
END
```

✔ *Erreur dûe à un DEF en trop :*

```
#lit_definition (ouvre_fichier "lit_def_erreur.logo");;
Uncaught exception: ErreurLecture
                    "Erreur ligne 1 : lexeme `retour` : Syntaxe erronée !"
```

✔ lit_programme

```
DEF fougere (n, p)
BEGIN
MOVE (n)
IF (p <= 0)
THEN
  BEGIN
  END
ELSE
  BEGIN
  ROTATE (10)
  CALL fougere
(n*0.8, p-1)
  ROTATE (-30)
  CALL fougere
(n*0.4, p-1)
  ROTATE (20)
  END
JUMP (-n)
END

BEGIN
JUMP   (100)
ROTATE (90)
JUMP   (100)
ROTATE (-90)
(* curseur en
(100, 100) *)
CALL fougere
(100, 10)
END
```

✔ *Fonctionnement nominal :*

```
#lit_programme (ouvre_fichier "fougere.logo");;

- : programme =
 Prog
  ([Def
    (IDENT "fougere", [Var "n"; Var "p"],
     [Move (Var "n");
      If
       (InfEq (Var "p", Const 0.0), [],
        [Rotate (Const 10.0);
         Call
          (IDENT "fougere",
           [Mult (Var "n", Const 0.8); Moins (Var "p", Const 1.0)]);
         Rotate (Moins (Const 0.0, Const 30.0));
         Call
          (IDENT "fougere",
           [Mult (Var "n", Const 0.4); Moins (Var "p", Const 1.0)]);
         Rotate (Const 20.0)]);
      Jump (Moins (Const 0.0, Var "n"))])],
   [Jump (Const 100.0); Rotate (Const 90.0); Jump (Const 100.0);
    Rotate (Moins (Const 0.0, Const 90.0));
    Call (IDENT "fougere", [Const 100.0; Const 10.0])])
```

```
DEF retour (l)
BEGIN
        ROTATE (30)
        JUMP(-l)
        ROTATE (-90)
END
```

✔ *Erreur car il n'y a pas de bloc principal :*

```
#lit_programme (ouvre_fichier "lit_def_erreur.logo");;
Uncaught exception: Failure "lexing: empty token"
```

```
BEGIN
      JUMP   (100)
      ROTATE (90)
      JUMP   (200)
      ROTATE (-90)
      MOVE (300)
      JUMP (-300)
      ROTATE (90)
END
```

- *Cas où il n'y a pas de définition de procédures :*

```
#lit_programme (ouvre_fichier "lit_prog_main.logo");;
- : programme =
 Prog
  ([],
   [Jump (Const 100.0); Rotate (Const 90.0); Jump (Const 200.0);
    Rotate (Moins (Const 0.0, Const 90.0)); Move (Const 300.0);
    Jump (Moins (Const 0.0, Const 300.0)); Rotate (Const 90.0)])
```

evalue_expression

- *Fonctionnement nominal :*

```
#evalue_expression (En([((Var("t")),100.)],[])) (Cosinus(Const(180.)));;
-   : float = -1.0

#evalue_expression (En([((Var("t")),100.)],[])) (Var("t"));;
-   : float = 100.0
```

- *Erreur :*

```
#evalue_expression (En([((Var("t")),100.)],[])) (Var("n"));;
Uncaught exception: Failure "La variable n'appartient pas à l'environnement."
```

evalue_condition

- *Fonctionnement nominal avec un test sur des expressions :*

```
#evalue_condition (En([((Var("t")),100.);((Var("n")),200.)],[])) (InfEq((Var("t")),(Var("t"))));;
-   : bool = true
```

- *Fonctionnement nominal avec un test sur des tests :*

```
#evalue_condition (En([((Var("t")),100.);((Var("n")),200.)],[]))
(And(InfEq((Var("t")),(Var("t"))),Equal((Var("t")),Const(150.))));;
-   : bool = false
```

✔ execute_instruction

✔ *Fonctionnement nominal :*

```
#execute_instruction (En([((Var("t")),100.);((Var("n")),200.)],[])) (Move(Var("t")))
(Etat((200.,200.),0.,(0.,0.,0.),false,false,[]));;
- : etat = Etat ((300.0, 200.0), 0.0, (0.0, 0.0, 0.0), false, true, [])

#execute_instruction (En([((Var("t")),100.);((Var("n")),200.)],[])) (Fill)
(Etat((200.,200.),0.,(0.,0.,0.),false,false,[]));;
- : etat = Etat ((200.0, 200.0), 0.0, (0.0, 0.0, 0.0), true, false, [])
```

✔ *Fonctionnement nominal avec une instruction IF :*

```
#execute_instruction (En([((Var("t")),100.);((Var("n")),200.)],[])) (If
       (InfEq (Var "t", Const 0.0), [],
        [Call
          (IDENT "fougere",
           [Mult (Var "t", Const 0.8); Moins (Var "n", Const 1.0)]);
         Rotate (Moins (Const 0.0, Const 30.0));
         Rotate (Const 20.0)])) (Etat((200.,200.),0.,(0.,0.,0.),false,false,[]));;
- : etat =
Etat
 ((200.0, 200.0), 0.0, (0.0, 0.0, 0.0), false, false,
  [Call
    (IDENT "fougere",
     [Mult (Var "t", Const 0.8); Moins (Var "n", Const 1.0)]),
   En ([Var "t", 100.0; Var "n", 200.0], []);
   Rotate (Moins (Const 0.0, Const 30.0)),
   En ([Var "t", 100.0; Var "n", 200.0], []);
   Rotate (Const 20.0), En ([Var "t", 100.0; Var "n", 200.0], [])])
```

✔ *Erreur :*

```
#execute_instruction (En([((Var("t")),100.);((Var("n")),200.)],[])) (Call
         (IDENT "fougere",
          [Mult (Var "t", Const 0.8); Moins (Var "n", Const 1.0)]))
(Etat((200.,200.),0.,(0.,0.,0.),false,false,[]));;
Uncaught exception: Failure "Le Call est traité dans la fonction execute_programme"
```

✔ add_env

✔ *Fonctionnement nominal :*

```
add_env [Rotate (Moins (Const 0.0, Const 30.0));(Move(Var("t")))]
(En([((Var("t")),100.);((Var("n")),200.)],[]));;
#- : (instruction * environnement) list =
 [Rotate (Moins (Const 0.0, Const 30.0)),
  En ([Var "t", 100.0; Var "n", 200.0], []);
  Move (Var "t"), En ([Var "t", 100.0; Var "n", 200.0], [])]
```

✔ execute_programme

Les tests de validation de cette fonction correspondent à ceux de la partie suivante : Jeu d'essais.

✔ recherche_def

✔ *Fonctionnement nominal :*

```
#recherche_def (En([((Var("t")),100.);((Var("n")),2.)],[Def(IDENT "fougere", [Var "n"; Var "p"],
     [Move (Var "n")])])) (Call(IDENT "fougere",
     [Mult (Var "t", Const 0.8); Moins (Var "n", Const 1.0)]));;
- : environnement * instruction list =
 En
  ([Var "n", 80.0; Var "p", 1.0],
   [Def (IDENT "fougere", [Var "n"; Var "p"], [Move (Var "n")])]),
 [Move (Var "n")]
```

✔ *Erreurs :*

```
#recherche_def (En([((Var("t")),100.);((Var("n")),200.)],[])) (Call(IDENT "fougere",
          [Mult (Var "t", Const 0.8); Moins (Var "n", Const 1.0)]));;
Uncaught exception: Failure "L'environnement ne contient pas la définition !"

#recherche_def (En([((Var("t")),100.);((Var("n")),2.)],[])) (Fill);;
Uncaught exception: Failure "La fonction recherche_def doit être appelé avec une instruction Call"
```

✔ make_list

✔ *Fonctionnement nominal :*

```
#make_list (En([((Var("t")),100.);((Var("n")),2.)],[])) [(Var("p"));(Var("c"))]
                                        [(Const(4.));(Moins (Var "n", Const 1.0))];;
-   : (expr * float) list = [Var "p", 4.0; Var "c", 1.0]
```

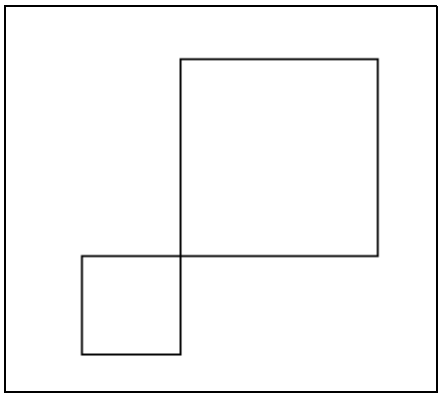
✔ *Erreur :*

```
#make_list (En([((Var("t")),100.);((Var("n")),2.)],[])) [(Var("p"));(Var("c"))] [(Const(4.))];;
Uncaught exception: Failure "Il n'y a pas le bon nombre d'arguments"
```

Jeu d'essais

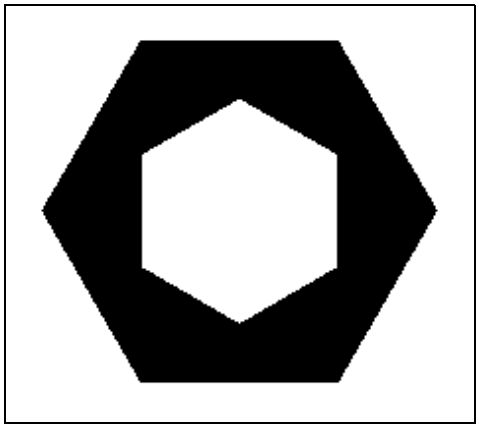
carre.logo

```
DEF carre (n)
BEGIN
REPEAT (4)
  BEGIN
  MOVE   (n)
  ROTATE (90)
  END
END

BEGIN
JUMP   (100)
ROTATE (90)
JUMP   (100)
ROTATE (-90)
(* curseur en (100, 100) *)
CALL carre (100)
ROTATE (180)
CALL carre (50)
END
```



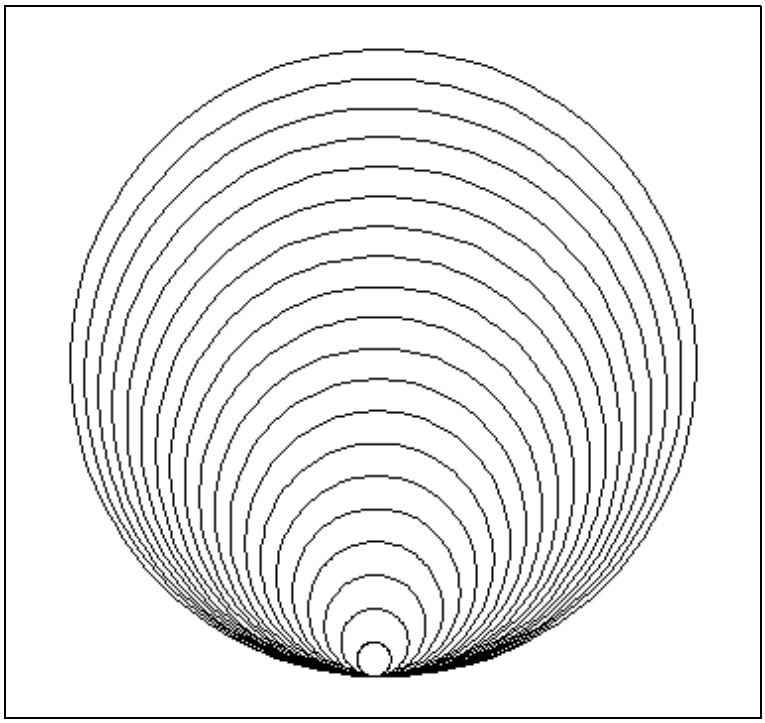
ecrou.logo

```
DEF ecrou (c, n)
BEGIN
REPEAT (n / 2)
  BEGIN
  FILL
  MOVE   (c)
  ROTATE (360 / n)
  MOVE   (c)
  ROTATE (360 / n)
  NOFILL
  END
MOVE   (c)
ROTATE (360 / n)
REPEAT (n / 2)
  BEGIN
  FILL
  MOVE   (c)
  ROTATE (360 / n)
  MOVE   (c)
  ROTATE (360 / n)
  NOFILL
  END
ROTATE (-360 / n)
MOVE   (-c)
END

BEGIN
JUMP   (200)
ROTATE (90)
JUMP   (100)
ROTATE (-90)
(* curseur en (200, 100) *)
CALL ecrou (100, 6)
END
```



cercle.logo

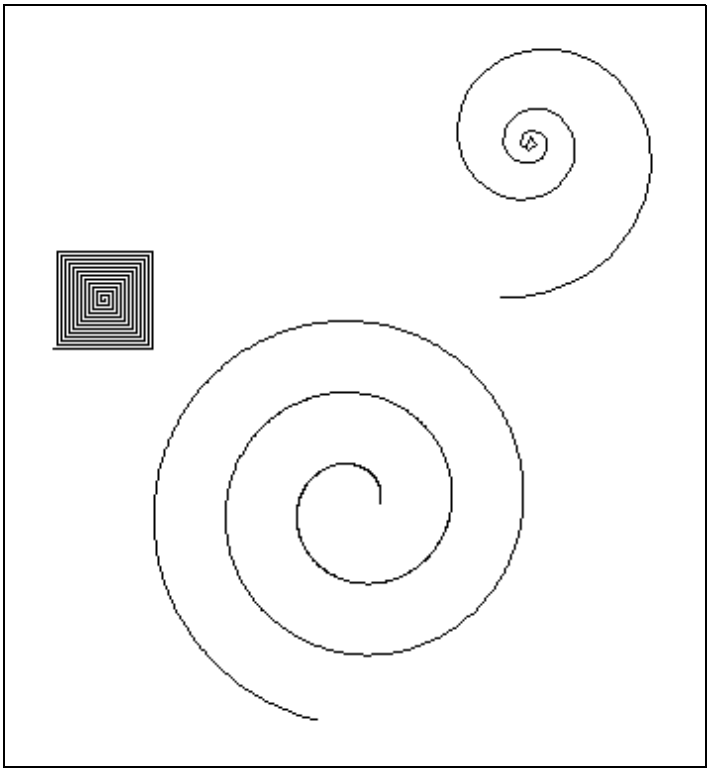
```
DEF cercle (r)
BEGIN
REPEAT (100)
  BEGIN
  MOVE   (r/100)
  ROTATE (360/100)
  END
END

DEF cercles (n, r)
BEGIN
IF (n <= 0)
THEN
  BEGIN
  END
ELSE
  BEGIN
  CALL cercle (20*sin(10*r)+50*r)
  CALL cercles (n-1, r+1)
  END
END

BEGIN
JUMP   (200)
ROTATE (90)
JUMP   (100)
ROTATE (-90)
(* curseur en (200, 100) *)
CALL cercles (20, 1)
END
```



spirales.logo

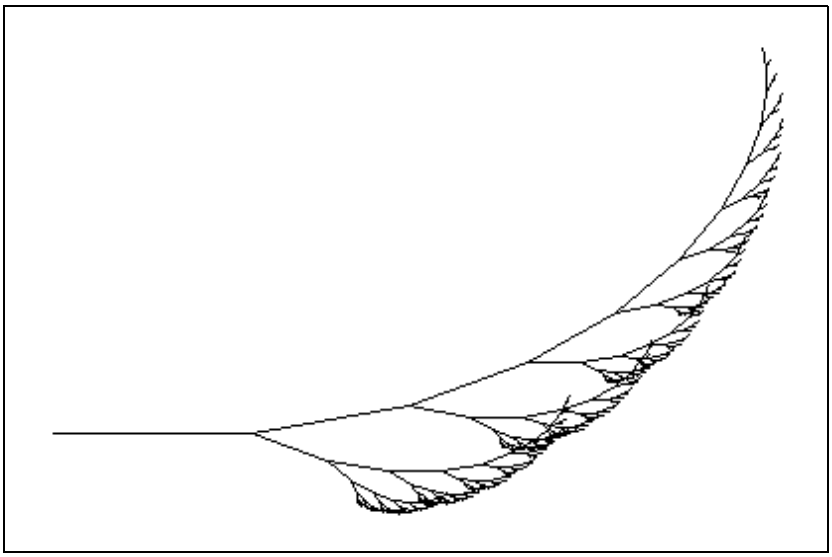
```
DEF spiraleCarre (n)
BEGIN
IF (n < 0)
THEN
  BEGIN
  END
ELSE
  BEGIN
  MOVE   (n)
  ROTATE (90)
  MOVE   (n-1)
  ROTATE (90)
  MOVE   (n-2)
  ROTATE (90)
  MOVE   (n-3)
  ROTATE (90)
  CALL spiraleCarre (n-4)
  END
END

DEF spiraleLineaire (r, k, p)
BEGIN
IF (p <= 0)
THEN
  BEGIN
  END
ELSE
  BEGIN
  MOVE   (r/57.3)
  ROTATE (1)
  CALL spiraleLineaire (r+k, k, p-1)
  END
END
```

```
DEF spiraleEscargot (r, p)
BEGIN
IF (p <= 0)
THEN
  BEGIN
  END
ELSE
  BEGIN
  MOVE   (r)
  ROTATE (360/p)
  CALL spiraleEscargot (r, p-1)
  END
END

BEGIN
JUMP   (100)
ROTATE (90)
JUMP   (300)
ROTATE (-90)
(* curseur en (100, 300) *)
CALL spiraleCarre (50)
JUMP (200)
CALL spiraleEscargot (5, 100)
JUMP (-200)
CALL spiraleLineaire (10, 0.1, 1000)
END
```



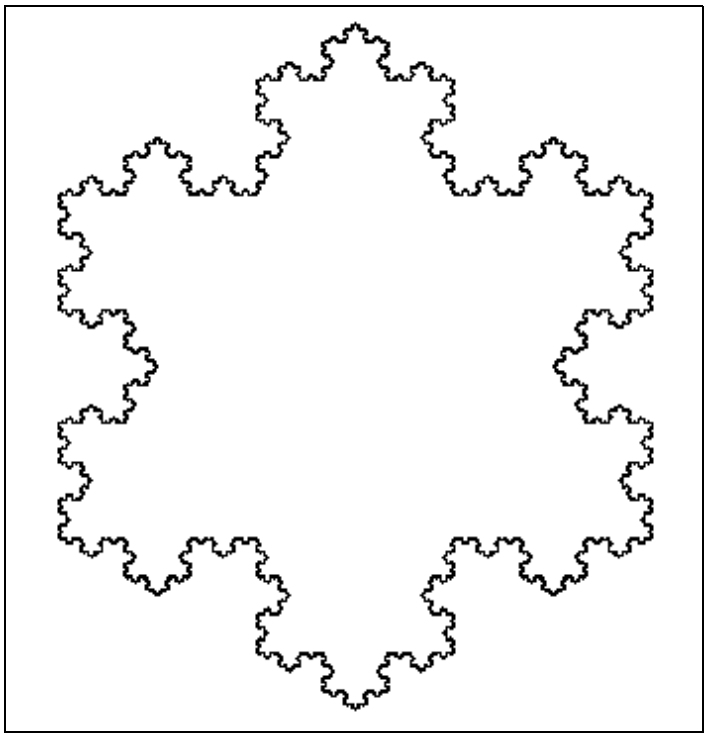
fougere.logo

```
DEF fougere (n, p)
BEGIN
MOVE (n)
IF (p <= 0)
THEN
  BEGIN
  END
ELSE
  BEGIN
  ROTATE (10)
  CALL fougere (n*0.8, p-1)
  ROTATE (-30)
  CALL fougere (n*0.4, p-1)
  ROTATE (20)
  END
JUMP (-n)
END

BEGIN
JUMP   (100)
ROTATE (90)
JUMP   (100)
ROTATE (-90)
(* curseur en (100, 100) *)
COLOR (0,200,0)
CALL fougere (100, 10)
END
```



vonkoch.logo

```
DEF vonkoch (n, p)
BEGIN
IF (p <= 0)
THEN
  BEGIN
  MOVE   (n)
  END
ELSE
  BEGIN
  CALL vonkoch (n/3, p-1)
  ROTATE (60)
  CALL vonkoch (n/3, p-1)
  ROTATE (-120)
  CALL vonkoch (n/3, p-1)
  ROTATE (60)
  CALL vonkoch (n/3, p-1)
  END
END

DEF etoile (n, p)
BEGIN
REPEAT (3)
  BEGIN
  CALL vonkoch (n, p)
  ROTATE (-120)
  END
END

BEGIN
JUMP   (150)
ROTATE (90)
JUMP   (300)
ROTATE (-90)
(* curseur en (150, 300) *)
CALL etoile (300, 6)
END
```

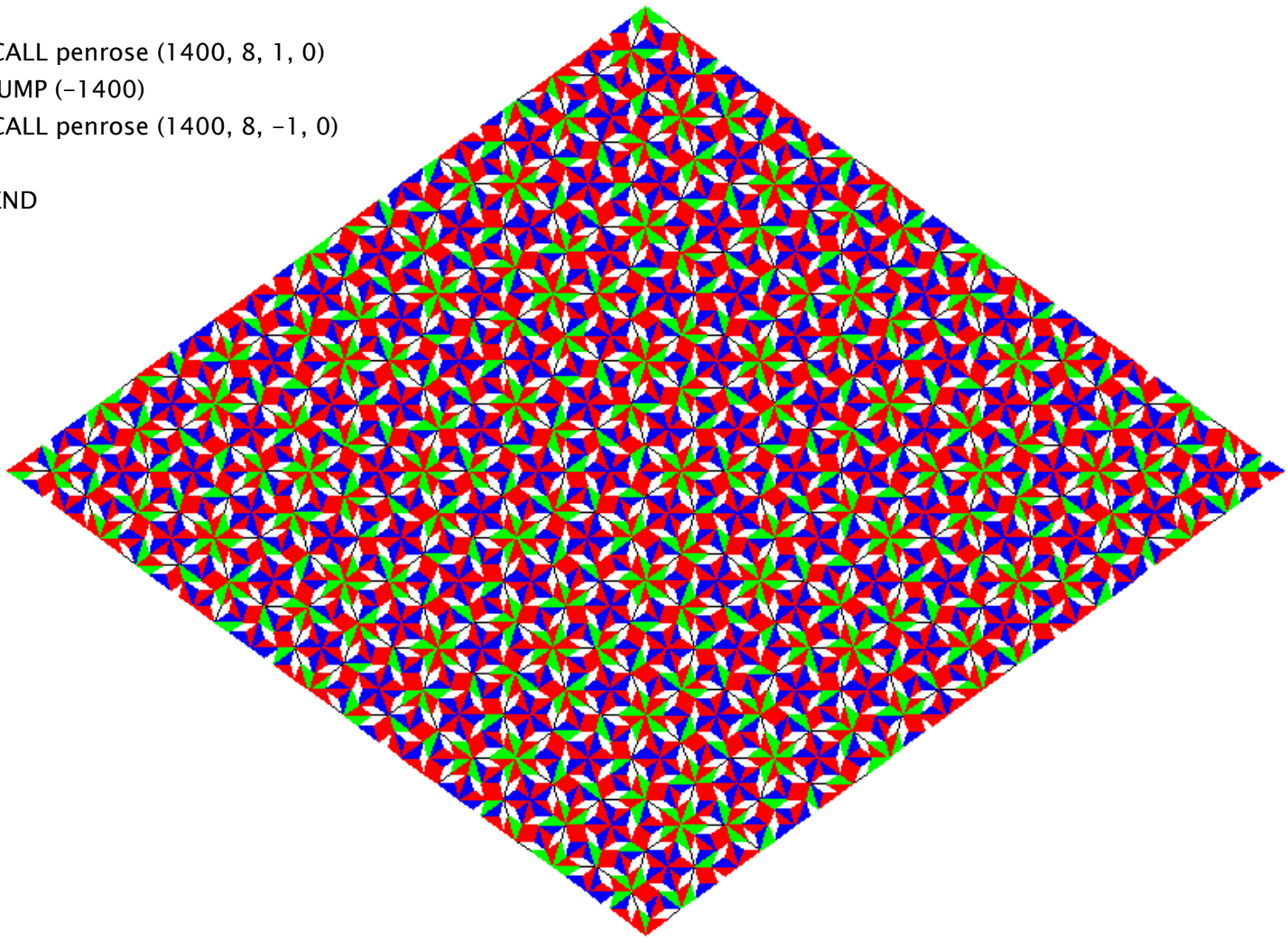


penrose.logo

```
DEF color (s, c)
BEGIN
IF (s=1)
THEN
BEGIN
IF (c=0)
THEN
BEGIN
COLOR (0, 250, 0)
END
ELSE
BEGIN
COLOR (250, 0, 0)
END
END
ELSE (* s = -1 *)
BEGIN
IF (c=0)
THEN
BEGIN
COLOR (250, 0, 0)
END
ELSE
BEGIN
COLOR (0, 0, 250)
END
END
END

DEF default()
BEGIN
COLOR (0, 0, 0)
END
```

```
DEF penrose (n, p, s, c)
BEGIN
IF (p <= 0)
THEN
  BEGIN
  CALL color (s, c)
  FILL
  ROTATE (-36*s)
  MOVE   (n*0.618033989)
  ROTATE (72*s)
  MOVE   (n*0.618033989)
  ROTATE (-36*s)
  NOFILL
  CALL default ()
  END
ELSE
  BEGIN
  IF (c = 0)
  THEN
    BEGIN
    JUMP   (n*0.618033989)
    ROTATE (180)
    CALL   penrose (n*0.618033989, p-1, -s, 1)
    ROTATE (180)
    JUMP   (n)
    ROTATE (-144*s)
    CALL   penrose (n*0.618033989, p-1, s , 0)
    JUMP   (-n*0.618033989)
    ROTATE (144*s)
    END
  ELSE
    BEGIN
    JUMP   (n*0.618033989)
    ROTATE (180)
    CALL   penrose (n*0.618033989, p-1, -s, 1)
    ROTATE (108*s)
    JUMP   (n*0.618033989)
    ROTATE (180)
    CALL   penrose (n*0.618033989, p-1,  s, 1)
    ROTATE (-108*s)
    JUMP   (n)
    ROTATE (-144*s)
    CALL   penrose (n*0.618033989, p-1,  s, 0)
    IF (p = 1)
    THEN
```

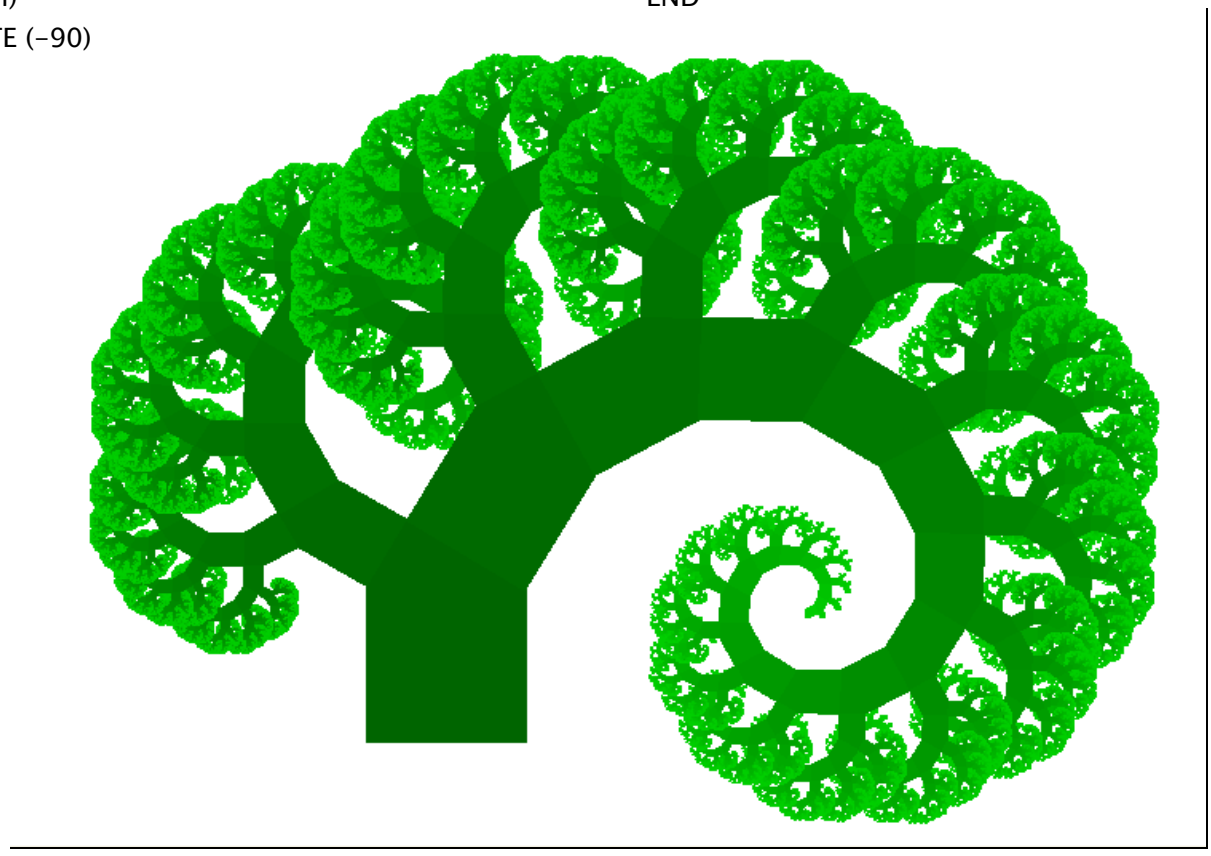
```
      BEGIN
        MOVE   (n*0.618033989*0.618033989)
        JUMP   (-n*0.618033989*0.618033989)
      END
    ELSE
      BEGIN
      END
    JUMP   (-n*0.618033989)
    ROTATE (144*s)
    END
  END
END

BEGIN
JUMP   (5)
ROTATE (90)
JUMP   (600)
ROTATE (-90)
(* curseur en (5, 600) *)
(* profondeur 8 *)
(*CALL penrose (400, 0, 1, 0)*)


CALL penrose (1400, 8, 1, 0)
JUMP (-1400)
CALL penrose (1400, 8, -1, 0)

END
```



arbre_pythagore.logo

```
DEF branche (l,c)
BEGIN
COLOR (0,c,0)
FILL
REPEAT (4)
  BEGIN
  MOVE (l)
  ROTATE (90)
  END
ROTATE (90)
JUMP (l)
ROTATE (-90)
JUMP (l)
ROTATE (90)
ROTATE (60)
MOVE (l*cos(30))
ROTATE (90)
MOVE (l*cos(60))
ROTATE (180)
NOFILL
END

DEF retour (l)
BEGIN
ROTATE (30)
JUMP(-l)
ROTATE (-90)
END
```

```
DEF arbre (l,c,p)
BEGIN
CALL branche (l,c)
IF (p<=0)
  THEN
  BEGIN
  END
  ELSE
  BEGIN
   CALL arbre ((l*cos(60)),c+5,p-1)
   JUMP (l*cos(60))
   ROTATE (-90)
   CALL arbre ((l*cos(30)),c+5,p-1)
   ROTATE (90)
   JUMP (-l*cos(60))
  END
CALL retour (l)
END

BEGIN
JUMP   (300)
ROTATE (90)
JUMP   (200)
ROTATE (-90)
CALL arbre (100,110,20)
END
```



Conclusion

Ce projet qui avait pour but la réalisation d'un interprète pour le mini-langage LOGO en CaML m'a permis de comprendre ce que pouvait être un projet de programmation. En effet, j'ai apprécié la liberté qui nous été laissée en ce qui concerne les types, cela m'a permis d'être libre dans la réalisation des fonctions. Bien que le raffinage général nous été donné par le biais des fonctions que nous avions à implémenter, il était intéressant d'essayer de décomposer ces fonctions, surtout lorsqu'elles étaient complexes.

J'ai rencontré les premières difficultés dès le départ, en effet la compréhension du sujet et des fichiers mli et ml qui nous été donnés m'a pris beaucoup de temps (environ 3h). En effet, il m'a fallu du temps pour comprendre le rôle et l'utilisation des types et fonctions déjà définis puis encore plus de temps pour comprendre le rôle de chaque type et de chaque fonction qu'il nous était demander de définir.

La réalisation du premier programme (lit_bloc) m'a pris beaucoup de temps (4h) car c'était la première fonction et je n'étais pas encore familiarisé avec les types et les fonctions du module logo_base.
Par la suite, j'ai rencontré quelques difficultés pour les fonctions execute_instruction et execute_programme. L'ensemble de la programmation m'a pris environ 30h.
Au départ, je n'avais pas associé un environnement à chaque instruction c'est pourquoi la plupart des fichiers logo ne fonctionnait pas.

Par la suite, la rédaction de ce rapport m'a pris environ 15h, en effet, il m'a fallu me souvenir des raisonnements que j'avais du faire pour faire mes choix de types et d'implémentation afin d'expliquer au mieux le fonctionnement de mes fonctions.

De plus, j'espère que vous aurez apprécié le résultat de mon propre fichier logo réalisant l'arbre de Pythagore, une fractale bien connue, car cela m'a pris du temps et m'a beaucoup intéressé.