Département IMA / 4A (S8)
Informatique Fondamentale 2012/2013

Laure Gonnord et Julien Forget
http://laure.gonnord.org/pro/

Projet Tutoré

Réalisation d'un Mini Logo

Introduction

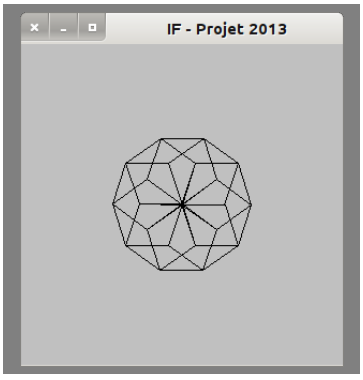
Le langage LOGO a été inventé dans les années 60 essentiellement à but pédagogique. C'est un langage très complet, donc nous n'étudierons que les primitives "graphiques", qui ont fait son succès dans les écoles primaires à la fin des années 1980.

Le programme ici sera une suite d'instructions qui permettent de faire bouger une "tortue" située initialement au milieu de la fenêtre graphique.

Dans ce tutorat, il s'agit d'écrire un compilateur d'un (sous-ensemble) du langage LOGO. Les instructions de déplacement de la "tortue" seront données sous forme d'un fichier texte, et votre programme sera capable de l'interpréter et de l'afficher dans une fenêtre. Par exemple si le fichier `tortue1.txt` contient une description de dessin :

```
./launch.sh tortue1.txt
```

fournira l'affichage :



Pour en savoir plus, vous pouvez vous reporter à la page Wikipédia qui est très complète :

http://fr.wikipedia.org/wiki/Logo_(langage)

1 Plantons le décor !

1.1 Syntaxe concrète textuelle et sémantique des programmes

Nous définissons maintenant le langage textuel complet sous forme de grammaire BNF (écriture standard) dans laquelle les mots clefs du langage sont écrits en minuscule pour les distinguer des non-terminaux. Attention, dans le langage, ils seront en majuscule. .

```
PROG        ::= LISTINST

LISTINST    ::= INST LISTINST
              | INST

INST        ::= INST_SHOW
              | INST_DEFVAR
              | INST_REPEAT
              | INST_TURTLE
              | INST_DEFPROC
              | INST_CALLPROC

INST_SHOW   ::= montre E
```

```
INST_DEFVAR ::= donne "VAR E

INST_REPEAT ::= repete n [ LISTINST ]

INST_TURTLE ::= AVANCE E
              |  TOURNED E
              |  TOURNEG E
              |  FIXEX E
              |  FIXEY E

INST_DEFPROC ::= pour VAR LISTINST end

INST_CALLPROC ::= VAR

E           ::=  ...
```

Explications :

- Un programme (`PROG`) est donc une suite d'instructions.
- E est une expression arithmétique formée de nombres, constantes, identifiants, opérateurs arithmétiques (+, *, ...) et d'autres opérateurs). La grammaire de E sera à compléter dans la section **??**.
- Une instruction (`INST`) est soit une `INST_SHOW` (évaluation d'une expression et impression de sa valeur), soit `INST_DEFVAR` (définition de variable), soit une `INST_REPEAT` (répétition d'expressions), soit `INST_TURTLE` (instructions pour la tortue), soit enfin `INST_DEFPROC` (définition d'une procédure) ou `INST_CALLPROC` (appel d'une procédure). Attention, les instructions ne sont pas séparées par un point virgule.
- Une instruction-show est constitué du mot clef `MONTRE` (en majuscule) suivi d'expression arithmétique (avec des variables).
- Une instruction-définition `DONNE "x` permet de donner une valeur à la variable x.
- Une instruction `REPEAT` permet de répéter la liste d'instructions passée en paramètre.
- Une instruction `AVANCE 40` avance la tortue de 40 pixels, une instruction `TOURNED 23.3` modifie son cap de 23.3 degrés vers la droite, `FIXEX` permet de fixer la nouvelle abscisse de la tortue...
- Une instruction `POUR toto ... END` permet de définir une procédure (liste d'instructions) nommée `toto`. L'appel se fait en écrivant `toto` (instruction `INST_CALLPROC`).

L'*évaluation* d'un programme consiste à réaliser l'évaluation dans l'ordre de l'écriture des instructions :

- L'évaluation d'une instruction-show consiste à imprimer la valeur de cette expression.
- L'évaluation d'une instruction-définition consiste à réaliser l'association entre la variable définie et une valeur d'expression.
- L'évaluation d'une instruction-repeat consiste à répéter n fois la liste d'instructions.

– L'évaluation d'une instruction-tortue consiste à calculer la nouvelle position de la tortue et à dessiner (si il est nouveau), le segment de droite sur la fenêtre graphique.
– L'évaluation d'une définition de procédure consiste à stocker l'association entre la variable et la liste d'instructions. L'évaluation de l'appel est uniquement l'évaluation de la suite d'instructions.

Exemple Un exemple de programme complet est décrit Figure **??**. Vérifier que ce programme est bien un programme syntaxiquement correct et réaliser son interprétation "à la main". Vous fournirez cette explication pas-à-pas dans votre rapport.

```
MONTRE PI + 2
DONNE "x (SOMME 2 5*8)
DONNE "y 42
MONTRE posx
POUR CARRE:
 REPETE 4 [AVANCE 50 TOURNED 90]
END
AVANCE 60
CARRE
TOURNED 180
AVANCE 60
```

FIGURE 1 – Exemple de programme LOGO

1.2 Travail demandé

On demande de traiter les différentes instructions de la grammaire précédente et d'afficher la trajectoire de la tortue dans une fenêtre graphique. Les étapes du travail sont détaillées dans le reste de l'énoncé.

2 Travail préliminaire

2.1 SVN

Tous les codes fournis pour le tutorat sont sur le SVN de l'école. On commencera donc par faire [1] :

```
svn checkout svn+ssh://synthe/ima4/ProjetIF
```

Cette archive a l'arborescence suivante :

```
api_graphique    // interface pour les dessins
expr             // expressions arithmétiques simples
launch.sh        // script pour lancer le programme
Makefile
testall.sh       // script pour lancer (tous) les tests
testsexpr        // fichiers de tests expr
testslogo        // fichier de tests langage LOGO
tools            // flex et cup
```

1. De l'extérieur, c'est plus compliqué, mais expliqué en cours de PA (voir la page WEB)

Remarque 1 *(2013) L'installation de jflex et jcup a déjà été vue en cours.*

Remarque 2 *JFlex a été téléchargé à l'adresse http://jflex.de/jflex-1.4.3.tar.gz et JavaCup à l'adresse http://www2.cs.tum.edu/projects/cup/java-cup-11a.jar.*

2.2 Installations, tests

Vous devez tout d'abord récupérer les outils `jflex` et `jcup` permettant de compiler votre grammaire, ainsi que les fichiers `Makefile`, `launch.sh` et `testall.sh` déjà écrits. Enfin, on travaillera à partir des expressions arithmétiques vues en cours.

- Installer JFlex et JCup si ce n'est pas déjà fait.
- Faire un premier `checkout` de votre projet :
  `svn checkout svn+ssh://synthe/ima4/<nombinome>`
- Copier à la racine de ce répertoire <nombinome> `Makefile`, `launch.sh` (compilation, exécution de vos programmes), ainsi que `testall.sh`.
- Ajouter les droits d'exécution aux fichiers `launch.sh` et `testall.sh`.
- Créer (toujours dans <nombinome>) un répertoire `iflogo` qui va contenir les sources du projet. Y copier `expr.flex`, `expr.cup`, `ASTExpr.java` et `Analyseur.java`. Ajouter `package iflogo;` à chaque début des fichiers précédents.
- Éditer `Makefile` et `launch.sh` pour indiquer les bons chemins vers `jflex` et `java-cup`, puis vérifier que tout compile bien.
- Créer un répertoire `tests` et y copier l'ensemble des tests fournis (attention, pas de cp récursif à partir du répertoire fourni, sous peine de cassage de votre dépôt).
- Vérifier que `./launch tests/ex01.txt` ne fait pas d'erreur à l'exécution.
- Ajouter à votre copie locale les fichiers à sauvegarder sur le serveur svn (`svn add`) et commitez (`svn commit`)

3 Expressions arithmétiques

Vous avez donc récupéré dans le dépôt SVN la grammaire des expressions arithmétiques vue en cours (avec construction d'arbre syntaxique). Il s'agit maintenant de la modifier (dans le flex, le cup, ou la classe `ASTExpr` suivant le cas) pour gérer :

- Les nombres entiers et réels, de la même façon. La valeur dans les deux cas sera un `double`.
- Les opérations binaires suivantes : addition (+), soustraction (-), multiplication (*), division (/) en n'oubliant pas les priorités.
- Les opérations unaires suivantes : sinus, cosinus.
- Les parenthèses.
- La constante Pi,

à la fin de cette partie, on sera donc capable de calculer l'expression suivante :

```
(2.8 + (12 / 3.5)) *  (PI+SIN(12.9))
```

On ajoutera aussi une fonction basique d'impression d'une `ASTExpr` dans le fichier `ASTExpr.java`. Créer des fichiers de test d'évaluation d'expressions (en plus de ceux du répertoire `testexpr`) et les utiliser pour tester votre parsing d'expression arithmétiques.

4 Traitement du langage Logo

Chacun des types d'instructions de la grammaire est traité séparément.

4.1 Instruction Show

- Écrire les classes Java (constructeur et attributs) `Prog.java` (classe pour un programme complet), `Instruction.java` (classe abstraite), et `InstructionShow.java` (classe pour les instructions constituées du mot clef `MONTRE` suivi d'une expression).
- Modifier la grammaire cup/flex pour traiter les programmes constitués d'une liste d'instructions (pour l'instant les instructions ne sont que des `InstructionShow`) :

```
non terminal Prog S;
non terminal List<Instruction>  ListInst;
non terminal Instruction OneInst;
[...]
S ::= ListInst:l  {: RESULT = new Prog(l); :}
;

ListInst ::=
      ListInst:l OneInst:i    {: l.add(i); RESULT = l; :}
     | OneInst:i              {: List<Instruction> l=new ArrayList<Instruction>();
                                 l.add(i);
                               RESULT = l; :}
;

OneInst ::= <avous>
```

FIGURE 2 – Cup pour traiter les listes d'instructions

- Ainsi le parsing retourne une instance de la classe `Prog`, qui contient une liste d'instructions. La méthode d'évaluation de `Prog` fera donc appel à la méthode d'évaluation de chaque classe `Instruction`. Construire les méthodes `eval()` des classes `Prog` et `InstructionShow`, et appeler l'évaluation de `Prog`, depuis la classe `Analyseur`.
- Rajouter vos exemples au fur et à mesure dans un répertoire `testlogos`

À la fin de cette partie, on sera donc capable de calculer l'expression suivante :

```
MONTRE PRODUIT (2.8 + (RESTE 12 3.5)) PI+SIN(12.9)
```

4.2 InstructionTurtle

Maintenant on va faire bouger la tortue Logo ! On va donc utiliser des primitives de déplacement des positions et du cap. Par exemple, le programme de la Figure **??** tracera un carré dans la fenêtre graphique.

```
AVANCE 40 TOURNED 90
AVANCE 40 TOURNED 90
AVANCE 40 TOURNED 90
     AVANCE 40
```

FIGURE 3 – Exemple exp03.txt

Étapes :

- Construire une classe `Turtle.java` qui peut contenir une position courante (abscisse, ordonnée, cap) de la tortue.

- Dans cup rajouter les règles de grammaire qui permettent de reconnaître les nouvelles instructions `AVANCE`, `TOURNED`, `TOURNEG`, `FIXEX` et `FIXEY`.
- Construire la classe `InstructionTurtle`, en utilisant (comme dans la classe `ASTExpr`) des entiers statiques pour déterminer de quel type est cette instruction :

  `final static int GO=0, TG=1, TD=2, ....`
- La fonction d'évaluation de cette instruction nécessite d'avoir accès aux valeurs précédentes de la position de la tortue (abscisse, ordonnée, cap). Il va donc falloir passer un objet de type `Turtle` aux fonction d'évaluation des `Instructions` (classe abstraite, et autres classes qui en dérivent). Un objet de type `Turtle` avec un cap 0, abscisse 250 et ordonnée 250 sera déclaré au début de l'évaluation d'un `Prog`.
- Dans un premier temps, la fonction d'évaluation de la classe `InstructionTurtle` imprimera sur le terminal les coordonnées des points des segments à tracer. Pour le calcul des nouvelles coordonnées, il y a un peu de trigo à faire (et on se fiche de la précision des calculs). Attention les angles rentrés par l'utilisateur sont en degrés (il va falloir convertir) et la fenêtre graphique est à l'envers (coordonnées (0,0) en haut à gauche, abscisses croissantes vers la droite, ordonnées croissantes vers le bas).

Par exemple, sur l'exemple **??**, le programme imprimera quelque chose du style :

```
draw line from (290.0,250.0) to (289.999,290.0)
draw line from (289.999,290.0) to (249.9999,289.99)
draw line from (289.999,290.0) to (249.9999,289.99)
draw line from (249.999,289.9999) to (250.00,249.999)
```

Une fois les tests effectués, vous utiliserez l'API fournie (dans `api_graphique`) pour stocker les droites d'avancement de la tortue, puis les afficher dans une fenêtre graphique.

Étapes :
- D'abord tester l'utilisation de la bibliothèque fournie (il y a un README).
- Copier les fichiers `Graphique.java` et `GraphiqueInterface.java` dans le répertoire de votre projet
- Modifier la première ligne de ces fichiers pour être cohérent avec votre projet.
- Ajouter un graphique à la classe `Prog` puis aux méthodes d'évaluation des `Instructions` de la même manière que la tortue.
- Tester !

4.3 Instruction DefVar

Attelons-nous à la définition des variables :

```
DONNE "x 7
DONNE "y x+8
MONTRE y
```

En ce qui concerne Flex/Cup, il faudra :
- Ajouter les nouveau *token*.
- Permettre à flex de reconnaître les identifiants (et retourner un token de type `String`).
- Rajouter la règle de grammaire qui permet de reconnaître ces nouvelles instructions, et de construire une nouvelle classe `InstructionDefVar`.

Pour les classes Java :
- Créer une classe `InstructionDefVar` dont le constructeur prend un `ASTExpr` en argument.
- Étendre la classe des expressions de façon à ajouter les variables (attributs, méthodes).

– Pour stocker les valeurs (double) des variables on utilisera une hashtable de type `HashMap<String,Double>` que l'on augmentera au fur et à mesure de la lecture des instructions d'un programme (elle sera donc passée en paramètre des fonctions d'évaluation des instructions). Il faudra donc modifier les classes d'instructions écrites jusqu'à présent pour permettre ce passage par paramètre.
– **Il faut aussi que votre compilateur lance une exception lorsque l'on utilise la valeur d'une variable qui n'existe pas.**
– Modifier la classe `InstructionTurtle` pour modifier dans la hashtable les valeurs des variables `posx`, `posy` et `cap` lors d'un mouvement de la tortue. Initialiser ces variables au début de la méthode `eval` de la classe `Prog`. Faire en sorte que ce soit des mots clefs du langage (flex/cup), donc qu'on ne puisse pas les modifier en faisant `DONNE`.

Traiter cette nouvelle instruction et tester sur les exemples fournis et d'autres exemples que vous jugez pertinents.

Remarque 3 *On fera attention à bien traiter les définitions de la forme :*

```
DONNE "x 12
DONNE "x x+1
```

4.4 InstructionRepeat

Dans cette section on ajoute l'instruction `REPEAT` qui permet de réaliser des instructions répétées plusieurs fois.

```
DONNE "x 1
REPETE 4 [ DONNE "x x+1 MONTRE x ]
```

Figure 4 – Exemple exp07.txt

Il faudra donc :
– Ajouter des mots clefs et la règle de grammaire pour les Instructions de type `REPETE`. Attention, le nombre après `REPETE` est forcément un entier.
– Dans la nouvelle classe `InstructionRepeat` implanter la fonction d'évaluation.
– Tester bien correctement.

Sur l'exemple `??`, le programme imprimera :

```
Instruction : Def of x
2.0
Instruction : Def of x
3.0
Instruction : Def of x
4.0
Instruction : Def of x
5.0
```

4.5 Procédures

Voici un exemple de définition de procédure (sans argument) et son appel :

```
POUR CARRE
REPETE 4 [ AVANCE 40 TOURNED 90 ]
FIN
CARRE
```

FIGURE 5 – Exemple de définition de procédure/appel

On s'inspirera du traitement des variables (déclaration, utilisation) pour effectuer le traitement de ces deux nouvelles instructions.

4.6 Pour aller plus loin

Voici quelques extensions :
- Cadrage automatique (dans `Graphique.java`). On pourra garder dans la classe tortue les abscisses et ordonnées max et min.
- Choix de couleur, et autres choses (voir la syntaxe LOGO) comme le lever de crayon, l'effaçage, etc.

5 Consignes pour le rendu

Vous devez rendre au plus tard **le 17 mai à 20h** (5 points par jour de retard) vos projets dans vos dépôts qui devront avoir la structure suivante :
- un fichier `Readme.txt` contiendra une description rapide de votre logiciel et de sa façon de l'utiliser.
- un Makefile et un `launch`.
- un répertoire `iflogo` contenant vos sources.
- un répertoire `tests` qui comprendra au moins les programmes de dessins fournis par l'énoncé
- éventuellement, un répertoire `old` contenant du code inutile.
- un fichier `nomdubinome.pdf` contiendra votre rapport. Le rapport ne comprendra pas plus de 5 pages, devra être clair et précis et notamment comporter les limitations de votre outil.

Attention ! votre dépôt SVN ne comprendra ni `jflex` ni `javacup` !