La classe et l'objet.

http://www.enseignement.polytechnique.fr/profs/informatique/Luc.Maranget/IF/05/

1 Encore les quatre opérations

On enrichit les expressions arithmétiques de la semaine dernière par une construction de liaison.

$$S \to E\ \mathbf{eof}$$

$$E \to E \texttt{ + } E \qquad E \to E \texttt{ - } E \qquad E \to E * E \qquad E \to E \texttt{ / } E \qquad E \to \mathbf{int} \qquad E \to (E)$$

$$E \to \texttt{let}\ \mathbf{id} \texttt{ = } E\ \texttt{in}\ E \qquad\qquad E \to \mathbf{id}$$

1. Modifier la grammaire de l'instituteur (grammaire non-ambigüe usuelle des expressions arithmétiques, avec les parenthèses) pour tenir compte des nouvelles constructions.

   $$S \to E\ \mathbf{eof}$$

   $$E \to P \qquad E \to E \texttt{ + } P \qquad E \to E \texttt{ - } P$$

   $$P \to F \qquad P \to P * F \qquad P \to P \texttt{ / } F$$

   $$F \to \mathbf{int} \qquad\qquad F \to (E)$$

   On se posera au préalable la question de savoir comment on doit comprendre l'expression `let x = 1 in x + x`.
2. Le document joint donne le code de la classe `Instit` qui est essentiellement un vérificateur de la grammaire de l'instituteur.
   Discuter l'ajout du modificateur de visibilité **private** aux champs et méthodes de la class `Instit`.
   Discuter ensuite le regroupement de trois classes `Token`, `Lexer` et `Instit` en un package de nom `expr`.
3. Modifier le vérificateur pour reconnaître les nouvelles constructions. On supposera que le lexeur (classe `Lexer`) est déjà modifier et fournit les nouveaux lexèmes (classe `Token`) de natures `Token.LET`, `Token.EQ`, `Token.IN` et `Token.ID`.

2 Extension par héritage

Le but de cet exercice est de construite le nouveau vérificateur de l'instituteur par héritage de l'ancien. On commence par se poser quelques question sur l'héritage lui-même. Soit une petite classe :

```
class A {
  void f() { System.out.println("Je suis f de A") ; }
  void g() { f() ; }
}
```

1. **Extension :** Ce code compile-t-il, si oui qu'affiche-t-il ?

   ```
   class B extends A {
    void h() { g() ; }

    public static void main (String [] argv) {
      B b = new B () ;
      b.h() ;
    }
   }
   ```

2. **Redefinition :** Même question.

```
class B extends A {
 void f() { System.out.println("Je␣suis␣f␣de␣B") ; }

 public static void main (String [] argv) {
   B b = new B () ;
   b.g() ;
 }
}
```

3. **Conversions entre classes :** On pensera bien que les objets possèdent une classe exacte qui ne change jamais, mais que le compilateur ne connaît pas forcément cette classe exacte.

```
class B extends A {
 void f() { System.out.println("Je␣suis␣f␣de␣B") ; }
 void h() { g() ; }
 public static void main (String [] argv) {
   A a = new B () ;
   a.g() ; a.h() ;
   B b = a ;
   b.g() ; b.h() ;
 }
}
```

4. Construire le nouveau vérificateur par héritage de l'ancien. Rediscuter rapidement l'usage des modificateurs de visibilité.

3 Les nouveaux habits de notre vielle amie

Re-voici donc notre amie la classe des listes, mais cette fois il s'agit de listes dites d'association.

```
class List {
  private String var ;
  private int val ;
  private List next ;

  List (String var, int val, List next) {
    this.var = var ; this.val = val ; this.next = next ;
  }
}
```

Les listes d'association offrent une unique fonctionnalité : on peut récupérer la valeur (type **int**) associée à une variable (type `String`) par une méthode `assoc`.

1. Pouquoi les champs sont ils tous privés ?
2. En supposant la méthode dynamique écrite, coder la méthode statique.
3. En supposant la méthode statique écrite, coder la méthode dynamique.
4. Finalement, la méthode `assoc` doit elle être statique ou dynamique ? Coder la bonne méthode.
5. Pourquoi la méthode `assoc` termine-t-elle toujours.
6. En supposant que les champs de `List` ne sont plus privés, montrer que la terminaison de `assoc` n'est plus garantie.

4 Environnements

On souhaite maintenant mieux séparer interface et implémentation des environnements. Autrement dit les listes d'association sont une implémentation possible des environnements. L'interface des environnements est définie, justememt par une **interface** de Java (voir la PC 1)

```
interface Env {
  void add(String v, int i) ; // Ajouter la liaison v→i.
  int get(String v)         ; // Renvoie la valeur associée à v.
}
```

1. Un environnement est donc un objet d'une classe qui implémente l'interface `Env`. Comment sera nécessairement représenté l'environnement vide ?
2. Coder une classe des environnements en utilisant la classe des listes d'association.
3. Identifier un problème de coût et dire comment le résoudre.

5 Arbre de syntaxe abstraite

On propose un codage objet des arbres de syntaxe abstraite, ce codage repose sur la définition d'une classe abstraite :

```
abstract class Expr {
  abstract int eval(Env e) ;
}
```

1. Peut-on créer des objets de la classe `Expr` ? À quoi peut elle bien servir ? Quand nous sommes nous déjà servi d'une classe abstraite ?
2. Commençons par supposer l'existence de classes bien nommés, `Mul`, `Div`, etc. et munies de constructeurs raisonnables. Écrire la méthode `parseP` qui cette fois renvoie l'arbre de syntaxe abstraite correspondant à l'entrée analysée.
3. Écrire les classes `Int`, `Id` et `Let` comme des classes héritant de `Expr`. On leur donnera une méthode `toString` dont on précisera si elle est une extension ou une redéfinition. On s'attachera à bien détailller leur méthode `eval` après avoir examiné le cas de `let x = 1 in (let x = x+x in x+x) + x`.
4. Ou pourrait construire directement les classes des opérateurs binaires par héritage de `Expr`, mais on choisit de définir une classe intermédiaire :

```
abstract class Binop extends Expr {
  Expr arg1, arg2 ;

  Binop(Expr arg1, Expr arg2) {
    this.arg1 = arg1 ;
    this.arg2 = arg2 ;
  }

  abstract String getOp() ;

  public String toString() {
    return getOp() + "(" + arg1 + ",␣" + arg2 + ")" ;
  }
}
```

   Ecrire les classes `Add` et `Sub`. Comment justifier la classe intermédiaire `Binop` ?
5. Comparer interfaces et classes abstraites.

Solutions des exercices

Exercice 1

1. On peut introduire un nouveau non-terminal L pour la construction de liaison. Les opération binaires sont « plus prioritaires » que la liaison.

$$S \to L$$

$$L \to \texttt{let}\ \mathbf{id} = L\ \texttt{in}\ L \qquad\qquad L \to E$$

$$E \to P \qquad\qquad E \to E + P \qquad\qquad E \to E - P$$

$$P \to F \qquad\qquad P \to P * F \qquad\qquad P \to P\ /\ F$$

$$F \to \mathbf{int} \qquad\qquad F \to (L)$$

   Et il ne faut pas oublier que la construction `let`... peut apparaître dans les parenthèses.

2. Au minimum, pour que l'on puisse faire quelque chose des objets de la classe `Instit` il faut exposer au monde extérieur le constructeur `Instit` et la méthode `parse`, ils pourront être qualifiés par **public** si on souhaite exposer ces fonctionalités au delà du package courant (nous y reviendront à la question suivante).

   Toutes les autres méthodes et tous les champs devraient être privés (qualifiés par **private**). Cela s'applique indiscutablement aux champs `tok` et à la méthode `step`, afin de contrôler strictement l'usage du tampon de un lexème.

   Le champ `lexer` doit aussi logiquement être privé, mais ici la protection est plus illusoire, car rien n'empêche le monde extérieur de lire un lexème (le `Lexer` provient du monde extérieur).

   Le cas des méthodes `parseF`, `parseP` et `parseE` est également discutable, on peut concevoir un usage risqué de l'analyseur par le monde extérieur (par exemple analyser une liste de facteurs...). Sans autre information, autant cacher ces méthodes (mot-clé **private**).

   En résumé, un usage raisonnable de notre classe est d'exporter le constructeur et la méthode `parse`, et de cacher tout le reste. Mais ce n'est pas si simple.

3. Pour constituer notre petit vérificateur en package, il faut.
   - Préciser par une déclaration **package** `expr`, en tête des fichiers `Token.java`, `Lexer.java` et `Instit.java` que les classes `Token`, `Lexer` et `Instit` appartiennent bien au package `expr`.
   - Ranger les `.class` dans répertoire de nom `expr`, accessible à partir du `CLASSPATH`. Le plus simple est de regrouper ces étapes et la précédente en mettant dès le départ les sources dans un sous-répertoire `expr` du répertoire courant (ou de tout autre répertoire présent dans le `CLASSPATH`).
   - Qualifer les classes `Lexer` (et son constructeur) et `Instit` (et son constructeur, et la méthode `parser`) de publiques, sinon, on ne pourra même pas en parler !.

   Alors, on peut écrire la classe `Main` suivante qui exploite les méthodes publiques du package `expr`.

```
import java.io.* ;
import expr.* ; // « importer » toutes les classes publiques du package expr


class Main {
  public static void main(String [] argv) {
```

```
    Lexer lexer = new Lexer (new StringReader (argv[0])) ;
    Instit i = new Instit (lexer) ;
    i.parse() ;
  }
}
```

La seconde déclaration **import** permet de désigner les classes `Lexer` et `Instit` par leurs noms abrégés (autrement, il faudrait les qualifier complètement `expr.Lexer` et `expr.Instit`).

4. La grammaire indique clairement comment procéder. Il faut modifier légèrement `parseF`, ajouter une nouvelle méthode `parseL` et modifier `parse` pour l'apeller.

```
 void parseF() {
  switch (tok.nature) {
  case Token.OPAR:
    step() ;
    parseL() ;
    if (tok.nature != Token.CPAR) error(")", tok) ;
    step() ;
    return ;
  case Token.ID: case Token.INT:
    step() ;
    return  ;
  default:
    error("entier ou identificateur", tok) ;
    return ; // Not Reached
  }
}

... // parseP et parseE inchangées

void parseL() {
  if (tok.nature == Token.LET) {
    step() ;
    if (tok.nature != Token.ID) error("identificateur", tok) ;
    step() ;
    if (tok.nature != Token.EQ) error("=", tok) ;
    step() ;
    parseL() ;
    if (tok.nature != Token.IN) error("in", tok) ;
    step() ;
    parseL() ;
  } else {
    parseE() ;
  }
}

void parse() {
  parseL() ;
  if (tok.nature != Token.EOF) error("EOF", tok) ;
}
```

Exercice 2

De façon générale, je vous invite quand vous vous posez des questions sur un mécanisme tel que l'héritage, à concevoir des petits exemple tels que ceux proposés.

1. La classe `B` démarre dans la vie avec toutes les méthodes de la classe parente `A`. L'objet `b` possède donc une méthode `g` et une méthode `f`. Le code compile et affichera :

   ```
   Je suis f de de A
   ```

2. La méthode `f` est redéfinie (*override*), tout objet de la classe `B` possède la nouvelle méthode `f`. Le code compile et affichera :

   ```
   Je suis f de de B
   ```

   On note que l'occurence de `f` dans le corps de `g` n'est pas résolue statiquement, c'est à dire en considérant la définition de `f` de la classe `A`. On parle de liaison tardive.

3. Le code ne compile pas. En effet, si la conversion de `b` (de type `B`) vers `A` peut rester implicite, (Java confond héritage et sous-typage, c'est à dire qu'une classe construite par héritage appartient naturellement au type de sa classe parente), la conversion inverse (de `a` de type `A` vers `B`) ne va pas de soit. En effet cette conversion doit être explicitée et elle entraîne une vérification à l'exécution (qui ici n'échoue pas car la variable `a` contient bien un objet de classe `B`). On doit donc écrire :

   ```
     public static void main (String [] argv) {
       A a = new B () ;
       a.g() ; a.h() ;
       B b = (B)a ; // ICI
       b.g() ; b.h() ;
     }
   }
   ```

   Le code ne compile pas encore, car pour le compilateur, un objet de type `A` (c'est-à-dire dont le compilateur pense qu'il est de classe `A`), n'a pas de méthode `h`, on doit donc supprimer l'appel de méthode `a.h()`

   Le code compile enfin, et son exécution affiche :

   ```
   Je suis f de de B
   Je suis f de de B
   Je suis f de de B
   ```

   En effet les conversions de types sont justes des vérifications et ne changent pas la classe de l'objet.

   Résumons nous.

   – Les objet sont d'une classe bien déterminée lors de l'exécution, essentiellement parce qu'ils sont crées par un constructeur bien déterminé.
   – Le compilateur ne connaît pas toujours bien la classe exacte d'un objet à cause des conversions de types. Les conversions qui perdent de l'infornation (de fille à parente par ex.) vont de soi. Les autres conversions sont douteuses et entraînent une vérification des classes exactes lors de l'exécution. Le compilateur exige d'expliciter les conversions douteuses.

   Note : La classe exacte de l'objet contenu dans `a` est bien `B`, mais le compilateur ne veut pas le savoir. Pour comprendre que le compilateur n'est pas si crétin. Il faut noter que parfois le compilateur ne peut pas connaître la classe exacte d'un objet :

   ```
   static void appelerH(A a) {
     a.h() ;
   }
   ```

   Il faut écrire :

```
static void appelerH(A a) {
  ((B)a).h() ;
}
```

Et se préparer à voir surgir l'exception idoine `ClassCastException`, que bizarrement on est pas obligé de déclarer.

4. On défnit la nouvelle classe `HInstit` par héritage :

```
import java.io.* ;

class HInstit extends Instit {

  HInstit(Lexer lexer) {
    super(lexer) ;
  }

  void parseF() { ... }

  void parseL() { ... }

  void parse() { ... }

  public static void main(String [] argv) {
    Lexer lexer = new Lexer (new StringReader (argv[0])) ;
    HInstit i = new HInstit (lexer) ;
    i.parse() ;
  }
}
```

Le code des méthodes a déjà été donné. Il est remarquable que, par la le mécanisme de liaison tardive, les méthodes `parseE` et `parseP` appellent nouvelles définitions des méthodes redéfines, bien que ces premières soient définies dans la classe parente `Instit`.

On notera l'appel au constructeur **super**`(lexer)` dans le constructeur de la classe héritante.

Enfin, il n'est plus possible de qualifier par exemple `tok` ou `step` par **private**, car alors la classe héritante n'y aurait plus accès. On notera que l'absence de qualificateur signifie « visible de toutes les classes du même package », que **protected** signifie « visible de toutes les classes du même package et des classes héritantes », et que **public** signifie « visible de partout ».

Exercice 3

1. Compte tenu de la spécification (une seule fonctionalité `assoc`), les autres classes n'ont pas besoin de connaître la structure exacte des objets de la classe `List`. Ainsi nous contrôlons toutes les interactions des autres classes avec la classe `List`.
2. La statique avec la dynamique.

```
static int assoc(List l, String v) {
  if (l == null) throw new Error (v + "inconnue") ;
  return l.assoc(v) ;
}
```

3. La dynamique avec la statique.

```
int assoc(String v) {
  return assoc(this,v) ;
}
```

Cela semble idiot mais nous dit qu'une méthode dynamique est essentiellement une méthode statique avec un premier argument caché **this**.

4. Puisque la liste vide est naturellement représentée par **null**, il faut choisir une méthode statique.

```
static int assoc(List l, String v) {
  for ( ; l != null ; l = l.next) {
    if (l.var.equals(v)) return l.val ;
  }
  throw new Error (v + "␣inconnue") ;
}
```

5. La méthode `assoc` termine toujours si la liste ne reboucle pas sur elle-même. Cette propriété est garantie sur les listes sont toutes construites avec le constructeur donné.
6. Si le champ `next` n'est pas privé, alors on consruit facilement une liste bouclée ainsi :

```
List l = new List("x", 1, null) ;
l.next = l ;
```

Exercice 4

1. Puisque `assoc` est dynamique l'environnement vide ne peut pas être représenté par **null**.
2. On a recourt à une technique objet bien classique et que j'aime bien : l'encapsulage, les objets de la classe `EnvList` possèdent un champ qui est une liste.

```
class EnvList implements Env {
  private List me ;

  EnvList () { me = null ; } // Constructeur des  environnements vides, amical.

  private EnvList (List l) { // Constructeur privé.
    me = l ;
  }

/* Les méthodes des interfaces sont nécessairement publiques,
   c'est la vie ! */
  public Env add (String x, int v) {
    return new EnvList(new List (x, v, me)) ;
  }

  public int get(String x) {
    return List.assoc(me, x) ;
  }
}
```

On remarquera que `add` renvoie un `Env` (et non pas un `EnvList`) et la conversion de type est ici sans douleur. En effet, il s'agit ici d'une perte d'information qui ne peut pas échouer.

3. Le problème de coût est algorithmique, l'accès à une liaison coûte au pire le nombre de liaisons. On peut résoudre ce problème en utilisant une structure de donnée plus performante comme par exemple les arbres binaires équilibrés (accès en log). Il est important de fournir alors une classe des environnements qui implémente encore l'interface `Env`.

Exercice 5

1. Non, on ne peut pas créer d'objets de la classe `Expr`. En effet le code de la méthode `eval` n'est pas défini. (La réponse, il n'y a pas de constructeur n'est pas vraiment bonne, cf. la question 4, et penser au constructeur par défaut `Expr ()`.)

   En revanche on poura créer des classes par héritage de `Expr`, et de fait on ne pourra construire des objets de ces classes filles que si la méthode `eval` est définie. C'est tout le sens de la méthode déclarée abstraite.

   Enfin, la classe `Reader` que nous connaissons déjà est en fait abstraite. Une de ses classes filles est par exemple `StringReader`.

2. Il faut se lancer bêtement.

```
private Expr parseP() {
  Expr r = parseF() ;
  while (tok.nature == Token.MUL || tok.nature == Token.DIV) {
    Token op = tok ;
    step() ;
    Expr f = parseF() ;
    if (op.nature == Token.MUL)
      r = new Mul (r, f) ;
    else
      r = new Div (r, f) ;
  }
  return r ;
}
```

   Pour les curieux, la version HTML contient un pointeur vers un parseur complet.

3. Une fois que le truc est compris, ça roule tout seul.

```
class Int extends Expr {
  private int me ;

  Int (int i) { me = i ; }

  int eval(Env e) { return me ; }

  public String toString() { return Integer.toString(me) ; }

}

class Id extends Expr {
  private String me ;

  Id (String s) { me = s ; }
  int eval(Env e) {
    return e.get(me) ;
  }

  public String toString() { return me ; }
}
```

   La méthode `toString` est la redéfintion de la méthode homonyme de la classe `Object` dont toutes les classes héritent implicitement. Si on ne la redéfinissait pas, l'affichage d'une expression La classe `Let` est à peine plus complexe.

```
class Let extends Expr {
```

```
  private String x ;
  private Expr ex, e ;

  Let (String x, Expr ex, Expr e) {
    this.x = x ; this.ex = ex ; this.e = e ;
  }

  int eval(Env env) {
    int vx = ex.eval(env) ;
    return e.eval(env.add(x,vx)) ;
  }

  public String toString() {
    return "LET (" + x + ", " + ex + ", " + e + ")";
  }
}
```

Cette classe nous montre bien la règle d'évaluation de la liaison, il s'agit justement ce qu'il convient d'appeler la liaison statique, ou parfois liaison lexicale. Sa méthode `toString` montre l'intérêt de redéfinir cette méthode dans toutes les classes filles de `Expr`.

4. L'intérêt de la classe intermédiaire est de factoriser un peu de code commun à tous les opérateurs binaires (deux champs, un contructeur, et une méthode `toString` essentiellement identiques.) On remarque que la méthode (concrète) `toString` de `Binop` appelle la méthode abstraite `getOp` sans complexe aucun.

```
class Add extends Binop {

  Add (Expr arg1, Expr arg2) {
    super(arg1,arg2) ;
  }

  String getOp() { return "+" ; }

  int eval(Env e) {
    return arg1.eval(e) + arg2.eval(e) ;
  }
}

class Sub extends Binop {
  Sub (Expr arg1, Expr arg2) {
    super(arg1,arg2) ;
  }

  String getOp() { return "-" ; }

  int eval(Env e) {
    return arg1.eval(e) - arg2.eval(e) ;
  }
}
```

On note l'appel au constructeur **super**(arg1,arg2).

Pour les curieux, la version HTML contient un pointeur vers une classe complète ds expressions.

5. Les classe abstraites sont plus puissantes que les interface en un sens, car elles ne se limitent pas, comme le montre bien `Binop` à la spécification de signatures de méthodes dynamiques

publiques (cf. supra). Mais l'héritage simple de Java (on hérite d'une seule classe) est parfois limitant, il faut recourir aux interfaces si on souhaite réaliser deux fonctionalités distinctes spécifiées indépendamment.