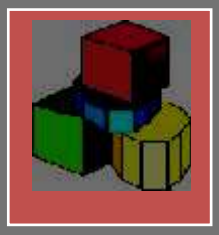


LISP

La cage aux parenthèses, une folle aventure

Réalisation d'une machine virtuelle

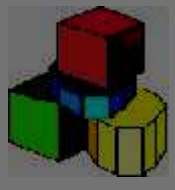


Table des matières

1 Objectif

Le but de ce travail est de réaliser une modélisation objet en utilisant le langage Java fournissant l'environnement **Lisp** décrit dans le document joint (Lisp.pdf). Vous prendrez soin de concevoir un programme structuré mettant en évidence les concepts majeurs, vous aurez à l'esprit lors de son élaboration que votre programme doit pouvoir évoluer dans le temps ; Admettre de nouvelles spécifications venant naturellement compléter votre réalisation, accepter des optimisations permettant une plus grande performance de votre environnement, etc.

Ce travail permet aussi de revoir les principes de l'approche fonctionnelle et des mécanismes d'évaluation qui sont sous-jacents à toute machine d'exécution.

2 Présentation générale de l'environnement Lisp

Le langage Lisp est un des premiers langages interprétés, il est aussi l'un des premiers langages fonctionnels. Ecrire un interprète de ce langage revient à réaliser une Machine Virtuelle Lisp (**L**isp **V**irtual **M**achine) capable d'évaluer des expressions de ce langage. C'est un langage non typé, c.à.d. que les variables ne possèdent pas de type, et qu'il s'ensuit qu'aucune vérification statique n'est faîte avant leurs utilisations. L'environnement Lisp est interactif et permet à l'utilisateur d'obtenir le résultat de l'évaluation d'une expression fonctionnelle qu'il fournit en entrée de l'interprète.

Plusieurs modes d'évaluation (catégories ou types de fonctions) seront considérés. On distinguera pour l'instant 2x2 types de fonctions dans notre système. En plus des EXPR et FEXPR, on typera de manières différentes les fonctions qui sont des primitives du système, c.à.d. des fonctions dont le code n'est pas écrit en Lisp mais pour notre part directement en Java. On parlera des types SUBR et FSUBR pour les primitives.

Les fonctions suivantes devront être présentes dans votre système, pour chaque fonction on indique son type et entre parenthèses son nombre d'arguments, la première colonne correspond aux fonctions déjà décrite dans le document joint et sont simplement listées, la seconde correspond à des fonctions supplémentaires dont on vous donne une spécification synthétique de leurs comportements à l'exécution :

SUBR (1):**car**	SUBR (1):**explode**	construit la liste des caractères de l'atome (explode '|a()b|) -> (a ( ) b)
SUBR (1):**cdr**	SUBR (1):**implode**	construit un atome à partir de la liste de caractères (implode '(a b c)) -> abc
SUBR (2):**cons**	SUBR (n):**print**	imprime les valeurs des paramètres, sans argument effectue un passage à la ligne, renvoie la valeur du dernier paramètre. (print 'a 1 '(b c) (eq 'a 'a)) imprime a1(b c)t et renvoie t
SUBR (2):**eq**	SUBR (1):**eprogn**	évalue séquentiellement une suite d'appels de fonction et rend la valeur du dernier appel (eprogn '((print 'a) (print 'b) (print 'c))) imprime abc et renvoie c
SUBR (1):**atom**	SUBR (2):**set**	modifie la valeur associée à une variable, renvoie la valeur affectée. Crée la variable si elle n'existe pas.

		Après (set 'v '(1 2 3)) v s'évalue à (1 2 3)
SUBR (2):**apply**	SUBR (1):**load**	évalue le programme contenu dans le fichier, renvoie ce que vous voulez ! (load '/boot.ll)
SUBR(1):**eval**	SUBR (0):**quit**	termine l'exécution de l'interprète
FSUBR (n):**de**	SUBR (1):**typefn**	fournit le type de la fonction par le symbole : expr, fexpr, subr ou fsubr. (typefn car) -> subr (typefn 'car) -> erreur ce n'est pas une fonction (typefn '(lambda (a) a)) -> expr (typefn (flambda (a) a)) -> fexpr
FSUBR (n):**df**	SUBR(0):**toplevel**	correspond à la boucle principale de l'interprète Lisp. On vous propose un code pour cette fonction dans la suite de ce document.
FSUBR (n):**cond**	SUBR (0):**scope**	imprime l'état du contexte sous forme d'une suite de ligne comportant le nom de la variable et la valeur associée. Vous pouvez utiliser la classe IO qui fournit le moyen d'imprimer simplement dans une fenêtre textuelle indépendante.

3 Identifier les concepts et les constituants majeurs

La modélisation objet se résume à élaborer des concepts en définissant des classes, des attributs, des méthodes ou des objets. L'approche objet favorise l'identification des concepts propres à l'application ou domaine de l'application que l'on souhaite réaliser, elle incite à produire des solutions proches du monde réel que l'on veut modéliser. A partir du texte fourni (document Lisp), surligner les mots qui vous paraissent importants dans le texte et affectez-les soit à une classe, un objet, un attribut de classe ou une méthode. Construisez une hiérarchie de classes à partir de cette première analyse du domaine. Fournissez la classification des termes retenus.

La structure hiérarchique des classes doit refléter les concepts du domaine, sa "géométrie" est nécessairement un élément d'appréciation de sa bonne élaboration. Vous serez attentif à définir les "classes interface" java nécessaires pour éviter les problèmes d'héritage multiple inhérents à ce langage.

4 Spécifier les classes principales

Vous compléterez ce travail en définissant plus précisément la spécification de chaque classe élaborée : constructeur et méthodes de son interface.

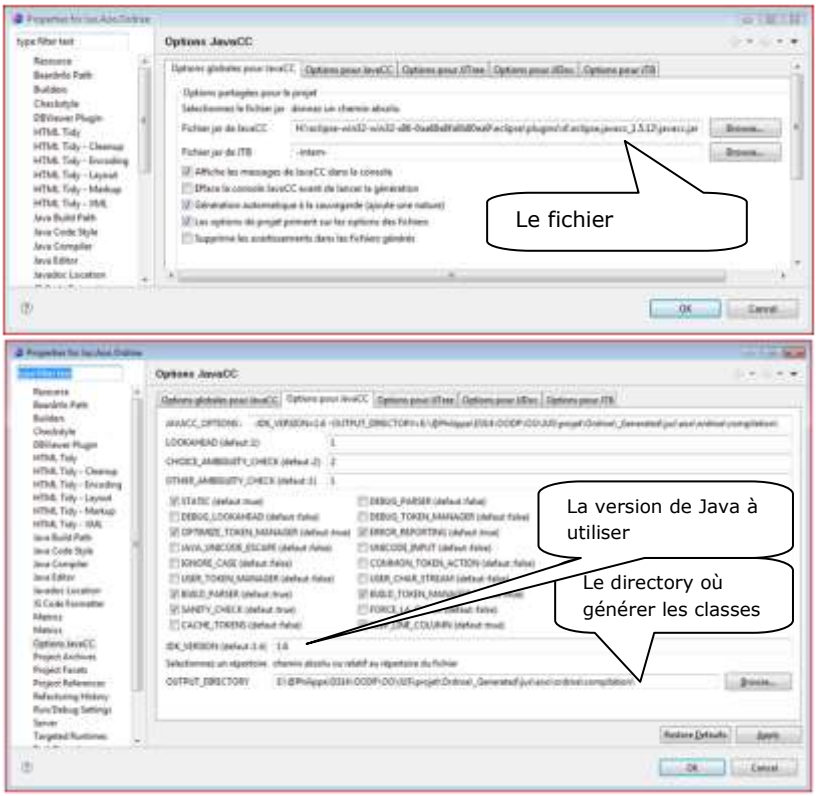
Chaque classe sera pertinemment commentée. Vous préciserez les choix, de façon circonstanciée, qui ont présidé aux solutions que vous avez retenues. Vous utiliserez à cet effet tous les moyens de spécifications qui sont à votre disposition.

5 L'acquisition de S-expr

Ceci se fera via une interaction avec l'utilisateur ou par lecture dans un fichier. Pour cela nous allons utiliser un générateur d'analyseur syntaxique qui s'intègre aisément à Java.

5.1 Configuration de JavaCC

Cette description se trouve dans un fichier qui peut se nommer « Reader.jj ». Si vous avez installé le plugin JavaCC (http://sourceforge.net/projects/eclipse-javacc) dans votre environnement Eclipse, la mise à jour et la génération des classes nécessaires se fera automatiquement du moment où vous aurez fixé les options dans les propriétés de votre projet. Celles-ci permettent de déterminer le fichier jar à utiliser pour l'analyse de la grammaire, ainsi que le lieu où vous souhaitez engendrer les classes de l'analyseur. On vous conseille d'utiliser un second directory source dans votre projet pour isoler correctement les différentes parties de votre application.



5.2 Prémisse de la grammaire

Le Reader permet d'acquérir les S-EXPR sur l'entrée sélectionnée. On vous fournit les prémisses du programme d'analyse syntaxique sous forme de la grammaire du langage Lisp décrit en JavaCC. Vous pourrez constater, à la lecture de cette grammaire, que ce langage admet des commentaires sous 2 formes comme dans le langage Java (!). Nous avons ajouté à la syntaxe fournie initialement une description supplémentaire pour les symboles permettant d'inclure presque tous les caractères dans un symbole sous la forme d'une suite de caractères compris entre barres verticales. Voici un exemple de texte source correct :

```
// les_predicats_de_base
(de false () ())
(de true () t)
/* définition de fonction mise en commentaire
// les fonctions I/O
(de println l ((lambda (a) (print) a) (apply print l))
*/
(df quote (quote_a) quote_a)
(de list l l)
(car '(|ce symbole comporte des caractères particuliers () ' non admis autrement|  ici))
```

Un commentaire sur une seule ligne

Un commentaire sur plusieurs lignes

Un symbole vraiment complexe

La classe Reader comportera principalement 3 méthodes permettant l'acquisition de S-EXPR : read(), read(String), importe() dont vous trouverez les spécifications ci-dessous . D'autres méthodes pourront être définies si vous en avez le besoin.

```
options {
     DEBUG_PARSER=false;
     STATIC=false;
}

PARSER_BEGIN(Reader)
package <le package de déclaration> ;

<Section d'import>

public class Reader {
  /** le support de lecture */
  protected static java.io.Reader in = new BufferedReader(new InputStreamReader(System.in));
  /** lecture d'une S-EXPR au terminal
   * @return Sexpr : la Sexpr construite.
   * @throws LispException une erreur de syntaxe
   */
  public static Sexpr read() throws LispException{ … }
  /** lecture d'une S-EXPR à partir de la chaîne
   * @param s : la chaîne
   * @return Sexpr : la Sexpr construite.
   * @throws LispException une  erreur de syntaxe
   */
  public static Sexpr read(String s) throws LispException{ … }
  /** évaluation de la séquence S-EXPRs à partir du fichier s
   * @param s : le nom du fichier
   * @return Sexpr : symbole du nom du fichier.
   * @throws LispException une erreur de lecture
   */
  public static Sexpr importe(String s) throws LispException{ … }
}

PARSER_END(Reader)
// les caractères ignorés
SKIP :
{
  " " | "\t" | "\r"| "\n"     // attention élimine la notion de fin de ligne
| "\u0000"| "\u0001"  | "\u0002"| "\u0003"| "\u0004"  | "\u0005"| "\u0006"| "\u0007"
| "\u0008"| "\u000B"  | "\u000C"| "\u000E"| "\u000F"  | "\u0010"| "\u0011"| "\u0012"
| "\u0013"| "\u0014"  | "\u0015"| "\u0016"| "\u0017"  | "\u0018"| "\u0019"| "\u001A"
| "\u001B"| "\u001C"  | "\u001D"| "\u001E"| "\u001F"  | "\u007F"
}
MORE :
{
  "//" : IN_SL_COMMENT
| "/*" : IN_ML_COMMENT
}
<IN_SL_COMMENT> SPECIAL_TOKEN :{<SL_COMMENT: "\n" | "\r" | "\r\n"> : DEFAULT}
<IN_ML_COMMENT> SPECIAL_TOKEN :{<ML_COMMENT: "*/" > : DEFAULT}
<IN_SL_COMMENT,IN_ML_COMMENT> SKIP :{< ~[] >}// les lexèmes du langage
TOKEN :{
Définir les lexèmes du langage
}
// les règles de grammaire de ce langage
Type1 SEXPRESSIONS() :
{Type2 s1;}
{
  (s1=SEXPR(){…})* <EOF>
  {
       …
  }
  <EOF>
}
```

Le type de ce que restitue l'analyse de SEXPRESSIONS

Le type de l'information devant être restituée par la méthode « SEXPR »

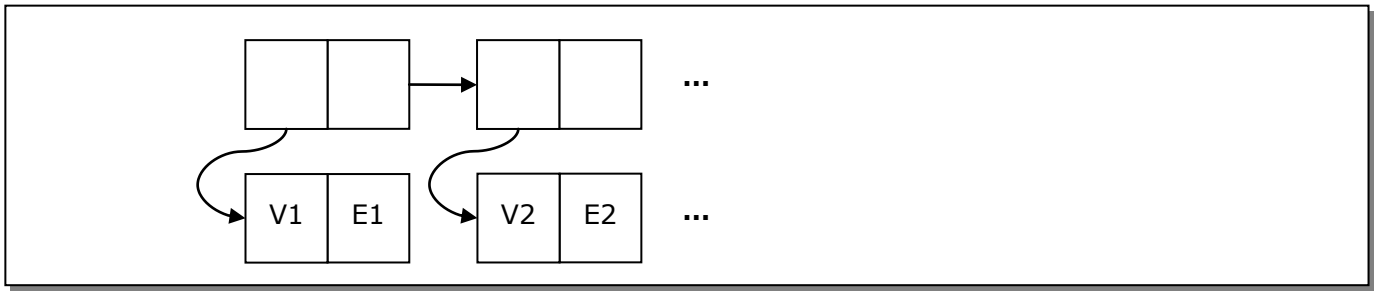
Les actions à exécuter pour vérifier et engendrer ce qui est nécessaire à ce point de l'analyse. Par simplification on conseille de regrouper toutes les actions sémantiques en ce lieu (la complexité du langage étant réduite).

Les autres règles qui ne comportent pas d'actions

6 Propositions techniques

6.1 Représentation du contexte

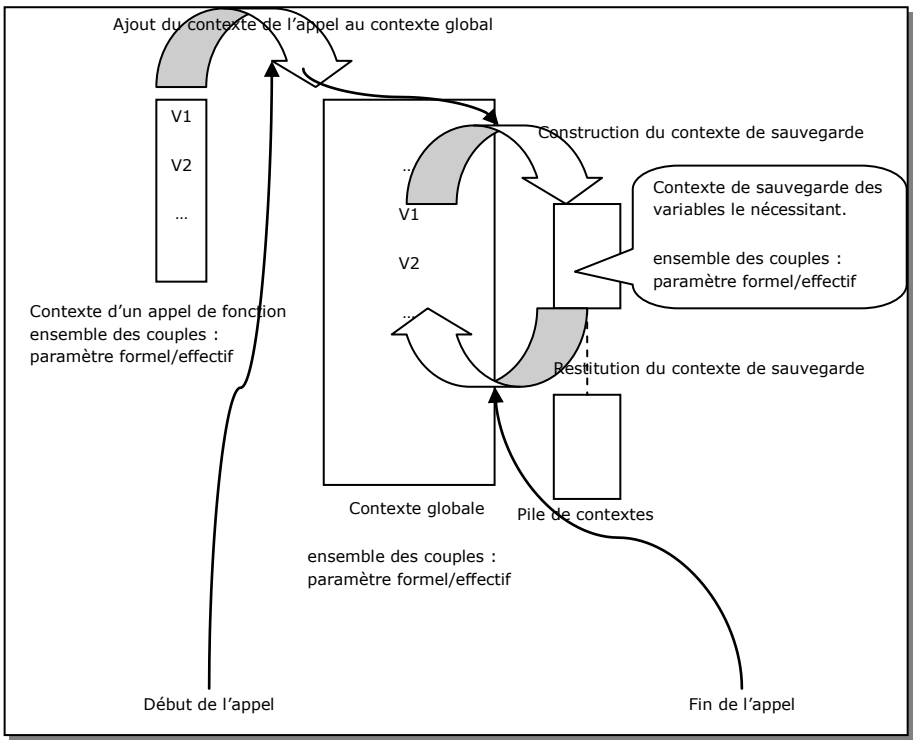
On rappelle que la portée est dynamique en Lisp, c.à.d. que les variables accessibles à un instant donné doivent être répertoriées. Le **contexte** maintient l'ensemble des variables accessibles à un instant donné. Deux solutions sont envisageables dans un premier temps. Chaque élément du contexte est une paire qui associe au nom de la variable la valeur de celle-ci. Le contexte peut être représenté par une liste qui contient l'ensemble des associations. Cette liste est passée en paramètre des méthodes qui ont besoin d'accéder au contexte (c.à.d. toutes). L'évolution de cette liste peut être assurée par le passage de paramètre du système hôte.



Cette solution a le défaut de rendre la recherche un peu coûteuse ($\mathcal{O}$n) et oblige à passer un paramètre de façon très systématique.

La seconde solution permet de palier à ces 2 défauts. On rappelle que les seules variables dans un tel système sont les paramètres des fonctions.

On définit une table dont les clés sont les noms des variables, ce qui permet un accès quasi-direct par ce biais. La table est globale au système, il faut donc assurer son évolution par ajout et retrait des associations. Chaque fois que l'on exécute une fonction on ajoute dans la table les associations paramètres formels/effectifs. Si une variable d'un même nom était déjà présente dans la table, il faut mémoriser dans une structure secondaire cette association pour la restituer après exécution de la fonction. On peut utiliser une pile de contexte qui contient les associations variables/valeurs à restituer en fin d'exécution de la fonction.



6.2 Démarrage du système

Au démarrage du système un fichier de boot sera automatiquement chargé, permettant d'effectuer systématiquement des évaluations initiales qui sont essentiellement de définition de fonctions dont l'usage est courant. Ci-contre un exemple des fonctions de base que pourrait contenir un tel fichier. On y retrouve la définition des fonctions quote, lambda flambda, de nombreux prédicats très utiles comme false, true, not, atomp, consp, null, la fonction mcons qui est la forme naire de cons se terminant par une paire pointée, une fonction let d'augmentation de contexte et des fonctions de confort d'impression et autre.

```
// les prédicats de base
(de false () ())
(de true () t)
(de null (l) (eq l nil))
(de not (a) (cond (a  nil) (t t)))
(de atomp (a) (atom a))
(de consp (a) (cond ((atom a) nil) (t a)))
// les fonctions standard
(df quote (a) a)
(de list l l)
(de mcons l
   (cond    ((null (cdr l)) (car  l))
            (t (cons (car l) (apply mcons (cdr l))))))
(df lambda args (cons 'lambda args))
(df flambda args (cons 'flambda args))
// les fonctions de contrôle
(df  progn c (apply eprogn c))
// les fonctions du contexte
(df let (f e . c)
          (eval (cons (cons 'lambda (cons f c)) e)))
// les fonctions I/O
(de println l ((lambda (a) (print) a) (apply print l)))
```

6.3 Les Entrées/sorties

Pour effectuer les entrées/sorties vous disposez dans le package **jus.util** de la classe IO fournissant des méthodes dédiées à cet usage. Afin d'être assez indépendant du moyen d'interaction utilisé, vous pouvez définir une classe intermédiaire Console qui fournit les abstractions nécessaires. Voici, ci-contre, une proposition pour une telle classe.

```
/**@author morat */
public class Console {
   public static void prompt(String s){ System.out.print(s);}
   public static void print(String s){ IO.print(s);}
   public static void println(String s){IO.println(s);}
   public static void println(){IO.println();}
   public static void resetOut() {IO.resetOut();}
   public static void setOut(){IO.setOut();}
   public static void setOut(String s){IO.setOut(s);}
   public static void debug(String s) {
      if(System.getProperty("DEBUG")!=null) System.err.println(s);
   }
   public static void printStack(Throwable e) {
      if(System.getProperty("DEBUG")!=null) e.printStackTrace();
   }
}
```

6.4 Les classes LVM & Lisp

On vous fournit un canevas pour la définition la Lisp Virtual Machine (LVM). Vous pouvez vous en inspirer. Il est certainement préférable de dissocier cette classe de la main-classe qui permettra le lancement de l'application et que l'on pourrait nommer Lisp ; Son rôle se cantonnerait, dans un premier temps, à lancer la machine virtuelle lisp. Par la suite on pourra étoffer celle-ci.

```
/**
 * définition de la machine Lisp.
 * @author   P.Morat ou http://imag.fr/Philippe.Morat
 */
public class LVM implements Serializable {
   /**  le démarrage de la machine Lisp */
   public LVM() {
      try {
         Console.println(Resources.getBundle("LispEnter"));
         XmlHandler.read("/Lisp.xml");
      } catch (LispException e) {
         Console.println("toplevel:");
         Console.printStack(e);
      }
      <appel du toplevel >
   }
   /**
    * arrêt de la machine lisp
    */
   public static void quit() {
      Console.println(Resources.getBundle("LispExit"));
      System.exit(0);
   }
   …
}
```

6.5 La primitive TopLevel

Voici ce que pourrait être le code de la primitive toplevel assurant la boucle d'interaction avec l'utilisateur. La boucle s'arrête lors de l'appel de la primitive « quit » terminant l'exécution du programme principal par l'instruction Java "exit". Une autre solution consiste à contrôler l'itération par la valeur d'un booléen dont l'état est modifié par cette fonction « quit ».

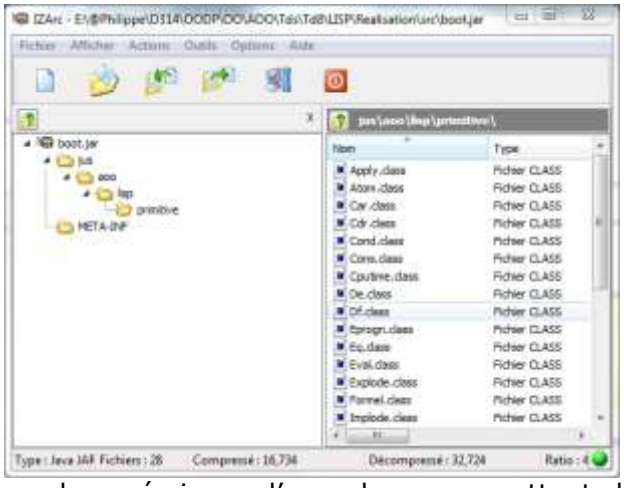
```
while (true) {
    Console.print(">");
    try { Console.afficherln(Reader.read().eval());
    } catch (LispException e) { Console.afficherln(e.getMessage());
    } catch (Exception e) { Console.afficherln("unexpectedException");  e.printStackTrace();
    } finally {}
 }
```

6.6 Gestion des ressources, internationalisation, …

Pour stocker et internationaliser les messages, vous utiliserez ce que fournit Java avec la classe ResourceBundle.

Pour permettre de configurer facilement l'environnement Lisp à utiliser, vous pourrez mettre en place une procédure d'initialisation décrite dans un fichier xml contenant les opérations à réaliser. Ci-contre, un exemple de ce que pourrait être ce fichier de description. Comme nous l'avons déjà dit le modèle minimal Lisp est réduit à quelques primitives, toutes les autres pouvant être élaborées à partir de celle-ci. Cependant pour que ce soit réellement utilisable il est préférable d'avoir un ensemble de fonctions de base déjà disponibles dans l'environnement. Une partie de ces fonctions de bases peuvent être sous forme primitive, les autres sont des expressions Lisp standard. La fonction SUBR (1): **loadlibrary** prend en argument un fichier jar qui contient une classe par primitive. Pour chaque classe (par exemple la classe Car) ont associe à la variable car la primitive jus.aoo.lisp.primitive.Car. L'utilisateur peut ainsi rendre aisée l'accès à une librairie de fonctions écrites en langage natif.

```
<!--
    Document   : Lisp.xml
    Author     : morat
    Description: L'initialisation du contexte de LVM.
-->
<root>
    <!-- Les types prédéfinis -->
      <type symbole="lambda" type="expr" class="…"/>
    <!-- Les variables prédéfinies -->
       <variable name="nil" value="()"/>
       <variable name="t" value=""/>
    <!-- la primitive de chargement de librairie -->
       <primitive name="loadlibrary" class="…"/>
    <!-- les libraries primitives préchargées -->
       <loadlibrary name="/boot.jar"/>
    <!-- les primitives lisp préchargées -->
       <load name="/boot.ll"/>
</root>
```



Vous pouvez utilisez la librairie SAX pour acquérir les éléments de ce fichier de configuration. Ci-dessous les prémisses d'une classe permettant de réaliser cette acquisition.

```
/**
 * @author morat
 *
 */
class XmlHandler extends DefaultHandler {
  /** la méthode traitant de l'évènement début d'élément
   * @param tag le nom de l'élément
   * @param attrs la liste des attributs de cet élément
   * @throws SAXException
   */
  public void startElement(String namespaceURI, String localName, String qName, Attributes atts)
throws SAXException{
    if(localName=="type") traiteType(atts);
    if(localName=="variable") traiteVariable(atts);
    if(localName=="primitive") traitePrimitive(atts);
    if(localName=="load") traiteLoad(atts);
    if(localName=="loadlibrary") traiteLoadLibrary(atts);
  }
  /** Traite un chargement de fichier
   * @param attrs les attributs
   * @throws SAXException
   */
  protected void traiteLoad(Attributes attrs)  throws SAXException {
    try {
      String name = attrs.getValue("name");
      Reader.importe(name);
    } catch(Exception e) {
      throw new SAXException("traiteLoad", e);
    }
  }
  <Autres méthodes de traitements>
  //partie ErrorHandler
  // treat validation errors as fatal
  /** traitement de erreur*/
  public void error(SAXParseException e)throws SAXParseException {throw e;}
  // dump warnings too
  /** traitement de warning*/
  public void warning(SAXParseException err) throws SAXParseException {
    Console.println("** Warning"+", line "+err.getLineNumber()+", uri "+err.getSystemId());
    Console.println("   " + err.getMessage());
  }
  /**
   * Acquisition des éléments de configuration de la session lisp
   * @param file le fichier de configuration
   */
  public static void read(String file) {
    //lecture du fichier de description du contexte
    try{
      XmlHandler handler = new XmlHandler();
      XMLReader parser = XMLReaderFactory.CreateXMLReader();
      parser.setContentHandler(handler); parser.setErrorHandler(handler);
      parser.parse(new InputSource(Reader.class.getResourceAsStream(file)));
    }catch (SAXParseException err) {
      System.err.println("** Error"+", line "+err.getLineNumber()+", uri "+err.getSystemId());
      System.err.println("   " + err.getMessage());
    }catch (SAXException e) {
      Console.printStack((e.getException()!=null)?e.getException():e);
    }catch (Throwable t) {
      Console.printStack(t);
    }
  }
}
```