Université Paul Sabatier
Toulouse III
IUT « A »
Département Informatique

Rapport de stage

—

Annexes

Compalgo 2.0
Compilation et contrôle d'exécution interactif

Damien Leroux
Année Spéciale 2006-2007

Sommaire

Formulation des besoins

Date : 04/07/2007

Objet : formulation des besoins et pistes stratégiques

Auteur : Damien Leroux

Version : 2

1. Préambule

L'équipe pédagogique de l'IUT « A » Paul Sabatier de Toulouse a développé un langage algorithmique, et le compilateur associé, aux fins de l'initiation à la programmation impérative pendant le DUT Informatique.

Ce langage dans sa version actuelle souffre malheureusement de quelques problèmes et limitations, notamment d'un défaut d'analyse des expressions mathématiques et de l'impossibilité de contrôler l'exécution des programmes compilés (pas à pas, points d'arrêt, affichage de l'état des variables…).

2. Formulation des besoins

L'équipe pédagogique de l'IUT souhaite disposer d'une version améliorée de ce langage. Il s'agira de corriger la grammaire existante et d'apporter de nouvelles fonctionnalités, tout en répondant à certaines contraintes. Il est également demandé de prévenir les évolutions futures en fournissant une application à la maintenance aisée.

Correction de la grammaire

Il existe quelques bugs connus au niveau de l'analyse syntaxique du langage, notamment en ce qui concerne la précédence des opérateurs arithmétiques. Aussi, il n'est actuellement pas vérifié que la grammaire, actuellement spécifiée sous forme EBNF, est bien de la classe LL(*k*), en particulier le *k* éventuel est inconnu. Il conviendra dans un premier temps de réécrire la grammaire actuelle en prenant soin de réparer ces problèmes et de rendre la grammaire LL(1) au format BNF.

Nouvelles fonctionnalités

Objectifs primaires

En plus de réécrire la grammaire, les nouvelles fonctionnalités concernent :
la séparation possible entre en-têtes (spécification) et corps de sous-programmes (implantation),
l'intégration de sous-programmes (inline), pour lesquels l'implantation est directement réalisée en code Java. Ceci nécessite que la machine virtuelle accepte des implantations de sous-programmes "à chaud".

Objectifs secondaires

Il faut augmenter l'interface utilisateur actuelle en fournissant un cadre de débogage des programmes des étudiants. Pour ce faire, il sera nécessaire de définir les mécanismes à l'œuvre dans la machine virtuelle du langage et dans le débogueur, ainsi que l'interface logicielle à implanter.

Enfin, Il sera souhaitable de pouvoir réaliser les diagrammes syntaxiques de la grammaire finale.

Contraintes

Pour des raisons de portabilité, Compalgo, son éditeur, et son débogueur utilisent la plateforme Java. Cela implique de n'utiliser que des outils Java, de préférence sans librairie native. Il est aussi nécessaire que la compilation s'effectue rapidement, afin de ne pas perturber la dynamique des séances de travaux pratiques.

L'aspect pédagogique est prépondérant dans ce projet.

Ainsi, les messages d'erreurs émis par le compilateur se devront d'être univoques et explicites, guidant l'étudiant dans le processus de correction du code. De même le cadre de débogage s'il en est aura pour unique but d'améliorer la compréhension par les étudiants des processus et traitements mis en œuvre sans les plonger dans les arcanes de la machine virtuelle. Il devra fournir une interface et des représentations de haut niveau, adaptées au contexte de l'apprentissage de la programmation impérative avec Compalgo.

3. Annexe : stratégies envisageables

Il existe de nombreux outils (cf. test comparatif) pour développer des compilateurs en Java, chacun avec ses caractéristiques différentes et prenant en charge tout ou partie de la chaîne de compilation. Le développement de Compalgo passera nécessairement par un de ces outils (à ce sujet, voir le test comparatif), mais dans tous les cas plusieurs stratégies sont possibles concernant la méthode de génération du code final et les mécanismes de gestion de l'exécution dudit code.

Génération du code compilé

Il est possible de générer le code directement lors de l'analyse syntaxique du langage.

Il est aussi possible de générer un AST (*Abstract Syntax Tree*, en français Arbre de Syntaxe Abstraite) représentant une forme intermédiaire entre le texte source et le code compilé, et ainsi séparer totalement l'analyse lexicale et syntaxique du texte source de la génération du code.

Dans tous les cas, la génération de la sortie du *parseur* se fait par l'adjonction d'actions dites sémantiques à la grammaire du langage, ainsi ce choix stratégique n'aura pas d'autre effet de bord que la séparation ou non de l'analyse syntaxique et de la génération du code final.

Moyennant une spécification formelle précise, il est probable que le choix de l'abstraction garantira la plus grande facilité de maintenance et l'évolutivité du produit.

Spécification de l'hôte de l'exécution des programmes

Il est possible de spécifier une machine virtuelle ad hoc :

Types de base (entier, réel, caractère, chaîne, tableau, enregistrement),

Opérations sur les types de base,
Exécution séquentielle,
Pile système,
Gestion du débogage :
Tables de symboles consultables,
Localisation dans le texte source,
Exécution pas à pas, et points d'arrêt.

Il est possible de profiter de la machine virtuelle de Java (stratégie utilisée dans la première version de Compalgo), tout en définissant par-dessus l'interface de débogage et de contrôle de l'exécution requise. L'interface de débogage implique alors l'abstraction et l'encapsulation des types de base ainsi que l'abstraction des opérations élémentaires de la machine virtuelle virtuelle (*sic*).

Une spécification formelle et abstraite de l'exécution et de son contrôle permet le libre choix de la stratégie, sans effet de bord là non plus (en particulier, l'interfaçage du contrôle d'exécution ne devra pas être affecté par le choix d'implémentation). Néanmoins, les fonctionnalités de la machine virtuelle de Compalgo étant un sous-ensemble des fonctionnalités de la machine virtuelle de Java, l'implémentation complète d'une machine virtuelle en Java semble être redondante.

Comparatif des outils existants

Date : 04/07/2007

Objet : test comparatif des outils existants pour le développement de compilateurs en Java.

Auteur : Damien Leroux

Version : 4

Résumé

Après avoir brièvement réexposé le principe du processus de compilation, nous nous pencherons sur les outils de génération de compilateurs existant en Java, avec une attention particulière sur leur éventuelle facilité d'utilisation dans le cadre du développement et de la validation d'un nouveau langage. Après avoir récapitulé quelques spécificités des générateurs et outils existants, nous conclurons grâce à un test comparatif sur ce qui apparaît être la solution optimale.

Table des matières

4. Le processus de compilation

L'analyse d'un fichier se passe en général en trois étapes, bien qu'il y ait presque autant de cas particuliers que d'implantation. Tout d'abord un « lexer » segmente le texte source en éléments lexicaux et les transforme en fonction de règles définies. Par exemple le lexer lit « tantque » et retourne la constante numérique TOKEN_TANTQUE. « 123 » deviendrait TOKEN_NUMBER avec en attribut la valeur attachée 123.

Le parser consomme la sortie du lexer et d'après sa grammaire réduit les séquences de symboles terminaux et de non-terminaux à des non-terminaux. À chaque réduction le parser exécute des instructions spécifiques, associées à chaque règle de la grammaire, pour générer par exemple directement le code compilé (dans des cas simples) ou le parse tree ou un AST destiné au générateur de code.

La dernière étape est aussi la partie la plus complexe du compilateur, la génération de code à partir de la sortie du parser, ou directement au cours des réductions syntaxiques. Elle se déroule souvent en plusieurs phases, dépendant du type de sortie du parser. À partir d'un parse tree, on crée un AST qui constitue un « nettoyage » en largeur et en profondeur du parse tree, afin de fournir une structure adaptée et non bruitée1 au compilateur. En général2 le parse tree n'existe pas explicitement en tant que tel, mais est sous-entendu dans la réentrance des règles de grammaire, et l'on construit directement l'AST ou le code compilé lors du parse.

[1] Un AST représente de façon structurée l'information contenu dans le texte source, indépendamment de la forme de la grammaire. Les symboles littéraux, tels la ponctuation, servent à structurer le texte mais ne contiennent aucune information utile. Ils représentent finalement du bruit du point de vue du compilateur, ou en soi une redondance inutile avec la représentation structurée du texte source..

[2] Cf. Principles of Compiler Design, Clifford Wolf, p. 16

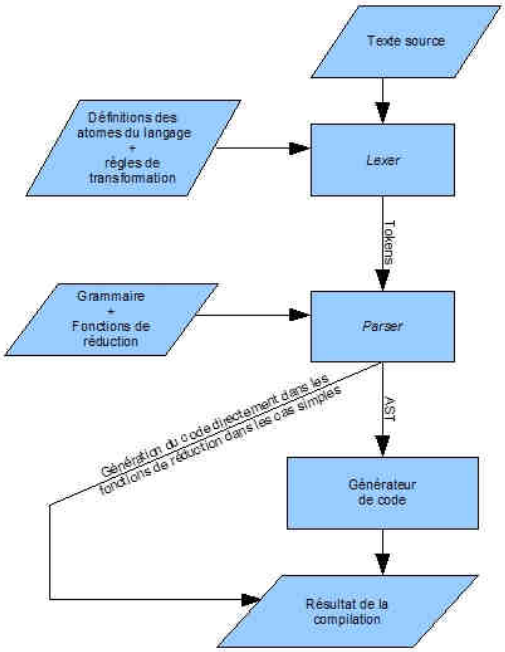

Figure 1 : Processus de compilation

5. Les générateurs de compilateurs et outils existants en Java

L'amalgame entre *parser* et compilateur se retrouve fréquemment dans la littérature. Ainsi, nombre de compilateurs de compilateurs s'affichent comme des « générateurs de *parsers* », alors qu'ils incluent généralement la génération du *lexer* associé et des actions sémantiques permettant la manipulation des *tokens* reconnus pour générer la sortie voulue.

a-visual-llk-parser-generator

Il s'agit d'un EDI proposant de développer sa grammaire sous forme d'arbre visuel, une sorte de « *click n' parse* ». La syntaxe de définition des actions n'est pas particulièrement claire.

ANTLRv3 / ANTLRWorks

Une référence en la matière. ANTLRv3 se présente comme un outil très complet et puissant pour développer une grammaire et le(s) compilateur(s) associé(s).

Un EDI (ANTLRWorks) est disponible et fournit un environnement apparemment très convivial et performant pour encadrer le développement et la maturation d'une grammaire, fournissant notamment de nombreuses fonctionnalité de parcours test et débogage de la grammaire en cours d'élaboration. À l'usage, il se dégage néanmoins un manque flagrant de robustesse.

Coco/R

Il s'agit d'un générateur de compilateurs utilisant une grammaire à attributs. Il génère un scanner fonctionnant comme un automate fini déterministe et un *parser* utilisant la descente récursive. Les conflits LL(1) sont résolus par des tests lexicaux sur le(s) symbole(s) suivant(s) ou sémantiques. La

classe de grammaires acceptée est ainsi LL(k). Coco/R est porté nativement sur de nombreux langages, notamment sur Java.

CUP-Lex

Un combo lex-yacc 100% Java. La spécification du *lexer* est sensiblement la même qu'avec lex. Les deux outils ne sont pas spécifiquement faits pour fonctionner ensemble, mais leur coordination ne demande pas beaucoup d'efforts.

Grammatica

Générateur de *parsers*, il est basé sur des expressions régulières pour définir les *tokens* et la syntaxe EBNF pour la grammaire. Il peut générer un AST ou utiliser des fonctions de rappel pour éviter de passer par l'AST.

Grammatica permet la création « au vol » de *parsers* pour, par exemple, simplifier le débogage d'une grammaire.

JavaCC [tm]

Un générateur en Java de *parsers* LL(1). Ecrit en Java, il est largement utilisé, notamment au sein de l'équipe SIG de l'IRIT, bien documenté, et basé sur des grammaires à attributs étendant l'EBNF.

SABLE CC

C'est un générateur de compilateurs en Java qui a fait l'objet d'une thèse.

Le paragraphe suivant est extrait de ladite thèse.

« En premier lieu, l'utilisation de techniques orientées-objet sert à construire un arbre syntaxique strictement typé qui est conforme à la syntaxe du langage compilé et qui simplifie le déverminage des programmes. En second lieu, l'environnement génère des classes qui traversent l'arbre syntaxique. Ces classes utilisent une version amendée du modèle de conception orienté-objet "le visiteur". Ceci permet d'ajouter des actions à exécuter sur les nœuds de l'arbre syntaxique en utilisant les techniques d'héritage objet. Ces deux décisions font de SableCC un outil qui permet d'abréger le cycle de programmation du développement d'un compilateur. »

SLK Parser Generator

Il s'agit d'un générateur de parsers LL(k) sans récurrence, en utilisant des tables. Des fonctions d'importation (conversion) de grammaires au format SLK sont fournies pour les formats IEEE, ISO, et YACC. Détail important, il ne génère pas de *lexer*.

TinyaP

Ce *parser* abstrait produit un AST à partir d'une grammaire sans attribut basée sur la BNF et regroupe les deux premières étapes du processus de compilation.

Il n'introduit pas de transformation des *tokens* en sortie du *lexer*, mais l'AST résultat offre la même puissance de représentation. TinyaP ne propose pas de définir des fonctions de transformation des *tokens* ni d'actions sémantiques, la forme de sa grammaire pilote entièrement la création de l'AST, sans ajout de code.

TinyaP fournit une API pour parser un texte, et une API pour parcourir ou (dé)sérialiser l'AST résultant.

Malheureusement, TinyaP souffre de deux gros défauts de jeunesse : il n'offre aucune facilité de débogage de la grammaire en cours de développement (en particulier la détection des récurrences à gauche, la détection des ambigüités grammaticales), et le pontage Java, utilisant du code natif, est problématique.

6. Test comparatif

Une grammaire succincte mais précise a été définie pour ce test, reconnaissant un dialecte proche de Scheme (qui est un dialecte de LISP).

```
Le texte source est composé d'un unique ATOME.

Un ATOME est soit un NOMBRE, soit un SYMBOLE, soit une CHAINE, soit une LISTE.

Un NOMBRE est soit « 0 », soit un chiffre non nul suivi d'éventuels chiffres. Cette
spécification implique que « 001234 » sera reconnu comme la séquence « 0 0 1234 ».

Un COMMENTAIRE débute avec « -- » et se termine à la fin de la ligne. On ne s'intéressera pas
aux blocs de commentaires.
```

```
Un SYMBOLE est composé d'un éventuel préfixe « # » ou « _ », suivi d'un radical composé de
lettres, de chiffres, ou du caractère « _ », et éventuellement terminé par un suffixe « ? » ou
« ! ».

Une CHAINE est encadrée par des guillemets doubles et est constituée de caractères forcément
différents des guillemets doubles (sans autre restriction).

Une LISTE démarre avec une parenthèse ouvrante « ( » et termine avec une parenthèse fermante
« ) ». Entre les deux peut apparaître un nombre quelconque d'atomes quelconques.
```

Cette grammaire permet un test rapide et suffisant pour se faire une idée de la facilité d'édition et de correction des fichiers de spécification des outils, ainsi que de la robustesse desdits outils.

Quatre outils ont été sélectionnés pour ce test : ANTLR avec son interface ANTLRWorks, puis JavaCC, Coco/R, et CUP-Lex, qui tous trois disposent d'un plugin Eclipse. ANTLR propose aussi un plugin Eclipse, mais il ne semble pas fonctionnel.

Les autres outils ont été exclus du test pour une ou plusieurs des raisons suivantes :

- documentation inconsistante ou introuvable (Grammatica, SableCC).
- inadaptation ou manque de fonctionnalités (TinyaP, SLK, a-visual-llk-parser-generator).

Aucune cible n'étant spécifiée ni utile dans le cadre de ce test, il est simplement demandé aux parsers de générer l'AST représentant l'information du texte source. Le seul intérêt dans le cadre du test est de montrer où et comment se définissent les actions sémantiques.

Fichiers de spécification

La présente section a uniquement pour but la comparaison formelle des diverses syntaxes en présence. Elle permettra de se rendre compte du caractère « agréable à l'œil » ou non de la syntaxe de spécification définie par chaque outil. Seul CUP-Lex nécessite deux fichiers de spécification, un pour le *lexer* et un pour le *parser*. Les trois autres permettent de tout spécifier en un seul fichier.JavaCC

```
options
{
	STATIC = false ;	// Instanciation d'un new parseur au lieu d'appeler ReInit

	USER_TOKEN_MANAGER	= false ;	// default is false
	USER_CHAR_STREAM	= false ;	// default is false

	JAVA_UNICODE_ESCAPE = true ;  // default is false
	UNICODE_INPUT		= true ;	// default is false
	// true value causes XXXTokenManager.java to be generated so it
	// can handle Unicode beyond the ASCII range
}

PARSER_BEGIN(Grammaire)

package fr.irit.sig.compilateur.JavaCC ;

import java.util.* ;
import java.io.* ;		/* FileNotFoundException, FileInputStream */
import java.util.regex.* ;

import fr.irit.sig.compilateur.Ast;
import fr.irit.sig.compilateur.AstBuilder;


public final class Grammaire extends AstBuilder
{
	public Grammaire(File ficAlgo) throws FileNotFoundException,
UnsupportedEncodingException
```

```
        {
                this(new GrammaireTokenManager(
                        new JavaCharStream(
                                new InputStreamReader(
                                        new FileInputStream(ficAlgo), "UTF-8")))) ;
        }
}

PARSER_END(Grammaire)

/* Espaces et fins de lignes */
SKIP : {" " | "\t" | "\n" | "\r" | "\f"}

/* Commentaires */
MORE :
{
 "--" : IN_SINGLE_LINE_COMMENT
}

<IN_SINGLE_LINE_COMMENT>
SPECIAL_TOKEN :
{
 <SINGLE_LINE_COMMENT: "\n" | "\r" | "\r\n" > : DEFAULT
}

<IN_SINGLE_LINE_COMMENT, IN_MULTI_LINE_COMMENT>
MORE :
{
 < ~[] >
}

// Certains Tokens sont préfixés par # : règle temporaire pour faciliter l'écriture des Tokens
TOKEN :
{ <#LETTER: ["a"-"z","A"-"Z","_","#"]>
| <STR:  "\"" (~["\""])* "\"" >
| <#SYMPRE: (["_","#"]) | (<LETTER>)) >
| <#SYMRAD: ( ["_","-","0"-"9"]) | (<LETTER>) ) >
| <#SYMSUF: (["!","?"]) >
| <SYM:  ((<SYMPRE>) (<SYMRAD>)* (<SYMSUF>)? | (["<",">"]) (["="])? | (["+","-","/","*","="]))
>
| <NUM:  ("0" | (["0"-"9"])+) >
| <LPAR: "(" >
| <RPAR: ")" >
}

/**
 * point d'entrée de la grammaire :
        Scheme ::= <Atome>*.
 */
void Scheme() : {}    { Atome()
                      }

/**
 * règle Atome :
 * Atome ::= <
 */
void Atome() : { Token val; }
        { Liste()
        | val=<NUM>                  { add( Ast.newOp("NUM").add(Ast.newLeaf(val.toString()))

        .setProps(val.beginLine,val.beginColumn,"(unset)")); }

        | val=<STR>                  { add( Ast.newOp("STR").add(Ast.newLeaf(val.toString()))

        .setProps(val.beginLine,val.beginColumn,"(unset)")); }

        | val=<SYM>                  { add( Ast.newOp("SYM").add(Ast.newLeaf(val.toString()))

        .setProps(val.beginLine,val.beginColumn,"(unset)")); }
        }

void Liste() : { Token val; }
        { val=<LPAR>                 { pushNode(Ast.newOp("LIST")

        .setProps(val.beginLine,val.beginColumn,"(unset)")); }
         ListeAtomes()
        }
```

```
void ListeAtomes() : {}
       { <RPAR>                    { validate(); }
       | Atome() ListeAtomes()
       }
```

CUP-Lex

- Spécification du *lexer*

```
package fr.irit.sig.compilateur.CUP_Lex;

import java_cup.runtime.*;
import java.io.IOException;
import java_cup.runtime.Symbol.*;

class TokenValue {
 public int lineBegin;
 public int charBegin;
 public String text;
 public String filename;

 TokenValue()
 {
 }

 TokenValue(String text, int lineBegin, int charBegin, String filename)
 {
 //System.out.println("new TokenValue "+text+" "+lineBegin+" "+charBegin+" "+filename);
 this.text = new String(text);
 this.lineBegin = 1+lineBegin;       // Jlex compte à partir de 0
 this.charBegin = 1+charBegin;       // Jlex compte à partir de 0
 this.filename = filename;
 }

 public String toString()
 {
 return text;
 }

 public boolean toBoolean()
 {
 return Boolean.valueOf(text).booleanValue();
 }

 public int toInt()
 {
 return Integer.valueOf(text).intValue();
 }
}

%%

%class FoobarLex

%init{
       // TODO: code that goes to constructor
%init}

%{
       public String sourceFilename;
       private StringBuffer sb;
       int ll,lc;

       private void error()
       throws IOException
       {
              throw new IOException("illegal text at line = "+yyline+", column = "
                                                              +yycolumn+", text =
'"+yytext()+"'");
       }
%}

%full
%line
```

```
%column

// %public
%final
// %abstract

%cupsym Symbol
%cup
// %cupdebug

%state COMMENTS
%state STRING

%eofval{
 return new Symbol(FoobarSym.EOF, null);
%eofval}

SYMPREFIX=[#A-Za-z_]
SYMRADIX=[A-Za-z_0-9-]
SYMSUFFIX=[!?]

SYM= ( ({SYMPREFIX}) ({SYMRADIX})* ({SYMSUFFIX})? | ">=" | "<=" | ([*+/><=-]) )

NUM = ( "0" | ([0-9])+ )
WHITE_SPACE=([\ \n\r\t\f])+

%%

<YYINITIAL> {SYM} {
 return new Symbol(FoobarSym.SYM, new TokenValue(yytext(),
                                     yyline, yycolumn, sourceFilename));
}

<YYINITIAL> {NUM} {
 return new Symbol(FoobarSym.NUM, new TokenValue(yytext(),
                                     yyline, yycolumn, sourceFilename));
}

<YYINITIAL> {WHITE_SPACE} { }

<YYINITIAL> "(" {
 return new Symbol(FoobarSym.LPAR, new TokenValue(yytext(),
                                     yyline, yycolumn, sourceFilename));
}

<YYINITIAL> ")" {
 return new Symbol(FoobarSym.RPAR, new TokenValue(yytext(),
                                     yyline, yycolumn, sourceFilename));
}

<YYINITIAL> "--" {
 yybegin(COMMENTS);
}

<COMMENTS> [^\n] {
}

<COMMENTS> [\n] {
 yybegin(YYINITIAL);
}


<YYINITIAL> [\"] {
        yybegin(STRING);
        sb=new StringBuffer("\"");
        ll=yyline;
        lc=yycolumn;
}

<STRING> [^\"] {
        sb.append(yytext());
}

<STRING> [\"] {
        sb.append("\"");
        yybegin(YYINITIAL);
        return new Symbol(FoobarSym.STR,
```

```
                                        new TokenValue(sb.toString(),ll,lc,sourceFilename));
}

<YYINITIAL> . {
 return new Symbol(FoobarSym.error, null);
}
```

- Spécification du *parser*

```
package fr.irit.sig.compilateur.CUP_Lex;

import java_cup.runtime.*;

import fr.irit.sig.compilateur.Ast;
import fr.irit.sig.compilateur.AstBuilder;

parser code
{:
	AstBuilder ast;

	Ast Parse()
	{
		//try { return (Ast)debug_parse().value; }
		try { return (Ast)parse().value; }
		catch(Exception e) { System.out.println("Exception during parse");
e.printStackTrace(); return null; }
	}

	Ast getAst()
	{
		return ast.getAst();
	}

:}

action code
{:
	void reset()
	{
		parser.ast=new AstBuilder();
	}
	void newPair(String car, String cdr)
	{
		parser.ast.add(Ast.newOp(car).add(Ast.newLeaf(cdr)));
	}
	void newPair(String car, String cdr,int l,int c, String fnam)
	{
		parser.ast.add(Ast.newOp(car).add(Ast.newLeaf(cdr)).setProps(l,c,fnam));
	}
	void newNum(String value,int l,int c, String fnam)
	{
		newPair("NUM",value,l,c,fnam);
	}
	void newSym(String value,int l,int c, String fnam)
	{
		newPair("SYM",value,l,c,fnam);
	}
	void newStr(String value,int l,int c, String fnam)
	{
		newPair("STR",value,l,c,fnam);
	}
	void startList(int l,int c, String fnam)
	{
		parser.ast.pushNode(Ast.newOp("LIST").setProps(l,c,fnam));
	}
	void startList()
	{
		parser.ast.pushNode(Ast.newOp("LIST"));
	}
	void endList()
	{
		parser.ast.validate();
	}
:}
```

```
/* Preliminaries to set up and use the scanner. */

terminal TokenValue
       LPAR, RPAR, SYM, NUM, STR;

non terminal Ast scheme;
non terminal TokenValue liste, atome, liste_atomes;

start with scheme;


scheme
::=                                        {: reset(); :}
       atome                          {: RESULT=parser.getAst(); :}
 | error
;

atome
::= NUM:x                             {: newNum(x.text,x.lineBegin,x.charBegin,"(unset)"); :}
 | SYM:x                              {: newSym(x.text,x.lineBegin,x.charBegin,"(unset)"); :}
 | STR:x                              {: newStr(x.text,x.lineBegin,x.charBegin,"(unset)"); :}
 | liste
;

liste
::= LPAR:x                            {: startList(x.lineBegin,x.charBegin,"(unset)"); :}
 liste_atomes
;

liste_atomes
::= RPAR                              {: endList(); :}
 | atome
 liste_atomes
 | error RPAR
;
```

Coco/R

```
/* imports */
import fr.irit.sig.compilateur.Ast;
import fr.irit.sig.compilateur.AstBuilder;

COMPILER FooBar

       /* le code Java suivant est inclus dans la classe Parser */
       static AstBuilder ast=new AstBuilder();
       public Ast getAst() { return ast.getAst(); }


/*---------------------------------------------------------------------*/

CHARACTERS
 letter = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz".
 digit = "0123456789".
 spc = ' '.
 cr = '\r'.
 lf = '\n'.
 tab = '\t'.
 stringCh = ANY - '"' - '\\' - cr - lf.
 charCh = ANY - '\'' - '\\' - cr - lf.
 printable = '\u0020' .. '\u007e'.
 hex = "0123456789abcdef".
 symPrefix = letter + "_#".
 symRadix = letter + "_-" + digit.
 symSuffix = "!?".
 oper = "+-=*/<>".

TOKENS
 SYM           = symPrefix {symRadix} [symSuffix] | ">=" | "<=" | oper.
 NUM           = digit { digit }.
 STR           = '"' { stringCh | '\\' printable } '"'.
 char          = '\'' ( charCh | '\\' printable { hex } ) '\''.

COMMENTS FROM "-[" TO "]-" NESTED
COMMENTS FROM "--" TO cr lf
```

```
IGNORE cr + lf + tab + spc

/*-------------------------------------------------------------------*/

PRODUCTIONS

liste
       = "("                (. ast.pushNode(Ast.newOp("LIST")
                                   .setProps(t.line,t.col+1,"(unset)")); .)
        listeAtomes
       .

listeAtomes
       = ")"                (. ast.validate(); .)
       | atome listeAtomes
       .

atome
       = NUM                (. ast.add( Ast.newOp("NUM").add(Ast.newLeaf(t.val))
                                   .setProps(t.line,t.col+1,"(unset)") ); .)

       | SYM                (. ast.add( Ast.newOp("SYM").add(Ast.newLeaf(t.val))
                                   .setProps(t.line,t.col+1,"(unset)") ); .)

       | STR                (. ast.add( Ast.newOp("STR").add(Ast.newLeaf(t.val))
                                   .setProps(t.line,t.col+1,"(unset)") ); .)

       | liste
       .

FooBar
       =                    (. ast.reset(); .)
        atome
       .

END FooBar.
```

ANTLRv3

```
grammar grammaire_astbuilder;

options {
       output=AST;
       backtrack=true;
}



@lexer::header {
       package fr.irit.sig.compilateur.ANTLR;
}

@parser::header {
       package fr.irit.sig.compilateur.ANTLR;

       import fr.irit.sig.compilateur.Ast;
       import fr.irit.sig.compilateur.AstBuilder;
}

@parser::members {
       private AstBuilder ast;

       public Ast getAst() { return ast.getAst(); }
}

/*
 * LEXER
 */

WS     :      ('\r'|'\t'|'\n'|' ')+ {$channel=HIDDEN;};

DQUOTE :      '"';

STR    :      '"' ((~('"'))*) '"';

NUM    :      '0' | ('1'..'9') ('0'..'9')+ ;
```

```
RPAREN :      ')';
LPAREN :      '(';

SYM    :      (('a'..'z'|'A'..'Z'|'#'|'_') ('a'..'z'|'A'..'Z'|'0'..'9'|'_')* ('!'|'?')?)
       |      '<' '='?
       |      '>' '='?
       |      '*'|'/'|'+'|'-'|'=' ;

/*
 * PARSER
 */


/*
chaine : DQUOTE
        (options {greedy=false;}:.)*
        DQUOTE;
/*/
chaine :      STR;
//*/

atome :( liste
       | SYM                { ast.add(Ast.newOp("SYM").add(Ast.newLeaf($text))); }
       | chaine             { ast.add(Ast.newOp("STR").add(Ast.newLeaf($text))); }
       | NUM                { ast.add(Ast.newOp("NUM").add(Ast.newLeaf($text))); }
       );

liste :( RPAREN             { ast.pushNode(Ast.newOp("LIST")); }
        atome*
        LPAREN       { ast.validate(); }
       );

scheme :                    { ast = new AstBuilder(); }
        atome ;
```

Résultats

Convivialité du langage de spécification

La spécification d'un langage avec CUP-Lex est notablement plus verbeuse qu'avec tous les autres outils, de plus on a largement l'impression de programmer directement les automates du *lexer* et du *parser*, ce qui est peu confortable. La grammaire, ses attributs, et le langage Java (notamment pour les actions sémantiques) sont étroitement corrélés et l'on génère très vite des bugs et inconsistances dans le code final du *lexer* et du *parser*. À côté de cela, ANTLR et JavaCC proposent une syntaxe plutôt confortable.

Coco/R, lui, pousse la convivialité à un niveau supérieur ; le plugin Eclipse propose de créer un squelette de fichier de spécification, pré-rempli avec quelques déclarations claires et explicites, ce qui permet de rentrer instantanément dans le bain.

Classement :

1. Coco/R
2. ANTLR
3. JavaCC
4. CUP-Lex

Propreté du code généré

Il est intéressant de comptabiliser les warnings émis par les fichiers générés par ces divers outils. Ces warnings concernent majoritairement des imports, des variables, ou des bouts de code inutilisés.

Ils ne sont en aucun cas rédhibitoires mais donnent une indication sur la propreté des mécanismes à l'œuvre dans les entrailles de ces outils.

Classement :

1. Coco/R 2 warnings répartis dans 2 fichiers
2. ANTLRv3 7 warnings répartis dans 2 fichiers
3. CUP-Lex 19 warnings répartis dans 2 fichiers
4. JavaCC 26 warnings répartis dans 3 fichiers

Robustesse et rapidité

Pour ce test, nous avons utilisé un fichier minuscule contenant la chaîne `(tata (yoyo "hop") 23 ())` ainsi qu'un fichier généré aléatoirement, composé uniquement d'éléments valides, long de 14809 octets.

ANTLR refuse de parser le gros fichier, et ne fournit aucune piste pour résoudre le problème. Au lieu de cela, il engloutit l'intégralité du tas de Java, jusqu'à ce qu'il soit plein et que la machine virtuelle lève l'exception correspondante. Dans ANTLRWorks, il arrive aussi que le débogage ne puisse se lancer pour une raison ou une autre, et il ne fournit aucune information sur la cause.

L'échec de ce test avec ANTLR montre d'un autre côté le manque de robustesse d'ANTLR, ainsi que le manque de retour d'information en cas de problème pendant et après la génération du code du *parser*, nous privant de la certitude d'un développement serein avec cet outil. De ce fait, étant avéré qu'il ne répond pas à nos attentes, ANTLR est retiré des conclusions de ce test.

Voici la sortie du programme de test incluant les trois *parsers* générés (en excluant ANTLR qui ne termine pas le test) :

```
Démarrage du test...
Coco/R         Temps (moyenne sur 10000 appels) : 0.0146985 s
CUP-Lex        Temps (moyenne sur 10000 appels) : 0.045936003 s
JavaCC         Temps (moyenne sur 10000 appels) : 0.0323656 s
Meilleure performance : Coco/R (0.0146985 s/parse)
```

Classement :

- Coco/R
- JavaCC
- CUP-Lex
- ANTLR *(DNF)*

Retour d'erreur (comportement par défaut)

Pour ce test on fournit aux *parsers* un fichier contenant un *token* invalide :

```
("ceci" est 23 un (test 42) ()
 (qui) (contient
              (une . erreur!)
 )
)
```

On s'attend à voir une erreur sur le « . » à la 3e ligne. Au début de la ligne sont deux tabulations, qui vont nous donner de plus amples détails sur le comportement des parsers face à l'erreur de syntaxe. On voit ici les différents comportements (par défaut) des parsers générés :

```
Démarrage du test...
Coco/R         -- line 3 col 8: invalid listeAtomes
Temps (moyenne sur 1 appels) : 0.0 s
CUP-Lex        Syntax error
Temps (moyenne sur 1 appels) : 0.0 s
JavaCC         Exception in thread "main" fr.irit.sig.compilateur.JavaCC.TokenMgrError:
Lexical error at line 3, column 22. Encountered: "." (46), after : ""
       at
fr.irit.sig.compilateur.JavaCC.GrammaireTokenManager.getNextToken(GrammaireTokenManager.java:5
74)
       at fr.irit.sig.compilateur.JavaCC.Grammaire.jj_ntk(Grammaire.java:195)
       at fr.irit.sig.compilateur.JavaCC.Grammaire.ListeAtomes(Grammaire.java:71)
       at fr.irit.sig.compilateur.JavaCC.Grammaire.ListeAtomes(Grammaire.java:81)
       at fr.irit.sig.compilateur.JavaCC.Grammaire.Liste(Grammaire.java:67)
[...]
```

Le message d'erreur de CUP a le mérite de nous signaler la non-acceptation du texte source, mais rien de plus. Le message d'erreur de Coco/R est déjà plus signifiant, sans conversion des caractères en colonnes. Le message d'erreur de JavaCC est pour ainsi dire parfait, précisant le *token* fautif et incluant même la conversion des caractères en colonnes (sur la base de huit espaces par tabulation visiblement).

Classement :

1. JavaCC
2. Coco/R

7. CUP-LexRécapitulatif

Le tableau suivant résume les propriétés des différents outils cités, ainsi que les résultats et conclusions du test comparatif. Les outils sont grossièrement classés par utilisabilité décroissante de gauche à droite.

	Coco/R	***JavaCC***	***CUP/Lex***	***ANTLRv3***	***TinyaP***	***SLK***	***Grammatica***	***SableCC***	***a-visual-llk-parser-generator***
Type[3]	End-to-	End-to-	End-to-	End-to-	Front-	Back-	End-	End-to-	End-to-

[3] End-to-end, Front-end, et Back-end se rapportent au positionnement de l'outil dans la chaîne de compilation. End-to-end englobe la totalité de la chaîne. Front-end ne définit au mieux que le *lexer* et le *parser*,

	end	end	end	end	end	end	to-end	end	end
Classes de grammaires reconnues	LL(1)	LL(1)	LALR(1)	LL(1)	LL(k)	LL(k)	LL(k)	LALR(1) LL(1)	LL(k)
Plateformes	Natif Java	Natif Java	Natif Java	Natif Java	Linux, Win32, JNI[4]	Linux, Win32	Natif Java	Natif Java	Natif Java
EDI	Eclipse	Eclipse	Eclipse	Eclipse, ANTLR-Works	*N/A*	*N/A*	*N/A*	*N/A*	*ibid.*
Convivialité	+++++	+++	++	+++	+++	*N/A*	*N/A*	*N/A*	*N/A*
Rapidité	+++++	+++	++++	*N/A*	*N/A*	*N/A*	*N/A*	*N/A*	*N/A*
Documentation	++++	++++	+++	+	+	++	-	+	+
Qualité du code généré	+++++	+	+	+++	*N/A*[5]	*N/A*	*N/A*	*N/A*	*N/A*
Licence	GPL	BSD	Compa-tible GPL	BSD	MIT	Propri-étaire	LGPL	LGPL	GPL

8. Conclusion

Il existe nombre de compilateurs de compilateurs et d'outils similaires tournant sur la plateforme Java, dont quatre ont été testés en relative profondeur.

Au vu des informations acquises, il apparaît qu'au moins pour la phase de développement du langage, le choix devra plutôt se porter sur les outils fournissant de fortes capacités de débogage et de surveillance de la grammaire.

ANTLRv3 + ANTLRWorks permet la visualisation des automates et diagrammes syntaxiques partiels générés, mais son manque de robustesse, à l'usage, est rédhibitoire. Grâce à la plateforme Eclipse, d'autres outils peuvent être utilisés dans un environnement convivial, réduisant ainsi énormément la valeur ajoutée d'ANTLRWorks.

---

réciproquement Back-end ne définit que le *parser* et/ou la génération de code.

[4] Le pontage Java utilise du code natif et n'est pas encore validé pour Windows.

[5] TinyaP ne génère pas de code, il utilise le même moteur pour parser les fichiers de grammaire (spécification) et les textes source dans le langage spécifié.

Les fonctionnalités apportées par les quatre outils testés sont à peu près équivalentes en termes de puissance, ainsi ce ne peut être un critère de départage.

Coco/R se révèle finalement être à la fois le plus convivial, le plus robuste, et le plus rapide des outils testés, abordant le dernier virage avec plusieurs longueurs d'avance sur ses confrères.

9. Références

- a-visual-llk-parser-generator

https://a-visual-llk-parser-generator.dev.java.net/

- ANTLRv3

http://www.antlr.org/works/index.html

- Coco/R

http://www.ssw.uni-linz.ac.at/coco/

- Grammatica

http://grammatica.percederberg.net/

- JavaCC

https://javacc.dev.java.net/doc/features.html

- Principles of Compiler Design, Clifford Wolf

http://www.clifford.at/papers/2004/compiler/slides.pdf

- SableCC, an object-oriented compiler framework

http://sablecc.sourceforge.net/thesis/thesis.html

- SLK Parser Generator

http://home.earthlink.net/~slkpg/

- TinyaP

http://code.google.com/p/tinyap/

- Wikipédia, parsers

http://en.wikipedia.org/wiki/Parser et pages similaires

Grammaire formelle

```
## (@@) les commentaires sont préfixés par ##, remplacer globalement au besoin.
## (@@) suivre le marqueur (@@) pour les commentaires importants.

## Grammaire de Compalgo
## Réécriture de la grammaire existante pour obtenir la nature LL(1)
## Auteur : Damien Leroux
## (NdA :
##    - pour mener à bien le refactoring j'opte pour une réécriture complète.
##                    pour notamment deux raisons :
##                           - on veut éviter d'utiliser les constructs EBNF massivement
##                             employés dans la version actuelle
##                           - la grammaire sous sa forme actuelle n'est pas exempte de bugs,
##                             il est beaucoup plus facile de repartir de zéro que de cher-
##                             cher les erreurs et les corriger.
##    - stratégie "diviser pour mieux rêgner".
##    - ce format sera à adapter aux outils de validation puis de génération de
##            parser finalement choisis. (@@)
##    - règles d'écriture des productions :
##        - factorisation à gauche des éléments communs à plusieurs règles
##        - une alternative est forcément et uniquement en partie
##                    droite d'une règle de production.
## )
##

##
## DEBUT Lexèmes
##

       ## (@@) Chaîne vide normalement implicite, il faudra forcément
       ## redéfinir cette chaîne "DEF EPSILON" lors de toute conversion
              DEF EPSILON

       ## Ponctuation

              POINT          : '.' .
              DEUXPOINTS     : ':' .
              PARG           : '(' .
              PARD           : ')' .
              POINTVIRGULE: ';' .
              VIRGULE        : ',' .
              CROCO          : '[' .
              CROCF          : ']' .

              ## Les < > nécessaires à l'encadrement des NomsDeTypes
              ## sont décrits dans la sections opérateurs booléens,
              ## resp. INF et SUP

       ## Intervalles d'indiçage des tableaux

              INTERVALLE     : '..' .

       ## Littéraux

              PROGRAMME      : 'programme' .
              GLOSSAIRE      : 'glossaire' .
              DEBUT          : 'début' .
              FIN                  : 'fin' .
              SINON          : 'sinon' .
              SI                   : 'si' .
              ALORS          : 'alors' .
              TANTQUE        : 'tantque' .
              FAIRE          : 'faire' .
              TYPEDECL       : 'type' .
              CONSTANTE      : 'constante' .
              RETOURNER      : 'retourner' .
              RETOURNE       : 'retourne' .

              IMPORTER       : 'importer' .
              EXTENSION_ALGO : 'algo' .
              EXTENSION_SPEC : 'spec' .

              TABLEAU        : 'tableau' .
              DE                   : 'de' .
```

```
                ENREGISTREMENT : 'enregistrement' .

        ## Traitement des exceptions

                TRAITE_EXCEPTION      : 'traite-exception' .
                LORSQUE                       : 'lorsque' .

                DECLENCHER     : 'déclencher' .
                DECLENCHE      : 'déclenche' .

        ## Sous-programmes

                FONCTION       : 'fonction' .
                PROCEDURE      : 'procédure' .

        ## Paramètres des sous-programmes

                PARAM_ENTREE: 'entrée' .
                PARAM_SORTIE: 'sortie' .
                PARAM_MAJ      : 'màj' .

        ## Opérateur d'affectation

                AFFECTATION    : '<-' .

        ## Opérateurs arithmétiques

                PLUS           : '+' .
                MOINS          : '-' .
                DIV                    : '/' .
                MULT           : '*' .
                MODULO         : 'modulo' .

        ## Opérateurs booléens

                ## Comparable -> Booléen

                NEQ                    : '/=' .
                SUPEQ          : '>=' .
                INFEQ          : '<=' .
                SUP                    : '>' .
                INF                    : '<' .
                EQ                     : '=' .

                ## Booléen -> Booléen

                NON                    : 'non' .

                ## Booléen x Booléen -> Booléen

                ET                     : 'et' .
                OU                     : 'ou' .

        ## Constantes booléennes

                VRAI           : 'VRAI'.
                FAUX           : 'FAUX'.

        ## Classes

                ENTIER : '_id_entier_' .
                REEL : '_id_reel_' .
                ## ID regroupe sous-programmes et variables
                ID : '_id_' .
                ID_CONSTANTE : '_id_constante_' .
                ID_TYPE : '_id_type_' .

        ## (@@) WORK IN PROGRESS
                TODO           : 'TODO' .
                FIXME          : 'FIXME' .

##
## FIN Lexèmes
##

##
## DEBUT Productions
```

```
##

## (@@) Renommer __start__ au besoin pour spécifier le point d'entrée du parser.
##      (e.g. CompAlgo)

##
## DEBUT STRUCTURE GLOBALE
##

        __start__
                : importations
                .

        ## * un fichier compalgo commence par une éventuelle série d'importations

        importations
                : decl_import importations
                | declarations_globales
                .

        ## * puis des déclarations de types et de constantes

        declarations_globales
                : decl_type_ou_constante_ou_xcep declarations_globales
                | declarations_sous_programmes
                .

        ## * prévoir des variables globales ? (@@)

        ## * on peut ensuite définir des sous-programmes

        declarations_sous_programmes
                : decl_sous_programme declarations_sous_programmes
                | declaration_programme
                | EPSILON
                .

        ## * finalement vient le programme principal, s'il en est.
        ##   Dans le cas d'une bibliothèque, il n'y a rien après les sous-programmes.

        declaration_programme
                : PROGRAMME ID decl_corps_sous_programme
                .

##
## FIN STRUCTURE GLOBALE
##

##
## DEBUT BLOC INSTRUCTIONS
##

        bloc_d_instructions
                : DEBUT sequence_d_instructions ( traitement_exception FIN
                                                | FIN
                                                ).

        sequence_d_instructions
                : instruction  ( sequence_d_instructions
                               | EPSILON
                               ).

        ## DEBUT TRAITEMENT EXCEPTIONS

        traitement_exception
                : TRAITE_EXCEPTION liste_lorsque_exception
                .

        lorsque_exception
                : LORSQUE ID_CONSTANTE FAIRE sequence_d_instructions FIN LORSQUE POINTVIRGULE
                .

        liste_lorsque_exception
                : lorsque_exception   ( liste_lorsque_exception
                                      | EPSILON
                                      ).
```

```
	## FIN TRAITEMENT EXCEPTIONS

## FIN BLOC INSTRUCTIONS


## DEBUT INSTRUCTIONS

	instruction
		: instruction_si
		| instruction_tantque
		| instruction_identifiant
		| instruction_retourner
		| instruction_declencher
		.

	## DEBUT RETOURNER

	instruction_retourner
		: RETOURNER expression POINTVIRGULE
		.

	## FIN RETOURNER

	## DEBUT DECLENCHER

	instruction_declencher
		: DECLENCHER PARG ID_CONSTANTE PARD POINTVIRGULE
		.

	## FIN DECLENCHER

	##
	## DEBUT SI

		instruction_si
			: SI expression ALORS sequence_d_instructions	( SINON
sequence_d_instructions finsi
									| finsi
									).

		finsi
			: FIN SI POINTVIRGULE
			.

	## FIN SI
	##

	##
	## DEBUT TANTQUE

		instruction_tantque
			: TANTQUE expression FAIRE sequence_d_instructions FIN TANTQUE
POINTVIRGULE
			.

	## FIN TANTQUE
	##

	##
	## DEBUT INSTRUCTION IDENTIFIANT

		instruction_identifiant
			: ID ( instruction_sous_programme
			     | variable_acces_element instruction_affectation
			     ).

		##
		## DEBUT INSTRUCTION AFFECTATION

			instruction_affectation
				: AFFECTATION expression POINTVIRGULE
				.

		## FIN INSTRUCTION AFFECTATION
		##
```

```
                ##
                ## DEBUT INSTRUCTION APPEL DE SOUS PROGRAMME

                        instruction_sous_programme
                                : PARG liste_params_effectifs POINTVIRGULE
                                | POINTVIRGULE
                                .

                ## FIN INSTRUCTION APPEL DE SOUS PROGRAMME
                ##

        ## FIN INSTRUCTION IDENTIFIANT
        ##

## FIN INSTRUCTIONS



##
## DEBUT DECLARATIONS
##

        ## DEBUT IMPORT

                decl_import
                        : IMPORTER ID POINT extension_valide POINTVIRGULE
                        .

                extension_valide
                        : EXTENSION_ALGO
                        | EXTENSION_SPEC
                        .

        ## FIN IMPORT

        ## DEBUT Types et constantes

                decl_type_ou_constante_ou_xcep
                        : decl_type
                        | decl_constante
                        | decl_exception
                        .

                ## DEBUT EXCEPTION

                        decl_exception
                                : ID_CONSTANTE INF ID_TYPE SUP POINTVIRGULE
                                .

                ## FIN EXCEPTION

                ## DEBUT CONSTANTE

                        decl_constante
                                : CONSTANTE ID_CONSTANTE type EQ expression POINTVIRGULE
                                .

                ## FIN CONSTANTE

        ## DEBUT TYPE
                ## DEBUT DECLARATION
                        decl_type
                                : TYPEDECL ID_TYPE DEUXPOINTS type_type
                                .

                        ## (@@) on force les enregistrements à avoir au moins deux champs,
                        ## sinon pourquoi faire un enregistrement ?

                        type_type
                                ## Tablocroco !
                                : TABLEAU CROCO expression INTERVALLE expression CROCF DE type
POINTVIRGULE
                                | ENREGISTREMENT decl_champ VIRGULE decl_champ
liste_decls_champs
                                | type POINTVIRGULE
                                .
```

```
                decl_champ
                        : ID type
                        .

                liste_decls_champs
                        : VIRGULE decl_champ liste_decls_champs
                        | POINTVIRGULE
                        .

        ## FIN DECLARATION

        ## DEBUT UTILISATION

                type
                        : INF ID_TYPE SUP
                        .

        ## FIN UTILISATION
## FIN TYPE

## FIN Types et Constantes

## DEBUT PARAMETRES

        param_effectif
                : expression
                .

        liste_params_effectifs
                : param_effectif      ( VIRGULE liste_params_effectifs
                                              | PARD
                                              ).

        decl_param_formel
                : direction_param ID type
                .

        liste_decls_params_formels
                : decl_param_formel   ( VIRGULE liste_decls_params_formels
                                              | PARD
                                              ).

        direction_param
                : PARAM_ENTREE
                | PARAM_SORTIE
                | PARAM_MAJ
                .

## FIN PARAMETRES

## DEBUT SOUS-PROGRAMMES

        decl_sous_programme
                : decl_fonction
                | decl_procedure
                .

        decl_fonction
                : FONCTION ID  ( PARG liste_decls_params_formels decl_retour_fonction
                                        | decl_retour_fonction
                                        ).

        decl_retour_fonction
                : RETOURNE type ( decl_exceptions
                                        | decl_corps_sous_programme
                                        | POINTVIRGULE
                                        ).

        decl_exceptions
                : DECLENCHE ID_CONSTANTE     ( decl_liste_exceptions
                                                              |
decl_corps_sous_programme
                                                              | POINTVIRGULE
                                                              ).

        decl_liste_exceptions
                : VIRGULE ID_CONSTANTE( decl_liste_exceptions
```

```
                                                        | decl_corps_sous_programme
                                                        ).


                decl_corps_sous_programme
                        : decl_glossaire
                        | bloc_d_instructions
                        .

                decl_procedure
                        : PROCEDURE ID        ( PARG liste_decls_params_formels    (
decl_exceptions

                                | decl_corps_sous_programme

                                | POINTVIRGULE

                                )
                                                        | decl_exceptions
                                                        | decl_corps_sous_programme
                                                        | POINTVIRGULE
                                                        ).

        ## FIN SOUS-PROGRAMMES

        ## DEBUT GLOSSAIRE

                decl_glossaire
                        : GLOSSAIRE decls_variables
                        .

                decls_variables
                        : decl_variable ( decls_variables
                                                        | bloc_d_instructions
                                                        ).

                decl_variable
                        : ID type POINTVIRGULE
                        .

        ## FIN GLOSSAIRE

##
## FIN DECLARATIONS
##




##
## DEBUT EXPRESSIONS
##

        expression
                : CHAINE
                | CARACTERE
                | expression_composee
                .

        expression_composee
                : expr_atome_max      ( expr_prio_max
                                      | EPSILON
                                      ).

        variable_ou_sous_programme
                : ID    ( PARG liste_params_effectifs
                        | variable_acces_element
                        ).

        variable_acces_element
                : CROCO expression_composee CROCF variable_acces_element
                | POINT ID variable_acces_element
                | EPSILON
                .
```

```
constante
       : ID_CONSTANTE
       .

## atomes évaluables du langage

expr_atome_0
       : variable_ou_sous_programme
       | constante
       | REEL
       | ENTIER
       | PARG expression PARD
       .

expr_prio_1
       : expr_op_1 expr_atome_1     ( expr_prio_1
                                    | EPSILON
                                    ).

## hardcodage de l'opérateur unaire "-"


expr_atome_1
       : MOINS expr_atome_0
       | expr_atome_0
       .

expr_prio_2
       : expr_op_2 expr_atome_2     ( expr_prio_2
                                    | EPSILON
                                    ).

expr_atome_2
       : expr_atome_1 ( expr_prio_1
                      | EPSILON
                      ).

expr_prio_3
       : expr_op_3 expr_atome_3     ( expr_prio_3
                                    | EPSILON
                                    ).

expr_atome_3
       : expr_atome_2 ( expr_prio_2
                      | EPSILON
                      ).

expr_prio_4
       : expr_op_4 expr_atome_4     ( expr_prio_4
                                    | EPSILON
                                    ).

expr_atome_4
       : VRAI
       | FAUX
       | expr_atome_3 ( expr_prio_3
                      | EPSILON
                      ).

expr_prio_5
       : expr_op_5 expr_atome_5     ( expr_prio_5
                                    | EPSILON
                                    ).

expr_atome_5
       : NON expr_atome_4
       | expr_atome_4 ( expr_prio_4
                      | EPSILON
                      ).

expr_prio_6
       : expr_op_6 expr_atome_6     ( expr_prio_6
                                    | EPSILON
                                    ).

expr_atome_6
       : expr_atome_5 ( expr_prio_5
```

```
                            | EPSILON
                            ).

        expr_atome_max
                : expr_atome_6
                .

        expr_prio_max
                : expr_prio_6
                .

        ## priorité 0 : MOINS unaire (arithmétique)


        expr_op_1
                : MULT
                | DIV
                .

        expr_op_2
                : PLUS
                | MOINS
                .

        expr_op_3
                : MODULO
                .

        expr_op_4
                : INF
                | INFEQ
                | SUP
                | SUPEQ
                | EQ
                | NEQ
                .

        ## priorité 4 bis (4.5) : NON logique unaire

        expr_op_5
                : ET
                .

        expr_op_6
                : OU
                .
##
## FIN EXPRESSIONS
##

##
## FIN Productions
##
```

Enumération des types de nœuds de l'AST

La documentation suivante est tirée de la *Javadoc* de l'énumération Java correspondante.

Dans la documentation des symboles, expression est un terme générique regroupant l'utilisation de variable ou de constante, ou l'appel d'une fonction, ou une composition de ces opérandes avec des opérateurs arithmétiques ou booléens.

Résumé des constantes symboliques
AccèsChamp Accède à un champ dans un enregistrement.
AccèsTableau Accède à un élément dans un tableau.
BlocDInstructions Une séquence d'instructions éventuellement suivie d'un traitement d'Exception.
Booléen Contient une valeur littérale de type Booléen.
Caractère Contient une valeur littérale de type Caractère.
Chaîne Contient une valeur littérale de type Chaîne.
Compalgo Contient un programme ou une bibliothèque.
DeclConstante Déclaration d'une constante.
DeclException Utilisation directe d'une constante.
DeclFonction Déclaration d'une fonction.
DeclParamètre Déclaration d'un paramètre formel.
DeclProcédure Déclaration d'une procédure.
DeclType Déclaration d'un type.
DeclTypeEnregistrement Déclaration d'un type enregistrement.
DeclTypeTableau Déclaration d'un type tableau.
DeclVariable Déclaration d'une variable locale.
Entier Contient une valeur littérale de type Entier.
Erreur Signifie une erreur.
Exception Déclare une Exception lançable par le sous-programme.

Feuille N'est pas un nœud.
Id Contient un nom symbolique.
IdAmbiguë Contient un nom symbolique dont la nature n'a pu être déterminée précisément.
InsAffectation Instruction d'affectation.
InsAppel Instruction d'appel de sous-programme.
InsDéclencher Déclenchement d'une Exception.
InsRetourner Retour d'une fonction.
InsSi Instruction conditionnelle.
InsTantque Répétition.
LorsqueException Traitement d'une Exception particulière.
Néant N'est pas un nœud.
OpAritDiv Division réelle.
OpAritDivEnt Division entière.
OpAritModulo Modulo.
OpAritMoins Soustraction.
OpAritMult Multiplication.
OpAritPlus Addition.
OpAritUnMoins Moins unaire.
OpBoolEt Et logique.
OpBoolNon Non logique.
OpBoolOu Ou logique.
OpCompEq Prédicat "est égal à".
OpCompInf Prédicat "est strictement inférieur à".

OpCompInfEq Prédicat "est inférieur ou égal à".
OpCompNeq Prédicat "n'est pas égal à".
OpCompSup Prédicat "est strictement supérieur à".
OpCompSupEq Prédicat "est supérieur ou égal à".
Programme Contient le programme principal.
Réel Contient une valeur littérale de type Réel.
TraitementException Traitement des Exceptions en fin de (sous-)programme.
UtilConstante Utilisation directe d'une constante.
UtilType Contient le nom symbolique d'un type.
UtilVariable Utilisation directe d'une variable.

Documentation détaillée des constantes symboliques

AccèsChamp

Accède à un champ dans un enregistrement.

Arité 2.

Contient :
UtilVariable (doit évaluer à un enregistrement)
Id (le nom du champ à atteindre)

AccèsTableau

Accède à un élément dans un tableau.

Arité 2.

Contient :
UtilVariable (doit évaluer à un tableau)
expression (l'indice de l'élément à atteindre)

BlocDInstructions

Une séquence d'instructions éventuellement suivie d'un traitement d'Exception.

Arité 1 à N.

Contient :
(Ins...) +
TraitementException ?

Booléen

Contient une valeur littérale de type Booléen.

Arité 1.

Contient :
chaîne (le booléen représenté)

Caractère

Contient une valeur littérale de type Caractère.

Arité 1.

Contient :
chaîne (le caractère représenté)

Chaîne

Contient une valeur littérale de type Chaîne.

Arité 1.

Contient :
chaîne

Compalgo

Contient un programme ou une bibliothèque.

Arité 0 à N.

Contient :
Import *
(DeclType | DeclConstante | DeclException) *
DeclVariable *
BlocDinstructions ?

DeclConstante

Déclaration d'une constante.

Arité 3.

Contient :
Id
UtilType
expression

DeclException

Utilisation directe d'une constante.

Arité 1.

Contient :
Id

DeclFonction

Déclaration d'une fonction.

Arité 2 à N.

Contient :
Id
DeclParametre *
UtilType
Exception *
DeclVariable *
BlocDInstructions ?

DeclParamètre

Déclaration d'un paramètre formel.

Arité 3.

Contient :
Id (mode)
Id (nom symbolique)
UtilType

DeclProcédure

Déclaration d'une procédure.

Arité 1 à N.

Contient :
Id
DeclParametre *
DeclVariable *
BlocDInstructions?

DeclType

Déclaration d'un type.

Arité 2.

Contient :
Id
DeclTypeTableau | DeclTypeEnregistrement | UtilType

DeclTypeEnregistrement

Déclaration d'un type enregistrement.

Arité 2 à N.

Contient :
DeclVariable *

DeclTypeTableau

Déclaration d'un type tableau.

Arité 3.

Contient :
Constante | Entier (indice min)
Constante | Entier (indice max)
UtilType (type des éléments)

DeclVariable

Déclaration d'une variable locale.

Arité 2.

Contient :
Id
UtilType

Entier

Contient une valeur littérale de type Entier.

Arité 1.

Contient :
chaîne (le nombre représenté)

Erreur

Signifie une erreur.

Arité 0.

Exception

Déclare une Exception lançable par le sous-programme.

Arité 1.

Contient :
chaîne (identifiant)

Feuille

N'est pas un nœud. Est une feuille. Représente une chaîne littérale.

CE TYPE DE NOEUD N'APPARAÎT JAMAIS DANS L'ARBRE. Il indique uniquement que le fils attendu est une chaîne littérale.

Arité 0.

Id

Contient un nom symbolique.

Arité 1.

Contient :
chaîne

IdAmbiguë

Contient un nom symbolique dont la nature n'a pu être déterminée précisément. Il peut correspondre soit à une variable, soit à un appel de procédure.

Arité 1.

Contient :
Id

InsAffectation

Instruction d'affectation.

Arité 2.

Contient :
UtilVariable
expression

InsAppel

Instruction d'appel de sous-programme.

Arité 1 à N.

Contient :
Id
expression * (une pour chaque paramètre)

InsDéclencher

Déclenchement d'une Exception.

Arité 1.

Contient :
Id

InsRetourner

Retour d'une fonction.

Arité 1.

Contient :
expression

InsSi

Instruction conditionnelle.

Arité 2 à 3.

Contient :
expression (condition)
BlocDInstruction + (bloc si, bloc sinon optionnel)

InsTantque

Répétition.

Arité 2.

Contient :
expression (condition)
BlocDInstruction

LorsqueException

Traitement d'une Exception particulière.

Arité 2 à N.

Contient :
Id
Instruction +

Néant

N'est pas un noeud. Constante symbolique purement abstraite.

OpAritDiv

Division réelle.

Arité 2.

Contient :
expression
expression

OpAritDivEnt

Division entière.

Arité 2.

Contient :
expression
expression

OpAritModulo

Modulo.

Arité 2.

Contient :
expression
expression

OpAritMoins

Soustraction.

Arité 2.

Contient :
expression
expression

OpAritMult

Multiplication.

Arité 2.

Contient :
expression
expression

OpAritPlus

Addition.

Arité 2.

Contient :
expression
expression

OpAritUnMoins

Moins unaire.

Arité 1.

Contient :
expression

OpBoolEt

Et logique.

Arité 2.

Contient :
expression
expression

OpBoolNon

Non logique.

Arité 1.

Contient :
expression

OpBoolOu

Ou logique.

Arité 2.

Contient :
expression
expression

OpCompEq

Prédicat "est égal à".

Arité 2.

Contient :
expression
expression

OpCompInf

Prédicat "est strictement inférieur à".

Arité 2.

Contient :
expression
expression

OpCompInfEq

Prédicat "est inférieur ou égal à".

Arité 2.

Contient :
expression
expression

OpCompNeq

Prédicat "n'est pas égal à".

Arité 2.

Contient :
expression
expression

OpCompSup

Prédicat "est strictement supérieur à".

Arité 2.

Contient :
expression
expression

OpCompSupEq

Prédicat "est supérieur ou égal à".

Arité 2.

Contient :
expression
expression

Programme

Contient le programme principal.

Arité 2 à N.

Contient :
Id
DeclVariable *
BlocDInstructions ?

Réel

Contient une valeur littérale de type Réel.

Arité 1.

Contient :
chaîne (le nombre représenté)

TraitementException

Traitement des Exceptions en fin de (sous-)programme.

Arité 1 à N.

Contient :
LorsqueException +

UtilConstante

Utilisation directe d'une constante.

Arité 1.

Contient :
chaîne

UtilType

Contient le nom symbolique d'un type.

Arité 1.

Contient :
chaîne

UtilVariable

Utilisation directe d'une variable.

Arité 1.

Contient :
Id | AccèsTableau | AccèsChamp

Rapport d'analyse

Auteur : Leroux Damien
Date : 21/07/2007

1. Introduction

Préalablement à l'élaboration de la solution logicielle implantant les évolutions du langage algorithmique et de son environnement (Compalgo), nous avons déjà spécifié le paradigme d'analyse du problème à résoudre et de conception de sa solution logicielle. Le client a spécifié que la grammaire du langage doit être de nature LL(1) et a décidé de l'outil générateur de *parser* à utiliser, Coco/R, qui utilise l'algorithme dit de descente récursive. L'ensemble de la chaîne logicielle Compalgo v2.0 utilisera exclusivement la plateforme Java.

Après un bref rappel sur les grammaires LL et la descente récursive, ce document définit le domaine d'analyse et présente les orientations globales qui président à la conception, en tenant compte du paradigme de la descente récursive dans une grammaire LL(1).

Nous distinguerons une chaîne de composants distincts, et pour chaque composant nous spécifierons les moyens d'implanter la solution logicielle correspondante.

2. Table des matières

3. Domaine d'analyse et problématique associée

Une grammaire est un ensemble de règles de construction et d'éléments terminaux comme les mots-clé et les ponctuations. Dans la théorie des langages, les grammaires sont classées en fonction de leurs propriétés. Les grammaires de la classe dite LL engendrent des langages lisibles de gauche à droite (le premier L, *Left to right*) et structurables de gauche à droite (le deuxième L, *Leftmost derivation*). Ces langages sont aisément *parsés* par l'algorithme de descente récursive. La notation LL(k), k étant un entier défini ou non, indique qu'il est nécessaire de lire au plus k *tokens* pour pouvoir décider de la structure du texte source lu (c'est-à-dire décider des règles de grammaire à appliquer). En particulier, la nature LL(1) d'une grammaire implique qu'à chaque nouvel élément du langage lu, le *parser* sait quelle règle s'applique.

La grammaire du langage algorithmique est réécrite en respectant le standard BNF, et de façon à obtenir la nature LL(1). Cette grammaire est ensuite portée dans un fichier de spécification de *parser* selon le format du générateur Coco/R, qui est en charge de créer le code concret du *parser* reconnaissant la grammaire stipulée.

Dans ce fichier de spécification, outre les règles définissant la forme du langage, on incorpore des actions qui permettent de créer la sortie du *parser*. Cette sortie pourrait être une réécriture du code, par exemple une réécriture du langage algorithmique dans le langage Java, une compilation immédiate, un Arbre de Syntaxe Abstraite, ou une interprétation directe du texte source...

Dans la suite, la volonté est de simplifier au maximum le code de ces actions intégrées à la grammaire, afin de préserver la clarté du fichier de spécification, et de faciliter les évolutions et *refactorings* futurs.

La première version de Compalgo fournit un éditeur et un compilateur intégrés. Le compilateur traduit le langage algorithmique en langage Java, puis invoque le compilateur Java. La grammaire actuelle souffre de quelques défauts, comme le non-respect de la précédence des opérateurs arithmétiques, et cette architecture de traduction ne fournit aucun cadre de débogage des programmes écrits en langage algorithmique.

La solution logicielle Compalgo v2.0 doit, sur la base de la grammaire et du fichier de spécification de *parser* de Coco/R déjà définis, implanter un compilateur du langage algorithmique, un interpréteur des programmes compilés, et à terme une interface de débogage des programmes compilés. Le compilateur et le débogueur devront être intégrés à l'environnement existant. Le client souhaite également que la bibliothèque du langage qui sera proposée aux étudiants soit facilement extensible au moyen de classes Java. Aussi, Compalgo v2.0 devra gérer des fichiers de spécification (fichiers à l'extension `.spec` dans lesquels sont autorisées les déclarations de types, de constantes, et d'entêtes de sous-programmes uniquement) et les fichiers d'implantation dits algorithmiques, à l'extension `.algo`.

La chaîne logicielle reliant un texte source à sa représentation compilée peut prendre diverses formes, ainsi que cette représentation compilée. Le client a décidé d'abstraire la sortie du compilateur en lui faisant générer un Arbre de Syntaxe Abstraite (par commodité nous utiliserons par la suite l'acronyme anglais AST, plus habituel que son pendant français ASA). Ce choix de l'abstraction permet de dissocier totalement la reconnaissance et la structuration de l'information contenue dans le texte source de l'utilisation informatique de cette information. Cela rend la solution logicielle modulaire et en favorise l'évolutivité.

4. Nous avons ainsi à définir l'implantation d'un AST, sa construction dans le paradigme de la descente récursive, et les moyens de pêcher l'information dans un AST bien formé. Les vérifications sémantiques s'opèrent sur l'arbre construit par le *parser*. Seuls les

arbres sémantiquement valides sont transmis au maillon suivant de la chaîne proposé par le client, l'interpréteur. Cet interpréteur doit être capable d'exécuter le programme représenté par l'AST fourni par le *parser*. À terme, cette exécution devra être contrôlée par une interface de débogage, et le contexte d'exécution intégralement interrogeable par celle-ci.AST

Le *parser* de Compalgo V2 doit générer un Arbre de Syntaxe Abstraite. Cet arbre est une structure récursive représentant de façon fortement structurée l'information contenue dans le texte *parsé*, et est quasiment identifiable à une description XML. Cette représentation est très pratique pour l'ensemble des opérations et évaluations nécessaires pour, dans un premier temps, vérifier la validité sémantique du texte *parsé*, et dans un deuxième temps d'interpréter le programme en langage algorithmique décrit dans l'arbre.

Après avoir décrit la structure et la construction d'un AST au cours d'une descente récursive, nous nous penchons sur les ambiguïtés sémantiques du langage algorithmique, puis nous décrivons le mécanisme de parcours d'un AST qui permet d'implanter l'ensemble des fonctionnalités nécessaires.

Structure

Chaque élément de l'AST est **typé**, et peut contenir soit d'autres éléments (on l'appelle un nœud), soit un simple label (on l'appelle une feuille). Le langage algorithmique permet de spécifier des imports, des types, des constantes typées, des sous-programmes (procédures et fonctions), et le programme principal. Dans chaque sous-programme et dans le programme principal, on peut trouver des déclarations de variables typées, suivies d'un bloc d'instructions. Le langage algorith-mique définit six instructions élémentaires : l'affectation, l'appel de sous-programme, l'exécution conditionnelle « si … alors … sinon … », la répétition « tantque … faire … fin tantque», le retour d'une valeur dans une fonction, et le déclenchement d'une exception nommée. Les exceptions sont traitées à la fin d'un bloc d'instructions (de sous-programme ou de programme principal).

Tous les concepts (mots-clé) énumérés dans le paragraphe précédent correspondent chacun à un type de nœud particulier. S'ajoutent à cette liste les opérateurs arithmétiques, les opérateurs de comparaison, ainsi que les opérateurs de la logique booléenne, les spécifications d'identifiants (type, constante, variable ou sous-programme...).

La totalité des types de nœuds, ainsi que leur structure, est énumérée en annexe.

Au total, l'énumération des types de nœuds dépasse la quarantaine d'éléments. Cette masse d'informations nécessite de définir, outre cette énumération, des moyens supplémentaires pour s'assurer de la bonne structure de l'arbre pendant sa construction. En effet, lors d'un traitement récursif tel que la descente opérée par le *parser*, les erreurs se propagent très vite, et il convient d'implanter des garde-fous pour détecter rapidement les erreurs de construction. Ces garde-fous doivent en particulier aider à la phase de développement de la construction de l'AST par le *parser* au moyen des actions sémantiques intégrées dans la grammaire, et par suite à la maintenance et l'évolution du *parser*.

D'une façon comparable à une DTD XML, chaque type de nœud stipule la liste des types de fils qu'il accepte en séquence (par exemple, une déclaration de constante comporte dans l'ordre un identifiant, un type, et une valeur, et uniquement cette combinaison). Cette information supplémentaire, assortie de la vérification correspondante avant chaque

ajout d'un nœud dans un autre, nous assure que l'AST à l'issue du processus sera bien formé.Construction dans le paradigme de la descente récursive

A priori, la descente récursive et la forme d'une grammaire LL correspondent parfaitement à la représentation récursive sous forme d'arbre. La construction de l'AST doit donc être réalisée simplement, sur la base du découpage conceptuel préalablement effectué lors de la rédaction de la grammaire. Typiquement, chaque entrée dans une règle délimitant une entité sémantique correspond à la création d'un nœud doté de ses fils.

Par exemple, la déclaration d'une constante « `constante K <Entier> = 3;` » engendre le noeud :

```
[ DeclConstante
   [ Id [ "K" ] ]
   [ UtilType [ "Entier" ] ]
   [ Entier  [ "3" ] ]
]
```

Nous pouvons implanter une telle construction en utilisant une pile de nœuds, pour la réentrance récursive, et en définissant trois opérations : « Ouvrir » un nouveau nœud, qui crée un nœud au sommet de la pile. « Fermer » le nœud courant, qui dépile un nœud et l'ajoute au sommet de la pile. « Feuille », qui crée un nœud et l'ajoute au nœud courant, synonyme d' « Ouvrir » suivi de « Fermer ». En reprenant l'exemple de la déclaration de la constante K, on obtiendrait la séquence suivante, en posant en colonne gauche le *token* courant, au centre les actions effectuées (écrites en pseudo code) et en colonne droite le nœud courant résultant des actions :

…		[…]
constante	Ouvrir("DeclConstante")	[DeclConstante]
K	Ouvrir("Id") Feuille("K")	[**Id** [K]]
	Fermer()	[DeclConstante [Id [K]]]
<	Ouvrir("UtilType")	[UtilType]
Entier	Feuille("Entier")	[**UtilType** [Entier]]
>	Fermer()	[**DeclConstante** [**Id** [K]] [**UtilType** [Entier]]]
=		
3	Ouvrir("Entier") Feuille("3")	[**Entier** [3]]
	Fermer()	[**DeclConstante** [**Id** [K]] [**UtilType** [Entier]] [**Entier** [3]]]

;	Fermer()	[… [**DeclConstante** [**Id** [K]] [**UtilType** [Entier]] [**Entier** [3]]]]

Ambiguïtés syntaxiques et sémantiques

Cette vision de la construction est valide pour toutes les formes dont le début et la fin sont décidables immédiatement, comme la déclaration de constante, préfixée par le mot-clé « constante », et suffixée par un point-virgule (qui plus est, la déclaration d'une constante est de longueur fixe, en termes d'entités sémantiques). Il en va autrement pour certaines formes engendrées par l'écriture LL(1) de la grammaire.

Ainsi, l'affectation « a <- b; » n'est pas distinguable de l'appel de sous-programme « a(b); » dès le premier *token*, « a ». L'ambiguïté sémantique ne peut être résolue qu'en avançant au *token* suivant, « <- » ou « (» respectivement. Ceci implique de tenir compte d'un *token* (c'est-à-dire créer le nœud lui correspondant, ici [Id [a]]) qui devra devenir le fils d'un autre nœud qui reste encore à créer.

Ceci nous oblige à définir une troisième opération, qui sera dénommée « Échanger ». Échanger enlève le dernier nœud du sommet de la pile, crée le nouveau nœud père sur le sommet de la pile, et y ajoute l'ancien sommet qu'il vient de dépiler.

De même les expressions arithmétiques et booléennes, décrites dans une grammaire LL(1), nécessitent de prendre en compte un *token* avant de pouvoir décider du type de nœud englobant. Voici pour illustrer l'opération Échanger la création d'un nœud représentant l'addition de deux entiers, par exemple « 3+1 » :

3	Ouvrir("Entier") Feuille("3")	[**Entier** [3]]
+	Échanger ("Addition")	[Addition [Entier [3]]]
1	Ouvrir("Entier") Feuille("3")	[**Entier** [1]]
	Fermer()	[**Addition** [**Entier** [3]] [**Entier** [1]]]

En outre, il existe dans la grammaire de Compalgo une forme indécidable syntaxiquement. Dans une expression, un identifiant seul peut être un appel de fonction sans paramètre ou l'utilisation d'une variable locale, ou d'un paramètre formel. La caractérisation de l'identifiant par l'accès à un élément de tableau, par l'accès à un champ d'enregistrement, ou par une liste d'expressions parenthésée est optionnelle, et l'on ne peut pas décider du type de nœud à construire sur la seule base de la syntaxe. Le but de la création de l'AST étant de créer une abstraction préalable à la vérification sémantique, il ne convient pas d'incorporer de vérification sémantique au milieu des actions de construction de l'AST. Nous utiliserons un type de nœud temporaire pour signifier cette ambiguïté qui devra être résolue lors des vérifications sémantiques postérieures à la création de l'AST.Paradigme d'arpentage de l'AST (en anglais, Tree Walking).

Concernant les parcours dans l'arbre, nous avons choisi le verbe arpenter et ses dérivations (arpentage, arpenteur) pour traduire le verbe anglais consacré *to walk*.

Il est nécessaire d'arpenter l'arbre pour toute évaluation de ou action sur un nœud de l'arbre. Nous définissons ici un moyen générique d'arpentage qui permettra l'implantation de toutes les actions et évaluations nécessaires à la vérification sémantique stricte du texte source, ainsi que plus tard à l'interprétation du programme *parsé*.

Selon les types d'actions ou d'évaluations à effectuer, le parcours dans l'arbre peut être linéaire, verticalement (de père à fils ou de fils à père) ou horizontalement (dans un père, on parcourt tous les fils), une recherche systématique en profondeur, ou encore procéder par sauts de nœud à nœud (pour un nœud donné, on voudra par exemple sauter à la déclaration correspondante), sans progression directe dans les chemins.

Le type et la structure du parcours sont étroitement liés au but du parcours, et à la structure de l'arbre. Néanmoins, tout parcours est une composition d'un petit nombre de déplacements élémentaires. On peut réduire cet ensemble de déplacements élémentaires à :

entrer dans un nœud (se positionner sur le premier fils),
sortir d'un nœud (aller ou revenir dans le nœud père),
aller au frère suivant du nœud courant (fait sens seulement après être entré dans un nœud),
sauter à un nœud donné,

Il convient d'ajouter à cet ensemble minimal un marqueur de fin correcte et un marqueur de fin avec erreur.

Pour un point de départ donné, ces six déplacements élémentaires enchaînés en séquence définissent un parcours dans l'arbre. L'arpentage intervenant pendant et après la construction de l'arbre, il est nécessaire de tenir compte de la pile du constructeur d'AST. En effet, tant qu'un nœud n'a pas été ajouté à son futur père, le nœud courant au sommet de la pile, il ne connaît pas l'identité de son futur père, et de ce fait la direction sortie déboucherait sur le néant. Il convient donc de considérer la pile comme l'ascendance latente du nœud en cours de construction. De ce fait, la direction « sortie » branche soit sur le père direct du nœud dont on sort, soit sur le sommet de la pile, qui correspondra à terme au père du nœud dont on sort.

L'Arpenteur générique est ainsi capable de se mouvoir dans l'arbre, mais ne définit lui-même aucun type de parcours. Il nécessite pour cela l'adjonction d'un Pilote, qui correspond à une évolution du patron classique dit du Visiteur. Un Pilote a pour rôle de contrôler le parcours de l'Arpenteur, en lui indiquant à chaque nœud la direction à prendre ensuite.

Le mécanisme est le suivant : lors du démarrage d'un arpentage avec un Pilote donné, l'Arpenteur interroge le Pilote pour connaître le point de départ requis, et se positionne dessus. L'Arpenteur présente son nœud courant au Pilote, qui effectue une action spécifique au type du nœud courant et retourne à l'Arpenteur la direction à prendre ensuite. L'arpenteur se positionne alors en fonction de la réponse du Pilote sur un nouveau nœud courant, et interroge à nouveau le Pilote, ce processus se répétant jusqu'à ce que le Pilote signifie à l'Arpenteur la fin correcte ou non de l'arpentage.

5. Ce mécanisme permet d'implanter toutes les fonctionnalités nécessaires comme nous le verrons par la suite, l'ensemble des pilotes spécialisés étant décrits dans le document de conception.Vérifications sémantiques

Concernant les vérifications sémantiques, il serait possible de créer des tables de symboles au cours de la construction de l'arbre pour gérer l'existence et l'unicité des diverses déclarations. Néanmoins, cette information est déjà contenue dans l'arbre, et la création des tables apparaît comme redondante, donc inutile, notre environnement n'étant pas un environnement bruité et sujet à l'apparition d'erreurs aléatoires. De plus, le paradigme d'arpentage de l'arbre étant suffisamment abstrait et générique, il convient amplement pour effectuer toutes les recherches et vérifications.

En outre, l'adjonction de tables de symboles ferait courir le risque d'une pollution sensible du fichier de spécification de *parser* du générateur Coco/R, l'orientation adoptée depuis le départ étant *a contrario* une simplification extrême des actions sémantiques à mêler à la grammaire, à des fins de clarté du fichier de spécification.

Il convient de remarquer à ce stade que toute vérification a lieu lorsqu'on valide (termine) la construction d'un nœud, et juste avant son incorporation dans le nœud père. Ainsi, l'on peut encapsuler l'action de vérification dans l'action de construction "Fermer", qui correspond à ladite validation. Lorsqu'un nœud est fermé, il est soumis à vérification, et s'il est légal, sera ajouté dans son père.

La validation sémantique de l'ensemble des types de nœud peut être implantée elle aussi au moyen d'un Pilote d'Arpenteur, regroupant de façon pratique toutes ces tâches dans un seul point d'entrée, et permettant d'effectuer la vérification « au vol » sans polluer aucunement le fichier de spécification du *parser*.

Nombreuses sont les vérifications à effectuer, dont voici une liste exhaustive.

Tout symbole utilisé (variable, type, constante, sous-programme) doit être préalablement défini. Ce type de vérification est trivial pendant la construction de l'arbre : par définition, une déclaration préalable existe déjà dans l'arbre (ou dans la pile du constructeur de l'arbre) lorsqu'on l'interroge.

Toute déclaration de type et de constante doit être unique. Tout identifiant de variable doit être unique dans le contexte de la déclaration (programme principal ou sous-programme). À ce titre, les paramètres formels sont considérés comme des variables locales, et leurs identifiants ne doivent pas masquer d’identifiants de variables locales.

Les exceptions ne sont pas déclarées au niveau global, mais au niveau de chaque sous-programme pouvant en déclencher. Chaque sous-programme pouvant déclencher ou propager des exceptions qu'il ne traite pas lui-même doit les stipuler dans son entête. Le programme principal ne peut déclencher d'exception qu'il ne traite pas lui-même (en outre, le programme principal n'a pas d'entête). Tout sous-programme appelant d'autres sous-programmes pouvant déclencher des exceptions est susceptible de propager celles-ci s'il ne les traite pas explicitement, et de ce fait, doit les déclarer dans son entête. Réciproquement, pour détecter d'éventuelles fautes de frappe dans la dénomination des exceptions, il conviendra de vérifier que chaque exception traitée est susceptible d'être déclenchée dans le contexte correspondant.

Compalgo définit trois modes de passage des paramètres des sous-programmes. Respectivement, *entrée*, *sortie*, et *màj* (mise à jour) correspondent à un passage en lecture seule, en écriture seule, et en lecture et écriture. Les paramètres dont l'écriture est autorisée doivent corres-pondre à des variables locales du contexte de l'appelant (ou aux paramètres formels de ce contexte). Les sous-programmes de type fonction n'acceptent que des arguments en *entrée*, les deux autres modes de passage sont interdits.

Concernant les déclarations de sous-programmes, la vérification est plus fine. On peut spécifier séparément entêtes et corps de sous-programmes. Une unique déclaration d'entête seul est autorisée pour chaque sous-programme, ainsi qu'une unique déclaration de corps de ce même sous-programme. La déclaration de l'entête est facultative. Il conviendra d'intégrer dans l'arbre le corps du sous-programme à la déclaration d'entête afin de simplifier la structure de l'arbre, et d'éviter les ambiguïtés. La surcharge est autorisée ; le même nom de sous-programme peut apparaître plusieurs fois avec des listes d'arguments différents. Il apparaît que la déclaration des exceptions potentiellement déclenchables ne peut faire partie de la signature des sous-programmes, il est facile de construire un cas où l'ambiguïté du choix du sous-programme à appeler ne peut être levée. De même, l'inclusion des modes de passage des paramètres formels dans la signature des sous-programmes génère des ambiguïtés indécidables.

La définition d'une procédure et d'une fonction de même nom et de mêmes arguments ne génère aucune ambiguïté : les appels de procédures ne peuvent se trouver que dans une instruction d'appel de sous-programme, hors expression, et les appels de fonctions ne peuvent se trouver que dans les expressions évaluables (partie droite d'une affectation, expression conditionnelle, arguments effectifs d'un appel de sous-programme).

Il découle de tout ceci que la signature d'un sous-programme est composée de son type, de son type de retour pour les fonctions, et de la séquence des types de ses paramètres formels.

Dans les expressions arithmétiques, il existe plusieurs stratégies possibles quant à la gestion des valeurs entières et réelles. Le comportement classique dans la plupart des langages est d'autoriser le transtypage implicite d'entier vers réel, et d'interdire la réciproque, de réel vers entier. Le langage algorithmique ayant une vocation purement pédagogique, le client a décidé d'interdire tout transtypage, et de ce fait, de mêler entiers et réels dans tout expression. Ainsi, tout opérateur binaire doit avoir les opérandes gauche et droit de même type. En cela les opérateurs arithmétiques rejoignent les opérateurs booléens, qui n'acceptent forcément que des opérandes de type booléen.

Le langage algorithmique définit deux types de fichiers source : des fichiers de spécification contenant des déclarations de types, constantes, et entêtes de sous-programmes, et des fichiers algorithmiques, contenant, outre ces mêmes déclarations, les corps des sous-programme, ainsi qu'un éventuel programme principal.

Les corps de sous-programme, ainsi que le programme principal, sont interdits dans les fichiers de spécification. Les fichiers algorithmiques importés ne doivent pas non plus déclarer de programme principal. Seul le fichier algorithmique "importateur" a le droit de déclarer le programme principal.

L'instruction retourner() est légale dans les fonctions, et illégale dans les procédures et le programme principal. Il conviendra de vérifier la présence dans chaque corps de fonction d'une occurrence de l'instruction retourner() par chemin, et une seule. Un chemin correspond à une séquence d'instructions, la condition "si ... alors ... fin si" et la répétition "tantque ... faire ... fin tantque" définissant un chemin optionnel, et la condition "si ... alors ... sinon ... fin si" définissant deux chemins alternatifs. Il n'est pas prévu de tester statiquement la valeur de vérité des expressions conditionnelles, pour s'assurer du passage par un chemin optionnel. Dans l'optique pédagogique du langage algorithmique, l'écriture de l'instruction retourner() dans un chemin optionnel sera de toutes façons présentée comme étant pour le moins mauvais style, sinon interdite, et pourra éventuellement constituer un avertissement ou une erreur de compilation.

6. Interprétation

À partir d'un AST, il est possible de choisir sans contrainte la forme finale des programmes pour leur exécution. Le client a proposé d'interpréter directement le programme dans sa forme arborescente produite par le *parser*.

L'exécution d'un programme suppose qu'il est bien formé, c'est-à-dire que sa syntaxe est correcte et qu'aucune vérification sémantique n'a généré d'erreur. Dans ce contexte idéal, l'information contenue dans l'AST permet la simulation de l'exécution complète d'un programme. Il est nécessaire pour mener à bien l'exécution de définir l'environnement qui permettra de stocker et manipuler les données traitées par le programme (variables, paramètres, constantes), ainsi que de gérer l'appel de sous-programmes.

Du point de vue de l'exécution, le programme principal est équivalent à une procédure sans paramètre et ne pouvant déclencher aucune exception. L'exécution du programme principal est équivalente à un appel de procédure.

Nous pouvons découper le processus global d'interprétation en plusieurs sous-parties spécialisées :

Le point d'entrée global de l'interprétation (à l'intérieur du nœud racine de type Compalgo) qui crée les données correspondant aux constantes globales, et « appelle » le programme principal.

L'appel de sous-programme (et du programme principal), qui doit créer le dictionnaire des variables locales et paramètres formels définis, exécuter le bloc d'instructions du sous-programme, et éventuellement traiter les exceptions qui auront été déclenchées si le sous-programme comporte une clause `traite-exception`. L'appel de sous-programme est une opération réentrante et doit de ce fait être gérée à part, en tant que telle.

L'exécution d'une séquence d'instructions, dans le bloc principal d'un sous-programme comme dans un sous-bloc d'instructions à l'intérieur d'une instruction `si` ou `tantque`. L'exécution d'une séquence est aussi une opération réentrante.

L'évaluation d'une expression doit valoriser les expressions booléennes et arithmétiques et est aussi une opération réentrante.

La résolution des accès aux éléments des variables de types composés (tableaux et enregistrements), également réentrante, est utilisable lors de la valorisation d'une expression.

L'arpentage piloté précédemment défini en 4.D nous permet d'implanter ces parties spécialisées.

L'interprétation devra s'effectuer, comme le reste des composants de la chaîne logicielle Compalgo v2.0, sur la plateforme Java.

Compalgo définit quatre types de données primaires, qui correspondent chacun à un type de donnée Java : <Caractère> (*Character*), <Chaîne> (*String*), <Entier> (*Integer*), <Réel> (*Float*). Il est nécessaire d'implanter un mécanisme permettant le passage de données bidirectionnel entre Java et Compalgo. En outre, Compalgo définit des exceptions, qui seront encapsulées dans les exceptions Java.

Il est nécessaire de définir un système de dictionnaire de données qui fera la liaison entre un nom symbolique (identifiant) du langage algorithmique et la donnée concrète qu'il représente (variable, constante, valeur intermédiaire), évaluable par Java. Un dictionnaire de constantes est nécessaire au niveau global, à la charge du point d'entrée principal de l'interprétation. Un dictionnaire de variables locales est nécessaire dans chaque sous-programme appelé, à la charge de l'interpréteur spécialisé des sous-programmes. Les paramètres formels pourront être confondus avec les variables et être contenus dans le même dictionnaire. La validité des accès en lecture et écriture aux paramètres formels a été préalablement vérifiée lors de la phase de vérifications sémantiques, en sortie du *parser*. L'interpréteur n'a pas à revérifier la validité de l'information qu'il traite.

Pour évaluer les expressions et passer les paramètres effectifs aux sous-programmes, le moyen le plus naturel est d'utiliser une pile de données. Les opérateurs arithmétiques et booléens consomment une ou deux données de la pile selon leur arité et produisent une donnée. Lors de l'appel d'un sous-programme, les déclarations d'arguments formels sont traitées de gauche à droite, l'évaluation des paramètres effectifs doit donc se faire de droite à gauche.

7. Intégration dans l'interface existante

L'utilisateur final de Compalgo est l'étudiant qui s'initie à la programmation impérative. Un environnement de développement a été réalisé dès la première version de Compalgo pour rendre cette initiation confortable pendant les séances de travaux pratiques. Dans sa version actuelle, il fournit un éditeur multi-fichiers dédié sachant mettre en forme le texte des fichiers algorithmiques, un bouton (marqué 1 sur la capture d'écran ci-dessous) permettant de déclencher la compilation du fichier en cours d'édition, et une console affichant les erreurs de compilation (marquée 2 sur la capture d'écran ci-dessous).

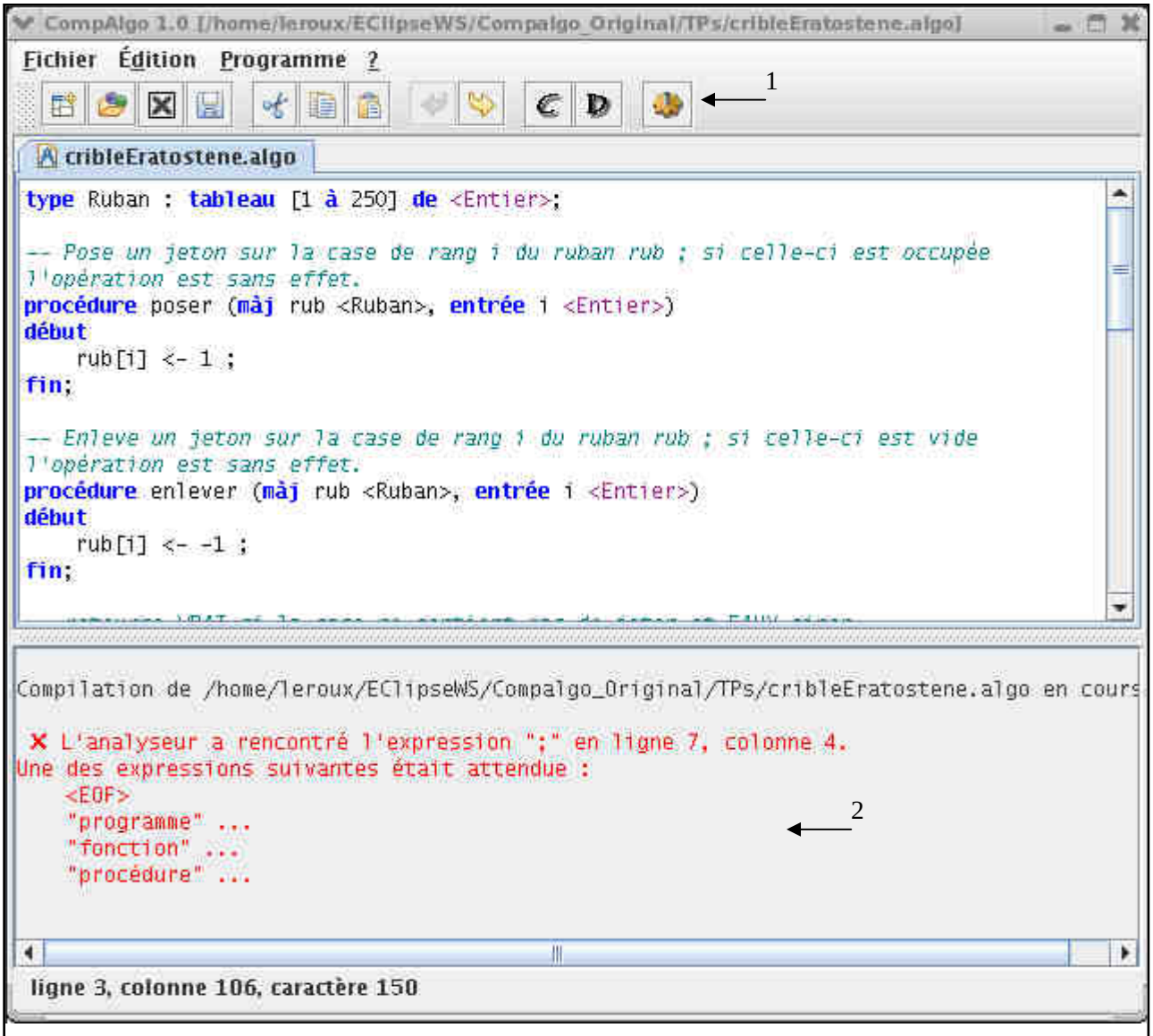

Illustration 1: Capture d'écran de la version 1 de l'environnement Compalgo

La nouvelle version du compilateur remplace la précédente de façon transparente pour l'utilisateur. Le processus d'édition et de compilation, vu par l'utilisateur, est inchangé. Le même bouton déclenchera simplement le nouveau compilateur. Il n'est pas actuellement prévu de sauvegarder le résultat de la compilation pour une réutilisation ultérieure, aussi l'environnement Compalgo devra lui-même stocker en mémoire les résultats de compilations.

La console d'erreurs apparaît être peu ergonomique, aussi nous proposons de la remplacer par une table listant les divers messages d'erreur et d'avertissement. Un clic sur un message dans cette table téléportera l'édition vers le fichier concerné à la ligne et la colonne concernées par le message.

L'exécution de programmes algorithmiques dans la première version se déroulait dans un terminal, au moyen d'un script *shell*. Ceci impliquait d'avoir une sauvegarde sur disque du programme compilé que l'on voulait exécuter. Deux options sont possibles pour la nouvelle version, soit que l'on sauvegarde l'AST pour qu'il puisse être traité (exécuté) par le même script, soit que l'on déclenche l'exécution dans l'environnement même de Compalgo. La première option impliquerait d'implanter la sauvegarde et le chargement d'un AST donné, ou de lancer l'exécution dans le terminal d'où a été lancé l'environnement Compalgo, et la seconde option impliquerait d'implanter une console d'exécution dans l'environnement Compalgo.

Si nous arrivons au but secondaire de contrôle de l'exécution et de l'interface associée, l'implantation d'une console d'exécution fait sens, pour être intégrée à l'interface de contrôle de l'exécution et de visualisation du contexte de cette exécution. L'ergonomie de l'interface de contrôle s'en trouvera renforcée.

8. Contrôle de l'exécution

Le contrôle de l'exécution suppose un contexte logiciel d'exécution indépendant du contexte logiciel de l'interface graphique. En effet, il est impératif de pouvoir agir sur l'interface indépendamment de l'exécution des programmes compilés. Typiquement, les lectures dans la console d'exécution (qu'elle soit le terminal d'où est lancé l'environnement Compalgo ou une console virtuelle intégrée à l'interface graphique) sont des opérations bloquantes, il est inconcevable de mêler leur exécution à celle de l'interface graphique. Le langage algorithmique étant destiné à des débutants, nous pouvons nous attendre à de nombreux cas de boucles infinies ou d'erreurs violentes d'exécution. Tout cela peut générer des blocages éventuellement irrécupérables, voire casser l'exécution de l'interface graphique.

Les fonctionnalités attendues sont de pouvoir exécuter un programme, de s'arrêter volontairement à un endroit donné du source (point d'arrêt), de s'arrêter sur une erreur d'exécution en pouvant scruter le contexte d'exécution du dit programme, et d'exécuter pas à pas tout ou partie d'un programme.

Éventuellement, le fait de pouvoir revenir en arrière dans l'exécution peut être intéressant, mais ne constitue qu'un besoin très secondaire. Nous reviendrons sur cette fonctionnalité plus loin.

Ainsi, nous considérons l'exécution des programmes compilés dans leur propre fil d'exécution (*thread*). Cette section décrit le mécanisme faisant le pont entre l'exécution proprement dite et son interface de contrôle, qui ont chacune un contexte logiciel indépendant.

Le contrôle de l'exécution repose sur un principe simple : rien ne se passe sans l'accord explicite ou implicite de l'utilisateur. En d'autres termes, la machine doit attendre les décisions de l'utilisateur avant toute action, et tenir compte de ces décisions.

Une action de la machine est l'exécution d'une instruction du langage, ou l'évaluation d'une expression. Il peut être intéressant pour l'étudiant néophyte de voir les étapes de l'évaluation d'une expression complexe avant par exemple son affectation dans une variable.

Nous appelons Moteur la partie en charge de l'exécution proprement dite et Client le composant représentant l'utilisateur (le composant à travers lequel l'utilisateur agit). Comme énoncé précédemment, le Moteur attend un feu vert explicite lorsqu'il veut exécuter quelque chose. Cette attente bloque sa propre exécution. Lorsqu'il reçoit un feu vert, il réalise son exécution puis envoie une notification comme quoi l'exécution s'est terminée. Du côté du Client, le processus est inversé. Le

Client, lorsqu'il envoie un feu vert pour exécuter un pas, attend jusqu'à recevoir la notification de fin du pas.

Pour gérer cet entrelacs d'exécutions et d'attentes, nous devons définir un Contrôleur, qui se retrouve à cheval sur les deux fils d'exécution du Client et du Moteur. Le Client comme le Moteur s'adressent au Contrôleur pour interagir. Le Contrôleur reçoit la demande de feu vert de la part du Moteur, et le fait patienter jusqu'à recevoir ce feu vert du Client. Le Moteur envoie au Contrôleur la notification de fin d'exécution, et celui-ci la retransmet au Client qui peut alors reprendre son exécution normale.

Avec ce mécanisme, l'exécution contrôlable est réentrante : une exécution peut contenir d'autres exécutions également contrôlables.

Pour une gestion fine du contrôle de l'exécution, il est nécessaire que le Moteur fournisse au Contrôleur le niveau de réentrance (noté N) et une référence sur l'instruction (terme impropre) à exécuter (notée I). La connaissance du niveau de réentrance permet d'implanter les modes pas-à-pas « par-dessus » (*Step over*) et « en-dedans » (*Step into*). La connaissance de l'instruction à exécuter permet d'implanter des filtres plus fin, comme des points d'arrêt arbitraires (*i.e.* « à telle ligne dans tel fichier ») et systématiques (*e.g.* « à chaque entrée dans un sous-programme »).

Le diagramme de séquence suivant illustre et récapitule les interactions entre les trois composants de ce mécanisme lors d'une exécution contrôlée.

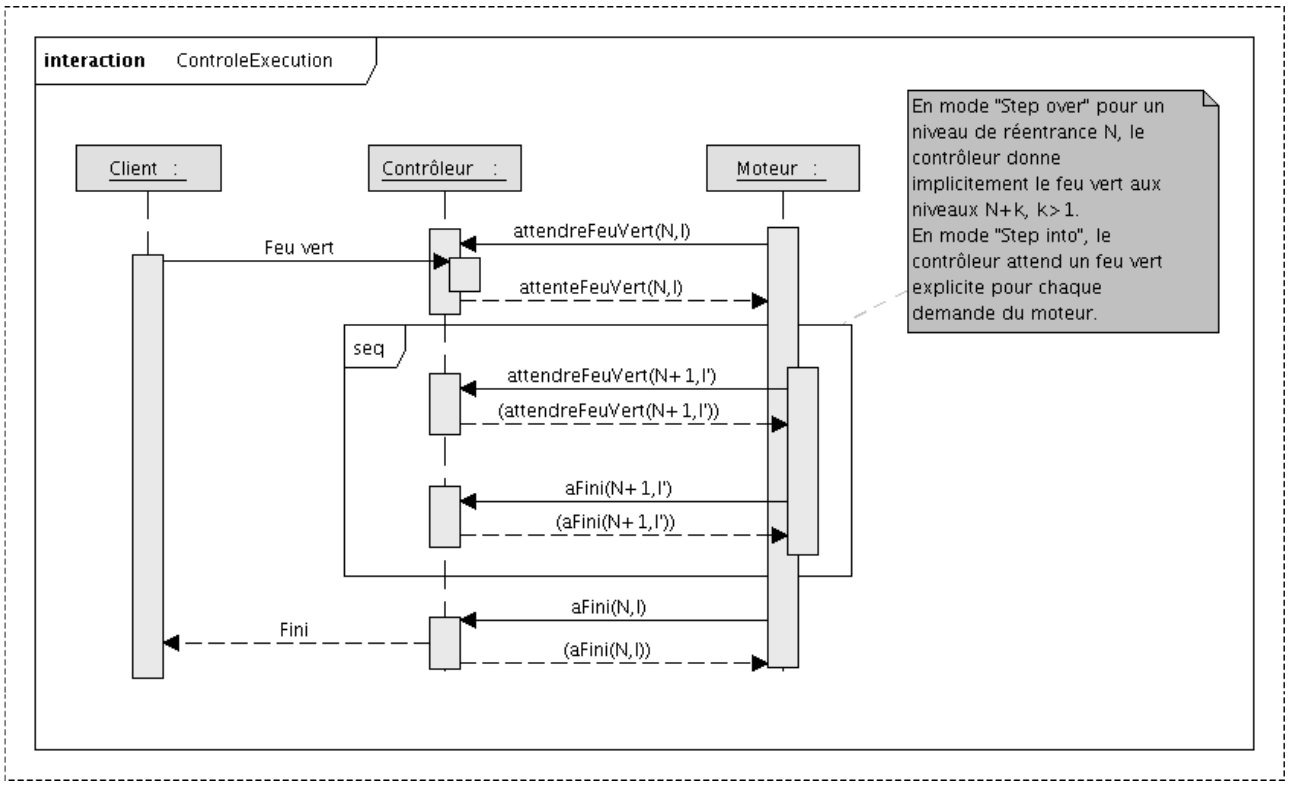

Illustration 2: Diagramme de séquence du contrôle de l'exécution

9. Interface graphique pour le contrôle de l'exécution

Le contrôle de l'exécution par l'utilisateur implique un certain nombre de fonctionnalités. L'utilisateur souhaite pouvoir déclencher et interrompre une exécution à tout moment, moyennant un programme compilé, visualiser l'état du contexte d'exécution, visualiser le « pointeur d'instruction », et poser des points d'arrêt à divers endroits du code, voire de demander un arrêt systématique lors de

l'exécution d'une opération donnée. L'ergonomie de cette interaction a depuis longtemps été défrichée, ce qui nous guide pour la conception de cette interface.

Nous voulons visualiser à la fois le texte source, l'état des glossaires, la pile d'appel et la console d'exécution s'il en est.

La pile d'appels peut être représentée de façon pratique par une liste textuelle sous forme de table, le programme principal apparaissant en haut. Il est souhaitable que l'on puisse consulter le glossaire correspondant à un sous-programme appelant en le sélectionnant dans cette table.

L'état des glossaires (le dictionnaire de constantes à ce niveau est assimilable à un glossaire globalement accessible) peut être représenté de façon pratique par une liste arborescente. Ce type de liste simplifie la représentation et la visualisation des variables de types composés, enregistrements et tableaux.

La mise en place des points d'arrêts pour être ergonomique doit s'effectuer d'un simple clic dans le texte source visualisé, typiquement dans la marge, où le marqueur d'existence du point d'arrêt est affiché. Naturellement, le texte source en cours de débogage ne peut être édité.

Les actions « démarrer l'exécution », « effectuer un pas par-dessus », « effectuer un pas en dedans », et « arrêter l'exécution » sont traditionnellement des boutons rangés en ligne en partie haute de la fenêtre.

Nous avons ainsi cinq zones distinctes : les boutons de contrôle de l'exécution, la vue de la pile d'appel, la vue des glossaires, la vue du texte source et des points d'arrêts définis, et la vue de la console d'exécution s'il en est.

Une représentation schématique de la fenêtre résultante est donnée dans l'illustration suivante, librement inspirée de la perspective *Debug* d'*Eclipse*.

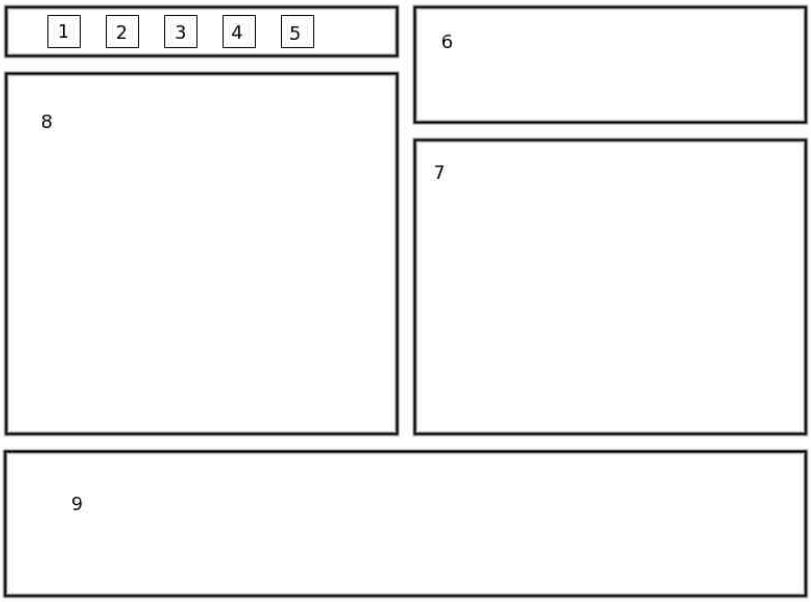

Illustration 3: Schéma de l'interface de débogage

Légende de l'illustration 3 :
Bouton de démarrage de l'exécution (*Run*).
Bouton de pause de l'exécution (*Pause*).
Bouton de pas « par-dessus » (*Step over*).
Bouton de pas « en-dedans » (*Step into*).
Bouton de terminaison de l'exécution (*Kill*).
Liste visualisant la pile d'appels (nom du sous-programme appelant, fichier, ligne).
La sélection d'une ligne entraîne la visualisation dans 7 du glossaire correspondant.

Liste arborescente visualisant le dictionnaire de constantes et le glossaire sélectionné.
Par défaut, le glossaire sélectionné est le glossaire du sous-programme courant.
Visualisation du texte source.
Un clic sur une ligne crée ou annule un point d'arrêt à cette ligne, s'il existe une instruction ou une déclaration de sous-programme Compalgo à cette ligne.
Console d'exécution.

10. Affiche le texte sorti par le programme et permet la saisie d'informations au clavier.Conclusion

Après un bref rappel des bases autour desquelles s'articule la chaîne logicielle mettant en œuvre la solution logicielle Compalgo v2.0, nous avons décomposé cette chaîne et en avons analysé chaque composant en respectant leur ordre conceptuel. La solution logicielle Compalgo v2.0, pour mettre en œuvre les fonctionnalités attendues, implante un *parser* générateur d'AST calibré sur la grammaire LL(1) du langage algorithmique, un moyen de vérification systématique de la validité sémantique de celui-ci, un moyen d'interpréter le programme représenté par cet AST, un moyen de contrôler l'exécution de cet interpréteur, l'intégration du *parser* dans l'interface graphique existante, et une nouvelle fenêtre d'interface pour le contrôle de l'exécution. La mise en œuvre de ces composants avec la plateforme Java sera détaillée dans le rapport de conception.

Rapport de conception

Auteur : Leroux Damien
Date : 21/07/2007

1. Introduction

Ce document décrit la conception de la solution logicielle Compalgo v2.0.

Pré-requis : la lecture de ce document suppose la connaissance du rapport d'analyse préliminaire et le bon entendement des éléments théoriques qu'il présente. Ce document faisant de nombreuses références au rapport d'analyse, le lecteur doit avoir ce rapport à disposition pendant la lecture de ce document.

Avertissement : les diagrammes de classes présentés dans ce document sont des diagrammes partiels destinés à faciliter la compréhension. Notamment, ni les accesseurs ni les constantes symboliques des énumérations ne sont représentés. Le lecteur est prié de se référer à la documentation de référence des classes et paquetages pour avoir le détail des classes.

2. Table des matières

3. Description de l'environnement Compalgo

Qui utilise Compalgo, pourquoi, et quand ?

L'utilisateur final de Compalgo est l'étudiant s'initiant à la programmation impérative dans le cadre des cours donnés à l'IUT « A » de l'Université Paul Sabatier de Toulouse. Il sera amené à écrire et exécuter les programmes donnés en exercice pendant ses séances de TP.

Cas d'utilisation

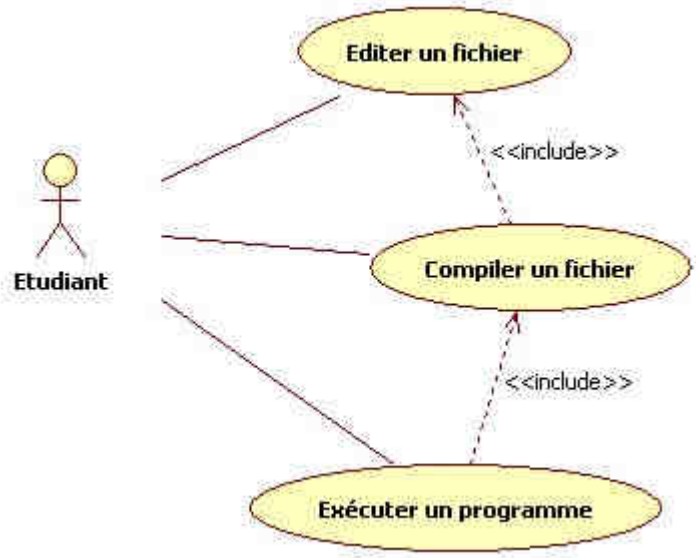

Figure 2 Cas d'utilisation de l'environnement Compalgo

Éditer un fichier

Rôle : créer un nouveau fichier ou modifier un fichier.

Entrées : le chemin du fichier à éditer.

Sorties : un fichier édité.

Pré-conditions : néant.

Post-conditions : il existe au moins un fichier édité.

Exceptions : le fichier n'a pu être ouvert.

Compiler un fichier

Rôle : générer l'AST représentant le code en langage algorithmique contenu dans le fichier.

Entrées : le fichier courant de l'éditeur.

Sorties : l'AST généré s'il n'y a pas eu d'erreur de compilation et/ou une liste de messages d'erreur et d'avertissement.

Pré-conditions : l'utilisateur a édité au moins un fichier.

Post-conditions : il existe un AST valide OU au moins un message d'erreur.

Exécuter un programme

Rôle : dérouler l'exécution d'un programme compilé. L'utilisateur peut contrôler l'exécution et surveiller l'état des variables et de la pile d'appels.

Entrées : un AST valide.

Sorties : néant.

Pré-conditions : il existe un AST valide généré par le compilateur.

Post-conditions : néant.

Description fine de l'interdépendance des utilisations possibles

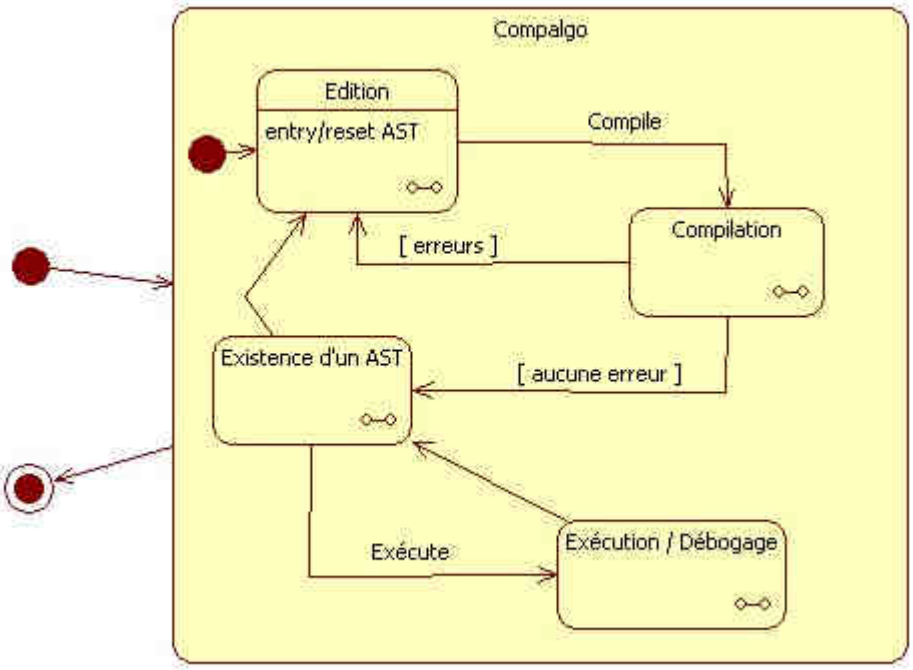

Figure 3 Diagramme états-transitions de l'environnement Compalgo

Le diagramme états-transitions ci-dessus résume les différents scénarios d'utilisation possibles de l'environnement Compalgo. L'utilisateur comment toujours par éditer un fichier, c'est-à-dire au moins le charger dans l'éditeur. Il peut ensuite compiler ce fichier, et uniquement si la compilation a abouti sans erreur, démarrer l'interprétation dans l'environnement de débogage. Le cycle reprend avec chaque édition de fichier, ce qui invalide l'AST existant.

4. Construction de l'AST

Implantation de l'AST

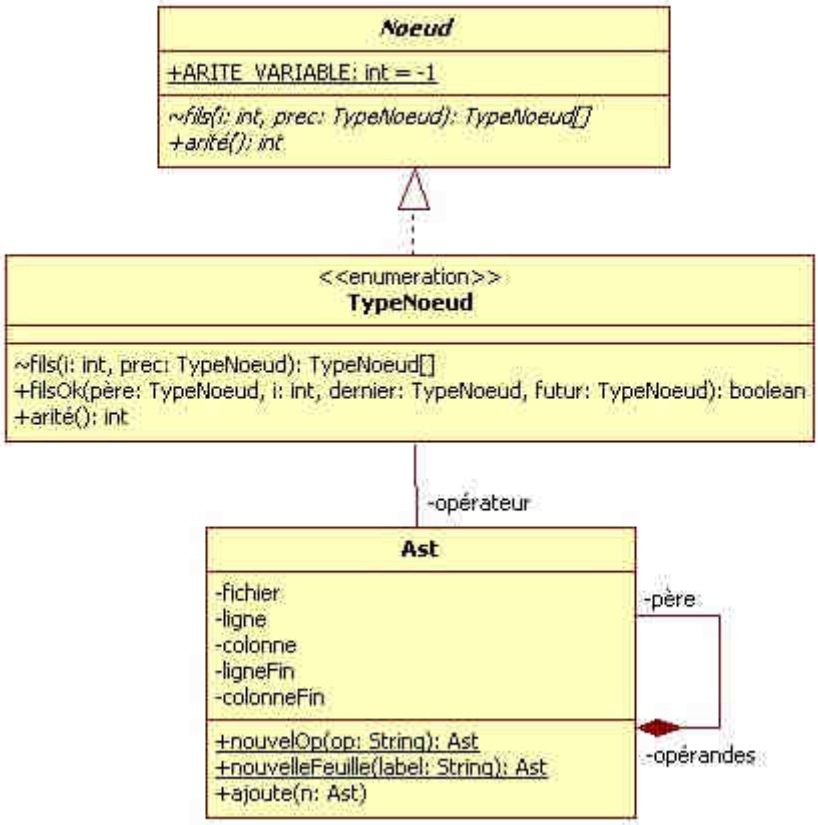

Figure 4 AST - Diagramme de classes

Les constantes de l'énumération TypeNoeud ne sont pas représentées dans ce diagramme pour des raisons de clarté.

interface Noeud

Cette interface définit le moyen de vérifier la structure de l'AST en cours de construction. La méthode *fils*(*i*,*prec*) retourne un tableau de TypeNoeuds indiquant les fils acceptés par le TypeNoeud interrogé au rang *i* et/ou après le fils de type *prec*. La définition de cette interface est nécessaire pour que chaque constante de l'énumération TypeNoeud soit dotée des méthodes *fils*() et *arité*().

enum TypeNoeud

Cette énumération définit l'ensemble des types de nœuds utilisés dans l'AST de Compalgo.

Chaque constante symbolique de l'énumération implante les méthodes *fils*(*i*,*prec*) et *arité*().

class Ast

Cette classe implante un élément de l'AST. Cet élément est un nœud si sa liste de fils *opérandes* existe, et est une feuille sinon. L'attribut *opérateur* est une chaîne qui contient le nom d'une constante de TypeNoeud pour les nœuds, ou pour les feuilles le token correspondant du texte source.

Un élément d'un AST connaît son père, le nom de fichier d'où il est issu, et les positions de début et de fin qui lui correspondent dans le texte source.

Construction

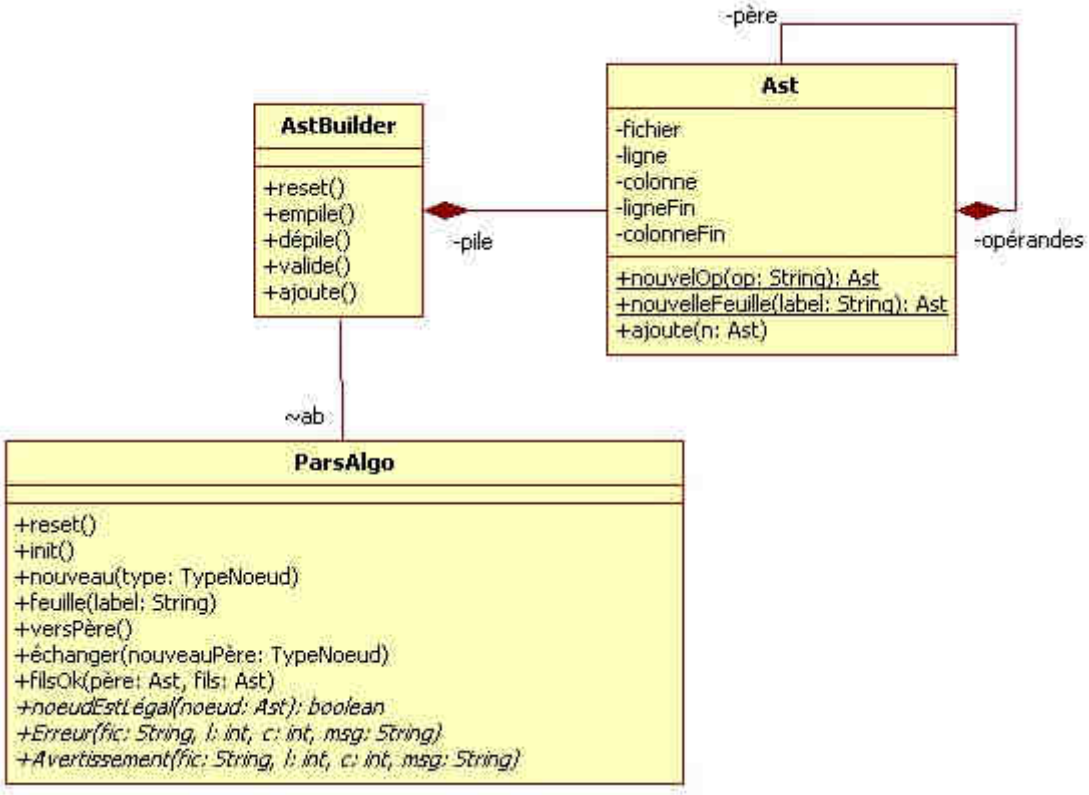

Figure 5 Construction de l'AST - Diagramme de classes

class AstBuilder

Cette classe implante les opérations basiques de construction d'AST au moyen d'une pile. Le nœud en cours d'édition est placé au sommet de la pile, on peut lui ajouter des fils et le retirer de la pile.

class ParsAlgo

Cette classe implante les opérations avancées de construction d'AST utiles dans le contexte de la descente récursive. *nouveau*(), *feuille*(), *échanger*() et *versPère*() sont les opérations définies dans le rapport d'analyse.

En outre, ParsAlgo définit trois méthodes abstraites pour gérer l'émission de messages (*Erreur* et *Avertissement*) et la vérification des nœuds (*noeudEstLégal*). Ces méthodes sont implantées lors de la construction de l'instance de ParsAlgo dans l'initialisation du parser.

5. Arpentage de l'Ast

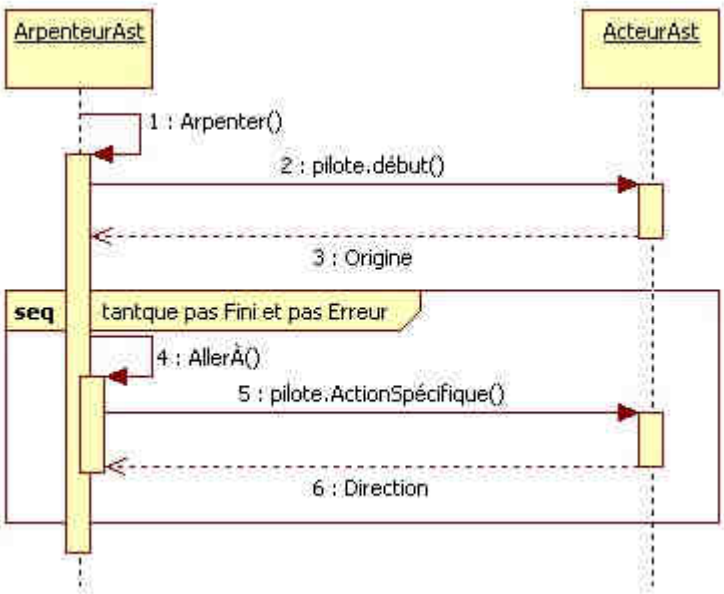

Figure 6 Arpentage de l'AST - Diagramme de séquence

Ce diagramme de séquence illustre le principe de l'arpentage piloté. *ActionSpécifique* représente la méthode correspondant au nom du type de nœud sur lequel se trouve l'Arpenteur.

Description des classes

class Arpenteur

Cette classe implante un itérateur générique de l'AST. Cet itérateur est capable de se mouvoir de nœud en nœud selon les directions définies dans l'énumération Direction. Il est nécessaire d'utiliser un acteur spécialisé (Pilote) pour effectuer un parcours d'AST complet.

abstract class ActeurAst

Cette classe définit la structure et le protocole de base communs à tous les Pilotes d'arpenteur. Les Pilotes définissent un comportement spécifique pour chaque type de nœud qui les intéresse pour mener à bien leur parcours et un comportement par défaut pour les autres types de nœud. Au démarrage d'un parcours, l'Arpenteur interroge son Pilote pour connaître le point de départ. À chaque nœud arpenté, l'Arpenteur montre le nœud au Pilote (i.e. invoque la méthode du Pilote correspondant au type du nœud), qui lui retourne la Direction à prendre ensuite.

interface EvaluateurAst

Cette interface définit le protocole unique de retour de valeur pour les pilotes qui auront à évaluer des informations de l'AST. Le typage générique de Java permet de conserver un typage fort et facilement extensible.

Spécification textuelle d'un acteur spécialisé (Pilote)

Chaque acteur spécialisé (Pilote) est décrit en deux parties. La première partie décrit le Pilote et la deuxième partie décrit son comportement par défaut et le comportement spécifique éventuel pour chaque type de nœud.

Nota **:** il est important pour concevoir un Pilote comme pour en comprendre le fonctionnement d'avoir la description de la structure de l'arbre en tête ou sous les yeux. Pour un but donné, le comportement d'un Pilote pour chaque type de nœud est essentiellement défini en fonction de cette structure.

Description :

Acteur ou Evaluateur	***Nom*** QUI ?
Rôle	(but/opération UNIQUE) POURQUOI ?
Contexte	Dans quel contexte il sera utilisé QUAND ?
Données en entrée	Le contexte supplémentaire que le pilote utilise (identifiant, noeud particulier..) AVEC QUOI ?
Données en sortie	Uniquement pour les Evaluateurs ÉVALUE À ?
Point de départ	L'origine de l'arpentage (cf. enum Origine) À PARTIR D'OÙ ?
Point d'arrivée	e.g. "La déclaration de la variable recherchée si existante, Erreur sinon." JUSQU'OÙ ?
Description du parcours	Description textuelle sommaire du parcours qui sera effectué. COMMENT ?
Utilise	Liste des Pilotes que ce Pilote utilise (y compris lui-même le cas échéant). DÉPEND DE QUI ?

Comportement par type de noeud :

Par défaut	Description du comportement par défaut utilisé quand un comportement spécifique n'a pas été précisé. → Direction retournée : description du cas entraînant ce retour. → Autre direction retournée : description du cas entraînant ce retour.
UnTypeDeNoeud	Description du comportement associé à ce type de nœud. → Direction retournée : description du cas entraînant ce retour. ...
...	...

6. Compilateur

Spécification de parser avec Coco/R

La spécification de *parser* de Coco/R commence par un entête permettant d'ajouter des méthodes et attributs à la classe Parser qui sera générée. Nous avons accès à cet entête et, pour l'utilisation, au point d'entrée *Parse*(), le fichier à *parser* étant donné en argument au constructeur du Parser.

Intégration de la construction d'AST

Une instance de ParsAlgo est ajoutée à la classe Parser, et les appels aux méthodes de construction sont *wrappés* pour simplifier les messages. Les informations de fichier, ligne, colonne et token courant sont implicitées. Cf. documentation de référence.

L'implantation des méthodes abstraites *ParsAlgo.Erreur* et *ParsAlgo.Avertissement* encapsule l'émission d'un message via la classe *Messages* et via l'API définie par Coco/R.

La classe Parser redéfinit une méthode *noeudEstLégal* qui est *wrappée* dans l'implantation de la méthode abstraite *ParsAlgo.noeudEstLégal*.

La class Parser définit aussi une méthode *importer* en charge de gérer les directives **importer** du langage algorithmique.

Vérifications sémantiques

Les vérifications sémantiques ont lieu juste avant une insertion d'un nœud dans l'arbre. La vérification est directement intégrée dans la méthode *versPère*() de la classe ParsAlgo, et l'ajout n'a lieu que si la vérification s'est bien passée.

L'acteur spécialisé en charge des vérifications est nommé GrandVérificateur. Sa description est donnée en annexe.

Classe principale du compilateur : Compalgo

Le compilateur est finalement la classe Compalgo héritant de la classe Parser. Cette classe définit le comportement des méthodes *noeudEstLégal*, qui invoque le GrandVérificateur sur le nœud à vérifier, et *importer*, qui invoque un nouveau compilateur pour le fichier importé, et intègre l'AST généré dans l'AST global.

7. Environnement de développement Compalgo

Intégration du compilateur

L'intégration du compilateur dans l'interface existante se fait en trois parties :

le bouton de compilation existant invoque la compilation du fichier courant par le nouveau compilateur,

le résultat de la compilation (AST valide) est stocké de façon globale et unique,

la console est remplacée par une table listant les messages d'erreur et d'avertissement. Le modèle de cette table est initialisé à partir de la liste de messages existant au moment de la construction de la table.

Interlude

A ce stade, le compilateur est directement utilisable au sein de l'environnement Compalgo. Si à l'issue d'une compilation aucune erreur n'a été générée, nous avons un AST bien formé que nous pouvons dès lors interpréter.

Intégration de l'interface de débogage

8. Nous créons un bouton supplémentaire à droite du bouton de compilation dont l'action est d'initialiser et d'ouvrir la fenêtre de débogage avec l'AST fraîchement généré. Ce bouton est activé après une compilation sans erreur et désactivé par toute édition ou changement de fichier courant.Interprétation et débogage des programmes compilés

Interprétation

Architecture

Comme défini dans le rapport d'analyse, l'interprétation est répartie dans quatre acteurs spécialisés, plus une classe définissant le contexte d'exécution et le point d'entrée de l'interprétation.
Le contexte d'exécution comporte :
un dictionnaire de constantes,
un dictionnaire de variables pour le programme principal et chaque sous-programme appelé,
une pile de données pour les évaluations et les passages de paramètres.

Représentation des données concrètes

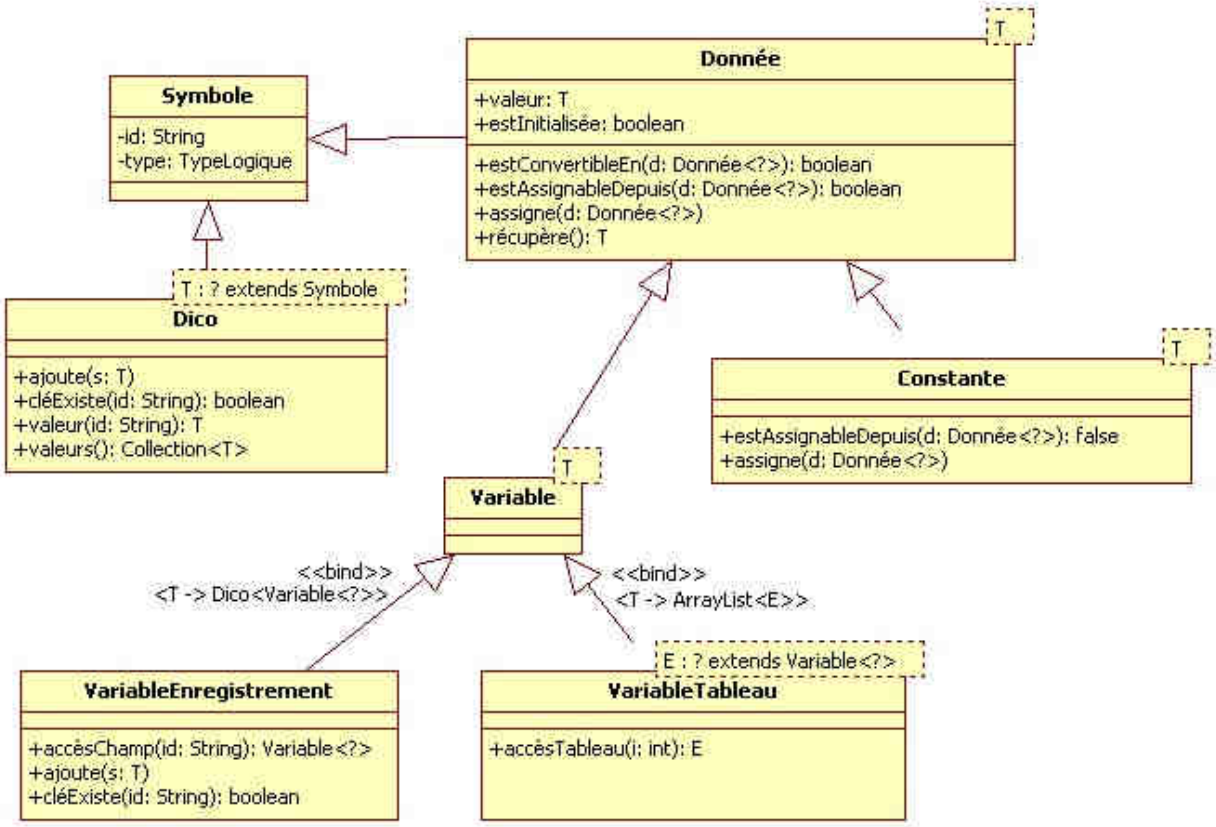

Figure 7 Représentation des données pendant une exécution

La classe Donnée<T> fait le lien entre les types concrets de la machine virtuelle de Java et les types logiques considérés dans Compalgo.

La classe Dico<T extends Symbole> implante les différents dictionnaires.

La classe Constante<T> interdit toute affectation.

La classe Variable<T> sert uniquement à la hiérarchie objet, et n'implante aucune fonctionnalité spécifique. En revanche, VariableEnregistrement et VariableTableau<E> implantent les accès aux champs et aux éléments respectivement. Le paramètre générique E est le type des éléments du tableau, restreint aux classes dérivées de Variable<?>.

La création d'une Donnée<T> à partir d'une donnée Java se fait de façon quasi-transparente à la construction, et l'utilisation de la valeur d'une Donnée<T> se fait de façon totalement transparente à travers l'attribut valeur ou la méthode récupère().

Bibliothèque de sous-programmes natifs Java

Le mécanisme générique d'invocation de méthodes Java (cf. Annexe 10.A) est utilisée pour implanter la bibliothèque native extensible. Lors de l'appel d'un sous-programme, si le nœud déclarant ne comporte pas de nœud BlocDInstructions, Compalgo considère que c'est un sous-programme natif, recherche une classe ayant le même nom que le fichier où est défini le sous-programme, d'abord dans le paquetage par défaut, puis dans le paquetage de la classe GestionnaireDeBibliothèques, fr.irit.sig.compalgo.interpréteur.bibliothèque. Il recherche dans la classe trouvée une méthode ayant la même signature que le sous-programme appelé. Cette méthode est alors invoquée après évaluation des paramètres effectifs.

La bibliothèque proprement dite est une HashMap d'objets Invocation, les clés étant les signatures des sous-programmes.

Optimisation

La réutilisation des nombreux acteurs spécialisés au cours de l'interprétation génère une énorme redondance des traitements effectués. Cela ralentit notablement l'interprétation et il est nécessaire d'implanter un cache pour les différents Évaluateurs.

Le principe de ce cache est simple : un Évaluateur donné utilisé pour un nœud donné produit toujours le même résultat, on peut donc associer le résultat de l'évaluation à la clé composite Évaluateur, nœud.

Contrôle de l'exécution

Modèle de base de "Machine Virtuelle Virtuelle"

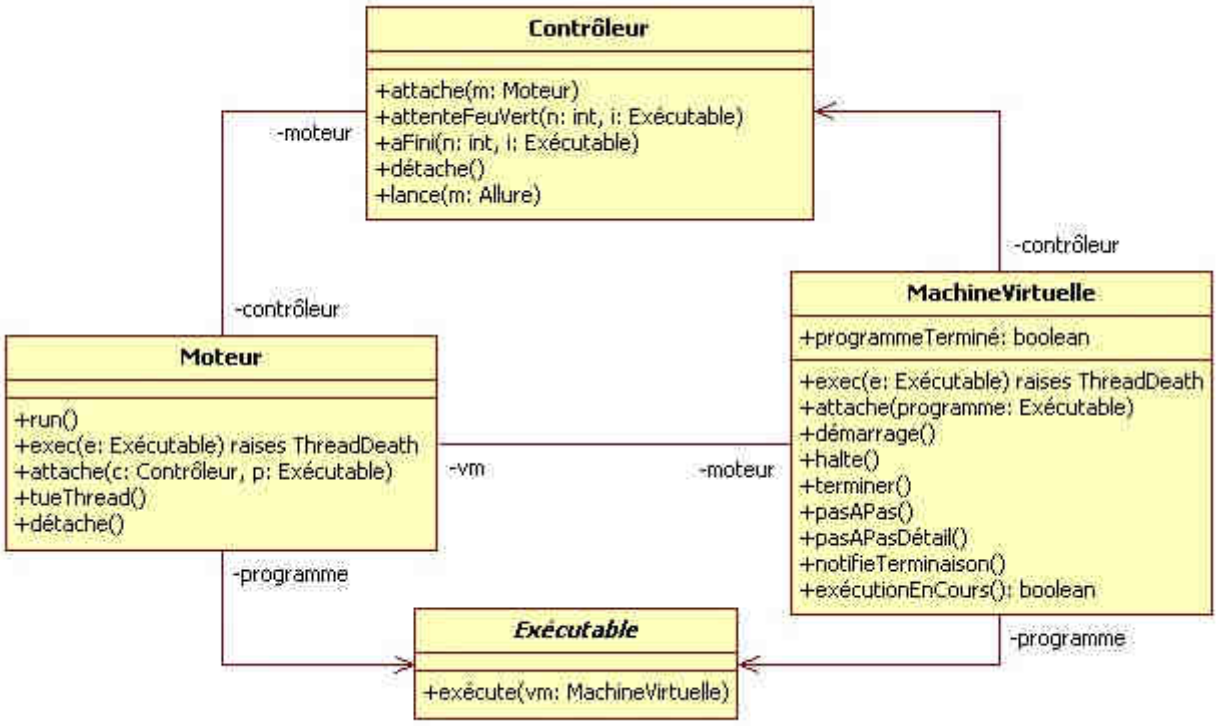

Figure 8 Machine Virtuelle Virtuelle - Diagramme de classes

La classe MachineVirtuelle est le *front-end* du système. Elle crée et dispose du Moteur et du Contrôleur.

Le Moteur crée un *thread* dédié qui démarre par l'exécution du programme spécifié.

Pour chaque exécution d'une instance d'Exécutable, le Moteur attend le feu vert du Contrôleur, qui est piloté par le mode d'exécution de la MachineVirtuelle (*démarrage*, *pasAPas*, *pasAPasDétail*). À la fin de l'exécution, le Moteur signale au contrôleur que cette exécution a fini.

Intégration de la MVV et de l'interpréteur

La classe ActeurAstExécutable sert de *Bridge* entre l'interface Exécutable et la classe ActeurAst. L'action de l'ActeurAst est ainsi soumise au contrôle d'exécution de la MachineVirtuelle.

Un ActeurAstExécutable est obligé de connaître la MachineVirtuelle et le nœud courant.

Points d'arrêt : mécanisme de filtrage supplémentaire

Les points d'arrêt définissables par l'utilisateur peuvent se situer sur une instruction, un début de bloc d'instructions, un bloc de traitement d'exceptions, ou un traitement d'exception particulière. L'utilisateur choisit un point d'arrêt à une ligne donnée d'un fichier donné, et le système a besoin de connaître le nœud de l'AST qui correspond à cet emplacement. Un acteur spécialisé, ChercheurDePA, est utilisé pour convertir l'information { Fichier, Ligne } fournie par l'utilisateur en une référence sur un nœud.

Les points d'arrêt définis sont stockés dans une liste associative de clé { Fichier, Ligne } et de valeur { Nœud }.

C’est le Contrôleur qui est en charge de donner le feu vert au Moteur, ainsi il fournit le prédicat estUnPointDArrêt(nœud). La classe MachineVirtuelle étant le *front-end* du système de MVV, elle réimplémente ce prédicat en le *wrappant*, afin que l’utilisateur de la MachineVirtuelle puisse à son tour l’implémenter.

Avant chaque envoi de feu vert, le Contrôleur teste le prédicat et interrompt l’exécution s’il évalue à vrai.

L’Interpréteur est en charge d’implémenter ce prédicat, étant dans le même contexte que la class ActeurAstExécutable. De ce fait, la liste de points d’arrêts est définie en attribut de l’Interpréteur.

L’Interpréteur fournit les méthodes de gestion de la liste des points d’arrêt :

réinitialisePA() : vide la liste,

alternePA(fichier,ligne) : crée ou annule le point d’arrêt à cet emplacement.

9. Interface au contrôle de l'exécution

L’illustration ci-après rappelle l’organisation de l’interface de contrôle telle que spécifiée dans le rapport d’analyse.

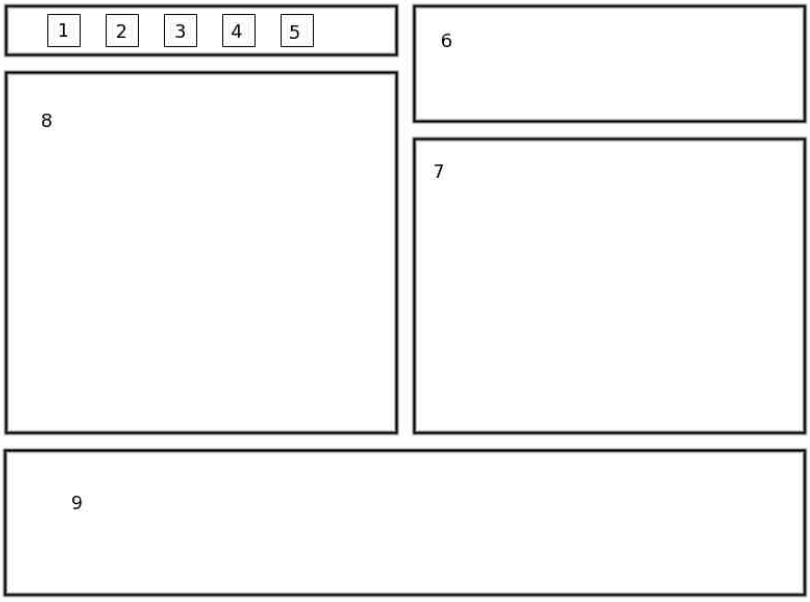

Illustration 4: Rappel du schéma de l'interface de débogage

Légende :

1. Bouton de démarrage de l'exécution (*Run*).
2. Bouton de pause de l'exécution (*Pause*).
3. Bouton de pas « par-dessus » (*Step over*).
4. Bouton de pas « en-dedans » (*Step into*).
5. Bouton de terminaison de l'exécution (*Kill*).
6. Liste visualisant la pile d'appels (nom du sous-programme appelant, fichier, ligne).
   La sélection d'une ligne entraîne la visualisation dans 7 du glossaire correspondant.
7. Liste arborescente visualisant le dictionnaire de constantes et le glossaire sélectionné.
   Par défaut, le glossaire sélectionné est le glossaire du sous-programme courant.
8. Visualisation du texte source.
   Un clic sur une ligne crée ou annule un point d'arrêt à cette ligne, s'il existe une instruction ou une déclaration de sous-programme Compalgo à cette ligne.
9. Console d'exécution.
   Affiche le texte sorti par le programme et permet la saisie d'informations au clavier.

Vue des données

Les données à visualiser sont :

[6] La pile d'appel : JTable à trois colonnes { Fichier, Ligne, Sous-programme appelé }. Les appels successifs sont ordonnés de haut en bas. Le sous-programme courant (dernier appel) est sélectionné par défaut.

[7] Le glossaire courant : JTree. La racine comporte deux éléments : Glossaire et Constantes. Ces deux éléments contiennent respectivement les variables du contexte courant et les constantes globales. Le contexte courant est défini par la sélection d'une ligne de [6].

[8] Le fichier source en cours d'exécution : une sélection de texte matérialise le code exécuté à chaque instant.

[9] La console d'exécution : affiche la sortie standard du programme et permet à l'utilisateur d'entrer des données. Les couleurs de texte de la sortie et de l'entrée diffèrent pour un meilleur confort visuel.

Arrangement

Les composants sont répartis dans la hiérarchie de conteneurs suivante :

JSplitPane (VERTICAL_SPLIT)

TOP : JSplitPane (HORIZONTAL_SPLIT)

LEFT : JPanel (BorderLayout)

NORTH : JPanel (GridLayout(1,5))

[1] à [5] : JButton

CENTER : [8] : JTextArea

RIGHT : JPanel (BorderLayout)

NORTH : [6] : JTable

CENTER : [7] : JTree

BOTTOM : [9] : Console

Actions

Bouton marche : déclenche l'exécution du programme sans pause.

Bouton pause : interrompt l'exécution.

Bouton arrêt : termine l'exécution.

Bouton pas en-dedans : exécute l'instruction courante. Si c'est une instruction d'appel, interrompt l'exécution avant la première instruction du sous-programme appelé.

Bouton pas par-dessus : exécute complètement l'instruction courante.

Clic sur une entrée de la pile d'appels : sélectionne cet appel et affiche le glossaire correspondant.

Double-clic dans le texte source : crée ou détruit le point d'arrêt à cet endroit du code si le clic a eu lieu sur une instruction.

10. Annexes

Mécanisme d'invocation de méthodes par leur nom

L'invocation de méthodes utilise le mécanisme d'introspection de Java. Une classe Invocateur définit une méthode statique Invocateur.créer(instance, nomDeMéthode, TypesArguments[]) retournant un objet de type Invocation. Cette Invocation est ensuite utilisable directement pour invoquer la méthode visée.

Répartition des classes en paquetages

fr.irit.sig.compalgo : implante la classe principale du compilateur.

ast : implante l'AST et sa construction.

acteurs : regroupe tous les pilotes d'arpenteur, à l'exception des pilotes mentionnés ci-dessous.

interpréteur : classe principale de l'interpréteur et pilotes d'arpenteur spécifiques à l'interprétation.

bibliothèque : gestionnaire de bibliothèques.

runtime : classes nécessaires à la gestion du contexte d'exécution.

vm : classes de la Machine Virtuelle Virtuelle.

message : classes gérant l'émission de messages d'erreur et d'avertissement.

parser : classes nécessaires au *parser* généré par Coco/R, et ce parser.

util : classes utilitaires.

invocation : classes gérant l'invocation de méthodes Java.

visu : classes permettant la visualisation des données.

Descriptions des Pilotes d'Arpenteur

ChercheurDeDeclChamp

Description :

Evaluateur	***ChercheurDeDeclChamp***
Rôle	Cherche un champ de nom donné dans une déclaration d'enregistrement donnée.
Contexte	Vérifications sémantiques.
Données en entrée	Un nœud DeclEnregistrement et un identifiant.
Données en sortie	Le sous-arbre contenant la déclaration du champ.
Point de départ	Nœud donné en entrée.
Point d'arrivée	La déclaration du champ recherché si existant, ou Erreur.
Description du parcours	Le Pilote entre dans le nœud DeclEnregistrement puis parcours les déclarations de champ jusqu'à trouver le bon.
Utilise	Néant.

Comportement par type de noeud :

Par défaut	→ Erreur.
DeclEnregistrement	→ Dedans.
DeclVariable	→ Fini : si le nom du champ déclaré correspond au nom recherché. → Suivant : sinon.

ChercheurDeDeclConst

Description :

Evaluateur	*ChercheurDeDeclConst*
Rôle	Recherche la déclaration de constante correspondant à un identifiant donné.
Contexte	Vérifications sémantiques.
Données en entrée	Un identifiant de constante.
Données en sortie	La déclaration de constante correspondante ou Erreur.
Point de départ	Racine.
Point d'arrivée	La déclaration de constante, ou le nœud Programme.
Description du parcours	Recherche linéairement dans le nœud racine Compalgo la bonne déclaration. L'arrivée sur le nœud Programme ou la sortie de Compalgo dénotent une erreur.
Utilise	Néant.

Comportement par type de noeud :

Par défaut	*Suivant.*
Compalgo	Les déclarations de constantes sont à l'intérieur de ce nœud. → Dedans.
DeclConstante	Vérifie si l'identifiant correspond à celui recherché. → Fini : si l'identifiant concorde. → Suivant : sinon.
Programme	Il est certain que la constante n'est pas déclarée. → Erreur.

ChercheurDeDeclSP

Description :

Evaluateur	*ChercheurDeDeclSP*
Rôle	Recherche la déclaration de sous-programme correspondant à une instruction d'appel donnée.
Contexte	Vérifications sémantiques et interprétation.
Données en entrée	Un nœud InsAppel.
Données en sortie	La déclaration du sous-programme correspondant ou Erreur.
Point de départ	Nœud.
Point d'arrivée	La déclaration de sous-programme, ou Erreur.
Description du parcours	En partant de l'instruction d'appel fournie, il remonte jusqu'à la racine, pour permettre la détection du sous-programme actuellement parsé. Une fois passé par le nœud Compalgo, il parcourt linéairement tous les fils jusqu'à trouver la déclaration correspondante.
Utilise	Néant.

Comportement par type de noeud :

Par défaut	On remonte à la recherche d'un contexte plus favorable. → VersPère.
Compalgo	Les déclarations de sous-programmes sont à l'intérieur de ce nœud. → Dedans.
DeclConstante	→ Suivant.
DeclType	→ Suivant.
DeclFonction DeclProcédure	→ Fini : si le nom du sous-programme, le nombre et les types des arguments sont les mêmes. → VersPère : sinon, et si le Pilote n'est pas encore passé par le nœud Compalgo. → Suivant : sinon.
Import	→ Suivant.
Programme	Il est certain que le sous-programme n'est pas déclaré. → Erreur.

ChercheurDeDeclType

Description :

Evaluateur	*ChercheurDeDeclType*
Rôle	Recherche la déclaration de type correspondant à un identifiant donné.
Contexte	Vérifications sémantiques.
Données en entrée	Un nœud UtilType.
Données en sortie	La déclaration de type correspondante ou Erreur.
Point de départ	Racine.
Point d'arrivée	La déclaration de type, ou le nœud Programme.
Description du parcours	Recherche linéairement dans le nœud racine Compalgo la bonne déclaration. L'arrivée sur le nœud Programme ou la sortie de Compalgo dénotent une erreur.
Utilise	Néant.

Comportement par type de noeud :

Par défaut	→ Suivant.
Compalgo	Les déclarations de types sont à l'intérieur de ce nœud. → Dedans.
DeclType	Vérifie si l'identifiant correspond à celui recherché. → Fini : si l'identifiant concorde. → Suivant : sinon.
Programme	Il est certain que la constante n'est pas déclarée. → Erreur.

ChercheurDeDeclVar

Description :

Evaluateur	*ChercheurDeDeclVar*
Rôle	Recherche la déclaration d'une variable dans le contexte courant
Contexte	Vérifications sémantiques.
Données en entrée	Un nœud Id représentant une variable.
Données en sortie	La déclaration de la variable, ou Erreur.
Point de départ	Nœud, si le nœud fait déjà partie de l'arbre. Sommet, sinon.
Point d'arrivée	La déclaration de la variable recherchée si existante, Erreur sinon.
Description du parcours	Le Pilote remonte dans l'arbre à la recherche d'un contexte de déclarations de variables (déclaration de programme ou sous-programme). S'il arrive à la racine, la variable n'est pas déclarée. S'il trouve un nœud DeclFonction, DeclProcédure, ou Programme, il le parcourt à la recherche de la déclaration de variable ou de paramètre correspondante.

Evaluateur	*ChercheurDeDeclVar*
Utilise	Néant.

Comportement par type de noeud :

Par défaut	→ VersPère.
Compalgo	Le pilote est allé trop loin. → Erreur.
Programme DeclProcédure DeclFonction	Point d'entrée. → Dedans.
DeclVariable DeclParamètre	Compare l'identifiant trouvé et l'identifiant recherché. → Fini : si l'identifiant correspond. → Suivant : sinon.
Exception Id UtilType	Non traité. → Suivant.

ChercheurDePA

Description :

Evaluateur	*ChercheurDePA*
Rôle	Recherche le nœud convenable pour un point d'arrêt donné.
Contexte	Création d'un point d'arrêt lors du débogage d'un programme.
Données en entrée	Le nom du fichier visualisé et la ligne du point d'arrêt désiré.
Données en sortie	Le nœud convenable, ou Erreur.
Point de départ	Racine.
Point d'arrivée	Le nœud instruction qui démarre à la ligne spécifiée, Erreur sinon.
Description du parcours	Le Pilote descend dans l'arbre à la recherche d'un nœud acceptable pour définir un point d'arrêt et commençant à la ligne demandée dans le fichier demandé.
Utilise	Néant.

Comportement par type de noeud :

F_{PA} représente le fichier visualisé, L_{PA} représente la ligne du point d'arrêt désiré.

F représente le nom de fichier correspondant au nœud courant, L_D représente la ligne de début du nœud courant, L_F représente la ligne de fin du fichier courant.

Par défaut	→ Dedans : si $L_D <= L_{PA}$ et $L_F >= L_{PA}$. → Suivant : si $L_F < L_{PA}$. → Erreur : sinon.
Compalgo	→ Dedans.
BlocDInstructions	→ Fini : si $L_D == L_{PA}$.
TraitementException	→ Dedans : si $L_D <= L_{PA}$ et $L_F >= L_{PA}$.

Par défaut	→ Dedans : si $L_D <= L_{PA}$ et $L_F >= L_{PA}$. → Suivant : si $L_F < L_{PA}$. → Erreur : sinon.
LorsqueException	
InsSi InsTantque InsRetourner InsAffectation InsDéclencher InsAppel	→ Fini : si $L_D == L_{PA}$. → Suivant : si $L_F < L_{PA}$. → Erreur : sinon.

GrandVérificateur

Description :

Acteur	***GrandVérificateur***
Rôle	Effectue les vérifications sémantiques au fur et à mesure de la création des nœuds de l'AST.
Contexte	Juste avant l'insertion dans l'AST d'un nouveau nœud, celui-ci est soumis à validation auprès du Grand Vérificateur. Le nœud n'est inséré que si le Grand Vérificateur termine la visite du nœud avec la direction Fini.
Données en entrée	Néant.
Données en sortie	Néant.
Point de départ	Nœud.
Point d'arrivée	Dépend du type du nœud à vérifier.
Description du parcours	Dépend du type du nœud à vérifier.
Utilise	Vérifications sémantiques.

Comportement par type de noeud :

Par défaut	Le nœud n'est pas sujet à vérification. → Fini.
InsAppel	Vérifie la validité de chaque argument effectif (expression) et la déclaration préalable du sous-programme appelé. → Fini : si tout va bien. → Erreur : sinon.
DeclVariable DeclParamètre DeclConstante DeclType	Vérifie l'unicité contextuelle de la déclaration. → Fini : si tout va bien. → Erreur : sinon.
UtilVariable UtilType UtilConstante	Vérifie l'existence de la déclaration préalable. → Fini : si tout va bien. → Erreur : sinon.

Par défaut	Le nœud n'est pas sujet à vérification. → Fini.
IdAmbiguë	Détermine si le symbole représente une variable ou une fonction sans argument. → Fini : si le symbole correspond exclusivement à une variable ou à une fonction. L'attribut opérateur du nœud a alors été mis à jour. → Erreur : sinon.

DeclFonction	Vérifie le que mode de passage de chaque argument formel est bien *entrée*. S'il est un corps : - Vérifie la consistance des déclarations d'exceptions déclenchables en fonction des traitements locaux d'exception et des exceptions déclenchées. - S'il existe une déclaration préalable : o Vérifie qu'elle ne contient pas déjà un corps. o Vérifie la concordance des modes de passage des paramètres formels, du type de retour et des déclarations d'exceptions déclenchables. o Ajoute le corps à la déclaration préalable. S'il n'est pas de corps : - Vérifie qu'il n'existe pas de déclaration préalable. → Fini : si tout va bien et qu'on n'a pas ajouté le corps à la déclaration préalable. → Erreur : sinon.
DeclProcédure	S'il est un corps : - Vérifie la consistance des déclarations d'exceptions déclenchables en fonction des traitements locaux d'exception et des exceptions déclenchées. - S'il existe une déclaration préalable : o Vérifie qu'elle ne contient pas déjà un corps. o Vérifie la concordance des modes de passage des paramètres formels, du type de retour et des déclarations d'exceptions déclenchables. o Ajoute le corps à la déclaration préalable. S'il n'est pas de corps : - Vérifie qu'il n'existe pas de déclaration préalable. → Fini : si tout va bien et qu'on n'a pas ajouté le corps à la déclaration préalable. → Erreur : sinon.
OpAritUnMoins	Vérifie que l'expression évalue à un Entier ou à un Réel. → Fini : si tout va bien. → Erreur : sinon.
OpAritMoins OpAritPlus OpAritMult OpCompEq OpCompNEq OpCompInf OpCompSup OpCompInfEq OpCompSupEq	Vérifie que les deux expressions évaluent toutes les deux au même type, Entier ou Réel. → Fini : si tout va bien. → Erreur : sinon.
OpAritDiv	Vérifie que les deux expressions évaluent toutes les deux au type Réel. → Fini : si tout va bien. → Erreur : sinon.
OpAritDivEnt OpAritModulo	Vérifie que les deux expressions évaluent toutes les deux au type Entier. → Fini : si tout va bien. → Erreur : sinon.

OpBoolNon	Vérifie que l'expression évalue à un Booléen. → Fini : si tout va bien. → Erreur : sinon.

OpBoolEt OpBoolOu	Vérifie que l'expression évalue à un Booléen. → Fini : si tout va bien. → Erreur : sinon.
InsRetourner	Vérifie que l'instruction est située dans une déclaration de fonction. → Fini : si tout va bien. → Erreur : sinon.
BlocDInstruction	Vérifie que le fichier parsé n'est pas un fichier de spécification → Fini : si tout va bien. → Erreur : sinon.
Programme	Vérifie que le fichier parsé n'est pas un fichier importé. → Fini : si tout va bien. → Erreur : sinon.

InterpréteurCompalgo

Description :

Acteur	***InterpréteurCompalgo***
Rôle	Point d'entrée de l'interprétation d'un AST.
Contexte	Appelé par l'Interpréteur une fois son contexte initialisé pour démarrer l'interprétation proprement dite.
Données en entrée	Néant.
Données en sortie	Néant.
Point de départ	Racine.
Point d'arrivée	Après le dernier fils du nœud racine.
Description du parcours	Parcourt linéairement les fils du nœud racine.
Utilise	InterpréteurSousProgramme

Comportement par type de noeud :

Par défaut	Ce cas n'est pas censé se produire. → Erreur.
Compalgo	Ce nœud comporte notamment les déclarations de constantes et le programme principal. → Dedans.
Import DeclType DeclFonction DeclProcédure	Non traité. → Suivant.
DeclConstante	Crée la donnée correspondante dans le dictionnaire de constantes. → Suivant.
Programme	Appelle un InterpréteurSousProgramme pour interpréter le programme principal. → Dedans.

InterpréteurExpression

Description :

Evaluateur	*InterpréteurExpression*
Rôle	Evalue une expression et empile le résultat de l'évaluation sur la pile de l'interpréteur.
Contexte	Utilisé pour évaluer les bornes d'indiçage d'un type tableau, les membres droits des affectations, les conditions des instructions *si* et *tantque*, les indices des tableaux, et les arguments effectifs des instructions d'appel.
Données en entrée	Le nœud expression à évaluer.
Données en sortie	La Donnée<?> résultat de l'évaluation, placée sur le sommet de la pile.
Point de départ	Nœud.
Point d'arrivée	Relatif au type du nœud à évaluer.
Description du parcours	Relatif au type du nœud à évaluer.
Utilise	InterpréteurExpression InterpréteurSousProgramme SolveurDeType SolveurDeVariable

Comportement par type de noeud :

Par défaut	Ce cas n'est pas censé arriver. → Erreur.
InsAppel	Exécute l'appel de sous-programme. La valeur retournée se trouve sur le sommet de la pile. → Fini.
Entier Réel Chaîne Caractère Booléen	Empile cette donnée. → Fini.
UtilVariable	Résouds l'accès à la variable et l'empile. → Fini.
OpAritUnMoins	Évalue l'opérande de type Entier ou Réel et empile son opposé. → Fini.
OpAritMoins OpAritPlus OpAritMult OpAritDiv OpAritDivEnt OpAritModulo	Évalue les deux opérandes, et empile le résultat de l'opération. → Fini.

OpCompEq OpCompNEq OpCompSup OpCompInf OpCompSupEq OpCompInfEq	Évalue les deux opérandes, et empile la valeur de vérité de la comparaison. → Fini.
OpBoolNon	Évalue l'opérande, et empile son complément logique. → Fini.
OpCompEt OpCompOu	Évalue les deux opérandes, et empile la valeur de vérité de l'opération logique. → Fini.

InterpréteurSéquence

Description :

Acteur	***InterpréteurSéquence***
Rôle	Effectue l'interprétation d'un bloc d'instructions.
Contexte	Utilisé initialement par l'InterpréteurSousProgramme.
Données en entrée	Le nœud contenant la séquence d'instructions à interpréter.
Données en sortie	Néant.
Point de départ	Nœud.
Point d'arrivée	La fin de la séquence ou un nœud InsDéclencher ou un nœud InsRetourner.
Description du parcours	Parcourt linéairement toutes les instructions jusqu'à finir ou interpréter un nœud InsDéclencher ou InsRetourner.
Utilise	InterpréteurExpression InterpréteurSéquence InterpréteurSousProgramme SolveurDeVariable

Comportement par type de noeud :

Par défaut	Ce cas n'est pas censé arriver. → Erreur.
InsRetourner	Evalue l'expression à retourner et déclenche une exception RetourDeValeur munie du résultat de l'évaluation. → N/A.
InsDéclencher	Déclenche une exception DéclenchementDException munie de l'identifiant d'exception. → N/A.
InsAffectation	Résoud la variable en membre gauche, évalue l'expression en membre droit, et assigne à la première la valeur de la seconde. → Suivant.
InsAppel	Invoque un InterpréteurSousProgramme pour cette instruction d'appel. → Suivant.

Par défaut	Ce cas n'est pas censé arriver. → Erreur.
InsSi	Evalue la condition, invoque un InterpréteurSéquence pour interpréter le premier bloc si elle est vraie, ou invoque un InterpréteurSéquence pour interpréter le deuxième bloc s'il est défini et que la condition est fausse. → Suivant.
InsTantque	Evalue la condition, et invoque un InterpréteurSéquence pour interpréter le bloc d'instructions si elle est vraie. → Saut : sur le même nœud si la condition est vraie. → Suivant : si la condition est fausse.
Id	Non traité. → Suivant.
BlocDInstruction	Point d'entrée. → Dedans.
LorsqueException	Point d'entrée. → Dedans.

InterpréteurSousProgramme

Description :

Evaluateur	***InterpréteurSousProgramme***
Rôle	Interprète l'appel d'un sous-programme ou le programme principal.
Contexte	Invoqué initialement par l'InterpréteurCompalgo, puis pour chaque instruction d'appel rencontrée.
Données en entrée	Un nœud InsAppel ou le nœud Programme.
Données en sortie	Booléen : vrai si le sous-programme a retourné une valeur, faux sinon.
Point de départ	Nœud.
Point d'arrivée	Dans le cas d'un sous-programme Java, pas de parcours. InsRetourner termine l'InterpréteurSousProgramme. InsDéclencher interrompt l'interprétation et branche sur le traitement d'exception du sous-programme s'il est définit. Le point d'arrivé devient alors le nœud LorsqueException correspondant à l'exception levée, ou la fin du bloc TraiteException s'il faut la propager.
Description du parcours	Dans le cas d'un sous-programme Java, pas de parcours. Après avoir évalué et empilé les arguments effectifs **de droite à gauche**, parcourt linéairement tous les fils de la déclaration de sous-programme correspondante. Si une exception est déclenchée, reprend le parcours à partir du nœud TraiteException à la fin du BlocDInstructions.
Utilise	ChercheurDeDeclSP InterpréteurExpression InterpréteurSéquence SolveurDeType

Comportement par type de noeud :

Par défaut	Ce cas n'est pas censé se produire. → Erreur.
Id	Non traité. → Suivant.
Programme DeclFonction DeclProcédure	Point d'entrée. → Dedans.
UtilType Exception	Non traité. → Suivant.
DeclParamètre	Dépile une donnée et initialise une nouvelle entrée du glossaire courant avec cette donnée. → Suivant.
DeclVariable	Crée une nouvelle entrée non initialisée dans le glossaire courant. → Suivant.

BlocDInstruction	Invoque un InterpréteurSéquence pour ce nœud. → Fini : si le traitement n'a pas été interrompu par un retour de valeur ou un déclenchement d'exception. → N/A : sinon.
TraitementAnomalie	Point d'entrée. → Fini : si aucune exception n'a été déclenchée. → Dedans : sinon.
LorsqueException	Point d'entrée. → Dedans : si l'identifiant d'exception traité est l'identifiant d'exception déclenchée. → Suivant : sinon.

SolveurDeType

Description :

Evaluateur	***SolveurDeType***
Rôle	Détermine le type logique qui correspond à un nœud donné.
Contexte	À chaque fois qu'il y a besoin de connaître un type de donnée.
Données en entrée	Le nœud à évaluer.
Données en sortie	Un TypeLogique.
Point de départ	Nœud.
Point d'arrivée	Relatif au type de nœud à évaluer.
Description du parcours	Relatif au type de nœud à évaluer.
Utilise	ChercheurDeDeclChamp ChercheurDeDeclConst ChercheurDeDeclSP ChercheurDeDeclType ChercheurDeDeclVar SolveurDeType

Comportement par type de noeud :

Par défaut	Le nœud n'a aucun type. → Fini.
OpAritUnMoins	Évalue le type de l'unique opérande. → Dedans.
OpAritMoins OpAritPlus OpAritMult	Évalue le type de chaque opérande. → Fini : si les opérandes sont toutes les deux de type Entier ou toutes les deux de type Réel. → Erreur : sinon.
OpAritDiv	Évalue le type de chaque opérande. → Fini : si les opérandes sont toutes les deux de type Réel. → Erreur : sinon.
OpAritDivEnt OpAritModulo	Évalue le type de chaque opérande. → Fini : si les opérandes sont toutes les deux de type Entier.

Par défaut Le nœud n’a aucun type.
→ Fini.

→ Erreur : sinon.

OpBoolNon	Évalue le type de l'unique opérande. → Fini : si l'opérande est de type Booléen. → Erreur : sinon.
OpBoolEt OpBoolOu	Évalue le type de chaque opérande. → Fini : si les opérandes sont toutes les deux de type Booléen. → Erreur : sinon.
OpCompInf OpCompSup OpCompEq OpCompNEq OpCompInfEq OpCompSupEq	Évalue le type de chaque opérande. → Fini : si les opérandes sont toutes les deux de type Booléen. → Erreur : sinon.
Id	Cherche la déclaration de la variable identifiée par ce nœud. → Saut : à la déclaration si elle existe. → Erreur : sinon.
UtilVariable UtilConstante	→ Saut : à la déclaration de la variable ou constante utilisée. → Erreur : si la variable ou la constante n'est pas déclarée.
DeclConstante DeclParamètre DeclType	→ Saut : au type utilisé dans la déclaration.
DeclTypeTableau	Vérifie que les bornes d'indiçage sont bien de type Entier, et résoud le type des éléments du tableau. → Fini : si tout va bien. → Erreur : sinon.
DeclTypeEnregistre-ment	→ Fini.
AccèsChamp	Cherche la déclaration du champ mentionné. → Saut : à la déclaration si existante. → Erreur : sinon.
AccèsTableau	Vérifie que l'indice est bien un Entier, puis résoud au type des éléments du tableau. → Fini : si tout va bien. → Erreur : sinon.
Entier Booléen Réel Chaîne Caractère Exception	Résolution immédiate du type. → Fini.
DeclVariable	Vérifie que l'indice est bien un Entier, puis résoud au type des éléments du tableau. → Fini : si tout va bien. → Erreur : sinon.

UtilType	Résoud le type utilisé. La résolution est immédiate s'il s'agit d'un type primitif. Sinon, un ChercheurDeDeclType donne la déclaration à un nouveau SolveurDeType qui se charge de la résoudre. → Fini : si tout va bien. → Erreur : sinon.
DeclFonction	→ Saut : au type de retour de la fonction.

SolveurDeTypeSP

Description :

Evaluateur	***SolveurDeTypeSP***
Rôle	Détermine dans quel contexte l'on se trouve à un instant donné.
Contexte	Utilisé quand il est nécessaire de savoir si l'on se trouve dans une procédure, une fonction, ou dans le programme principal.
Données en entrée	Le nœud dont le contexte doit être déterminé.
Données en sortie	Le type de nœud du contexte englobant (DeclFonction, DeclProcédure ou Programme), ou Erreur.
Point de départ	Nœud.
Point d'arrivée	La donnée en sortie ou la racine.
Description du parcours	Ce pilote se contente de remonter dans l'arbre jusqu'à trouver le type de sous-programme englobant, ou la racine.
Utilise	Néant.

Comportement par type de noeud :

Par défaut	Monte dans l'arbre. → VersPère.
DeclFonction DeclProcédure Programme	→ Fini.
Compalgo	→ Erreur.

SolveurDeVariable

Description :

Acteur ou Evaluateur	***SolveurDeVariable***
Rôle	Va chercher la donnée concrète voulue à travers les accès à des éléments de tableau ou à des champs d'enregistrement.
Contexte	Utilisé pour évaluer les expressions et déterminer la donnée sujette à une affectation.
Données en entrée	Le nœud représentant la variable recherchée.
Données en sortie	Une Variable< ?>.

Acteur ou Evaluateur	***SolveurDeVariable***
Point de départ	Nœud.
Point d'arrivée	Relatif au contenu du nœud.
Description du parcours	Recherche la variable de proche en proche en entrant récursivement dans les nœuds AccèsChamp et AccèsTableau.
Utilise	InterpréteurExpression SolveurDeVariable

Comportement par type de noeud :

Par défaut	Ce cas n'est pas censé se produire. → Erreur.
UtilVariable	Point d'entrée. → Dedans.
Id	Cherche la variable correspondante dans le glossaire courant. → Fini : si la variable a été trouvée. → Erreur : sinon.
AccèsTableau	Récupère le (indice)-ième élément dans la variable de type tableau. → Fini : si tout va bien. → N/A : sinon. Une exception de l'interpréteur est déclenchée.
AccèsChamp	Récupère le champ nommé dans la variable de type enregistrement. → Fini : si tout va bien. → Erreur : sinon.

Fiche de procédure
—
Maintenance de la grammaire formelle

But

Apporter des modifications aux règles de grammaire du fichier de spécification de *parser* de Coco/R.

Prérequis

- Fichiers :
  - Documents/
    - grammaire_compalgo_LL1.txt : grammaire formelle de Compalgo.
  - test_compalgo_coco-r/fr/irit/sig/compalgo/parser/
    - compalgo.atg : fichier de spécification de parser de Coco/R,
    - compalgo.atg.header : entête du fichier de spécification,
    - gen_Coco-R.cygbash : script de régénération de compalgo.atg
    - conflit_resolu.sh : script de validation des fichiers générés
- Outils :
  - Interpréteur de commandes bash
- Connaissances :
  - Diff3 : utilisation et formattage des conflits

Procédure

- Éditer la grammaire formelle pour modifier les règles de grammaire du langage.
- Éditer l'entête du fichier de spécification pour modifier les imports ou les ajouts dans la classe Parser.
- Éditer le script gen_Coco-R.cygbash pour modifier la déclaration des commentaires et des caractères de retour de ligne à ignorer.
- Lancer depuis le répertoire qui le contient le script gen_Coco-R.cygbash dans un shell bash.
- Si aucun conflit n'est détecté, le fichier compalgo.atg est mis à jour.
- Si au moins un conflit a été détecté :
- Éditer le fichier compalgo.atg.nouveau. Il contient le texte de compalgo.atg et les parties conflictuelles sont délimitées par des <<< === ||| >>>. Éliminer manuellement ces conflits.
- Valider la résolution des conflits en lançant le script conflit_resolu.sh. Le fichier compalgo.atg contient alors la nouvelle version de l'entête et de la grammaire.

Fiche de procédure
—
Création d'une bibliothèque native

But

Augmenter la bibliothèque native de Compalgo fournie aux étudiants.

Prérequis

- Fichiers :
  - compalgo.jar : contient la librairie Java de Compalgo.
- Connaissances :
  - Langage Java.
  - Langage algorithmique.

Procédure

- Créer le fichier de déclarations des sous-programmes en langage algorithmique MaNouvelleBibliothèque.spec.
- Les types Entier, Booléen, Chaîne, Réel, et Caractère sont autorisés en paramètres des sous-programmes et en valeur de retour des fonctions. Les autres types ne sont pas reconnus.
- Créer une classe **publique** Java du même nom (ici, MaNouvelleBibliothèque) dans le paquetage **fr.irit.sig.compalgo.interpréteur.bibliothèque**.
- Chaque sous-programme défini dans le fichier .spec correspond à une méthode **publique** de même nom dans la classe Java.
- Les arguments passés en **entrée** dans le langage algorithmique de type Entier, Réel, Booléen, Chaîne, Caractère sont reçus par Java comme étant respectivement des InEntier, InRéel, InBooléen, InChaîne, InCaractère.
- Les arguments passés en **sortie** et **màj** dans le langage algorithmique de type Entier, Réel, Booléen, Chaîne, Caractère sont reçus par Java comme étant respectivement des OutEntier, OutRéel, OutBooléen, OutChaîne, OutCaractère.
- Les valeurs retournées par Java doivent être de type Integer, Float, Character, String, ou Boolean. Les autres types de retour ne sont pas reconnus.
- Ajouter la classe compilée au classpath de Java ou dans compalgo.jar.
- Ajouter le fichier .spec au répertoire standard de déploiement de la bibliothèque de Compalgo.