

PLAN DU COURS 7

Intégration (pour *inlining*) de fonctions

- Préliminaires
- Normalisation
- Graphe d'appel
- Intégration
- Réflexions

ILP4 = ILP3 + intégration.

Master d'informatique 2007-2008

Spécialité STL

« Implantation de langages »

ILP – MI016

Cours 7

C. Queinnee^a

^a<http://www-spi.lip6.fr/~queinnee/>

Mais les variables peuvent être modifiées :

```
function f (x y) {      { z = 34;
  x = x + y;             y = ({ x = 1;
    in 2*x;              y = z;
  }                      }
  { z = 34;              x = x + y;
    y = f(1, z);         in 2*x;
  }                      }) ;
}
```

Il faut donc recréer les liaisons correspondantes pour les affectations.

INTÉGRATION

L'optimisation reine : 20% d'amélioration.

Transformation de programme. On remplace des appels de fonctions par leur corps, après substitution de leurs variables par les paramètres d'appel.

```
function f (x y) {      { z = 34;
  let t = x + y          y = ({ let t = 1 + z;
    in 2*t;              in 2*t;
  }                      }) ;
}
{ z = 34;
  y = f(1, z);
}
```

INTÉRÊTS

- Supprimer des invocations (sauvegarde registres, allocation en pile, restauration registres, responsabilité des registres (appelant ou appelé ?) etc.)
- Rapprocher des fragments de code indépendants (surtout avec précalculs statiques (*constant folding*) et suppression du code mort (*dead code elimination*)).

PROBLÈMES

- Respect des portées
- Équivalence expression/instruction
- Prévention des conflits de noms

Mais attention aux portées :

```

z = 2;
function f (x y) {
    return x + y + z;
}

{ z = 2;
  { z = 34;
    y = ( { x = 1;
           y = z;
         }
        );
  }
}

{ z = 34;
  y = f(1, z);
}
```

Capture de la variable locale **z** par la référence libre à la variable globale **z** depuis la fonction **f**.

```

function f(x, y) {
    if (x > 1 ) {
        x + y
    } else {
        x
    }
}

{ z = 2;
  print(3 + f(z, t));
}
```

```

z = 2;

function f (x1 y1) {
    let z1 = x1 + y1 + z;
    in 2*z1;
}

z = 2;
{ z0 = 34; // alpha-conversion :
  x0 = 5;
  y0 = ({
    // liaisons :
    let x1 = 1
    and y1 = x0 + z0 in
    // corps :
    let z1 = x1 + y1 + z;
    in 2*z1;
  })
}

```

9

SOLUTION

Renommage de toutes les variables locales (alpha-conversion) :

```

z = 2;

function f (x y) {
    let z = x + y + z;
    in 2*z;
}

z = 2;
function f (x1 y1) {
    let z1 = x1 + y1 + z;
    in 2*z1;
}

{ z = 34;
  x = 5;
  y = f(1, x+z);
}

{ z0 = 34;
  x0 = 5;
  y0 = f(1, x0+z0);
}

```

12

ÉQUIVALENCE EXPRESSION/INSTRUCTION

Ressource: [Grammars/grammar4.mc](#)

Nouvelle grammaire où toute instruction est aussi une expression.

Je n'ai pas réussi à la définir incrémentiellement à partir des précédentes !

```

programme4 = element programme4 {
  definitionEtExpressions
}

definitionEtExpressions =
  definitionFonction *,
  expression +

```

11

PAQUETAGES POUR ILP4

Le super-paquetage `fr.upmc.ilp.ilp4` contient les paquetages

habituels :

<code>fr.upmc.ilp.ilp4.interfaces</code>	interfaces diverses
<code>fr.upmc.ilp.ilp4.ast</code>	AST et analyses statiques
<code>fr.upmc.ilp.ilp4.runtime</code>	bibliothèque d'interprétation
<code>fr.upmc.ilp.ilp4.cgen</code>	compilation vers C

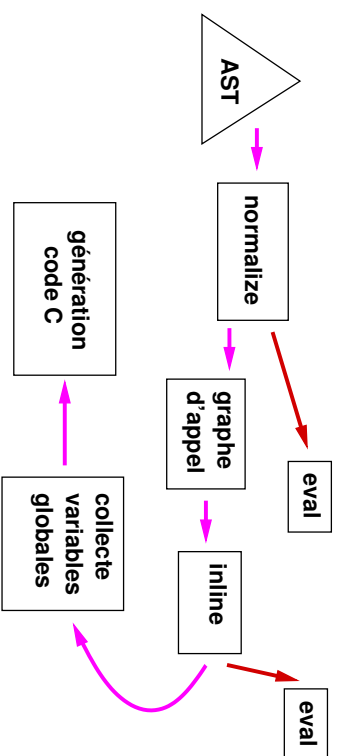
Grammaire *Grammars/grammar4.mc*

Nouveau patron *CtemplateTest3.c*

Programmes ILP4 additionnels *Grammars/Samples/*-4.xml*

PASSES DE TRAITEMENT

Quatre analyses statiques dont la normalisation des expressions de l'AST.



PRÉVENTION DES CONFLITS DE NOMS

- Deux références à une même variable (locale ou globale) sont représentées par le même objet en mémoire.
- Taxonomie des variables locales, globales, globales fonctionnelles, prédéfinies.
- Les séquences d'une seule expression sont normalisées à cette seule expression.
- Vérification de l'arité des appels aux fonctions globales.

L'identification des variables :

- améliore la comparaison (et notamment la vitesse de l'interprète (comme en Lisp avec la notion de symbole))
- réalise l'alpha-conversion.

NORMALISATION

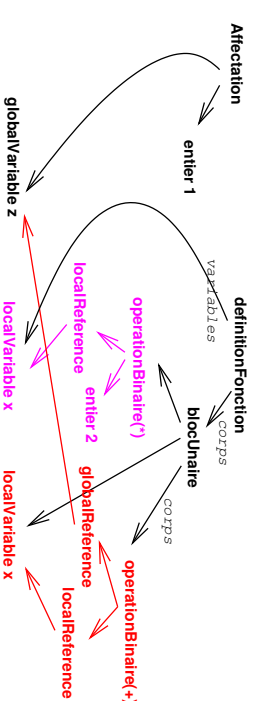
Partage physique des objets représentant les variables.

Taxonomie des variables locales, globales, globales fonctionnelles, prédéfinies.

Au passage, les séquences d'une seule expression sont normalisées à cette seule expression. Vérification de l'arité des appels aux fonctions globales.

```

z = 1;
function f(x) {
  let x = 2*x
  in z+x
}
  
```



expression =

- | | |
|-------------|---------------------|
| alternative | sequence |
| blocUnaire | bloclocal |
| boucle | try |
| affectation | invocation |
| constante | variable |
| operation | invocationPrimitive |

NORMALISATION

La normalisation est encore un parcours avec deux environnements :

- l'environnement lexical `INormalizeLexicalEnvironment`
- l'environnement global `INormalizeGlobalEnvironment`

Chaque nœud de l'AST procure des méthodes pour ces différents passes.

```
public interface IAST4 {
    extends IAST2 {

        IAST4 normalize (INormalizeLexicalEnvironment lexenv,
                        INormalizeGlobalEnvironment common )
        throws NormalizeException;

        void findInvokedFunctions ()
        throws FindngInvokedFunctionsException;

        Set<IAST4globalFunctionVariable> getInvokedFunctions ();

        void inline () throws InliningException;
    }
}
```

COMPARAISON

Comparaison physique plutôt que structurelle :

```
// depuis LexicalEnvironment
public Object lookup (final IVariable otherVariable)
    throws EvaluationException {
    if ( variable == otherVariable ) {
        return value;
    } else {
        return next.lookup(otherVariable);
    }
}
```

locale	CEASTlocalVariable	CEASTlocalAssignment	CEASTInvocation
globale	CEASTglobalVariable	CEASTglobalAssignment	CEASTglobalInvocator
pré définie	CEASTPredefinedVariable		CEASTprimitiveInvocat

Les variables globales fonctionnelles permettent de retrouver la définition de fonction ainsi nommée.

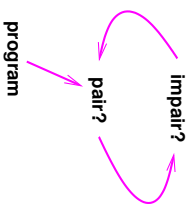
TAXONOMIE DES VARIABLES

- Variables locales `CEASTlocalVariable`
- Variables globales `CEASTglobalVariable`
- Noms de fonctions globales `CEASTglobalFunctionVariable`
- Noms de fonctions pré définies `CEASTPredefinedVariable`

```
function foo (x) {
    ...
}

let x = 3
in x := 4;
g := foo(x);
foo := g;

CEASTlocalAssignment, localReference, localVariable
CEASTglobalAssignment
CEASTglobalVariable, CEASTglobalFunction Variable
```



```
function pair? (n) {
  if ( n == 0 ) {
    return true;
  } else {
    return impair?(n - 1);
  }
}

function impair? (n) {
  if ( n == 0 ) {
    return false;
  } else {
    return pair?(n - 1);
  }
}
```

CALCUL DU GRAPHE D'APPELS

CALCUL FERMETURE TRANSITIVE

Passe 1

```
pair? → impair?
impair? → pair?
program → pair?
```

Passe 2

```
pair? → impair?, pair?
impair? → pair?, impair?
program → pair?, impair?
```

Passe 3 : point fixe

```
pair? → impair?, pair? →récursive
impair? → pair?, impair? →récursive
program → pair?, impair?
```

Après calcul du graphe d'appels,

- ☐ on n'intègre que les fonctions non récursives.
- ☐ Par contre, on intégrera toutes les fonctions non récursives.

// NOTA: la précédente méthode change la collection que l'on
// est en train d'inspecter ce qui pose des problèmes
// d'accès simultanés à cette collection d'où l'emploi d'un
// toArray() plus haut.

```
    }  
}  
// Savoir ce qu'invoque le programme est de peu d'utilité!  
findAndAddJointlyInvokedFunctions(getBody());  
}  
protected static final IAST4GlobalFunctionVariable[] IAST4GFV_EMPTY_ARRAY =  
    new IAST4GlobalFunctionVariable[0];
```

Resource: [Java/src/fr/upmc/ilp/ilp-4/asl/CEASTprogram.java](http://upmc.fr/upmc/ilp/ilp-4/asl/CEASTprogram.java)

```
public void findInvokedFunctions ()  
throws FindingInvokedFunctionsException {  
    IAST4FunctionDefinition[] definitions = getFunctionDefinitions();  
    for ( int i = 0 ; i<definitions.length ; i++ ) {  
        definitions[i].findInvokedFunctions();  
    }  
    boolean shouldContinue = true;  
    while ( shouldContinue ) {  
        shouldContinue = false;  
        for ( int i = 0 ; i<definitions.length ; i++ ) {  
            final IAST4FunctionDefinition currentFunction = definitions[i];  
            for ( IAST4GlobalFunctionVariable gv :  
                currentFunction.getInvokedFunctions() )  
                .toArray(IAST4GFV_EMPTY_ARRAY) ) {  
                // currentFunction invoque gv donc elle invoque  
                // (indirectement) les fonctions qu'invoque gv.  
                final IAST4FunctionDefinition other =  
                    gv.getFunctionDefinition();  
                shouldContinue = shouldContinue  
                    || currentFunction.addInvokedFunctions(  
                        other.getInvokedFunctions());  
            }  
        }  
    }  
}
```

28

INTÉGRATION

Pour tout nœud `CEASTGlobalInvocation` et si la fonction invoquée n'est pas réursive : l'intégrer.

Le résultat de l'intégration est stocké dans le champ `inlined`.

L'ordre d'intégration importe peu à condition de passer partout mais **seulement une seule fois**.

```
public void inline () throws InliningException {  
    if ( this.inlined != null ) {  
        return;  
    } else {  
        // On analyse les arguments!  
        for ( IAST4expression arg : getArguments() ) {  
            arg.inline();  
        }  
        if ( getFunction() instanceof CEASTGlobalFunctionVariable ) {  
            final IAST4GlobalFunctionVariable gv =  
                (CEASTGlobalFunctionVariable) getFunction();  
            final IAST4FunctionDefinition function = gv.getFunctionDefinition();  
            if ( function.isRecursive() ) {
```

- `findInvokedFunctions` calcule un sous-ensemble des fonctions invoquées.
- `getInvokedFunctions` renvoie ce sous-ensemble.

MOCHE d'avoir ajouté un champ `invokedFunctions` à chaque expression !

GÉNÉRATION DE CODE

Les fonctions non récursives sont éliminées jusqu'intégrées.

Pour les invocations intégrées, on utilise l'expression dans le champ `inlined`.

```
// On n'intègre pas les fonctions récursives!
return;
} else {
    // La fonction a toutes les qualités requises, on l'intègre!
    this.inlined = new CEASTLocalBlock(
        (IAST4variable[]) function.getVariables(),
        getArguments(),
        (IAST4expression) function.getBody());
    // inlined.inline(); // déjà fait quand fonction fut analysée.
    return;
}
} else {
    // La fonction est le résultat d'un calcul, on ne l'intègre pas.
    return;
}
}
```

TECHNIQUES JAVA

- Inversion expression/instruction
 - ▶ Usage plus fin des destinations
- Nouvel analyseur syntaxique plus générique
- Délégation
- Méthodes pour `inline` et annotations
- Visiteur

CONCLUSIONS SUR INTÉGRATION

- L'intégration supprime les instructions d'appel donc améliore la vitesse
- mais augmente la taille du code donc diminue l'efficacité de la mémoire virtuelle.
- Il est possible de déplier finement les fonctions récursives.
- On peut prendre en compte l'augmentation de taille (globale ou locale) et en faire un critère d'intégration.
- On peut ajouter des déclarations `inline` pour indiquer les intégrations utiles.

Sous-Typage

Typier `o.message(arguments)` signifie que l'on sait statiquement que `o` est d'un type qui implante une méthode pour `message` et que les arguments de cette méthode sont appropriés.

A sous-typé de $B \equiv$ un A peut partout remplacer un B sans perturber le typage.

La classe A étend la classe B (en Java) implique que A est sous-typé de B.

La classe A implante l'interface B (en Java) implique que A est sous-typé de B.

L'interface A étend l'interface B (en Java) implique que A est sous-typé de B.

IAST4 ET IAST2

Toutes les catégories syntaxiques ont une interface étendue IAST4*. Elles dérivent toutes d'IAST4 (pour les nouvelles méthodes d'ILP4) et de l'interface équivalente IAST2*.

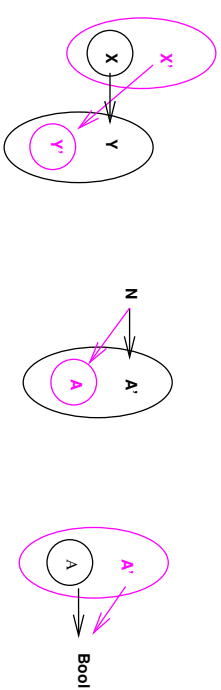
```

ilp4. IAST4
      IAST4program
      IAST4functionDefinition
      IAST4expression
      IAST4instruction
      IAST4while
      IAST4variable
      IAST4localVariable
      IAST4globalVariable
      IAST4globalFunctionVariable
      IAST4predefinedVariable
  
```

```

      étend IAST2expression
      étend IAST2instruction
      étend IAST2while
  
```

Méthode unifiée `compile`. Du coup `compileExpression` et `compileInstruction` sont rendus obsolètes (`@Deprecated`).



Cas des tableaux : si $A \subset A'$ alors $N \rightarrow A$ sous-typé de $N \rightarrow A'$ donc $A[]$ sous-typé de $A'[]$.

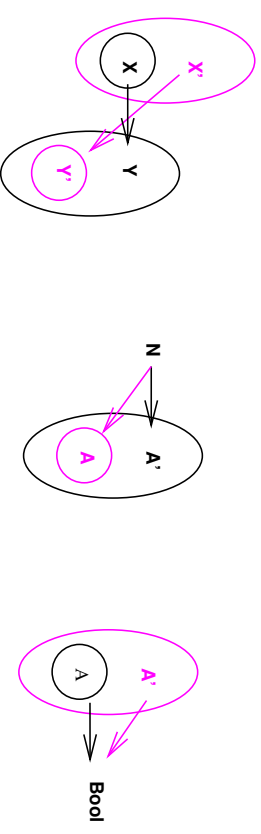
Attention, en Java, le type d'un tableau est statique et ne dépend pas du type de ses éléments :

```

Point[] ps = new Point[] { new PointColore() };
// PointColore[] pcs = (PointColore[]) ps // erreur!
PointColore[] pcs = new PointColore[ps.length];
for ( int i=0 ; i<pcs.length ; i++ ) {
    pcs[i] = (PointColore) ps[i];
}
  
```

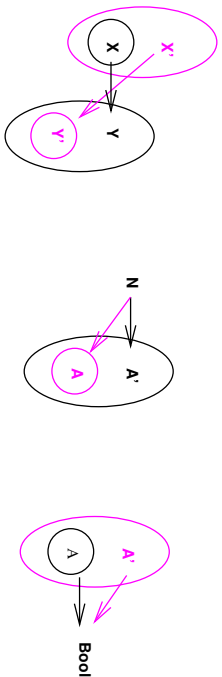
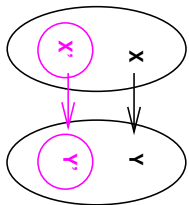
CONTRAVARIANCE/COVARIANCE

Une fonction $X' \rightarrow Y'$ est un sous-typé de $X \rightarrow Y$ ssi $X \subset X'$ et $Y' \subset Y$.



NB : J'utilise l'inclusion ensembliste comme notation pour le sous-typage.

Exemple de covariance :



Cas des ensembles : Si $A \subset A'$ alors $A' \rightarrow \text{Bool}$ sous-type de $A \rightarrow \text{Bool}$ mais $\text{Set}\langle A' \rangle$ n'est pas en Java un sous-type de $\text{Set}\langle A \rangle$ ni vice-versa. Par

contre $\text{Set}\langle A \rangle$ est un sous-type de $\text{Collection}\langle A \rangle$.

```
Set<Point> ss = new HashSet<Point>();
ss.add(new PointColore());
Set<PointColore> spc = new HashSet<PointColore>();
// spc.add(new Point()); // évidemment illégal!
// spc.addAll(ss); // illégal!
// ss = (Set<Point>) spc; erreur!
// spc = (Set<PointColore>) ss; erreur!
ss.addAll(spc);
```

```
public class CEASTAssignment // ilp4.ast.CEASTExpression
extends CEASTExpression
implements IAST4Assignment {

    public CEASTAssignment (final IAST4variable variable,
                             final IAST4expression value) {
        ...
    }

    @ILPvariable // annotation
    public IAST4variable getVariable () {
        return CEAST.narrowToIAST4variable(...);
    }

    @ILPexpression // annotation
    public IAST4expression getValue () {
        return CEAST.narrowToIAST4expression(...);
    }
}
```

EN ILP4

Certaines méthodes retournent des types plus précis que suggérés dans l'interface : des instances satisfaisant IAST4* plutôt qu'IAST2*.

Certaines méthodes s'attendent à des types plus précis qu'imposés par l'interface : des instances satisfaisant IAST4* plutôt qu'IAST2*. Il faut donc les convertir et pour aider à la mise au point existent les méthodes statiques `CEAST.narrowToIAST4*()`.

```
public interface IAST4Assignment
extends IAST4expression, IAST2Assignment {
    IAST4variable getVariable (); // raffinement
    IAST4expression getValue (); // raffinement
}
```

```
public class CEASTassignment // ilp4.ast.CEASTexpression
extends CEASTexpression
implements IAST4assignment {

    public CEASTassignment (final IAST4variable variable,
                             final IAST4expression value) {
        this.delegate =
            new fr.upmc.ilp.ilp2.ast.CEASTassignment(
                variable, value );
    }
    private fr.upmc.ilp.ilp2.ast.CEASTassignment delegate;

    @Override
    public IAST2assignment getDelegate () {
        return this.delegate;
    }

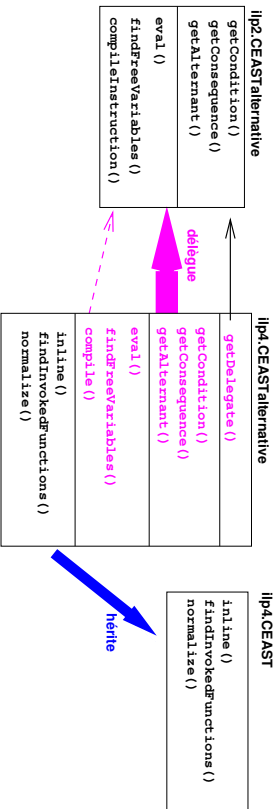
    @ILPvariable
    public IAST4variable getVariable () {
        return CEAST.narrowToIAST4variable(getDelegate().getVariable());
    }

    @ILPexpression
    public IAST4expression getValue () {
        return CEAST.narrowToIAST4expression(getDelegate().getValue());
    }
}
```

41

DÉLÉGATION

Les nouvelles classes `ilp4.CEAST*` réalisent les interfaces `IAST4*` mais héritent d'`ilp4.CEAST` pour partager le code de certaines nouvelles méthodes, elles n'héritent donc pas d'`ilp[23].CEAST*` pourtant les comportements qui s'y trouvent (`eval`, `compile`, `findFreeVariables`) sont ceux que l'on désire.



44

NOUVEL ANALYSEUR PAR RÉFLEXION

Toutes les classes d'AST ont une méthode statique `create` (`Element`, `IParser`) pour analyser un noeud DOM.

```
public class CEASTParser extends AbstractParser {
```

```
    public CEASTParser ()
        throws CEASTParseException {
        this.parsers = new HashMap<String, Method>();
        addParser("programme4", CEASTProgram.class);
        addParser("alternative", CEASTAlternative.class);
        ...
    }
    private final HashMap<String, Method> parsers;

    /** Ajout d'une caractéristique a ILP. Lorsque l'element XML nommé
     * name est lu, la methode clazz.create(e, parser) sera invoquée. */
    public void addParser (String name, Class clazz)
        throws CEASTParseException {
        try {
            final Method method = clazz.getMethod("parse",
                }

        public abstract class CEASTexpression
        extends CEAST
        implements IAST4expression {
            public abstract IAST2 getDelegate ();

            public Object eval (final ILexicalEnvironment lexenv,
                               final ICommon common)
                throws EvaluationException {
                return getDelegate().eval(lexenv, common);
            }

            public void compile (final StringBuffer buffer,
                                final IGenLexicalEnvironment lexenv,
                                final IGenEnvironment common,
                                final IDestination destination)
                throws CgenerationException {
                final IAST2expression delegate = (IAST2expression) getDelegate();
                delegate.compileExpression(buffer, lexenv, common, destination);
            }
        }
    }
}
```

```

        case Node.ELEMENT_NODE: {
            final Element e = (Element) n;
            final String name = e.getTagName();

            if ( parsers.containsKey(name) ) {
                final Method method = parsers.get(name);
                try {
                    Object result = method.invoke(null, new Object[] {e, this});
                    return CEAST.narrowToIAST4(result);
                } catch (IllegalArgumentException exc) {
                    throw new CEASTParseException(exc);
                } catch (IllegalAccessException exc) {
                    throw new CEASTParseException(exc);
                } catch (InvocationTargetException exc) {
                    Throwable t = exc.getTargetException();
                    if ( t instanceof CEASTParseException ) {
                        throw (CEASTParseException) t;
                    } else {
                        throw new CEASTParseException(exc);
                    }
                }
            } else {
                }
            }
        }
    }
}

```

```

        new Class[] { Element.class, IParser.class } );
        if ( ! Modifier.isStatic(method.getModifiers()) ) {
            final String msg = "Non static parse() method!";
            throw new CEASTParseException(msg);
        };
        parsers.put(name, method);
    } catch (SecurityException e1) {
        final String msg = "Cannot access parse() method!";
        throw new CEASTParseException(msg);
    } catch (NoSuchMethodException e1) {
        final String msg = "Cannot find parse() method!";
        throw new CEASTParseException(msg);
    }
}

public IAST4 parse (final Node n)
throws CEASTParseException {
    switch ( n.getNodeType() ) {
        case Node.DOCUMENT_NODE: {
            final Document d = (Document) n;
            return this.parse(d.getDocumentElement());
        }
    }
}

```

ANALYSES STATIQUES

La détermination des fonctions invoquées est un simple parcours de l'AST. À partir de chaque noeud, on explore les sous-arbres contenant des expressions.

Pour `findFreeVariables()`, on ne traitait spécialement que les `CEASTGlobalFunctionVariable` en remplissant un `Set<IAST2variable>`.

Pour `inline()` et `findInvokedFunctions()`, on ne traite spécialement que les `CEASTGlobalInvocation`. Pour `inline()` on renseigne le champ `inlined`. Pour `findInvokedFunctions()` on remplit un `Set<IAST4globalFunctionVariable>`.

Pour `normalize()` on reconstruit un AST équivalent.

```

        final String msg = "Unknown element name: " + name;
        throw new CEASTParseException(msg);
    }
    }
    default: {
        final String msg = "Unknown node type: " + n.getNodeName();
        throw new CEASTParseException(msg);
    }
    }
}

```

ANNOTATIONS

```
package fr.upmc.ilp.annotation;
import java.lang.annotation.*;

@Documented
@Retention(RetentionPolicy.RUNTIME)
@Inherited
@Target(ElementType.METHOD)
public @interface ILPExpression {

    /** Indique si la valeur obtenue par la methode peut etre null. */
    boolean neverNull () default true;

    /** Indique si la valeur obtenue est en fait un vecteur d'expressions. */
    boolean isArray () default false;
}
```

Malheureusement pas d'héritage d'annotation !

OÙ SONT LES SOUS-EXPRESSIONS ?

Dans tous ces cas, on se demande où sont les sous-expressions composant l'expression. On identifie les accesseurs menant à des sous-expressions avec l'annotation `@ILPExpression` et ceux menant à des variables avec l'annotation `@ILPVariable`. On pourrait avoir une méthode par défaut qui arpente tout nœud et ses sous-expressions.

```
final Object result =
    m.invoke(this, EMPTY_ARGUMENT_ARRAY);
if ( e.neverNull() || result != null ) {
    final IAST4expression component =
        CEAST.narrowToIAST4expression(result);
    this.findAnddJoinToInvokedFunctions(component);
};
```

```
    }
    } catch (IllegalArgumentException e) {
        throw new FindingInvokedFunctionsException(e);
    } catch (IllegalAccessException e) {
        throw new FindingInvokedFunctionsException(e);
    } catch (InvocationTargetException e) {
        throw new FindingInvokedFunctionsException(e);
    }
}

private final Set<IAST4globalFunctionVariable> invokedFunctions;
// NOTE: un tel champ par instance est dispendieux!
```

DÉTERMINATION DES INVOCATIONS

La méthode par défaut est :

```
public void findInvokedFunctions ()
throws FindingInvokedFunctionsException {
    final Class clazz = this.getClass();
    for ( final Method m : clazz.getMethods () ) {
        try {
            final ILPExpression e = m.getAnnotation(ILPExpression.class);
            if ( e != null ) {
                // FIXME: mettre en cache cette recherche!
                if ( e.isArray () ) {
                    final Object[] results = (Object[])
                        m.invoke(this, EMPTY_ARGUMENT_ARRAY);
                    for ( Object result : results ) {
                        if ( e.neverNull() || result != null ) {
                            final IAST4expression component =
                                CEAST.narrowToIAST4expression(result);
                            this.findAnddJoinToInvokedFunctions(component);
                        }
                    }
                } else {
                    m.invoke(this, new Object[] {});
                }
            }
        } catch (Exception e) {
            // ...
        }
    }
}
```

```
// en ILP2
public void compileInstruction (final StringBuffer buffer,
                               final IGenLexicalEnvironment lexenv,
                               final IGenEnvironment common,
                               final IDestination destination)
throws CGenerationException {
    buffer.append(" if ( ILP_isEquivalentToTrue( ") ;
    getCondition().compileExpression(buffer, lexenv, common);
    buffer.append(" ) ) {\n");
    getConsequent().compileInstruction(buffer, lexenv, common, destination);
    buffer.append("; \n");
    getAlternant().compileInstruction(buffer, lexenv, common, destination);
    buffer.append("; \n");
}
```

PLUS DE DESTINATION

L'inversion expression/instruction fait que toute expression ILP peut maintenant contenir des fragments qui ne peuvent être compilés que sous forme d'instructions C.

***\xrightarrow{d}** alternative*

```
if ( ILP_isEquivalentToTrue(  $\overline{condition}$  ) ) {
     $\overline{d}$  consequence ;
} else {
     $\xrightarrow{d}$  alternant ;
}
```

```
// en ILP4
public void compile (final StringBuffer buffer,
                    final IGenLexicalEnvironment lexenv,
                    final IGenEnvironment common,
                    final IDestination destination)
throws CGenerationException {
    final IAST4Variable tmp = CEASTLocalVariable.generateVariable();
    buffer.append("{ ");
    tmp.compileDeclaration(buffer, lexenv, common);
    final IGenLexicalEnvironment bodyLexenv = lexenv.extend(tmp);
    getCondition().compile(buffer, lexenv, common, new AssignDestination(tmp));
    buffer.append("; \n");
    tmp.compile(buffer, bodyLexenv, common, NoDestination.create());
    buffer.append(" } ) {\n");
    getConsequent().compile(buffer, lexenv, common, destination);
    buffer.append("; \n");
    getAlternant().compile(buffer, lexenv, common, destination);
    buffer.append("; \n } \n");
}
```

***\xrightarrow{d}** alternative*

```
{ ILP_Object tmp;
     $\xrightarrow{tmp}$  condition;
    if ( ILP_isEquivalentToTrue( tmp ) ) {
         $\overline{d}$  consequence ;
    } else {
         $\xrightarrow{d}$  alternant ;
    }
}
```

```
public class XMLwriter implements IAST4visitor {
    public void visit (IAST4alternative iast) {
        ...
        iast.getCondition().accept(this);
        iast.getConsequent().accept(this);
        iast.getAlternant().accept(this);
        ...
    }
    public void visit (IAST4assignment iast) {
        ...
        iast.getVariable().accept(this);
        iast.getValue().accept(this);
        ...
    }
}
```

VISITEUR

```
57 public interface IAST4visitor {
    void visit (IAST4alternative iast);
    void visit (IAST4assignment iast);
    ...
}
public interface IAST4vistable {
    void accept (IAST4visitor visitor);
}
```

POUR LA PROCHAINE FOIS

- Tester ILP4 sur des exemples
- Lire le code de ce grand saut technologique
- Refaire la collecte des variables globales comme un visiteur.

Mais la discrimination n'a plus à être écrite :

```
// dans CEASTalternative:
public void accept (IAST4visitor visitor) {
    visitor.visit(this);
}
// dans CEASTassignment:
public void accept (IAST4visitor visitor) {
    visitor.visit(this);
}
```

L'état du parcours est stocké dans le visiteur, voir exemple `XMLwriter`. La différence avec la méthode `analyse` d'ILP1 est qu'il n'y a plus besoin d'écrire le code de discrimination, on a mis à profit le mécanisme d'envoi de message pour ce faire (au prix d'une duplication de code toutefois).