Revision: 1.16

Master d'informatique 2014-2015
Spécialité STL
« Implantation de langages »
DLP – 4I501
épisode ILP2

C.Queinnec

Buts

- ILP2 = ILP1 +
  - fonctions globales
  - boucle
  - affectation
- Compilation par destination
- Analyse statique

Plan du cours 5

- Présentation d'ILP2
- Syntaxe
- Sémantique
- Génération de C
- Nouveautés techniques Java

Adjonctions

ILP2 = ILP1 + définition de fonctions globales + boucle `while` + affectation + bloc local n-aire.

```
let deuxfois x =
   x + x;;
let fact n = if n = 1 then 1 else n * fact (n-1);;
let x = 1 and y = "foo" in
   while x < 100 do
      x := deuxfois (fact(x));
      y := deuxfois y;
   done
   y;;
```


Organisation des fichiers

Tout dans le super-paquetage : `fr.upmc.ilp.ilp2`

```
ilp2.interfaces // interfaces diverses
ilp2.ast        // AST (et analyse syntaxique)
ilp2.runtime    // bibliotheque d'interpretation
ilp2.cgen       // Compilation vers C
```

Grammaire : `Grammars/grammar2.rnc`
Programmes ILP2 additionnels : `Grammars/Samples/*-2.xml`

Organisation Java

```
      // En ilp2.interfaces
.IAST2<Exc extends Exception>
  .IAST2program<Exc extends Exception>
  .IAST2functionDefinition<Exc extends Exception>
  .IAST2instruction<Exc extends Exception>
     .IAST2alternative<Exc extends Exception>
        .IAST2expression<Exc extends Exception>
           .IAST2Constant<Exc extends Exception>
           .IAST2assignment<Exc extends Exception>
           .IAST2reference<Exc extends Exception>
     .IAST2while<Exc extends Exception>
.IAST2variable
IDestination  // Robustesse
```

IAST2 = IAST + eval() + findGlobalVariables()
IAST2instruction = IAST2 + compileInstruction()
IAST2expression = IAST2instruction + compileExpression()
Tous les IAST2* sont génériques sur l'exception signalable.

Généricité

Outre les IAST2* génériques,

```
public interface IEnvironment<V> {
  boolean isPresent (V variable);
  boolean isEmpty ();
  IEnvironment<V> getNext ();
  V getVariable ();
}

// Heritage monomorphe
public interface ILexicalEnvironment
extends IEnvironment<IAST2variable> {
  ...
  ILexicalEnvironment extend (
     IAST2variable variable, Object value);
}
```

mais `instanceof X<Y>` plus possible !

Destinations

Pour renforcer le typage donc la détection d'erreurs :

```
ilp2.cgen.NoDestination
         ReturnDestination
         VoidDestination
         AssignDestination
```

Hiérarchie pour AST

```
ilp2.ast.CEAST
           CEASTprogram
           CEASTfunctionDefinition
           CEASTinstruction
             CEASTalternative
             CEASTexpression
               CEASTinvocation
                 CEASTprimitiveInvocation
               CEASTreference                  //
         CEASTvariable
           CEASTpredefinedVariable
```


Ajouts à bibliothèque d'exécution

```
ilp2.runtime.UserFunction
             UserGlobalFunction
```

Autres paquetages presque vides ou obtenus par héritage d'ILP1

```
ilp2.interfaces    800 loc
ilp2.ast          2600 loc
ilp2.runtime       600 loc
ilp2.cgen          400 loc
ilp2               300 loc
```


Grammaire

Extensibilité des schémas RelaxNG avec `include` et `|=`

```
include "grammar1.rnc"
start |= programme2
instruction |= blocLocal
instruction |= boucle
expression |= affectation
expression |= invocation

programme2 = element programme2 {
   definitionFonction *,
   instructions
}
definitionFonction = element definitionFonction {
   attribute nom      { xsd:Name - ( xsd:Name { patter
   element variables { variable * },
   element corps      { instructions }
}
```



```
blocLocal = element blocLocal {
   element liaisons {
      element liaison {
        variable,
        element initialisation {
          expression
        }
      } *
   },
   element corps { instructions }
}
boucle = element boucle {
   element condition { expression },
   element corps     { instructions }
}
affectation = element affectation {
   attribute nom  { xsd:Name - ( xsd:Name { pattern =
   element valeur { expression }
}
```

Analyseur

Les classes de l'AST sont des CEAST* (qui implantent les IAST2*). Elles ont des méthodes `eval` et `compileQuelqueChose`. Elles procurent aussi une méthode statique (pour rapprocher l'analyse syntaxique de la définition de l'AST) :

```
public static CEAST* parse (
    Element e,
    IParser<CEASTparseException> parser)
  throws CEASTparseException;
```

Pourquoi `static` ?

L'analyseur prend une fabrique à sa construction.

```
public class CEASTParser extends AbstractParser {

  public IAST2 parse (final Node n)
    throws CEASTparseException {
    switch ( n.getNodeType() ) {

    case Node.ELEMENT_NODE: {
      final Element e = (Element) n;
      final String name = e.getTagName();
      switch(name) {
      case "alternative":
        return CEASTalternative.parse(e, this);
      case "sequence":
        return CEASTsequence.parse(e, this);
      ...
```


Hiérarchie des analyseurs

```
public abstract class AbstractParser
implements IParser<CEASTparseException> {
   ...
}
public interface IParser<Exc extends Exception> {
  IAST2Factory<Exc> getFactory ();

  IAST2program<Exc> parse (Document n) throws Exc;

  IAST2<Exc> parse (Node n) throws Exc;

  List<IAST2<Exc>> parseList (NodeList nl) throws Ex
  ...
```


Analyseur des alternatives

```
public static CEASTalternative<CEASTparseException>
    Element e, IParser<CEASTparseException> parser)
throws CEASTparseException {
  final NodeList nl = e.getChildNodes();
  IAST2expression<CEASTparseException> condition =
    (IAST2expression<CEASTparseException>)
      parser.findThenParseChildAsUnique(
                nl, "condition");
  IAST2instruction<CEASTparseException>
    consequence = (IAST2instruction<CEASTparseExcept
      parser.findThenParseChildAsSequence(
                nl, "consequence");
  ...
  return parser.getFactory().newAlternative(
           condition, consequence, alternant);
}
```

Analyseur des blocs n-aires avec XPath

```
<blocLocal>
  <liaisons>
    <liaison>
      <variable nom='x'/>
      <initialisation>   <!-- conteneur -->
         ... expression ...
      </initialisation>
    </liaison>
    ...
  </liaisons>
  <corps>
    ... instruction ...
  </corps>
</blocLocal>
```

Récupération des variables avec `liaisons/liaison/variable`


```
private final IAST2variable[] variable;
private final IAST2expression<CEASTparseException>[] initializati
private final IAST2instruction<CEASTparseException> body;

private static final XPath xPath =
  XPathFactory.newInstance().newXPath();

public static IAST2localBlock<CEASTparseException> parse (
  Element e, IParser<CEASTparseException> parser)
 throws CEASTparseException {
  IAST2variable[] variables = new IAST2variable[0];
  IAST2expression<CEASTparseException>[] initializations;
  try {
    XPathExpression bindingVarsPath =
      xPath.compile("./liaisons/liaison/variable");
    NodeList nlVars = (NodeList)
      bindingVarsPath.evaluate(e, XPathConstants.NODESET);
    List<IAST2variable> vars = new Vector<IAST2variable>();
    for ( int i=0 ; i<nlVars.getLength() ; i++ ) {
      Element varNode = (Element) nlVars.item(i);
      IAST2variable var =
        parser.getFactory().newVariable(varNode.getAttribute("nom
    ...
```


Sémantique discursive

Boucle comme en C (sans sortie prématurée)
Affectation comme en C (expression) sauf que (comme en JavaScript) l'affectation sur une variable non locale crée la variable globale correspondante

```
let n = 1 in
  while n < 100 do
    f = 2 * n
  done;
print f
```


Fonctions globales en récursion mutuelle (comme en JavaScript, pas comme en C ou Pascal)

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

Bloc n-aire comme en Scheme et pas comme en C

```
(let ((x 1))
  (let ((x (* 2 x))
        (y (* 2 x)) )
    (= x y) ) )          ; est vrai
```


Boucle : définition

```
public class CEASTwhile extends CEASTinstruction
  implements IAST2while<CEASTparseException> {

  public CEASTwhile(IAST2expression condition, //interfac
                    IAST2instruction body)     //interfac
  {
    this.condition = condition;
    this.body = body;
  }
  private final IAST2expression condition;     //interfac
  private final IAST2instruction body;         //interfac

  public IAST2expression getCondition () {     //interfac
      return condition;
  }
  public IAST2instruction getBody () {         //interfac
      return body;
  }
```


Boucle : interprétation

Usage systématique des interfaces IAST2*

```
public Object eval (ILexicalEnvironment lexenv,
                    ICommon common)
  throws EvaluationException {
  while ( true ) {
    Object bool = getCondition().eval(lexenv, comm
    if ( Boolean.FALSE == bool ) {
      break;
    }
    getBody().eval(lexenv, common);
  }
  return Boolean.FALSE;
}
```


Boucle : compilation

Il y a un équivalent en C que l'on emploie !
boucle = (condition, corps)

```
                  →d
              boucle


while ( ILP_isEquivalentToTrue( →condition ) ) {
  →(void)
    corps   ;
}
    →d
nImporteQuoi  ;
```

Usage systématique des interfaces IAST2*

```
public void compileInstruction (
     StringBuffer buffer,
     ICgenLexicalEnvironment lexenv,
     ICgenEnvironment common,
     IDestination destination)
throws CgenerationException {
  buffer.append(" while ( ILP_isEquivalentToTrue( ");
  getCondition().compileExpression(
       buffer, lexenv, common);
  buffer.append(") ) { ");
  getBody().compileInstruction(
       buffer, lexenv, common,
       VoidDestination.create() );
  buffer.append("\n}\n");
  CEASTInstruction.voidInstruction()
    .compileInstruction(buffer, lexenv, common,
                        destination);
}
```


Boucle : exemple

```
;;; $Id: u52-2.scm 405 2006-09-13 17:21:53Z queinnec $
(comment "boucle tant-que")
(let ((x 50))
  (while (< x 52)
    (set! x (+ x 1)) )
  x )

;;; end of u52-2.scm
```



```
{ /* (let ((x 50)) */
  ILP_Object TMP133 = ILP_Integer2ILP (50);
  ILP_Object x = TMP133;
  { /* while (< x 52) */
    while (ILP_isEquivalentToTrue (
             ILP_LessThan (x, ILP_Integer2ILP (52)))
      { /* (set! x (+ x 1)) */
        {
          (void) (x =
             ILP_Plus (x, ILP_Integer2ILP (1)));
        }
      }
    (void) ILP_FALSE;
    /* x */
    return x;
  }
}
```


Affectation

Les variables sont maintenant modifiables. Les interfaces des environnements d'interprétation doivent donc procurer cette nouvelle fonctionnalité.

```
public interface ILexicalEnvironment extends
fr.upmc.ilp.ilp1.runtime.ILexicalEnvironment {

  void update (IAST2variable variable,
               Object value)
    throws EvaluationException;
}
```

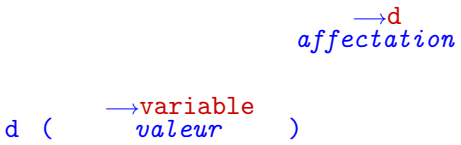
```
public interface ICommon
extends fr.upmc.ilp.ilp1.runtime.ICommon {

  Object primitiveLookup (
        IAST2predefinedVariable variable)
    throws EvaluationException;
  void bindPrimitive (String primitiveName,
                      Object value)
    throws EvaluationException;

  Object globalLookup (IAST2variable variable)
    throws EvaluationException;
  void updateGlobal (String variable, Object value)
    throws EvaluationException;

  boolean isPresent (IAST2variable variable);
}
```


Affectation : interprétation

```
public Object eval (ILexicalEnvironment lexenv,
                    ICommon common)
  throws EvaluationException {
  Object newValue = getValue().eval(lexenv, common);
  try {
    lexenv.update(getVariable(), newValue);
  } catch (EvaluationException e) {
    common.updateGlobal(getVariable().getName(),
                        newValue);
  }
  return newValue;
}
```


Affectation : compilation

Là encore, on utilise les ressources de C.
affectation = (variable, valeur)



Affectation : génération de code

```
public void compileExpression (
      StringBuffer buffer,
      ICgenLexicalEnvironment lexenv,
      ICgenEnvironment common,
      IDestination destination)
throws CgenerationException {
    destination.compile(buffer, lexenv, common);
    buffer.append(" (");
    getValue().compileExpression(buffer, lexenv,
            common,
            new AssignDestination(getVariable()) );
    buffer.append(") ");
}
```

Variables globales

L'affectation sur une variable non locale réclame, en C, que l'on ait déclaré au préalable cette variable globale.

1. il faut collecter les variables globales
2. pour chacune d'entre elles, il faut l'allouer et l'initialiser.

Une méthode `computeGlobalVariables` est définie sur `CEASTprogram` et une méthode `findGlobalVariables` tous les nœuds de l'AST pour collecter ces variables.
Première analyse statique : collecte des variables globales.
Réalisation : par arpentage de l'AST (un visiteur).


```
;;; $Id: u59-2.scm 405 2006-09-13 17:21:53Z queinnec $
(comment "variable globale non fonctionnelle")
(let ((x 1))
  (set! g 59)
  g )

;;; end of u59-2.scm
```



```
/* Variables globales: */
static ILP_Object g = NULL;
/* Prototypes: */
/* Fonctions globales: */
/* Code hors fonction: */

ILP_Object
program ()
{
  {
    ILP_Object TMP137 = ILP_Integer2ILP (1);
    ILP_Object x = TMP137;
    {
      (void) (g = ILP_Integer2ILP (59));
      return g;
    }
  }
}
```


Collecte des variables globales

Toute variable non locale est globale.
Parcours récursif de l'AST (comme pour `eval` ou `compile`) : une méthode par classe.

```
// CEASTwhile
@Override
public void findGlobalVariables (
      Set<IAST2variable> globalvars,
      ICgenLexicalEnvironment lexenv ) {
    getCondition().findGlobalVariables(
       globalvars, lexenv);
    getBody().findGlobalVariables(
       globalvars, lexenv);
}
```

Collecte des variables globales

GV(sequence(i1, i2, ...) = GV(i1) ∪ GV(i2) ∪ ...
GV(alternative(c,it,if)) = GV(c) ∪ GV(it) ∪ GV(if)
GV(boucle(c,s)) = GV(c) ∪ GV(s)
GV(affectation(n,v)) = { n } ∪ GV(v)
GV(constante) = ∅
GV(variable) = { variable }
GV(definitionFonction(n,(v1,v2,...),c) = GV(c) - { v1, v2, ... }
GV(blocUnaire(v,e,c)) = GV(e) ∪ (GV(c) - { v })

Collecte variables globales (suite)

```
// CEASTlocalBlock
public void findGlobalVariables (
      Set<IAST2variable> globalvars, // Resultat
      ICgenLexicalEnvironment lexenv,
      ICgenEnvironment common ) {
  ICgenLexicalEnvironment bodylexenv = lexenv;
  for ( IAST2variable var : getVariables() ) {
     bodylexenv = bodylexenv.extend(var);
  }
  for ( IAST2expression expr : getInitializations() )
     expr.findGlobalVariables(
      globalvars, lexenv, common);
  }
  getBody().findGlobalVariables(
     globalvars, bodylexenv, common);
}
```

GV(let v = e in i+) = GV(e) ∪ (GV(i+) - {v})

Collecte variables globales (suite)

```
// CEASTreference
public void findGlobalVariables (
          Set<IAST2variable> globalvars,
          ICgenLexicalEnvironment lexenv ) {
  if ( ! lexenv.isPresent(getVariable()) ) {
    if ( ! (getVariable() instanceof
              CEASTpredefinedVariable ) ) {
      globalvars.add(getVariable());
    }
  }
}
```


Fonctions : interprétation

```
// Class CEASTfunctionDefinition
public Object eval (ILexicalEnvironment lexenv,
                    ICommon common)
  throws EvaluationException {
  final Object function =
      new UserGlobalFunction(
              getFunctionName(),
              getVariables(),
              getBody());
  common.updateGlobal(getFunctionName(), function);
  return function;
}
```

repose sur un nouvel objet de la bibliothèque d'exécution.

```
public class UserGlobalFunction
  implements IUserFunction {
  public Object invoke (final Object[] arguments,
                        final ICommon common)
    throws EvaluationException {
      IAST2variable[] variables = getVariables();
      if ( variables.length != arguments.length ) {
          final String msg =
            "Wrong arity for function:" + name;
          throw new EvaluationException(msg);
      };
      ILexicalEnvironment lexenv = getEnvironment();
      for ( int i = 0 ; i<variables.length ; i++ ) {
          lexenv = lexenv.extend(variables[i],
                                 arguments[i]);
      }
      return getBody().eval(lexenv, common);
  }
}
```


Fonctions : compilation

fonctionGlobale = (nom, variables..., corps)

$\longrightarrow$ *fonctionGlobale*

```
// Declaration
static ILP_Object nom (
       ILP_Object variable, ... );
...
// Definition
ILP_Object nom (
       ILP_Object variable,
       ...
) {
   →return
    corps
}
```


Programme

Un programme, `CEASTprogram`, contient

- syntaxiquement :
  - une liste de fonctions
  - un corps
- et (après calcul) une liste de variables globales.

Transformation de programme

Pour simplifier l'appel depuis C, on effectue la transformation

```
fonction1(...) { ...}
fonction2(...) { ...}
instruction1
instruction2
instruction3
```

→

```
fonction1(...) { ...}
fonction2(...) { ...}
program() {
  instruction1
  instruction2
  instruction3
}
program()
```

Mise en œuvre

```
////////////////////// CEASTprogram.java
public void compileInstruction (
       StringBuffer buffer,
       ICgenLexicalEnvironment lexenv,
       ICgenEnvironment common,
       String destination)
throws CgenerationException {
  final IAST2functionDefinition[] definitions =
       getFunctionDefinitions();
  // Declarer les variables globales:
  buffer.append("/* Variables globales: */\n");
  findGlobalVariables(lexenv, common);
  for (IAST2variable var : getGlobalVariables()) {
     buffer.append("static ILP_Object ");
     var.compileExpression(buffer, lexenv, common);
     buffer.append(" = NULL;\n");
  }
```



```
  // emettre le code des fonctions:
  buffer.append("/* Prototypes: */\n");
  for ( IAST2functionDefinition fun : definitions )
     fun.compileHeader(buffer, lexenv, common);
  }
  buffer.append("/* Fonctions globales: */\n");
  for ( IAST2functionDefinition fun : definitions )
     fun.compile(buffer, lexenv, common);
  }
  // emettre les instructions regroupees en fonction
  buffer.append("/* Code hors fonction: */\n");
  IAST2functionDefinition bodyAsFunction =
      new CEASTfunctionDefinition(
              "program",
              CEASTvariable.EMPTY_VARIABLE_ARRAY,
              getBody() );
  bodyAsFunction.compile(buffer, lexenv, common);
}
```


Patron C

Le script `compileThenRun.sh` reçoit des arguments car le patron a changé.

```
#include <stdio.h>
#include <stdlib.h>
#include "ilp.h"

/* Ici l'on inclut le code C produit: */

int main (int argc, char *argv[])
{
     ILP_print(program());
     ILP_newline();
     return EXIT_SUCCESS;
}
```


Quelques nouveautés

- Tous les champs des classes `CEAST*` sont typés avec des interfaces `IAST2*` étendant les interfaces `IAST*`
- Hors constructeurs, tous les accès aux champs passent par les méthodes `get*` ainsi qu'indiquées dans les `IAST2*`
- analyseur syntaxique `parse()` par fonctions statiques
- introduction de `CEASTreference`
- première analyse statique `findGlobalVariables`
- introduction des `IDestination`
- Utilisation d'XPath (cf. `CEASTlocalBlock`)
- `BasicEnvironment` et `BasicEmptyEnvironment` génériques

80 classes ou interfaces, 4700 lignes de Java.

Revision: 1.16

Master d'informatique 2014-2015
Spécialité STL
« Implantation de langages »
ILP – MI016
épisode ILP3

C.Queinnec

Buts

- Portée
- Durée de vie
- Échappement
- Technologie C

Plan du cours 6

ILP3 = ILP2 + exceptions

- Syntaxe
- Évaluation
- Génération de C
- Bibliothèque d'exécution

Pourquoi des exceptions ?

Trop de programmeurs ne testent pas les codes de retour !
Les exceptions rompent la structure normale du programme et ne peuvent donc pas être ignorées (surtout quand associées au typage).

Caractéristiques

Un mécanisme d'exception permet de

- signaler des exceptions : `throw`
- rattraper des exceptions : `try`/`catch`

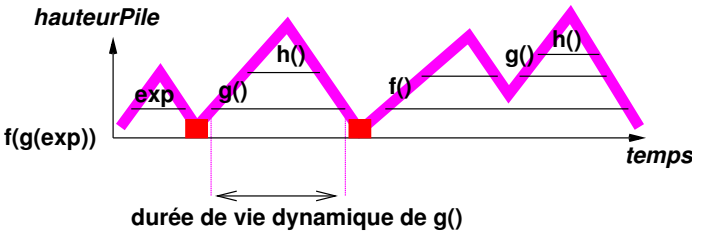
On y ajoute souvent la possibilité de détecter la terminaison d'un calcul : `try`/`finally`
L'évaluateur, les bibliothèques prédéfinies doivent signaler des exceptions !
Quelle sera la taxonomie des exceptions prédéfinies ?

Durée de vie dynamique

Une expression peut s'achever en

- retournant une valeur ou,
- signalant une exception.

Les calculs forment une structure bien emboîtée qui n'a rien à voir avec la structure lexicale du programme. On parle de **durée de vie dynamique** qui correspond très précisément à la pile d'évaluation.



Durée de vie dynamique en C

```
// portee globale + duree de vie totale:
extern int g;
// portee fichier + duree de vie totale:
static int gfile = 1;
void f () {
  // portee lexicale + duree de vie dynamique:
  int lf = 2;
  // portee lexicale + duree de vie totale:
  static int sg = lf;
  g(&lf);
}
void g (int *pi) {
  // lf invisible ici
  // portee lexicale + duree de vie indefinie:
  int *mg = malloc(sizeof(int));
  *pi = mg;       // N'importe quoi!!!
  free(mg);
  // Pourquoi ne pas avoir alloue mg en pile ?
}
```

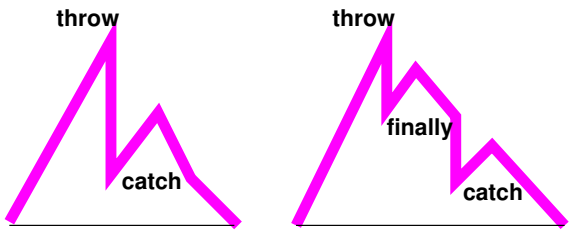

Exemples

Rattrapage d'exception, suspension temporaire d'exception :

```
try {
  throw 1;
  print 2;
} catch (e) {
  print e;
} finally {
  print 3;
}  // imprime 13
```

```
try {
  try {
    throw 1;
    print 2;
  } finally {
    print 3;
  }
  print 4;
} catch (e) {
  print e;
}  // imprime 31
```

Exemples



Attention ! Que signifient les signalisations d'exceptions depuis les clauses `catch` et `finally` ?

```
try {
  try {
    throw 1;
    print 2;
  } catch (e) {
    throw (10*e);
    print 3;
  }
  print 4;
} catch (e) {
  print e;
}  // imprime 10
```

```
try {
  try {
    throw 1;
    print 2;
  } catch (e) {
    throw (10*e);
    print 3;
  } finally {
    throw 111;
  }
  print 4;
} catch (e) {
  print e;
} // imprime 111
```


Souhaits

- Le traitement des exceptions `catch` coûte cher : le traitement doit donc être exceptionnel.
- Le confinement de calcul `try` doit être le moins coûteux possible : idéalement 0 instruction !
- **Ne doivent payer pour une caractéristique que ceux qui s'en servent !**

Syntaxe abstraite

```
# grammar3.rnc
include "grammar2.rnc"
start |= programme3
instruction |= try

programme3 = element programme3 {
   definitionFonction *,
   instructions
}
try = element try {
   element corps { instructions },
   (  catch
    | finally
    | ( catch, finally )
   )
}
```

```
catch = element catch {
   attribute exception { xsd:Name - ( xsd:Name { patte
   instructions
}
finally = element finally {
   instructions
}
```

et une fonction de plus prédéfinie : `throw` !

Organisation

- Un seul paquetage `fr.upmc.ilp.ilp3`
- seulement 12 classes utiles
- 700 loc

Évaluation

On se repose sur le `try`/`catch`/`finally` de Java.
N'importe quelle valeur d'ILP3 peut être signalée comme une exception.
Deux sortes d'exceptions :

- celles signalées par l'utilisateur par `throw`
- celles signalées par la machine sous-jacente (`java.lang.RuntimeException`).


```
public class ThrownException
extends EvaluationException {
  public ThrownException (final Object value) {
    super("Thrown value");
    this.value = value;
  }
  private final Object value;
  public Object getThrownValue () {
    return value;
  }
  public String toString () {
    return "Thrown value: " + value;
  }
}
```

```
public class ThrowPrimitive
extends AbstractInvokable {
  ...
  public Object invoke (final Object exception)
    throws EvaluationException {
    if (exception instanceof EvaluationException)
      EvaluationException exc =
        (EvaluationException) exception;
      throw exc;
    } else
    if (exception instanceof RuntimeException) {
      RuntimeException exc =
        (RuntimeException) exception;
      throw exc;
    } else {
      throw new ThrownException(exception);
    }
  }
```



```
// CEASTtry
  public Object eval (ILexicalEnvironment lexenv,
                      ICommon common)
    throws EvaluationException {
    Object result = Boolean.FALSE;
    try {
      result = getBody().eval(lexenv, common);
    } catch (ThrownException e) {
      final ILexicalEnvironment catcherLexenv =
        lexenv.extend(getCaughtException(),
                      e.getThrownValue());
      getCatcher().eval(catcherLexenv, common);
    } catch (Exception e) {
      final ILexicalEnvironment catcherLexenv =
        lexenv.extend(getCaughtException(), e);
      getCatcher().eval(catcherLexenv, common);
    } finally {
      getFinallyer().eval(lexenv, common);
    }
    return result;
  }
```


Compilation

Usage de `setjmp` et `longjmp`

```
#include <setjmp.h>
int setjmp(jmp_buf env);
void longjmp(jmp_buf env, int val);

if ( 0 == setjmp(jmp_buf) ) {
    // essai
    ... longjmp(jmp_buf, 1) ...
} else {
    // traitement d'anomalie (du longjmp)
}
```


Problèmes

- la valeur passée est un `int`, pas une valeur d'ILP
- comment transmettre la connaissance du `jmp_buf` entre `setjmp` et `longjmp` ?
- les instructions des clauses `catch` et `finally` doivent être sous le contrôle du `try` englobant.

Une solution :

- variable globale `ILP_current_exception` pour passer l'exception (une valeur ILP)
- liste chaînée de rattrapeurs référencée par une variable globale `ILP_current_catcher`

Interface

```
// depuis ilpException.h
struct ILP_catcher {
     struct ILP_catcher *previous;
     jmp_buf _jmp_buf;
};

extern struct ILP_catcher *ILP_current_catcher;
extern ILP_Object ILP_current_exception;
extern ILP_Object ILP_throw(ILP_Object exception);
extern void ILP_establish_catcher(
    struct ILP_catcher *new_catcher );
extern void ILP_reset_catcher(
    struct ILP_catcher *catcher );
```



```
// depuis ilpException.c
static struct ILP_catcher
  ILP_the_original_catcher = {
     NULL
};
struct ILP_catcher *ILP_current_catcher =
   &ILP_the_original_catcher;

ILP_Object ILP_current_exception = NULL;
```



```
ILP_Object
ILP_throw (ILP_Object exception)
{
     ILP_current_exception = exception;
     if ( ILP_current_catcher ==
              &ILP_the_original_catcher ) {
          ILP_die("No current catcher!");
     };
     longjmp(ILP_current_catcher->_jmp_buf, 1);
     /** UNREACHABLE */
     return NULL;
}
```



```
void
ILP_establish_catcher(
    struct ILP_catcher *new_catcher)
{
     new_catcher->previous = ILP_current_catcher;
     ILP_current_catcher = new_catcher;
}

void
ILP_reset_catcher (struct ILP_catcher *catcher)
{
     ILP_current_catcher = catcher;
}
```

Compilation de try

```
{ struct ILP_catcher *current_catcher =
    ILP_current_catcher;
  struct ILP_catcher new_catcher;
  if ( 0 == setjmp(new_catcher._jmp_buf) ) {
      ILP_establish_catcher(&new_catcher);
      corps
      /* NULL n'est pas une valeur ILP */
      ILP_current_exception = NULL;
  };
  /* ici, soit ILP_current_exception est NULL et
     c'est un retour normal sinon c'est un
     echappement qu'on doit rattraper. */
```

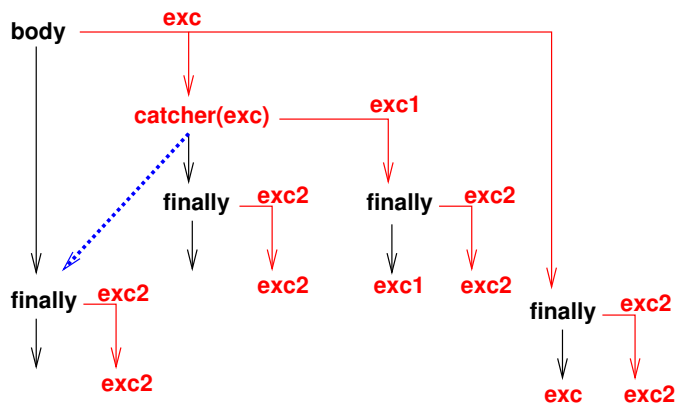


```
/* Ces instructions ne sont presentes que s'il
   y a un rattrapeur. Dans ce cas, il faut
   confiner le rattrapeur au cas ou il
   chercherait lui aussi a s'echapper car il
   y a encore le finaliseur a tourner. Attention
   ce code n'est present que si un rattrapeur
   est mentionne. */
ILP_reset_catcher(current_catcher);
if ( NULL != ILP_current_exception ) {
    if ( 0 == setjmp(new_catcher._jmp_buf) ) {
        ILP_establish_catcher(&new_catcher);
        { ILP_Object exception =
            ILP_current_exception;
          ILP_current_exception = NULL;
          catcher
        }
    };
};
```



```
  /* Ici il faut tourner le finaliseur. Attention,
     ce code n'est present que si un rattrapeur
     est mentionne. */
  ILP_reset_catcher(current_catcher);
  finally
  /* (re)prendre l'echappement si suspendu ou
     demande par finally */
  if ( NULL != ILP_current_exception ) {
      ILP_throw(ILP_current_exception);
  };
}
```


Flots de contrôle des exceptions



Patron C

et un nouveau patron tel que (choix personnel) :

$P \equiv$

```
try P catch (e) { return e; }
```

```
#include <stdio.h>
#include <stdlib.h>
#include "ilp.h"
#include "ilpException.h"

/* Ici l'on inclut le code C produit: */
```



```
static ILP_Object
ilp_caught_program ()
{
  struct ILP_catcher* current_catcher =
      ILP_current_catcher;
  struct ILP_catcher new_catcher;

  if ( 0 == setjmp(new_catcher._jmp_buf) ) {
    ILP_establish_catcher(&new_catcher);
    return program();
  };
  /* Une exception est survenue. */
  return ILP_current_exception;
}
```



```
int
main (int argc, char *argv[])
{
     ILP_print(ilp_caught_program());
     ILP_newline();
     return EXIT_SUCCESS;
}
```


Remarques

- permet d'écrire des tests qui doivent échouer.
- Ne traite pas les erreurs de la machine C :
  - une division par zéro n'est pas transformée en une exception rattrapable.
- Utilise des variables globales (ne permet pas le multi-tâches)
- Définition du rattrapeur par défaut.

Conclusions

- Modèle d'exception standard (Ada, Java, Javascript, ILP) :
  - descendre en pile
  - jusqu'à trouver un rattrapeur
  - et le tourner là.
- Pas d'exception continuable
- Coûteux en C :
  - à l'établissement
  - à l'usage

Récapitulation

- `setjmp`/`longjmp`
- Ne change que la bibliothèque d'exécution
- Permet d'écrire des tests dont on peut vérifier qu'ils échouent
- 12 classes, 700 lignes de Java et C.