Revision: 1.12

Master d'informatique 2007-2008
Spécialité STL
« Implantation de langages »
ILP – MI016
Cours 5

C.Queinnec[a]

[a] http://www-spi.lip6.fr/~queinnec/

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` !

`fr.upmc.ilp.ilp2.interfaces`	interfaces diverses
`.ast`	AST (et analyse syntaxique)
`.runtime`	bibliothèque d'interprétation
`.cgen`	Compilation vers C

Grammaire *Grammars/grammar2.rnc*

Nouveau patron *C/templateTest2.c*

Programmes ILP2 additionnels *Grammars/Samples/*-2.xml*

ORGANISATION JAVA

```
ilp2.interfaces.IAST2
                    IAST2program
                    IAST2functionDefinition
                    IAST2instruction
                      IAST2alternative
                      IAST2expression
                        IAST2assignment
                        IAST2invocation
                        ...
                      ...
                  IDestination  // Robustesse
```

IAST2 = IAST + eval() + findFreeVariables()

IAST2instruction = IAST2 + compileInstruction()

IAST2expression = IAST2instruction + compileExpression()

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

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

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

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 *,
   instruction +
}
definitionFonction = element definitionFonction {
   attribute nom      { xsd:Name - ( xsd:Name { pattern = "(ilp|ILP)" } ) },
   element variables { variable * },
   element corps      { instruction + }
}
```

```
blocLocal = element blocLocal {
   element liaisons {
      element liaison { variable, expression } *
   },
   element corps { instruction + }
}
boucle = element boucle {
   element condition { expression },
   element corps     { instruction + }
}
affectation = element affectation {
   attribute nom  { xsd:Name - ( xsd:Name { pattern = "(ilp|ILP)" } ) },
   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

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

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();

      if ( "alternative".equals(name) ) {
        return CEASTalternative(e, this);
      } else if ( "sequence".equals(name) ) {
        return CEASTsequence(e, this);
      ...
```

ANALYSEUR DES ALTERNATIVES

```
public static CEASTalternative<CEASTparseException> parse (
            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<CEASTparseException>)
      parser.findThenParseChildAsSequence(nl, "consequence");
  try {
    IAST2instruction<CEASTparseException> alternant =
        (IAST2instruction<CEASTparseException>)
        parser.findThenParseChildAsSequence(nl, "alternant");
    return parser.getFactory().newAlternative(
              condition, consequence, alternant);
  } catch (CEASTparseException exc) {
    return parser.getFactory().newAlternative(condition, consequence);
  }
}
```

ANALYSEUR DES BLOCS N-AIRES AVEC XPATH

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

```
private final IAST2variable[] variable;
private final IAST2expression[] initialization;
private final IAST2instruction body;

private static final XPath xPath =
    XPathFactory.newInstance().newXPath();
public static CEASTlocalBlock parse (
    Element e, IParserCEASTparseException> parser)
throws CEASTparseException {
    IAST2variable[] variables = new IAST2variable[0];
    IAST2expression<CEASTparseException>[] initializations;
    try {
      final XPathExpression bindingVarsPath =
          xPath.compile("./liaisons/liaison/*[position()=1]");
      final NodeList nlVars = (NodeList)
          bindingVarsPath.evaluate(e, XPathConstants.NODESET);
      final List<IAST2variable> vars = new Vector<IAST2variable>();
      for ( int i=0 ; i<nlVars.getLength() ; i++ ) {
        final Element varNode = (Element) nlVars.item(i);
        final IAST2variable var =
             parser.getFactory().newVariable(varNode.getAttribute("nom"));
        vars.add(var);
      }
```

```
 this.variable = vars.toArray(CEASTvariable.EMPTY_VARIABLE_ARRAY);
...
```

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.

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

BOUCLE : DÉFINITION

```
public class CEASTwhile
  extends CEASTinstruction
  implements IAST2while {

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

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

BOUCLE : INTERPRÉTATION

Usage systématique des interfaces IAST2*

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

BOUCLE : COMPILATION

Il y a un équivalent en C que l'on emploie !

$\longrightarrow$***d***
boucle

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

Usage systématique des interfaces IAST2*

```
public void compileInstruction (final StringBuffer buffer,
                                final ICgenLexicalEnvironment lexenv,
                                final ICgenEnvironment common,
                                final 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 4052006-09-1317:21:53Z queinnec
(comment "boucle tant-que")
(let ((x 50))
  (while (< x 52)
    (set! x (+ x 1)) )
  x )

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

```
{
  ILP_Object TMP133 = ILP_Integer2ILP (50);
  ILP_Object x = TMP133;
  {
    while (ILP_isEquivalentToTrue (ILP_LessThan (x, ILP_Integer2ILP (52))))
      {
        {
          (void) (x = ILP_Plus (x, ILP_Integer2ILP (1)));
        }
      }
    (void) ILP_FALSE;
    return x;
  }
}
```

AFFECTATION

Les variables sont maintenant modifiables. Les interfaces des environnements 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 (final ILexicalEnvironment lexenv,
                    final 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 ⟶ **d**

d (*valeur* ⟶ **variable**)

```
public void compileExpression (final StringBuffer buffer,
                               final ICgenLexicalEnvironment lexenv,
                               final ICgenEnvironment common,
                               final 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 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 `findGlobalVariables` est définie sur `CEASTprogram` et une méthode `findFreeVariables` tous les nœuds de l'AST pour collecter ces variables.

```
;;; Id : u59 – 2.scm 4052006 – 09 – 1317 : 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.

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

FONCTIONS : INTERPRÉTATION

```
// Class CEASTfunctionDefinition
  public Object eval (final ILexicalEnvironment lexenv,
                      final 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

$\longrightarrow$ *fonctionGlobale*

```
static ILP_Object nom (
       ILP_Object variable, ... );

...

ILP_Object nom (
       ILP_Object variable,
       ...
) {
   ⟶return
     corps
}
```

PROGRAMME

Un programme, `CEASTprogram`, contient

- syntaxiquement :
  - une liste de fonctions
  - un corps
- et 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 (final StringBuffer buffer,
                                final ICgenLexicalEnvironment lexenv,
                                final ICgenEnvironment common,
                                final String destination)
throws CgenerationException {
    final IAST2functionDefinition[] definitions = getFunctionDefinitions();
    // Déclarer 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");
    }
    // Émettre 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);
    }
    // Émettre les instructions regroupées dans une 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: */
#include FICHIER_C

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 `CEASTprogram`
- première analyse statique `findGlobalVariables`
- introduction des `IDestination`
- Utilisation d'XPath (cf. `CEASTlocalBlock`)
- `BasicEnvironment` et `BasicEmptyEnvironment` génériques

Revision: 1.8

Master d'informatique 2007-2008
Spécialité STL
« Implantation de langages »
ILP – MI016
Cours 6

C.Queinnec[a]

[a] http://www-spi.lip6.fr/~queinnec/

PLAN DU COURS 6

Exceptions en ILP3 :

- 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 !

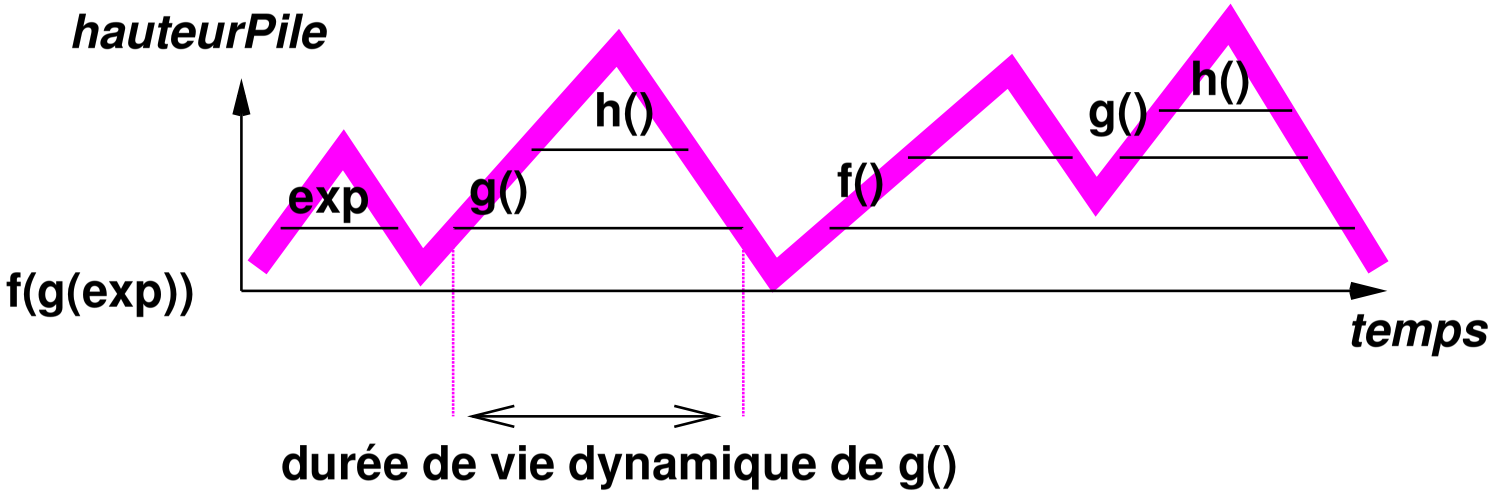
Quelle sera la taxonomie des exceptions prédéfinies ?

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

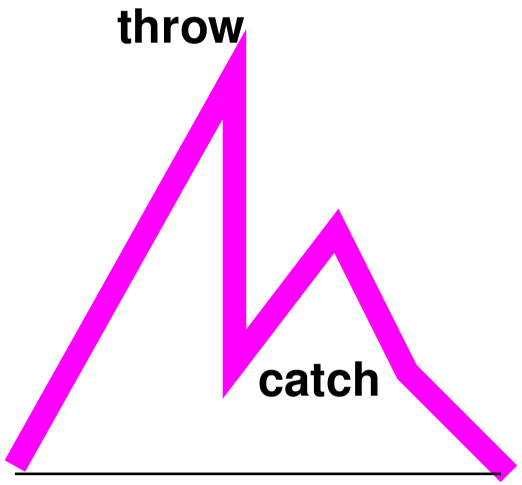
```
extern int g;              // portée globale + durée de vie totale
static int gfile = 1;      // portée fichier + durée de vie totale
void f () {
  int lf = 2;              // portée lexicale + durée de vie dynamique
  g(&lf);
}
void g (int *pi) {         // lf invisible
  int *mg = malloc(sizeof(int));   // durée de vie indéfinie
  *pi = mg;                // N'importe quoi!!!
  free(mg);                // Pourquoi ne pas allouer 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
```



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

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

programme3 = element programme3 {
   definitionFonction *,
   instruction +
}
try = element try {
   element corps { instruction + },
   (  catch
    | finally
    | ( catch, finally )
   )
}
catch = element catch {
   attribute exception { xsd:Name - ( xsd:Name { pattern = "(ilp|ILP)" } ) },
   instruction +
```

```
}
finally = element finally {
   instruction +
}
```

et une fonction de plus prédéfinie : `throw` !

ORGANISATION

Un seul paquetage `fr.upmc.ilp.ilp3` et seulement 7 classes utiles.

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);
    }
  }
}
```

```
public Object eval (final ILexicalEnvironment lexenv,
                    final 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 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` sont 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;
}
```

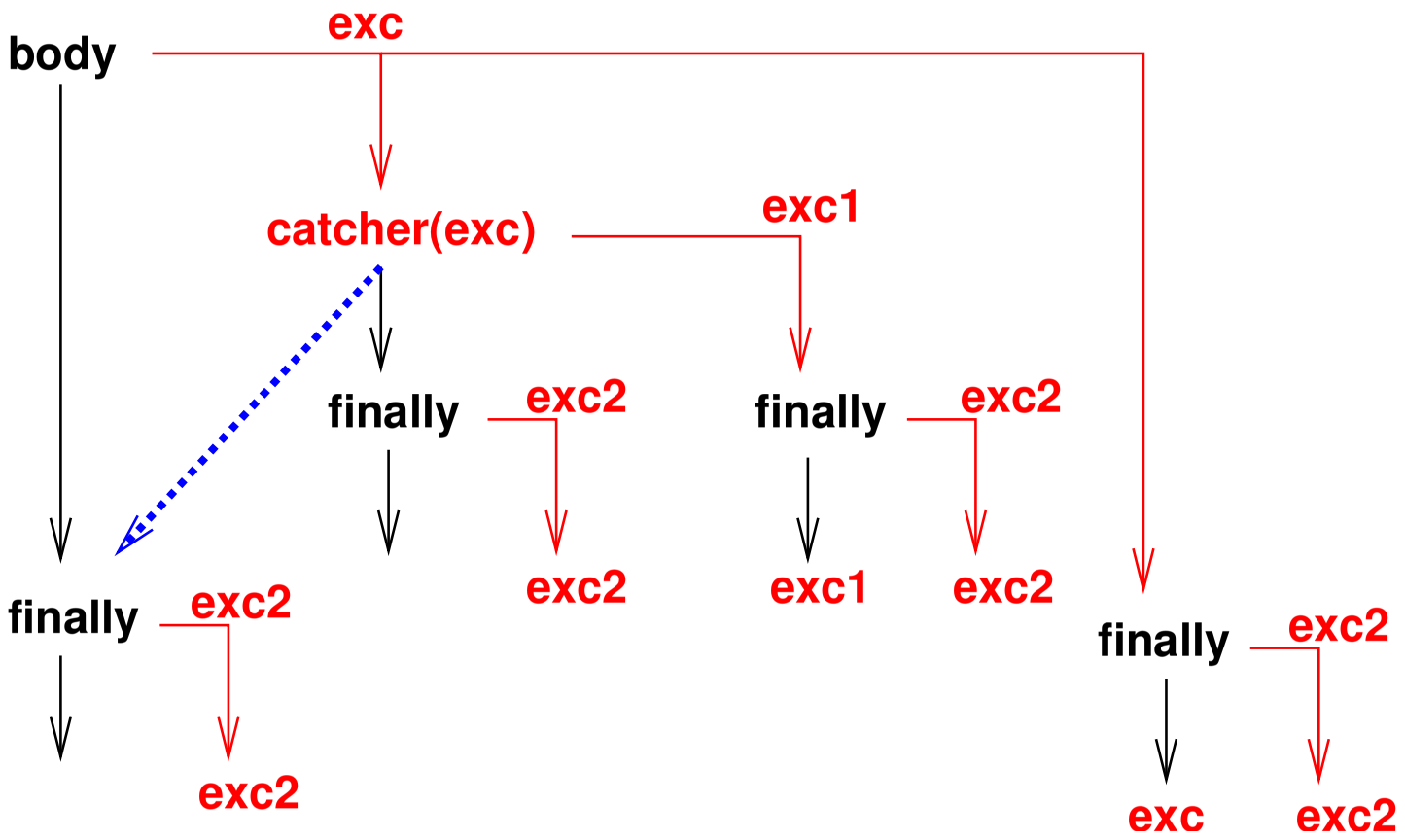
COMPILATION DE `try`

```
{ 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
      ILP_current_exception = NULL; /* pas une valeur ILP */
  };
  /* ici, soit ILP_current_exception est NULL et c'est un retour
     normal sinon c'est un échappement qu'on doit rattraper. */
```

```
/* Ces instructions ne sont présentes que s'il y a un rattrapeur.
   Dans ce cas, il faut confiner le rattrapeur au cas où il
   chercherait lui aussi à s'échapper car il y a encore
   le finaliseur à tourner! Attention, ce code n'est présent que si
   un rattrapeur est mentionné. */
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 */
  ILP_reset_catcher(current_catcher);
  finally
  /* (re)prendre l'échappement si suspendu ou demandé 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: */
#include FICHIER_C
```

```
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 :
  - 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

TECHNIQUES JAVA

Plein de nouveautés en ILP4 !

POUR LA PROCHAINE FOIS

- Lire le code d'ILP3 (7 classes seulement)
- lire les 2 fichiers C additionnels.
- Que fait le (petit) programme `throw 11;` ?