

# Les exceptions

DLP : Développement d'un langage de programmation  
Master STL, UPMC

Antoine Miné

Année 2015–2016

Cours 6  
20 octobre 2015

# Plan général

Développement d'un **langage** de programmation : **interprète** et **compilateur**.

Par étapes, avec ajout progressif de fonctionnalités.

- **ILP1** : langage de base
  - constantes (entiers, flottants, booléens, chaînes),
  - opérateurs (+, -, ×, ...),
  - appels de primitives (print, ...),
  - blocs locaux (let x = ... in ...),
  - alternatives (if ... then ... else)
- **ILP2** : ajout des boucles, affectations, fonctions globales
- **ILP3** : ajout des **exceptions** et fonctions de première classe
- **ILP4** : ajout des classes et des objets

Ce cours : **les exceptions dans ILP3**

- Les exceptions : rappels et généralités
- Rappels : comment étendre le langage
- Extension de la syntaxe
- Extension de l'interprète (évaluation directe en Java)
- Extension du compilateur (génération du C)
- Extension de la bibliothèque d'exécution (exécution du C)

# Objectifs du cours

- revoir les étapes pour étendre ILP (utile en TME !)
- constructions de contrôle non-locales (au delà du `return`)
- notion de pile d'appel
- ajouter des exceptions à un langage qui ne le supporte pas (C)
- maîtriser `setjmp` et `longjmp` en C

# Les exceptions

---

# Un langage sans exception : le C

Le C n'a pas d'exceptions.

Comment gérer les erreurs ?

Pas de règle dans le langage, mais des conventions de bibliothèques :

- **Valeur de retour spéciale** : `-1` (e.g, `open`), `NULL` (e.g., `malloc`).

Peu pratique quand toutes les valeurs du type de retour sont déjà utilisées.

- **Variable** contenant un code d'erreur : `errno`.

Utilisé en combinaison avec une valeur de retour spéciale, pour préciser l'erreur.

- **Fonction** renvoyant un code d'erreur : `glGetError` (OpenGL).

En théorie, permet une gestion asynchrone des erreurs.

Dans tous les cas : il faut écrire du code pour **vérifier la présence d'erreur** !

⇒ coûteux en temps de programmation, et il est facile de se tromper.

Attention à ne pas confondre une valeur "normale" et une valeur d'erreur.

gare aux accès à un pointeur `NULL` après un `malloc`

Cas particulier : les flottants IEEE, avec propagation des *Not a Number*.

# Exemples de code C avec gestion des erreurs

## lire deux entiers dans un fichier

```
int f = open(name,O_RDONLY);  
if (f < 0) /* gestion d'erreur */  
if (read(f,&x,4) != 4) /* gestion d'erreur */  
if (read(f,&y,4) != 4) /* gestion d'erreur */  
if (close(f) < 0) /* gestion d'erreur */
```

## fonction "read" plus sûre

```
int safe_read(int f, char* buf, ssize_t size) {  
    while (size > 0) {  
        ssize_t r = read(file, buf, size);  
        if (r == 0) return 0; // erreur  
        if (r < 0) {  
            if (errno == EINTR) continue;  
            return 0; // erreur  
        }  
        buf += r; size -= r;  
    }  
    return 1; // OK  
}
```

# Gestion des erreurs avec exceptions, en Java

## lecture de fichier en Java

```
try {  
    FileReader r = new FileReader("toto");  
    BufferedReader br = new BufferedReader(r);  
    String s = br.readLine();  
    br.close();  
}  
catch (IOException e) {  
    /* toutes les erreurs de fichier */  
}
```

En cas d'erreur de fichier dans le bloc `try`, les instructions suivantes du bloc sont sautées et le programme commence à exécuter le bloc `catch`. En l'absence d'erreur, le bloc `catch` n'est pas exécuté.

- les exceptions **rompent le flot d'exécution normal** du programme ;
- inversion des responsabilités : inutile de tester l'absence d'erreur car la présence d'une erreur est signalée ;
- permet de factoriser le code de gestion des erreurs.



# Créer des exceptions

Il est également possible de signaler soi-même des exceptions :

```
throw new UnsupportedOperationException("Note implemented yet...")
```

En Java, les exceptions sont des **objets** d'une classe descendant de **Throwable** ou d'une de ses nombreuses sous-classes.

Il est facile de définir ses propres exceptions :

```
public class MyException extends Exception {  
    protected Object data;  
    public MyException (String message, Object data) {  
        super(message);  
        this.data = data;  
    }  
}
```

Le filtrage des exceptions permet de ne capturer que les exceptions qui nous intéressent.

```
catch (MyException | MyOtherException e)
```

# Typage et exceptions

## exemple de clause throws

```
public void f() throws MyException;
public void g() throws MyException {
    f();
}
public void h() {
    try {
        f();
    } catch (MyException m) {
    }
}
```

En Java, les méthodes doivent **déclarer** avec **throws** la liste des exceptions que leur appel peut signaler.

- **g** : une exception signalée par **f** peut échapper, il faut le déclarer.
- **h** : les exceptions de **f** sont traitées, nul besoin de **throws** ;

Le compilateur Java vérifie que toutes les exceptions sont bien rattrapées. Cela évite d'oublier des cas d'erreur.

# RunTimeExceptions en Java

Toutes les exceptions ?

**Non**, car Java signale aussi des exceptions qui peuvent “arriver dans le cours normal de l'exécution de la JVM”.

Elles dérivent toutes de **RuntimeException** :

- **ArithmeticException** : division par zéro, etc.
- **ArrayStoreException** : erreur de typage lors d'une écriture dans un tableau ;
- **ClassCastException** : erreur de conversion d'objet, ...

Les exceptions dérivant de **RuntimeException** n'ont pas besoin d'être déclarées par **throws**.

Leur bonne gestion n'est donc **pas vérifiée** par le compilateur.

Choix est très controversé...

(À ne pas faire : signaler exprès une **RuntimeException** pour contourner la vérification de type de Java.)

# try-finally

## exemple de finally

```
BufferedReader br = new BufferedReader(new FileReader(path));
try {
    return br.readLine();
}
catch(IOException e) {
    System.out.println("erreur !");
}
finally {
    if (br != null) br.close();
}
```

Le bloc **finally** est **toujours exécuté**, même en cas d'exception :

- pas d'exception : **finally** est exécuté après le bloc **try** ;
- exception rattrapé dans le **catch** : **finally** est exécuté à la fin du bloc **catch** correspondant ;
- exception non rattrapée dans le **catch** : le bloc **finally** est exécuté, et l'exception est relancée à la fin du bloc **finally**.

⇒ très utile pour gérer proprement les ressources.

# Gestion de ressources en Java 8

Le mécanisme `try-finally`, même sans bloc `catch` est si utile que Java 8 introduit une syntaxe spécifique.

(sucre syntaxique)

## `try-with-resources`

```
try (BufferedReader br = new BufferedReader(new FileReader(path))) {  
    return br.readLine();  
}
```

Est équivalent à :

## `try-finally`

```
BufferedReader br = new BufferedReader(new FileReader(path));  
try {  
    return br.readLine();  
}  
finally {  
    br.close();  
}
```

Il suffit que la ressource `br` implante l'interface `AutoCloseable` et donc une méthode `void close()`.

# Les exceptions dans ILP

ILP supportera la construction :

```
try {  
    throw "Help!";  
}  
catch (e) {  
    /* e utilisable ici */  
}  
finally {  
}
```

ILP étant un langage non typé :

- `throw` peut utiliser toute valeur comme exception ;
- `catch` gère toutes les exceptions (pas de filtrage) ;
- inutile de déclarer quelles exceptions peuvent échapper.

# Flot de contrôle (1/3)

**Problème principal** : quelle est la prochaine instruction exécutée ?

Où “saute-t-on” après un `throw`, à la fin d'un bloc `catch` ou `finally` ?

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

Résultat : 1 3.

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

Résultat : 3 1.

# Flot de contrôle (2/3)

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

Résultat : 10.

- throw dans un catch
- throw dans un finally

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

Résultat : 111.



# Flot de contrôle (3/3)

## exemple

```
f(x) {  
    if (x < 0) throw "Erreur";  
    x + 1;  
}  
g(y) {  
    try {  
        f(y);  
    } catch (e) {  
    }  
}
```

## exemple (cont.)

```
h() {  
    try {  
        f(u);  
        g(v);  
    }  
    } catch (e) {  
    }  
}
```

Exceptions et appels de fonctions :

le gestionnaire d'exception ne peut pas être déterminé statiquement,  
il dépend de la **pile des appels**.

# Durée de vie et portée lexicale des variables

## exemple

```
void f() {
    int x;
    // x est accessible
    // x et z existent
    // y l'exécution vient de y
}
void g() {
    int y;
    // y est accessible
    // y et z existent parfois
    f();
}
```

## exemple (cont.)

```
void h() {
    int z;
    // z est accessible
    // z existe
    f();
    g();
    {
        int t;
        // z et t sont accessibles
        // z et t existent
    }
}
```

Portée lexicale : visibilité d'une variable, notion **statique**.

Durée de vie : du début de l'exécution du bloc à la sortie du bloc.

Structure de pile, notion **dynamique**.

# Durée de vie et exceptions

## exemple

```
void f() {
    int x;
    {
        int y;
        if (...) {
            throw "Bla"; // désalloue ???
        }
        return ; // désalloue x et y
    } // désalloue y
} // désalloue x
```

## exemple (cont.)

```
void g() {
    int z;
    f();
}
void h() {
    try {
        f();
        g();
    }
    catch (e) {
    }
}
```

La présence des exceptions complique la gestion des variables.

Quand désallouer une variable ?

- sortie normale de bloc, déterminée **statiquement** ;
- **return**, déterminé **statiquement** ;
- **throw**, déterminé **dynamiquement** ;  
dépend de la pile d'appels, de la position du **catch**.

# Les limites des exceptions (1/2)



**Premier vol du lanceur Ariane 5, 4 juin 1996.**

Le logiciel est programmé en Ada, langage très sûr, avec des exceptions.

# Les limites des exceptions (2/2)



**Premier vol du lanceur Ariane 5, 4 juin 1996**  
**40 secondes plus tard...**

Un dépassement de capacité dans une conversion de flottant 64-bit en entier 16-bit génère une exception.

L'exception n'est pas rattrapée, le calculateur s'arrête.

Il y a redondance, mais les autres calculateurs exécutent le même programme.

Tous les calculateurs s'arrêtent.

La fusée s'autodétruit.

# L'erreur qui valait un milliard

Tony Hoare, prix Turing 1980, inventeur de Quicksort, écrit en 2009 :

*I call it my **billion-dollar mistake**. It was **the invention of the null reference** in 1965. At that time, I was designing the first comprehensive type system for references in an object oriented language (ALGOL W). My goal was to ensure that all use of references should be absolutely safe, with checking performed automatically by the compiler. But I couldn't resist the temptation to put in a null reference, simply because it was so easy to implement. This has led to innumerable errors, vulnerabilities, and system crashes, which have probably caused a billion dollars of pain and damage in the last forty years.*

Ce défaut de conception est repris dans Java.

`NullPointerException` est une `RuntimeException`, et échappe donc aux règles de typage de Java.

Il faut se contenter d'annotations souvent purement décoratives `@Nullable`.

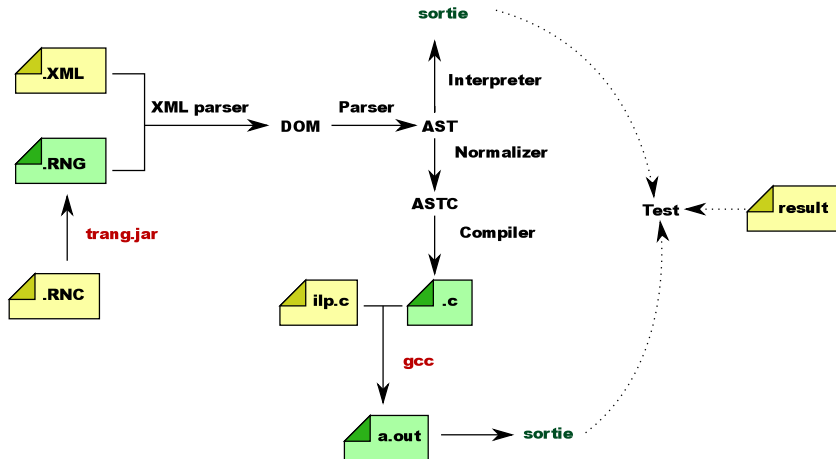
Tony Hoare est aussi un des pionniers des **méthodes formelles**, permettant la vérification des logiciels !

À faire : venir écouter Xavier Leroy au Colloquium, Amphi 25, ce soir 18h !

## Rappel : extension d'ILP

---

# Structure d'ILP





# Étapes d'une extension (utile aussi en TME)

Extension de la syntaxe :

- enrichir la grammaire **RNC** et générer le **RNG** ;  
`java -jar $ILPDIR/Java/jars/trang.jar grammaire.rnc grammaire.rng`
- ajouter des nœuds **IAST**, **AST**, **ASTC** ;
- mettre à jour la classe **Parser**.

Extension de la base de tests (`.xml`, `.result`, `.print`).

Extension des visiteurs : **IASTvisitor** :

- classes **Interpreter**, **Normalizer**, **Compiler**.

Extension des primitives et opérateurs de l'interprète.

Extension de la bibliothèque d'exécution C :

- type **ILP\_Object** dans `ilp.h` ;
- primitives dans `ilp.c`.

Toutes les étapes ne sont pas nécessaires à chaque extension !

# Règles de programmation pour les extensions

En Java :

- **pas de modification du code existant** ;
- **nouvelles classes** dans des “packages” séparés  
`comp.paracamplus.ilpX...` ;
- réutilisation par **héritage** ;
- **motifs** visiteurs et composite facilitant l'extensibilité.

En RelaxNG :

- nouveau fichier qui fait un `include` de la grammaire à étendre ;
- extension des règles avec `|=`.

En C :

- `ilp.c` et `ilp.h` implantent déjà tout ILP1 à ILP4...
- difficile d'étendre un type `struct` → autorisation de modifier `ilp.h` ;
- déclarer et définir les fonctions **dans des .c et .h séparés**.

# Syntaxe des exceptions

---

# Syntaxe abstraite (1/2)

grammar3.rnc

```
expression |=  
  try  
  | codedefinitions  
  | lambda  
  
try = element try {  
  element body { expression + },  
  ( catch  
    | finally  
    | ( catch, finally )  
  )  
}
```

Clause `try` avec :

- soit un `catch` ;
- soit un `finally` ;
- soit un `catch` et un `finally`.

# Syntaxe abstraite (2/2)

grammar3.rnc

```
catch = element catch {  
    attribute exception  
    { xsd:Name - ( xsd:Name { pattern = "(ilp|ILP)" } ) },  
    expression +  
}  
  
finally = element finally {  
    expression +  
}
```

Le `catch` a un argument “nom de variable”; c'est un `lieur`.  
(comme une variable de bloc ou un argument formel de fonction)

Pas de syntaxe spéciale pour `throw`, ce sera une primitive.  
La syntaxe `invocation` sera donc utilisée, comme pour `print`.

# Exemple XML

```
<try>
  <body>
    <invocation>
      <function><variable name='throw' /></function>
      <arguments><string>Bla</string></arguments>
    </invocation>
  </body>
  <catch exception='exc'>
    <invocation>
      <function><variable name='print' /></function>
      <arguments><variable name='exc' /></arguments>
    </invocation>
  </catch>
  <finally>
    <invocation>
      <function><variable name='print' /></function>
      <arguments><string>finally...</string></arguments>
    </invocation>
  </finally>
</try>
```

# Arbre syntaxique : interface (1/2)

## IASTtry.java

```
package com.paracampus.ilp3.interfaces;  
public interface IASTtry extends IASTinstruction {  
    IASTexpression getBody ();  
    @Nullable IASTlambda getCatcher ();  
    @Nullable IASTexpression getFinallyer ();  
}
```

- Simple miroir de la syntaxe abstraite...
- L'implantation `comp.paracampus.ilp3.ast.ASTtry` est un simple conteneur :
  - champs `private` constants pour `body`, `catcher`, `finallyer` ;
  - constructeur `public ASTtry(IASTexpression body, IASTlambda catcher, IASTexpression finallyer)` ;
  - accesseurs `getBody`, etc. (suivant l'interface) ;
  - visiteur en trois versions : `ilp1`, `ilp2`, `ilp3`.  
(nous y reviendrons dans quelques transparents)

# Arbre syntaxique : interface (2/2)

## IASTLambda.java

```
package com.paracampus.ilp3.interfaces;  
public interface IASTLambda extends IASTExpression {  
    IASTvariable[] getVariables();  
    IASTExpression getBody();  
}
```

- En première approximation, un conteneur pour :
  - un ensemble de variables locales;
  - une expression utilisant ces variables locales.
- Cette semaine, pour notre utilisation dans `IASTtry` :
  - `getVariables` est la variable d'exception;
  - `getBody` est le gestionnaire d'exception utilisant la variable.
- La semaine prochaine : utiliser `IASTLambda` pour implanter des fonctions de première classe.

À faire également : étendre les fabriques `IParserFactory`, `ASTFactory`.



# Interprétation des exceptions

---

# Principe d'interprétation

L'interprète est écrit en Java, qui a des exceptions.  
Nous en profitons !

- Une exception ILP est implantée par une exception Java.
- `try`, `catch`, `finally` d'ILP sont implantés avec `try`, `catch`, `finally` de Java.

**Attention** cependant, deux sources d'exceptions :

- les exceptions lancées par l'interprétation de `throw` d'ILP, lancées par un programme ILP ;
- les exceptions en cas d'erreur Java dans l'interprète par la faute du programme ILP ou à cause d'un bug dans l'interprète.

(exemple : division par zéro)

Question : un programme ILP peut-il capturer une exception Java ?

# Rappel : ajout d'une primitive à l'interprète

Créer une classe `Throw`

`Throw.java` (schéma)

```
package com.paracamplus.ilp3.interpreter.primitive;
public class Throw extends UnaryPrimitive {
    public Throw () { super("throw"); }
    public Object apply (Object value) throws ThrownException {
        // transparent suivant
    }
}
```

et enregistrer la primitive dans `GlobalVariableStuff`.

(dans `com.paracamplus.ilp3.interpreter`, pas `compiler`!)

`GlobalVariableStuff.fillGlobalVariables`

```
env.addGlobalVariableValue(new Throw());
```

# Évaluation de throw

## Throw.java (implantation)

```
public static class ThrownException
extends EvaluationException {
    private final Object value;
    public ThrownException (Object value) {
        super("Throwing value");
        this.value = value;
    }
    public Object getThrownValue () { return value; }
}

public Object apply(Object value) throws ThrownException {
    ThrownException exc = new ThrownException(value);
    throw exc;
}
```

La classe `ThrownException` des exceptions ILP est une **classe interne** à `Throw`; elle stocke la valeur ILP de l'exception.

# Extension du visiteur d'interprétation (1/4)

Nouvelle interface visiteur :

## IASTvisitor.java

```
package com.paracampus.ilp3.interfaces;  
public interface IASTvisitor  
    <Result, Data, Anomaly extends Throwable>  
    extends com.paracampus.ilp2.interfaces.IASTvisitor  
        <Result, Data, Anomaly>  
{  
    Result visit(IASTtry iast, Data data) throws Anomaly;  
    // ...  
}
```

Ajout du support pour IASTtry.

# Extension du visiteur d'interprétation (2/4)

## ASTtry.java

```
package com.paracamplus.ilp3.ast;
import com.paracamplus.ilp3.interfaces.IASTvisitor;
public class ASTtry extends ASTinstruction
implements IASTtry, IASTvisitable {
    public </*@...*/> Result accept(IASTvisitor</*@...*/> visitor,
    Data data) throws Anomaly {
        return visitor.visit(this, data);
    }
    public </*@...*/> Result accept(
    com.paracamplus.ilp1.interfaces.IASTvisitor</*@...*/> visitor,
    Data data) throws Anomaly {
        return ((IASTvisitor</*@...*/>) visitor).visit(this, data);
    }
    // idem pour com.paracamplus.ilp2.interfaces.IASTvisitor
}
```

Un **cast** peut être nécessaire pour s'assurer du type du visiteur !

Question : une **ClassCastException** est-elle possible ?

# Extension du visiteur d'interprétation (3/4)

## Interpreter.java (début)

```
public Object visit(IASTtry iast, ILexicalEnvironment lexenv)
throws EvaluationException {
    Object result = Boolean.FALSE;
    IFunction fcatcher = null;
    IASTlambda catcher = iast.getCatcher();
    if ( null != catcher ) {
        fcatcher = (IFunction) catcher.accept(this, lexenv);
    }
    try {
        result = iast.getBody().accept(this, lexenv);
    }
    catch (ThrownException exc) {
        if ( null != fcatcher ) {
            Object value = exc.getThrownValue();
            fcatcher.apply(this, new Object[]{ value });
        } else { throw exc; }
    }
}
```

Note : `catcher.accept` renvoie un objet qui, en s'évaluant avec `apply`, lance l'interprétation du corps du gestionnaire d'erreur. Plus de détails sur ce mécanisme la semaine prochaine.

# Extension du visiteur d'interprétation (4/4)

## Interpreter.java (suite)

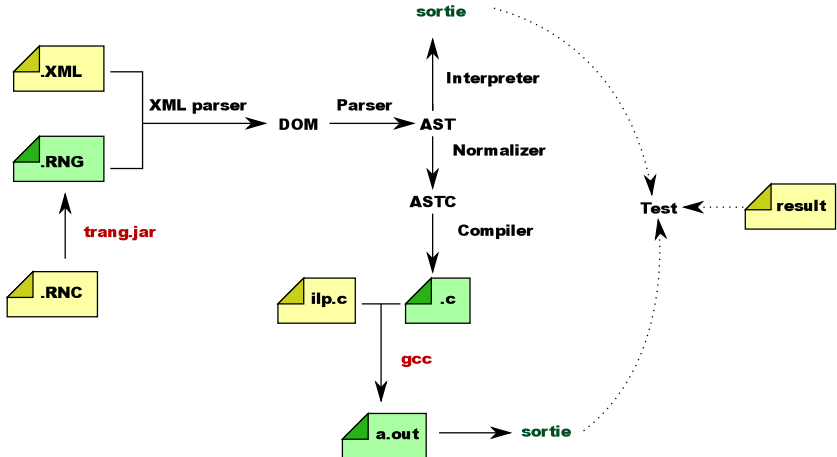
```
catch (EvaluationException exc) {
    if ( null != fcatcher ) {
        fcatcher.apply(this, new Object[]{ exc });
    } else { throw exc; }
}
catch (Exception exc) {
    if ( null != fcatcher ) {
        EvaluationException e = new EvaluationException(exc);
        fcatcher.apply(this, new Object[]{ e });
    } else { throw exc; }
}
finally {
    IASTexpression finallyer = iast.getFinallyer();
    if ( null != finallyer ) {
        finallyer.accept(this, lexenv);
    }
}
return result;
}
```



# Compilation des exceptions

---

# Rappel : structure d'ILP



# Rappel : normalisation de l'AST

Le compilateur commence par transformer l'AST en ASTC.

Différence essentielle entre AST et ASTC : l'encodage des variables.

Note : il existe trois notions d'égalité entre variables :

- elles ont le même nom, ou
- elles représentent la même variable dans le programme ILP, ou
- elles sont représentées par le même nœud AST : == en Java.

Différence entre AST et ASTC :

- deux nœuds ASTvariable != peuvent représenter la même variable ou des variables différentes (même si elles ont le même nom) ;
- deux nœuds ASTCvariable != représentent des variables différentes.

Exemple : `let x = 2 in (let x = 3 in x + 1) * x`

⇒ ASTC facilite la manipulation d'ensembles de variables et d'environnements  
e.g., utilisation de `Set<IASTClocalVariable>` dans `FreeVariableCollector`.

# Normalisation des exceptions (1/3)

La classe `Normalizer` se charge de transformer l'AST en ASTC :

- parcours récursif de l'AST avec un visiteur ;
- reconstruction d'un nœud après avoir visité les sous-nœuds ;
- maintient de l'environnement : nom de variable → `IASTCvariable` dans la classe `INormalizationEnvironment` ;
- la normalisation résout donc la **portée lexicale** des variables !

Ajouts nécessaires pour le support des exceptions :

- pas de `ASTCtry`, nous réutilisons `ASTtry`, mais il faut quand même normaliser les sous-expressions du nœud puis reconstruire le `ASTtry` ;
- `ASTlambda` transformé en `ASTClambda`, et mise à jour de l'environnement avec les variables introduites par le nœud.

# Normalisation des exceptions (2/3)

## Normalizer.java (début)

```
package com.paracamplus.ilp3.compiler.normalizer;

public IASTexpression visit(IASTtry iast, INormalizationEnvironment env)
throws CompilationException {
    IASTexpression newbody = iast.getBody().accept(this, env);
    IASTlambda newcatcher = null;
    IASTlambda catcher = iast.getCatcher();
    if ( catcher != null ) {
        newcatcher = (IASTlambda) catcher.accept(this, env);
    }
    IASTexpression newfinallyer = null;
    IASTexpression finallyer = iast.getFinallyer();
    if ( finallyer != null ) {
        newfinallyer = finallyer.accept(this, env);
    }
    return ((INormalizationFactory)factory).
        newTry(newbody, newcatcher, newfinallyer);
}
```

# Normalisation des exceptions (3/3)

## Normalizer.java (suite)

```
public IASTexpression visit(IASTlambda iast, INormalizationEnvironment
env) throws CompilationException {
    IASTvariable[] variables = iast.getVariables();
    IASTvariable[] newvariables = new IASTvariable[variables.length];
    INormalizationEnvironment newenv = env;
    for (int i=0; i<variables.length; i++) {
        IASTvariable variable = variables[i];
        IASTvariable newvariable =
            factory.newLocalVariable(variable.getName());
        newvariables[i] = newvariable;
        newenv = newenv.extend(variable, newvariable);
    }
    IASTexpression newbody = iast.getBody().accept(this, newenv);
    return ((INormalizationFactory)factory).
        newLambda(newvariables, newbody);
}
```

# Support pour les exceptions en C

---

# Sauts longs en C

Motif de programmation :

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

jmp_buf buf;
if (setjmp(buf) == 0) {
    // code
    longjmp(buf, 1);
    // non accessible
} else {
    // destination du longjmp
}
```

- établissement d'un point de restauration avec `setjmp`;
- saut avec `longjmp`;
- la valeur de retour de `setjmp` distingue l'établissement du point de l'arrivée au point depuis `longjmp`.



# Sauts longs en C : cas inter-procédural (1/2)

`longjmp` peut aussi traverser les appels de fonctions !

```
jmp_buf buf;

void f() {
    if (setjmp(buf) == 0) {
        g();
        // cas sans erreur (A)
    }
    else {
        // gestion de l'erreur (B)
    }
    // code commun (C)
}
```

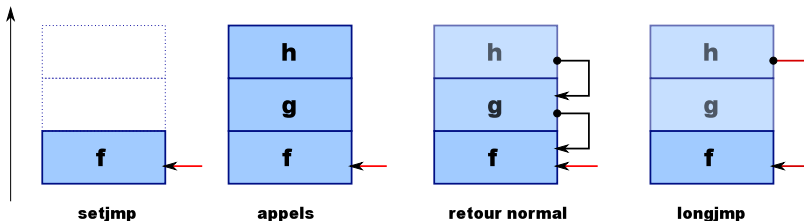
```
void g() {
    h();
    // cas sans erreur (D)
}

void h() {
    if (erreur()) {
        longjmp(buf);
    }
    // cas sans erreur (E)
}
```

- flot d'exécution sans erreur : E, D, A, C
- flot d'exécution sans erreur : B, C

# Sauts longs en C : cas inter-procédural (2/2)

sens des  
appels



## La pile d'appels

⇒ parfait pour implanter les exceptions en C !

# Complications

- Une valeur `IPL_Object`, à passer du `try` au `catch` ;

or `longjmp` ne permet de passer qu'un entier  
(et la valeur 0 est réservée).

⇒ Une variable globale, `ILP_current_exception`, s'en charge.

- Les blocs `try` peuvent être imbriqués ;

nécessité de gérer plusieurs gestionnaires `jmp_buf` ;  
`throw` doit ensuite trouver le bon gestionnaire

⇒ une liste chaînée de `jmp_buf` est utilisée,  
pointée par la variable globale `ILP_current_catcher` ;  
`try` empile / dépile les gestionnaires,  
`throw` utilise la tête de liste (le gestionnaire le plus récent).

# Motif de code C généré (1/3)

code généré (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 du try */
        ILP_current_exception = NULL;
    }
    /* fin normale du bloc try, ou exception dans le bloc */
}
```

- un bloc allouant un nouveau gestionnaire comme variable locale ;
- empilement du gestionnaire ;
- fin du bloc `try` sans exception : `ILP_current_exception = NULL` ;
- saut en fin de bloc `try` par `throw` : `ILP_current_exception`  $\neq$  `NULL`.

# Motif de code C généré (2/3)

code généré (catch)

```
ILP_reset_catcher(current_catcher);
if (NULL != ILP_current_exception) {
    if (0 == setjmp(new_catcher._jmp_buf)) {
        ILP_establish_catcher(&new_catcher);
        {
            ILP_Object exc2 = ILP_current_exception;
            ILP_current_exception = NULL;
            /* corps du catch */
        }
    }
}
/* fin normale du bloc catch ou exception dans le bloc */
```

- valeur d'exception `exc2` fournie au corps du `catch`;
- gestionnaire spécial : en cas d'exception dans le corps du `catch`, on sort du bloc en se souvenant de l'exception dans `ILP_current_exception`.

# Motif de code C généré (3/3)

code généré (finally)

```
ILP_reset_catcher(current_catcher);  
/* corps du finally */  
if ( NULL != ILP_current_exception ) {  
    ILP_throw(ILP_current_exception);  
}  
}
```

- dépilement du gestionnaire (`ILP_reset_catcher`);
- exécution du bloc `finally`;
- en cas d'exception dans le `catch`,  
l'exception est relancée **après** avoir exécuté le bloc `finally`;
- en cas d'exception dans le bloc `finally`,  
c'est le gestionnaire englobant qui est utilisé.

# Bibliothèque d'exécution (1/2)

## ilp.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;
```

## ilp.c (début)

```
/** Install a new catcher. */  
void ILP_establish_catcher (struct ILP_catcher *new_catcher) {  
    new_catcher->previous = ILP_current_catcher;  
    ILP_current_catcher = new_catcher;  
}  
  
/** Reset an old catcher. */  
void ILP_reset_catcher (struct ILP_catcher *catcher) {  
    ILP_current_catcher = catcher;  
}
```

# Bibliothèque d'exécution (2/2)

## ilp.c (suite)

```
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;

/** Raise exception. */
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 ILP_die("longjmp botch");
}
```



# Exceptions générées par la bibliothèque d'exécution

## ilp.c

```
ILP_Object ILP_domain_error(char *message, ILP_Object o) {
    snprintf(ILP_the_exception._content.asException.message,
             ILP_EXCEPTION_BUFFER_LENGTH,
             "Domain error: %s\nCulprit: 0x%p\n",
             message, (void*) o);
    fprintf(stderr, "%s",
            ILP_the_exception._content.asException.message);
    ILP_the_exception._content.asException.culprit[0] = o;
    ILP_the_exception._content.asException.culprit[1] = NULL;
    return ILP_throw((ILP_Object) &ILP_the_exception);
}

ILP_Object ILP_make_modulo (ILP_Object o1, ILP_Object o2) {
    if ( ILP_isInteger(o1) ) {
        /*...*/
    } else {
        return ILP_domain_error("Not an integer", o1);
    }
}
```

# Retour sur le compilateur

---

# Rappel : ajout d'une primitive au compilateur

La primitive à un argument `throw` correspond à la fonction `ILP_throw` de la bibliothèque d'exécution C.

Enregistrement nécessaire de la primitive dans `GlobalVariableStuff` (dans `com.paracampus.ilp3.compiler`).

**`GlobalVariableStuff.fillGlobalVariables`**

```
env.addGlobalFunctionValue(new Primitive("throw", "ILP_throw", 1));
```

# Générateur de code C

Enrichissement du visiteur [Compiler](#).

## Compiler.java (extrait)

```
public Void visit(IASTtry iast, Context context)
throws CompilationException {
    emit("{ struct ILP_catcher* current_catcher = "
        "ILP_current_catcher; \n");
    emit(" struct ILP_catcher new_catcher; \n");
    emit(" if ( 0 == setjmp(new_catcher._jmp_buf) ) { \n");
    emit("     ILP_establish_catcher(&new_catcher); \n");
    iast.getBody().accept(this, context);
    emit("     ILP_current_exception = NULL; \n");
    emit(" }; \n");
    /* suite omise ... */
```

(lire le code par vous même, rien de bien compliqué quand on a compris le motif de code généré)

# Point d'entrée du C généré

code C généré (fixe)

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

Emballer le programme complet dans un gestionnaire d'exception global. Cela permet au programme C de terminer normalement en cas d'exception non rattrapée (pas de **ILP\_die**).

# Résumé

Nous avons appris à :

- implanter le **contrôle non-local** en C, grâce à **setjmp**, **longjmp** ;
- gérer des **gestionnaires d'exceptions imbriqués** grâce à une liste.

Nous avons ajouté du code au compilateur et à la bibliothèque d'exécution sans modifier ce qui existe déjà.

Les exceptions sont donc **indépendantes** du reste du langage.

Bien comprendre la distinction entre **statique** et **dynamique** :

- ce qui est fixé à la compilation est **statique** ;
- ce qui nécessite du travail à l'exécution est **dynamique** ;
- la portée lexicale (quelle variable est visible) est **statique** ;
- la durée de vie des variables est **dynamique** ;
- quel gestionnaire d'exception est utilisé est **dynamique**.

# Bonus

---

# Efficacité de la gestion des exceptions

Dans notre implantation, l'emploi des exceptions ralentit le programme :

- Lors d'un `throw`.

Ce n'est pas gênant : signaler une exception est supposé être un évènement rare, exceptionnel.

Ce coût est sûrement négligeable par rapport à celui du traitement de l'erreur (exécution du `catch`).

- À chaque emploi d'un bloc `try/catch/finally`.

Dans ILP : empilement/dépilement de la liste des gestionnaires.

C'est bien plus gênant !

Les blocs `try` peuvent (doivent ?) être nombreux, même si le `catch` n'est jamais exécuté.

Il ne faut pas pénaliser les programmes qui ne font pas d'erreur.

Il ne faut pas pénaliser la programmation défensive (gérer trop de cas d'erreur, plutôt que pas assez).



# Méthode à “coût zéro” (1/2)

La “vraie” pile d’un programme (simplifiée).

```
void main() {
11: f(99);
12:
}
void f(int x) {
    try {
13:         g(x+1);
14:         /* ... */
    }
    catch {
15:         /* ... */
    }
16: g(x+2);
17: /* ... */
}
void g(int y) {
    throw(y);
}
```

s1:	arguments	x
s2:	adresse de retour	&l2
s3:	pile de l'appelant	&s?
s4:	variables locales	—
s5:	arguments	y
s6:	adresse de retour	&l4
s7:	pile de l'appelant	&s3
s8:	variables locales	—

- main empile 99
- main empile &l2
- main saute à &l3
- f empile le pointeur de pile
- f met le pointeur de pile à s4
- f empile 100 (x+1)
- f empile &l4...

# Méthode à “coût zéro” (2/2)

Principe : au moment du `throw`

- remonter dans la pile la séquence des appels (`g`, `f`, `main`) ;
- avec pour chaque appel le site précis (`adresse &l4`, `&l2`) ;
- jusqu'à retrouver un site ayant un gestionnaire `catch` ;
- inutile donc de maintenir à part une liste des gestionnaires.

Possible en assembleur, mais difficile à implanter en C !

Le compilateur génère une table site d'appel → gestionnaire :

- la table est statique ;
- le recherche dans la table est dynamique.

⇒ Un bloc `try` a un **coût nul à l'exécution**.  
Un `throw` est un peu plus complexe et coûteux.