Les fonctions de première classe

MU4IN501 – DLP : Développement d'un langage de programmation
Master STL, Sorbonne Université

Antoine Miné

Année 2020–2021

Cours 7
17 novembre 2020

Plan général

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

Par étapes, avec ajout progressif de fonctionnalités.

- **ILP1** : langage de base
- **ILP2** : ajout des boucles, affectations, fonctions globales
- **ILP3** : ajout des exceptions et des fonctions de première classe
- **ILP4** : ajout des classes et des objets

Fonctions de première classe

ILP2 ne supporte que les fonctions globales (comme en C).

```
function f(x) ( x + 1 );
f(2)
```

ILP3 ajoute les fonctions de première classe, c'est à dire :
les fonctions en tant que valeurs (comme les entiers, chaînes, etc.)

Nous pouvons alors :

- stocker des fonctions dans des variables et les utiliser

  `let x = f in x(2)`

- passer des fonctions en argument ou les retourner

  `function apply2(f) ( f(2) )...`

- déclarer des fonctions locales

  `function g(x) (x + y) in g(2)`

- créer des fonctions anonymes : les lambda expressions

  `apply2 (lambda (x) (x + y))`

  équivalent à retourner une fonction locale : `apply2 (function g(x) (x + 2) in g)`

$\Longrightarrow$ (illusion de) création dynamique de fonctions

Plan du cours

Ce cours : les fonctions de première classe dans ILP3

- introduction aux fonctions locales, anonymes et de première classe
- extension de la syntaxe et de l'AST (facile)
- notion d'environnement (rappel) et de clôture (nouveauté)
  (problème : où sont mes variables ?)
- implantation naïve de l'environnement (interprète)
- implantation optimisée (compilateur)
- distinction entre aspects statiques et dynamiques
- analyse statique
  (variables libres, variables liées)
- ajouter les lambda expressions à un langage qui ne les supporte pas (C)

Les fonctions

Aspects statiques en C : la portée des variables

```
int a;
void f() {
    int x;
    // a et x sont accessibles
}
void g() {
    int y;
    f();
    {
        int z;
        // a, y et z sont accessibles
    }
}
```

- toutes les fonctions sont globales ;
- une fonction n'accède directement qu'aux variables globales et à ses variables locales dans leur portée ;

$\Longrightarrow$ résolution statique (*linking* : édition de liens).

Portée lexicale : l'apport d'ALGOL 60

```
int x = 1;
void g() {
    int x = 12;
    f();
}
```

```
void f() {
    print(x);
}
```

- portée dynamique : variable créée la plus récemment au moment de l'exécution ⇒ dynamique

  ⟹ variable x locale à g ⇒ affiche 12
- portée lexicale : variable la plus proche dans l'imbrication des blocs syntaxiques { }

  dépend du source du programme ⇒ statique

  ⟹ variable x globale ⇒ affiche 1

Les premiers langages utilisent la portée dynamique.

comportement plus facile à implanter : environnent = table d'association, aucun travail à la compilation

ALGOL 60 propose la portée lexicale, et les autres langages suivent.

comportement plus facile à prévoir et plus robuste, mais la compilation nécessite une analyse statique

Aspects dynamiques en C : les pointeurs

```
void f(int* y) {
    *y = 12;
    // accès direct à y
    // accès indirect à x
}
void g() {
    int x;
    f(&x);
}
```

```
void f(void (*p)(int)) {
    (*p)(12);
    // appel indirect à h
}
void h(int v) {
}
void g() {
    f(&h);
}
```

Quelle variable est accédée par *y ?
Quelle fonction est appelée par *p ?

- les pointeurs sur les données permettent d'accéder à des variables hors de leur portée (mais toujours pendant leur durée de vie) ;
- les pointeurs de fonction permettent les appels indirects ;

$\Longrightarrow$ résolution dynamique.

Fonctions emboîtées en Pascal

```
function F(x:integer) : integer
begin
    function G(y:integer) : integer
    begin
        if x > y then return y;
        return G(2*y);
    end;
    return G(1);
end;
```

- G est emboîtée dans la définition de F ;
- G a accès aux arguments et variables locales de F.

Règle d'accès aux variables : portée lexicale.

Il suffit de trouver le bloc begin / end :

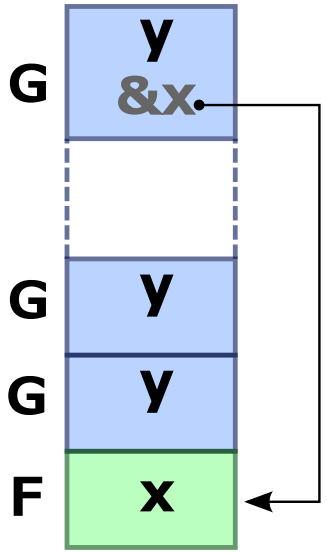
- déclarant x (variable locale ou argument formel de fonction)
- englobant l'accès (l'accès est dans la durée de vie de x)
- le plus petit (le plus proche de l'accès à x)

$\Longrightarrow$ résolution statique.

Implantation des fonctions emboîtées : traduction en C

```
int G(int y, int* xref) {
    if (*xref > y) return y;
    return G(2*y, xref);
}

int F(int x) {
    return G(1, &x);
}
```



Avec des fonctions globales et un modèle d'exécution par pile, retrouver la position en mémoire de `x` n'est pas évident.

L'empilement dynamique des appels de fonctions ne correspond pas forcément à l'imbrication des blocs de portée lexicale.
(par exemple, en cas d'appels récursifs)

Exemple d'implantation : passer un pointeur sur `x` en argument caché.

Langages fonctionnels : OCaml

```
let rec map f list =
    if list = [] then [] else
    concat (f (hd list)) (map f (tl list))

let add x = map (fun y -> x + y)

add 2 [1;2;3]
```

- `map` prend une fonction `f` en argument (et une liste `list`) ;
  (et applique `f` à chaque élément de la liste)
- `add` retourne une fonction ;
  (prend une liste en argument, ajoute `x` à chaque élément de la liste)
- `fun y -> x + y` est une fonction anonyme ;
  (ajoute `x` à `y` et retourne le résultat)
- lors de l'utilisation de la fonction anonyme dans `map`, la valeur de `x` utilisée est l'argument passé à `add`, donc `2` ;

$\Longrightarrow$ toujours la règle de portée lexicale.

Fonction = code + environnement

En C, une fonction est assimilée à son code exécutable,
c'est un objet constant, fixé à la compilation (ou un pointeur vers cet objet).

En OCaml, `let add x = fun y -> x + y`
crée une nouvelle fonction dynamiquement à chaque appel de `add`.

Mais il n'y a pas réellement de création dynamique de code exécutable...

Une valeur fonction est en réalité une structure de données contenant :

- un pointeur vers un morceau de code
  $\Rightarrow$ quoi exécuter ?
- un environnement
  $\Rightarrow$ où trouver la valeur des variables ?

`add 2` et `add 3` partagent le même code,
mais ont des environnements différents, donnant une valeur différente à `x`.

En Java : les fonctions comme objets

Avant Java 8, Java n'a pas de fonctions de première classe, mais elles sont simulables par des objets.

java.util.Comparator

```
interface Comparator<T>
{
    int compare(T o1, T o2);
    boolean equals(Object obj);
}
```

java.lang.Object.Collections

```
public class Collections
{
    static <T> void sort(List<T> list, Comparator<? super T> c);
}
```

- un objet `Comparator` regroupe deux fonctions ;
- il est passé en argument à la fonction `sort`.

En Java : classes anonymes, fonctions anonymes

classe anonyme

```
List<String> list;
Collections.sort(list,
                 new Comparator<String>()
                 {
                     public int compare(String x, String y)
                     { ... }
                 });
```

La notion de classe anonyme correspond alors
à celle de fonction anonyme.

En Java : portée lexicale

classe interne dans une méthode

```
public class True
{
    public Comparator<String> m()
    {
        final int x = 12;
        class C implements Comparator<String>
        {
            public int compare(String a, String b)
            {
                if (x > 19) // ...
            }
        };
        return new C();
    }
}
```

La classe anonyme est un cas particulier de classe interne ou locale.
Toutes deux peuvent apparaître dans une classe ou une méthode.

Elles accèdent à tous les champs et les variables locales visibles, en suivant les règles de portée lexicale.

Les classes internes et locales sont compilées comme des classes séparées (`MyClass$C` ou `MyClass$1`), avec des champs cachés pour accéder aux champs et variables des portées englobantes.

En Java 8 : les lambda expressions

tri avec des lambdas

```
Collections.sort(list, (p1, p2) -> p1.compareTo(p2));
```

streams

```
import java.util.stream.*;
List<String> list;
list.stream()
    .filter(s -> s.compareTo("X") > 0)
    .collect(Collectors.toList());
```

Syntaxe plus légère pour les fonctions anonymes (inférence de type).

Implantation plus efficace que par la création de classes anonymes.

Bibliothèque standard enrichie avec des aspects fonctionnels.

En ILP

Ajout de deux constructions syntaxiques :

- codéfinitions :

```
function even(x) (x = 0 || odd (x-1))
and function odd(x) (x < 0 || even (x-1))
in ...
```

  fonctions locales,
  mutuellement récursives

- lambdas :

```
lambda (x) (x + 1)
```

  fonctions anonymes,
  utilisables dans toutes les expressions

La gestion des invocations est enrichie pour traiter le cas où l'argument est une expression complexe, s'évaluant en une fonction, et pas juste un nom de fonction codé « en dur ».

Grammaire et AST

Syntaxe concrète

ILPMLgrammar3.g4

```
// déclaration de fonction locale nommée
localFunDef
returns [com.paracamplus.ilp3.interfaces.IASTnamedLambda node]
    : 'function' name=IDENT '(' vars+=IDENT? (',' vars+=IDENT)* ')'
        body=expr
    ;
expr
returns [com.paracamplus.ilp1.interfaces.IASTexpression node]
    : ...
    // déclaration de fonctions locales (mutuellement récursives)
    | defs+=localFunDef ('and' defs+=localFunDef)* 'in' body=expr
    # Codefinitions


    // fonction anonyme
    | 'lambda' '(' vars+=IDENT? (',' vars+=IDENT)* ')' body=expr
    # Lambda
```

Arbre syntaxique : interfaces (1/2)

IASTcodefinitions.java

```
package com.paracamplus.ilp3.interfaces;

public interface IASTcodefinitions extends IASTexpression
{
    IASTnamedLambda[] getFunctions();
    IASTexpression getBody();
}
```

- ensemble de fonctions (nommées) mutuellement récursives ;
- et un corps (portée des fonctions).

Note : les définitions locales sont des expressions ;
les définitions globales `IASTfunctionDefinition` ne le sont pas !

Arbre syntaxique : interfaces (2/2)

IASTlambda.java

```
public interface IASTlambda extends IASTexpression
{
    IASTvariable[] getVariables();
    IASTexpression getBody();
}
```

IASTnamedLambda.java

```
public interface IASTnamedLambda extends IASTlambda
{
    IASTvariable getFunctionVariable();
}
```

- `IASTlambda` = ensemble de variables (arguments de la fonction) + corps (portée des arguments) ;
- `IASTnamedLambda` = `IASTlambda` + nom de fonction.

Rien à dire sur les classes implantant ces interfaces : `ASTlambda`, `ASTnamedLambda`, `ASTcodefinitions` sont de simples conteneurs. . .

Interprète

Rappel : les environnements

L'environnement associe une valeur à chaque variable.

Il est disponible à tout visiteur de nœud lors de l'interprétation.

L'environnement est décomposé en :

- l'environnement global `IGlobalVariableEnvironment` ;
- l'environnement lexical `ILexicalEnvironment`.

Les opérateurs ont aussi leur propre environnement global constant `IOperatorEnvironment`, et pas d'environnement lexical car on ne peut pas les redéfinir en ILP.

Rappel : environnement lexical

Signatures des principales opérations.

LexicalEnvironment.java

```
package com.paracamplus.ilp1.interpreter;
public class LexicalEnvironment implements ILexicalEnvironment
{
    public Object getValue(IASTvariable key) ...
    public void update(IASTvariable key, Object value) ...
    public ILexicalEnvironment extend(IASTvariable variable, Object value) ...
}
```

Liste chaînée d'associations : variable → valeur.

- évolue durant l'interprétation, au gré des portées lexicales ;
- s'enrichit par les déclarations locales ;
  et s'appauvrit en sortie de bloc lexical ;
- est passée en argument aux visiteurs
  `public Object visit(..., ILexicalEnvironment lexenv)`.

Au contraire, l'environnement global est stocké dans un champ `globalVariableEnvironment` de `Interpreter` ; il est enrichi par des affectations mais jamais appauvri.

Rappel : environnement lexical et portée (1/2)

Exemple d'enrichissement : variable locale.

Interpreter.java

```
public Object visit(IASTblock iast, ILexicalEnvironment lexenv)
throws EvaluationException
{
    ILexicalEnvironment lexenv2 = lexenv;
    for ( IASTbinding binding : iast.getBindings() )
    {
        Object initialisation = binding.getInitialisation().accept(this, lexenv);
        lexenv2 = lexenv2.extend(binding.getVariable(), initialisation);
    }
    return iast.getBody().accept(this, lexenv2);
}
```

Rappel : environnement lexical et portée (2/2)

Où trouver la variable en cas de déclarations multiples ?

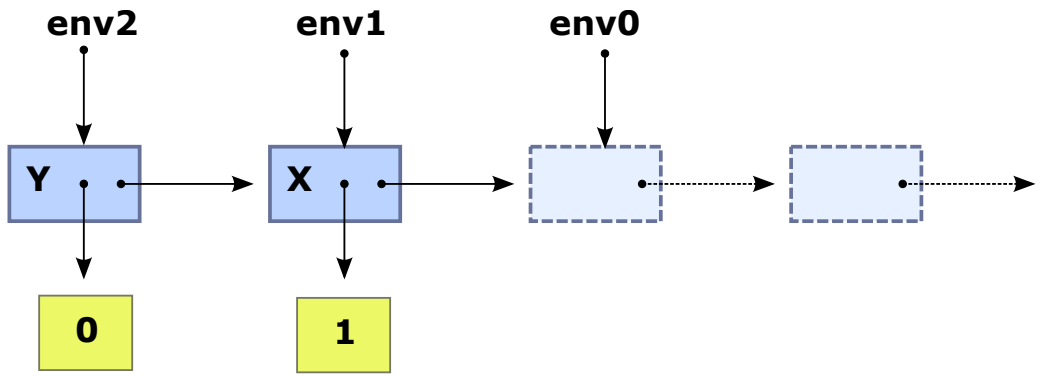
- une déclaration locale a priorité sur une déclaration globale ;
- la déclaration locale la plus récente a priorité
  - l'environnement lexical est une liste ordonnée
  - `extend` ajoute en tête de liste,
  - `getValue` et `update` s'arrêtent à la première variable trouvée qui a le bon nom dans l'environnement

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

point d'évaluation	**environnement**	**eval(x)**
`(let x = 1 in (let x = 2 in x + 1) + x)`	[ ]	—
`let x = 1 in ((let x = 2 in x + 1) + x)`	[(x,1)]	1
`let x = 1 in (let x = 2 in (x + 1)) + x`	[(x,2), (x,1)]	2
`let x = 1 in (let x = 2 in x + 1) + (x)`	[(x,1)]	1
`let x = 1 in (let x = 2 in x + 1) + x ()`	[ ]	—

Note : environnement lexical et partage

```
env1 = env0.extend(new ASTvariable("X"), new Integer(0));
env2 = env1.extent(new ASTvariable("Y"), new Integer(0));
env2.update(new ASTvariable("X"), new Integer(1));
```



- `env1.extend` ne modifie pas son argument `env1`,
  elle retourne une nouvelle liste `env2` ;
- mais les queues de `env2` et de `env1` sont **partagées en mémoire** ;
- une modification de `X` par `env2.update`
  est aussi visible par `env1.getValue` !

$\Longrightarrow$ important pour assurer que l'affectation ILP fonctionne bien !

Rappel : interface des fonctions

`ASTlambda` retourne une fonction...
$\Longrightarrow$ il faut donc créer une valeur `ILP` pour les fonctions.

(fonction de première classe $\Longrightarrow$ les fonctions sont manipulables comme des valeurs)

Invocable.java

```
package com.paracamplus.ilp1.interpreter.interfaces;
public interface Invocable
{
    int getArity();
    Object apply(Interpreter interpreter, Object[] argument) throws EvaluationException;
}
```

L'interface dit peu : une fonction a une arité et peut être appelée.

Toute l'intelligence sera dans l'implantation `Function` de l'interface !

- état interne de la valeur fonction,
  nécessaire pour retrouver nos variables lors de l'évaluation ;
- constructeur : construction de l'état interne ;
- implantation de `apply` (transparents suivants).

Note : l'interface des fonctions est en fait `IFunction`, qui hérite de `Invocable` sans l'enrichir.

Fonction close (1/2)

Function.java (début)

```
package com.paracamplus.ilp1.interpreter;
public class Function implements IFunction
{
    private final IASTvariable[] variables;
    private final IASTexpression body;
    private final ILexicalEnvironment lexenv;

    public Function (IASTvariable[] variables,
                     IASTexpression body,
                     ILexicalEnvironment lexenv)
    {
        this.variables = variables;
        this.body = body; this.lexenv = lexenv;
    }
```

Implantation `Function` : fonction close, i.e., sans variable non-définie.

Le constructeur capture un état complexe :

- état statique : arguments formels `variables` et corps `body` ;
  disponible dans l'AST
- état dynamique : environnement lexical `lexenv` lors de l'interprétation.
  disponible seulement lors de l'interprétation, lors de l'évaluation de `function` ou `lambda`

Fonction close (2/2)

Function.java (fin)

```
    public Object apply(Interpreter interpreter, Object[] argument)
    throws EvaluationException
    {
        if (argument.length != getArity()) throw new EvaluationException("Wrong arity");
        ILexicalEnvironment lexenv2 = lexenv;
        IASTvariable[] variables = variables;
        for ( int i=0 ; i<argument.length ; i++ )
            lexenv2 = lexenv2.extend(variables[i], argument[i]);
        return getBody().accept(interpreter, lexenv2);
    }
}
```

Pour l'évaluation de l'appel, `apply` :

- nous repartons de l'environnement lors de la création de la fonction maintenu dans l'attribut `lexenv` ;
  l'environnement au moment de l'application de la fonction importe peu
- nous y ajoutons les associations : argument formel $\rightarrow$ argument réel `variables[i]` $\rightarrow$ `argument[i]` ;
- et évaluons le corps de la fonction.

Interprétation du lambda

Interpreter.java

```
public Object visit(IASTlambda iast, ILexicalEnvironment lexenv)
throws EvaluationException
{
    IFunction fun = new Function(iast.getVariables(), iast.getBody(), lexenv);
    return fun;
}
```

Création de la valeur fonction avec l'environnement courant lors de l'évaluation du lambda.

$\Longrightarrow$ implante bien la portée lexicale.

Aucun changement dans la manière dont les fonctions sont invoquées.
(cf. ILP1)

Exemple de lambda

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

point d'évaluation	**environnement**
`((let x = 1 in lambda (y) (x + y)) (2 + 3))`	[ ]
`(let x = 1 in lambda (y) (x + y)) (2 + 3)`	[ ]
`(let x = 1 in (lambda (y) (x + y))) (2 + 3)`	[(x,1)]
`(let x = 1 in lambda (y) (x + y)) (2 + 3)`	[ ]
Application de (`lambda (y) (x + y)`,(x,1)) à 5	
`(lambda (y) (x + y))(5)`	[(x,1)]
`(x + y)`	[(y,5), (x,1)]
`6`	[ ]

L'environnement capturé par `lambda (y) (x + y)` est [(x,1)].

Interprétation des codéfinitions

visiteur de IASTcodefinitions

```
ILexicalEnvironment lexenv2 = lexenv;

for ( IASTnamedLambda fun : iast.getFunctions() )
{
    IASTvariable variable = fun.getFunctionVariable();
    lexenv2 = lexenv2.extend(variable, null);
}

for ( IASTnamedLambda fun : iast.getFunctions() )
{
    Object f = this.visit(fun, lexenv2);
    IASTvariable variable = fun.getFunctionVariable();
    lexenv2.update(variable, f);
}

Object result = iast.getBody().body.accept(this, lexenv2);
```

Le lambda défini est évalué dans l'environnement lexenv2
où les variables déclarées existent déjà (mais sont affectées à `null`).

$\Longrightarrow$ la codéfinition permet les fonctions (mutuellement) récursives.

Efficacité des environnements

LexicalEnvironment.java

```
public Object getValue(IASTvariable key)
{
    if ( key.getName().equals(getKey().getName()) ) return getValue();
    return getNext().getValue(key); /* appel récursif */
}
```

`LexicalEnvironment` est une liste chaînée

- coût linéaire à chaque lecture ou modification de variable !
  $\Longrightarrow$ très peu efficace

Coût des lambdas :

- construction en temps constant : copie du pointeur `lexenv`
- mais on garde des références sur des variables inutiles
  $\Longrightarrow$ coût reporté sur le ramasse-miettes !

Peut-on faire mieux ?
Oui, avec les clôtures, utilisées dans le compilateur.

Compilation : code C généré et bibliothèque d'exécution C

Rappel : les variables ILP dans le C généré

```
let x = 2 in
  let y = 3 in
    x + y
```

$\Longrightarrow$

```
{
  ILP_Object ilptmp1 = ILP_Integer2ILP(2);
  {
    ILP_Object x = ilptmp1;
    {
      ILP_Object ilptmp2 = ILP_Integer2ILP(3);
      {
          ILP_Object y = ilptmp2;
          // utilisation de x et y...
      } // fin de portée de y
    }
  } // fin de portée de x
}
```

- variable locale ILP $\longrightarrow$ variable locale C ;
- portée de la variable C = portée du bloc ILP.

Les fonctions locales ILP dans le C généré (principe)

```
(let x = 2 in
 lambda (y)
        (x + y)
) 10
```

$\Longrightarrow$

```
ILP_Object anon(/*omis*/ ILP_Object y) {
    ILP_Object x = ??? // valeur non disponible !
    return ILP_Plus(x, y);
}

ILP_Object ilp_program() {
    ILP_Object ilptmp1;
    ILP_Object ilptmp2 = ILP_Integer2ILP(10);
    {
        ILP_Object x = ILP_Integer2ILP(2);
        ilptmp1 = ??? // fonction anon ?
    }
    return ILP_invoke(ilptmp1, 1, ilptmp2);
}
```

- une fonction ILP est aussi traduite en une fonction C ;
- pas de fonction locale en C $\Longrightarrow$ `anon` est globale ;
- `anon` est donc déclarée hors de la portée de `x` !
  $\Longrightarrow$ il faut un moyen à `ilp_program` pour communiquer `x` à `anon`.

Principe de compilation, clôture

L'interprète avait un accès direct à l'ensemble de l'environnement lexical et pouvait facilement en garder une référence.

Ce n'est pas le cas du compilateur !

Le compilateur doit donc générer du code C pour :

- à la définition d'une fonction :
  stocker dans une structure une copie de l'environnement ;
- lors d'un appel : transmettre cette structure à la fonction ;
- dans la fonction locale : extraire les variables de la structure.

Cette structure supplémentaire s'appelle une **clôture**.

Optimisation :

Nous ne stockons dans la clôture que ce qui est réellement nécessaire.

Les variables libres, comme expliqué plus loin.

Bibliothèque d'exécution : les clôtures (1/2)

ilp.h (début)

```
typedef struct ILP_Object {
    struct ILP_Class*  _class;
    union {
        ...
        struct asClosure {
            ILP_general_function function;
            short                arity;
            struct ILP_Object*   closed_variables[1];
        } asClosure;
        ...
    } _content;
} *ILP_Object;
```

Une clôture contient :

- un pointeur sur une fonction C : `function` ;
- `arity` : le nombre d'arguments ;
- l'environnement : un tableau de valeurs `closed_variables`.

Exemple :

`lambda (y) (x + y)` a un argument (`y`) et une valeur d'environnement (`x`).

Bibliothèque d'exécution : les clôtures (2/2)

ilp.h (suite)

```
ILP_Object ILP_make_closure(ILP_general_function f, int arity, int argc, ...);
ILP_Object ILP_invoke(ILP_Object f, int arity, ...);
```

`ILP_make_closure` :

- alloue la structure et la remplie avec l'environnement
- `f` pointe sur la fonction C à appeler
- `arity` est le nombre d'arguments de `f`
- `argc` est la taille de l'environnement
- `...` est une liste de `argc` valeurs (l'environnement)

`ILP_invoke` :

- vérifie que `f` est une clôture
- appelle la fonction pointée par `f`
  avec la clôture `f` en premier argument, suivie des `arity` arguments.

Par soucis d'uniformité et pour éviter les cas particuliers dans la génération de code, toutes les fonctions C générées par le compilateur prennent une clôture en premier argument, même si c'est inutile (cas des fonctions globales).

Environnement lexical et partage (version compilateur)

La clôture stocke une **copie** d'une variable locale.
Que se passe-t-il si la copie est modifiée ?

Exemple : `let x = 1 in (function f () (x = x + 1) in f(); x)`

Le résultat attendu est 2 mais, avec une copie simple :
- `f` modifierait la copie `x` ;
- le programme retournerait la valeur du `x` original : 1.

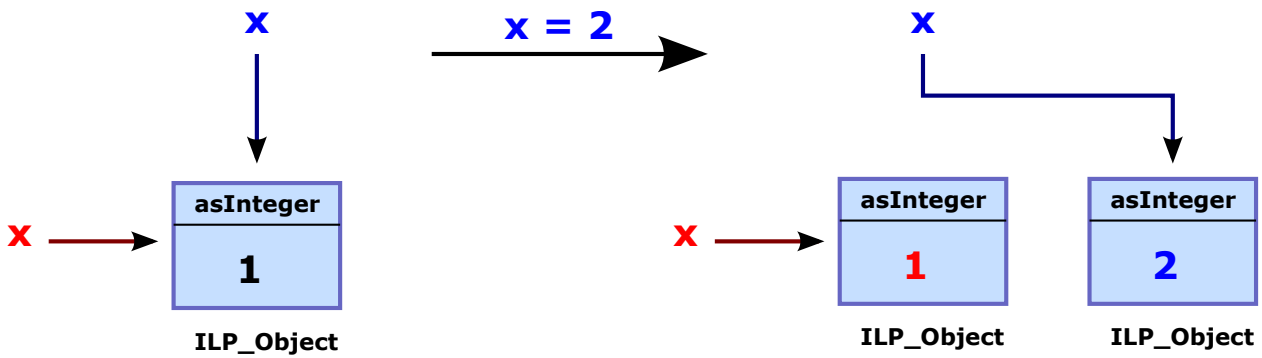
Solution : les **boîtes**, qui ajoutent une indirection.

- la valeur n'est plus stockée directement dans la variable ;
- la variable contient un pointeur vers une boîte ;
- la boîte contient la valeur ;
- c'est le pointeur sur la boîte qui est copiée dans les clôtures, pas le contenu de la boîte ;
- toutes les copies accèdent à la même boîte
  donc manipulent la même valeur en mémoire !

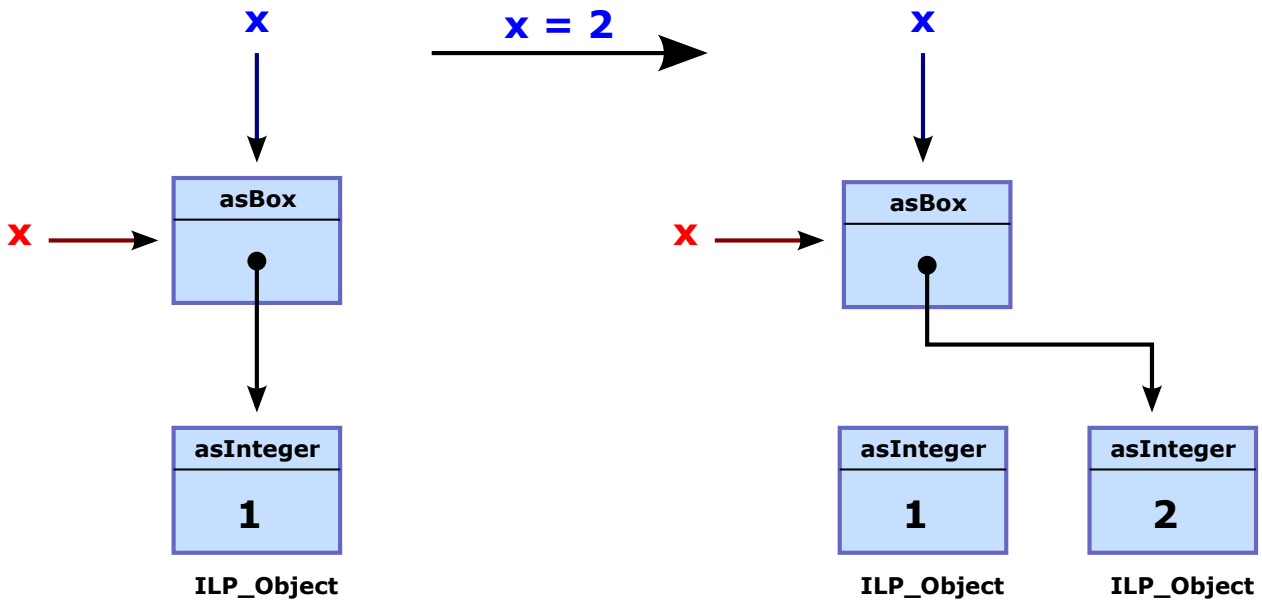
Inconvénient :
niveau d'indirection supplémentaire, donc perte d'efficacité en temps et en mémoire.

Partage et boîtes : illustration

Sans boîte



Avec boîte



Note : la solution employée en Java

classe interne dans une méthode

```
void m()
{
    final int x = 12;

    class C implements Comparator<String> {
        public int compare(String a, String b) {
            if (x > 19) // ...
        }
    }
}
```

Toute variable utilisée dans une classe interne doit être déclarée final
$\Longrightarrow$ après initialisation, sa valeur est constante.

Ainsi, même si `x` est dupliquée lors de la création de l'instance de la classe interne, on a l'assurance que les deux valeurs correspondent toujours !

Mais que se passe-t-il si `x` est un objet ayant un champ `int` ?

Bibliothèque d'exécution : les boîtes

ilp.h

```
typedef struct ILP_Object
{
    struct ILP_Class*  _class;
    union {
        ...
        struct asBox {
            struct ILP_Object* value;
        } asBox;
        ...
    } _content;
} *ILP_Object;

#define ILP_Box2Value(box)          (((ILP_Box)(box))->_content.asBox.value)
#define ILP_Value2Box(o)            ILP_make_box(o)
#define ILP_SetBoxedValue(box, o)   (((ILP_Box)(box))->_content.asBox.value = (o))
```

Optimisation :
ne sera mis dans une boîte que ce qui peut être copié dans une clôture.

Exemple complet : fonction anonyme (1/2)

Exemple : `(let x = 2 in lambda (y) (x + y)) (10)`

Génération de la fonction `lambda (y) (x + y)`,
prenant en argument une clôture contenant `x` (dans une boîte).

code C généré (début)

```
ILP_Object ilpclosure3(ILP_Closure ilp_closure, ILP_Object y2)
{
    ILP_Object x1 = ilp_closure->_content.asClosure.closed_variables[0];
    {
        ILP_Object ilptmp535;
        ILP_Object ilptmp536;
        ilptmp535 = ILP_Box2Value(x1);
        ilptmp536 = y2;
        return ILP_Plus(ilptmp535, ilptmp536);
    }
}
```

Exemple complet : fonction anonyme (2/2)

Exemple : `(let x = 2 in lambda (y) (x + y)) (10)`

Génération de la clôture de `lambda (y) (x + y)` dans la portée de `x`, et appel de fonction avec clôture en dehors de la portée de `x`.

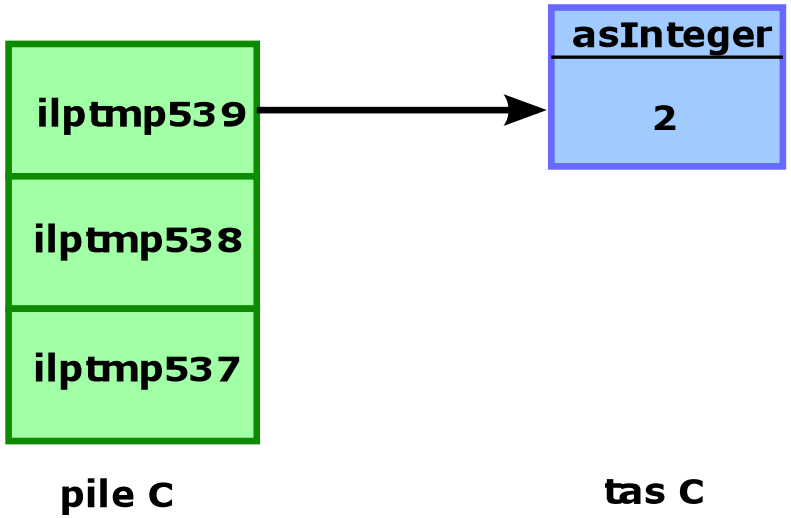
code C généré (suite)

```
ILP_Object ilp_program()
{
    {
        ILP_Object ilptmp537;
        ILP_Object ilptmp538;
        {
            ILP_Object ilptmp539;
            ilptmp539 = ILP_Integer2ILP(2);
            {
                ILP_Object x1 = ILP_Value2Box(ilptmp539);
                ilptmp537 = ILP_make_closure(ilpclosure3, 1, 1, x1);
            }
        }
        ilptmp538 = ILP_Integer2ILP(10);
        return ILP_invoke(ilptmp537, 1, ilptmp538);
    }
}
```

Exemple complet : illustration (1/9)

```
(let x = 2 in lambda (y) (x + y)) (10)
```

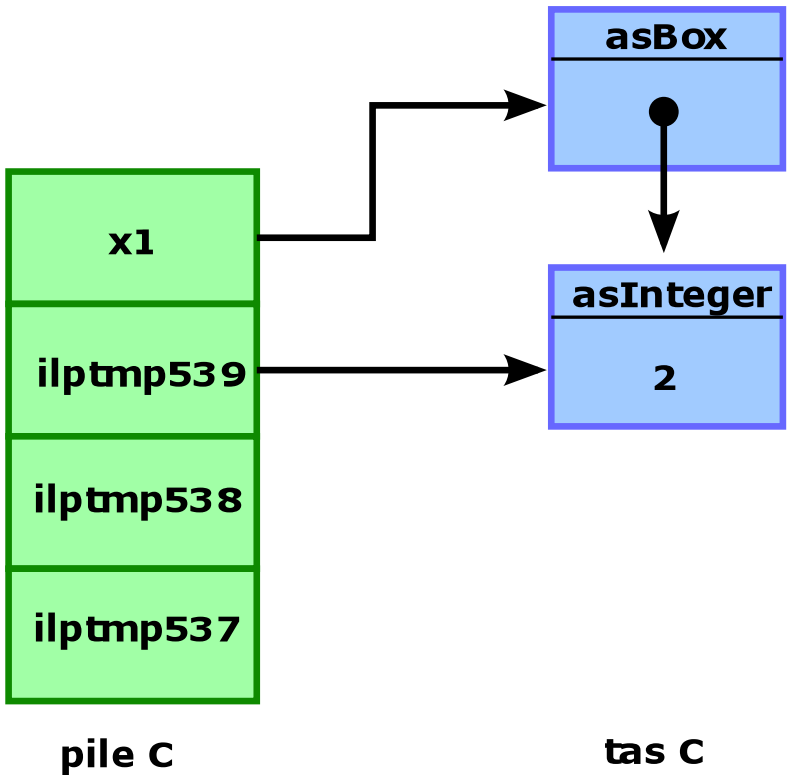
ilp_program : ilptmp539 = ILP_Integer2ILP(2)



Exemple complet : illustration (2/9)

```
(let x = 2 in lambda (y) (x + y)) (10)
```

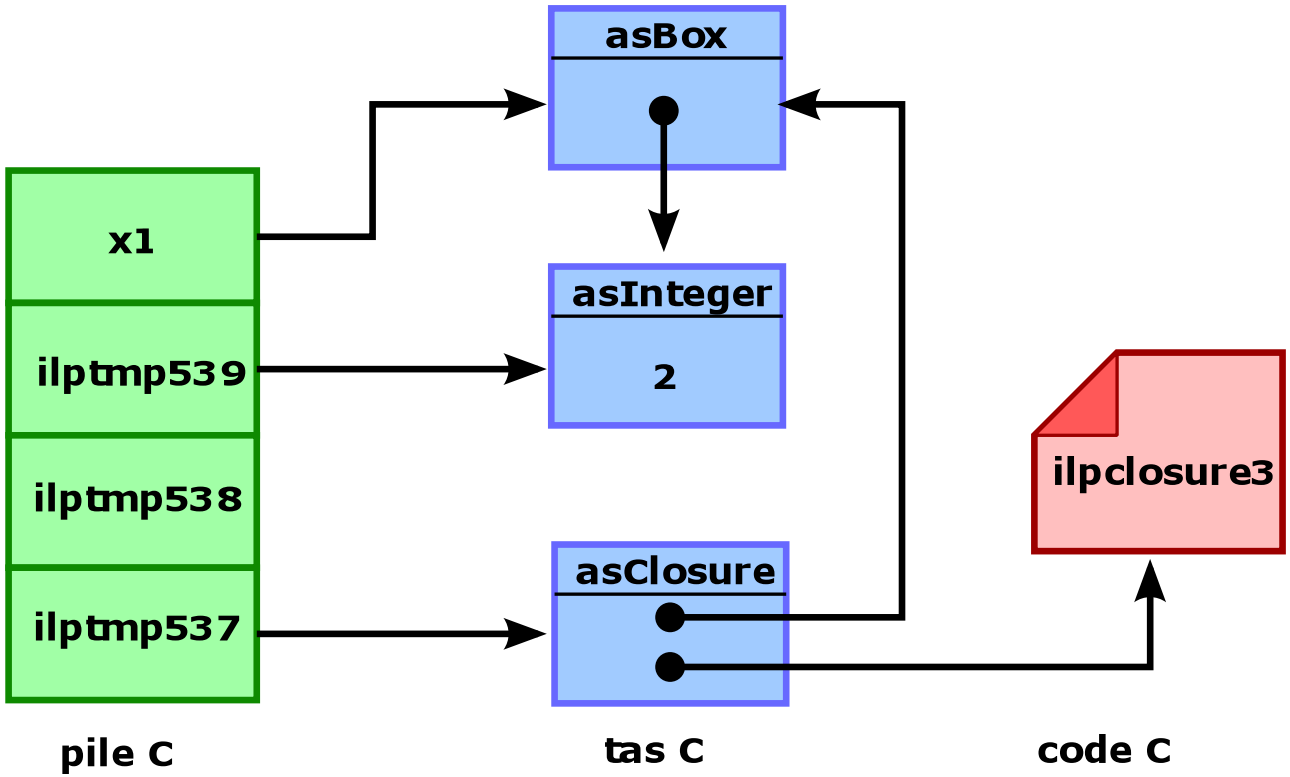
ilp_program : x1 = ILP_Value2Box(ilptmp539)



Exemple complet : illustration (3/9)

```
(let x = 2 in lambda (y) (x + y)) (10)
```

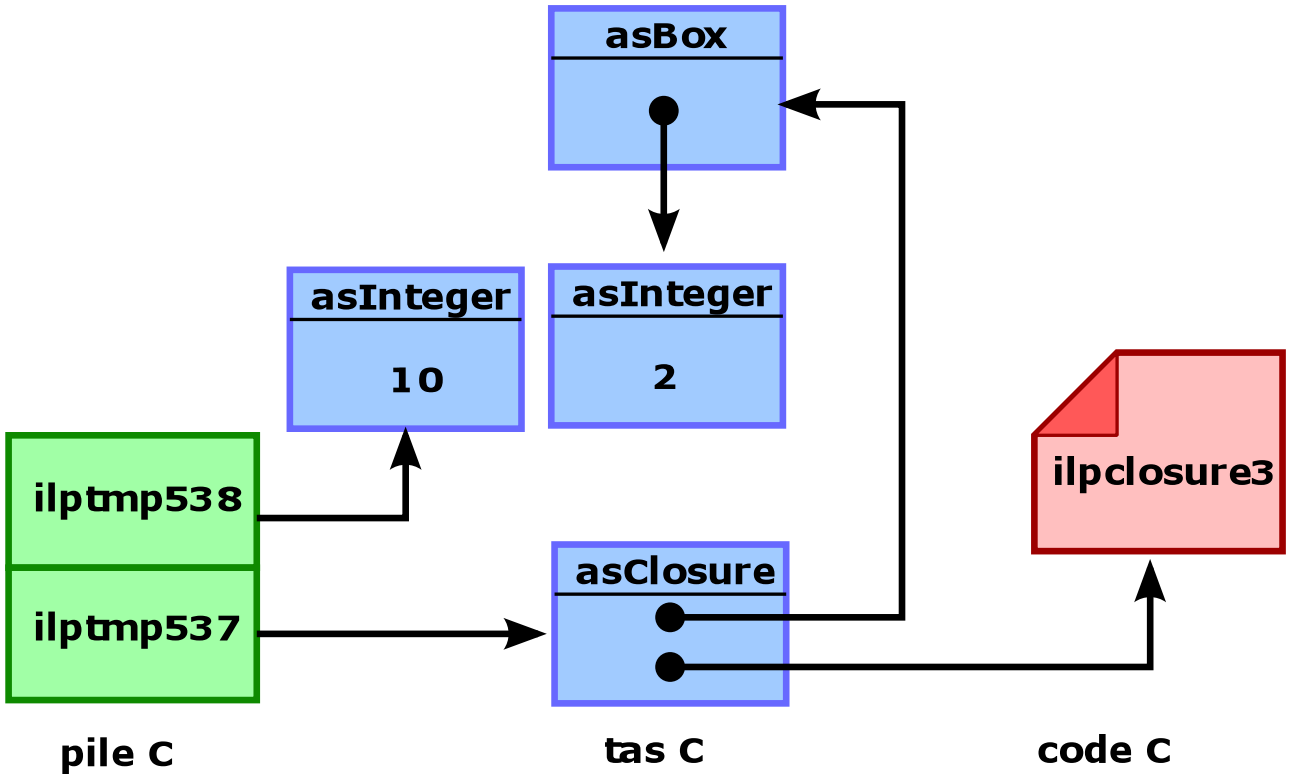
ilp_program : ilptmp537 = ILP_make_closure(ilpclosure3,1,1,x1)



Exemple complet : illustration (4/9)

```
(let x = 2 in lambda (y) (x + y)) (10)
```

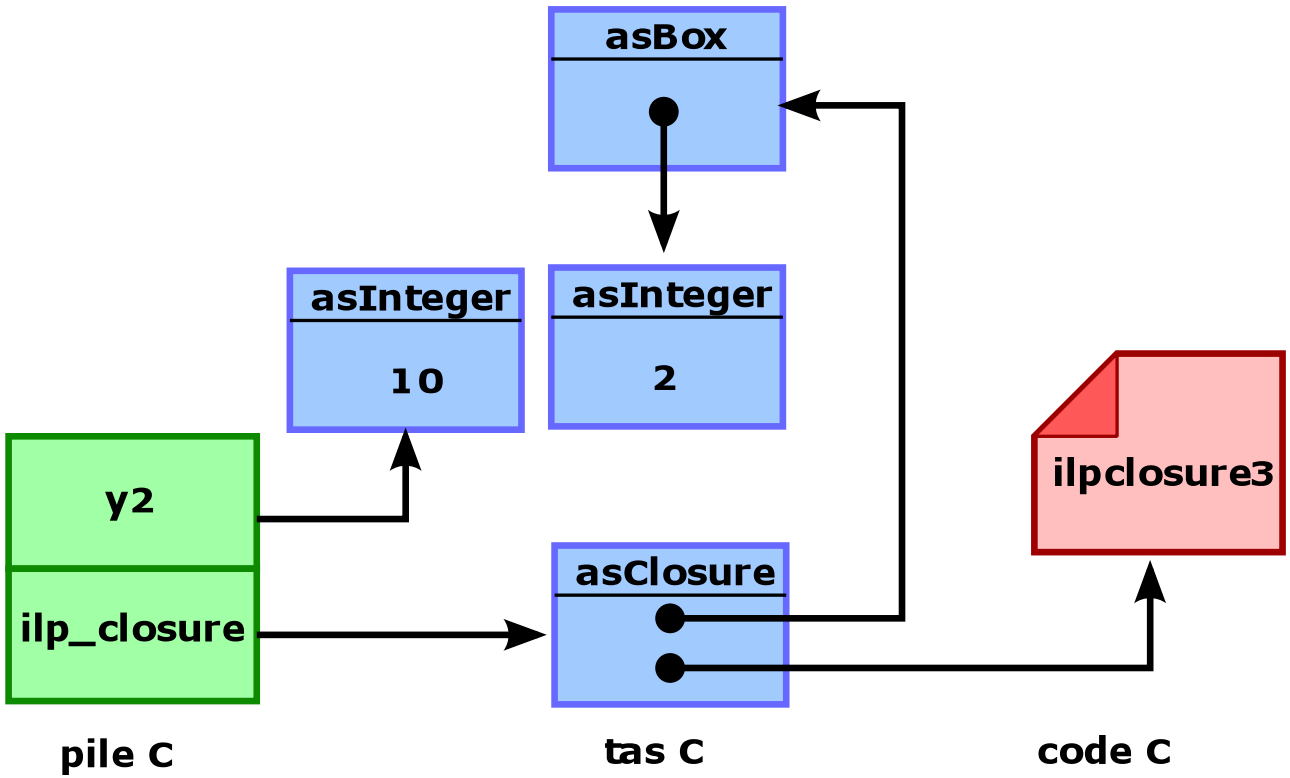
ilp_program : ilptmp538 = ILP_Integer2ILP(10)



Exemple complet : illustration (5/9)

```
(let x = 2 in lambda (y) (x + y)) (10)
```

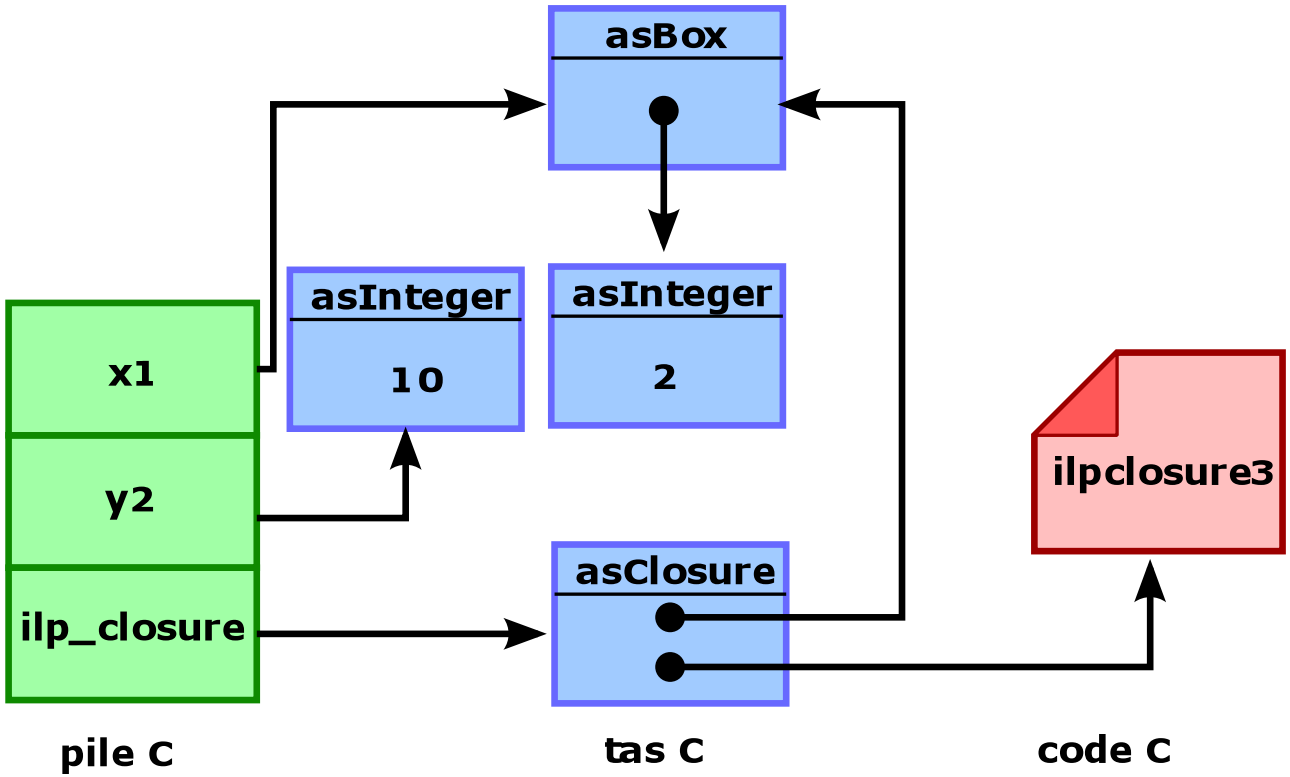
ilpclosure3(ilptmp537,ilptmp538)



Exemple complet : illustration (6/9)

```
(let x = 2 in lambda (y) (x + y)) (10)
```

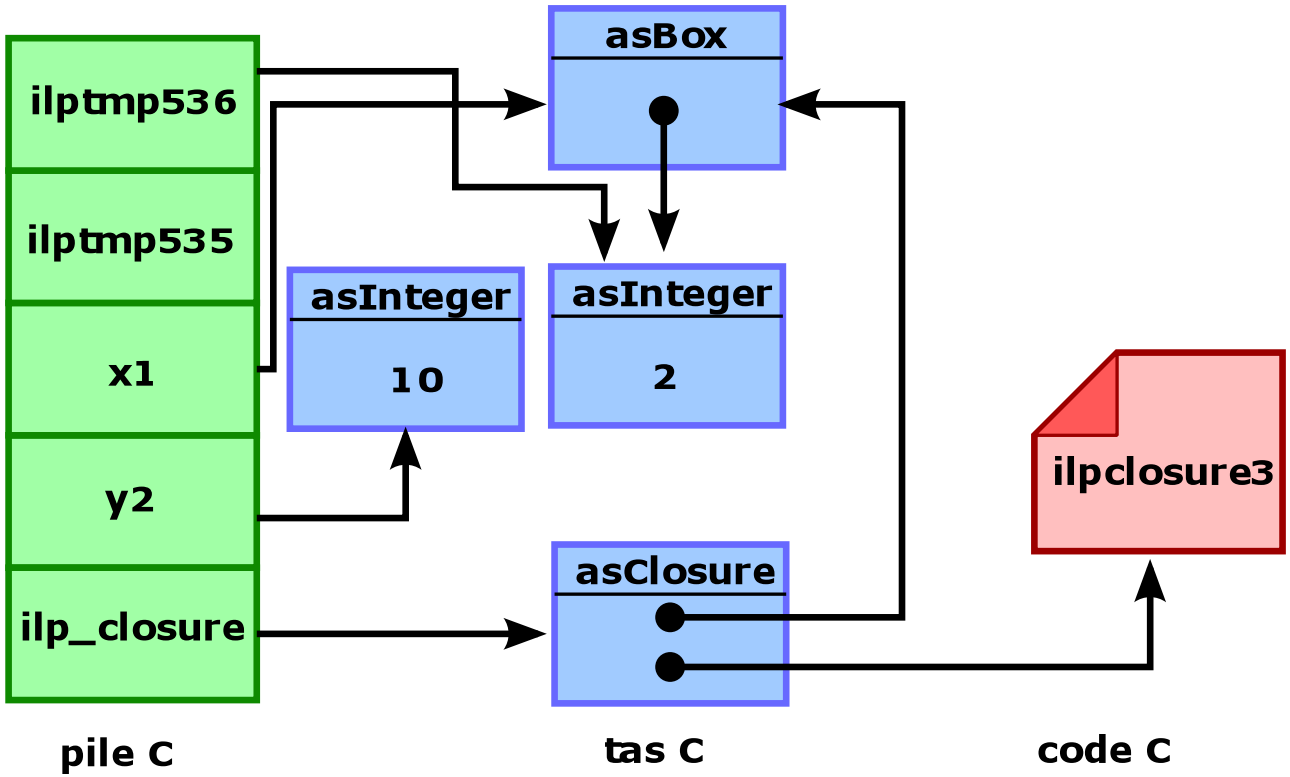
ilpclosure3 : x1 =ilp_closure->_content.asClosure.closed_variables[0]



Exemple complet : illustration (7/9)

```
(let x = 2 in lambda (y) (x + y)) (10)
```

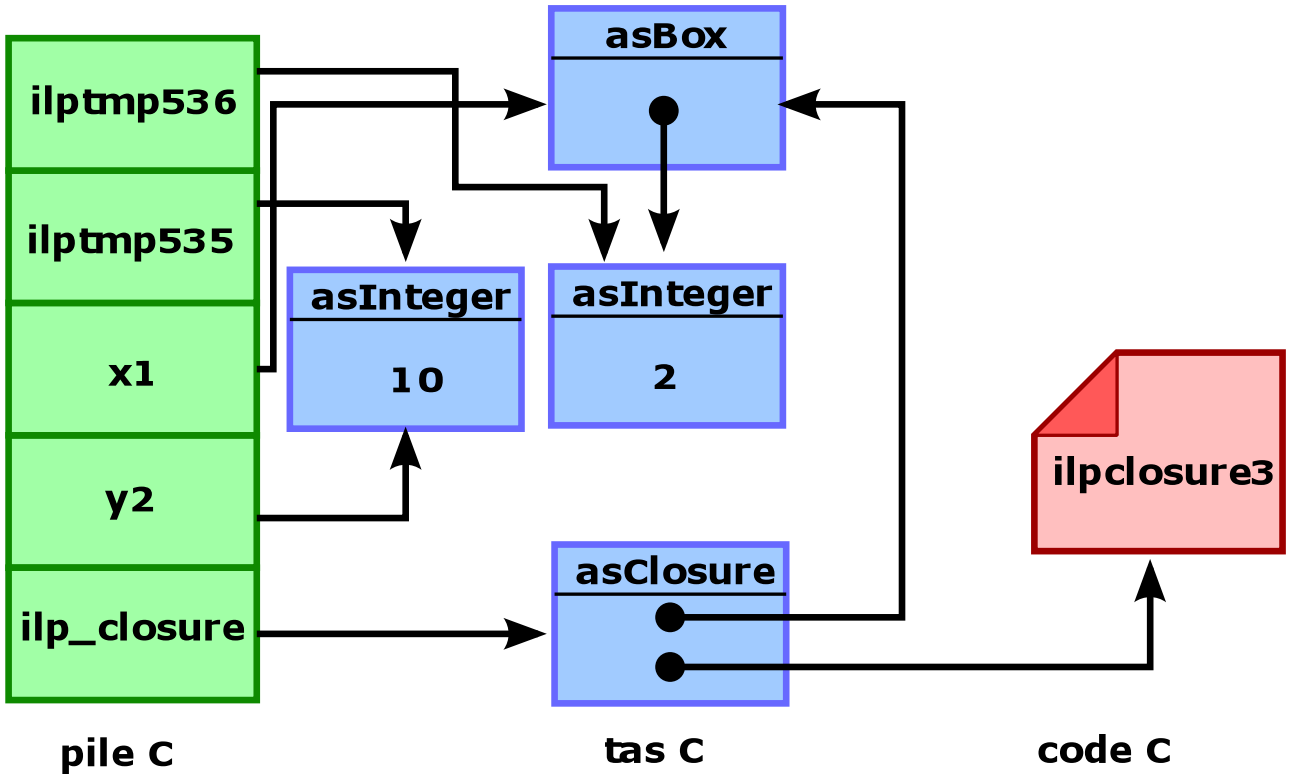
ilpclosure3 : ilptmp536 = ILP_box2Value(x1)



Exemple complet : illustration (8/9)

```
(let x = 2 in lambda (y) (x + y)) (10)
```

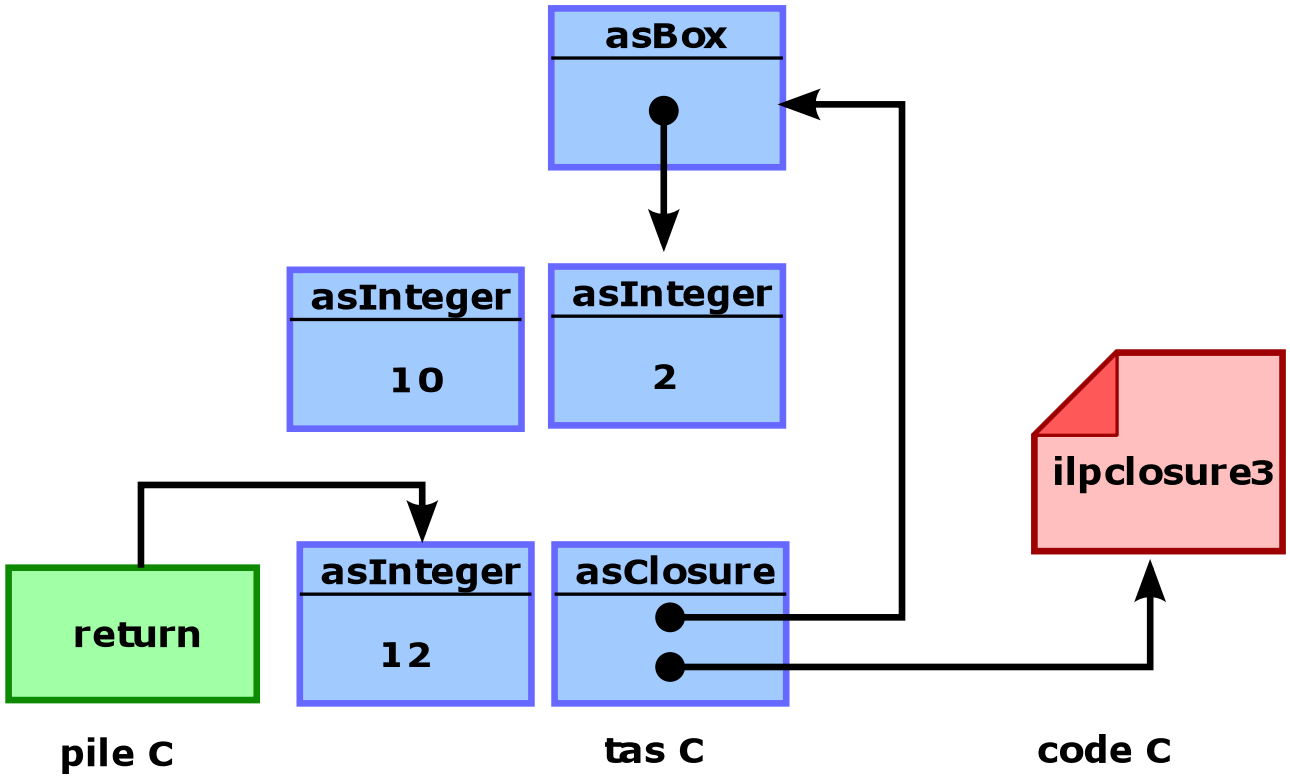
ilpclosure3 : ilptmp535 = y2



Exemple complet : illustration (9/9)

```
(let x = 2 in lambda (y) (x + y)) (10)
```

ilpclosure3 : return ILP_Plus(ilptmp535, ilptmp536)



Exemple de code généré pour les codéfinitions

Exemple : `function f (x) (f(x - 1)) in f(12)`

code C généré (simplifié)

```
ILP_Object f_1(ILP_Closure ilp_closure, ILP_Object x)
{
    ILP_Object f_2 = ilp_closure->_content.asClosure.closed_variables[0];
    ILP_Object ilptmp1 = ILP_Minus(x, ILP_Integer2ILP(1));
    ILP_Object ilptmp2 = ILP_Box2Value(f_2);
    return ILP_invoke(ilptmp2, 1, ilptmp2);
}

ILP_Object ilp_program()
{
    ILP_Object f_2 = ILP_Value2Box(NULL);
    ILP_SetBoxedValue(f_2, ILP_make_closure(f_1, 1, 1, f_2));
    ILP_Object ilptmp3 = ILP_Box2Value(f_2);
    return ILP_invoke(ilptmp3, 1, ILP_Integer2ILP(12));
}
```

`f` est dans l'environnement de `f`

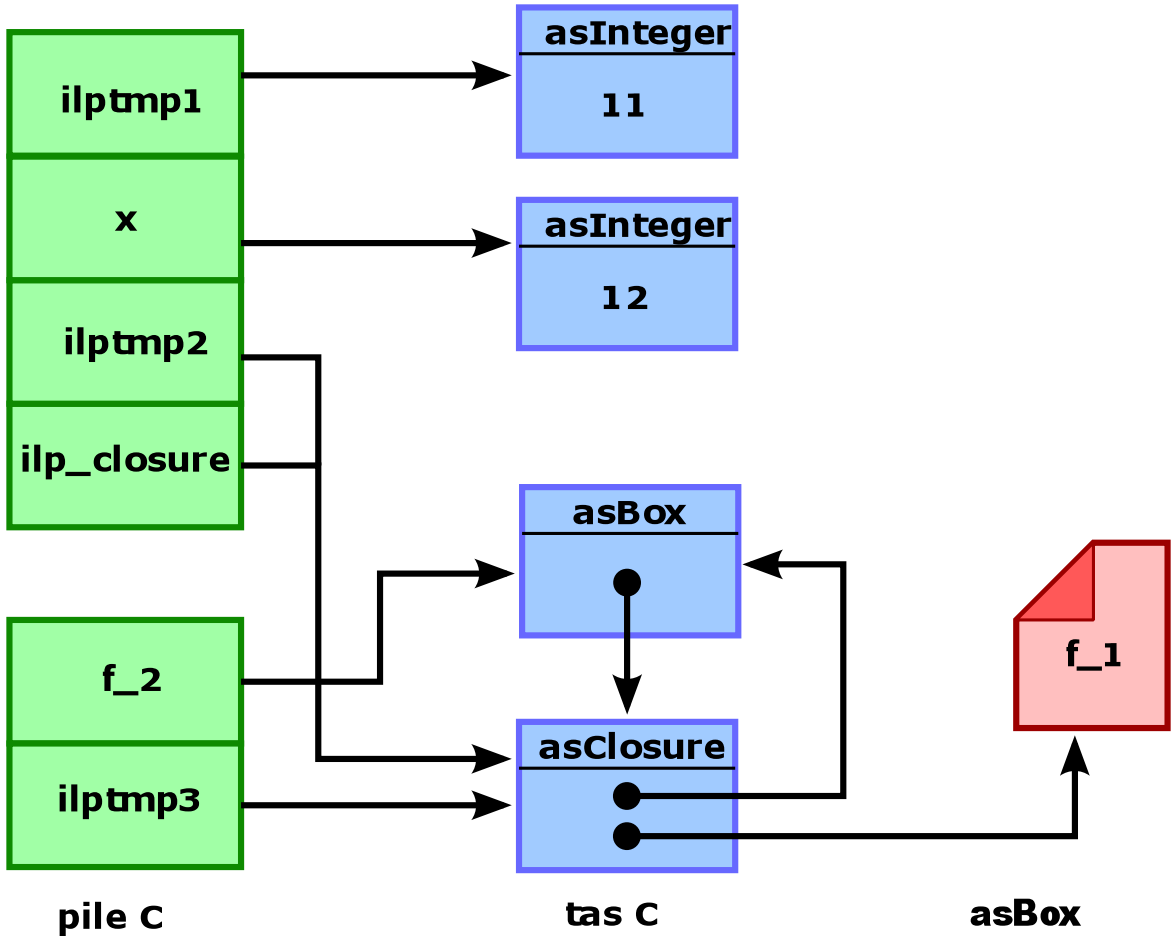
nécessaire car `f` est récursive

$\Longrightarrow$ la clôture `f_2` de `f_1` pointe donc sur elle-même !

nécessite une création en deux temps

allocation avec `ILP_Value2Box(NULL)`, puis remplissage avec `ILP_SetBoxedValue`

Code généré pour les codéfinitions : illustation




```
function f (x) (f(x - 1)) in f(12)
```

Compilation

Étapes spécifiques à la compilation

Quoi de neuf ?

Des nouveaux nœuds `AST` à traiter, mais aussi :

- `Normalizer`
  classification des variables (locales, globales)
  classification des types d'appels de fonctions (optimisation)
- `GlobalVariableCollector`
  rien de neuf
- `FreeVariableCollector`
  extraction des fonctions locales
  calcul de leurs variables libres
- `Compiler`
  assez simple quand on a compris le reste.

Rappel : classification des variables

`Normalizer` se charge aussi de classer les variables `IASTvariable` :

- `IASTCglobalFunctionVariable`
  référence à une fonction globale déclarée dans le programme
- `IASTClocalVariable`
  référence à une variable introduite par un bloc
  ou un argument formel de fonction
- `IASTCglobalVariable`
  toute autre variable non locale
- `IASTClocalFunctionVariable`
  référence à une fonction introduite par une codéfinition
  (ajout d'ILP3).

Par ailleurs, chaque variable du programme correspond
à un unique objet `IASTCvariable`.

si `a != b`, alors `a` et `b` représentent des variables différentes,
même si `a.getName()` et `b.getName()` sont égaux

Classification des appels

Normalizer.java

```
public IASTexpression visit(IASTinvocation iast, INormalizationEnvironment env)
throws CompilationException
{
    IASTexpression funexpr = iast.getFunction().accept(this, env);
    IASTexpression[] args = /* omis */;
    if ( funexpr instanceof IASTCglobalVariable )
    {
        IASTCglobalVariable f = (IASTCglobalVariable) funexpr;
        return ((INormalizationFactory)factory).newGlobalInvocation(f, args);
    }
    else if ( funexpr instanceof IASTClocalFunctionVariable )
    {
        IASTClocalFunctionVariable f = (ASTClocalFunctionVariable) funexpr;
        return ((INormalizationFactory)factory).newLocalFunctionInvocation(f, args);
    }
    else
    {
        return ((INormalizationFactory)factory).newComputedInvocation(funexpr, args);
    }
}
```

1. fonction globale ou primitive ;
2. fonction locale, introduite par `IASTcodefinition` ;
3. autre : fonction stockée dans une variable, retournée par un appel.

Analyse des variables libres : principe

Exemple : pour appliquer la fonction `lambda (y) (x + y)` il suffit de connaître la valeur de `x` : `x` est libre.

Une variable locale V est **libre** dans une expression e si :

- V est utilisée dans e
- et V n'est pas définie dans e.

Une variable qui apparaît dans e mais est non libre est dite **liée**.

Calcul des variables libres FV :

$FV(e)$ par récurrence sur la syntaxe de e

- variable : $FV(\texttt{V}) = \{V\}$
- affectation : $FV(\texttt{V = e}) = \{V\} \cup FV(e)$
- opération : $FV(\texttt{e + f}) = FV(e) \cup FV(f)$
- fonction : $FV(\texttt{lambda (x) (e)}) = FV(e) \backslash \{x\}$
- bloc : $FV(\texttt{let x = e in f}) = FV(e) \cup (FV(f) \backslash \{x\})$

Point essentiel : un lieur transforme une variable libre en variable liée.

Exemple : dans `lambda (y) (x + y)`, `x` est libre, `y` est lié.

Analyse des variables libres : implantation (1/4)

FreeVariableCollector.java (début)

```
public class FreeVariableCollector
implements IASTCvisitor<Void, Set<IASTClocalVariable>, CompilationException>
{
    protected final IASTCprogram program;

    public FreeVariableCollector(IASTCprogram program)
    {
        this.program = program;
    }

    public IASTCprogram analyze () throws CompilationException
    {
        Set<IASTClocalVariable> newvars = new HashSet<>();
        program.getBody().accept(this, newvars);
        return program;
    }
```

Un visiteur, défini dans ILP1, étendu dans ILP2–ILP4.

Prend en argument un ensemble de variables locales à enrichir : `Set<IASTClocalVariable>`.
Pas de valeur de retour : le résultat est directement stocké dans l'ASTC.

Analyse des variables libres : implantation (2/4)

FreeVariableCollector.java (suite)

```
public Void visit(IASTClocalVariable iast, Set<IASTClocalVariable> variables)
throws CompilationException
{
    variables.add(iast);
    return null;
}

public Void visit(IASTbinaryOperation iast, Set<IASTClocalVariable> variables)
throws CompilationException
{
    iast.getLeftOperand().accept(this, variables);
    iast.getRightOperand().accept(this, variables);
    return null;
}
```

- exemple d'ajout de variable : `IASTClocalVariable` ;
- exemple de parcours récursif : `IASTbinaryOperation`.

L'ensemble `variables` passé en argument est modifié.

Analyse des variables libres : implantation (3/4)

FreeVariableCollector.java (suite)

```
public Void visit(IASTCblock iast, Set<IASTClocalVariable> variables)
throws CompilationException
{
    Set<IASTClocalVariable> currentVars = new HashSet<>();
    for (IASTCblock.IASTCbinding binding : iast.getBindings())
    {
        binding.getInitialisation().accept(this, variables);
        currentVars.add(binding.getVariable());
    }
    Set<IASTClocalVariable> newvars = new HashSet<>();
    iast.getBody().accept(this, newvars);
    newvars.removeAll(currentVars);
    variables.addAll(newvars);
    return null;
}
```

Exemple de lieur : `let x = e in f`

- ajout des variables libres de `e` ;
- calcul des variables libres de `f` : `newvars` ;
- ajout de `newvars`, privé des variables liées : `currentVars` (e.g., `x`).

Analyse des variables libres : implantation (4/4)

FreeVariableCollector.java (suite)

```
public Void visit(IASTClambda iast, Set<IASTClocalVariable> variables)
throws CompilationException
{
    Set<IASTClocalVariable> newvars = new HashSet<>();
    iast.getBody().accept(this, newvars);
    IASTvariable[] vars = iast.getVariables();
    newvars.removeAll(Arrays.asList(vars));
    iast.setClosedVariables(newvars);
    ((IASTCprogram) program).addClosureDefinition(iast);
    for ( IASTvariable v : newvars)
    {
        ((IASTClocalVariable)v).setClosed();
    }
    variables.addAll(newvars);
    return null;
}
```

`IASTClambda` : similaire à un `IASTCblock` mais en plus :

- nous stockons la clôture avec `addClosureDefinition` ;
- ses variables libres sont fixées par `setClosedVariables` ;
- `setClosed` indique qu'une variable appartient à une clôture.

Génération des fonctions C (simplifié)

Compiler.java (extrait simplifié)

```
public Void visit(IASTCprogram iast, Context context) throws ...
{
    /*...*/
    for (IASTfunctionDefinition ifd : iast.getFunctionDefinitions()) {
        emitPrototype(ifd, c);
    }
    for (IASTClambda closure : iast.getClosureDefinitions()) {
        emitPrototype(closure, c);
    }
    for (IASTfunctionDefinition ifd : iast.getFunctionDefinitions())
    {
        visit(ifd, c);
        emitClosure(ifd, c);
    }
    for (IASTClambda closure : iast.getClosureDefinitions()) {
        emitFunction(closure, c);
    }
    /*...*/
}
```

- pour chaque fonction ILP globale : génère une fonction C ;
  génère aussi une clôture constante ;
  (utile si la fonction est copiée dans une variable ou passée en argument)
- pour chaque fonction ILP locale : génère une fonction C.

Génération des accès aux variables (simplifié)

Compiler.java (extrait simplifié)

```
public Void visit(IASTClocalVariable iast, Context context)
throws CompilationException
{
    emit(context.destination.compile());
    if ( iast.isClosed() ) {
        emit("ILP_Box2Value(");
        emit(iast.getMangledName());
        emit(")");
    }
    else {
        emit(iast.getMangledName());
    }
    emit("; \n");
    return null;
}
```

- si la variable apparaît dans une clôture, il faut regarder dans la boîte ;
- sinon, la valeur est stockée directement dans la variable C.

Idem pour l'affectation.

Génération des appels de fonction (simplifié)

Selon le type d'appel, nous distinguons deux cas :

- `Void visit(IASTCglobalInvocation iast, Context ctx)`

  L'appel peut être traduit directement par un appel de fonction C.

  Cas des fonctions globales et des primitives.
  e.g. : `print(12)`.

- `Void visitGeneralInvocation(IASTinvocation iast, Context ctx)`

  L'appel se fait par `ILP_invoke`, en passant une clôture en argument.

  Cas d'un appel de fonction locale : la clôture est une variable C.
  e.g. : `function f (x) (x + 1) in f(2)`

  Cas où la fonction est le résultat d'une expression.
  e.g. : `(lambda (x) (x + 1) ) (2)`
  e.g. : `let x = lambda (y) (y + 1) in x(2)`

Nous ne donnons pas le détail du code ; lire `Compiler.java` chez soi.

Génération des créations de clôtures (simplifié)

`ILP_make_closure` est généré pour :

- Les nœuds `ASTClambda`.

  La clôture est la valeur retournée par l'expression.

- Les nœuds `ASTCcodefinition`.

  Pour chaque fonction, une variable locale C du nom correspondant est déclarée, et initialisée avec la clôture.
  Puis le corps est généré.

- Chaque fonction globale.

  Pour chaque fonction, une variable global C du nom correspondant est déclarée, et initialisée avec la clôture au démarrage du programme.

Idem, nous ne donnons pas le détail du code ; lire `Compiler.java` chez soi.

Conclusion

Compilation des fonctions locales avec portée lexicale :

- création de clôtures,
  pour transporter des morceaux d'environnement ;
- analyse statique des variables libres,
  pour savoir quelle partie de l'environnement transporter ;
- emboîtage des variables,
  pour l'accès cohérent à travers des références multiples.

Clés :

- savoir quelles variables sont référencées à l'exécution ;
- s'assurer statiquement que ces variables sont toujours accessibles.