

Les fonctions de première classe

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

Antoine Miné

Année 2015–2016

Cours 7
27 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 des **fonctions de première classe**

- **ILP4** : ajout des classes et des objets

Fonctions de première classe

ILP2 ne supporte que des **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 :

- déclarer des **fonctions locales**
`let g(x) = x + 1 in f 2`
- **stocker** des fonctions dans des variables et les **utiliser**
`let x = f in x(2)`
- **passer des fonctions en argument** ou les **retourner**
`let apply2 f = f(2)`
- créer des **fonctions anonymes** : les **lambda expressions**
`twice (fun x -> x + 2)`

Ce cours : **les fonctions de première classe dans ILP3**

- Introduction aux fonctions locales, anonymes et de première classe
- 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

- re-revoir l'extension d'ILP
- notion d'environnement et de clôture
(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)
- voir les lambdas expressions déguisées (ou pas) en Java

Les fonctions de première classe

Aspects statiques en C

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

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

⇒ **résolution 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 (*x)(int)) {
    (*x)(12);
    // appel indirect à h
}

void h(int v) {
}

void g() {
    f(&h);
}
```

Quelle variable est accédée ?

Quelle fonction est appelée ?

- 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 fonctions permettent les **appels indirects**

⇒ **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 à toutes les variables locales de **F** ;

Règle d'accès aux variables : **portée lexicale**

Dans le texte du programme, trouver le bloc **begin** / **end** :

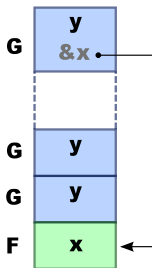
- déclarant **x**
- englobant l'accès (l'accès est dans la durée de vie de **x**)
- le plus petit (le plus proche de l'accès à **x**)

⇒ **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 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.

(par exemple, en cas d'appels récursifs)

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

Note historique : l'apport d'ALGOL 60

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

- portée **lexicale** : affiche 1
- portée **dynamique** : affiche 12

- portée **lexicale** : dépend du source du programme, **statique**
- portée **dynamique** : variable créé la plus récemment au moment de l'exécution **dynamique**

Les premiers langages utilisent la portée dynamique.

(plus facile à implanter : environnement = table d'association)

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

(comportement plus facile à prévoir, et plus robuste)

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** renvoie 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**)
- lors de l'utilisation de la fonction anonyme dans **map**
la valeur de **x** utilisée est l'argument passé à **add**
donc **2**

⇒ toujours la **règle de portée lexicale**

Note historique : le lambda calcul

Une fonction peut être une valeur d'un langage de programmation.
Il est même possible de concevoir un langage où **tout est fonction**.

Lambda-calcul : inventé par Alonzo Church (1932)

Syntaxe très simple : $e ::= X \mid e \ e \mid \lambda X. e$

- une expression peut être une variable X ;
- une expression peut être appliquée à une autre expression $e \ e$;
- une fonction peut être créée : $\lambda X. e$ (notation pour **fun** $X \rightarrow e$).

Une seule règle d'évaluation : $(\lambda X. e) \ f \rightarrow e[X/f]$,
substitution de X par f dans e , le calcul se fait par réécriture. . .

Outil mathématique pour l'étude des modèles de calcul et pour la logique.
Transformé en vrai langage, **LISP** (John McCarthy, 1958).
Inspiration pour les langages fonctionnels comme ML (Robin Milner, ~1970).

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 exécution.

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

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

- un pointeur vers un morceau de code
⇒ **quoi exécuter ?**
- un environnement
⇒ **où trouver la valeur des variables ?**

En Java : les fonctions comme objets

Java n'a pas des fonctions de première classe,
mais les fonctions 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 anonyme, fonction anonyme

classe anonyme

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

La notion de **classe anonyme** correspond bien celle de fonction anonyme.

En Java : portée lexicale

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) // ...  
        }  
    }  
}
```

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 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 lambdas

tri avec des lambda

```
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.

Implantation un peu **plus efficace** que par la création de classes anonymes.

Bibliothèque standard **enrichie** avec des aspects fonctionnels.

En ILP

Ajout de deux constructions syntaxiques :

- **codéfinitions** :

```
let rec even x = x = 0 || odd (x-1)
and odd x = x < 0 || even (x-1)
```

fonctions locales,
mutuellement récursives

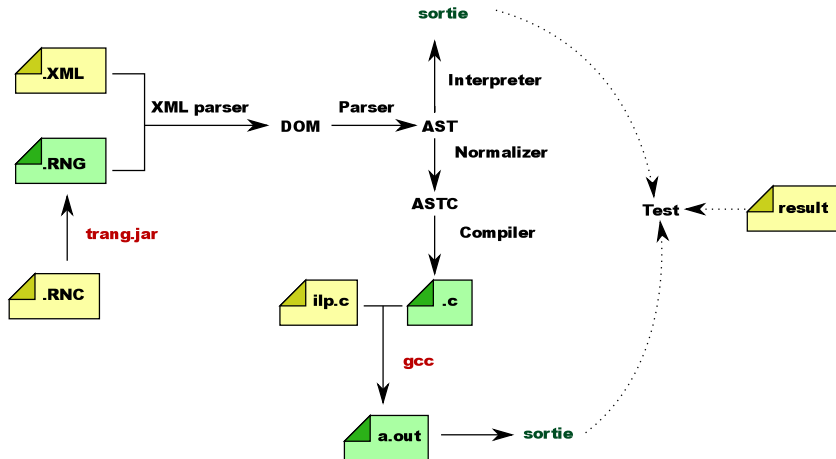
- **lambdas** :

```
fun 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.

Rappel : structure d'ILP



Syntaxe des fonctions

Syntaxe abstraite

grammar3.rnc

```
expression |=  
  try  
  | codedefinitions  
  | lambda  
  
lambda = element lambda {  
  element variables { variable * },  
  element body      { expression + }  
}  
  
codedefinitions = element codedefinitions {  
  element functions {  
    functionDefinition *  
  },  
  element body { expression + }  
}
```

`functionDefinition` était déjà défini dans `grammar2.rnc` :
{ attribute name, elements variables, element body }

Exemple XML : fonction locale

```
<codedefinitions>
  <functions>
    <functionDefinition name='f'>
      <variables><variable name='x' /></variables>
      <body><variable name='x' /></body>
    </functionDefinition>
  </functions>
  <body>
    <invocation>
      <function><variable name='f' /></function>
      <arguments><integer value='1' /></arguments>
    </invocation>
  </body>
</codedefinitions>

<!-- let rec f x = x in f 1 -->
```

Exemple XML : fonction anonyme (lambda)

```

<invocation>
  <function>
    <block>
      <bindings> <binding> <variable name='x' />
        <initialisation><integer value='1' /></initialisation>
      </binding> </bindings>
      <body>
        <lambda> <variables><variable name='y' /></variables>
          <body> <binaryOperation operator='+'>
            <leftOperand><variable name='x' /></leftOperand>
            <rightOperand><variable name='y' /></rightOperand>
          </binaryOperation> </body>
        </lambda>
      </body>
    </block>
  </function>
  <arguments><integer value='2'></arguments>
</invocation>

<!-- (let x = 1 in fun y -> x + y) 2 -->

```


Arbre syntaxique (1/2)

IASTcodedefinitions.java

```
package com.paracamplus.ilp3.interfaces;  
public interface IASTcodedefinitions 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 (2/2)

IASTlambda.java

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

IASTnamedLambda.java

```
package com.paracampus.ilp3.interfaces;  
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**, **ASTcodedefinitions** sont de simples conteneurs...

Interprétation

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 : l'environnement global

GlobalVariableEnvironment

- implante une table d'association `Map<String, Object>`;
- d'abord initialisé avec les primitives, par `GlobalVariableStuff` avant même la création d'`Interpreter`;
- puis enrichi avec les fonctions globales par `Interpreter` dans `visit(IASTprogram iast, ...)`;
- l'environnement global est **constant** pendant l'interprétation;
- il est stocké dans un champ `globalVariableEnvironment` de `Interpreter`.

Rappel : l'environnement lexical (1/2)

Principales opérations :

LexicalEnvironment.java

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

Liste chaînée d'associations variable → valeur.

- évolue durant l'interprétation ;
- passé en argument aux visiteurs
`public Object visit(..., ILexicalEnvironment lexenv);`
- au démarrage, environnement vide : `EmptyLexicalEnvironment` ;
- enrichi par les déclarations locales.

Rappel : l'environnement lexical (2/2)

LexicalEnvironment.java

```
private final IASTvariable variable;  
private Object value;  
private final ILexicalEnvironment next;  
  
public ILexicalEnvironment extend(IASTvariable variable, Object value) {  
    return new LexicalEnvironment(variable, value, this);  
}  
  
public Object getValue(IASTvariable key) throws EvaluationException {  
    if ( key.getName().equals(getKey().getName()) ) {  
        return getValue();  
    } else {  
        return getNext().getValue(key);  
    }  
}
```

EmptyEnvironment.java

```
public Object getValue(IASTvariable key) throws EvaluationException {  
    throw new EvaluationException("No such variable " + key.getName());  
}
```

Rappel : environnement lexical et portée (1/2)

Exemple d'enrichissement : variable locale.

Interpreter.java

```
public Object visit([]blueIASTblock 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 à priorité sur une déclaration globale ;
- la déclaration locale **la plus récente** a priorité
 - **extend** ajoute en tête de liste,
 - **getValue** et **update** cherchent de la tête vers la queue (**getNext**).

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

point d'évaluation	environnement	eval(x)
<code>(let x = 1 in (let x = 2 in x + 1) + x)</code>	empty	—
<code>let x = 1 in ((let x = 2 in x + 1) + x)</code>	(x,1)	1
<code>let x = 1 in (let x = 2 in (x + 1)) + x</code>	(x,2) :: (x,1)	2
<code>let x = 1 in (let x = 2 in x + 1) + (x)</code>	(x,1)	1
<code>let x = 1 in (let x = 2 in x + 1) + x ()</code>	empty	—

Note : environnement lexical et partage (1/2)

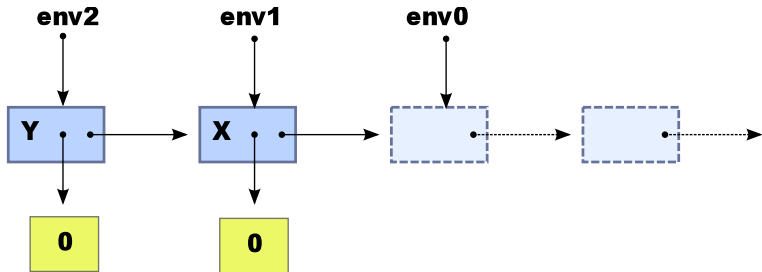
```
env1 = env0.extend(new ASTvariable("X"), new Integer(0));  
env2 = env1.extent(new ASTvariable("Y"), new Integer(0));  
env2.update(new ASTvariable("X"), new Integer(1));
```

- `env1.extend` ne modifie pas son argument `env1`, elle renvoie une nouvelle liste `env2`;
- mais la queue de `env2` et `env1` sont **partagés en mémoire**;
- modifier `X` par `env2.update` modifie aussi `env1.getValue` !

⇒ important pour assurer que l'affectation ILP fonctionne bien !

(penser "liste de références" en OCaml)

Note : environnement lexical et partage (2/2)



Interface des fonctions

ASTlambda retourne une fonction

⇒ il faut donc créer une valeur **ILP** pour les fonctions.

(fonction de première classe = les fonctions sont manipulables comme des valeurs)

Invokable.java

```
package com.paracampus.ilp1.interpreter.interfaces;  
public interface Invokable {  
    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** !

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

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

Fonction close (1/3)

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 et corps;
disponible dans l'AST
- état dynamique : environnement lors de l'interprétation.
disponible seulement lors de l'interprétation

Fonction close (2/3)

Function.java (suite)

```
public int getArity() {  
    return variables.length;  
}  
public IASTvariable[] getVariables() {  
    return variables;  
}  
public IASTexpression getBody() {  
    return body;  
}  
protected ILexicalEnvironment getClosedEnvironment() {  
    return lexenv;  
}
```

Accès à l'état fixé lors de la construction.

Fonction close (3/3)

Function.java (fin)

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

- nous repartons de l'environnement lors de la création de la fonction `getClosedEnvironment`
- et y ajoutons l'association 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(IASLambda 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.

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

Exemple de lambda

Exemple : `(let x = 1 in fun y -> x + y) (2 + 3)`

point d'évaluation	environnement
<code>((let x = 1 in fun y -> x + y) (2 + 3))</code>	empty
<code>(let x = 1 in fun y -> x + y) (2 + 3)</code>	empty
<code>(let x = 1 in (fun y -> x + y)) (2 + 3)</code>	<code>(x,1)</code>
<code>(let x = 1 in fun y -> x + y) (2 + 3)</code>	empty
Application de <code>(fun y -> x + y, (x,1))</code> à 5	
<code>(fun y -> x + y) 5</code>	<code>(x,1)</code>
<code>(x + y)</code>	<code>(y,5) : (x,1)</code>
<code>6</code>	empty

L'environnement capturé par `fun y -> x + y` est `(x,1)`.

Interprétation des codéclarations

Interpreter.java

```
public Object visit(IASTcodedefinitions iast,
ILexicalEnvironment lexenv)
throws EvaluationException {
    IASTnamedLambda[] functions = iast.getFunctions();
    ILexicalEnvironment lexenv2 = lexenv;
    for ( IASTnamedLambda fun : functions ) {
        IASTvariable variable = fun.getFunctionVariable();
        lexenv2 = lexenv2.extend(variable, null);
    }
    for ( IASTnamedLambda fun : functions ) {
        Object f = this.visit(fun, lexenv2);
        IASTvariable variable = fun.getFunctionVariable();
        lexenv2.update(variable, f);
    }
    IASTexpression body = iast.getBody();
    Object result = body.accept(this, lexenv2);
    return result;
}
```

Déclarations locales vs. codéclarations (1/2)

Il est possible d'introduire des fonctions locales nommées grâce à un bloc et des lambdas :

```
<block>
  <bindings> <binding> <variable name='f' /> <initialisation>
    <lambda><variables>...</variables><body>...</body></lambda>
  </initialisation> </binding> </bindings>
  <body>...</body>
</block>
```

Quelle différence avec une codéclaration ?

```
<codefinitions>
  <functions> <functionDefinition name='f'>
    <variables>...</variables> <body>...</body>
  </functionDefinition> </functions>
  <body>...</body>
</codefinitions>
```

Déclarations locales vs. codéclarations (2/2)

```
let f x = 12 in (let f x = f (x + 1))
```

- bloc : $f (x + 1)$ dénote f
- codéclaration : $f (x + 1)$ dénote f

visiteur de IASTcodeDefinitions

```
...
for ( IASTNamedLambda fun : functions ) {
    IASTvariable variable = fun.getFunctionVariable();
    lexenv2 = lexenv2.extend(variable, null);
}
for ( IASTNamedLambda fun : functions ) {
    Object f = this.visit(fun, lexenv2);
    IASTvariable variable = fun.getFunctionVariable();
    lexenv2.update(variable, f);
}
...
```

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

⇒ la codéclaration permet les fonctions (mutuellement) récursives.
 (en OCaml, différence entre `let` et `let rec`)

Efficacité des environnements

LexicalEnvironment.java

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

LexicalEnvironment est une liste chaînée

- coût **linéaire** à **chaque lecture** ou **modification** de variable !
⇒ **très peu efficace**

Coût des lambdas :

- construction en temps constant : copie du pointeur **lexenv**.
- mais on garde des références sur des variables inutiles
⇒ **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
  ⇒ {
      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
  fun y -> x + y
) 10
  ⇒
ILP_Object anon( /*omis*/ ILP_Object y)
{
    ILP_Object x = ??? // valeur non disponible !
    return ILP_Plus(x, y);
}
ILP_Object ilp_program()
{
    ILP_Object iltmp1;
    ILP_Object iltmp2 = ILP_Integer2ILP(10);
    {
        ILP_Object x = ILP_Integer2ILP(2);
        iltmp1 = ??? // fonction anon ?
    }
    return ILP_invoke(iltmp1, 1, iltmp2);
}

```

- une fonction ILP est aussi traduite en une fonction C ;
- pas de fonction locale en C → `anon` est globale ;
- `anon` est donc déclarée hors de la portée de `x` !
 ⇒ 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 :

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;
```

clôture =

- un pointeur sur une fonction **f** ;
- **arity** : le nombre d'arguments ;
- l'environnement : une liste de **argc** valeurs.

Exemple :

fun 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 argc, ...);
```

`ILP_make_closure` alloue la structure et la remplit avec l'environnement.

`ILP_invoke`

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

Remarquons d'ailleurs que toutes les fonctions 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 let f () = x := x + 1 in f(); x`

Le résultat attendu est 2, mais :

- `f` modifie une copie `x` ;
- le programme renvoie le `x` original.

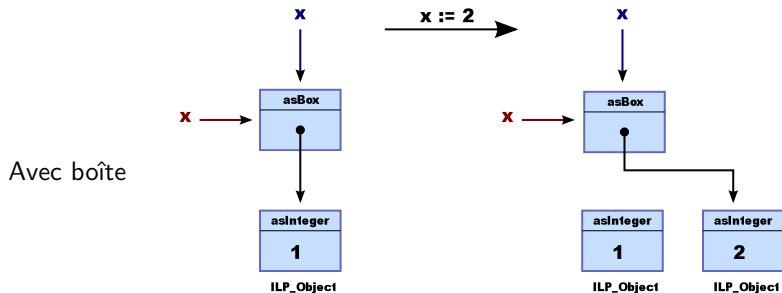
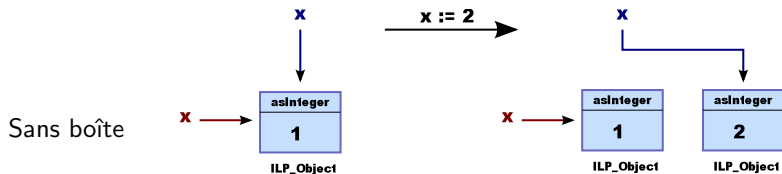
Solution : les **boîtes**.

- 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



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**
⇒ 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 !

(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)

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 : `(let x = 2 in fun y -> x + y) 10.`

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

Exemple complet : fonction anonyme (2/2)

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);
    }
}

```

Génération de la clôture de `fun y -> x + y` dans la portée de `x`,
et appel de fonction avec clôture en dehors de la portée de `x`.

Exemple de code généré pour les codéfinitions

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));
}
```

Exemple : `let rec f x = f (x-1) in f 12`

`f` est dans l'environnement `f` ; nécessaire car `f` est récursive

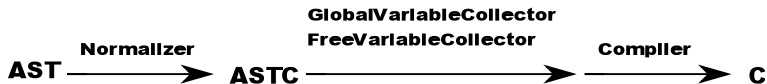
⇒ la clôture `f_2` de `f_1` pointe donc sur elle-même.

nécessite une création en deux temps

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

Compilateur

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.

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 `IASTcodedefinition`;
- ③ autre : fonction stockée dans une variable, retournée par un appel.

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).

Analyse des variables libres : principe

Exemple : pour appliquer la fonction `fun y -> x + y` il suffit de connaître la valeur de `x`.

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

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

Une variable locale non libre est dite **liée**.

Calcul des variables libres FV :

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

- variable : $FV(V) = \{V\}$
- affectation : $FV(V=e) = \{V\}$
- fonction : $FV(\text{fun } x \rightarrow e) = FV(e) \setminus \{x\}$
- bloc : $FV(\text{let } x = e \text{ in } f) = FV(e) \cup (FV(f) \setminus \{x\})$
- opération : $FV(e + f) = FV(e) \cup FV(f)$

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

Exemple : dans `fun 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<...> variables) throws ... {  
    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<...> variables) throws ... {
    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 (résumé)

Compiler.java (extrait simplifié)

```
public Void visit(IASTCprogram iast, Context context) throws ... {  
    for (IASTCglobalVariable gv : iast.getGlobalVariables()) ...  
    for (IASTfunctionDefinition ifd : iast.getFunctionDefinitions())  
        this.emitPrototype(ifd, c);  
    for (IASTClambda closure : iast.getClosureDefinitions())  
        this.emitPrototype(closure, c);  
    for (IASTfunctionDefinition ifd : iast.getFunctionDefinitions()) {  
        this.visit(ifd, c);  
        emitClosure(ifd, c);  
    }  
    for (IASTClambda closure : iast.getClosureDefinitions())  
        this.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 (résumé)

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 (résumé)

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

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

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 context)`

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. : `let f x = x + 1 in f 2`

Cas où la fonction est le résultat d'une expression.

e.g. : `(fun x -> x + 1) 2`

e.g. : `let x = fun y -> y + 1 in x 2`

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

Génération des créations de clôtures (résumé)

`ILP_make_closure` est généré pour :

- Les nœuds `ASTClambda`

La clôture est la valeur renvoyée 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 `variable 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` 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 quoi transporter ;
- **emboîtement** des variables, pour l'accès cohérent à travers des références multiples.

Clés :

- bien savoir quelle variable est référencée à l'exécution ;
- s'assurer statiquement que la variable est toujours accessible.