Correction du partiel

4I501 – DLP : Développement d'un langage de programmation
Master STL, Sorbonne Université

Antoine Miné

Année 2018–2019

Cours 10 bis
11 décembre 2018

Sujet : comptage des appels de fonctions

exemple

```
function :comp deuxfois(x)
    ( 2 * x );

function puissancesix(x)
    (deuxfois(x) * deuxfois(x) * deuxfois(x));

let x = puissancesix in x(2)
```

résultat

```
La fonction deuxfois a été appelée 3 fois.
```

Comptage du nombre d'appels à une ou plusieurs fonctions :

- pour toutes les fonctions marquées du mot-clé `:comp`
- information dynamique, qui doit être déterminée à l'exécution
  (ne pas confondre avec le nombre de sites d'appels syntaxiques de la fonction)
- affiche les compteurs en fin de programme

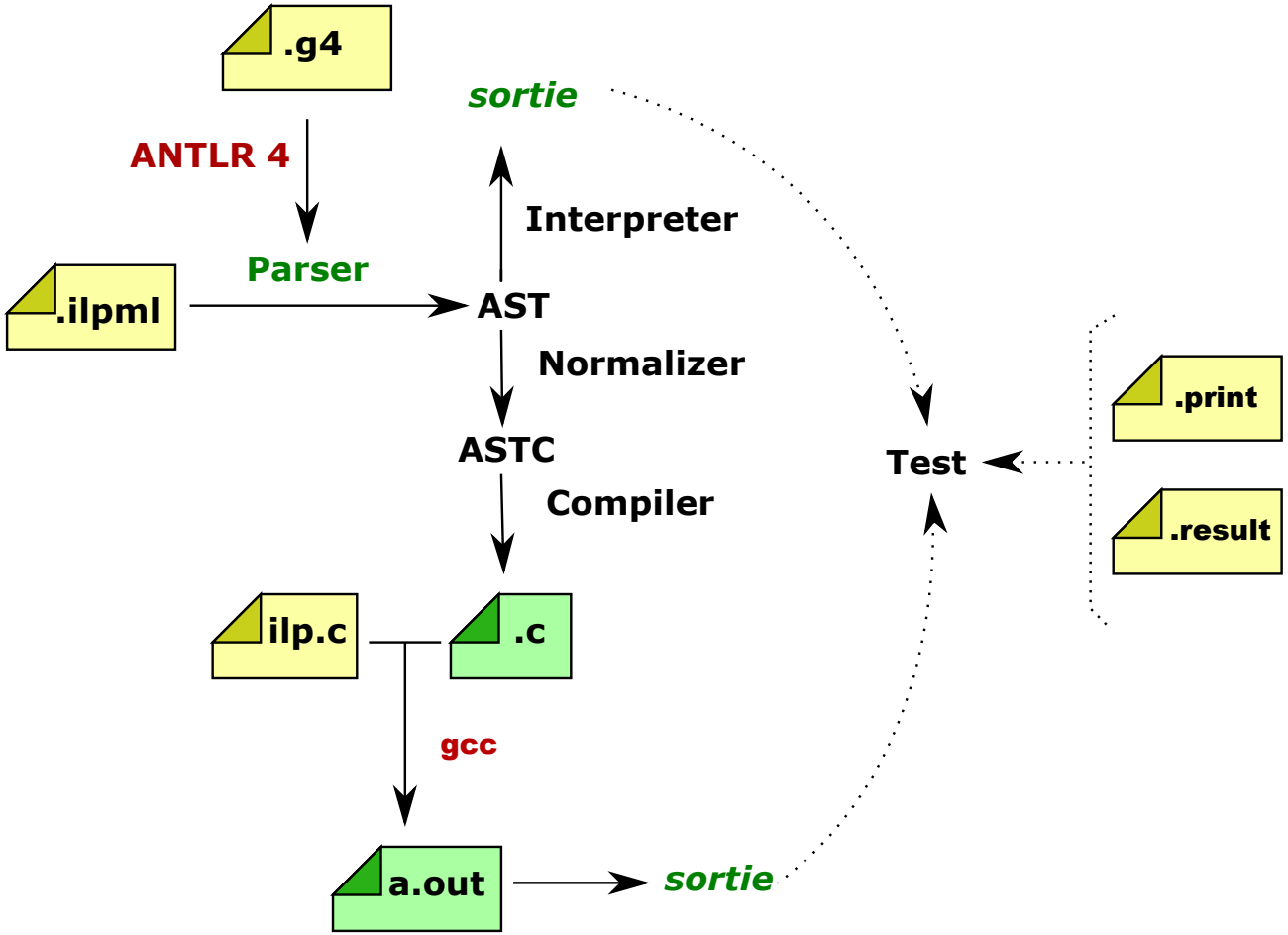
$\Longrightarrow$ information de *profiling*, utile pour guider les optimisations.

Plan

- Définition d'une extension (rappels).
  Stratégie d'implantation pour cette extension.
- Questions 1–3 : Grammaire, AST, analyse syntaxique.
- Question 4 : Interprétation.
- Question 5 : Compilation.
- Extra :
  - les fonctions de première classe (ILP3) ;
  - les techniques de *profiling*.

Rappels, stratégie

Rappels : structure d'ILP



Rappels : étapes d'une extension

Extension de la syntaxe :

- écrire une grammaire ANTLR 4 ;
- ajouter des nœuds `IAST`, `AST`, `ASTC` ;
- écrire un *Listener* obéissant à l'interface produite par ANTLR.

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

Extension des visiteurs :

- classes `Interpreter`, `Normalizer`, `Compiler`.

Extension des primitives et opérateurs de l'interprète Java.

Extension de la bibliothèque d'exécution C :

- type `ILP_Object` dans `ilp.h` ;
- primitives dans `ilp.c`.

Extension des classes de test `InterpreterTest` et `CompilerTest`.

Ces étapes ne sont pas toutes nécessaires pour chaque extension.

Rappels : règles de programmation pour les extensions

En Java :

- **pas de modification du code existant** ;
- nouvelles classes dans des "packages" séparés `com.paracamplus.ilp2.partiel1819...` ;
- réutilisation par héritage ;
- motifs visiteur et composite facilitant l'extensibilité.

En ANTLR 4 :

- difficile d'hériter d'une grammaire `.g4` pour y ajouter des règles ;
- si la grammaire change, la classe *Listener* ne peut pas être réutilisée ;
  (ANTLR génère une nouvelle interface *Listener* sans lien d'héritage avec l'ancienne)

$\Longrightarrow$ copie nécessaire, puis modification de la grammaire et du *Listener*.

En C :

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

Stratégies possibles

Problèmes :

- où stocker les compteurs ?
- quand incrémenter les compteurs ?

Plusieurs choix possibles :

- compteurs dans une table globale de l'interprète (Java) ;
- compteurs comme variables globales (C) ;
- compteur comme champ dans un objet fonction (C, Java) ;
- compter dans l'appelant, ou dans l'appelé.

Note : la cible d'un appel de fonction n'est pas toujours connu statiquement

e.g., `function :comp f() ( ...); let g = f in g()`

$\Longrightarrow$ il est donc plus facile de compter dans l'appelé.

Travail à faire

- AST :
  - ajout d'un nœud AST pour la définition de fonctions qui comptent en restant compatible dans le cas "pas de comptage"
  - enrichir la fabrique `IASTfactory`
  - **ne pas** enrichir l'interface de visiteur `IASTvisitor`

    `IASTfunctionDefinition` n'est pas une expression ! pas de `visit` associé
- grammaire :

  remplacer les règles de déclaration de fonctions

  les nouvelles règles reconnaissent les anciens programmes
- interprète :

  enrichir `Function` avec une version qui compte
  - ajout d'un attribut compteur, avec *getter*
  - modification de `apply` pour mettre à jour le compteur
- compilateur :
  - version ASTC du nœud AST et extension de la normalisation
  - générer des variables globales compteurs
  - générer la mise à jour au début des fonctions `:comp`
  - modifier `main` pour afficher tous les compteurs avant de quitter
- bibliothèque d'exécution C : rien à faire

Classes et interfaces à ajouter 1/3

- interfaces AST, dans `com...partiel1819.interfaces`
  - `IASTcountingFunctionDefinition`
    étend `IASTfunctionDefinition`
  - `IASTfactory`
    étend la version ILP2
- classes AST, dans `com...partiel1819.ast`
  - `ASTcountingFunctionDefinition`
    étend `ASTfunctionDefinition`, implante `IASTcountingFunctionDefinition`
  - `ASTfactory`
    étend la version ILP2, implante `IASTfactory`
- *parseur*
  - grammaire ANTLR : `ILPMLgrammarPartiel1819.g4`
  - *listener* : `ILPMLListener`
    implante `ILPMLgrammarPartiel1819Listener`, généré par ANTLR
  - `ILPMLParser`
    étend la version ILP2

Classes et interfaces à ajouter 2/3

- interface interprète
  dans `com...partiel1819.interpreter.interfaces`
  - `ICountingFunction`
    étend `IFunction`
- classes interprète
  dans `com...partiel1819.interpreter`
  - `CountingFunction`
    étend `Function`, implante `ICountingFunction`
  - `Interpreter`
    étend la version ILP2

Classes et interfaces à ajouter 3/3

- interfaces ASTC
  dans `com...partiel1819.compiler.interfaces`
  - `IASTCcountingFunctionDefinition`
    étend `IASTCfunctionDefinition` et `IASTcountingFunctionDefinition`
- classe ASTC, dans `com...partiel1819.compiler.ast`
  - `ASTCcountingFunctionDefinition`
    étend `ASTCfunctionDefinition`, implante `IASTCcountingFunctionDefinition`
- normalisation, dans `com...partiel1819.compiler.normalizer`
  - `INormalizationFactory`
    étend la version ILP2
  - `NormalizationFactory`
    étend la version ILP2, implante `INormalizationFactory`
  - `Normalizer`
    étend la version ILP2
- classe compilateur, dans `com...partiel1819.compiler`
  - `Compiler`
    étend la version ILP2

Question 1–3 : Grammaire et AST

Grammaire ANTLR4

ILPgrammarPartiel1819.g4

```
grammar ILPMLgrammarPartiel1819;

@header { package antlr4; }

prog returns [com.paracamplus.ilp2.interfaces.IASTprogram node]
    : (defs+=globalCountingFunDef ';'?)*  (exprs+=expr ';'?) * EOF;

globalCountingFunDef
returns [com ... IASTcountingFunctionDefinition node]
    : 'function' (count=':comp')? name=IDENT
      '(' vars+=IDENT? (',' vars+=IDENT)* ')'
      body=expr;
...
```

- remplacement de globalFunDef par globalCountingFunDef
- avec un mot-clé optionnel :comp (utilisation de l'opérateur ?)
- la définition de expr est identique à celle de ILP2

Note sur la génération de grammaires

ANTLR permet d'étendre une grammaire avec le mot-clé `import` :

- permet d'ajouter ou modifier complètement une règle
  pas d'ajouter ou changer individuellement des cas dans une règle
  e.g., ajouter un type d'expressions
- et crée des dépendances entre les fichiers générés
  erreurs fréquentes de package et de chemin de source → erreurs de compilation Java

$\Longrightarrow$ à éviter

Solution pratique : garder les grammaires indépendantes

- ne pas utiliser `import`
- recopier les parties non modifiées des grammaires parents
  toutes les règles lexicales de ILP1 : `IDENT`, `INT`, ...
  en plus, pour le partiel : `expr`
- ne pas oublier `@header { package antlr4l }` en tête de fichier
- générer avec l'option :

  ```
  -o ~/workspace/ILP-UPMC/target/generated-sources/antlr4/
  ```

  sous Eclipse : clic droit sur le fichier `.g4` → Run As → External Tools Configurations ... → onglet Tools → zone Arguments → ajouter l'option `-o ...`

Interface d'AST

IASTcountingFunctionDefinition.java

```
package com.paracamplus.ilp2.partiel1819.interfaces;

import com.paracamplus.ilp2.interfaces.IASTfunctionDefinition;

public interface IASTcountingFunctionDefinition
extends IASTfunctionDefinition
{
    boolean isCounting();
}
```

Drapeau "counting", indiquant si la fonction compte, ou pas.

Classe d'AST

ASTcountingFunctionDefinition.java

```
package com.paracamplus.ilp2.partiel1819.ast;
import ...

public class ASTcountingFunctionDefinition
extends ASTfunctionDefinition
implements IASTcountingFunctionDefinition
{
    public ASTcountingFunctionDefinition
        (IASTvariable functionVariable, IASTvariable[] variables,
         IASTexpression body, boolean counting)
   {
       super(functionVariable, variables, body);
       this.counting = counting;
   }

   private boolean counting;
   @Override public boolean isCounting()  return counting;
}
```

`counting` est défini à la création du nœud.

Fabrique d'AST

IASTfactory.java (partiel)

```
package com.paracamplus.ilp2.partiel1819.interfaces;
...
public interface IASTfactory extends com.paracamplus.ilp2.interfaces.IASTfactory
{
    ASTcountingFunctionDefinition newCountingFunctionDefinition
        (IASTvariable functionVariable, IASTvariable[] variables,
         IASTexpression body, boolean counting);
}
```

ASTfactory.java (partiel)

```
package com.paracamplus.ilp2.partiel1819.ast;
...
public class ASTfactory
extends com.paracamplus.ilp2.ast.ASTfactory implements IASTfactory
{
    @Override public ASTcountingFunctionDefinition newCountingFunctionDefinition
        (IASTvariable functionVariable, IASTvariable[] variables,
         IASTexpression body, boolean counting)
    {
        return new ASTcountingFunctionDefinition
            (functionVariable, variables, body, counting);
    }
}
```

Ajout de la création de nœuds `IASTcountingFunctionDefinition`.

Listener ANTLR

ILPMLListener.java (partiel)

```
package com.paracamplus.ilp2.partiel1819.parser;
...
public class ILPMLListener implements ILPMLgrammarPartiel1819Listener
{
    ...
    @Override
    public void exitGlobalCountingFunDef(GlobalCountingFunDefContext ctx)
    {
        ctx.node = factory.newCountingFunctionDefinition(
            factory.newVariable(ctx.name.getText()),
            toVariables(ctx.vars, false),
            ctx.body.node,
            (ctx.count==null) ? false : true
        );
    }
}
```

- copie du *listener* ILP2
  en changeant `GlobalFunDef` en `GlobalCountingFunDef` ;
- utilisation de la fabrique `factory` pour créer le nœud ;
- la présence du mot-clé dans la règle `(count=':comp')?` est indiqué par un contexte `ctx.count` non `null`
  le contenu réel, ici la chaîne ":comp" importe peu ; seule la présence compte

Lancement de l'analyse syntaxique ANTLR 4

ILPMLParser.java (partiel)

```
package com.paracamplus.ilp2.partiel1819.parser;
import antlr4.ILPMLgrammarPartiel1819Lexer;
import antlr4.ILPMLgrammarPartiel1819Parser;
import com.paracamplus.ilp2.partiel1819.interfaces.IASTfactory;
...
public class ILPMLParser
extends com.paracamplus.ilp2.parser.ilpml.ILPMLParser
{
  public ILPMLParser(IASTfactory factory) { super(factory); }

  @Override public IASTprogram getProgram() throws ParseException
  {
    try {
      ANTLRInputStream in = new ANTLRInputStream(input.getText());
      ILPMLgrammarPartiel1819Lexer lexer = new ILPMLgrammarPartiel1819Lexer(in);
      CommonTokenStream tokens = new CommonTokenStream(lexer);
      ILPMLgrammarPartiel1819Parser parser = new ILPMLgrammarPartiel1819Parser(tokens);
      ILPMLgrammarPartiel1819Parser.ProgContext tree = parser.prog();
      ParseTreeWalker walker = new ParseTreeWalker();
      ILPMLListener extractor = new ILPMLListener((IASTfactory)factory);
      walker.walk(extractor, tree);
      return tree.node;
    } catch (Exception e) { throw new ParseException(e); }
  }
}
```

Question 4 : Interprète

Rappels : fonctions globales

Principe :

L'interprète associe à chaque fonction globale `f` :

- un objet `IFunction` (à ne pas confondre avec un nœud AST)
- et l'associe au nom `f` dans l'environnement global
- lors de l'initialisation (visite du nœud `IASTprogram`).

Lors d'un appel de fonction `f(arg1,...,argN)`, l'interprète :

- évalue récursivement les arguments `arg1`,...,`argN` ;
- retrouve l'objet `IFunction` associé à `f` ;
- appelle sa méthode `apply`.

Avantages :

- l'objet `IFunction` peut être stocké dans une variable et retrouvé ;
- vision "fonctions comme valeurs" qui préfigure les fonctions de première classe.

Rappels : définition des fonctions

IFunction / Invocable (ILP2)

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

public interface IFunction extends Invocable { }
```

- conteneur pour le corps, l'arité et les arguments formels ;
- méthode `apply` qui lie les arguments formels et réels
  et appelle récursivement l'interprète sur le corps de la fonction.

Rappels : implantation des fonctions

Function.java (ILP2)

```
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;}

    @Override
    public Object apply(Interpreter interpreter, Object[] arguments)
    throws EvaluationException
    {
        if ( arguments.length != getArity() )
            throw new EvaluationException("Wrong arity");

        ILexicalEnvironment lexenv2 = getClosedEnvironment();
        IASTvariable[] variables = getVariables();
        for ( int i = 0 ; i < arguments.length ; i++ )
            lexenv2 = lexenv2.extend(variables[i], arguments[i]);
        return getBody().accept(interpreter, lexenv2);
    }
}
```

Rappels : appel de fonction

Interpreter.java (ILP2)

```
@Override public Object visit(IASTinvocation iast, ILexicalEnvironment lexenv)
throws EvaluationException {
    Object function = iast.getFunction().accept(this, lexenv);
    if ( function instanceof Invocable ) {
        Invocable f = (Invocable)function;
        List<Object> args = new Vector<Object>();
        for ( IASTexpression arg : iast.getArguments() ) {
            Object value = arg.accept(this, lexenv);
            args.add(value);
        }
        return f.apply(this, args.toArray());
    } else {
        String msg = "Cannot apply " + function;
        throw new EvaluationException(msg);
    }
}
```

- évaluation de l'expression qui donne la fonction ;
- vérification que la valeur est bien une fonction (`Invocable`) ;
- évaluation des expressions des arguments ;
- puis appel à l'`Invocable` qui fait le reste.

Rappel : les arguments sont évalués dans le contexte lexical de l'appelant, et le corps est évalué dans le contexte lexical de la définition de fonction (différents) !

Fonctions avec compteur : interface

ICountingFunction.java (partiel)

```
package com.paracamplus.ilp2.partiel1819.interpreter.interfaces;

public interface ICountingFunction extends IFunction
{
    int getCounter();
}
```

Principe :

Ajouter une version de `IFunction` qui :

- maintient un compteur ;
- incrémente le compteur à chaque appel (`apply`).

L'exécution du programme mélangera des `ICountingFunction` et des `IFunction` classiques (pour les fonctions sans mot-clé `:comp`).

Fonctions avec compteur : implantation

CountingFunction.java (partiel)

```
package com.paracamplus.ilp2.partiel1819.interpreter;
...
public class CountingFunction
extends Function implements ICountingFunction
{
    private int counter;
    @Override public int getCounter() { return counter; }

    public CountingFunction
        (IASTvariable[] variables, IASTexpression body, ILexicalEnvironment lexenv)
    { super(variables, body, lexenv); }

    @Override public Object apply
        (com.paracamplus.ilp1.interpreter.Interpreter interpreter, Object[] argument)
    throws EvaluationException
    {
        counter++;
        return super.apply(interpreter,  argument);
    }
}
```

Interprète : création des fonctions avec compteur (1/2)

Interpreter.java (partiel) (début)

```
package com.paracamplus.ilp2.partiel1819.interpreter;

public class Interpreter extends com.paracamplus.ilp2.interpreter.Interpreter
{
    @Override public Object visit(IASTprogram iast, ILexicalEnvironment lexenv)
    throws EvaluationException
    {
        for ( IASTfunctionDefinition fd : iast.getFunctionDefinitions) )
        {
            Object f = this.visit((IASTcountingFunctionDefinition)fd, lexenv);
            String v = fd.getName();
            getGlobalVariableEnvironment().addGlobalVariableValue(v, f);
        }
        try {
            return iast.getBody().accept(this, lexenv);
        } catch (Exception exc) {
            return exc;
        } finally {
            printCounters(iast);
        }
    }
```

- visite des nœuds de fonctions `IASTcountingFunctionDefinition`
- le *cast* n'échoue pas car tous nos nœuds sont de classe `ASTcountingFunctionDefinition` (et pas `ASTfunctionDefinition`)

Interprète : création des fonctions avec compteur (2/2)

Interpreter.java (partiel) (suite)

```
public Invocable visit(IASTcountingFunctionDefinition iast, ILexicalEnvironment lexenv)
throws EvaluationException
{
    Invocable fun;
    if ( iast.isCounting() )
        fun = new CountingFunction(iast.getVariables(),
                                   iast.getBody(),
                                   new EmptyLexicalEnvironment());
    else
        fun = new Function(iast.getVariables(),
                           iast.getBody(),
                           new EmptyLexicalEnvironment());
    return fun;
}
```

Selon la valeur de `isCounting()`, création :

- d'une fonction classique `IFonction`
- ou avec comptage `ICountingFunction`

Note : malgré son nom, cette méthode `visit` ne fait pas partie de l'interface de `IASTvisitor` ; c'est une méthode utilitaire, locale à la classe. `IASTcountingFunctionDefinition` n'obéit pas au motif composite.

Interprète : affichage des compteurs

Interpreter.java (partiel) (fin)

```
public void printCounters(IASTprogram iast) throws EvaluationException
{
    UnaryPrimitive print =
        (UnaryPrimitive) getGlobalVariableEnvironment().getGlobalVariableValue("print");
    for ( IASTfunctionDefinition fd : iast.getFunctionDefinitions() )
    {
        Object o = getGlobalVariableEnvironment().getGlobalVariableValue(fd.getName());
        if (o instanceof ICountingFunction)
        {
            ICountingFunction f = (ICountingFunction) o;
            print.apply("La fonction " + fd.getName() +
                        " a été appelée " + f.getCounter() + " fois.");
        }
    }
}
```

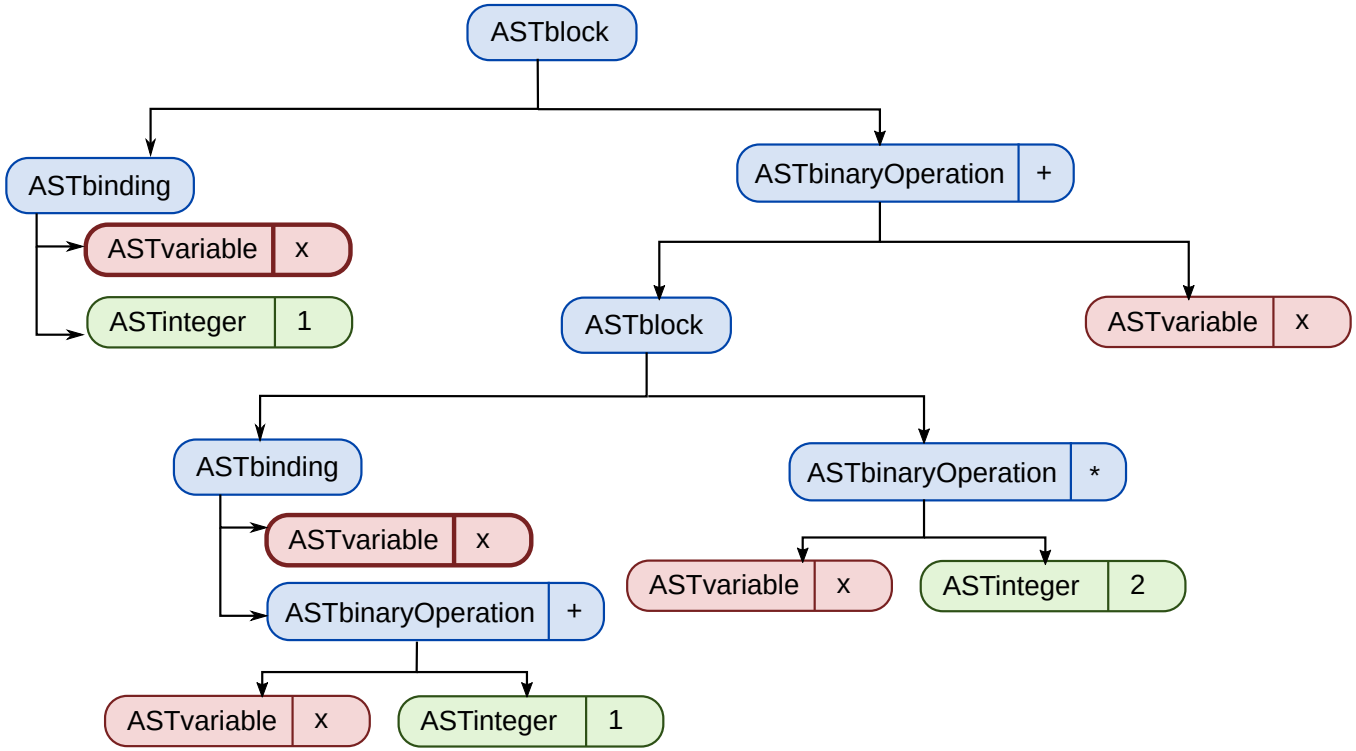
Pour retrouver la liste des fonctions avec compteur :

- itération sur toutes les fonctions de l'AST : `getFunctionDefinitions()` ;
- récupération de la valeur-fonction dans l'environnement global ;
- test que la valeur est bien de type `ICountingFunction` ;
- l'affichage se fait dans le canal de sortie de l'interprète (`InterpreterTest`) ; c'est nécessaire pour faire fonctionner les tests JUnit
  $\Longrightarrow$ utilisation de la primitive ILP `print` et pas de `System.out.print`.

Question 5 : Compilateur

Rappels : ASTC normalisé (1/4)

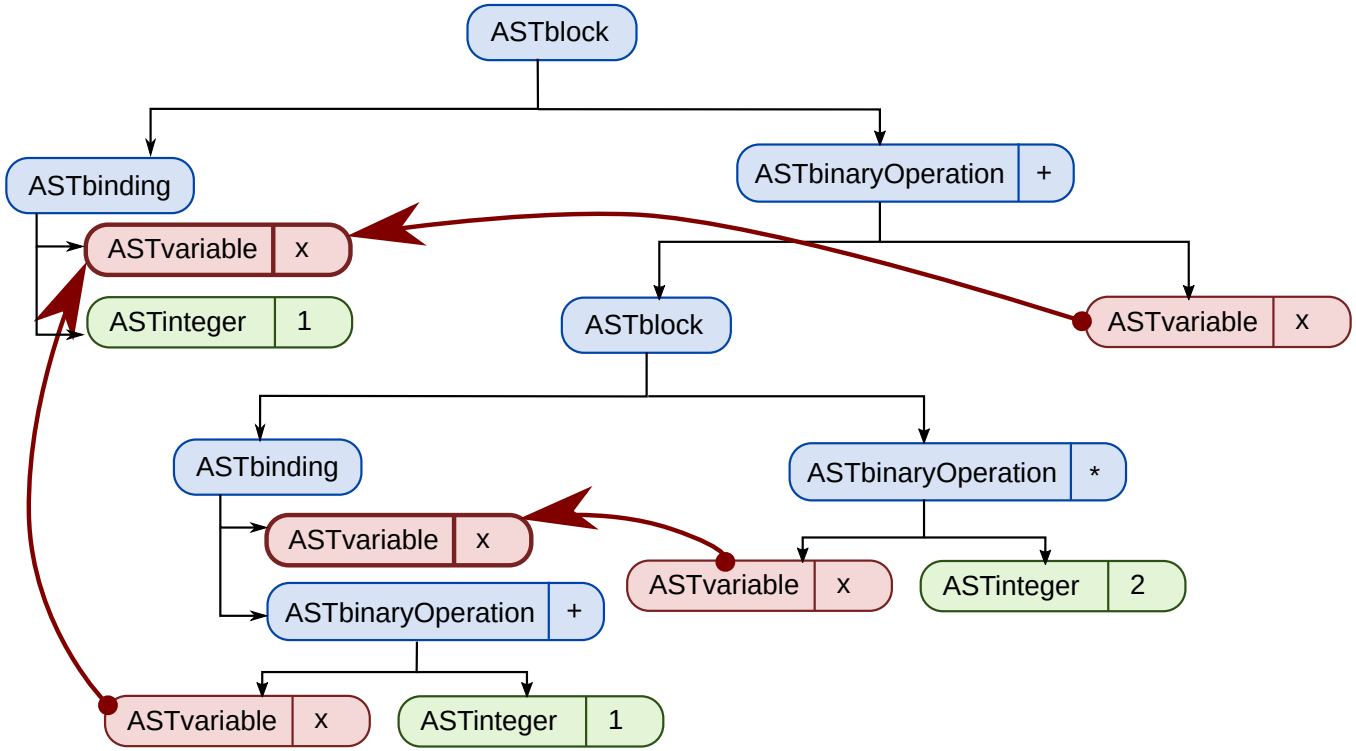
```
let x = 1 in
  (let x = x + 1 in 2 * x) + x
```



Problème : définition et utilisation de variables de même nom (`x`).

Rappels : ASTC normalisé (2/4)

```
let x = 1 in
  (let x = x + 1 in 2 * x) + x
```



Solution : lier chaque utilisation d'une variable à sa définition.

Rappels : ASTC normalisé (3/4)

Le compilateur commence par transformer l'AST en ASTC.

Classification :

- distinction de type d'objet C et de portée :
  - fonction globale : `IASTCglobalFunctionVariable` ;

  $\Longrightarrow$ correspond à une fonction C
  - variable locale : `ASTClocalVariable` ;
  - variable globale : `ASTCglobalVariable` ;

  $\Longrightarrow$ correspond à une variable C contenant un `ILP_Object`

  Il faut distinguer une fonction d'une variable globale contenant une fonction
  e.g., dans `let f = double`, `f` est une `IASTCglobalVariable`
  `double` est une `IASTCglobalFunctionVariable`

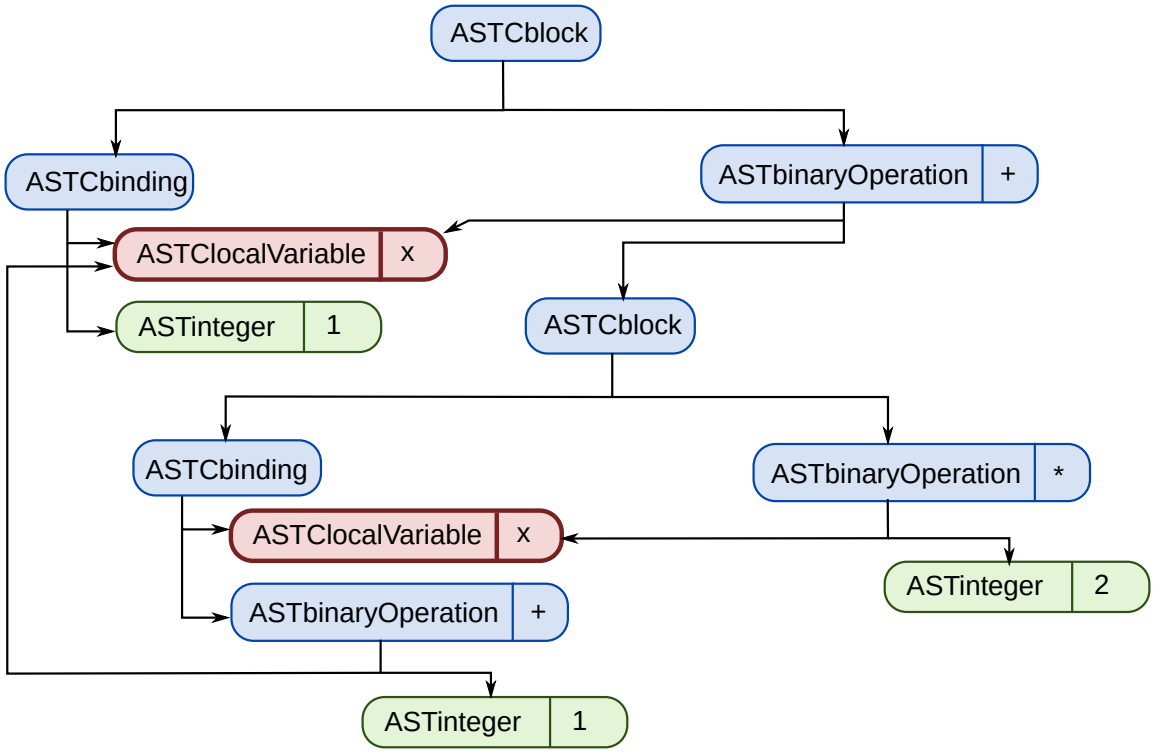
  Une variable référence une fonction avec une valeur de type `ILP_Closure`.
- distinction du type d'appel :
  - direct, par nom : `ASTCglobalInvocation` ;
  - indirect, par variable : `ASTCcomputedInvocation`.

Partage et identification :

- une `ASTCvariable` identifie de manière unique un identificateur ILP et C ;
- toutes les utilisations de la même variable partagent le même nœud.

Rappels : ASTC normalisé (4/4)

```
let x = 1 in
  (let x = x + 1 in 2 * x) + x
```



Extension de l'ASTC

Règles d'extension du compilateur :

- ajouter un nœud ASTC
  pour tout nœud AST contenant des variables ;
- mettre à jour la normalisation ;
- mettre à jour la collecte des variables globales et des variables libres ;
- bien penser à appeler récursivement les visiteurs
  sur les attributs de nœuds.

`ASTcountingFunctionDefinition` référence des variables

$\Longrightarrow$ une version ASTC et la normalisation sont nécessaires.

Extension de l'ASTC

IASTCcountingFunctionDefinition.java (partiel)

```
package com.paracamplus.ilp2.partiel1819.compiler.interfaces;
...
public interface IASTCcountingFunctionDefinition
extends IASTCfunctionDefinition, IASTcountingFunctionDefinition { }
```

ASTCcountingFunctionDefinition.java (partiel)

```
package com.paracamplus.ilp2.partiel1819.compiler.ast;
...
public class ASTCcountingFunctionDefinition
extends ASTCfunctionDefinition implements IASTCcountingFunctionDefinition
{
    private boolean counting;
    @Override public boolean isCounting() { return counting; }

    public ASTCcountingFunctionDefinition
           (IASTvariable functionVariable, IASTvariable[] variables,
            IASTexpression body, boolean counting)
     {
        super(functionVariable, variables, body);
        this.counting = counting;
     }
}
```

`ASTCcountingFunctionDefinition` ajoute l'information `counting`.

Fabrique étendue d'ASTC

INormalizationFactory.java (partiel)

```
package com.paracamplus.ilp2.partiel1819.compiler.normalizer;
...
public interface INormalizationFactory
extends com.paracamplus.ilp2.compiler.normalizer.INormalizationFactory
{
    IASTCcountingFunctionDefinition newCountingFunctionDefinition(
        IASTvariable functionVariable, IASTvariable[] variables,
        IASTexpression body, boolean counting);
}
```

NormalizationFactory.java (partiel)

```
package com.paracamplus.ilp2.partiel1819.compiler.normalizer;
...
public class NormalizationFactory
extends com.paracamplus.ilp2.compiler.normalizer.NormalizationFactory
implements INormalizationFactory
{
    @Override
    public IASTCcountingFunctionDefinition newCountingFunctionDefinition(
        IASTvariable functionVariable, IASTvariable[] variables,
        IASTexpression body, boolean counting)
    {
        return new ASTCcountingFunctionDefinition
            (functionVariable, variables, body, counting);
    }
}
```

Normalisation des fonctions avec compteur

Normalizer.java (partiel)

```
package com.paracamplus.ilp2.partiel1819.compiler.normalizer;
...
public class Normalizer extends com.paracamplus.ilp2.compiler.normalizer.Normalizer
{
    @Override public IASTCfunctionDefinition visit
        (IASTfunctionDefinition iast, INormalizationEnvironment env) throws ...
    {
        String funName = iast.getName();
        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);
        IASTvariable funVar =
            ((INormalizationFactory)factory).newGlobalFunctionVariable(funName);
        if (!(iast instanceof IASTcountingFunctionDefinition))
            throw new CompilationException(...);
        boolean counting = ((IASTcountingFunctionDefinition)iast).isCounting();
        return ((INormalizationFactory)factory)
                .newCountingFunctionDefinition(funVar, newvariables, newbody, counting);
    }
}
```

Rappels : collecte des variables locales et globales

Le compilateur utilise plusieurs passes de visiteur sur l'ASTC :

- `GlobalVariableCollector`

  Collecte les variables et fonctions globales.

  La liste de toutes les globales ILP doit être connue avant de générer le code C, pour générer les déclarations et prototypes C correspondant aux objets globaux.

- `FreeVariableCollector`

  Collecte les variables libres, nécessaire pour les clôtures (ILP3).

Collecte des variables dans les fonctions à compteur

Les nœuds `ASTCcountingFunctionDefinition` :

- implantent `IASTCfunctionDefinition`

  $\Longrightarrow$ les visiteurs ILP2 traitent automatiquement ces nœuds avec la méthode `visit(IASTCfunctionDefinition,...)` ;

- ne contiennent pas de `IASTCvariable` supplémentaire

  $\Longrightarrow$ les visiteurs ILP2 pour `IASTCfunctionDefinition` sont aussi corrects pour nos nouveaux nœuds.

$\Longrightarrow$ inutile de redéfinir `GlobalVariableCollector` et `FreeVariableCollector`.

La situation est donc différente entre la normalisation, qui doit recréer les nœuds, et les collecteurs, qui doivent uniquement retrouver toutes les variables.

Rappels : code généré pour les fonctions

appel direct (ILP)

```
function double(x) (2 * x);
double(27)
```

appel indirect (ILP)

```
function double(x) (2 * x);
let f = double in f(3) - 8
```

appel direct (C généré)

```
ILP_Object ilp__double
  (ILP_Closure ilp_useless,
   ILP_Object x1)
{
  ILP_Object ilptmp2267;
  ilptmp2267 = ILP_Integer2ILP (2);
  return ILP_Times (ilptmp2267, x1);
}

ILP_Object ilp_program ()
{
  return ilp__double (
     NULL,
     ILP_Integer2ILP (27));
}
```

appel indirect (C généré)

```
struct ILP_Closure double_closure_object = {
  &ILP_object_Closure_class,
  {{ilp__double, 1, {NULL}}}
};

ILP_Object ilp_program ()
{
  ILP_Object f2 = &double_closure_object;
  {
    ILP_Object ilptmp2412 =
      ILP_invoke (f2, 1, ILP_Integer2ILP(3));
    return ILP_Minus (ilptmp2412,
                      ILP_Integer2ILP(8));
  }
}
```

Difficulté : appeler une fonction référencée par une variable.

Rappels : motivation pour le schéma de compilation

Quizz : Comment compiler `(let x = 2 in x + 1) * 2` ?

difficulté : en C classique un bloc ne peut pas retourner de valeur

La classe `Compiler`, en Java, génère du C :

- par parcours récursif de l'ASTC
  e.g. : évaluer les arguments d'un opérateur, avant d'évaluer l'opérateur
- en utilisant des variables temporaires
  e.g. : stocker le résultat de la compilation d'une expression
- qui fournit le résultat de l'évaluation au code englobant
  en ILP, tout est expression, tout renvoie une valeur

Le schéma de compilation présente ces étapes de manière concise :

- en donnant le code C généré plutôt que le code Java qui le génère
- en restant générique grâce à un "code à trou" (appels récursifs)
- en utilisant un contexte pour savoir que faire de la valeur de retour

Rappels : schéma de compilation des fonctions

Définition de fonction

`function` *name*(*arg*1, . . . , *argN*) *expr*

```
ILP_Object ilp__name(ILP_Closure ilp_useless,
                     ILP_Object arg1, ..., ILP_Object argN)
{
    ⟶(return)
      expr
}
```

Déclaration de prototype et clôture

```
ILP_Object ilp__name(ILP_Closure ilp_useless,
                     ILP_Object arg1, ..., ILP_Object argN);

struct ILP_Closure name_closure_object = {
    &ILP_object_Closure_class,
    {{ ilp__name, N, {NULL}}}
};
```

Rappels : schéma de compilation des appels (1/2)

Cas IASTCglobalFunctionVariable

$$\frac{\longrightarrow^{d} \mathit{var}(\texttt{arg1}, \ldots, \texttt{argN})}{}$$

```
{
    ILP_Object tmp1;
    ...
    ILP_Object tmpN;
    ⟶(tmp1=)
      arg1
    ...
    ⟶(tmpN=)
      argN
    d  ilp_var.getMangledName() (tmp1, ..., tmp2);
}
```

Le nom de la fonction appelée est connu statiquement.
On génère un appel direct à la fonction C correspondante.

Rappel :

le contexte `d` peut avoir la forme `return`, `temp =` ou `(void)` ;
les temporaires permettent de décomposer les expressions ILP en expressions C simples.

Rappels : schéma de compilation des appels (2/2)

Cas général

$$\frac{\longrightarrow^{d} \texttt{expr}(\texttt{arg1}, \ldots, \texttt{argN})}{\quad}$$

```
{
    ILP_Object tmpF;
    ILP_Object tmp1;
    ...
    ILP_Object tmpN;
    ⟶(tmpF=) expr
    ⟶(tmp1=) arg1
    ...
    ⟶(tmpN=) argN
    d  ILP_invoke(tmpF, tmp1, ..., tmp2);
}
```

Le nom de la fonction appelée n'est pas connu statiquement.
Il est nécessaire d'évaluer une expression à l'exécution pour trouver la fonction, puis de l'appeler par pointeur avec `ILP_invoke`.

Schéma de compilation pour les fonctions à compteur

Définition de fonction avec compteur

$$\texttt{function} : \texttt{comp}\ \mathit{name}(\mathit{arg}1, \ldots, \mathit{arg}N)\ \mathit{expr}$$

```
int  ILP_counter_ilp__name= 0;

ILP_Object ilp__name(ILP_Closure ilp_useless,
                     ILP_Object arg1,..., ILP_Object argN)
{
    ILP_counter_ilp__name++;

    ⟶(return)
      expr
}
```

- génération d'une variable globale `ILP_counter` pour chaque fonction ;
- incrémentation du compteur en début de fonction ;
- puis génération classique du corps ;
- la génération des appels, directs ou indirects, n'a pas changé.

Compilation

Aspects statiques et dynamiques :

- statique : liste des fonctions avec compteur ;
- statique : nom des compteurs de fonction ;
- dynamique : valeur des compteurs de fonction ;
- dynamique : quelle fonction est appelée à chaque invocation.
  (appels indirects à des fonctions, stockées dans des variables, possible dès ILP2)

Bibliothèque d'exécution :

Aucune modification nécessaire à la bibliothèque d'exécution.

Fonction d'affichage des compteurs

Compilateur.java (partiel)

```
public class Compiler
extends com.paracamplus.ilp2.compiler.Compiler {

    @Override
    public Void visit(IASTCprogram iast, Context context)
    throws CompilationException {
        ...
        emit(cBodySuffix);

        emit("void ILP_print_counters() {\n");
        for ( IASTCfunctionDefinition ifd : iast.getFunctionDefinitions() )
          if (isCounting(ifd))
            emit("printf(\"La fonction " + ifd.getName() + " a été appelée %i fois.\",
                         ILP_counter_" + ifd.getCName()+ ");\n ");

        emit("}\n");
        emit(cProgramSuffix);
    }
    ...
}
```

Une fonction d'affichage `ILP_print_counters()` est généré en fin de programme.

`isCounting(ifd)` indique si `ifd` est une `IASTCcountingFunctionDefinition` avec l'attribut `counting` vrai.

Fonction main

Compilateur.java (partiel)

```
protected String cProgramSuffix = "\n"
          + "static ILP_Object ilp_caught_program () {\n"
          + "  struct ILP_catcher* current_catcher = ILP_current_catcher;\n"
          + "  struct ILP_catcher new_catcher;\n\n"
          + "  if ( 0 == setjmp(new_catcher._jmp_buf) ) {\n"
          + "    ILP_establish_catcher(&new_catcher);\n"
          + "    return ilp_program();\n"
          + "  };\n"
          + "  return ILP_current_exception;\n"
          + "}\n\n"
          + "int main (int argc, char *argv[]) \n"
          + "{ \n"
          + "  ILP_START_GC; \n"
          + "  ILP_Object r = ilp_caught_program(); \n"
          + "  ILP_print_counters();\n"
          + "  ILP_print(r); \n"
          + "  ILP_newline(); \n"
          + "  return EXIT_SUCCESS; \n"
          + "} \n";
```

Appel à `ILP_print_counters()` en fin de `main` ;
après avoir évalué le corps (pour avoir la valeur des compteurs)
mais avant d'afficher la valeur de retour (compatibilité avec la sortie de l'interprète).

Extra

Rappel : les fonctions de première classe

ILP3 ajoute les fonctions de première classe,
donc la création d'objets représentant des fonctions à la volée :

- `Function` dans l'interprète ;
- `ILP_Closure` dans le code C compilé ;

au lieu d'une création au démarrage du programme seulement (ILP2).

(Note : le mécanisme d'appel de fonctions, par `apply` / `ILP_invoke`, reste inchangé)

ILP3 ajoute également :

- les fonctions locales ;
- les fonctions anonymes (lambda expressions).

Difficultés liées aux fonctions de première classe

création de fonctions

```
function f(x)
  function :comp g(y) (x+y)
  in g;

f(1)(2);
f(3)(4);
```

fonctions locales

```
x = function :comp f(a) (a+1)
    in f(12);

y = function :comp f(a) (a-1)
    in f(12);
```

fonction anonyme

```
x = (lambda :comp (a) (a+1)) 12;
```

Problèmes :

- faut-il ajouter un compteur par instance de `Function` / `ILP_Closure` ?
  non !
- faut-il distinguer les fonctions locales de même nom ?
  oui !
- comment parler des fonctions anonymes ?
  en indiquant leur position dans le source

Exemple de choix possible

Associer un compteur à chaque fonction syntaxique dans l'AST :

- nœuds `IASTfunctionDefinition` (fonctions globales)
- nœuds `IASTnamedLambda` (fonctions locales)
- nœuds `IASTlambda` (fonctions anonymes)

et référencer les fonctions par leur nom (optionnel) et numéro de ligne.

Ceci permet de :

- garder un ensemble de compteurs fixé statiquement
  - le compilateur continue d'utiliser des variables globales ;
  - l'interprète utilise une table globale à `Interpreter` (`Map`) ;
- compter les exécutions de chaque morceau de code syntaxique.

Le *profiling*

Ensemble de méthodes d'analyse dynamique permettant d'évaluer :

- le nombre d'appels à chaque fonction
  (en distinguant éventuellement par sites d'appel, voir calculer l'arbre d'appels complet)
- le temps d'exécution de chaque fonction
  (temps total ou pire cas, en comptant les fonctions appelées ou pas)
- la consommation mémoire maximale
- la vitesse d'allocation mémoire
- l'utilisation du cache (cache miss)
- etc.

Application principale : guider l'optimisation en évaluant

- les parties de programme inefficaces ;
- les parties de programme qui bénéficieraient d'optimisation ;
- le gain apporté par une optimisation.

Quelques méthodes de *profiling*

Nombreuses méthodes :

- interruption périodique pour inspection
  e.g. : `gprof`
- utilisation de compteurs de performance des processeurs
  e.g. : `oprofile`
- machines virtuelles modifiées
  e.g. : `ocamlprof`, Java
- instrumentation de code binaire par réécriture
  e.g. : `cachegrind`
- etc.

Attention :

- les données peuvent être approximatives ;
- les données ne concernent que les exécutions profilées ;
- l'instrumentation peut perturber le comportement du programme en particulier les données mesurées.