Correction du partiel

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

Antoine Miné

Année 2016–2017

Cours 10 bonus
29 novembre 2016

Sujet : arguments optionnels

exemple

```
function toto1 (a,b &keys c 3, d 4)
(print(a); print(b); print(c); print(d); newline());

toto1(1,2);                          -> 1234

toto1(1,2,:d 4+1);                   -> 1235

toto1(1,2, :d 5, :c 6);              -> 1265
```

Ajout des arguments optionnels à ILP 2.

Les arguments optionnels :

- ont une valeur par défaut (définie dans la déclaration de la fonction)
- ont un nom (défini dans la déclaration, utilisé dans l'appel)
- peuvent être redéfinis ou omis lors de l'appel
- l'ordre des arguments optionnels lors de l'appel n'est pas significatif
  mais ils apparaissent toujours après les arguments obligatoires

Arguments optionnels dans d'autres langages

- OCaml : mécanisme assez similaire
- Java : pas d'argument optionnel
  mais peut être simulé par surcharge de méthodes
- C++ : arguments optionnels et surcharge de fonctions
- C : pas d'arguments optionnels
  mais support non typé pour les arguments variables : `va_arg`
- Python : arguments optionnels avec valeur par défaut
  en Python, les arguments forment un dictionnaire, facilement extensible
  contrepartie : pas de vérification statique possible

Pour éviter les ambiguïtés, il faut généralement :

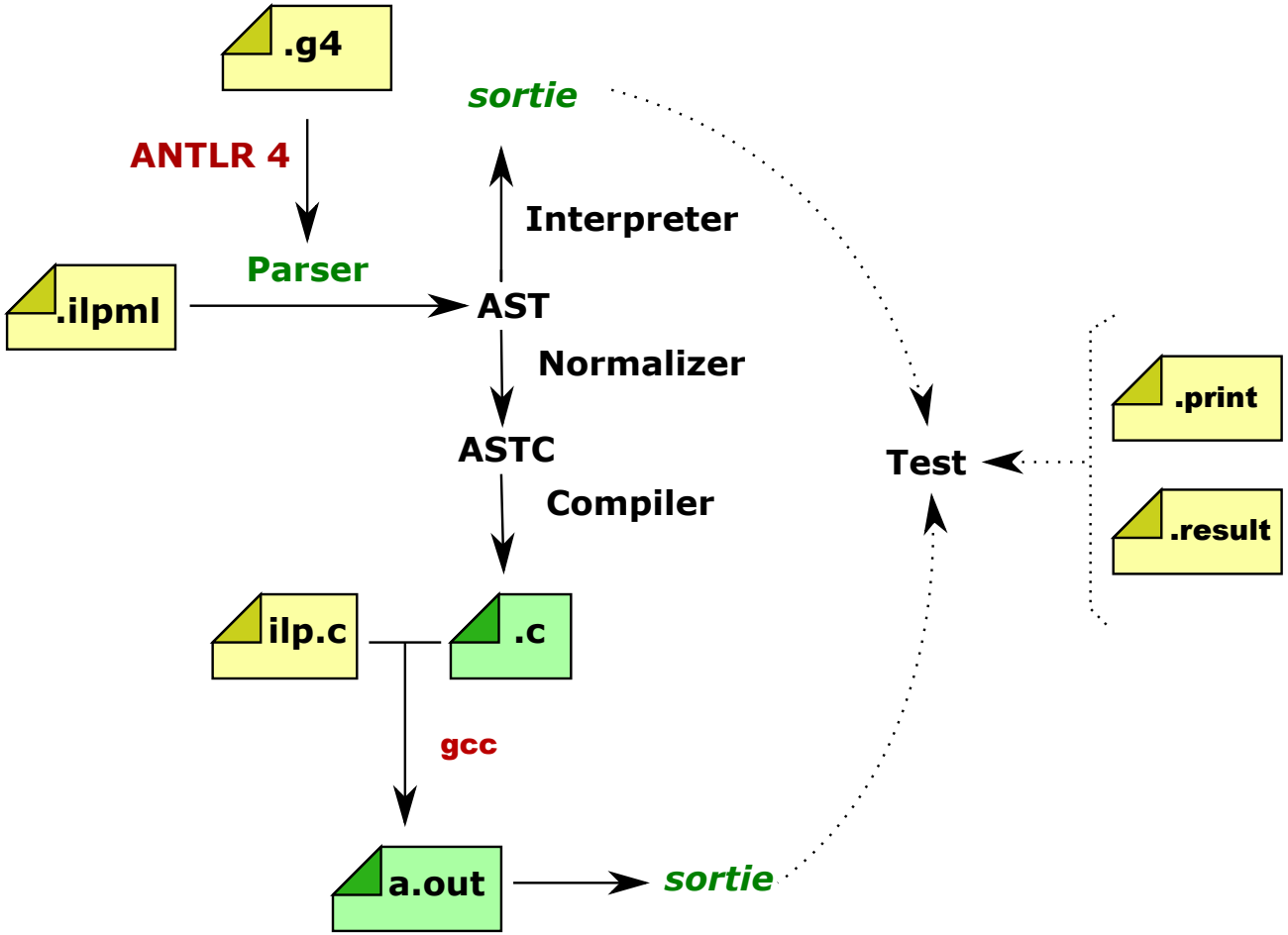
- nommer les arguments optionnels explicitement lors de l'appel
- restreindre les mélanges entre arguments optionnels et obligatoires
- sélectionner statiquement une fonction "meilleure" que toutes les autres et renvoyer une erreur de compilation si aucune fonction n'est meilleure

Plan

- Question 1 : Stratégie
  (rappels, liste des classes à implanter)
- Question 2 : Grammaire et AST
- Question 3 : Interprétation
- Question 4 : Compilation
  (rappels sur les schémas de compilation)
- Discussion et choix d'implantation alternative

Question 1 : Stratégie

Rappel : 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 : `IASTvisitor` :

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

Extension des primitives et opérateurs de l'interprète.

Extension de la bibliothèque d'exécution C :

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

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

Ces étapes ne sont pas toutes nécessaires pour chaque extension !

Rappel : règles de programmation pour les extensions

En Java :

- **pas de modification du code existant** ;
- nouvelles classes dans des "packages" séparés `com.paracamplus.ilp2.partial_16...` ;
- 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.

Choix pour l'ajout des arguments optionnels

- AST :
  ajout de nouveaux nœuds AST pour la déclaration et l'appel
  s'appuyer sur l'existant et ajouter un attribut pour les arguments optionnels
  en restant compatible dans le cas "pas d'argument optionnel"
- grammaire :
  remplacer les règles de déclaration et d'appel de fonction
  les nouvelles règles reconnaissent les anciens programmes
- interprète :
  enrichir l'interprète pour nos nouveaux nœuds
  ⇒ passage des arguments optionnels dans un dictionnaire (`Map` Java)
  enrichir `Function` qui réifie les appels (`apply`)
- compilateur :
  version ASTC des nœuds AST
  extension de la normalisation pour nos nouveaux nœuds
  génération de code des appels de fonctions
  ⇒ ajout des arguments optionnels manquants, puis appel classique
  d'autres choix sont possibles ! voir la fin du cours
- bibliothèque d'exécution C :
  rien à faire ! (tout est fait dans le compilateur)

Classes et interfaces à ajouter (1/5)

Classes et interfaces à ajouter, classées par *package*.

- Grammaire : `com.paracamplus.ilp2.partial_16.parser.ilpml`
  - `ILPMLgrammar2partial16.g4`
    copie modifiée de `ILPMLgrammar2.g4`
  - `ILPMLListner`
    copie modifiée de `ILPMLgrammar2.g4`,
    implémente `ILPMLgrammar2partial16Listener`
  - `ILPMLParser`
    étend la version ILP2

Classes et interfaces à ajouter (2/5)

- interfaces AST : `com...partial_16.interfaces`
  - `IASTfunctionDefinitionOpt` étend `IASTfunctionDefinition`
  - `IASTinvocationOpt` étend `IASTinvocation`
  - `IASTfactory` étend la version ILP2
  - `IASTvisitable` étend la version ILP2
  - `IASTvisitor` étend la version ILP2
- classes AST : `com...partial_16.ast`
  - `ASTfunctionDefinitionOpt`
    étend `ASTfunctionDefinition`, implante `IASTfunctionDefinitionOpt`
  - `ASTinvocationOpt`
    étend `ASTinvocation`, implante `IASTinvocationOpt`
  - `ASTfactory`
    étend la version ILP2, implante `IASTfactory`

Classes et interfaces à ajouter (3/5)

- interface interprète : `com...partial_16.interpreter.interfaces`
  - `IFunctionOpt`
    similaire à `IFunction`, mais avec arguments optionnels
- interprète : `com...partial_16.interpreter`
  - `FunctionOpt`
    étend `Function`, implante `IFunctionOpt`
  - `Interpreter`
    étend la version ILP2
- test interprète : `com...partial_16.interpreter.test`
  - `InterpreterTest`
    étend la version ILP2

Classes et interfaces à ajouter (4/5)

- interfaces ASTC : `com...partial_16.compiler.interfaces`
  - `IASTCfunctionDefinitionOpt`
    étend `IASTfunctionDefinitionOpt`, `IASTCfunctionDefinition`
  - `IASTCglobalInvocationOpt`
    étend `IASTinvocationOpt`, `IASTCglobalInvocation`
  - `IASTCvisitable`
    étend la version ILP2
  - `IASTCvisitor`
    étend la version ILP2
- ASTC : `com...partial_16.compiler`
  - `ASTCfunctionDefinitionOpt`
    étend `ASTCfunctionDefinition`, implante `IASTCfunctionDefinitionOpt`
  - `ASTCglobalInvocationOpt`
    étend `ASTCglobalInvocation`
    implante `IASTCglobalInvocationOpt` et `IASTCvisitable`

Classes et interfaces à ajouter (5/5)

- normalisation : `com...partial_16.compiler.normalizer`
  - `INormalizationFactory` étend la version ILP2
  - `NormalizationFactory` étend la version ILP2
  - `Normalizer` étend la version ILP2
- compilateur : `com...partial_16.compiler`
  - `Compiler` étend la version ILP2
  - `FreeVariableCollector` étend la version ILP2
  - `GlobalVariableCollector` étend la version ILP2
- test compilateur : `com...partial_16.compiler.test`
  - `CompilerTest` étend la version ILP2

Question 2 : Grammaire et AST

Grammaire ANTLR4

ILPgrammar2partial16.g4

```
globalFunDef returns [....ilp2.interfaces.IASTfunctionDefinition node]
: 'function' name=IDENT '('
     vars+=IDENT? (',' vars+=IDENT)*
     ('&keys' nvars+=IDENT nvals+=expr
         (',' nvars+=IDENT nvals+=expr)* )?
   ')' body=expr
;

expr returns [com.paracamplus.ilp1.interfaces.IASTexpression node]
: ...
| fun=expr '(' args+=expr? (',' args+=expr)*
  (',' ':'  names+=IDENT nargs+=expr)*  ')' # Invocation
```

- ajout de parties optionnelles, avec la syntaxe `(...)?` et `(...)*`
- les noms de règles sont inchangés : `globalFunDef`, `Invocation`
- ces règles généralisent les règles précédentes
  $\Longrightarrow$ tout ancien programme est reconnu par la nouvelle grammaire
- utilisation de la règle générique `expr` pour les constantes
  la vérification qu'il s'agit bien d'une constante sera faire plus tard, dans le *Listener*

Interface d'AST : définition de fonction

IASTfunctionDefinitionOpt.java

```
package com.paracamplus.ilp2.partial_16.interfaces;

public interface IASTfunctionDefinitionOpt
extends IASTfunctionDefinition
{
    Map<IASTvariable,IASTconstant> getOptVariables();
}
```

Nous héritons de plus depuis `IASTfunctionDefinition` de :

- `IASTvariable[] getVariables()` : arguments non optionnels
- `IASTexpression getBody()` : corps de la fonction
- `IASTvariable getFunctionVariable()` : nom de fonction

Ajout pour les arguments optionnels d'un dictionnaire `Map`
nom d'argument optionnel → valeur par défaut (expression constante)

`IASTconstant` regroupe toutes les constantes (`IASTboolean`, `IASTfloat`, ...)

Implantation d'AST : définition de fonction

ASTfunctionDefinitionOpt.java

```
package com.paracamplus.ilp2.partial_16.ast;
public class ASTfunctionDefinitionOpt
extends ASTfunctionDefinition implements IASTfunctionDefinitionOpt
{
    private Map<IASTvariable,IASTconstant> optVariables;

    public ASTfunctionDefinitionOpt(IASTvariable functionVariable,
           IASTvariable[] variables,
           Map<IASTvariable,IASTconstant> optVariables,
           IASTexpression body)
    {
        super(functionVariable, variables, body);
        this.optVariables = optVariables;
    }

    public Map<IASTvariable,IASTconstant> getOptVariables()
    { return optVariables; }
}
```

Simple classe conteneur ajoutant le dictionnaire `optVariables`.

Interface d'AST : appel de fonction

IASTinvocationOpt.java

```
package com.paracamplus.ilp2.partial_16.interfaces;

public interface IASTinvocationOpt
extends IASTinvocation, IASTvisitable
{
    Map<IASTvariable,IASTexpression> getOptArguments();
}
```

Nous héritons également de `IASTinvocation` :

- `IASTexpression getFunction()` : la fonction appelée
- `IASTexpression[] getArguments()` : arguments non optionnels

Ajout pour les arguments optionnels spécifiés lors de l'appel d'un dictionnaire `Map` : nom d'argument optionnel $\rightarrow$ expression

$\Rightarrow$ seul le nom, et pas l'ordre, des arguments optionnels est significatif !

Implantation d'AST : appel de fonction

ASTfunctionDefinitionOpt.java (début)

```
package com.paracamplus.ilp2.partial_16.ast;

public class ASTinvocationOpt
extends ASTinvocation implements IASTinvocationOpt
{
    private Map<IASTvariable,IASTexpression> optArguments

    public ASTinvocationOpt(IASTexpression function,
        IASTexpression[] arguments,
        Map<IASTvariable,IASTexpression> optArguments)
        {
            super(function, arguments);
            this.optArguments = optArguments;
        }

        public Map<IASTvariable,IASTexpression> getOptArguments()
        { return optArguments; }
        ...
```

Simple classe conteneur pour ajouter le dictionnaire `optArguments`.

Interfaces de visiteur

IASTvisitor

```
public interface IASTvisitor<Result, Data, Anomaly extends Throwable>
extends com.paracamplus.ilp2.interfaces.IASTvisitor
<Result, Data, Anomaly>
{
    Result visit(IASTinvocationOpt iast, Data data) throws Anomaly;
}
```

IASTvisitable

```
public interface IASTvisitable
extends com.paracamplus.ilp2.interfaces.IASTvisitable
{
    <Result, Data, Anomaly extends Throwable>
    Result accept(IASTvisitor<Result, Data, Anomaly> visitor,
                  Data data) throws Anomaly;
}
```

Attention : classes et interfaces de même nom que pour ILP2, mais dans un *package* différent.

Support de visiteur dans l'AST

ASTfunctionDefinitionOpt.java (suite)

```
@Override
public <Result, Data, Anomaly extends Throwable>
Result accept(IASTvisitor<Result, Data, Anomaly> visitor, Data data) throws Anomaly
{
  return visitor.visit(this, data);
}

@Override
public <Result, Data, Anomaly extends Throwable> Result
accept(com.paracamplus.ilp2.interfaces.IASTvisitor<Result, Data, Anomaly> visitor,
       Data data) throws Anomaly
{
  return accept((IASTvisitor<Result,Data,Anomaly>) visitor, data);
}

/* idem pour com.paracamplus.ilp1.interfaces.IASTvisitor */
```

Pour que `visit(IASTinvocationOpt, Data)` soit appelé, il faut que `visitor` soit de type statique `com.paracamplus.ilp2.partial_16.ast.IASTvisitor`.

$\Longrightarrow$ un *cast* est nécessaire si le type statique n'est pas correct

Attention : en cas d'oubli, le visiteur pour `IASTinvocation` sera appelé !

Fabrique d'AST : interface

IASTfactory.java

```
package com.paracamplus.ilp2.partial_16.interfaces;

public interface IASTfactory
extends com.paracamplus.ilp2.interfaces.IASTfactory
{
    IASTfunctionDefinitionOpt newFunctionDefinitionOpt(
        IASTvariable functionVariable,
        IASTvariable[] variables,
        Map<IASTvariable,IASTconstant> optVariables,
        IASTexpression body);

    IASTinvocationOpt newInvocationOpt(
        IASTexpression function,
        IASTexpression[] arguments,
        Map<IASTvariable,IASTexpression> optArguments);
}
```

Fabrique d'AST : implantation

ASTfactory.java

```
package com.paracamplus.ilp2.partial_16.ast;

public class ASTfactory
extends com.paracamplus.ilp2.ast.ASTfactory implements IASTfactory
{
    @Override
    public IASTfunctionDefinitionOpt newFunctionDefinitionOpt(
    IASTvariable functionVariable, IASTvariable[] variables,
    Map<IASTvariable,IASTconstant> optVariables, IASTexpression body) {
        return new ASTfunctionDefinitionOpt(functionVariable, variables,
                                            optVariables, body);
    }


    @Override
    public IASTinvocationOpt newInvocationOpt(
    IASTexpression function, IASTexpression[] arguments,
    Map<IASTvariable,IASTexpression> optArguments) {
        return new ASTinvocationOpt(function, arguments, optArguments);
    }
}
```

Listener ANTLR 4 (1/2)

ILPMLListener.java (début)

```
package com.paracamplus.ilp2.partial_16.parser.ilpml;

public class ILPMLListener
implements ILPMLgrammar2partial16Listener
{
    protected IASTfactory factory;

    public ILPMLListener(IASTfactory factory)
    { super(); this.factory = factory; }

    /* copie du code de ILPMLListener d'ILP 2 */

    @Override
    public void exitInvocation(InvocationContext ctx)
    {
        Map<IASTvariable,IASTexpression> opt = new HashMap<>();
        if (ctx.nargs != null) {
            for (int i = 0; i < ctx.nargs.size(); i++) {
                IASTvariable v = factory.newVariable(ctx.names.get(i).getText());
                IASTexpression e = ctx.nargs.get(i).node;
                opt.put(v, e);
            }
        }
        ctx.node = factory.newInvocationOpt(
            ctx.fun.node, toExpressions(ctx.args), opt);
    }
```

Listener ANTLR 4 (2/2)

ILPMLListener.java (fin)

```
    @Override
    public void exitGlobalFunDef(GlobalFunDefContext ctx) {
        Map<IASTvariable,IASTconstant> opt = new HashMap<>();
        if (ctx.nvars != null) {
            for (int i = 0; i < ctx.nvars.size(); i++) {
                IASTvariable v = factory.newVariable(ctx.nvars.get(i).getText());
                IASTexpression e = ctx.nvals.get(i).node;
                if (!(e instanceof IASTconstant)) {
                    System.out.println("Constant expected, replaced with false");
                    e = factory.newBooleanConstant("false");
                }
                opt.put(v, (IASTconstant)e);
            }
        }
        ctx.node = factory.newFunctionDefinitionOpt(
            factory.newVariable(ctx.name.getText()),
            toVariables(ctx.vars, false),
            opt,
            ctx.body.node);
    }
}
```

Vérification à la construction de l'AST que les arguments optionnels dans les définitions de fonctions ont bien une valeur constante.

Lancement de l'analyse syntaxique ANTLR 4

ILPMLParser.java

```
package com.paracamplus.ilp2.partial_16.parser.ilpml;

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());
            // flux de caractères -> analyseur lexical
            ILPMLgrammar2partial16Lexer lexer = new ILPMLgrammar2partial16Lexer(in);
            // analyseur lexical -> flux de tokens
            CommonTokenStream tokens =        new CommonTokenStream(lexer);
            // flux tokens -> analyseur syntaxique
            ILPMLgrammar2partial16Parser parser = new ILPMLgrammar2partial16Parser(tokens);
            // démarage de l'analyse syntaxique
            ILPMLgrammar2partial16Parser.ProgContext tree = parser.prog();
            // parcours de l'arbre syntaxique et appels du Listener
            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 3 : Interprétation

Principe

Rappel : décomposition d'un appel de fonction en trois étapes

1. évaluation du nom de la fonction cible
   $\Longrightarrow$ création d'un objet `IFunction` appelable
2. évaluation des arguments
   $\Longrightarrow$ tableau d'`Object`
3. invocation de la fonction cible sur les objets, avec `apply`
   $\Longrightarrow$ retourne un `Object`

Modifications prévues :

- création d'un objet `IFunctionOpt` contenant
  les valeurs par défaut des paramètres optionnels
- stockage des arguments optionnels spécifiés lors de l'appel dans un dictionnaire `Map`, passé en argument à `apply`
- méthode `apply` qui fusionne dans l'environnement
  - les arguments optionnels spécifiés lors de l'appel
  - les arguments optionnels non-spécifiés lors de l'appel

  avant d'exécuter le corps de la fonction

Fonctions : interface

IFunctionOpt.java

```
package com.paracamplus.ilp2.partial_16.interpreter.interfaces;

public interface IFunctionOpt
{
    Object apply(Interpreter interpreter, Object[] argument,
                 Map<String, Object> opt)
    throws EvaluationException;
}
```

Nouvel appel spécifique aux arguments optionnels, passés dans `opt`.

Note : la table est indexée par des `String` contrairement aux nœuds de l'AST, plus facile à comparer et placer dans des dictionnaires que les `ASTvariables`

Pour comparaison, `IFunction` demande (via `Invocable`) :

- `Object apply (Interpreter interpreter, Object[] argument)`
- `int getArity()`

pour la compatibilité, nous imposerons à `FunctionOpt` d'implanter `IFunctionOpt` et `IFunction`

Fonctions : implantation (1/2)

FunctionOpt.java (début)

```
package com.paracamplus.ilp2.partial_16.interpreter;

public class FunctionOpt
extends Function implements IFunctionOpt
{
    private Map<IASTvariable,IASTconstant> opt;

    public FunctionOpt(IASTvariable[] variables,
          Map<IASTvariable,IASTconstant> opt,
          IASTexpression body,
          ILexicalEnvironment lexenv) {
        super(variables, body, lexenv);
        this.opt = opt;
    }

    public Map<IASTvariable,IASTconstant> getOpt() { return opt; }
```

Le dictionnaire `opt` des valeurs par défaut des arguments optionnels fait partie de la fonction
au même titre que le corps (`body`) et la liste des arguments obligatoires (`variables`).

Note : `lexenv` n'est pas utile pour ILP 2, cf. cours sur ILP 3

Fonctions : implantation (2/2)

FunctionOpt.java (suite)

```
    @Override public Object apply
    (Interpreter interpreter, Object[] argument, Map<String, Object> opt)
    throws EvaluationException {
        // Check arity of non-optional arguments
        if ( argument.length != getArity() ) {
            String msg = "Wrong arity";
            throw new EvaluationException(msg);
        }
        // Add non-optional arguments to scope
        ILexicalEnvironment lexenv2 = getClosedEnvironment();
        IASTvariable[] variables = getVariables();
        for ( int i=0 ; i<argument.length ; i++ )
            lexenv2 = lexenv2.extend(variables[i], argument[i]);
        // Add optional arguments to scope (evaluate default arguments)
        Map<IASTvariable, IASTconstant> optvars = getOpt();
        for (IASTvariable v : optvars.keySet()) {
                    Object o;
            if (opt.containsKey(v.getName()))
                o = opt.get(v.getName());
            else
                o = optvars.get(v).accept(interpreter, lexenv2);
            lexenv2 = lexenv2.extend(v, o);
        }
        return getBody().accept(interpreter, lexenv2);
    }
}
```

Interprète (1/3)

Interpreter.java (début)

```
package com.paracamplus.ilp2.partial_16.interpreter;

public class Interpreter
extends com.paracamplus.ilp2.interpreter.Interpreter
implements IASTvisitor<Object, ILexicalEnvironment, EvaluationException>
{
    public Interpreter(IGlobalVariableEnvironment globalVariableEnvironment,
                       IOperatorEnvironment operatorEnvironment)
    { super(globalVariableEnvironment, operatorEnvironment); }

    public Object visit(IASTprogram iast, ILexicalEnvironment lexenv)
    throws EvaluationException
    {
        for ( IASTfunctionDefinition fd : iast.getFunctionDefinitions() ) {
            Object f;
            if (fd instanceof IASTfunctionDefinitionOpt)
                f = this.visit((IASTfunctionDefinitionOpt)fd, lexenv);
            else
                f = this.visit(fd, lexenv);
            String v = fd.getName();
            getGlobalVariableEnvironment().addGlobalVariableValue(v, f);
        }
        try { return iast.getBody().accept(this, lexenv); }
        catch (Exception exc) { return exc; }
    }
```

Test `instanceof` pour distinguer les anciennes définitions de fonctions des nouvelles.

Interprète (2/3)

Interpreter.java (suite)

```
public Invocable visit(IASTfunctionDefinitionOpt iast, ILexicalEnvironment lexenv)
throws EvaluationException
{
    Invocable fun = new FunctionOpt(iast.getVariables(),
                                    iast.getOptVariables(),
                                    iast.getBody(),
                                    new EmptyLexicalEnvironment());
    getGlobalVariableEnvironment().addGlobalVariableValue(iast.getName(), fun);
    return fun;
}
```

Interprète (3/3)

Interpreter.java (fin)

```
    @Override
    public Object visit(IASTinvocationOpt iast, ILexicalEnvironment lexenv)
    throws EvaluationException {
        Object function = iast.getFunction().accept(this, lexenv);
        List<Object> args = new Vector<Object>();
        for ( IASTexpression arg : iast.getArguments() ) {
            Object value = arg.accept(this, lexenv);
            args.add(value);
        }
        if ( function instanceof IFunctionOpt ) {
            IFunctionOpt f =(IFunctionOpt) function;
            Map<IASTvariable, IASTexpression> opt = iast.getOptArguments();
            Map<String,Object> vopt = new HashMap<>();
            for (IASTvariable v : opt.keySet())
                vopt.put(v.getName(), opt.get(v).accept(this, lexenv));
            return f.apply(this, args.toArray(), vopt);
        }
        else if ( function instanceof Invocable ) {
            // Needed for primitives, that are not IFunctionOpt!
            Invocable f =(Invocable) function;
            return f.apply(this, args.toArray());
        }
        else throw new EvaluationException("Cannot apply " + function);
    }
}
```

Test `instanceof` pour distinguer les anciennes fonctions des nouvelles.

Question 4 : Compilation

Principe

Transformation de l'AST en ASTC :

liée à la classification des variables et des appels
$\Longrightarrow$ fortement impactée par l'extension !

- 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

Génération de code

Principe : résolution statique des arguments optionnels à chaque appel

- génération à chaque site d'appel des arguments optionnels manquants
- la fonction C a tous les arguments optionnels spécifiés

  donc une seule fonction C générée par fonction ILP

$\Longrightarrow$ nécessite de connaître à chaque site d'appel la fonction appelée !
c'est une information statique

Arbre syntaxique normalisé : interfaces

IASTCfunctionDefinitionOpt.java

```
package com.paracamplus.ilp2.partial_16.compiler.interfaces;

public interface IASTCfunctionDefinitionOpt extends IASTCfunctionDefinition
{
    Map<IASTvariable, IASTconstant> getOptVariables();
    Map<IASTvariable, String> getOptNames();
}
```

IASTCglobalInvocationOpt.java

```
package com.paracamplus.ilp2.partial_16.compiler.interfaces;

public interface IASTCglobalInvocationOpt extends IASTCglobalInvocation
{
    Map<String, IASTexpression> getOptArguments();
}
```

Une définition de fonctions stocke :

- la valeur par défaut de chaque argument optionnel ;
- le nom original de l'argument (la normalisation peut changer le nom des variables)

L'appel de fonction utilise un dictionnaire des arguments optionnels, indexés par leur nom (permet une correspondance facile avec les définitions de fonctions).

Arbre syntaxique normalisé : classes (1/2)

ASTCfunctionDefinitionOpt.java

```
package com.paracamplus.ilp2.partial_16.compiler;

public class ASTCfunctionDefinitionOpt
extends ASTCfunctionDefinition implements IASTCfunctionDefinitionOpt
{
    private Map<IASTvariable,IASTconstant> optVariables;
    private Map<IASTvariable,String> optName;

    public ASTCfunctionDefinitionOpt(IASTvariable functionVariable,
                                     IASTvariable[] variables,
                                     Map<IASTvariable,IASTconstant> optVariables,
                                     Map<IASTvariable,String> optName,
                                     IASTexpression body) {
        super(functionVariable, variables, body);
        this.optVariables = optVariables;
        this.optName = optName;
    }

    @Override public Map<IASTvariable, IASTconstant> getOptVariables()
    { return optVariables; }

    @Override public Map<IASTvariable, String> getOptNames()
    { return optName; }
}
```

Simple conteneur.

Arbre syntaxique normalisé : classes (2/2)

ASTCglobalInvocationOpt.java

```
package com.paracamplus.ilp2.partial_16.compiler;

public class ASTCglobalInvocationOpt
extends ASTCglobalInvocation implements IASTCglobalInvocationOpt, IASTCvisitable
{
    private Map<String,IASTexpression> optArguments;

    public ASTCglobalInvocationOpt(IASTexpression function,
                                   IASTexpression[] arguments,
                                   Map<String,IASTexpression> optArguments) {
        super(function, arguments);
        this.optArguments = optArguments;
    }

    @Override public Map<String, IASTexpression> getOptArguments()
    { return optArguments; }

    @Override public <Result, Data, Anomaly extends Throwable> Result
    accept(IASTCvisitor<Result, Data, Anomaly> visitor, Data data) throws Anomaly
    { return visitor.visit(this, data); }

    @Override public <Result, Data, Anomaly extends Throwable> Result
    accept(com.paracamplus.ilp1.compiler.interfaces.IASTCvisitor<ldots>, Data data)
    ...
}
```

Conteneur et visiteur.

Arbre syntaxique normalisé : interfaces visiteur

IASTCvisitable.java

```
package com.paracamplus.ilp2.partial_16.compiler.interfaces;

public interface IASTCvisitable
extends com.paracamplus.ilp2.compiler.interfaces.IASTCvisitable
{
    <Result, Data, Anomaly extends Throwable>
    Result accept(IASTCvisitor<Result, Data, Anomaly> visitor,
                  Data data) throws Anomaly;
}
```

IASTCvisitor.java

```
package com.paracamplus.ilp2.partial_16.compiler.interfaces;

public interface IASTCvisitor<Result, Data, Anomaly extends Throwable>
extends com.paracamplus.ilp2.interfaces.IASTvisitor<Result, Data, Anomaly>
{
    Result visit(IASTCglobalInvocationOpt iast, Data data) throws Anomaly;
}
```

Arbre syntaxique normalisé : fabrique (1/2)

INormalizationFactory.java

```
package com.paracamplus.ilp2.partial_16.compiler.normalizer;

public interface INormalizationFactory
extends com.paracamplus.ilp2.compiler.normalizer.INormalizationFactory {

    IASTCfunctionDefinitionOpt newFunctionDefinitionOpt(
        IASTvariable functionVariable,
        IASTvariable[] variables,
        Map<IASTvariable,IASTconstant> optVariables,
        Map<IASTvariable,String> optName,
        IASTexpression body);

    IASTCglobalInvocationOpt newGlobalInvocationOpt(
        IASTexpression function,
        IASTexpression[] arguments,
        Map<String,IASTexpression> optArguments);
}
```

Arbre syntaxique normalisé : fabrique (2/2)

NormalizationFactory.java

```
package com.paracamplus.ilp2.partial_16.compiler.normalizer;

public class NormalizationFactory
extends com.paracamplus.ilp2.compiler.normalizer.NormalizationFactory
implements INormalizationFactory {

   @Override
    public IASTCfunctionDefinitionOpt newFunctionDefinitionOpt
        (IASTvariable functionVariable, IASTvariable[] variables,
         Map<IASTvariable, IASTconstant> optVariables,
         Map<IASTvariable,String> optName, IASTexpression body)
    {
        return new ASTCfunctionDefinitionOpt(functionVariable, variables,
                                             optVariables, optName, body);
    }

    @Override
    public IASTCglobalInvocationOpt newGlobalInvocationOpt
        (IASTexpression function, IASTexpression[] arguments,
         Map<String, IASTexpression> optArguments)
    {
         return new ASTCglobalInvocationOpt(function, arguments, optArguments);
    }
}
```

Normalisation (1/4)

Normalizer.java (début)

```
package com.paracamplus.ilp2.partial_16.compiler.normalizer;

public class Normalizer
extends com.paracamplus.ilp2.compiler.normalizer.Normalizer
implements
  IASTvisitor<IASTexpression, INormalizationEnvironment, CompilationException>
{
    public Normalizer(INormalizationFactory factory)
    { super(factory); }

    INormalizationFactory getFactory()
    { return (INormalizationFactory)factory; }
```

Visiteur étendu aux AST du partiel.

Paramétré par une fabrique étendue (cast nécessaire, fait par `getFactory`)

Normalisation (2/4)

Normalizer.java (suite)

```
public IASTCprogram transform(IASTprogram program) throws CompilationException
{
    INormalizationEnvironment env = NormalizationEnvironment.EMPTY;
    IASTfunctionDefinition[] functions = program.getFunctionDefinitions();
    IASTCfunctionDefinition[] funs =
            new IASTCfunctionDefinition[functions.length];
    for ( IASTfunctionDefinition function : functions ) {
        IASTCglobalFunctionVariable gfv =
           getFactory().newGlobalFunctionVariable(function.getName());
        env = env.extend(gfv, gfv);
    }
    for ( int i=0 ; i<functions.length ; i++ ) {
        IASTfunctionDefinition function = functions[i];
        IASTCfunctionDefinition newfunction =
                (function instanceof IASTfunctionDefinitionOpt) ?
                visit((IASTfunctionDefinitionOpt)function, env) :
                visit(function, env);
        funs[i] = newfunction;
    }
    IASTexpression body = program.getBody();
    IASTexpression newbody = body.accept(this, env);
    return getFactory().newProgram(funs, newbody);
}
```

Point d'entrée : copie de la version ILP2, avec dispatch vers le visiteur `IASTfunctionDefinitionOpt` par un test `instanceof`.

Normalisation (3/4)

Normalizer.java (suite)

```
public IASTCfunctionDefinition visit(
        IASTfunctionDefinitionOpt iast,
        INormalizationEnvironment env) throws CompilationException
{
    /* omis: newvariables = ... de ILP 2 */

    Map<IASTvariable,IASTconstant> opt = iast.getOptVariables();
    Map<IASTvariable,IASTconstant> newopt = new HashMap<>();
    Map<IASTvariable,String> optname = new HashMap<>();
    for (IASTvariable variable : opt.keySet()) {
        IASTconstant cst = opt.get(variable);
        IASTvariable newvariable = factory.newLocalVariable(variable.getName());
        newenv = newenv.extend(variable, newvariable);
        newopt.put(newvariable, cst);
        optname.put(newvariable, variable.getName());
    }
    IASTexpression newbody = iast.getBody().accept(this, newenv);
    IASTvariable functionVariable =
        getFactory().newGlobalFunctionVariable(functionName);
    return getFactory().newFunctionDefinitionOpt(
        functionVariable, newvariables, newopt, optname, newbody);
 }
```

Normalisation des noms des arguments optionnels (`IASTClocalVariable`) dans les déclarations des fonctions. On se souvient de la valeur par défaut (`newOpt`) et du nom de l'argument avant normalisation (`optName`).

Normalisation (4/4)

Normalizer.java (fin)

```
public IASTexpression visit(IASTinvocationOpt iast,
                            INormalizationEnvironment env)
throws CompilationException {
    /* début omis: similaire à ILP 2 */

    Map<String,IASTexpression> newopt = new HashMap<>();
    for (IASTvariable variable : opt.keySet()) {
        IASTexpression expr = opt.get(variable);
        newopt.put(variable.getName(), expr.accept(this, env));
    }

    if ( funexpr instanceof IASTCglobalVariable ) {
        IASTCglobalVariable f = (IASTCglobalVariable) funexpr;
        return getFactory().newGlobalInvocationOpt(f, args, newopt);
    } else {
        String msg = "Optional arguments not supported for computed invocations";
        throw new CompilationException(msg);
    }
}
```

- Normalisation des expressions passées en arguments optionnels.
- Seuls les appels de fonctions globales sont supportés !
  pas de `IASTCcomputedInvocation` généré

Collecte des variables globales

GlobalVariableCollector.java

```
package com.paracamplus.ilp2.partial_16.compiler;

public class GlobalVariableCollector
extends com.paracamplus.ilp2.compiler.GlobalVariableCollector
implements
  IASTCvisitor<Set<IASTCglobalVariable>, Set<IASTCglobalVariable>, CompilationException>
{
    @Override public Set<IASTCglobalVariable> visit
    (IASTCglobalInvocationOpt iast, Set<IASTCglobalVariable> data)
    throws CompilationException
    {
        result = iast.getFunction().accept(this, result);
        for ( IASTexpression arg : iast.getArguments() )
            result = arg.accept(this, result);
        Map<String,IASTexpression> opts = iast.getOptArguments();
        for ( String s : opts.keySet() )
            result = opts.get(s).accept(this, result);
        return result;
    }
}
```

Collecte des variables globales utilisées par les arguments optionnels lors des appels.

Collecte des variables libres (1/2)

FreeVariableCollector.java (début)

```
package com.paracamplus.ilp2.partial_16.compiler;

public class FreeVariableCollector
extends com.paracamplus.ilp2.compiler.FreeVariableCollector
implements IASTCvisitor<Void, Set<IASTClocalVariable>, CompilationException>
{
    public FreeVariableCollector(IASTCprogram program)
    { super(program); }

    @Override public Void visit
      (IASTCglobalInvocationOpt iast, Set<IASTClocalVariable> variables)
    throws CompilationException {
        iast.getFunction().accept(this, variables);
        for ( IASTexpression expression : iast.getArguments() )
            expression.accept(this, variables);
        Map<String,IASTexpression> opts = iast.getOptArguments();
        for ( String s : opts.keySet() )
            opts.get(s).accept(this, variables);
        return null;
    }
```

Collecte des variables libres des arguments optionnels lors des appels.

Collecte des variables libres (2/2)

FreeVariableCollector.java (fin)

```
    public Void visit(IASTfunctionDefinition fd,
            Set<IASTClocalVariable> variables) throws CompilationException {
          if (fd instanceof IASTCfunctionDefinitionOpt)
                return visit((IASTCfunctionDefinitionOpt)fd, variables);
            else return super.visit(fd,variables);
    }

    public Void visit(IASTCfunctionDefinitionOpt fd,
            Set<IASTClocalVariable> variables) throws CompilationException {
        Set<IASTClocalVariable> newvars = new HashSet<>();
        fd.getBody().accept(this, newvars);
        IASTvariable[] vars = fd.getVariables();
        newvars.removeAll(Arrays.asList(vars));

        for (IASTvariable v : fd.getOptVariables().keySet())
            newvars.remove(v);

        /* fin identique à ILP 2 */
    }
}
```

Dans une définition de fonction :

- réorientation vers le visiteur `IASTCfunctionDefinitionOpt` par test `instanceof`
- les arguments formels optionnels ne sont pas libres !

Rappel : 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

Exemple de schéma de compilation : invocation

Exemple : compilation de l'appel :

```
invocation = fonction(argument1, argument2, ...)
```

$$\overset{\rightarrow \texttt{d}}{\texttt{invocation}}$$

$$\texttt{d}\ \texttt{fonctionCorrespondante(}\ \overset{\rightarrow}{\texttt{argument1}}\texttt{,}\ \overset{\rightarrow}{\texttt{argument2}}\texttt{,}\ \ldots\ \texttt{)}$$

- un appel ILP est directement un appel de fonction C
- $\overset{\rightarrow}{\texttt{argumentX}}$ indique que, avant de générer l'appel, il faut générer le code d'évaluation de l'argument `argumentX` dans une temporaire
- $\overset{\rightarrow}{\texttt{argumentX}}$ est ensuite remplacé par la temporaire dans le code C
- le contexte `d` peut avoir la forme `return`, `temp =` ou `(void)`

Schéma de compilation pour les arguments optionnels

appel : `fonction(arg1,...,argN, :a1 opt1, ..., :aM, optM)`

avec N arguments obligatoires et M arguments optionnels spécifiés

```
ILP_Object tmp1, ..., tmpK;
→tmp1=
expr1
...
→tmpK=
exprK
d fonctionCorrespondante(  arg1, ..., argN,
                           tmp1, ..., tmpK   )
```

où

- `fonction` a le dictionnaire d'arguments optionnels :
  `name1` $\mapsto$ `def1`, ..., `nameK` $\mapsto$ `defK`
  avec K arguments optionnels
- `exprX` vaut `optY` si `nameX`=`aY` pour un certain `Y`
- `exprX` vaut `defX` si `nameX` n'est égal à aucun `aY`

Déterminer `exprX` doit se faire statiquement, à la génération de l'appel.

Exemple de code C généré

source

```
function toto1 (a,b &keys c 3, d 4)
( print(a); print(b); print(c); print(d); newline(); );

toto1(1,2,:d 4+1);
```

C généré (simplifié)

```
/* Global prototypes */
ILP_Object ilp__toto1(ILP_Closure, ILP_Object, ILP_Object, ILP_Object, ILP_Object);

/* Global functions */
ILP_Object ilp__toto1(ILP_Closure ilp_useless, ILP_Object a1,
                      ILP_Object b2, ILP_Object c4, ILP_Object d3)
{ ... }

ILP_Object ilp_program() {
  ILP_Object ilptmp6 = ILP_Integer2ILP(1);
  ILP_Object ilptmp7 = ILP_Integer2ILP(2);
  ILP_Object ilptmp8;
  {
    ILP_Object ilptmp9 = ILP_Integer2ILP(4);
    ILP_Object ilptmp10 = ILP_Integer2ILP(1);
    ilptmp8 = ILP_Plus(ilptmp9, ilptmp10);
  }
  ILP_Object ilptmp11 = ILP_Integer2ILP(3);
  return ilp__toto1(NULL, ilptmp6, ilptmp7, ilptmp8, ilptmp11);
}
```

Note sur l'efficacité

Dans notre interprète comme dans le compilateur
les valeurs par défaut sont réévaluées à chaque appel :

- évite l'évaluation si la valeur n'est jamais utilisée
  que se passe-t-il si l'évaluation de la valeur par défaut provoque une erreur ?
- évaluation à chaque appel qui ne spécifie pas de valeur optionnelle
  $\Longrightarrow$ calculs inutiles, coût élevé

Solutions possibles : évaluer les arguments optionnels

- une fois pour toutes au début du programme
  dans une variable globale
- lors de la première utilisation
  dans un cache global
  (évaluation par nécessité)

Compilation alternative

Idée : reporter la résolution des arguments sur la fonction appelée

- lors de l'appel : construction d'un dictionnaire
- le dictionnaire est passé comme (unique) argument supplémentaire
  cf. la clôture utilisée en ILP 3
- en début de fonction, l'appelé explore le dictionnaire
  pour remplacer les valeurs par défaut par les valeurs spécifiées

Avantages et inconvénients :

- surcoût dû à la création et l'exploration dynamique du dictionnaire
- ne nécessite pas d'information statique sur la fonction appelée
  $\Longrightarrow$ utilisable en ILP 3 avec des `IASTCcomputedInvocation`