Correction du partiel

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

Antoine Miné

Année 2019–2020

Cours 10 bis
10 décembre 2019

Sujet : cache de résultat de fonction

Chaque fonction est enrichie d'un cache
qui permet de se souvenir d'une valeur de retour.

- `freeze(f,a1,...,aN)`
  - appelle `f(a1,...,aN)`
  - stocke le résultat retourné par `f`
    (et écrase le dernier résultat stocké dans `f`)
  - le cache des autres fonctions n'est pas changé
- `frozen(f)`
  - n'appelle pas `f` (pas d'effet de bord)
  - retourne immédiatement la valeur trouvée dans le cache de `f`
    ou signale une erreur si le cache est vide
    (si `freeze` n'a jamais été appelé sur `f`)
- `f(a1,...,aN)`
  - appel normal à `f`
  - n'utilise pas et ne modifie pas le cache de `f`

Exemple

exemple ILP

```
function test(x) (
  print("test "); print(x); print(" ");
  x
);

print (test (3));           newline();
print (freeze (test, 5));   newline();
print (frozen (test));      newline();
print (freeze (test, 6));   newline();
print (frozen (test));      newline();
print (test (7));           newline();
print (frozen (test));      newline();
```

résultat

```
test 3 3
test 5 5
5
test 6 6
6
test 7 7
6
```

Aspects statiques, aspects dynamiques

Syntaxe :

- `freeze(f,a1,...,aN)`
- `frozen(f)`

Les `a1`, ..., `aN` sont des expressions.

Deux choix possibles pour `f` :

1. une chaîne représentant une fonction $\Rightarrow$ résolution statique
2. une expression qui s'évalue en une fonction $\Rightarrow$ résolution dynamique

$\Longrightarrow$ nous optons pour le 2ème choix

Dans les deux cas :

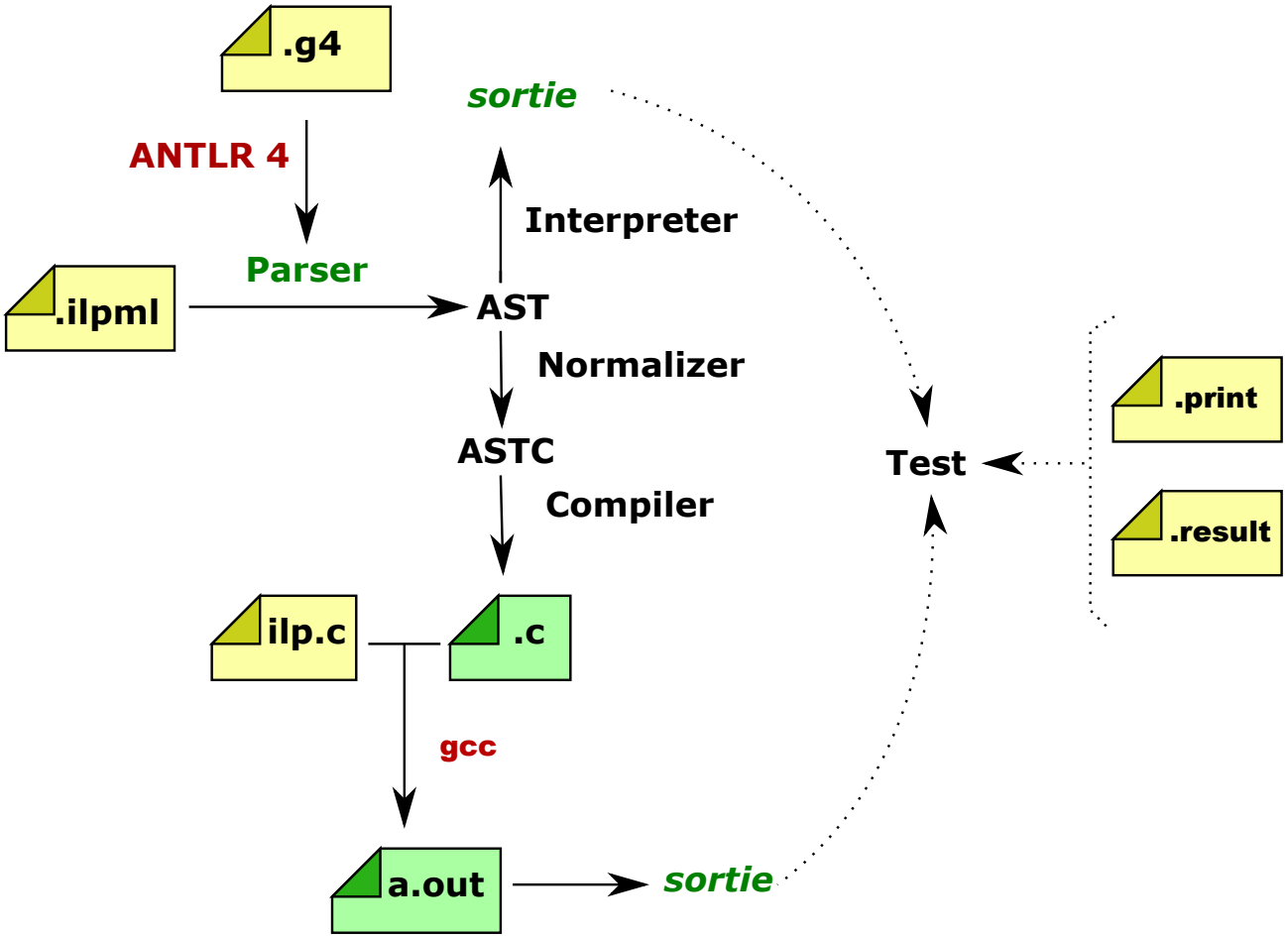
- nombre d'arguments : statique (mais varie d'un appel à `freeze` à l'autre)
- valeur des arguments : dynamique
- valeur du cache : dynamique

Plan

- Rappels : définition d'une extension.
- Stratégie d'implantation pour cette extension.
- Questions 1–3 : Grammaire, AST, analyse syntaxique.
- Question 4 : Interprète.
- Question 5 : Compilateur, bibliothèque d'exécution.

Rappels

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.ilp2partiel...` ;
- 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.

Stratégie d'implantation

Gestion du cache

Problèmes :

- où stocker le cache ?
- comment retrouver le cache pour l'utiliser ou le mettre à jour ?
  (la cible d'un appel de fonction n'est pas connue statiquement)

$\Longrightarrow$ nous stockons le cache dans la structure représentant une fonction lors de l'exécution (pas dans l'AST définissant la fonction).

- interprète : classe `Function`

  L'interprète associe à chaque fonction déclarée un objet `Function` dans l'environnement global.

- compilateur : structure `ILP_Closure`

  Le compilateur génère une variable C globale de type `ILP_Closure` pour chaque fonction globale.

  La variable est utilisée dès ILP2 pour les appels indirects de fonction, et est exploitée à nouveau en ILP3 pour les fonctions de première classe.

Ajout d'un champ qui indique la valeur du cache ou `null` (Java) ou `NULL` (C) si `freeze` n'a jamais été appelée.

Travail à faire

- AST :
  - ajout d'un nœud AST pour `freeze`
  - ajout d'un nœud AST pour `frozen`

    alternativement : ajout d'une primitive pour `frozen`

    (ce n'est pas possible pour `freeze` car le nombre d'arguments est variable)
  - enrichir la fabrique `IASTfactory` et le visiteur `IASTvisitor`
- grammaire :
  - ajout de règles pour `freeze` et `frozen`
- interprète :
  - ajout d'un attribut `cache` à `Function`, avec *getter* et *setter*
  - extension du visiteur d'interprète pour `freeze` et `frozen`
- bibliothèque d'exécution C :
  - ajout d'un champ `cache` à `ILP_Closure`
- compilateur :
  - version ASTC des nœuds AST ajoutés, extension de la normalisation
  - génération de l'initialisation du champ `cache` (à `NULL`)
  - génération de code pour `freeze` et `frozen`

Question 1–3 : Grammaire et AST

Grammaire ANTLR4

ILPgrammarPartiel.g4

```
grammar ILPMLgrammarPartiel;

@header { package antlr4; }

...

expr returns [com.paracamplus.ilp1.interfaces.IASTexpression node]
    : ...
    | 'freeze' '(' fun=expr (',' args+=expr)* ')' # Freeze
    | 'frozen' '(' fun=expr ')' # Frozen
    ;
```

- ajout de deux règles dans `expr`
- attention à l'utilisation de `+=` et `*`

Interfaces d'AST (1/2)

IASTfreeze.java

```
package com.paracamplus.ilp2.ilp2partiel.interfaces;

import com.paracamplus.ilp1.interfaces.IASTexpression;
import com.paracamplus.ilp1.interfaces.IASTinvocation;

public interface IASTfreeze extends IASTinvocation {
}
```

`freeze(f,a1,...,aN)` ressemble à un appel de fonction
$\Longrightarrow$ nous réutilisons `IASTinvocation`, pour hériter de :

- `IASTexpression getFunction()`
- `IASTexpression[] getArguments()`

Aucun ajout vis à vis de `IASTinvocation`,
mais on peut distinguer `IASTfreeze` de `IASTinvocation` avec `instanceof`.

Interfaces d'AST (2/2)

IASTfrozen.java

```
package com.paracamplus.ilp2.ilp2partiel.interfaces;

import com.paracamplus.ilp1.interfaces.IASTexpression;

public interface IASTfrozen extends IASTexpression {
   IASTexpression getFunction();
}
```

`frozen(f)` a un seul attribut : la fonction `f`

Interface de visiteur

IASTvisitor.java (partiel)

```
package com.paracamplus.ilp2.ilp2partiel.interfaces;

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

- ajout du visiteur pour les deux nouveaux nœuds AST
- attention aux interfaces et classes définies avec le même nom mais dans des *packages* différents

Classes d'AST (1/2)

ASTfreeze.java

```
package com.paracamplus.ilp2.ilp2partiel.ast;
import com.paracamplus.ilp2.ilp2partiel.interfaces.IASTvisitor;
import ...

public class ASTfreeze extends ASTinvocation implements IASTfreeze {

    public ASTfreeze(IASTexpression func, IASTexpression args[]) {
        super(func,args);
    }

    @Override public <Result, Data, Anomaly extends Throwable>
    Result accept(com.paracamplus.ilp1.interfaces.IASTvisitor<Result, Data, Anomaly>
                  visitor, Data data) throws Anomaly
    {
        return ((IASTvisitor<Result, Data, Anomaly>) visitor).visit(this, data);
    }
}
```

- le constructeur délègue à `ASTinvocation`
- ne pas oublier de redéfinir `accept`
  en utilisant la bonne interface de visiteur !

Classes d'AST (2/2)

ASTfrozen.java

```
package com.paracamplus.ilp2.ilp2partiel.ast;
import com.paracamplus.ilp2.ilp2partiel.interfaces.IASTvisitor;
import ...

public class ASTfrozen extends ASTexpression implements IASTfrozen {

    private final IASTexpression function;

    public ASTfrozen(IASTexpression function) {
        this.function = function;
    }

    @Override public IASTexpression getFunction() {
        return function;
    }

    @Override public <Result, Data, Anomaly extends Throwable>
    Result accept(com.paracamplus.ilp1.interfaces.IASTvisitor<Result, Data, Anomaly>
                  visitor, Data data) throws Anomaly
    {
        return ((IASTvisitor<Result, Data, Anomaly>) visitor).visit(this, data);
    }
}
```

- attribut `function` fixé à la construction, avec un *getter*

Fabrique d'AST

IASTfactory.java (partiel)

```
package com.paracamplus.ilp2.ilp2partiel.interfaces;
...
public interface IASTfactory extends com.paracamplus.ilp2.interfaces.IASTfactory
{
    IASTfreeze newFreeze(IASTexpression function, IASTexpression args[]);
    IASTfrozen newFrozen(IASTexpression function);
}
```

ASTfactory.java (partiel)

```
package com.paracamplus.ilp2.ilp2partiel.ast;
...
public class ASTfactory
extends com.paracamplus.ilp2.ast.ASTfactory implements IASTfactory
{
    @Override
    public IASTfreeze newFreeze(IASTexpression function, IASTexpression[] args) {
        return new ASTfreeze(function, args);
    }

    @Override
    public IASTfrozen newFrozen(IASTexpression function) {
        return new ASTfrozen(function);
   }
}
```

Ajout de la création de nœuds `ASTfreeze` et `ASTfrozen`.

Listener ANTLR

ILPMLListener.java (partiel)

```
package com.paracamplus.ilp2.ilp2partiel.parser;
...
public class ILPMLListener implements ILPMLgrammarPartielListener
{
    protected IASTfactory factory;

    ...

    @Override
    public void exitFreeze(FreezeContext ctx) {
        ctx.node =
            factory.newFreeze(ctx.fun.node, toExpressions(ctx.args));
    }

    @Override
    public void exitFrozen(FrozenContext ctx) {
        ctx.node = factory.newFrozen(ctx.fun.node);
    }
}
```

- copie du *listener* ILP2
- utilisation de la fabrique `factory` pour créer les nœuds

Lancement de l'analyse syntaxique ANTLR 4

ILPMLParser.java (partiel)

```
package com.paracamplus.ilp2.ilp2partiel.parser;
import antlr4.ILPMLgrammarPartielLexer;
import antlr4.ILPMLgrammarPartielParser;
import com.paracamplus.ilp2.ilp2partiel.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());
      ILPMLgrammarPartielLexer lexer = new ILPMLgrammarPartielLexer(in);
      CommonTokenStream tokens = new CommonTokenStream(lexer);
      ILPMLgrammarPartielParser parser = new ILPMLgrammarPartielParser(tokens);
      ILPMLgrammarPartielParser.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 cache : interface

IFunction.java (partiel)

```
public interface IFunction
extends com.paracamplus.ilp1.interpreter.interfaces.IFunction
{
      public Object getCache();
      public void   setCache(Object obj);
}
```

- le cache contient une valeur ILP ou `null` ⇒ type `Object`
- cette version d'`IFunction` remplacera totalement celle d'ILP1 dans notre interprète.

Fonctions avec cache : implantation

Function.java (partiel)

```
public class Function
extends com.paracamplus.ilp1.interpreter.Function {

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

    private Object cache; // = null

    public Object getCache() {
        return cache;
    }

    public void setCache(Object obj) {
         cache = obj;
    }
}
```

Interprète : création des fonctions avec cache

Interpreter.java (partiel)

```
package com.paracamplus.ilp2.ilp2partiel.interpreter;

public class Interpreter extends com.paracamplus.ilp2.interpreter.Interpreter
{
    @Override
    public Invocable visit(IASTfunctionDefinition iast, ILexicalEnvironment lexenv)
    throws EvaluationException
    {
        Invocable fun = new Function(iast.getVariables(),
                                     iast.getBody(),
                                     new EmptyLexicalEnvironment());
        return fun;
    }
    ...
```

A priori identique au code de l'interprète d'ILP2,
mais utilise en réalité notre nouvelle classe `Function`.

$\Longrightarrow$ il est nécessaire de redéfinir `visit` pour `IASTfunctionDefinition`.

Interprète : `freeze`

Interpreter.java (partiel, suite)

```
 ...
@Override
public Object visit(IASTfreeze iast, ILexicalEnvironment lexenv) throws ... {
    Object r = visit((IASTinvocation)iast, lexenv);
    Function function = (Function) iast.getFunction().accept(this, lexenv);
    function.setCache(r);
    return r;
}
...
```

- délégation au visiteur de `IASTinvocation` pour l'appel
- l'expression retournée par `iast.getFunction()` doit être évaluée pour connaître la fonction

  $\Longrightarrow$ inconvénient : cette expression est donc évaluée deux fois...

  (on peut mieux faire, en recopiant le code de gestion de `IASTinvocation`)
- conversion de la fonction retournée en `Function` avec cache
- mise à jour du cache avec `setCache`

Interprète : `frozen`

Interpreter.java (partiel, suite)

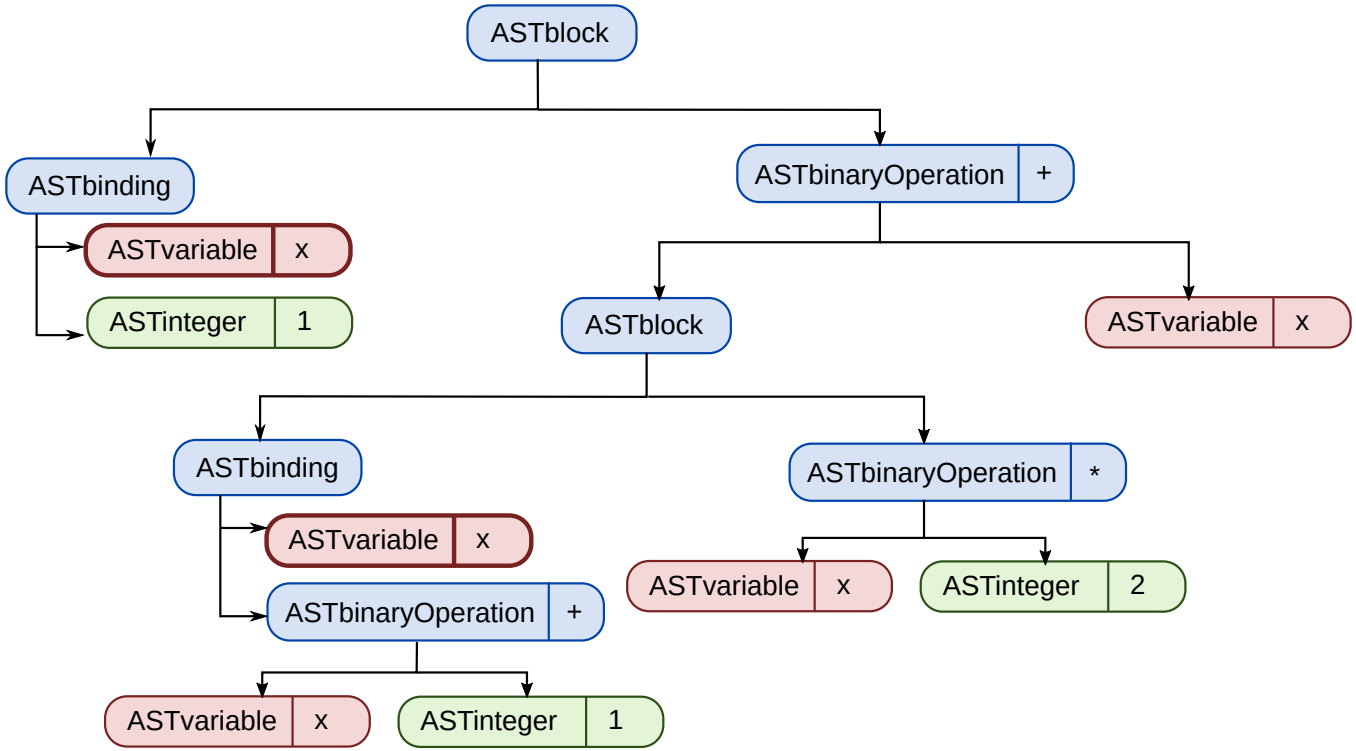
```
...
 @Override
 public Object visit(IASTfrozen iast, ILexicalEnvironment lexenv) throws ... {
     Function function = (Function) iast.getFunction().accept(this, lexenv);
     if ( function instanceof Function ) {
         Function f = (Function) function;
         Object r = f.getCache();
         if (r == null) {
             throw new EvaluationException("freeze not called");
         }
         return r;
     }
     else {
         throw new EvaluationException("not a function");
     }
 }
```

- l'expression retournée par `iast.getFunction()` doit encore être évalué pour connaître la fonction
- vérification de la présence du cache
  et retour de la valeur dans le cache avec `getCache`

Question 5 : Compilateur

Rappels : ASTC normalisé (1/4)

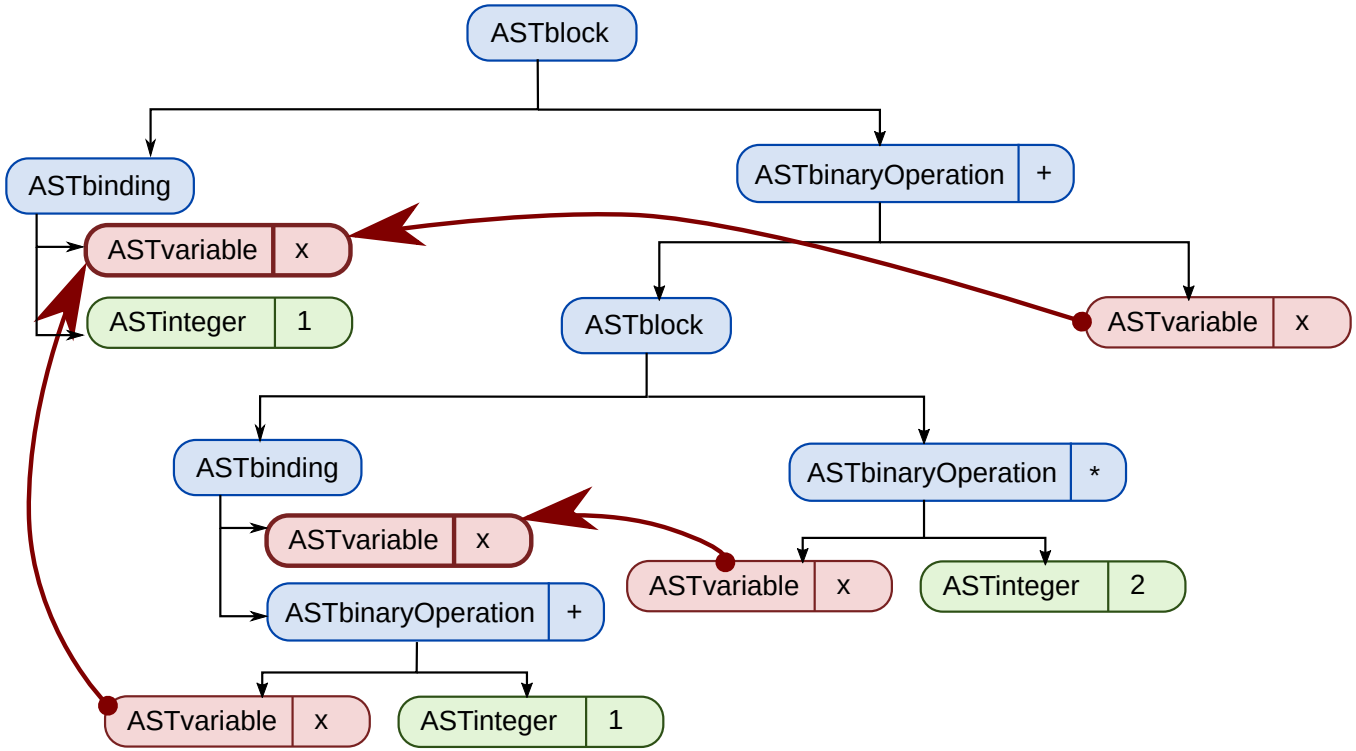
```
let x = 1 in
  (let x = x + 1 in 2 * x) + x
```



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`.

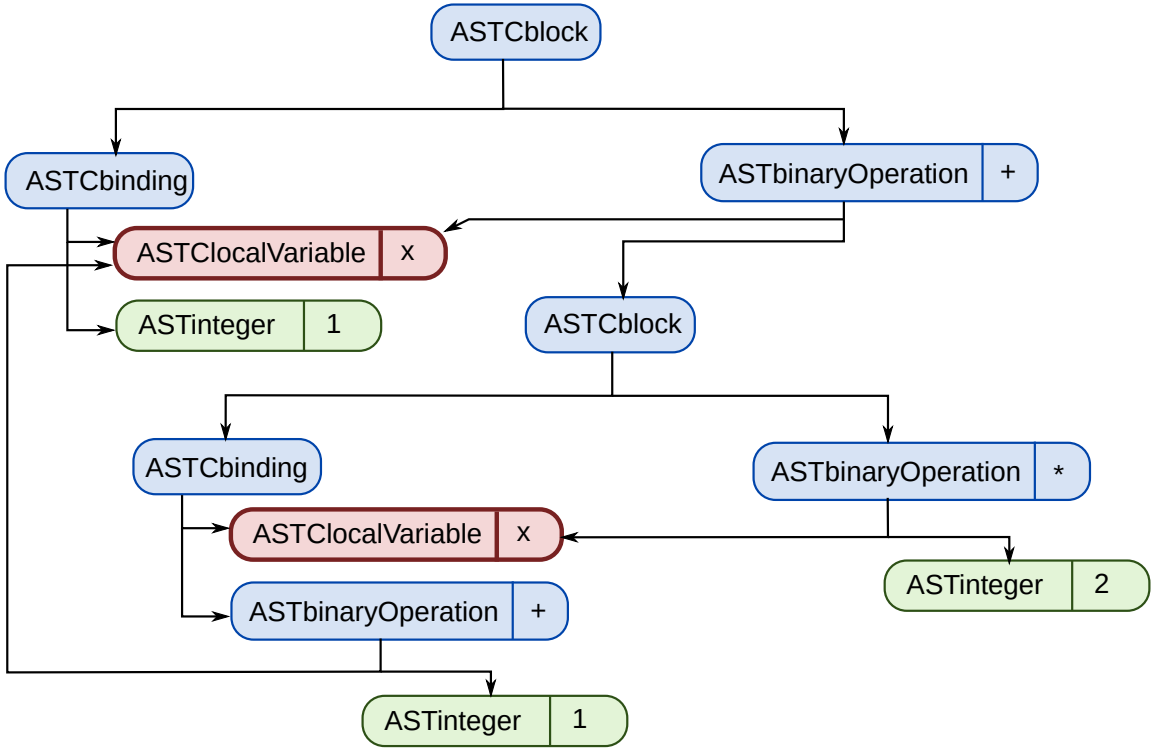
- distinction du type d'appel :
  - direct, par nom : `ASTCglobalInvocation` ;
  - indirect, par variable : `ASTCcomputedInvocation`.

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 ou des appels de fonction ;
- mettre à jour la normalisation ;
- mettre à jour la collecte des variables globales et des variables libres

Dans notre extension :

- `ASTfreeze` ressemble à un appel de fonction `ASTinvocation` $\Longrightarrow$ création d'un nœud `ASTCfreeze` similaire à `ASTCcomputedInvocation`

  `ASTCglobalInvocation` n'est qu'une optimisation pour un cas courant où l'expression est réduite à un nom de fonction globale
  $\Longrightarrow$ nous ne faisons pas cette distinction pour `freeze` et utilisons le nœud le plus général

- `ASTfrozen` ne contient pas directement de variable ou d'appel $\Longrightarrow$ pas de nœud ASTC nécessaire

  Il faudra néanmoins appeler récursivement les visiteurs sur tous les attributs de `ASTfrozen` !

Extension de l'ASTC

IASTCfreeze.java (partiel)

```
package com.paracamplus.ilp2.ilp2partiel.compiler.interfaces;
...
public interface IASTCfreeze extends IASTfreeze {
}
```

ASTCfreeze.java (partiel)

```
package com.paracamplus.ilp2.ilp2partiel.compiler.ast;
...
public class ASTCfreeze extends ASTfreeze implements IASTCfreeze {

    public ASTCfreeze(IASTexpression function, IASTexpression[] arguments) {
        super(function, arguments);
    }
}
```

`ASTCfreeze` ne se distingue de `ASTCcomputedInvocation` uniquement par son typage.

Fabrique étendue d'ASTC

INormalizationFactory.java (partiel)

```
package com.paracamplus.ilp2.ilp2partiel.compiler.normalizer;
...
public interface INormalizationFactory
extends com.paracamplus.ilp2.compiler.normalizer.INormalizationFactory
{
    IASTCfreeze newFreeze(IASTexpression function, IASTexpression[] arguments);
    IASTfrozen  newFrozen(IASTexpression function);
}
```

NormalizationFactory.java (partiel)

```
package com.paracamplus.ilp2.ilp2partiel.compiler.normalizer;
...
public class NormalizationFactory
extends com.paracamplus.ilp2.compiler.normalizer.NormalizationFactory
implements INormalizationFactory
{
    @Override public IASTCfreeze newFreeze(IASTexpression function,
                                           IASTexpression[] arguments) {
        return new ASTCfreeze(function, arguments);
    }

    @Override public IASTCfrozen newFrozen(IASTexpression function) {
        return new ASTfrozen(function);
    }
}
```

Normalisation de `freeze` et `frozen`

Normalizer.java (partiel)

```
package com.paracamplus.ilp2.ilp2partiel.compiler.normalizer;
...
public class Normalizer extends com.paracamplus.ilp2.compiler.normalizer.Normalizer
{
    @Override public IASTexpression visit(IASTfreeze iast,
                                          INormalizationEnvironment env)
    throws CompilationException {
        IASTexpression funexpr = iast.getFunction().accept(this, env);
        IASTexpression[] arguments = iast.getArguments();
        IASTexpression[] args = new IASTexpression[arguments.length];
        for (int i = 0 ; i < arguments.length ; i++) {
            IASTexpression argument = arguments[i];
            IASTexpression arg = argument.accept(this, env);
            args[i] = arg;
        }
        return factory.newFreeze(funexpr, args);
    }

    @Override public IASTexpression visit(IASTfrozen iast,
                                          INormalizationEnvironment env)
    throws CompilationException {
        IASTexpression funexpr = iast.getFunction().accept(this, env);
        return factory.newFrozen(funexpr);
    }
}
```

Visiteur d'ASTC

IASTvisitor.java (partiel)

```
package com.paracamplus.ilp2.ilp2partiel.compiler.interfaces;

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

Rappels : collecte des variables locales et globales

Le compilateur utilise plusieurs passes de visiteurs 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 `freeze` et `frozen`

FreeVariableCollector.java (partiel)

```
public class FreeVariableCollector
extends com.paracamplus.ilp2.compiler.FreeVariableCollector
implements IASTCvisitor<Void, Set<IASTClocalVariable>, CompilationException> {

     @Override
     public Void visit(IASTCfreeze iast, Set<IASTClocalVariable> variables)
     throws CompilationException {
        return visit((IASTinvocation) iast, variables);
     }

     @Override
     public Void visit(IASTfrozen iast, Set<IASTClocalVariable> variables)
     throws CompilationException {
         return iast.getFunction().accept(this, variables);
     }

}
```

- `freeze` délègue à `IASTinvocation` (dont il dérive)
- `frozen` visite récursivement sa sous-expression
- `GlobalVariableCollector` est similaire...

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.

Extension de ILP_Closure (ilp.h)

ilp.h (partiel)

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

- ajout d'un champ `cache`
- contenant une valeur ILP : `ILP_Object*`, éventuellement `null`
- ajout avant le tableau `closed_variables` de taille arbitraire
  pour assurer que le champ a une position constante dans la structure
- pas d'ajout de fonction nécessaire à la bibliothèque d'exécution...

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 (ILP2)

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

$$\texttt{ILP\_Object ilp\_\_}\mathit{name}\texttt{(ILP\_Closure ilp\_useless, ILP\_Object }\mathit{arg}1, \ldots, \texttt{ILP\_Object }\mathit{arg}N\texttt{)}$$

$$\{$$

$$\longrightarrow \underset{\mathit{expr}}{(\mathit{return})}$$

$$\}$$

Déclaration de prototype et clôture (ILP2)

```
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}}}
};
```

Extension de la compilation des fonctions

Déclaration de la clôture (partiel)

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

Mise à jour de la génération de la clôture des fonctions globales pour tenir compte du champ `cache` ajouté, initialisé à `NULL`.

Rappels : schéma de compilation des appels généraux

Cas IASTCcomputedInvocation

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

```
{
    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 de `freeze`

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

```
{
    ILP_Object tmpF;
    ILP_Object tmp1;
    ...
    ILP_Object tmpN;
    ILP_Object tmpR;
    ⟶(tmpF=) expr
    ⟶(tmp1=) arg1
    ...
    ⟶(tmpN=) argN
    tmpR = ILP_invoke(tmpF, tmp1, ..., tmp2);
    tmpF->_content.asClosure.cache = tmpR;
    d tmpR;
}
```

Stockage du résultat de l'appel dans un nouveau temporaire *tmpR*, copié dans le cache et retourné.

Schéma de compilation de `frozen`

$$\frac{\longrightarrow^{d} \texttt{frozen}(expr)}{\quad}$$

```
{
    ILP_Object tmpF;
    ⟶(tmpF=) expr
    if (! ILP_IsA(tmpF, Closure)) {
        ILP_domain_error("Not a closure");
    }
    else if (! tmpF->_content.asClosure.cache) {
        ILP_domain_error("Freeze absent");
    }
    else {
        d   tmpF->_content.asClosure.cache;
    }
}
```