UFR 919 Informatique – Master Informatique

Spécialité STL – UE MI016 – ILP

TME3 — Extension de la bibliothèque d'exécution

Christian Queinnec

Objectif : Étendre la bibliothèque d'exécution d'ILP1 avec des fonctions et un nouveau type de données : le vecteur.

Buts

— Ajouter une primitive à l'interprète
— Ajouter une primitive au compilateur
— Ajouter une primitive à la bibliothèque d'exécution (en C)
— Ajouter un nouveau type de données à la bibliothèque d'exécution (en C)
— Écrire un nouveau patron C

Vous aurez à suivre le mémento en `http://www-master.ufr-info-p6.jussieu.fr/site-annuel-courant/Ext/queinnec/ILP/refcard.pdf`. Vos fichiers Java seront donc dans le paquetage `fr.upmc.ilp.tme3`. Vos fichiers C seront dans le répertoire C.

1 Une nouvelle primitive : sinus

En ILP1, il n'est pas possible de calculer directement le sinus d'un nombre. On souhaite dans cet exercice ajouter une primitive, qui sera nommée `sinus` en ILP, pour pouvoir effectuer ce calcul. On essaiera d'adopter une approche générale qui permettra d'ajouter simplement par la suite d'autres fonctions si on le souhaite.

1. Quelles sont les grandes étapes des modifications à apporter ?
2. Implanter ces modifications.

Remarque : vous pouvez vous inspirer de l'implantation des primitives `print` ou `newline` qui sont présentes dans ILP1.

On fera attention à la comparaison des valeurs de retour flottantes dans l'interprétation et la compilation. En effet, le formatage en Java et en C peut être différent [1]

Comme pour les précédents travaux, les extensions ne doivent pas modifier le code existant mais l'étendre.

1. On pourra utiliser la classe `DecimalFormat` en Java pour obtenir des flottants dans le même format qu'en C.

***Pour enseignant*: Correction**

— Un exemple de programme ILP utilisant la primitive `sin` :

```
<?xml version='1.0' encoding='utf-8' ?>
<!--
(comment "appel de sin(pi/2)")
 -->
<programme1>
  <invocationPrimitive fonction='sin'>
    <operationBinaire operateur='/'>
      <operandeGauche>
        <variable nom="pi"/>
      </operandeGauche>
      <operandeDroit>
        <entier valeur='2'/>
      </operandeDroit>
    </operationBinaire>
  </invocationPrimitive>
</programme1>
```

Il faut faire attention. Par ailleurs, pour les tests, le flottant 1.0 est affiché 1 en C.
Voir la méthode `interpret` de `Process`.
Créer une classe `Trigo` selon le modèle de `PrintStuff`

```
package fr.upmc.ilp.ilp1.tme3;

import java.math.BigInteger;

import fr.upmc.ilp.ilp1.interfaces.IASTvariable;
import fr.upmc.ilp.ilp1.runtime.AbstractInvokableImpl;
import fr.upmc.ilp.ilp1.runtime.EvaluationException;
import fr.upmc.ilp.ilp1.runtime.ILexicalEnvironment;

public class Trigo {

 public Trigo () {
 }

 /** étendre un environnement lexical pour y installer la primitive
     sin(x). */

 public ILexicalEnvironment
 extendWithPrintPrimitives (final ILexicalEnvironment lexenv) {
   final ILexicalEnvironment lexenv1 = lexenv.extend(
       new IASTvariable() {
         public String getName () {
           return "sin";
         }
       }, new SinusPrimitive());
   return lexenv1;
 }

 /** Cette classe implante la fonction sin(x) */

 protected class SinusPrimitive
 extends AbstractInvokableImpl {
  public SinusPrimitive () {}
  // La fonction sin() est unaire:
  @Override
  public Object invoke (Object value)
```

```
  throws EvaluationException{
    if (value instanceof Double)
     return Math.sin((Double) value);
    else if (value instanceof BigInteger)
     return Math.sin(((BigInteger) value).doubleValue());
    else throw new EvaluationException("Invalid argument: number expected");
  }
 }
}
```

— Pour l'interface `ICgenEnvironment`, on ajoute les méthodes rajoutant les primitives et les constantes.

```
package fr.upmc.ilp.ilp1.tme3;

import fr.upmc.ilp.ilp1.cgen.ICgenLexicalEnvironment;

public interface ICgenEnvironment
extends fr.upmc.ilp.ilp1.cgen.ICgenEnvironment {

    /** L'enrichisseur d'environnement lexical avec la primitive
    sinus. */

    ICgenLexicalEnvironment
      extendWithTrigoPrimitives (ICgenLexicalEnvironment lexenv);

    /** L'enrichisseur d'environnement lexical avec les primitives
    hashtable. */

    ICgenLexicalEnvironment
      extendWithHashPrimitives (ICgenLexicalEnvironment lexenv);
}
```

— Pour la classe `CgenEnvironment` :

```
package fr.upmc.ilp.ilp1.tme3;

import fr.upmc.ilp.ilp1.cgen.ICgenLexicalEnvironment;
import fr.upmc.ilp.ilp1.interfaces.IASTvariable;

public class CgenEnvironment
extends fr.upmc.ilp.ilp1.cgen.CgenEnvironment
implements ICgenEnvironment{

  private static IASTvariable createVariable (final String name) {
    return new IASTvariable () {
      public String getName () {
        return name;
      }
    };
  }

  public ICgenLexicalEnvironment extendWithTrigoPrimitives (
  ICgenLexicalEnvironment lexenv){
      final ICgenLexicalEnvironment lexenv2 =
        lexenv.extend(CgenEnvironment.createVariable("sin"),
        "ILP_sin");
      return lexenv2;
  }

```

```
    /** L'enrichisseur d'environnement lexical avec les primitives
    hashtable. */

  public ICgenLexicalEnvironment extendWithHashPrimitives (
  ICgenLexicalEnvironment lexenv){
      final ICgenLexicalEnvironment lexenv2 =
        lexenv.extend(CgenEnvironment.createVariable("makehash"),
        "ILP_makehash");
      final ICgenLexicalEnvironment lexenv3 =
        lexenv2.extend(CgenEnvironment.createVariable("hashput"),
        "ILP_hashput");
      final ICgenLexicalEnvironment lexenv4 =
        lexenv3.extend(CgenEnvironment.createVariable("hashget"),
        "ILP_hashget");
      return lexenv4;
  }
}
```

— Dans `ilp_sin.c`, voici les ajouts à apporter :

```
#include <math.h>

ILP_Object ILP_sin (ILP_Object o)
{
  switch (o->_kind) {
   case ILP_INTEGER_KIND: {
     return ILP_make_float(sin(o->_content.asInteger));
   }
   case ILP_FLOAT_KIND: {
     return ILP_make_float(sin(o->_content.asFloat));
   }
   default: {
     return ILP_domain_error("Not a number", o);
   }
  }
}
```

Mettre la ligne suivante dans `ilp_sin.h` :

```
 extern ILP_Object ILP_sin (ILP_Object o);
```

Prendre en compte les nouveaux fichiers C (les `.c` et `.h`) lors de l'invocation de `compileThenRun.sh` (cf. `CgeneratorTest`).

— Pour la classe `Process` (concerne également l'exercice suivant) :

```
import fr.upmc.ilp.ilp1.cgen.CgenLexicalEnvironment.Empty;
import fr.upmc.ilp.ilp1.cgen.Cgenerator;
import fr.upmc.ilp.ilp1.cgen.ICgenLexicalEnvironment;
import fr.upmc.ilp.ilp1.runtime.CommonPlus;
import fr.upmc.ilp.ilp1.runtime.ConstantsStuff;
import fr.upmc.ilp.ilp1.runtime.EmptyLexicalEnvironment;
import fr.upmc.ilp.ilp1.runtime.ICommon;
import fr.upmc.ilp.ilp1.runtime.ILexicalEnvironment;
import fr.upmc.ilp.ilp1.runtime.PrintStuff;

public class Process
extends fr.upmc.ilp.ilp1.Process {

  public Process(){
    super();
  }
```

```
17
18    /** Interprétation */
19    @Override
20    public void interpret() {
21      try {
22        final ICommon intcommon = new CommonPlus();
23        ILexicalEnvironment intlexenv = EmptyLexicalEnvironment.create();
24        final PrintStuff intps = new PrintStuff();
25        intlexenv = intps.extendWithPrintPrimitives(intlexenv);
26        final ConstantsStuff intcs = new ConstantsStuff();
27        intlexenv = intcs.extendWithPredefinedConstants(intlexenv);
28        final Trigo intt = new Trigo();
29        intlexenv = intt.extendWithPrintPrimitives(intlexenv);
30        final HashStuff inths = new HashStuff();
31        intlexenv = inths.extendWithPrintPrimitives(intlexenv);
32        this.result = this.east.eval(intlexenv, intcommon);
33        this.printing = intps.getPrintedOutput().trim();
34        this.interpreted = true;
35      } catch (Throwable e) {
36        this.interpretationFailure = e;
37      }
38      notifyInterpretationToListeners();
39    }
40
41    /** Compilation vers C. */
42    @Override
43    public void compile() {
44      try {
45        final ICgenEnvironment common = new CgenEnvironment();
46        final Cgenerator compiler = new Cgenerator(common);
47        ICgenLexicalEnvironment lexenv = Empty.create();
48        lexenv = common.extendWithPrintPrimitives(lexenv);
49        lexenv = common.extendWithTrigoPrimitives(lexenv);
50        lexenv = common.extendWithHashPrimitives(lexenv);
51        this.ccode = compiler.compile(east, lexenv, "return");
52        this.compiled = true;
53      } catch (Throwable e) {
54        System.out.println(e);
55        this.compilationFailure = e;
56      }
57      notifyCompilationToListeners();
58    }
59  }
```

— Enfin, la classe ProcessTest :

```
1  import java.util.Collection;
2
3  import org.junit.Before;
4  import org.junit.runner.RunWith;
5
6  import fr.upmc.ilp.tool.AbstractProcessTest;
7  import fr.upmc.ilp.tool.File;
8  import fr.upmc.ilp.tool.Parameterized;
9  import fr.upmc.ilp.tool.Parameterized.Parameters;
10
11 @RunWith(Parameterized.class)
12 public class ProcessTest
13 extends fr.upmc.ilp.ilp1.ProcessTest {
14
```

```
15      @Before
16      public void setUp () {
17        this.process = new Process();
18      }
19
20      @Parameters
21      public static Collection<File[]> data() {
22        AbstractProcessTest.staticSetUp("u\\d+-1tme3bis");
23          // Pour un (ou plusieurs) test(s) en particulier:
24          // AbstractProcessTest.staticSetUp("u01-1");
25          return AbstractProcessTest.data();
26      }
27
28      /** Le constructeur du test sur un fichier. */
29      public ProcessTest (final File file) {
30        super(file);
31      }
32  }
```

— Pour tester la fonction sinus, on peut par exemple vérifier que la fonction renvoie les bonnes valeurs pour quelques appels bien connus (*eg.*, sin(0)=0, sin(pi/2)=1, sin(pi)=-1), plus un appel moins trivial (*eg.*, sin(pi/4)>0.7 && sin(pi/4)<0.71).

2 Un nouveau type : les vecteurs

On souhaite maintenant pouvoir gérer des données dans des vecteurs. Il faut donc ajouter ce nouveau type dans ILP. À ce type, on ajoutera trois nouvelles primitives qui auront pour noms en ILP : `make-vector`, `vector-length`, `vector-get` (attention aux tirets qui ne sont pas des blancs soulignés) dont voici les signatures plus précises :

```
1 make-vector(taille, objet)
2 vector-length(vecteur)
3 vector-get(vecteur, index)
```

La primitive `make-vector` crée un vecteur ayant pour taille son premier argument, chaque cellule de ce vecteur sera initialisée avec le second argument.

La primitive `vector-length` renvoie la taille du vecteur qu'elle reçoit en argument.

La primitive `vector-get` renvoie le index-ième objet du vecteur.

Comme ILP1 est sans effet de bord, on n'ajoutera pas `vector-set` qui permettrait d'écrire dans le vecteur.

1. Quelles sont les grandes étapes des modifications à apporter ? Suivre les indications de la question précédente.
2. Implanter ces modifications.

   Indication : Vous regarderez les directives de compilation conditionnelles d'`ilp.h` pour imposer votre nouvelle définition de `ILP_Object`.

Supplément : Mise au point en C

Pour ce TME, il peut être bon de savoir un peu mettre au point un programme C (compilé avec l'option `-g`). L'outil de base est `gdb` que l'on peut aussi utiliser au travers d'Emacs, de `xdgb` ou encore `ddd`. Eclipse peut probablement aussi être mis à contribution après incorporation du greffon CDT (C Development Tool).