Examen réparti de mi-semestre d'ILP

Jacques Malenfant

26 octobre 2009

Directives

- Le contrôle dure 2h30.
- Tous les documents sont autorisés, et notament ceux du cours. Vous disposez à ce titre d'un ordinateur avec accès http aux documents de cours et avec Eclipse pour vous permettre de rédiger les parties code de l'examen.
- À l'exception des clés USB en lecture seule, tous les appareils électroniques sont **prohibés** (y compris les téléphones portables, les assistants numériques personnels et les agendas électroniques).
- Le barème total est fixé à 20.
- Votre travail pour l'examen sera fait dans l'espace de travail (*workspace*) Eclipse ILP dans lequel vous créerez le paquetage `fr.upmc.ilp.ilp2partiel2009` c'est-à-dire le répertoire `$HOME/workspace/ILP/Java/src/fr/upmc/ilp/ilp2partiel2009` qui contiendra les programmes (pas nécessairement compilables) que vous aurez écrits ainsi que le fichier textuel `$HOME/workspace/ILP/Java/src/fr/upmc/ilp/ilp2partiel2009/partiel2009.txt` qui contiendra le reste. Des consignes plus précises seront énoncées pour chaque question. Plus généralement, tout fichier ou répertoire, placé sous `$HOME/workspace/ILP/` et dont le nom contiendra `partiel2009` sera ramassé pour constituer votre copie. Rien d'autre ne sera ramassé !
- Vos fichiers textuels seront codés en UTF-8, ils seront formés de lignes ne dépassant pas 78 caractères pour en faciliter la lecture.
- L'examen sera corrigé à la main, il est donc absolument inutile de s'acharner sur un problème de compilation ou sur des méthodes à contenu informatif faible. Il est beaucoup plus important de rendre aisé, voire plaisant, le travail du correcteur et de lui indiquer, par tout moyen à votre convenance, de manière claire, compréhensible et terminologiquement précise, comment vous surmontez cette épreuve.
- Le langage à étendre est ILP2.

Le problème : générateurs de nombre aléatoires

La génération de nombres aléatoires est utile dans de nombreuses applications, en simulation tout d'abord, dans les jeux, mais aussi à chaque fois qu'on souhaite accomplir une action quelconque parmi en ensemble d'actions possibles sans avoir de séquences d'actions trop répétitives ou prédictibles.

Par exemple, Java offre pour ce faire la classe `java.util.Random` dont voici les principales méthodes :

```
public class Random {

  /** Creates a new random number generator. */
  public Random() ;
  /** Creates a new random number generator using a single long seed. */
  public Random(long seed) ;
```

```
  /** Returns the next pseudorandom, uniformly distributed float value
      between 0.0 and 1.0 from this random number generator's sequence. */
  public float nextFloat() ;

  /** Sets the seed of this random number generator using a single long seed. */
  public void setSeed(long seed) ;
}
```

Ces générateurs produisent des nombres réels dans l'intervalle [0, 1] (ici par la méthode `nextFloat`), après avoir été initialisé par un germe qui est un entier sur 32 bits (ici par la méthode `setSeed`). À partir d'un germe donné, un bon générateur peut produire un séquence de nombres pseudo-aléatoires selon une loi uniforme $\mathcal{U}[0,1]$ dont la période avant de se répéter peut atteindre 2^{32} voire plus.

Expressions à ajouter à ILP2

Vous devez ajouter deux nouvelles expressions à ILP2 :

- L'expression `createRNG with` *seed* crée un nouveau générateur de nombres pseudo-aléatoires et retourne un entier permettant ensuite d'identifier ce générateur.
- L'expression `generateRN from` *no* génère le prochain nombre pseudo-aléatoire pour le générateur *no* préalablement créé et initialisé par l'expression `createRNG with` appropriée.

Notez bien qu'il s'agit de deux *expressions* au sens syntaxique du terme, et non des fonctions primitives. Nous voulons en effet vous voir implanter de nouveaux traits syntaxiques.

Par exemple, le programme suivant imprime 10 nombres pseudo-aléatoires à partir de deux générateurs :

```
let rng1 = createRNG with 1147400000 + 83647 in
  let rng2 = createRNG with 1400000000 - 14783467 in
    let i = 0 in
      while (i < 10)
        // genere alternativement du generateur rng1 puis de rng2
        print generateRN from (i mod 2) * rng1 + ((i+1) mod 2) * rng2
        newline
        i := i + 1
```

L'exécution de ce programme donnerait quelque chose comme :

```
0,70024
0,16254
0,39696
0,1847
0,18899
0,66487
0,95865
0,61719
0,10129
0,29258
```

Étapes dans vos réponses

Vous devez implanter les deux expressions de bout en bout en suivant les étapes suivantes parmi celles données pour les extensions à ILP2 lors du TD-TME 5 (à chaque étape, il est demandé de mettre des explications dans le fichier `partiel2009.txt` (ou en commentaires des fichiers Java ou RelaxNG produits), explications qui seront prises en compte même si le code correspondant n'est pas fait ou ne fonctionne pas) :

Étape 1 : Choisir une syntaxe externe pour les deux expressions. (2 points)
Livraison : des exemples expliqués dans votre fichier partiel.txt.

Étape 2 : Créer une nouvelle grammaire pour cette extension. (2 points)
Livraison : un fichier `grammar2-partiel2009.rnc`.

Étape 4 : Créer les interfaces nécessaires. (1 point)
Livraison : vos interfaces `IAST2createRNGwith` et `IAST2generateRNfrom`.

Étape 5 : Créer les classes pour les nœuds d'AST des expressions avec leur méthodes `eval`. (4 points)
Livraison : vos classes `CEASTcreateRNGwith` et `CEASTgenerateRNfrom`.

Étape 6 : Ajouter la fabrique nécessaire. (2 points)
Livraison : votre interface `IAST2Factory` et votre classe `CEASTFactory`.

Étape 7 : Ajouter l'analyseur pour créer l'AST des programmes utilisant cette extension. (3 points)
Livraison : votre classe `CEASTParser` et les méthodes `parse` qui doivent apparaître dans les classes `CEASTcreateRNGwith` et `CEASTgenerateRNfrom`.

Étape 9 : Ajouter les méthodes `compileExpression` dans vos classes de nœuds d'AST. (4 points)
Livraison : elles devraient apparaître dans vos classes `CEASTcreateRNGwith` et `CEASTgenerateRNfrom`.

Étape 11 : Ajouter les classes `Process` et `ProcessTest` nécessaires. (2 points)
Livraison : les classes en question.

Note : Nous ne vous demandons pas de faire les extensions aux exécutifs Java pour l'interprète et C pour le générateur de code. La prochaine section vous donne plutôt ce que vous pourrez utiliser pour programmer l'interprétation et la compilation de vos expressions. Ces fichiers sont disponibles en `/Infos/lmd/2009/master/ue/ilp-2009oct/E0/`

Exécutifs Java et C

Pour étendre l'exécutif Java, vous pouvez considérer que la classe suivante, qui permet de créer et d'appeler les générateurs, existe :

```
package fr.upmc.ilp.ilp2partiel2009.runtime;

import java.util.Random;

import fr.upmc.ilp.ilp1.runtime.EvaluationException;

public class RandomNumberGenerators {

    public static int MAX_NGENERATORS = 100 ;
    protected Random[] generators ;
    protected int ngenerators ;

    public RandomNumberGenerators() {
        super();
        this.generators = new Random[MAX_NGENERATORS];
        this.ngenerators = 0 ;
    }

    public int createRNGwith(long seed) throws EvaluationException {
        if ( ngenerators == MAX_NGENERATORS - 1 ) {
            throw new EvaluationException("creating too many generators!") ;
        }
        this.generators[ngenerators] = new Random(seed) ;
        return ngenerators++ ;
    }

    public float generateRNfrom(int gnumber) throws EvaluationException {
```

```
        if ( gnumber < 0 || gnumber > ngenerators ) {
            throw new EvaluationException("wrong generator number!") ;
        }
        return this.generators[gnumber].nextFloat() ;
    }
}
```

Pour étendre l'exécutif C, voici le fichier `templateTest2Partiel2009.c` qui fait les ajouts nécessaires :

```
#include <stdio.h>
#include <stdlib.h>

#include "ilp.h"

char *ilpTemplate2_Id = "$Id: templateTest2Partiel2009.c $";

/** Ajouts pour la solution du partiel 2009 */

#define MAX_SEEDS 100

unsigned int ILP_seeds[MAX_SEEDS] ;
int ILP_number_of_seeds = 0 ;

ILP_Object
ILP_create_rng_with(ILP_Object seed)
{
  if (ILP_number_of_seeds == MAX_SEEDS - 1) {
    return ILP_die("Creating too many rng's!\n") ;
  }
  ILP_seeds[ILP_number_of_seeds++] = seed->_content.asInteger ;
  return ILP_make_integer(ILP_number_of_seeds - 1) ;
}

ILP_Object
ILP_generate_rn_from(ILP_Object seed_number)
{
  if ( seed_number->_content.asInteger < 0 ||
                    seed_number->_content.asInteger > ILP_number_of_seeds ) {
      ILP_die("wrong generator number!") ;
  }
  srand(ILP_seeds[seed_number->_content.asInteger]) ;
  ILP_seeds[seed_number->_content.asInteger] = rand() ;
  return ILP_make_float(((float)ILP_seeds[seed_number->_content.asInteger])/
                                                    ((float)2147483647)) ;
}

/* Ici l'on inclut le code C produit: */
#include FICHIER_C

int
main (int argc, char *argv[])
{
     ILP_print(ilp_program());
     ILP_newline();
     return EXIT_SUCCESS;
}
```

```
/* end of templateTest.c */
```