Contrôle partiel ILP
Revision: 1.4

Christian Queinnec

12 novembre 2004

Conditions générales

Cet examen est formé d'un exercice en plusieurs questions auxquelles vous pouvez répondre dans l'ordre qui vous plait. L'examen dure 2 heures. Tous les documents sont autorisés.

1 Variables globales

On souhaite ne pas restreindre l'ensemble des variables que l'on peut utiliser dans le corps des blocs locaux. Il sera ainsi possible d'utiliser les noms `print` et `newline` correspondant aux deux fonctions primitives prédéfinies d'ILP1, comme noms de variable locale.

Mais comment imprimer si de tels noms sont utilisés ? Certains langages de programmation ont introduit la possibilité de référencer, depuis n'importe où, n'importe quelle variable globale à l'aide d'une graphie appropriée. La graphie `global V` signifie, dans n'importe quel contexte lexical, que l'on souhaite accéder à la variable globale `V` même si l'environnement courant contient une variable locale nommée `V`. Ainsi, avec une syntaxe concrète Caml-ienne :

```
let x = 3 in
  let print = 2 + x in
      (global print)(print);
(* imprime 5 *)
```

L'expression `(global print)` a pour valeur la fonction primitive d'impression qui est immédiatement appliquée à la valeur de la variable locale nommée `print` qui vaut ici 5.

Le reste de ce problème examine les conséquences de cette décision sur les plans syntaxiques et sémantiques.

Question 1

Écrire un programme (en syntaxe concrète Caml-ienne) non erroné où l'expression `global print` apparaît en initialisation d'un bloc unaire.

Question 2

Décrivez la syntaxe XML que vous proposez pour la représentation de ces variables globales.

Étendre la grammaire (le schéma RelaxNG compact) d'ILP1 pour accepter cette nouvelle expression. Vous ne mentionnerez que ce qui est relatif à cette nouvelle expression en partant de la grammaire *grammar4.rnc* en annexe.

Question 3

Les variables globales pouvant être considérées comme des variables, on décide que la classe `ASTglobal` les représentant implantera l'interface `IASTvariable`.

Écrire donc une classe `ASTglobal.java` pour représenter les variables globales. On veillera à ce que la méthode `toXML()` engendre ce qui est approprié.

Question 4

Supposons (ce qui est souvent vrai) que l'accès à la valeur d'une variable globale soit bien plus efficace que l'accès à la valeur d'une variable locale. Il est alors intéressant de détecter les accès aux variables globales `V` pour les transformer en des `global V`. Ainsi le programme de gauche peut-il être réécrit en celui de droite :

```
let x = 3 in          let x = 3 in
  print(x);             (global print)(x);
```

Écrivez (en français ou en pseudo-code compréhensible) cette transformation prenant un programme et le transformant en un nouveau programme où toutes les variables non locales (ne faisant pas l'objet d'un bloc unaire) sont transformées en variables globales.

Question 5

Pour obtenir la valeur d'une variable globale il faut avoir une représentation de l'environnement global. Nous allons nous intéresser à deux représentations possibles. La première se fonde sur le fait que l'environnement lexical initial est en fait l'environnement global.

Écrire une classe `GlobalEnvironment` implantant l'interface suivante :

```
public interface IGlobalEnvironment extends ILexicalEnvironment {

  void bind (IASTvariable variable, Object value);

}
```

La méthode `bind` permet d'enregistrer la valeur d'une nouvelle variable globale. La classe `GlobalEnvironment` comportera un unique constructeur prenant un `ILexicalEnvironment` et créant un environnement global comportant, comme variables globales, les variables contenues dans l'environnement fourni en argument.

L'interprète sera lancé comme suit :

```
ICommon common = new CommonPlus();
ILexicalEnvironment lexenv = EmptyLexicalEnvironment.create();
PrintStuff ps = new PrintStuff();
lexenv = ps.extendWithPrintPrimitives(lexenv);
final IGlobalEnvironment globenv = new GlobalEnvironment(lexenv);
// print et newline sont dans globenv. On y ajoute
autreVariableGlobale:
IASTvariable avg = new ASTglobal("autreVariableGlobale");
globenv.bind(avg, autreValeur);
// et on utilise globenv comme un ILexicalEnvironment:
Object result = east.eval(globenv, common);
```

Indices : la classe `GlobalEnvironment` peut être une simple façade à `ILexicalEnvironment` ou contenir une table associative mettant en correspondance des `IASTvariable`s et des valeurs.

Question 6

Écrire une classe `EASTglobal` héritant d'`EAST` implantant l'interface `IASTvariable` dotée d'une méthode `eval` permettant de rendre la valeur de la variable globale.

Question 7

Puisque l'environnement global est global pourquoi ne pas le stocker dans `ICommon` où il sera plus aisément trouvable que dans l'environnement lexical ?

L'interprète sera maintenant lancé comme suit :

```
ICommon common = new CommonPlus();
ILexicalEnvironment emptyenv = EmptyLexicalEnvironment.create();
PrintStuff ps = new PrintStuff();
ILexical lexenv = ps.extendWithPrintPrimitives(emptyenv);
final IGlobalEnvironment globenv = new CommonEnvironment(lexenv, common);
// print et newline sont dans globenv. On y ajoute autreVariableGlobale:
IASTvariable avg = new ASTglobal("autreVariableGlobale");
globenv.bind(avg, autreValeur);
// et on utilise globenv comme un ICommon:
Object result = east.eval(emptyenv, globenv);
```

Écrire cette classe nommée `CommonEnvironment` ainsi que la nouvelle classe `EASTglobal` correspondante.

A Grammaire d'ILP4 — fichier *grammar4.rnc*

```
# ***********************************************************************
# ILP -- Implantation d'un langage de programmation.
# Copyright (C) 2004 <Christian.Queinnec@lip6.fr>
# $Id: grammar4.rnc,v 1.8 2004/08/23 16:14:55 queinnec Exp $
# GPL version>=2
# ***********************************************************************

# Quatrième version du langage étudié: ILP4 pour « Incongru Langage
# Poilant ». Il sera complété dans les cours qui suivent. La grande
# nouveauté est que (à la différence de C, de Java, de JavaScript,
# etc.) toute instruction est maintenant aussi une expression. Cela
# permet de s'affranchir des différences de syntaxe entre les deux
# types d'alternatives (if-else et ?:) mais autorise le bloc local (ce
# que n'autorise pas C ni JavaScript). Le grand avantage est que cela
# simplifie le code de compilation et permet de parler plus simplement
# d'intégration de fonctions (inlining).

start =
   programme4
 | programme3
 | programme2
 | programme1

# Un programme4 est composé de définitions de fonctions globales
# suivies d'expressions les mettant en œuvre.

programme4 = element programme4 {
   definitionEtExpressions
}
programme3 = element programme3 {
   definitionEtExpressions
}
programme2 = element programme2 {
   definitionEtExpressions
}
```

```
programme1 = element programme1 {
   expression +
}

definitionEtExpressions =
   definitionFonction *,
   expression +

# Définition d'une fonction avec son nom, ses variables (éventuellement
# aucune) et un corps qui est une séquence d'expressions.

definitionFonction = element definitionFonction {
   attribute nom     { xsd:Name },
   element variables { variable * },
   element corps     { expression + }
}

# Les expressions possibles:

expression =
   alternative
 | sequence
 | blocUnaire
 | blocLocal
 | boucle
 | try
 | affectation
 | invocation
 | constante
 | variable
 | operation
 | invocationPrimitive

# Le si-alors-sinon. L'alternant est facultatif.

alternative = element alternative {
   element condition   { expression },
   element consequence { expression + },
   element alternant   { expression + } ?
}

# La séquence qui permet de regrouper plusieurs expressions en une seule.
# Il est obligatoire qu'il y ait au moins une expression dans la séquence.

sequence = element sequence {
   expression +
}

# Le bloc local unaire. Il est conservé pour garder les tests associés.
# Mais on pourrait s'en passer au profit du blocLocal plus général.

blocUnaire = element blocUnaire {
   variable,
   element valeur { expression },
   element corps  { expression + }
}
```

```
# Un bloc local qui introduit un nombre quelconque (éventuellement nul)
# de variables locales associées à une valeur initiale (calculée avec
# une expression).

blocLocal = element blocLocal {
   element liaisons {
      element liaison {
         variable, expression
      } *
   },
   element corps { expression + }
}

# La boucle tant-que n'a de sens que parce que l'on dispose maintenant
# de l'affectation.

boucle = element boucle {
   element condition { expression },
   element corps     { expression + }
}

# L'affectation prend une variable en cible et une expression comme
# valeur. L'affectation est une expression.

affectation = element affectation {
   attribute nom  { xsd:Name },
   element valeur { expression }
}

# L'invocation d'une fonction définie.

invocation = element invocation {
   element fonction { expression },
   element arguments { expression * }
}


# Cette définition permet une clause catch ou une clause finally ou
# encore ces deux clauses à la fois.

try = element try {
   element corps { expression + },
   (  catch
    | finally
    | ( catch, finally )
   )
}

catch = element catch {
   attribute exception { xsd:Name },
   expression +
}

finally = element finally {
   expression +
```

```
}

# Une variable n'est caractérisée que par son nom. Les variables dont
# les noms comportent la séquence ilp ou ILP sont réservés et ne
# peuvent être utilisés par les programmeurs.

variable = element variable {
   attribute nom { xsd:string - ( xsd:string { pattern = "(ilp|ILP)" } ) },
   empty
}

# L'invocation d'une fonction primitive. Une fonction primitive est
# procurée par l'implantation et ne peut (usuellement) être définie
# par l'utilisateur. Les fonctions primitives sont, pour être
# utilisables, prédéfinies. Une fonction primitive n'est caractérisée
# que par son nom (éventuellement masquable).

invocationPrimitive = element invocationPrimitive {
   attribute fonction { xsd:string },
   expression *
}

# Les operations sont en fait des sortes d'invocations a des fonctions
# primitives sauf que ces fonctions sont implantées par le matériel
# par des instructions particulières. On ne distingue que les
# opérations unaires et binaires (les plus usuelles):

operation =
   operationUnaire
 | operationBinaire

operationUnaire = element operationUnaire {
   attribute operateur { "-" | "!" },
   element operande { expression }
}
operationBinaire = element operationBinaire {
   element operandeGauche { expression },
   attribute operateur {
      "+" | "-" | "*" | "/" | "%" |                    # arithmétiques
      "|" | "&" | "^" |                                # booléens
      "<" | "<=" | "==" | ">=" | ">" | "<>" | "!="      # comparaisons
   },
   element operandeDroit { expression }
}

# Les constantes sont les données qui peuvent apparaître dans les
# programmes sous forme textuelle (ou littérale comme l'on dit
# souvent). Ici l'on trouve toutes les constantes usuelles à part les
# caractères:

constante =
   element entier    {
      attribute valeur { xsd:integer },
      empty }
 | element flottant  {
    attribute valeur { xsd:float },
```

```
    empty }
 | element chaine    { text }
 | element booleen   {
    attribute valeur { "true" | "false" },
    empty }

# fin de grammar4.rnc
```