

Semaine 12 : Interprétation de Scheme—

Nous présentons maintenant un interpréteur pour Scheme—.

C'est utile pour deux raisons.

1. De nombreux systèmes informatiques incluent un petit langage, qui est souvent interprété.
2. L'interpréteur nous dit de manière précise comment les programmes Scheme— en particulier, et les programmes fonctionnels en général, sont évalués.

Conception de l'interpréteur

L'interpréteur prend en entrée une expression Scheme— (de type *Data*) qui peut être

- un nombre,
- une chaîne de caractères,
- un symbole, ou
- une liste d'expressions.

L'interpréteur retourne une autre expression *Data* en sortie, qui représente une valeur Scheme—.

Par exemple, s'il est appliqué à l'expression d'entrée

```
List('*', 2, 7)
```

l'interpréteur doit retourner l'expression de sortie

Mais maintenant, une question se pose :

Quand l'entrée de l'interpréteur est un symbole, tel que 'x', quelle doit-être la sortie ?

Cela dépend si le nom x est défini au point où le symbole est évalué, et si oui, à quelle valeur il est lié.

L'interpréteur a besoin de mémoriser les noms définis dans un *environnement*.

Environnements

Un environnement est une structure de données qui associe des valeurs Scheme— à des noms.

Les deux opérations fondamentales sur un environnement sont :

- *lookup* : étant donné un nom, renvoie la valeur associée à ce nom.
- *extend* : étant donné une liaison nom/valeur, étend l'environnement avec cette liaison en renvoyant un nouvel environnement.

Cela nous amène à la conception de classe suivante :

```
class Environment {  
    def lookup(n: String): Data = ...  
    def extend(name: String, v: Data): Environment = ...  
}
```

On a aussi besoin de définir une valeur *EmptyEnvironment*, qui représente l'environnement vide.

Différentes structures de données peuvent être utilisées pour implanter les environnements avec différents compromis de performance. (Exemples ?)

Nous utilisons ici une approche directe, sans structure de données auxiliaire.

```
abstract class Environment {  
  def lookup(n: String): Data  
  def extend(name: String, v: Data) = new Environment {  
    def lookup(n: String): Data =  
      if (n == name) v else Environment.this.lookup(n)  
  }  
}
```

```
val EmptyEnvironment = new Environment {  
  def lookup(n: String): Data = error("non défini : " + n)  
}
```

C'est court et simple, mais la méthode *lookup* prend un temps proportionnel au nombre de liaisons dans l'environnement.

Remarque : La construction *Environment.this* fait référence à l'objet courant englobant qui est une instance de la classe *Environment* (par opposition à l'objet créé par la classe anonyme englobante).

Fonctions prédéfinies

Il doit aussi exister un moyen d'interpréter les symboles “standards” tels que `*`.

On peut utiliser un **environnement initial** pour enregistrer les valeurs de tels symboles.

Mais quelle doit être la valeur de `*` ? Ce n'est ni un nombre, ni une chaîne de caractères, ni un symbole ou une liste !

On a besoin de créer un type de donnée qui représente les fonctions agissant sur des expressions Scheme— :

```
case class Lambda(f: List[Data] => Data)
```

Par exemple, l'opération `*` sera associée à la valeur :

```
Lambda { case List(arg1: Int, arg2: Int) => arg1 * arg2 }
```

Remarque : l'exemple montre la puissance du filtrage de motif — en une seule expression, on peut spécifier que

- la liste d'arguments doit avoir pour longueur deux, que
- les deux arguments doivent être des entiers,
- et on rend accessibles les éléments des deux arguments à travers les noms *arg1* et *arg2*.

Notez que l'expression **case** est une fonction (partielle), de type $List[Data] \Rightarrow Data$, qui filtre les arguments.

L'évaluateur

On présente maintenant l'évaluateur Scheme—.

Il prend en entrée la représentation interne d'une expression Scheme—, qui est soit

- un atome : symboles, nombres, chaînes de caractères, ou
- une combinaison, qui à son tour peut être une forme spéciale ou une application.

La fonction d'évaluation

```
def eval(x: Data, env: Environment): Data = x match {  
  case _ : String ⇒ x  
  case _ : Int ⇒ x  
  case Symbol(name) ⇒  
    env lookup name  
  case 'val :: Symbol(name) :: expr :: rest :: Nil ⇒  
    eval(rest, env.extend(name, eval(expr, env)))  
  case 'if :: cond :: thenpart :: elsepart :: Nil ⇒  
    if (eval(cond, env) != 0) eval(thenpart, env)  
    else eval(elsepart, env)  
  case 'quote :: y :: Nil ⇒ y  
  case 'lambda :: params :: body :: Nil ⇒  
    mkLambda(asList(params) map paramName, body, env)  
  case operator :: operands ⇒  
    apply(eval(operator, env), operands map (x ⇒ eval(x, env)))
```

Explications

- Si l'expression d'entrée x est un nombre, une chaîne de caractères ou une fonction, on retourne l'expression elle-même.
- Sinon, si l'expression est un symbole, on retourne le résultat de la recherche du symbole dans l'environnement courant.
- Sinon, si l'expression est une forme spéciale

`(val <name> <expr> <rest>)`

on évalue l'expression $\langle rest \rangle$ dans un environnement étendu par la liaison de $\langle name \rangle$ au résultat de l'évaluation de $\langle expr \rangle$.

- Sinon, si l'expression est une forme spéciale

`(if <cond> <thenpart> <elsepart>)`

on évalue $\langle cond \rangle$. Si le résultat est le nombre 0 , on continue avec l'évaluation de $\langle thenpart \rangle$; sinon avec l'évaluation de $\langle elsepart \rangle$.

- Sinon, si l'expression est une forme spéciale

`(quote <expr>)`

on retourne *expr* comme résultat de l'évaluation.

- Sinon, si l'expression est une forme spéciale

(`lambda <params> <body>`)

on crée une nouvelle fonction en utilisant l'opération *mkLambda* (voir ci-dessous).

- Enfin, si l'expression est une combinaison qui n'est aucune des formes spéciales ci-dessus, ce ne peut être qu'une application de fonction.

Dans ce cas, on évalue l'opérateur de l'application, ainsi que toutes ses opérandes, et on *applique* ensuite la valeur de l'opérateur aux valeurs des opérandes.

La fonction *apply* vérifie que l'opérateur est bien un nœud *Lambda* et applique ensuite l'opération associée à la liste d'arguments.

```
def apply(fn: Data, args: List[Data]): Data = fn match {  
  case Lambda(f)  $\Rightarrow$  f(args)  
  case _  $\Rightarrow$  error("application d'une non-fonction: " + fn + " à " + args)  
}
```

C'est tout, sauf que nous devons encore définir comment une expression *Lambda* est convertie en une fonction qui peut être appliquée.

Construction des fonctions

La conversion d'une expression Scheme— en une fonction Scala utilise la fonction auxiliaire *mkLambda*.

L'appel à *mkLambda(params, body, env)* renvoie un objet *Lambda* qui contient une fonction (Scala).

Cette fonction associe à une liste d'arguments, qui correspond à la liste des paramètres formels *params*, le résultat de l'évaluation de *body* dans *env*.

Voici comment ça marche :

```
def mkLambda(ps: List[String], body: Data, env: Environment) =  
  Lambda { args ⇒ eval(body, env.extendMulti(ps, args)) }
```

- Ainsi, une application de la fonction provoque l'évaluation de *body* dans un environnement étendu par les liaisons qui associent aux noms des paramètres formels les valeurs des arguments effectifs correspondants.
- Notez l'analogie avec le modèle par substitution, où les paramètres formels étaient *remplacés* dans *body* par les arguments effectifs *args*.
- On peut voir un environnement comme une substitution différée : plutôt que de faire le remplacement immédiatement, on le fait quand un symbole est recherché dans l'environnement.
- Il serait aussi possible d'écrire un interpréteur basé sur des substitutions plutôt que sur des environnements ; mais cet interpréteur serait alors plus compliqué et moins efficace.

Remarque : La fonction *asList* permet de promouvoir un objet quelconque en une liste. Elle est définie ainsi :

```
def asList(x: Data): List[Data] = x match {  
  case xs: List[_] ⇒ xs  
  case _ ⇒ error("liste mal formée : " + x)  
}
```

List[_] est un motif de type, où le paramètre de type de *List* n'est pas spécifié.

Il reste à définir la méthode *extendMulti* qui étend l'environnement par une liste de liaisons entre les paramètres formels et les valeurs des arguments.

Cette méthode est définie ainsi :

```
abstract class Environment {  
  // ...  
  def extendMulti(ps: List[String], vs: List[Data]): Environment =  
    (ps, vs) match {  
      case (List(), List()) =>  
        this  
      case (p :: ps1, arg :: args1) =>  
        extend(p, arg).extendMulti(ps1, args1)  
      case _ =>  
        error("nombre d'arguments incorrect")  
    }  
}
```

- La méthode vérifie que les deux listes, celle des noms des paramètres et celle des valeurs des arguments, ont la même longueur.
- Pour chaque couple nom/valeur, elle applique la méthode *extend*.

Environnements pour les lambdas

Notez que l'opération *mkLambda* conserve une référence vers l'environnement qui était l'environnement courant lorsque la fonction a été construite.

Elle utilise alors cet environnement dans une application.

Voici un exemple d'évolution des environnements durant une exécution du programme

```
def f(x: Int) = g(y ⇒ x + y)
def g(x: Int ⇒ Int) = x(2)
f(1)
```

(voir le tableau).

Portée dynamique dans le Lisp original

La première version de Lisp utilisait une pile globale pour les environnements.

A l'entrée d'une fonction, les liaisons pour les paramètres et les variables locales étaient empilées.

A la sortie d'une fonction, la pile était effacée.

Mais les fonctions d'ordre supérieur se comportent alors très étrangement !

Par exemple, l'évaluation de $f(1)$ dans le programme ci-dessus résulterait en une erreur à l'exécution, car la fonction anonyme $y \Rightarrow x + y$ accède à la dernière valeur de x sur la pile, qui est une fonction, pas une valeur entière.

Ce problème avec ce programme particulier pourrait être résolu en renommant les noms de paramètres pour les rendre uniques.

Par exemple, après renommage :

```
def f(x: Int) = g(y ⇒ x + y)
def g(z: Int ⇒ Int) = z(2)
f(1)
```

le programme retourne 3, comme attendu.

Mais le programme suivant montre qu'on ne peut pas toujours contourner le problème par le renommage.

```
def fact(n: Int, f: () ⇒ Int): Int =
  if (n == 0) f()
  else fact(n - 1, () ⇒ n * f())
```

Question : Quel effet a l'évaluation de `fact(7, () ⇒ 1)` avec l'environnement de pile du Lisp original ?

Un autre problème avec le schéma du Lisp original est qu'il est très dangereux de retourner des fonctions. Considérons :

def incrementer(x: Int) = y ⇒ y + x

Question : Quel effet a *incrementer(2)(3)*?

Le schéma d'implantation du Lisp original est appelé **portée dynamique**.

Cela signifie que l'association entre un identificateur et sa valeur est déterminée dynamiquement — elle dépend de la forme de l'environnement au moment où la valeur est référencée.

La portée dynamique offre des possibilités intéressantes, mais elle nuit sévèrement à l'utilisation des fonctions d'ordre supérieur.

Si bien que la plupart des versions modernes de Lisp telles que Scheme ont une portée statique, exactement comme Scala.

Les versions plus anciennes, elisp inclus, ont encore la portée dynamique.

Common Lisp, bien sûr, a les deux.

Prise en compte de la récursivité

Nous avons vu que le schéma d'environnement du Lisp original conduisait à des problèmes.

Mais le nôtre aussi, quand on ajoute la récursivité !

La récursivité est introduite en ajoutant à notre interpréteur une forme spéciale (`def <name> <expr> <rest>`).

Un `def` est comme un `val` mis à part qu'il définit une fonction `<name>` qui peut s'appeler elle-même récursivement.

La méthode pour traiter `val` dans l'interpréteur,

```
case 'val' :: Symbol(name) :: expr :: rest :: Nil =>  
  eval(rest, env.extend(name, eval(expr, env)))
```

ne marche pas pour `def`, pour deux raisons :

- Le corps `expr` est évalué trop tôt ; il ne devrait être évalué que lorsqu'on accède à `name`.
- `name` ne fait pas partie de l'environnement visible par `expr`, la récursivité est donc impossible.

On résout ces problèmes en ajoutant une nouvelle méthode *extendRec* qui étend un environnement avec une liaison contenant un calcul possiblement récursif.

Évaluation de `def`

Voici comment `def` peut être traité :

*case 'def' :: Symbol(name) :: expr :: rest :: Nil ⇒
eval(rest, env.extendRec(name, env1 ⇒ eval(expr, env1)))*

- Cela résout le problème de l'évaluation prématurée, car on étend maintenant l'environnement avec une fonction qui évalue *expr*, et non le résultat de l'évaluation de *expr*.
- De plus, on rend la récursivité possible en changeant la méthode *lookup* d'un environnement.

Nouveaux environnements

Voici la nouvelle définition des environnements.

```
abstract class Environment {  
  def lookup(n: String): Data  
  def extendRec(name: String, expr: Environment ⇒ Data) =  
    new Environment {  
      def lookup(n: String): Data =  
        if (n == name) expr(this) else Environment.this.lookup(n)  
    }  
  def extend(name: String, v: Data) = extendRec(name, env1 ⇒ v)  
}
```

Notez que *extendRec* est maintenant l'opération principale pour étendre un environnement ; la méthode *extend* est définie en termes de *extendRec*.

Notez aussi que *lookup* permet la récursivité en passant l'environnement courant à la fonction trouvée.

Récurtivité par auto-application

Cette technique met en évidence une connexion profonde en programmation : on peut modéliser la récursivité par l'auto-application.

En fait, toute récursivité est au bout du compte traitée par l'auto-application dans le lambda-calcul, la théorie sous-jacente à la programmation fonctionnelle.

Pour le mettre en évidence, sans plus d'explications, voici une version de la factorielle qui n'utilise ni la récursivité, ni les boucles !

```
(lambda (n)
  ((lambda (fact)
    (fact fact n))
   (lambda (ft k)
    (if (= k 1)
        1
        (* k (ft ft (- k 1)))))))
```

L'environnement global

On évalue les expressions Scheme— dans un environnement initial (global), qui contient les définitions pour les opérations et constantes couramment utilisées telles que `+`, `cons`, ou `nil`.

Voici une version utile minimale d'un tel environnement.

```
val globalEnv = EmptyEnvironment
  .extend("=", Lambda{
    case List(arg1, arg2) => if(arg1 == arg2) 1 else 0})
  .extend("+", Lambda{
    case List(arg1: Int, arg2: Int) => arg1 + arg2
    case List(arg1: String, arg2: String) => arg1 + arg2})
  .extend("-", Lambda{
    case List(arg1: Int, arg2: Int) => arg1 - arg2})
```

```
.extend("*", Lambda{
  case List(arg1: Int, arg2: Int) ⇒ arg1 * arg2})
.extend("/", Lambda{
  case List(arg1: Int, arg2: Int) ⇒ arg1 / arg2})
.extend("nil", Nil)
.extend("cons", Lambda{
  case List(arg1, arg2) ⇒ arg1 :: asList(arg2)})
.extend("car", Lambda{
  case List(x :: xs) ⇒ x})
.extend("cdr", Lambda{
  case List(x :: xs) ⇒ xs})
.extend("null?", Lambda{
  case List(Nil) ⇒ 1
  case _ ⇒ 0})
```

La fonction d'interprétation principale

Voici la fonction principale de l'interpréteur :

```
def evaluate(x: Data): Data = eval(x, globalEnv)
```

Elle évalue un programme Scheme— dans l'environnement global et retourne l'objet résultant.

Pour ajouter les formes spéciales dérivées vues la dernière fois comme **and**, **or**, ou **cond**, on ferait la modification suivante :

```
def evaluate(x: Data): Data = eval(normalize(x), globalEnv)
```

Pour faciliter la vie du programmeur qui tape des expressions Scheme— sur la ligne de commande, on inclut aussi une autre version de *evaluate* qui prend et renvoie des expressions Scheme— sous forme de chaînes de caractères.

```
def evaluate(s: String): String = lisp2string(evaluate(string2lisp(s)))
```

Utiliser Scheme— à partir de Scala

Voici un scénario d'utilisation de l'interpréteur Scala pour évaluer du Scheme— :

On définit tout d'abord une fonction Scheme— comme une chaîne de caractères :

```
> def factDef =  
    "def factorial (lambda (n) +  
    " (if (= n 0) +  
    " 1" +  
    " (* n (factorial (- n 1))))"
```

Ensuite, on peut appliquer cette fonction comme suit :

```
> evaluate("(" + factDef + "(factorial 4)")  
24
```

Exercice

Augmenter l'interpréteur Scheme— de telle manière qu'il ait sa propre boucle d'interprétation, qui accepte des définitions et des expressions individuelles.

Étant donnée une définition en entrée, l'interpréteur ajoute la liaison définie à l'environnement global.

Étant donnée une expression en entrée, il l'évalue et affiche le résultat.

Résumé

Nous avons vu comment les programmes fonctionnels sont évalués en écrivant un interpréteur pour un dialecte de Lisp.

La structure de donnée auxiliaire centrale était l'environnement, qui représente toutes les liaisons connues au moment du calcul.

Les environnements remplacent les substitutions présentes dans le modèle formel d'évaluation.