Construction d'une mini machine virtuelle avec un visiteur

Alain Plantec

2017-2018

1 Introduction

Le sujet de l'exercice consiste à mettre en oeuvre une machine virtuelle pour un langage spécifique permettant d'effectuer des calculs sur des entiers. Le langage met en oeuvre des expressions simples (addition, soustraction, division et multiplication), des variables, un opérateur d'affectation de la valeur d'une expression dans une variable et une procédure "println" qui affiche son argument sur une console.

Voici un exemple de programme. Après exécution par la machine virtuelle, le résultat affiché sur la console est "6". On remarque qu'il n'y a pas de déclaration de variable :

```
1 v1 := 4;
2 v2 := v1 + 1;
3 println(v2 + 1);
```

2 Cadre général

Comme le montre la figure 1, pour un programme à exécuter, le processus comprend deux étapes principales :

- étape d'analyse et de compilation : le programme source est analysé syntaxiquement et compilé ; en entrée, le programme est stocké dans un fichier. Ce fichier est lu et est analysé syntaxiquement. Le résultat de cette étape est la construction d'une représentation en mémoire du programme.
- étape de calcul : la machine virtuelle exécute le programme à partir de la représentation interne produite par la première étape.

Pour la première étape, on construit un analyseur. Comme il a été vu en cours de compilation, pour construire un tel analyseur, on peut utiliser un compilateur de compilateur (comme Yacc ou Cup). Un tel outils utilise la description formelle de la grammaire du langage décorée d'actions sémantiques. Ce sont ses actions sémantiques exécutées en cours d'analyse syntaxique qui construisent la représentation interne utilisée par la machine virtuelle. La production de la représentation est la compilation.

Dans cet exercice, nous nous concentrons sur la deuxième étape, la mise en oeuvre de la machine virtuelle pour effectuer les calculs et maintenir les variables. La suite de ce rapport présente la mise en oeuvre de la machine virtuelle à l'aide d'un visiteur.

3 Mise en oeuvre de la machine virtuelle

La machine virtuelle est composée des classes qui décrivent la représentation interne et du visiteur associé. Dans cette section nous présentons tout d'abord les classes de la représentation interne puis le visiteur.

3.1 La représentation interne

La représentation interne se compose des classes qui permettent le stockage des informations issues d'un programme source. Notre langage dispose d'expressions simples pour les calculs, de deux instructions que sont l'affectation et l'affichage et du concept de variable pour le stockage de valeurs entières. La figure 2

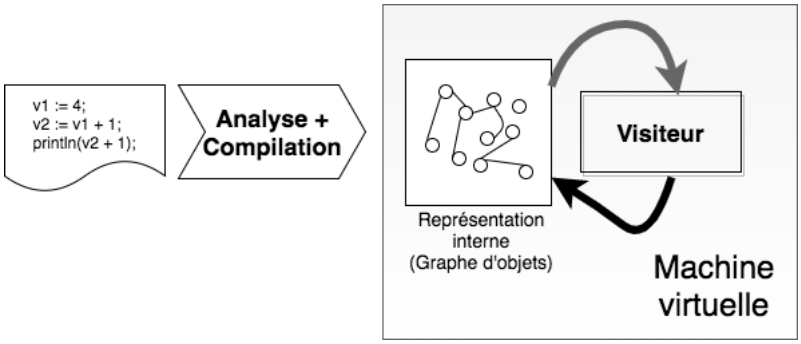

FIGURE 1 – Le processus d'interprétation : analyse du code, compilation et calcul par la machine virtuelle

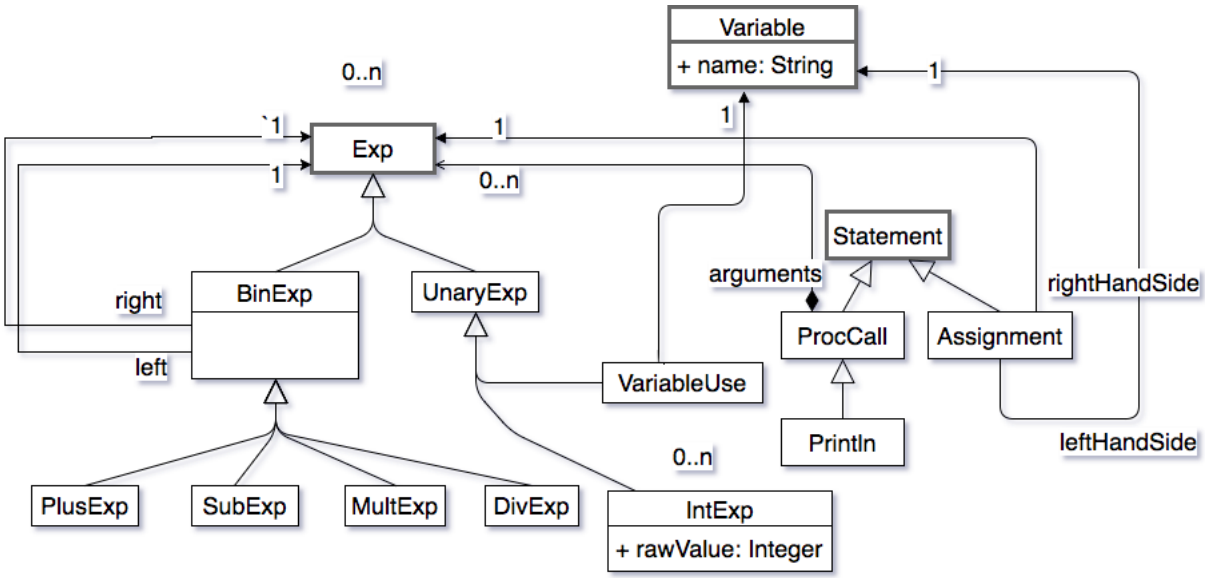

FIGURE 2 – Le diagrame des classes de la représentation interne

montre le diagramme des classes de la représentation interne. La représentation interne se compose des trois classes principales suivantes :

— *Exp* est la classe abstraite qui représente le concept d'expression ;
— *Statement* est la classe abstraite qui représente le concept d'instruction ;
— *Variable* met en oeuvre le concept de variable

3.1.1 Les expressions

On dispose de la représentation des concepts de nombre, d'opération et de reference à une variable.

Un nombre est une expression unaire. Dans cette première version, on ne met en oeuvre que les entiers représentés par la classe *IntExp*. La valeur entière manipulée est stockée dans la variable d'instance *rawValue*. Le code suivant permet de créer une représentation interne pour l'expression entière 28 :

```
1 IntExp new rawValue: 28
```

Concernant les opérations, le langage ne permet que les opérations binaires simples. La classe *BinExp* représente l'interface commune à toutes les opérations binaires possibles. Un BinExp est en relation avec deux expressions via les attributs *left* et *right*. Ces deux attributs permettent l'association entre une opération binaire et ses deux opérandes. Les opérations concrètes (addition, soustraction, multiplication

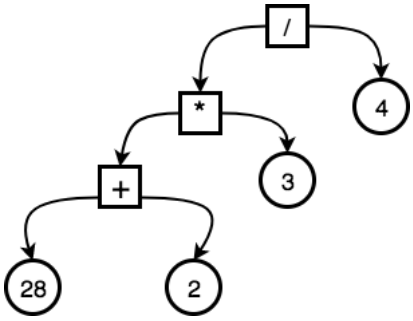

FIGURE 3 – Représentation en arbre de l'expression arithmétique $(28 + 2) * 3/4$

et division) sont représentées par les sous-classes de *BinExp*.

Le code suivant permet de créer une représentation interne pour l'expression $28 + 2$:

```
| vl vr a |
vl := IntExp new rawValue: 28.
vr := IntExp new rawValue: 2.
a := PlusExp new left: vl; right: vr.
```

On observe qu'on construit un arbre dont les noeuds feuilles sont des expressions unaires. Les noeuds intermédiaires sont des opérations. Le nombre de fils d'un noeud intermédiaire dépend de l'arité de l'opération ; deux dans le cas des opérations binaires.

Le code suivant montre comment construire la representation interne correspondant à l'arbre représenté par la figure 3 pour l'expression $(28 + 2) * 3/4$.

```
DivExp new
  left: (MultExp new
    left: (PlusExp new
      left: (IntExp new rawValue: 28)
      right: (IntExp new rawValue: 2))
    right: (IntExp new rawValue: 3))
  right: (IntExp new rawValue: 4)
```

Une variable permet de stocker une valeur et de lui associer un nom. La classe *VariableUse* représente la reference à une variable. Il s'agit d'une expression unaire qui permet de référencer une variable dans une expression.

Le code suivant permet de créer une représentation interne pour l'expression $a + 2$ qui reference une variable dont le nom est a :

```
PlusExp new left: (VariableUse new variable: (self getVariableNamed: 'a')); right: (IntExp new rawValue: 2).
```

L'envoi du message *getVariableNamed :* permet de récupérer la variable à partir de son nom. La méthode correspondante serait mise en oeuvre par le compilateur en utilisant l'ensemble des variables déjà existantes. Une table d'associations (une instance de Dictionary) pourrait être utilisée pour stocker l'ensemble des instances de *Variable* et permettre l'accès à une variable par son nom.

L'envoi du message *at :ifAbsentPut :* au dictionaire permet de récupérer la variable associée au nom passé en premier argument. Si l'association n'est pas trouvée alors le bloc passé en second argument est exécuté pour créer et stocker une nouvelle association.

```
Compiler>>getVariableNamed: aName
  ^ variables at: aName ifAbsentPut: [Variable new name: aName]
```

La valeur d'une variable n'est cependant pas stockée dans la représentation interne. La gestion de la valeur est de la responsabilité de la machine virtuelle (voir la section 3.2.2).

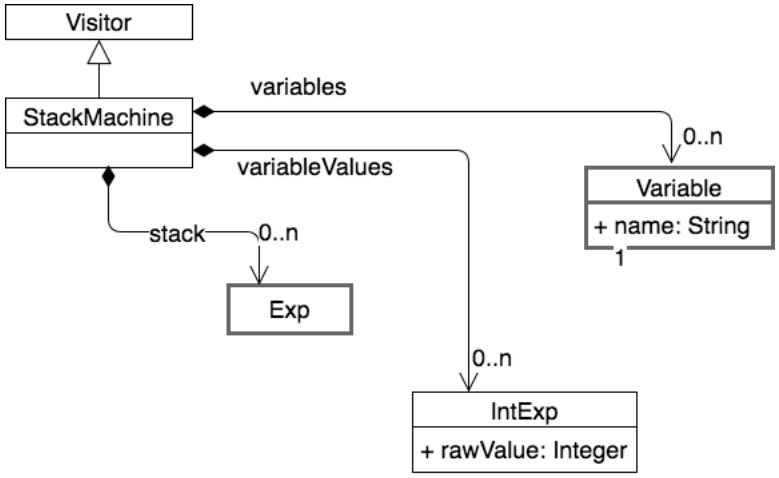

FIGURE 4 – Le diagrame des classes de la machine à pile

3.1.2 Les instructions

Le langage met en oeuvre l'affectation et dispose de la procédure *println* pour l'affichage d'une expression.

Concernant l'affectation, elle est mise en oeuvre par la classe *Assignement*. Une instance de *Assignment* est associée avec une expression (attribut *rightHandSide*) et à une instance de *Variable* (attribut *leftHandSide*). Un *Variable* représente une variable.

Pour traiter une affectation, le compilateur doit disposer de l'ensemble des variables déjà déclarées pour ne pas le dupliquer la représentation d'une variable. Voici un exemple de code pemettant de construire la représentation interne pour $a := 4;$:

```
1 Assignment new leftHandSide: (self getVariableNamed: 'a'); rightHandSide: (IntExp new rawValue: 4).
```

L'execution d'une affectation consiste à calculer la valeur de l'expression de droite et à stocker le résultat comme valeur de la variable à gauche.

Concernant la procedure *println*, il s'agit d'une primitive puisque son comportement est fixe. Son comportement est en effet codé dans la machine virtuelle. le code suivant permet de créer une représentation interne pour *println*(4); :

```
1 Println new argument: (IntExp new rawValue: 4)
```

3.2 La machine à pile

Pour exécuter un programme, il nous faut une machine. Nous voulons disposer d'une machine qui sait interpréter directement une représentation interne telle que présentée dans le chapitre précédent. La machine programmée est donc virtuelle par opposition à une machine réelle qui sait interpréter le binaire.

3.2.1 Structure de la machine à pile

La machine est dite "à pile" car elle utilise une pile pour le passage des arguments et le stockage du résultat des calculs.

Le langage permet d'utiliser des variables. La machine doit donc disposer aussi d'une liste de variables. Pour faciliter l'accès à une variable par son nom, la mise en oeuvre sera effectuée à l'aide d'une table de hash (instance de *Dictionary*).

La figure 4 montre la classe *StackMachine* qui représente la mise en oeuvre de la machine virtuelle pour notre langage. Une instance de *StackMachine* est en relation avec une liste d'expressions (la pile) et

une liste de variables (les variables globales) et une liste pour le stockage des valeurs des variables. Voici la déclaration de la classe *StackMachine* et le code de la méthode *initialize* :

```
Visitor subclass: #StackMachine
  instanceVariableNames: 'stack variables variableUses'
  classVariableNames: ''
  package: 'StackMachine-Core'

StackMachine>>initialize
  super initialize.
  stack := Stack new.
  variables := Dictionary new.
  variableUses := Dictionary new.
```

On remarque que des tables d'association sont utilisées pour stocker les variables et les valeurs des variables. Pour ces deux tables, la clé de l'association est le nom de la variable. Pour la liste des variables, la valeur de l'association est l'instance de Variable. Pour la liste des valeurs des variables, la valeur de l'association est la valeur de la variable.

3.2.2 Fonctionnement de la machine à pile

Le mécanisme d'exécution du patron Visiteur est exploité pour le calcul. Les fonctions de visite de la machine à pile mettent en oeuvre un parcours de la représentation interne en profondeur d'abord. Les calculs sont effectués pendant le parcours.

- Pour un noeuds expression feuille, aucun calcul n'est effectué. La visite d'un tel noeud provoque tout simplement l'empilement de l'expression associée : le noeud lui même pour une instance d'*IntExp* ou la valeur de la variable pour une instance de *VariableUse*.
- Pour un noeud intermédiaire représentant une expression :
  - avant le calcul d'une opérations, les noeuds opérandes sont d'abord parcourus ; suite à leur parcours, les opérandes à utiliser sont disponibles sur la pile ;
  - pour effectuer les calculs, la machine dépile les arguments et empile le résultat de sorte que le résultat courant est toujours en tête de pile.
- Pour un noeud représentant une instruction,
  - avant le calcul d'une instruction, les noeuds opérandes sont d'abord parcourus ; suite à leur parcours, les opérandes à utiliser sont disponibles sur la pile ;
  - pour exécuter l'instruction, la machine dépile les arguments avant de les utiliser ; la pile est donc normalement vide après exécution d'une instruction.

Voici le code de la visite pour un noeud *IntExp* et un noeud *VariableUse* qui empile tout simplement l'expression résultat.

```
StackMachine>>visitIntExp: anIntExp
  "on empile directement l'expression argument"
  stack push: anIntExp

StackMachine>>visitVariableUse: aVariableUse
  "on empile l'expression valeur du VariableUse argument"
  stack push: (variableValues at: aVariableUse variable name)
```

Voici le code de la visite pour une addition :

```
StackMachine>>visitPlusExp: aPlusExp
  | l r |
  "visite en profondeur d'abord a gauche puis a droite "
  aPlusExp left accept: self.
  aPlusExp right accept: self.
  "recuperation des arguments du + sur la pile "
  r := stack pop.
  l := stack pop.
  "empilement du resultat "
```

```
10    stack push: (IntExp new rawValue: (l rawValue + r rawValue))
```

Voici le code de la visite pour un *println*. L'expression argument est récupérée depuis l'instance du *Println*. Cette expression est visitée. Suite à cette visite, la valeur à afficher est en tete de pile. Cette valeur est dépilée et affichée sur le *Transcript*. :

```
1  StackMachine>>visitPrintln: aPrintln
2    | arguments |
3    "recuperation de l'argument (le premier de la collection)"
4    arguments := aPrintln arguments.
5    "visite de l'expression argument"
6    arguments first accept: self
7    "affichage sur le transcript"
8    stack pop logCr.
9    "la pile est vide"
```

Enfin, voici le code de la visite pour un *Assignment*. Tout d'abord on assure une entrée dans le dictionaire des variables en visitant la partie gauche. Ensuite, l'expression en partie droite est visitée. Le résultat en tête de pile est ensuite placé dans le dictionaire des variables comme valeur associée au nom de la variable.

```
1  StackMachine>>visitAssignment: anAssignment
2    "visite de la variable partie gauche (Variable) pour assurer qu'une entrÇe existe dans la table des valeurs"
3    anAssignment leftHandSide accept: self.
4    "calcul de la partie droite"
5    anAssignment rightHandSide accept: self.
6    "stockage de la valeur calculÇe dans le dictionaire"
7    variables at: anAssignment leftHandSide name put: stack pop
8    "la pile est vide"
```

Pour l'interprétation de l'affectation, il faut s'assurer qu'une entrée existe bien pour la variable en partie gauche dans le dictionnaire des variables. C'est ce qui est fait pour la visite d'une variable : une entrée est ajoutée dans le dictionnaire variables si cette entrée n'existe pas déjà. La valeur par défaut pour une variable est 0 :

```
1  StackMachine>>visitVariable: aVariable
2    "assure une entree dans la table stockant la valeur des variables ( 0 est la valeur par defaut)"
3    variableValues at: aVariable name ifAbsentPut: [ IntExp new rawValue: 0 ]
```