ISTIC
UFR INFORMATIQUE - ELECTRONIQUE

Licence STS 2ème année mentions Informatique & Mathématiques

CAL - TP 2 - UN INTERPRETEUR

La sémantique formelle du langage WHILE donnée au chapitre 5 pages 56 à 58, complétée page 61 de ce même chapitre pour la conditionnelle et au chapitre 6 page 78 pour la boucle `for`, se prête bien à une programmation en Scala. Le résultat sera un interpréteur de programmes WHILE écrit en Scala : $\texttt{interp}^{Scala}_{WHILE}$. Cet interpréteur réalise la fonction

$$\begin{array}{rcl} \texttt{interp} : \text{WHILE} \times \mathbb{D} & \rightarrow & \mathbb{D} \\ \langle \texttt{p}, \texttt{in} \rangle & \mapsto & \texttt{out} = \mathcal{SEM}(\texttt{p})(\texttt{in}) \end{array}$$

Ce sujet est traité en trois séances. Il suit à la lettre la définition de la sémantique du langage WHILE.

Spécification

On fixe le cahier des charges suivant :

- L'interpréteur reçoit en paramètre l'arbre de syntaxe abstraite d'un programme à interpréter et celui d'une donnée, représentant les valeurs d'entrée de ce programme.
- L'interpréteur implémente la sémantique formelle du langage WHILE augmenté des conditionnelles et des boucles FOR. L'expression de la sémantique doit être légèrement adaptée car le constructeur de séquence de commandes, le ';', n'est pas représenté dans l'arbre de syntaxe abstraite. On trouve à la place une liste d'arbres de syntaxe abstraite.

Plan de développement

Le plus simple est de suivre la définition de la sémantique formelle du langage WHILE.

1. Définir les fonctions d'accès à la mémoire. On suggère de représenter un ensemble d'associations $[v \mapsto d]$ par une liste de paires (v, d).

   La lecture de la valeur d'une variable v revient à parcourir la liste à la recherche d'une paire (v, d) ; le résultat est alors d. L'écriture d'une valeur d dans la variable v revient à insérer une paire (v, d) dans la liste ; on n'oubliera pas de retirer une éventuelle paire (v, d') [1].

   L'ordre de rangement des paires (v, d) dans la liste est indifférent. Il faut donc ne pas en tenir compte lorsque des exemples sont fournis. En revanche, les cas de test doivent être cohérents avec la programmation choisie puisqu'ils vont révéler un rangement particulier.

   Le type `Value` introduit ci-dessous décrit le type des valeurs c-à-d. les élements du domaine $\mathbb{D}$:

   ```
   sealed trait Value
   case object NlValue extends Value
   case class CstValue(name: String) extends Value
   case class ConsValue(arg1: Value, arg2: Value) extends Value
   ```

1. Il est possible de se passer de cette précaution. Cependant, il n'est pas souhaitable que la taille de la mémoire croisse systématiquement avec le temps d'exécution.

On définit le type `Memory` afin de représenter une mémoire :

```
type Memory = List[(Variable, Value)]
```

Les spécifications des fonctions d'accès à une mémoire, `lookUp` pour la lecture avec sa notation $m(v)$ et `assign` pour l'écriture avec sa notation $m[v \mapsto d]$, sont décrites page 57 du chapitre 5 :

```
/**
* @param v : une variable
* @param mem : une mémoire
* @return m(v), c'est-à-dire la valeur de la variable v dans la mémoire mem,
* la valeur par défaut si la variable v n'est pas présente dans la mémoire mem
*/
def lookUp(v: Variable, mem: Memory): Value

/**
@param v : une variable
@param d : une valeur
@param mem : une mémoire
@return la mémoire modifiée par l'affectation [v->d]
/
def assign(v: Variable, d: Value, mem: Memory): Memory
```

2. Définir une fonction d'évaluation `interpreterExpr` des expressions. Une telle fonction prend en paramètre une expression et retourne une valeur. Il suffit de suivre la sémantique formelle, $\mathcal{SEM}_e$.

```
/**
* @param expression : un AST décrivant une expression du langage WHILE
* @return SEM_e(expression, m), c'est-à-dire la valeur de l'expression
*/
def interpreterExpr(expression: Expression, mem: Memory): Value
```

La fonction `interpreterExpr` ci-dessus calcule la valeur associée à une expression. Il peut être utile de produire à l'inverse une expression associée à une valeur. Définir une fonction `valueToExpression` qui construit l'expression la plus simple associée à une valeur.

```
/**
* @param value : une valeur du langage WHILE
* @return l'AST décrivant l'expression de cette valeur
*/
def valueToExpression(value: Value): Expression
```

3. Définir une fonction d'interprétation de chaque sorte de commande. Elle traite toutes les commandes traitées par $\mathcal{SEM}_c$ sauf la séquence, ';', qui est représentée par des listes dans l'arbre de syntaxe abstraite.

```
/**
* @param command : un AST décrivant une commande du langage WHILE
* @param memory : une mémoire
* @return SEM_c(command, m), c'est-à-dire la mémoire après l'interprétation
* de la commande
*/
def interpreterCommand(command: Command, memory: Memory): Memory
```

4. Définir une fonction qui interprète les listes de commandes [2]. Elle traite le cas des séquences dans $\mathcal{SEM}_c$.

```
/**
* @param commands : une liste non vide d'AST décrivant une liste non vide de commandes
* @param memory : une mémoire
* @return la mémoire après l'interprétation de la liste de commandes
*/
def interpreterCommands(commands: List[Command], memory: Memory): Memory
```

   On remarquera que `InterpreterCommand` et `InterpreterCommands` sont mutuellement récursives. On ne peut donc les tester entièrement que lorsqu'elles sont toutes deux écrites. En revanche, on peut les développer et tester progressivement en commençant par les cas de base, qui ne causent pas d'appels récursifs (voir la stratégie de développement proposée en dernière page de l'annexe "Environnement pour les travaux pratiques").

5. Définir une fonction qui prend en paramètre une liste de variables [3] et une liste de valeurs (toutes deux supposées de même longueur) et enregistre les valeurs dans les variables : la première valeur dans la première variable, la deuxième dans la deuxième, etc. Cela revient à initialiser la mémoire avec des valeurs d'entrée.

```
/**
* @param vars : une liste non vide décrivant les variables d'entrée d'un programme
* @param vals : une liste non vide de valeurs, de même longueur que vars
* @return une mémoire associant chaque valeur à la variable d'entrée correspondant
*/
def interpreterMemorySet(vars: List[Variable], vals: List[Value]): Memory
```

6. Définir une fonction qui extrait les valeurs de sortie de la mémoire. Elle prend en paramètre une liste de variables de sortie et une mémoire, et retourne la liste des valeurs prises par les variables de sortie dans la mémoire. Les valeurs sont retournées dans l'ordre des variables.

```
/**
* @param vars : une liste non vide décrivant les variables de sortie d'un programme
* @param memory : une mémoire
* @return la liste des valeurs des variables de sortie
*/
def interpreterMemoryGet(vars: List[Variable], memory: Memory): List[Value]
```

7. Définir une fonction qui exécute la sémantique, $\mathcal{SEM}$, d'un programme, avec des valeurs d'entrée passées en paramètre.

```
/**
* @param program : un AST décrivant un programme du langage WHILE
* @param vals : une liste de valeurs
* @return la liste des valeurs des variables de sortie
*/
def interpreter(program: Program, vals: List[Value]): List[Value]
```

2. La grammaire de WHILE est telle qu'il n'existe pas de liste vide de commandes
3. La grammaire de WHILE est telle qu'il n'existe pas de liste vide de variables