

TME – Sémantique Opérationnelle

Remarque : Les personnes qui suivent le module de Compilation Avancée sont invitées à implanter un analyseur syntaxique (avec les outils CamlLex et CamlYacc) pour les expressions et les programmes manipulés dans ce TME.

1 Expressions

On considère le langage d'expressions, construit à partir d'un ensemble V de symboles de variable et de l'ensemble $\mathbb{C} = \mathbb{Z} \cup \mathbb{B}$ de constantes, dont la syntaxe abstraite est définie par :

$$\begin{array}{ll}
 e ::= & x \quad \text{(variable)} \\
 & | k \quad \text{(constante)} \\
 & | e + e \mid -e \mid e \times e \mid e/e \quad \text{(exp. arithmétiques)} \\
 & | e = e \mid \text{not } e \mid e \text{ or } e. \quad \text{(exp. booléennes)}
 \end{array}$$

Les expressions regroupent donc les variables, les constantes, les expressions arithmétiques et les expressions booléennes.

On choisit de représenter les variables par des chaînes de caractères. On définit donc les types suivants :

```

type var = VAR of string;;

type cst = B of bool | Z of int;;

type expr = V of var          | C of cst
           | PLUS of expr * expr | MOINS of expr
           | FOIS of expr * expr | DIV of expr * expr
           | EGAL of expr * expr | NON of expr
           | OU of expr * expr;;

```

Evaluer une expression consiste à lui associer une valeur appartenant à un certain ensemble \mathbb{V} . Trois situations se présentent :

1. l'évaluation produit un entier ou un booléen
2. l'expression est "mal-typée" (par exemple `true + 2`, `not 8`, ...) ou nécessite de connaître la valeur associée à une variable "non définie" (i.e. l'accès à la mémoire ne fournit pas de valeur pour cette variable) et dans ces deux cas le résultat de l'évaluation est `Wrong`
3. l'évaluation de l'expression conduit à une division par 0 et dans ce cas le résultat de l'évaluation est `Err`

On a donc $\mathbb{V} = \mathbb{C} \cup \{\text{Wrong}, \text{Err}\}$. On décide d'implanter les valeurs particulières `Wrong` et `Err` par des exceptions :

```

exception Wrong;;
exception Err;;
type valeur = cst;;

```

L'évaluation d'une expression s'obtient à partir d'un "état" permettant de connaître la valeur associée à certaines variables. La notion d'état est implantée par une liste d'association :

```
type env = (var * valeur) list;;
```

Q1. Définir une fonction `valeur_de:env -> var -> valeur` qui calcule la valeur associée à une variable par un état (penser à utiliser la fonction prédéfinie `List.assoc`). Si la variable n'est pas définie par l'état, cette fonction lèvera l'exception `Wrong`.

Q2. L'évaluation des expressions s'obtient de manière classique en donnant une signification à chacun des constructeurs :

$$\mathcal{E}[e]_\rho = \begin{cases} \rho(x) & \text{si } e = x \\ c & \text{si } e = c \\ \mathcal{E}[e_1]_\rho \ [+] \ \mathcal{E}[e_2]_\rho & \text{si } e = e_1 + e_2 \\ [-] \ \mathcal{E}[e_0]_\rho & \text{si } e = -e_0 \\ \mathcal{E}[e_1]_\rho \ [\times] \ \mathcal{E}[e_2]_\rho & \text{si } e = e_1 \times e_2 \\ \mathcal{E}[e_1]_\rho \ [/] \ \mathcal{E}[e_2]_\rho & \text{si } e = e_1/e_2 \\ [\text{not}] \ \mathcal{E}[e_0]_\rho & \text{si } e = \text{not } e_0 \\ \mathcal{E}[e_1]_\rho \ [=] \ \mathcal{E}[e_2]_\rho & \text{si } e = (e_1 = e_2) \\ \mathcal{E}[e_1]_\rho \ [\text{or}] \ \mathcal{E}[e_2]_\rho & \text{si } e = e_1 \text{ or } e_2 \end{cases}$$

Définir une fonction `eval_expr:env -> expr -> cst` qui calcule la valeur d'une expression (étant donné un état). Cette fonction lèvera les exceptions `Wrong` ou `Err` dans les situations décrites plus haut.

Q3. La fonction `eval_expr` définit la sémantique des expressions et définit donc les opérateurs `[[+]]`, `[[−]]`, Compléter les tables suivantes définissant ces opérateurs :

[[+]]	Wrong	Err	$n_2 \in \mathbb{Z}$	autres	[[×]]	Wrong	Err	$n_2 \in \mathbb{Z}$	autres
Wrong					Wrong				
Err					Err				
$n_1 \in \mathbb{Z}$					$n_1 \in \mathbb{Z}$				
autres					autres				

[[−]]	Wrong	Err	$n \in \mathbb{Z}$	autres

[[/]]	Wrong	Err	$n_2 \in \mathbb{Z}$	autres
Wrong				
Err				
$n_1 \in \mathbb{Z}$			si $n_2 \neq 0$ sinon	
autres				

[[=]]	Wrong	Err	$n_2 \in \mathbb{Z}$	$b_2 \in \mathbb{B}$
Wrong				
Err				
$n_1 \in \mathbb{Z}$			si $n_1 = n_2$ sinon	
$b_1 \in \mathbb{B}$				si $b_1 = b_2$ sinon

	<code>[[or]]</code>	Wrong	Err	$b_2 \in \mathbb{B}$	autres
<code>[[not]]</code>	Wrong	Err	$b \in \mathbb{B}$	autres	
	Wrong				
				si	
				sinon	

Les opérations binaires sont-elles commutatives ? Pourquoi ? Cela pose-t-il un problème ? Y-aviez vous pensé lors de l'implantation ?

Remarque : En OCaml l'évaluation d'une paire (e_1, e_2) se fait de "droite à gauche" (i.e. l'expression e_2 est évaluée avant l'expression e_1 et donc si l'évaluation de e_2 lève une exception, e_1 n'est pas évaluée).

2 Programmes

A partir des expressions, on définit la syntaxe abstraite des programmes comme suit :

```

c ::=  x := e           (affectation)
      | input x         (lecture)
      | if e then c else c (conditionnelle)
      | while e do c.   (boucle)
      | c; c            (séquence)
      | skip.

```

On définit donc le type suivant :

```

type com = SET of var * expr
          | INPUT of var
          | IF of expr * com * com
          | WHILE of expr * com
          | SEQ of com * com
          | SKIP;;

```

Q4. Définir une fonction `semop:env -> com -> env` qui permet de calculer, s'il existe, l'état obtenu après exécution d'un programme c dans un état σ . Cette fonction rattrapera les exceptions levées par la fonction d'évaluation des expressions et dans ce cas, lèvera une exception et affichera un message adéquat.

Q5. Tester votre implantation avec :

1. un programme qui calcule le PGCD de deux nombres
2. un programme qui boucle
3. un programme "mal-typé"
4. un programme dont l'exécution conduit à une division par zéro

Q6. Pour aller plus loin ... ajouter la construction `repeat` à votre programme, écrire une fonction qui remplace les constructions `repeat` d'un programme par des constructions `while` (et *vice-versa*), ajouter une fonctionnalité permettant de suivre pas à pas l'exécution d'un programme ...