Série d'exercices #8

IFT-2035

June 3, 2025

8.1 Définition par cas

Soit une macro iftcase en ELisp qui peut s'utiliser comme suit:

```
(iftcase exp
  ((1 3 5) exp1)
  ((4) exp2)
  (_ exp3))
```

qui signifierait: evalue d'abord *exp* puis évalue exp_1 si *exp* rend 1, 3, ou 5; ou évalue exp_2 si *exp* rend 4; ou évalue exp_3 sinon.

1. Montrer l'expansion désirée pour ce code
2. Identifier les risques possibles d'évaluation répétée excessive et de capture de nom, si la macro est définie trop naïvement.
3. Donner une définition de cette macro qui ne souffre pas de ces problèmes.

8.2 Passage de paramètres

Soit le code suivant dans le langage hypothétique Zlika:

```
VAR a : INTEGER, b, c : ARRAY OF INTEGER;
...
PROCEDURE move (x, y : INTEGER, n : INTEGER)
LOCAL VAR a
BEGIN
   a := y;
   x := a + n;
   y := y - n;
END
...
move (a, b[a], c[a])
```

Réécrire 4 fois le code ci-dessus en C:

- Une fois, en supposant que Zlika passe les arguments par valeur.

- Une fois, en supposant que Zlika passe les arguments par référence.
- Une fois, en supposant que Zlika passe les arguments par valeur-résultat.
- Une fois, en supposant que Zlika passe les arguments par nom.

Questions complémentaires:

- Combien de styles de passage de paramètres le langage C offre-t-il?
- Que se passe-t-il si on essaie de faire de telles simulations dans un langage qui utilise un autre style de passage de paramètres?

8.3 Raisonner

Soit le morceau de code suivant, où `f` est une fonction quelconque que l'on ne connaît pas et où le langage utilise une syntaxe de style C:

```
{
  int table[2] = {0, 1};
  int size = 2;
  int tmp = 0;

  f (table, size);
  ...
}
```

On aimerait savoir si certaines conditions sont nécessairement toujours vraies aprés l'appel à `f`. On s'intéresse plus particulièrement aux conditions suivantes:

- `table[0] == 0`
- `size == 2`
- `tmp == 0`

Indiquer lesquelles de ces trois conditions sont nécessairement vraies dans chacun des cas suivants:

1. Le langage est exactement comme C: portée statique, passage d'arguments par valeur, affectation autorisée.
2. Le langage est comme C sauf que l'affectation (autre que l'initialization) est interdite.
3. Le langage est comme C sauf que les arguments sont passés par référence.
4. Le langage est comme C mais avec portée dynamique.

8.4 Programmation impérative en langage fonctionnel

Re-voici le code d'un évaluateur en Haskell:

```
type Var = String

-- Expressions du code source en forme ASA.
data Exp = Enum Int         -- Une constante
         | Evar Var         -- Une variable
         | Elambda Var Exp  -- Une fonction
         | Ecall Exp Exp    -- Un appel de fonction

data Val = Vnum Int           -- Un nombre entier
         | Vfun (Val → Val)  -- Une fonction

elookup ((x1,v1):env) x =
    if x == x1 then v1 else elookup env x

env0 = [("+", Vfun (λ(Vnum x1) → Vfun (λ(Vnum x2) → Vnum (x1 + x2))))]

eval env (Enum n) = Vnum n
eval env (Evar x) = elookup env x
eval env (Elambda arg body) =
    Vfun (λactual → eval ((arg,actual):env) body)
eval env (Ecall fun actual) =
    case eval env fun of
      Vfun f → f (eval env actual)
```

Changer le code en utilisant le monade `IO` pour y ajouter une fonction primitive *print_hello* dans l'environnement initial qui imprime le mémorable message "`Hello:`" suivi de son argument numérique (qui est aussi renvoyé comme valeur de retour). Pour cela, il utilisera *putStrLn* :: `String -> IO ()`.