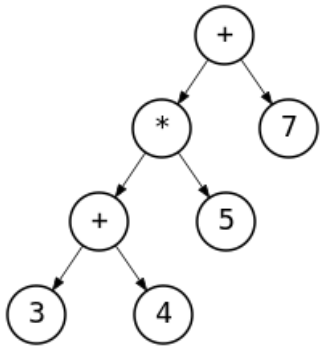
TP 3 & 4 : Patron de Conception *Stratégie* et Stratégies d'évaluation

©2022 Ghiles Ziat
ghiles.ziat@epita.fr

Nous allons dans ce TP étendre l'évaluateur arithmétique du premier TP pour implémenter différentes stratégies d'évaluation pour les appels de fonctions. Finissez d'abord le premier TP (dont le sujet vous est donné ci-dessous) si ce n'est pas déjà fait avant de commencer celui-ci.

Exercice I : (TP1) Patron de conception *Composite*

Ce patron de conception (*design pattern*) permet de concevoir une structure arborescente récursive, permettant d'implémenter la même interface pour les feuilles et les noeuds afin qu'ils soient manipulés de la même manière. On appelle ce genre de représentation arborescente un arbre de syntaxe abstraite (AST), où chaque noeud de l'arbre désigne une construction apparaissant dans le texte. Par exemple l'expression $(3 + 4) * 5 + 7$ peut être représentée par l'AST suivant :



Dans cet exercice, nous allons utiliser le design pattern *composite* pour implémenter un évaluateur d'expressions arithmétiques. On supposera dans un premier temps que nos expressions arithmétiques sont soit des constantes, soit des opérations binaires parmi $\{+, -, *, /\}$.

Q1 – Créez une classe `Constant` pour modéliser des constantes entières avec un slot `value`.

Q2 – Créez une classe `Binop` pour modéliser des opérations binaires. Une opération binaire possède

deux opérandes (qui sont elles-memes des expressions arithmétiques) et un opérateur représenté par une chaine de caractères.

Q3 – Définissez des fonctions `plus`, `minus`, `mul` et `div` qui prennent deux expressions en paramètres et retournent une instance de la classe `Binop`. Définissez de plus une fonction `cst` qui prend un entier en paramètre et fabrique une instance de `Constant`.

Q4 – Créez une fonction générique `evaluate` avec deux méthodes, l'une qui prend une instance de la classe `Constant` et l'autre une instance de la classe `Binop` en paramètre. Elles devront retourner la valeur de l'expression arithmétique correspondante. Pour les opération binaires, vous pourrez utiliser la fonction annexe suivante pour choisir la bonne opération à appliquer :

```
(defun op-to-func (op)
  (cond ((string= op "+") #'+)
        ((string= op "*") #'*)
        ((string= op "-") #'-)
        ((string= op "/") #'/)
        (t (error "unrecognized operator"))))
```

Q5 – Vérifiez que `(evaluate (plus (cst 2) (mul (cst 4) (cst 10))))` donne bien 42.

Exercice II : (TP1) Des fonctions comme objets

Nous allons à présent ajouter des fonctions à notre langage. Pour cela on veut pouvoir représenter trois nouveaux types d'expression : les fonctions, les appels de fonction et les variables. Une modélisation possible consiste à faire référence aux arguments passés à une fonction par leur position (et non par leur nom), comme en bash, par exemple :

```
identity() {
  echo "$1"; #print first argument
}
```

Q1 – Pour cela ajoutez trois classes :

- une classe `fun` avec un slot `body` qui sera une expression,
- une classe `argument` avec un slot `index` entier,
- une classe `call`, pour les appels de fonctions, avec un slot `fun` qui sera une instance de `fun` et un slot `args` qui sera une liste d'expressions.

Q2 – Créez des constructeurs `make-fun`, `make-arg` et `make-call` pour ces classes. Par exemple, la fonction carré pourra se déclarer comme suit :

```
  (defvar square (make-fun :body (mul (make-arg 0) (make-arg 0))))
```

Q3 – Pour pouvoir évaluer les appels de fonctions, il nous faut une structure de données qui stockera la valeur des arguments dans l'ordre où ils sont passés. Nous allons donc modifier les méthodes `evaluate`

des constantes et des opérations binaires pour y ajouter un argument `context` qui jouera ce role. Il n'est pas nécessaire de modifier le corps de ces méthodes.

Q4 – Ajoutez deux méthodes `evaluate`, l'une qui prend une instance de la classe `argument` et une autre une instance de la classe `call` en paramètre et qui retournent la valeur de l'expression correspondante en respectant les règles suivantes :

- l'évaluation du n-ième argument consiste à récupérer la n-ième valeur de `context`.
- l'évaluation d'un appel de fonction est celui d'un évaluateur fonctionnel strict : évaluer les arguments, remplir le contexte d'évaluation avec la liste des résultats et évaluer le corps de la fonction appellée. Vous pourrez pour l'évaluation des arguments utiliser la fonction de mapping fonctionnel de Lisp, `mapcar`, qui prend une fonction f et une liste de valeurs $(a_1 \dots a_n)$ en arguments, et construit la liste $((f\ a_1) \dots (f\ a_n))$.

Exemple | `(mapcar (lambda(x) (+ x 1)) '(1 2 3)) = '(2 3 4)`

Q5 – Vérifiez que

- `(evaluate (plus (make-arg 0) (make-arg 1)) (list 1 2))` donne bien 3
- `(evaluate (make-call square (list (cst 7))) ())` donne bien 49.

Exercice III : Tracer l'évaluation

Pour avoir un meilleur recul sur l'ordre d'execution de notre evaluateur, nous allons dans un premier temps definir une classe `show` pour encapsuler les expressions.

Q1 – Définissez une classe `show` avec un slot `expr` et un slot `cnt` initialisé par défaut à 0. Définissez de plus un constructeur `show`, qui prend une expression en paramètre et retourne une instance de `show`.

Q2 – Ajoutez une méthode `evaluate` pour les instance de `show`. Celle-ci devra d'abord afficher la valeur du slot `cnt` et l'expression sous-jacente [1] avant d'incrementer `cnt` et de procéder à l'évaluation de l'expression.

Par exemple, le code suivant :

```
(defvar s42 (show (plus (cst 1) (cst 41))))
(evaluate s42 ())
(evaluate s42 ())
```

devra produire :

```
evaluation number 1 of #<BINOP {100210D753}>
evaluation number 2 of #<BINOP {100210D753}>"
```

1. Vous pourrez utiliser la fonction `format` avec la directive `"~A"`, il n'est pas nécessaire de définir une méthode d'affichage plus "précise".

Q3 – Définissez une fonction `appshow` qui construit un appel de fonction en encapsulant chaque argument dans une instance de `show`. Vous pourrez utiliser la fonction `map`, comme déja vu dans le premier TP.

Q4 – On se donne la fonction `twice` suivante, qui additionne le parametre de la fonction à lui-même :

```
(defvar twice
  (make-fun :body (plus (make-arg 0) (make-arg 0))))
```

Vérifiez que l'évaluation du code suivant n'évalue qu'une seule fois l'argument de la fonction.

```
(evaluate (appshow twice (list (minus (cst 8) (cst 2)))) ())
```

Q5 – On se donne de plus la fonction `first-of-two` qui prend deux [2] paramètres et qui retourne le premier :

```
(defvar first-of-two
  (make-fun :body (make-arg 0)))
```

Vérifiez que le code suivant évalue tous ses arguments et produit bien une erreur.

```
(evaluate (appshow first-of-two (list (cst 1) (div (cst 8) (cst 0)))) ())
```

Exercice IV : Evaluation Stricte *VS* Evaluation Paresseuse

On se propose dans cet exercice de rendre notre évaluateur paramétrable en la strategie d'évaluation qu'il utilise. L'idée est de dire qu'une stratégie d'évaluation fait principalement deux choses :

- Lors de l'appel, elle applique un traitement (potentiellement l'identité) aux arguments d'une fonction, et remplit le contexte d'appel avec les arguments traités.
- Lorsqu'on rencontre un argument dans le corps d'une fonction, on lui applique un traitement (potentiellement l'identité) à ce moment.

Nous allons Ajouter à nos méthodes `evaluate` un argument `strategy`. Pour rappel, vous devez en avoir 5, spécialisées sur chacune des différents types d'expression, de signatures :

- `(defmethod evaluate ((c constant) context)`
- `(defmethod evaluate ((b binop) context)`
- `(defmethod evaluate ((a argument) context)`
- `(defmethod evaluate ((c call) context)`
- `(defmethod evaluate ((s show) context)`

2. Notre evaluateur n'effectue aucune vérification sur l'arité des appels de fonctions, on peut donc appeller cette fonction avec n'importe quel nombre d'argument. On supposera cependant que toutes les fonctions sont appellées avec le bon nombre d'argument.

Q1 – Spécialisez les méthodes `evaluate` pour les appels de fonctions et les arguments pour avolir deux traitements différents selon que la stratégie passée en paramètre soit stricte (*call-by-value*) ou paresseuse (*call-by-name*).

Pour ce faire nous pouvons utiliser un *eql-specializer*, qui permet de spécifier que l'argument spécialisé doit être égal à l'objet correspondant pour que la méthode soit applicable. Vous devrez donc avoir deux méthodes pour les appels de fonctions :

- `(defmethod evaluate ((c call) context (strategy (eql :by-value))) ...)`
- `(defmethod evaluate ((c call) context (strategy (eql :by-name)))...)`

Ainsi que deux méthodes pour les arguments :

- `(defmethod evaluate ((a argument) context (strategy (eql :by-value))) ...)`
- `(defmethod evaluate ((a argument) context (strategy (eql :by-name))) ...)`

On rappelle que le *call-by-value* a l'avantage de n'évaluer chaque argument de la fonction qu'une seule fois, lors de l'appel, tandis que le *call-by-name* a l'avantage qu'un argument de fonction ne sera pas évalué tant que la valeur correspondante n'est pas utilisée dans le corps de la fonction. Par contre, un argument peut etre évalué plusieurs fois.

Q2 – Vérifiez à présent que l'évaluation du code suivant évalue deux fois l'argument.

```
(evaluate (appshow twice (list (minus (cst 8) (cst 2)))) () :by-name)
```

Q3 – Vérifiez de plus que l'évaluation du code suivant ne produit plus d'erreur :

```
(evaluate (appshow first-of-two (list (cst 1) (div (cst 8) (cst 0)))) () :by-name)
```

Exercice V : Expressions paresseuses et *call-by-need*

Nous allons ajouter dans cette exercice des expressions memoïsées, dont l'évaluation n'aura lieu que la première fois qu'elle sont utilisées, puis sera mise en cache pour les utilisations futures.

Q1 – Définissez une classe `memocell` avec les *slots* suivants :

- `is_cached`, initialisé par défaut à *nil*. Ce slot servira à stocker un booléen qui indiquera si on a déjà évalué l'expression encapsulée.
- `cache_value` initialisé par défaut à *nil*. Ce slot servira à stocker le résultat de l'évaluation de l'expression encapsulée.
- `expression` qui encapsulera une expression.

Q2 – Définissez de plus un constructeur `make-cell`, qui prend une expression en paramètre et retourne une instance de `memocell`.

Q3 – Ajoutez une méthode `evaluate` pour les expression memoïsées. Celle-ci devra vérifier si l'expression a dejà été calculée, et si oui, renvoyer la valeur mise en cache. Sinon, elle calculera la valeur correspondante et mettra à jour le cache.

Q4 – Vérifiez que l'évaluation du code suivant n'évalue qu'une fois l'argument.

```
(evaluate (make-call twice (list (make-cell (show (minus (cst 8) (cst 2)))))) () :
   by-name)
```

Nous avons à présent tous les ingrédients nécessaires à l'implémentation d'une stratégie d'évaluation paresseuse optimisée : le *Call-By-Need*. Comme le *Call-By-Name* celle ci n'évaluera ses arguments que lorsqu'ils seront rencontrés dans le corps d'une fonction. Par contre ceux-ci seront maintenant memoïsés.

Q5 – Spécialisez la méthode `evaluate` des appels de fonction pour une evaluation paresseuse optimisée : vous devrez simplement transformer les arguments passés à la fonction en instance de la classe `memocell` lors de l'appel de fonction. L'evaluation des arguments reste le même que pour le *Call-by-Name*.

Q6 – Vérifiez que le code suivant n'évalue qu'une seule fois son argument.

```
(evaluate (appshow twice (list (minus (cst 8) (cst 2)))) () :by-need)
```