TP 1 : Classes et Objets

©2022 Ghiles Ziat
ghiles.ziat@epita.fr

Exercice I : Visibilité (LISP)

Q1 – Créez une classe `player`

Q2 – Ajoutez à votre classe des slots correspondant au nom et à l'age d'un joueur ; fournissez des readers pour ces slots.

Q3 – Définissez une fonction d'instanciation `make-player` avec paramètres nommés (keywords) pour la classe `player`.

Q4 – Ajoutez un champ entier `count` qui compte le nombre d'instances créées. Vous devrez modifier votre fonction d'instanciation pour qu'elle incrémente la valeur de *count* à chaque appel.

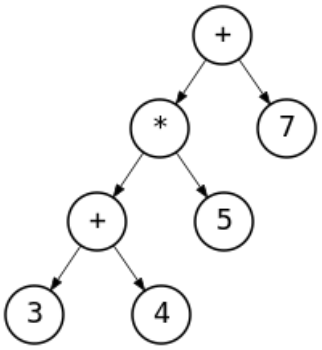
Q5 – Ajoutez à la classe un slot `team` dont la valeur devra être 0 ou 1 seulement (il y a deux équipes). On veut garantir la parité de telle sorte à ce que le nombre d'instances ayant un champ `team` à `0` soit le même (à 1 près) que le nombre d'instances ayant le champ `team` à `1`. Comment faire ?

Q6 – Définissez une fonction `print-player` qui affiche sur la sortie standard les informations relatives à une instance de la classe `player`.

Q7 – Créez quelques instances de la classe `player` et affichez leurs informations.

Exercice II : 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 III : 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.