Programmation
Examen

A la fin des années 1950, John McCarthy créa un nouveau langage de programmation appelé *Lisp* (list processing). Ce langage repose sur la manipulation de listes. Le but de ce problème est de réaliser un interpréteur pour un sous-ensemble du langage Lisp.
Un programme *Lisp* est composé d'une suite d'expressions. Une expression est soit un atome (entier, chaîne de caractères, ou symbole) soit une liste. Une liste est soit *empty* (la liste vide) soit une suite d'expressions entre parenthèses.
Exemples :
Atomes :

- Entiers : 1 123
- Chaînes : "abc" "1 23 jk"
- Symboles : abc a12 xy

Listes :

- Liste vide : *empty*
- Listes non vides : (1 2 3) (abc (1 2) 5)

Quelques symboles sont des symboles prédéfinis :

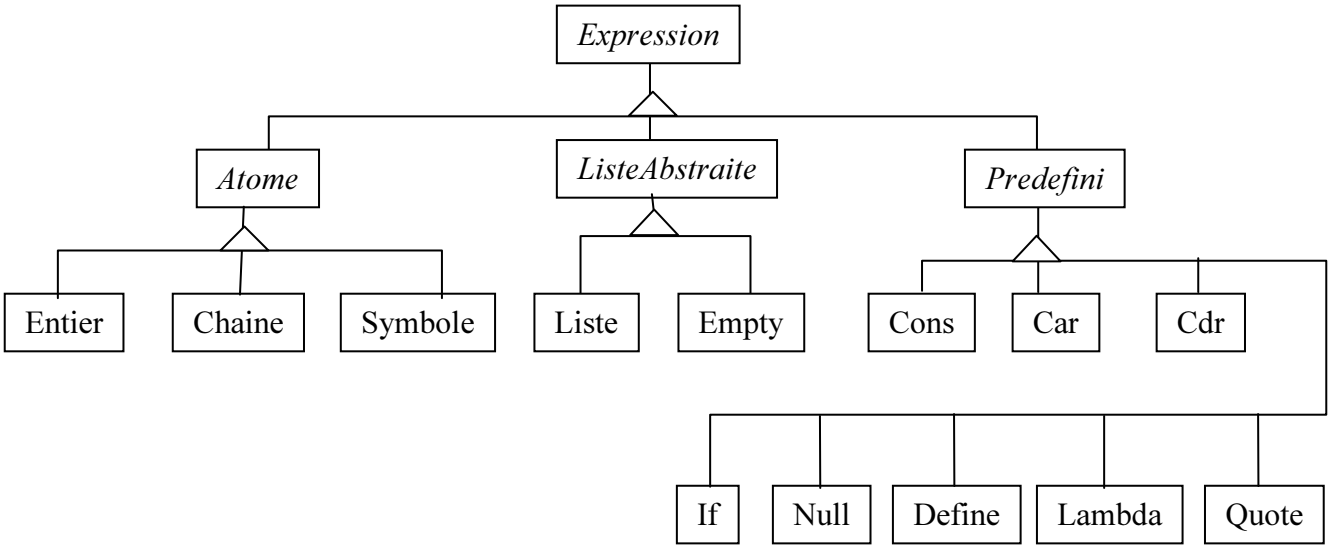
- **cons** : construit une liste à partir d'un élément et d'une liste.
- **car** : retourne le premier élément de la liste en argument (l'argument ne peut pas être la liste vide).
- **cdr** : retourne la liste argument moins le premier élément (l'argument ne peut pas être la liste vide).
- **define** : définit un symbole en lui associant une valeur.
- **quote** : retourne l'expression sans l'évaluer.
- **lambda** : permet de définir une fonction.
- **if** : retourne une expression en fonction d'une condition.
- **null** : retourne *vrai* si la liste en argument est vide et *faux* sinon.

Exemples :

expression	**Evalué en :**
123	123
"abc"	"abc"
(cons 1 empty)	(1)
(cons 1 (cons 2 empty))	(1 2)
(car (cons 1 (cons 2 empty)))	1
(cdr (1 2 3 4 5))	(2 3 4 5)
(quote (cdr (1 2 3 4 5)))	(cdr (1 2 3 4 5))
(define liste (1 2 3 4))	Retourne (define liste (1 2 3 4)), et le symbole liste est associé à l'expression (1 2 3 4)
(if (null liste) 12 21)	21
(define copie (lambda (x) (if (null x) empty (cons (car x) (cdr x)))))	Définit la fonction *copie* qui a un argument et qui permet de copier la liste en argument, et retourne

	(define copie (lambda (x) (if (null x) empty (cons (car x) (cdr x)))))
(copie (1 2 3))	(1 2 3)

Nous nous proposons de construire un interpréteur pour un sous ensemble du langage *Lisp*, que nous appellerons *MicroLisp*. Nous définissons les classes suivantes pour représenter les différents éléments du langage *MicroLisp*:



Pour chacune de ces classes nous nous proposons de définir les constructeurs, la méthode *toString()*, et la méthode *evaluer()*, qui est l'interpréteur du langage *MicroLisp*.

Question 1. (2 points)
Pour les classes *Entier* et *Chaine*, définir :

- le constructeur.
- la méthode *toString()*. La méthode *toString()* retourne la valeur de l'entier ou de la chaîne.
- la méthode *evaluer()*. La méthode *evaluer()* retourne l'objet lui-même

Question 2. (3 points)
Pour la classe *Symbole*, définir :

- le constructeur. Le paramètre est le nom du symbole.
- les méthodes *toString()*. La méthode *toString()* retourne le nom du symbole.
- les méthodes *evaluer()*. La méthode *evaluer()* retourne l'évaluation de l'expression qui est la valeur du symbole si cette valeur existe, et le symbole lui-même si cette valeur n'existe pas. Les couples (symbole, expression) sont représentés dans la variable de classe *environnement*, qui est une table hachée.

Question 3. (3 points)
Une liste peut être non vide ou vide.

- Une liste non vide est représentée par une instance de la classe *Liste*, qui a deux attributs : sa tête et sa queue. La tête d'une liste est le premier élément de la liste, la queue d'une liste est la liste des éléments autres que le premier.

- La liste vide est représentée par une instance de la classe *Empty*.

Définir les constructeurs des classes *Empty* et *Liste*. Le constructeur *List (ListeAbstraite l)* a en paramètre une liste dont le premier élément contient la tête de la liste et le deuxième élément contient la queue de la liste à construire. Si la liste en paramètre n'a pas exactement 2 éléments, il faut lever une exception.

Question 4. (3 points)
Définir les méthodes *nbElements()* et *toString()* des classes *Liste* et *Empty*. L'appel de *toString()* pour la classe *Empty* retourne la chaîne "empty", et pour la classe *Liste* retourne entre parenthèses la mise en chaîne de caractères de tous les éléments de la liste.

Définir la méthode *evaluer()* des classes *Liste* et *Empty*.

- *Empty* : L'appel de *evaluer()* pour la classe *Empty* retourne l'instance elle-même.
- *Liste* : Si le premier terme de la liste est un symbole fonctionnel, c'est-à-dire que c'est un symbole dont la valeur est une expression de type *Lambda*, l'évaluation de la liste est l'appel à l'évaluation de l'expression lambda associée au symbole avec comme paramètre la queue de la liste. Dans tous les autres cas l'évaluation retourne la liste des évaluations de chaque terme de la liste.

Les classes dérivées de la classe *Predefini* sont des classes permettant de représenter des instance de fonctions de base du langage *MicroLisp*.

Question 5. (3 points)
La classe *Cons* permet de représenter la fonction *cons* qui construit une liste à partir de son premier élément et de la liste qui sera la queue de la liste créée ; `(cons 1 (2 3 4))` a pour valeur la liste `(1 2 3 4)`
Définir pour la class *Cons* :

- le constructeur : le constructeur a en paramètre une liste dont le premier élément est le premier argument de cons et le deuxième élément est le deuxième argument de cons. Si la liste en paramètre n'a pas exactement 2 éléments, il faut lever une exception.
- la méthode *toString()* : retourne, entre parenthèses, la chaîne "cons", un espace, la mise en chaîne du premier argument, un espace, et la mise en chaîne du deuxième argument.
- la méthode *evaluer()* : la méthode *evaluer()* retourne une liste dont le premier élément est l'évaluation de *premier* et la liste de queue est l'évaluation de *fin*. Si l'évaluation de *fin* retourne une expression qui n'est pas une *ListeAbstraite*, il faut lever une exception.

Question 6. (3 points)
La classe *Car* permet de représenter la fonction *car* qui sélectionne le premier élément d'une liste.
Définir pour la class *Car* :

- le constructeur : le constructeur a en paramètre une liste dont le premier élément est l'argument de car. Si la liste en paramètre n'a pas exactement 1 élément, il faut lever une exception.
- la méthode *toString()* : retourne, entre parenthèses, la chaîne "car", un espace, la mise en chaîne de l'argument.

- la méthode *evaluer()* : la méthode *evaluer()* retourne le premier élément de l'évaluation de la liste. L'évaluation de la liste doit retourner une liste non vide, sinon on lève une exception.

Question 7. (3 points)
La classe *Define* permet de définir une valeur pour un symbole.
Définir pour la class *Define*:

- le constructeur : le constructeur a en paramètre une liste dont le premier élément est le symbole et le deuxième élément est la valeur associée au symbole. Si la liste en paramètre n'a pas exactement 2 éléments, il faut lever une exception.
- la méthode *toString()* : retourne, entre parenthèses, la chaîne "define", un espace, la mise en chaîne de l'attribut *symbole*, un espace, et la mise en chaîne de l'attribut *valeur*.
- la méthode *evaluer()* : la méthode *evaluer()* commence par stocker dans la variable de classe *environnement* le couple (évaluation du symbole, valeur) puis retourne l'objet lui-même. Si l'évaluation du symbole ne retourne pas un objet de la classe *Symbole*, on lève une exception.

Question 8. (3 points)
La classe *Lambda* permet de définir une fonction et sa liste de paramètres.
Définir pour la class *Lambda* :

- le constructeur : le constructeur a en paramètre une liste dont le premier élément est la liste des paramètres (cette liste ne peut pas être vide), et le deuxième élément est le corps de la fonction. Si la liste en paramètre du constructeur n'a pas exactement 2 éléments, il faut lever une exception.
- la méthode *toString()* : retourne, entre parenthèses, la chaîne "lambda", un espace, la mise en chaîne de la liste des paramètres, un espace, et la mise en chaîne de l'attribut *corps*.
- la méthode *evaluer()* : la méthode retourne l'objet lui-même.
- la méthode *evaluer(ListeAbstraite p)* : retourne l'évaluation du corps de la fonction, en remplaçant les paramètres formels par les paramètres effectifs. La liste *p* est la liste des paramètres effectifs. Pour réaliser l'association paramètres formels-paramètres effectifs, on utilise une table hachée, initialisée avec l'environnement courant (contenu dans l'attribut *environnement*), puis on ajoute dans cette table hachée tous les couples paramètre formel-paramètre effectif, puis on retourne l'évaluation du corps dans ce nouvel environnement local. Il faut ensuite restaurer l'environnement tel qu'il était avant l'appel de *evaluer*.

```
abstract public class Expression {
  public static  HashMap environnement = null;
  public Expression()
  abstract public String toString();
  abstract public Expression evaluer();
}
```

```
public abstract class Atome
     extends Expression {
}
```

```
public class Entier extends Atome {
  int val;
  public Entier(int v)
  public Expression evaluer()
  public String toString}
```

```
public class Chaine extends Atome {
  String val;
  public Chaine(String v)
  public Expression evaluer()
  public String toString()
}
```

```
public class Symbole
    extends Atome {
String nom;
  public Symbole(String n)
  public String toString()
  public Expression evaluer()
}
```

```
public abstract class ListeAbstraite extends Expression {
  abstract int nbElements();
}
```

```
public class Liste
    extends ListeAbstraite {
  Expression tete;
  ListeAbstraite queue;

  public Liste( ListeAbstraite l)
     throws Exception
  public Liste(Expression tete,
            ListeAbstraite queue)
  int nbElements()
  public Expression evaluer()
  public String toString()
}
```

```
public class Empty
extends ListeAbstraite {
  public Empty()
  int nbElements()
  public Expression evaluer()
  public String toString()
}
```

```
public class Cons
    extends Predefini {
  Expression premier;
  Expression fin;
  public Cons(ListeAbstraite l)
    throws Exception
  public Expression evaluer()
  public String toString()
}
```

```
public class Car
    extends Predefini {
  Expression liste;
  public Car(ListeAbstraite l)
    throws Exception
  public Expression evaluer()
  public String toString()
}
```

```
public class Define extends Predefini {
  Expression symbole;
  Expression valeur;
  public Define(ListeAbstraite l)
        throws Exception
  public Expression evaluer()
  public String toString()}
```

```
public class Lambda
    extends Predefini {
  Liste params;
  Expression corps;
  public Lambda(ListeAbstraite p)throws Exception
  public Expression evaluer()
  public Expression evaluer(ListeAbstraite paramsE)
                    throws Exception
  public String toString()
}
```

Principaux constructeurs et méthodes de la classe *HashMap* :

`HashMap()`	Construit un *HashMap* avec une capacité initiale de 11 et un facteur de charge de 0.75
`HashMap (Map c)`	Construit une table hachée à partir d'une autre table
`void` `clear()`	Enlève tous les objets de la table.
`boolean` `containsKey(Object cle)`	Retourne *true* si la table contient la clé
`boolean` `containsValue(Object valeur)`	Retourne *true* si la table contient la valeur.
`Object get( Object cle)`	Retourne la valeur associée à la clé.
`Object put( Object cle, Object valeur)`	Ajoute une valeur et une clé à la table si la clé n'est pas déjà présente, et dans ce cas retourne *null*, sinon modifie la valeur associée à la clé, et retourne l'ancienne valeur associée à la clé.
`boolean` `isEmpty()`	Retourne *true* si la table est vide et *false* sinon.
`int` `size()`	Retourne le nombre d'éléments présents dans la table