Année **2022-2023** Session **1**

Systèmes de Types et Programmation
IT234

Filière : Informatique, 2ème Année

Date de l'examen : 24/05/2023 Durée de l'examen : 2h00

Sujet de : D. Renault

Documents ☐ autorisés ☒ non autorisés Calculatrice ☐ autorisée ☒ non autorisée

Cet examen se présente comme un ensemble de questions courtes, demandant des réponses courtes (même pour le code). La précision des réponses fournies est un critère important de l'évaluation. La syntaxe des langages de programmation utilisés ne sera pas vérifiée à la lettre. Le barême est purement indicatif.

Exercice 1: 2 points

Donner une définition du terme *polymorphisme* dans le contexte des langages de programmation. Illustrer cette définition par deux exemples différents.

✓ Selon le cours, une expression d'un langage de programation est dite *polymorphe* si elle peut être typée par plusieurs types différents. Selon les formes que prennent ces différents types, on parle de familles de polymorphisme. Le polymorphisme *paramétrique* est caractérisé par le fait que les types partagent des paramètres communs. Par exemple, List[T] en Scala représente l'ensemble des types de listes. Le polymorphisme d'*inclusion* est caractérisé par des familles de types partageant des relations d'inclusion (en tant qu'ensemble de valeurs). Par exemple, Cloneable en Java représente l'ensemble des types contenant une méthode `clone`.

Exercice 2: 4 points

Le langage Ruby dispose depuis 2020 d'un langage de spécification des types nommé RBS (cf. `https://github.com/ruby/rbs`), activement maintenu (la version 3.1.0 date d'avril 2023). A l'intérieur de ce langage, sont en particulier définis des types paramétriques. Ci-joint un extrait (à peine simplifié) de la documentation [1] :

1. cf. `https://github.com/ruby/rbs/blob/master/docs/syntax.md#generics`

For classes with type parameters, you may specify if they are "invariant" (default), "covariant" (out) or "contravariant" (in).
For example, an Array of String can be considered to be an Array of Object, but not the reverse, so we can think of :

```
# The 'T' type parameter is covariant.
class Array[out T]
  # etc.
end
```

There's a limitation with this for mutable objects (like arrays) : a mutation could invalidate this.

1. À partir de cet exemple, expliquer le concept de *covariance* du point de vue des types.
2. Comment expliquer la dernière phrase « a mutation could invalidate this » ?
3. Cette invalidation, si elle existe, est-elle détectée à la compilation ou à l'exécution ?

✓

1. La covariance est une propriété indiquant les relations de sous-typage pouvant exister entre divers types appartenant au même type paramétré. Dans l'exemple, Array[T] est un type paramétré par T, et on considère les membres Array[Object] et Array[String]. Dans cet exemple, la covariance indique que :

$$U <: V \quad \Rightarrow \quad \text{Array}[U] <: \text{Array}[V]$$

2. La covariance est une propriété qui ne cohabite pas bien avec la mutabilité. Typiquement à cause des exemples de la forme :

```
# @type var x: Array[String]
x = [ "a", "b" ]
# x.push(5)    # Error : Cannot pass a value of type `Integer` as an argument of type `String`

# @type var y: Array[Object]
y = x           # Valid because of covariance
y.push(5)       # Sould give an error, because mixing `Integer` and `String` inside `x`
puts x.inspect # → ["a", "b", 5]
```

   Les tableaux étant mutables, le fait de les rendre aussi covariants rend le code *unsafe*. Dans l'exemple ci-dessus, le tableau `x` finit par contenir des valeurs comme `5` qui sont incompatibles avec son type. Le code précédent est actuellement accepté par le type checker, mais les langages comme Java et C# n'autorisent pas le mélange des deux propriétés.
3. La covariance est une propriété sur les relations de typage, et donc amène à des erreurs à la compilation. Quand bien même on parle ici de mutations, dont le résultat est dynamique, c'est le bien compilateur les considère comme des erreurs avant toute exécution du code.

Exercice 3: 4 points

Le compilateur Typescript considère le code suivant comme étant typé correctement. Pourtant, la méthode `concat` n'existe que sur les valeurs de type string, et la méthode `repeat` prend en paramètre un number.

```
1  function padLeft(padding: number | string, input: string) : string {
2      if (typeof padding === "number") {
3          return "␣".repeat(padding).concat(input);
4      } else {
5          return padding.concat(input);
6      }
7  }
8
9  padLeft(5,       "x"); // → "    x"
10 padLeft("-----", "x"); // → "-----x"
```

Toute variation du même code échangeant les branches ou n'en gardant qu'une seule amène à une erreur de compilation. La documentation appelle le test fait à la ligne 2 une *garde de type* (*type guard*), et cette propriété du système de type un *rétrécissement* (*narrowing*).

1. Comment appelle t'on le type "`number | string`" ?
2. Avec votre connaissance de l'algorithme d'inférence de type vu en cours pour le lambda-calcul, proposez une explication de l'algorithme utilisé par le compilateur Typescript pour vérifier un tel code.
3. Expliquer ce qui, dans ce code, est vérifié statiquement, et ce qui est vérifié dynamiquement (du point de vue des types).
   En quoi est-ce plus intéressant qu'une vérification purement statique ?

✓

1. Il s'agit d'un *type union* (*union type*), représentant l'ensemble des valeurs apparaissant dans l'union des deux types (et donc ici à la fois tous les nombres et toutes les chaînes de caractères).
2. Si on se réfère à l'algorithme vu en cours, celui-ci procède en deux étapes : premièrement, la constitution d'une liste de contraintes sur les types utilisés, et deuxièmement, une unification de ces contraintes.

   Dans l'exemple précédent, au début de la fonction (ligne 1), la contrainte sur la variable `padding` est d'être du type `number | string`. A la ligne 3 (resp. l.5), cette contrainte se précise, et devient de type `number` (resp. `string`).

   Le compilateur doit dont être capable de gérer des contraintes différentes pour les variables selon les blocs où elles apparaissent. Une propriété équivalente est vue en cours dans le cas du *let-in polymorphism*.
3. Dans l'exemple, la fonction complète est vérifiée statiquement (en utilisant un algorithme comme ci-dessus), mais il existe une part du code qui est vérifiée dynamiquement, à savoir la garde de type. Le test est effectué à l'exécution.

   Une vérification purement statique obligerait ici le compilateur à connaître à l'avance les types de tous les paramètres que l'on peut passer à `padLeft`. Il y a évidemment des cas où c'est possible (penser à de la surcharge, par exemple en C++), mais il existe des contextes dans lesquels c'est déraisonnable (penser à une entrée utilisateur). La possibilité de préciser dynamiquement les types (*narrowing*) permet donc plus de flexibilité.

Exercice 4: 4 points

Le code suivant en Go[2] est un exemple d'utilisation des interfaces dans un cadre culinaire.

Remarque : les noms des variables en Go sont placés *avant* leur type.

2. Inspiré de `https://duncanleung.com/understand-go-golang-interfaces`.

```go
// The type of electrical devices
type Blender   struct {}
type Microwave struct {}

func (b Blender) Draw(power int)     { fmt.Println("Welcome to the mix !") }
func (b Blender) Mix()               { fmt.Println("Making some orange juice") }

func (m Microwave) Draw(power int)   { fmt.Println("Heater powered at", power) }
func (m Microwave) Heat()            { fmt.Println("Heating some noodles") }

// The type of the power socket
type PowerSocket struct { }

// Appliance represents an electrical device that can draw power.
type Appliance interface { Draw(power int) }

// Plug in any Appliance device to the power socket
func (s PowerSocket) PlugIn(appliance Appliance, name string) {
     fmt.Println("Pluggin", name)
     appliance.Draw(100)
}

func main() {
     var wallOutlet PowerSocket = PowerSocket{}
     var blender Blender        = Blender{}
     var microwave Microwave    = Microwave{}
     wallOutlet.PlugIn(blender, "a blender")      // Log : "Pluggin a blender", "Welcome to the mix !"
     blender.Mix()                                // Log : "Making some orange juice"
     wallOutlet.PlugIn(microwave, "a microwave")  // Log : "Pluggin a microwave", "Heater powered at 100"
     microwave.Heat()                             // Log : "Heating some noodles"
}

// Another not related type
type Circle    struct {}
func (c Circle) Draw(scale int)      { fmt.Println("Drawing a circle at scale", scale) }
```

1. Quel type de polymorphisme est en jeu ici ?
2. Quelles relations de sous-typage interviennent dans ce code ?

Les interfaces en Go fournissent un moyen de travailler avec des types abstraits en connaissant simplement le comportement de ce type. La forme de typage utilisée ici est appelée *typage structurel*, mais on trouve aussi en anglais le terme « *compile-time duck typing* »[3], qui est souvent utilisée à côté de l'expression :

> Si cela marche comme un canard et cancane comme un canard,
> alors ce doit être un canard.

3. En quoi le code précédent est-il différent de ce qu'on pourrait écrire dans un langage comme Java ?
4. En quoi est-ce une propriété intéressante du système de types ? N'apporte t'elle que des avantages ?

3. Aucune traduction en français raisonnable n'a été trouvée.

✓

1. Le code met en jeu des enregistrements (*struct*) partageant des relations type/sous-type. Il s'agit de polymorphisme d'inclusion. La question peut se poser s'il s'agit de surcharge ou de redéfinition pour la méthode `Draw`. Mais son utilisation dans la méthode `PlugIn` se fait sur le type apparent `Appliance`, et donc la sélection de méthode se fait dynamiquement. Il s'agit donc de redéfinition.
2. Blender <: Appliance et Microwave <: Appliance.
3. En Java, écrire le même code imposerait de spécifier de manière explicite les relations type/sous-type précédentes. On parle aussi de typage *nominal*.
4. C'est une propriété intéressante du système de types, parce qu'elle permet au programmeur de ne pas écrire explicitement l'ensemble des interfaces implémentées par une classe donnée. Dans l'exemple ci-dessus, la méthode `PlugIn` est une méthode générique dépendant de l'interface `Appliance` (et il est nécessaire d'écrire cette interface). Mais les classes concrètes `Blender` et `Microwave` n'ont pas besoin de l'implémenter explicitement. Plus généralement, le typage structurel facilite l'apparition de relations type/sous-type, et donc favorise la généricité en permettant d'utiliser des codes n'ayant pas connaissance des interfaces à implémenter.

   A contrario, la relation de sous-typage se fait uniquement sur les prototypes des méthodes, et donc il existe toujours un risque que deux méthodes avec un même prototype n'aient pas vocation à être utilisées par la même fonction. Dans le code ci-dessus, le système de type ne peut pas empêcher d'appliquer la méthode `PlugIn` à un `Circle`, alors même qu'il n'y a pas de raison logique de le faire.

Exercice 5: 2 points

Inférer le prototype le plus générique possible pour la fonction suivante `filter` en Java, et proposer un morceau de code réalisant une application correcte de cette fonction.

```
______ filter( ______ aFunction,  ______ aList) {
     ______ aResult = new ArrayList<>();
    for ( ______ anElement : aList)
        if (aFunction.test(anElement))
            aResult.add(anElement);
    return aResult;
}
```

Les symboles "_____" indiquent les endroits où des types ont été effacés.

✓ Le prototype `<U> List<U> filter(Predicate<U> f, List<U> l)` est un prototype raisonnable pour cette fonction. Pour un bonus, il est envisageable d'utiliser les jokers avec un prototype comme `<U> List<U> filter(Predicate<? super U> f, List<? extends U> l)`. Il devient alors possible de l'appliquer avec des lambdas, par exemple sur des nombres :

```
filter((x) → x % 2 == 0, Arrays.asList(1, 2, 3, 4));
```

qui renvoie une liste ne contenant que $[2; 4]$. Si l'on ne connaît pas l'interface fonctionnelle `Predicate`, il est toujours possible de proposer une interface dans le style suivant :

```
interface Testable<T> {
    boolean test(T x);
}
static <U> List<U> filter(Testable<U> f, List<U> l)
```

Les propositions à base de `Object` sont forcément moins précises du point de vue des types (même si elles sont génériques).

Exercice 6: 4 points

Le code suivant en C++ [4] permet de calculer les racines d'un polynôme du second degré en utilisant un type particulier nommé `std::variant`. Ce type est usuellement considéré comme étant une forme de *runtime polymorphism* en C++.

4. Exemple tiré d'un article du site Walletfox `https://www.walletfox.com/course/patternmatchingcpp17.php`

```cpp
using roots_t = std::variant<
  std::pair<double,double>,    // two real roots
  double,                      // one double real root
  std::monostate               // no real roots
>;

roots_t compute_roots(double a, double b, double c){
    auto d = b*b-4*a*c; // Value of discriminant
    if (d > 0.0) {
        auto p = sqrt(d) / (2*a);
        return std::make_pair(-b + p, -b - p);
    } else if (d == 0.0)
        return (-1*b)/(2*a);
    else
        return std::monostate();
}

struct PrintVisitor {
  void operator()(const std::pair<double,double>& arg) {
    std::cout << "2 roots found: " << arg.first << " " << arg.second << std::endl;
  };
  void operator()(double arg) {
    std::cout << "1 root found: " << arg << std::endl;
  };
  void operator()(std::monostate) {
    std::cout << "No real roots found" << std::endl;
  };
};

void print_roots(const roots_t& v){ std::visit(PrintVisitor(), v); }

int main(void) {
  struct PrintVisitor a_visitor;
  a_visitor(std::make_pair(4,8));       // 2 roots found: 4 8
  a_visitor(12345678);                  // 1 root found: 1.23457e+07
  a_visitor(std::monostate());          // No real roots found
  print_roots(compute_roots(1,0,-1));   // 2 roots found: 1 -1
  print_roots(compute_roots(1,-2,-2)); // 2 roots found: 3.73205 0.267949
  print_roots(compute_roots(1,6,9));    // 1 root found: -3
  print_roots(compute_roots(1,-3,4));   // No real roots found
}
```

Remarque : le type `std::monostate` peut être considéré dans cet exercice comme l'équivalent de `unit` en OCaml[5].

1. Écrire le type OCaml équivalent au type `roots_t`.
   Quel nom porte un tel type ? Quel est le nom de la technique décrite ici ?

La structure `struct PrintVisitor` définie dans les lignes 18–28 possède trois méthodes. Son écriture fait qu'elle est utilisable comme une fonction, prenant des paramètres de types potentiellement différents. Un exemple d'utilisation est donné aux lignes 33–36.

2. Quelle est la forme de polymorphisme en jeu ici ? A quel moment de la vie du code la sélection de la méthode à appliquer est-il fait ? En quoi cela peut-il être considéré comme étonnant pour du code C++ ?
3. Proposer des avantages et les inconvénients de ce genre de construction.

5. La réalité est légèrement différente, cf. https://en.cppreference.com/w/cpp/utility/variant/monostate

✓

1. Le type OCaml associé peut s'écrire de la manière suivante :

```
type roots = Pair of float*float | Single of float | None of unit
```

   Il s'agit d'un *type algébrique*, aussi appelé type *variant* ou type *somme*. La technique décrite ici est le *filtrage de motif* ou *pattern-matching* (donné dans l'URL de l'article). Ces exemples sont à rapprocher de ceux vus en cours pour décrire les expressions du lambda-calcul.

2. Il s'agit de polymorphisme d'inclusion, et plus précisément de redéfinition. La différence entre redéfinition et surcharge n'était pas clairement identifiable avec l'énoncé, et donc les deux réponses ont été acceptées.

   La clé de la compréhension de cette question est la fonction `print_roots`, qui prend un `roots_t` en paramètre. Au vu des lignes 37 à 40, cette fonction accepte n'importe quel résultat de `compute_roots`. Il ne peut donc y avoir de sélection statique, ou alors cela signifierait que le compilateur exécute le code de `compute_roots`.

   La sélection de méthode est donc faite à l'exécution, comme le suggère l'expression *runtime polymorphism* de l'énoncé. C'est étonnant en C++, parce que la redéfinition et les autres mécanismes dynamiques se font usuellement de manière explicite (e.g avec le mot-clé `virtual`). Ici, l'utilisation du pattern "visiteur" laisse un indice assez fort.

   Si on a compris à la question précédente qu'il s'agissait de pattern-matching, et qu'on comprend comment le mécanisme fonctionne en OCaml (en gros un switch), on avait une meilleure intuition du fonctionnement de `std::visit`.

3. Il s'agit d'une construction intéressante, même si elle va à revers de certaines idées du C++ (comme le côté dynamique semi-explicite). Les types variants permettent de réunir des types sans relation les uns avec les autres, au contraire des arbres d'héritage classiques (on parle de polymorphisme adhoc). Aussi, il est facile de construire de nouveaux visiteurs, qui jouent le rôle de méthodes particulières pour les valeurs typées par `std::variant`, cela sans modifier les types existant.

   A contrario, l'écriture d'un variant fixe de manière forte (statiquement) la liste des types autorisés dans le variant, rendant difficile l'ajout d'autres possibilités. De plus, les méthodes écrites dans les visiteurs sont appliquables à des variants particuliers, et leur réutilisation pour d'autres variants est certainement malaisée.