Année **2025-2026** Session **CC**

Systèmes de Types et Programmation
EIN6-PROG1

Filière : Informatique, 2ème Année

Date de l'examen : 12/05/2026 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: 3 points

Le type *option* est un type utilisé pour représenter une valeur qui peut soit être concrète, soit prendre une valeur par défaut. En OCaml, une valeur de type `int option` peut être soit `None`, soit `Some 7`. En Java, une valeur de type `Optional<Integer>` peut être soit `Optional.empty()`, soit `Optional.of(7)`. Les deux codes suivants fournissent la documentation de la fonction `map` dans ces deux langages :

— En OCaml :

```
type 'a option = None | Some of 'a
val map : ('a → 'b) → 'a option → 'b option
```

"`map f o` is `None` if `o` is `None` and `Some (f v)` if `o` is `Some v`."

— En Java :

```
class Optional<T> {
  <U> Optional<U> map(Function<? super T,? extends U> mapper);
}
```

"If a value is present, apply the provided mapping function to it, and if the result is non-null, return an `Optional` describing the result."

1. Quelle grande famille de polymorphisme est en jeu dans ces deux exemples ?
2. Expliquer pourquoi les types en Java contiennent-ils des raffinements par rapport aux types en OCaml ?

✓

1. Il s'agit de *polymorphisme paramétrique* ou *universel* (le terme *generics* était considéré comme acceptable mais moins parlant).
2. Les types en Java prennent en compte les possibilités de sous-typage, le langage étant de manière inhérente très orienté objet. Et donc en Java, on voit apparaître des indications de *variance*, sous la forme de *wildcards*. Ces indications sont donc la marque de l'interaction entre le polymorphisme paramétrique et le polymorphisme d'inclusion.

Exercice 2: 6 points

L'ORM Prisma (`https://www.prisma.io/`) est un *object-relational mapping tool*, à savoir une bibliothèque permettant de traduire les données à l'intérieur d'une base de données en valeurs utilisables dans un langage de programmation. Il est développé en Javascript/Typescript. Concrètement, il part d'un fichier de configuration qui décrit le schéma de la base de données, par exemple :

```
model Post {
  id        Int      @id @default(autoincrement())
  title     String
  content   String?
  published Boolean  @default(false)
  createdAt DateTime @default(now())
}
```

À partir de ce modèle, la commande `npx prisma generate` permet de générer un certain nombre de fichiers (notés `@prisma/client` dans le code suivant), et il devient possible d'écrire du code (ici en Typescript) correctement typé de la forme suivante :

```
1  import { Prisma, PrismaClient } from '@prisma/client';
2  import type { Post } from '@prisma/client';
3  import { expectTypeOf } from 'vitest';
4
5  const prisma = new PrismaClient({ /* ... */ });
6
7  async function main() {
8      // Create a post
9      const post : Post = await prisma.post.create({
10         data: {
11             title: 'First Post',
12             content: 'This is a sample post.',
13         } as Prisma.PostCreateInput,
14     })
15     expectTypeOf(post).toMatchTypeOf<{ id: number, title: string, content: string | null }>();
16     console.log('Created Post:', post)
17     // Find/Read a post
18     const posts : Post[] = await prisma.post.findMany({
19         where: { id: { gte: 5, lte: 9 } } as Prisma.PostWhereInput
20     })
21     console.log('All Posts:', posts)
22     // Update a post
23     const updated : Post = await prisma.post.update({
24         where: { id: post.id } as Prisma.PostWhereUniqueInput,
25         data: { published: true } as Prisma.PostUpdateInput,
26     })
27     console.log('Updated Post:', updated)
28     // Delete a post
29     const deleted : Post = await prisma.post.delete({
30         where: { id: post.id } as Prisma.PostWhereUniqueInput,
31     })
32     console.log('Deleted Post:', deleted)
33 }
```

La bibliothèque annonce qu'elle assure une forme de sûreté de typage *type-safety*[1], sans entrer dans les détails. Néanmoins, à l'utilisation, toute requête faite à travers cette interface sur des types incompatibles se verra rejeter par une erreur de compilation. Pour aider à la compréhension, la l.15 donne une assertion sur le type envoyé par `prisma.post.create`.

1. Proposer un type en Typescript pour le type `Post` utilisé entre autres l.9.

Les deux codes suivants (nommés A et B) sont des modifications légères du code précédent :

1. Cf. https://www.prisma.io/docs/orm/prisma-client/type-safety

Code A

```
const post : Post = await prisma.post.create({
    data: {
        // title: 'First Post',
        nonsensical: true,       // Superfluous key
        content: 'This is a sample post.',
    } as Prisma.PostCreateInput,
})
```

Ce code ne compile pas avec le message :

```
Property 'title' is missing in type
'{ nonsensical: boolean; content: string; }'
```

Code B

```
const post : Post = await prisma.post.create({
    data: {
        title: 'First Post',
        nonsensical: true,       // Superfluous key
        content: 'This is a sample post.',
    } as Prisma.PostCreateInput,
})
```

Ce code compile sans erreurs.

2. Pouvez-vous expliquer pourquoi il y a une erreur de compilation dans un cas et pourquoi dans l'autre le code est-il valide ?

Les fonctions fournies par la bibliothèque Prisma sont des fonctions classiques d'ORM suivant le motif CRUD (*create*/*read*/*update*/*delete*). Pour une base de données concrète, on aurait ainsi (au moins) 4 fonctions par table.

3. Quel type proposeriez-vous pour la fonction `delete` utilisée l.29 ?
   (*Remarque* : il est demandé d'utiliser au maximum les types concrets (et donc pas `PostWhereUniqueInput`), mais on pourra réutiliser le type `Post` de la question 1.
4. Dans un contexte avec plus d'une table, est-ce que la présence de multiples implémentations de la fonction `delete` est comparable à du polymorphisme de surcharge ? Comment la sélection de la méthode est-elle faite ?
5. Le site utilise l'expression "*zero-cost type-safety*" pour mettre en valeur sa bibliothèque. Pouvez-vous expliquer ce que cela signifie à votre avis ? Connaissez-vous un contexte de programmation dans lequel l'utilisation des types ne serait pas "*zero-cost*" ?

✓

1. Au vu de la configuration, de l'utilisation qui est faite dans le code aux lignes 9–14 pour créer une valeur de type `Post`, et de l'assertion de type l.15, le type suivant convient :

```
type Post = {
    id: number,
    title: string,
    content: string | null,
    published: boolean,
    createdAt: DateTime,
}
```

2. Dans le code A, le paramètre passé à la fonction `create` n'est pas suffisant pour créer une valeur de type `Post`, il manque un champ obligatoire nommé `title`. Plus précisément, la valeur passée en paramètre *n'est pas* un sous-type de `Post`. Alors que dans le code B, il s'agit bien d'un sous-type, quand bien même il existe une clé superflue qui n'est pas nécessaire. On retrouve la propriété de sous-typage des records vue en cours, aussi appelée "*qui peut le plus peut le moins*".
3. Au vu de l'utilisation de la fonction l.29–31, il semble que cette fonction soit de type `(PostWhereUniqueInput) ⇒ Post`. Si on veut rentrer dans les types concrets, on peut imaginer un type de la forme :

```
type DeleteType = ({
    id: number | null,
    title: string | null,
    content: string | null,
    published: boolean | null,
    createdAt: DateTime | null,
}) ⇒ Post
```

   Après, une réponse de la forme suivante est aussi acceptable :

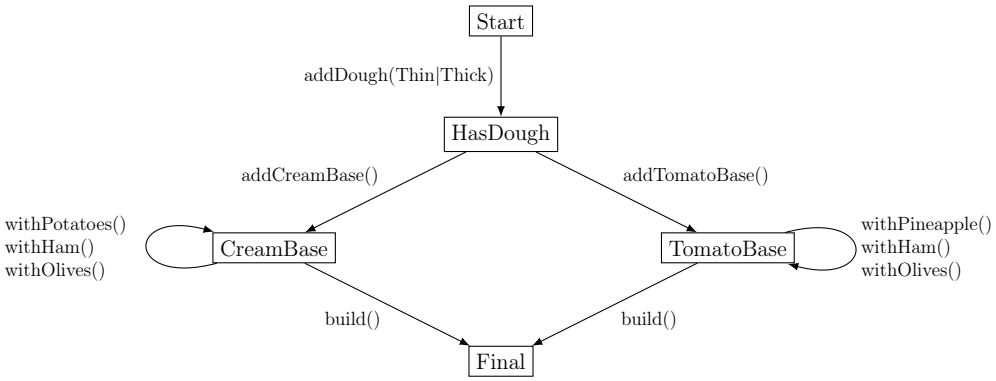
```
type DeleteType = (Post ⇒ boolean) ⇒ Post
```

4. Cette question est un piège, au sens où les différentes fonctions `delete` *ne réfèrent pas* au même identifiant dans le code. Par exemple, avec une table `post` et une table `user`, il y aurait deux identifiants distincts `prisma.post.delete` et `prisma.user.delete`. Ce n'est donc *pas* du polymorphisme de surcharge, mais simplement plusieurs fonctions différentes. Il n'y a donc pas besoin de sélection.
5. La bibliothèque Prisma ajoute manifestement une surcouche de types à son ORM pour assurer que le compilateur vérifie (autant que possible, au vu de la question 2) que les types des données soient corrects pour toutes les utilisations des fonctions de la bibliothèque. C'est le côté "*type-safety*". On pouvait se référer aux définitions du cours, mais manifestement la bibliothèque ne s'encombre pas de définitions formelles.

   Ensuite, le "*zero-cost*" se réfère sans doute au fait que cette vérification se fait à la compilation, et donc n'a pas d'impact à l'exécution des requêtes. De nombreux systèmes de vérification de types sont "zero-cost", puisqu'ils sont faits par un compilateur et donc avant l'exécution. Néanmoins, il est vrai que certaines vérifications des types sont aussi faites à l'exécution (et donc ont un coût). C'est le cas d'une part des langages "à typage dynamique", comme par exemple Python, et d'autre part nous avons vu en cours un exemple de surcoût pour la sélection dynamique (*late* ou *dynamic binding*) en polymorphisme objet.

Exercice 3: 7 points

Le motif de programmation (en anglais *pattern*) Monteur (en anglais *Builder*[2]) est un *design pattern* adapté à la création d'objets complexes en plusieurs étapes. Dans l'exemple suivant[3], il est utilisé pour modéliser la confection de pizzas, avec un algorithme de confection représenté par l'automate suivant :



Noter que l'auteur semble ne pas apprécier l'ananas sur ses pizzas à la crème. Pour implémenter une solution en Kotlin, le code suivant est proposé :

```kotlin
sealed interface Crust
data object ThinCrust  : Crust
data object ThickCrust : Crust

sealed interface BaseState
data object CreamBase  : BaseState
data object TomatoBase : BaseState

sealed interface Topping
data object Ham       : Topping
data object Olives    : Topping
data object Potatoes  : Topping
data object Pineapple : Topping

class DoughBuilder internal constructor (
    internal val crust: Crust
)

class PizzaBuilder<out S : BaseState> internal constructor (
    internal val crust: Crust,
    internal val base: BaseState,
    internal val toppings: List<Topping>,
)

fun dough(crust: Crust): DoughBuilder = DoughBuilder(crust)

fun DoughBuilder.withCreamBase() =                PizzaBuilder<CreamBase>(crust, CreamBase, emptyList())
fun DoughBuilder.withTomatoBase() =               PizzaBuilder<TomatoBase>(crust, TomatoBase, emptyList())
fun PizzaBuilder<CreamBase>.withPotatoes() =      PizzaBuilder<CreamBase>(crust, base, toppings + Potatoes)
fun PizzaBuilder<TomatoBase>.withPineapple() =    PizzaBuilder<TomatoBase>(crust, base, toppings + Pineapple)
fun <S : BaseState> PizzaBuilder<S>.withHam() =   PizzaBuilder<S>(crust, base, toppings + Ham)
fun <S : BaseState> PizzaBuilder<S>.withOlives() = PizzaBuilder<S>(crust, base, toppings + Olives)
fun <S : BaseState> PizzaBuilder<S>.build() =     Pizza(crust, base, toppings.toSet())
```

2. Cf. la référence Gamma E., Helm R., Johnson R., Vlissides J. (1994) *Design Patterns : Elements of Reusable Object-Oriented Software* Addison-Wesley ou https://refactoring.guru/design-patterns/builder

3. Tiré de l'article de blog *"Making illegal state unrepresentable"* de N. Fränkel, disponible à https://blog.frankel.ch/illegal-state-unrepresentable/.

Pour aider à la compréhension, ce code Kotlin est écrit en spécifiant :

1. d'abord les types de base (l. 1–13),
2. ensuite les classes principales (`DoughBuilder` et `PizzaBuilder`, l. 15–23) en indiquant uniquement leurs attributs,
3. et enfin des *fonctions d'extension* (l. 25–33), une technique pour ajouter une méthode à une classe après sa définition.

Une fonction d'extension est toujours appelée avec implicitement une variable `this` du type de la classe. Ainsi, les deux codes suivants peuvent être considérés comme équivalents :

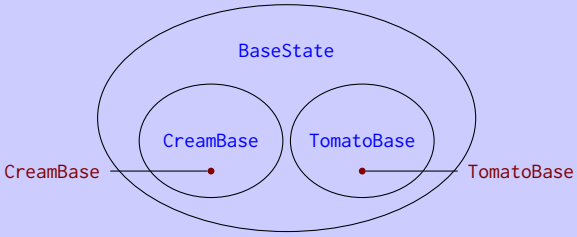
```
fun DoughBuilder.withCreamBase() =
  PizzaBuilder<CreamBase>(crust, CreamBase, emptyList())
```

```
class DoughBuilder {
    auto withCreamBase() {
        return new PizzaBuilder<CreamBase>(
            this.crust, CreamBase, emptyList());
    }
}
```

Enfin, considérer que la syntaxe des types en Kotlin est très proche de celle de Java.

1. Les lignes 5–7 du code définissent un type nommé `BaseState`. Quel est le sens des mot-clés `interface` et surtout `object` dans cette définition ?
2. A la ligne 27, l'identifiant `CreamBase` est-il utilisé comme une valeur, un type, ou les deux ? Justifier.
3. Quelle construction de type vue en cours vous évoque la définition de `CreamBase` ? Dessiner les relations types / sous-types correspondant à cette définition, en représentant les types et les valeurs.
4. La définition de la classe `PizzaBuilder` à la ligne 19 contient deux notations intéressantes dans le code entre chevrons. Pour commencer, à quel concept vu en cours correspond la partie avec le `S : BaseState` ? Justifier.
5. A quel concept vu en cours correspond le mot-clé `out` ? Justifier en expliquant les relations entre les types qu'il induit.
6. En quoi le code proposé permet-il d'implémenter le problème initial de confection de pizza ? En vous appuyant sur le code fourni, vous pourrez proposer des exemples de codes bien typés et des exemples mal typés.
7. Quelle construction de type vue en cours ce code met-il en valeur au final ?

1. `BaseState` est une définition de type dans un langage pour lequel les types sont proches de ceux de Java. Il s'agit donc en même temps d'une interface (a priori sans méthodes), comme le souligne le mot-clé utilisé. A côté de cela, le mot-clé `object` (renforcé par le mot-clé `data`) est une manière de définir un *objet immédiat* du type de cette interface (et non une classe comme on le ferait en Java).
2. À la ligne 27, l'identifiant `CreamBase` est utilisé à la fois comme une valeur (dans les arguments de la fonction, juste à côté de `emptyList()`, mais aussi comme un type à l'intérieur des chevrons.
3. Cette construction ressemble très fortement aux types inductifs vus en OCaml. Dans ce cas, `CreamBase` est une valeur possible de ce type inductif, et aussi un sous-type de `BaseState`. Les relations types / sous-types prennent donc une forme comme la suivante, pour laquelle les types sont en bleu et les valeurs sont en rouge :



4. La notation `S : BaseState` est une forme de *polymorphisme paramétrique contraint* (aussi appelé *constrained polymorphism* en cours). Il s'agit d'une forme de polymorphisme universel, qui accepte n'importe quelle classe qui soit un sous-type de `BaseState` (dans le cas présent, `TomatoBase` et `CreamBase`, mais pas `Crust` ou `Topping`).
5. La notation `out` est une indication de *variance*, permettant de fixer les relations entre les types de la forme `PizzaBuilder<S>`. Dans le cas présent, il s'agit même d'une relation de *covariance*. Ainsi, `PizzaBuilder<CreamBase>` est un sous-type de `PizzaBuilder<BaseState>` (cf. aussi `https://kotlinlang.org/docs/generics.html#declaration-site-variance`).
6. Le code proposé contraint les types de `PizzaBuilder` dans les fonctions d'extension `withPineapple` et `withPotatoes`. Ainsi, le code suivant est bien typé :

   ```
   DoughBuilder(Crust).withCreamBase().withPotatoes().build
   ```

   Mais le code suivant ne type pas :

   ```
   DoughBuilder(Crust).withCreamBase().withPineapple().build
   ```

   En effet, le type du résultat de la fonction `withCreamBase` est un `PizzaBuilder<CreamBase>` qui est incompatible avec le type du paramètre de la fonction `withPineapple` (qui prend des `PizzaBuilder<TomatoBase>`).
7. Il s'agit d'un exemple de code contenant des types contraignant les possibilités du code. Des exemples ont été donnés en cours dans la partie *Proof with types*. Le type `PizzaBuilder<S>` utilise son paramètre `S` pour rendre certains appels de fonctions mal typés. Il ne s'agit pas réellement d'un *type fantôme* (même si la réponse était acceptée) parce que le type est aussi utilisé comme une valeur à l'exécution. Cette construction est en fait plus proche d'un *GADT* ou *generalized abstract data type*.

Exercice 4: 4 points

Le code suivant en C++-20 montre comment il est possible d'utiliser de la métaprogrammation pour définir des fonctions ne s'appliquant qu'à certains types de valeurs. Pour cela, il utilise une construction nommée *concept*, apparaissant ici aux lignes 13–19 :

```
1  // Custom struct
2  struct Person {
3    std::string name;
4    int age;
5  };
6
7  // Overload 'operator<<' for printing
8  std::ostream& operator<<(std::ostream& os, const Person& p) {
9    return os << "{ Name: " << p.name << ", Age: " << p.age << " }";
10 }
11
12 // An example of a concept
13 template <typename T>
14 concept OstreamInsertable =                    // A requirement, made of :
15   requires(std::ostream& os, const T& value)   // - some parameters
16 {                                              // - a 'requires' clause, made of :
17   { os << value }                              //    * a (non-evaluated) expression
18   -> std::same_as<std::ostream&>;              //    * a constraint on the return
19 };
20
21 // The 'serialize' function that outputs the 'value' on 'stdout'
22 template <OstreamInsertable T>
23 void serialize(const T& value) {
24   std::cout << "Serializing -> " << value << "\n";
25 }
26
27 int main() {
28   serialize(42);
29   serialize("Coucou");
30   serialize(Person{"Alice", 30});
31   return EXIT_SUCCESS;
32 }
```

Le code s'exécute sans erreur de compilation et son exécution affiche le texte suivant :

```
Serializing -> 42
Serializing -> Coucou
Serializing -> { Name: Alice, Age: 30 }
```

Néanmoins, si l'on commente les lignes 7–10, on obtient une erreur de compilation qui (si l'on résume, c'est une erreur de compilation de C++) revient à :

```
error: no matching function for call to 'serialize(const Person&)'
note: template argument deduction/substitution failed:
note: constraints not satisfied
```

1. Le concept définit un ensemble de contraintes sur le type `T`. Lesquelles ?
2. À quelle construction vue en cours cela vous fait-il penser ?
   Bonus : donner le type correspondant dans le langage de votre choix.
3. Par rapport aux constructions citées à la question précédente, en quoi le fait de ne spécifier ici qu'une *expression non évaluée* est-il remarquable ?

1. Au vu du code apparaissant dans les lignes 13–19, et des exemples de compilation, le concept vérifie la présence de l'opérateur `<<` sur le type `T`, et plus précisément qu'il existe une surcharge de l'opérateur `<<` du type :

   ```
   std::ostream& operator<<(std::ostream& os, const T& p)
   ```

2. Cette construction fait penser à la notion d'*interface* en programmation orientée objet, voire même de *classe de type* en Haskell, si l'on considère que l'on peut surcharger l'opéateur `<<` pour n'importe quel type en C++. La fonction `serialize` est générique mais ne peut s'appliquer qu'aux types possédant le bon opérateur. En Java, si on avait une interface `Printable<T>` pour l'opérateur `<<`, cela donnerait :

   ```
   String serialize<T extends Printable<T> >(T t);
   ```

   En Haskell, si on considère que la classe de type `Show a` est celle des types contenant l'opérateur `<<`, cela donnerait :

   ```
   serialize :: Show a ⇒ a → String
   ```

3. Classiquement, les interfaces ou les classes de types sont définies par la liste des prototypes de fonctions contraignant le type (`serialize` dans les exemples précédents). Ici, la contrainte n'est pas fixée par le prototype d'une fonction, mais par une expression placée dans un contexte et non-évaluée. Cela permet une manière différente d'exprimer les contraintes sur les types, directement basée sur la syntaxe d'utilisation des valeurs. L'exemple suivant, tiré de `https://en.cppreference.com/cpp/language/requires`, montre comment on peut imposer la contrainte d'avoir des valeurs échangeables (swappable) :

   ```
   template<class T, class U = T>
   concept Swappable = requires(T&& t, U&& u)
   {
       swap(std::forward<T>(t), std::forward<U>(u));
       swap(std::forward<U>(u), std::forward<T>(t));
   };
   ```

   . . . et donc la présence d'un opérateur `swap` avec les valeurs typées placées dans leur contexte. Dans cet exemple, l'expression complète, si elle était évaluée, n'aurait pas de sens, puisqu'elle échangerait deux fois les valeurs. Seule la possibilié de fait les échanges dans les deux sens compte.