Projet de programmation fonctionnelle et de traduction des langages

Année 2020/2021

Le but du projet de programmation fonctionnelle et de traduction des langages est d'étendre le compilateur du langage RAT réalisé en TP de traduction des langages pour traiter de nouvelles constructions : les **pointeurs**, la **surcharge des fonctions**, les **types énumérés** et la structure de contrôle **switch/case**.

Le compilateur sera écrit en OCaml et devra respecter les principes de la programmation fonctionnelle étudiés lors des cours, TD et TP de programmation fonctionnelle.

Table des matières

Préambule

- Le projet est à réaliser en binôme (même binôme qu'en TP).
- L'échange de code entre les différents binômes est interdit.
- Les sources fournies doivent compiler sur les machines des salles de TP.
- Les sources et le rapport sont à déposer sous Moodle avant le **jeudi 14 Janvier - 23h**.
- Les sources seront déposées sous forme d'une unique archive `<rat_xxx_yyy>.tar` où `xxx` et `yyy` sont les noms du binôme. Cette archive devra créer un répertoire `rat_xxx_yyy` (pensez à renommer le répertoire nommé `sourceEtu` donné en TP) contenant tous vos fichiers.
- le rapport (`rapport.pdf`) doit être dans le même répertoire, à la racine. Il n'est pas nécessairement long, mais doit expliquer les évolutions apportées au compilateur (voir section sur les critères d'évaluation).

1 Extension du langage RAT

Le compilateur demandé doit être capable de traiter le langage RAT étendu comme spécifié dans la figure 1.

Le nouveau langage permet de manipuler :

1. des pointeurs ;
2. la surcharge de fonctions ;
3. les types énumérés ;
4. la structure de contrôle switch/case.

Le choix a été fait d'avoir une grammaire lisible et simple, mais cela implique la présence d'un lourd parenthésage des expressions pour éviter les conflits au moment de la génération de l'analyseur syntaxique.

De même, pour simplifier la grammaire, les identifiants (*tid*) de type énuméré ainsi que leurs valeurs commencent par une lettre majuscule, alors que les autres identifiants (*id*) commencent par une minuscule.

1.1 Les pointeurs

RAT étendu permet de manipuler les pointeurs à l'aide d'une notation proche de celle de C :
— $A \rightarrow (* \ A)$: déréférencement : accès en lecture ou écriture à la valeur pointée par A ;
— $TYPE \rightarrow \ TYPE*$: type des pointeurs sur un type TYPE ;
— $E \rightarrow null$: pointeur null ;
— $E \rightarrow (new \ TYPE)$: initialisation d'un pointeur de type TYPE ;
— $E \rightarrow \& \ id$: accès à l'adresse d'une variable.

Le traitement des pointeurs a été étudié lors du dernier TD, il s'agit ici de coder le comportement défini en TD.

La libération de la mémoire n'est pas demandée.

Exemple de programme valide

```
main{
  int * px = (new int);
  int x = 3;
  px = &x;
  int y = (*px);
  print y;
}
```

1.2 La surcharge de fonctions

Pour l'instant dans RAT, il est impossible de définir deux fonctions de même nom même si le type des paramètres est différent. Modifiez le compilateur pour que la surcharge de fonctions soit autorisée. La surcharge sur le type de retour n'est pas demandée.

Il n'y a aucune modification de la grammaire.

1. ~~$MAIN \rightarrow PROG$~~
2. $MAIN \rightarrow ENUMS\ PROG$
3. $ENUMS \rightarrow \Lambda$
4. $ENUMS \rightarrow ENUM\ ENUMS$
5. $ENUM \rightarrow enum\ tid\ \{IDS\}\ ;$
6. $IDS \rightarrow tid$
7. $IDS \rightarrow tid\ ,\ IDS$
8. $PROG \rightarrow FUN\ PROG$
9. $FUN \rightarrow TYPE\ id\ (\ DP\ )\ \{IS\ return\ E\ ;\}$
10. $PROG \rightarrow\ id\ \ BLOC$
11. $BLOC \rightarrow\ \{\ IS\ \}$
12. $IS \rightarrow\ I\ \ IS$
13. $IS \rightarrow$
14. $I \rightarrow\ TYPE\ \ id\ =\ E\ ;$
15. $I \rightarrow\ A\ =\ E\ ;$
16. $I \rightarrow\ const\ \ id\ =\ entier\ ;$
17. $I \rightarrow\ print\ \ E;$
18. $I \rightarrow\ if\ \ E\ \ BLOC\ \ else\ \ BLOC$
19. $I \rightarrow\ while\ \ E\ \ BLOC$
20. $I \rightarrow switch\ (E)\ \{\ LCase\ \}$
21. $LCase \rightarrow Case\ LCase$
22. $LCase \rightarrow \Lambda$
23. $Case \rightarrow case\ tid\ :\ IS\ B$
24. $Case \rightarrow case\ entier\ :\ IS\ B$
25. $Case \rightarrow case\ true\ :\ IS\ B$
26. $Case \rightarrow case\ false\ :\ IS\ B$
27. $Case \rightarrow default\ :\ IS\ B$
28. $B \rightarrow \Lambda$
29. $B \rightarrow break;$
30. $A \rightarrow id$
31. $A \rightarrow (*\ A)$
32. $DP \rightarrow$
33. $DP \rightarrow TYPE\ id\ DP$
34. $TYPE \rightarrow\ bool$
35. $TYPE \rightarrow\ int$
36. $TYPE \rightarrow\ rat$
37. $TYPE \rightarrow\ TYPE\ *$
38. $TYPE \rightarrow\ tid$
39. $E \rightarrow call\ id\ (\ CP\ )$
40. $CP \rightarrow$
41. $CP \rightarrow E\ CP$
42. $E \rightarrow\ [\ E\ /\ E\ ]$
43. $E \rightarrow\ num\ \ E$
44. $E \rightarrow\ denom\ \ E$
45. ~~$E \rightarrow id$~~
46. $E \rightarrow\ true$
47. $E \rightarrow\ false$
48. $E \rightarrow\ entier$
49. $E \rightarrow\ (\ E\ +\ \ E\ )$
50. $E \rightarrow\ (\ E\ *\ \ E\ )$
51. $E \rightarrow\ (\ E\ =\ \ E\ )$
52. $E \rightarrow\ (\ E\ <\ \ E\ )$
53. $E \rightarrow A$
54. $E \rightarrow null$
55. $E \rightarrow (new\ TYPE)$
56. $E \rightarrow \&\ id$
57. $E \rightarrow tid$

FIGURE 1 – Grammaire du langage RAT étendu

Exemple de programme valide

```
int f (int i1 int i2) { return 1; }
int f (rat r1 rat r2) { return 2; }

main{
  int i = call f (1 2) ;
  print i;
  i = call f ([1/2] [1/3]);
  print i;
}
```

1.3 Les types énumérés

RAT étendu permet de définir des types énumérés à l'aide d'une notation proche de celle de C :

- $MAIN \rightarrow ENUMS\ PROG$: ajout de la possibilité de définir des types énumérés en début de programme RAT ;
- $ENUMS \rightarrow \Lambda$: fin de la liste des types énumérés ;
- $ENUMS \rightarrow ENUM\ ENUMS$: récursivité pour définir plusieurs types énumérés ;
- $ENUM \rightarrow enum\ tid\ \{IDS\}\ ;$: définition d'un type énuméré avec son identifiant et la liste de ses valeurs ;
- $IDS \rightarrow tid$: fin de la liste des valeurs (elle ne peut pas être vide) ;
- $IDS \rightarrow tid\ ,\ IDS$: récursivité pour définir plusieurs valeurs (séparées par des ",") ;
- $TYPE \rightarrow\ tid$: utilisation d'un type énuméré (déclaration de variable ou de paramètre, type retour d'une fonction)
- $E \rightarrow tid$: utilisation une valeur d'un type énuméré dans une expression.

Comme indiqué précédemment, les identifiants (tid) de type énuméré ainsi que leurs valeurs commencent par une lettre majuscule, alors que les autres identifiants (id) commencent par une minuscule.

Deux types énumérés ne peuvent pas avoir le même nom et une même valeur ne peut pas appartenir à deux types énumérés différents.

Il faudra surcharger l'opérateur "=" pour tester l'égalité de deux expressions de type énuméré.

Exemple de programme valide

```
enum Mois {Janvier, Fevrier, Mars, Avril, Mai, Juin, Juillet , Aout, Septembre, Octobre, Novembre, Decembre};
enum Jour {Lundi, Mardi, Mercredi, Jeudi, Vendredi, Samedi, Dimanche};

bool and (bool b1 bool b2){
  bool r = b1;
  if (r){r=b2;}else{}
  return r;
}

bool estDateRenduProjet (Jour j int d Mois m int a){
  bool b1 = (j=Jeudi);
```

```
  bool b2 = (d=14);
  bool b3 = (m=Janvier);
  bool b4 = (a=2021);
  return (call and (call and ( (call and (b1 b2))b3)b4));
}

test{
  print (call estDateRenduProjet (Mercredi 5 Novembre 2020) );
  print (call estDateRenduProjet (Jeudi 14 Janvier 2021) );
}
```

Ce programme doit afficher "falsetrue".

1.4 La structure de contrôle switch/case

RAT étendu permet de définir la structure de contrôle switch/case proche de celle de C :

- $I \rightarrow switch\ (E)\ \{\ LCase\ \}$: Instruction switch / case
- $LCase \rightarrow Case\ LCase$: Liste de cas
- $LCase \rightarrow \Lambda$
- $Case \rightarrow case\ tid\ :\ IS\ B$: Switch sur une valeur d'un type énuméré
- $Case \rightarrow case\ entier\ :\ IS\ B$: Switch sur un entier
- $Case \rightarrow case\ true\ :\ IS\ B$: Switch sur true
- $Case \rightarrow case\ false\ :\ IS\ B$: Switch sur false
- $Case \rightarrow default\ :\ IS\ B$: Cas par défaut
- $B \rightarrow \Lambda$: Présence ou non d'un break en fin de cas
- $B \rightarrow break;$

Cette structure devra permettre de faire un switch sur un entier, un booléen ou un type énuméré.

Syntaxe

```
switch(expression)
{
    case constante1:
        instruction1_1
        instruction1_2
        ...
    case constante2:
        instruction2_1
        instruction2_2
        ...
    ...
    default:
        instruction_1
        instruction_2
        ...
}
```

Sémantique

Compare successivement la valeur de l'expression aux constantes `constante1`, `constante2`, ...

Si la valeur de l'expression correspond à l'une des constantes, l'exécution débute à l'instruction correspondante. Si aucune constante ne correspond, l'exécution débute à l'instruction correspondant à `default` si présent ; si absent, l'exécution continue après le switch. Le traitement par défaut n'est pas nécessairement présent et n'est pas nécessairement le dernier cas.

L'exécution se termine à l'accolade fermante du switch ou avant si l'instruction `break;` est utilisée. En général, l'instruction `break` sépare les différents cas, mais si elle n'est pas présente, les instructions suivantes (du cas suivant) sont alors exécutées.

Le comportement du (bout de) programme suivant

```
switch (a) {
case 0 :
        int x= 0;
        print x;
        break;
default  :
        print 3;
case 1 :
        int y = 4;
        print y:
        break;
}
```

doit être le même que le bout de programme suivant où un bloc est associé à chaque cas (observer la duplication du code du cas 1 pour construire un cas "default" autonome) :

```
if (a =0)
        {int x= 0;
        print x;}
else{
        if (a=1)
                {int y = 4;
                print y;}
        else{
                print 3;
                int y = 4;
                print y;}
}
```

Exemples de programme valide

Pour l'exemple, nous supposerons que l'opérateur de soustraction sur les entiers existe dans RAT et dans TAM (subroutine `ISub`).

```
int fib (int x) {
  int res = 1;
  switch (x)
  {
    case 1 :
    case 2 :
        break ;
    default :
        res = (call fib ((x-1))
            + call fib ((x-2)));
        break;
  }
  return res;
}

main{
  print call fib (6);
}
```

Ce programme affiche 8.

```
exemple2{
  int x = ...
  switch(x) {
    case 1 : print 42;
    case 2 : print 78;
             break;
    case 3 : int y = 3;
             print y;
             break;
    case 4 : int z = 3;
             print (z+x);
             break;
    case 5 : print 20;
    default: print 96;
  }
}
```

Ce programme affiche :

- `4278` si `x` vaut `1` ;
- `78` si `x` vaut `2` ;
- `3` si `x` vaut `3` ;
- `7` si `x` vaut `4` ;
- `2096` si `x` vaut `5` ;
- `96` sinon.

Avec un type énuméré :

```
enum Mois {Janvier, Fevrier, Mars, Avril, Mai, Juin, Juillet , Aout, Septembre, Octobre, Novembre, Decembre};

int getInt(Mois m){
  int e = 0;
  switch (m) {
    case Janvier : e = 1; break;
    case Fevrier : e = 2; break;
    case Mars : e = 3; break;
    case Avril : e = 4; break;
    case Mai : e = 5; break;
    case Juin : e = 6; break;
    case Juillet : e = 7; break;
    case Aout : e = 8; break;
    case Septembre : e = 9; break;
    case Octobre : e = 10; break;
    case Novembre : e = 11; break;
    case Decembre : e = 12; break;
  }
  return e;
}

test{
  Mois m = Janvier;
```

```
  Mois m2 = Septembre;
  print (call getInt (m));
  print (call getInt (m2));
}
```

Ce programme affiche `19`.

1.5 Combinaisons des différentes constructions

Bien sûr ces différentes constructions peuvent être utilisées conjointement.

Exemple de programme valide

```
enum Mois {Janvier, Fevrier, Mars, Avril, Mai, Juin, Juillet , Aout, Septembre, Octobre, Novembre, Decembre};
enum Comp {Ls, Eq, Gt};

int getInt(Mois m){
  int e = 0;
  switch (m) {
    case Janvier : e = 1; break;
    case Fevrier : e = 2; break;
    case Mars : e = 3; break;
    case Avril : e = 4; break;
    case Mai : e = 5; break;
    case Juin : e = 6; break;
    case Juillet : e = 7; break;
    case Aout : e = 8; break;
    case Septembre : e = 9; break;
    case Octobre : e = 10; break;
    case Novembre : e = 11; break;
    case Decembre : e = 12; break;
  }
  return e;
}

Comp compare (int p1 int p2) {
  Comp res =Gt;
        if (p1<p2){ res = Ls; }
  else { if (p1=p2){ res = Eq; }
  else             { res = Gt;} }
  return res;
}

Comp compare (Mois m1 Mois m2) {
   return (call compare ((call getInt (m1)) (call getInt (m2))));
}

int permute (int* p1 int* p2){
  int sauve = (*p1);
  (*p1) = (*p2);
```

```
  (*p2) = sauve;
  return 0;
}

int  permute (Mois* p1 Mois* p2) {
  Mois sauve = (*p1);
  (*p1) = (*p2);
  (*p2) = sauve;
  return 0;
}

bool ordonne (int* d1 Mois* m1 int* d2 Mois* m2){
  bool modif = false;
  switch (call compare ((*m1) (*m2))){
    case Gt :
      int  p1 = call permute (d1 d2);
      int  p2 = call permute (m1 m2);
      modif = true;
      break;
    case Eq :
      switch (call compare ((*d1) (*d2))){
        case Gt :
          int  p1 = call permute (d1 d2);
          int  p2 = call permute (m1 m2);
          modif = true;
          break;
        default :
          break;
      }
      break;
    case Ls :
      break;
  }
  return modif;
}

int  printDate (int  d1 Mois m1){
  print ([d1/call getInt  (m1)]);
  return 0;
}

prog {
  int* d1 = (new int);
  int* d2 = (new int);
  Mois* m1 = (new Mois);
  Mois* m2 = (new Mois);

  (*d1) = 15;
  (*m1) = Janvier;
  (*d2) = 8;
```

```
  (*m2) = Decembre;

  bool modif = call ordonne (d1 m1 d2 m2);
  int  err  = call printDate ((*d1) (*m1));
  err  = call printDate ((*d2) (*m2));

  (*d1) = 15;
  (*m1) = Janvier;
  (*d2) = 8;
  (*m2) = Janvier;

  modif = call ordonne (d1 m1 d2 m2);
  err  = call printDate ((*d1) (*m1));
  err  = call printDate ((*d2) (*m2));

  (*d1) = 15;
  (*m1) = Fevrier;
  (*d2) = 8;
  (*m2) = Janvier;

  modif = call ordonne (d1 m1 d2 m2);
  err  = call printDate ((*d1) (*m1));
  err  = call printDate ((*d2) (*m2));

}
```

Ce programme doit afficher `[15/1] [8/12] [8/1] [15/1] [8/1] [15/2]`.

2 Travail demandé

Vous devez compléter le compilateur écrit en TP pour qu'il traite le langage RAT étendu. Vous devrez donc :

- modifier l'analyseur lexical (`lexer.mll`) pour ajouter les expressions régulières associées aux nouveaux terminaux de la grammaire ;
- modifier l'analyseur syntaxique (`parser.mly`) pour ajouter / modifier les règles de productions, et modifier si nécessaire les actions de construction de l'AST ;
- compléter les passes de gestion des identifiants, de typage, de placement mémoire et de génération de code.

Attention, il est indispensable de bien respecter la grammaire de la figure 1 pour que les tests automatiques qui seront réalisés sur votre projet fonctionnent.

D'un point de vue contrôle d'erreur, seules les vérifications de bonne utilisation des identifiants et de typage sont demandés. Les autres vérifications (déréférencement du pointeur null, ...) ne sont pas demandées.

3 Conseils d'organisation du travail

Il est conseillé d'attendre la fin des TP pour commencer à coder le projet (le dernier TD porte sur les pointeurs), néanmoins le sujet est donné au début des TP pour que vous commenciez à réfléchir à la façon dont vous traiterez les extensions et que vous puissiez commencer à poser des questions aux enseignants lors des TD / TP.

Il est conseillé de finir la partie demandée en TP et que tous les tests unitaires fournis passent avant de commencer le projet, afin de partir sur des bases solides et saines.

Il est conseillé d'ajouter les fonctionnalités les unes après les autres en commençant par les pointeurs dont le traitement aura été présenté lors du dernier TD.

Il est conseillé, pour chaque nouvelle fonctionnalité de procéder par étape :

1. compléter la structure de l'arbre abstrait issu de l'analyse syntaxique ;
2. modifier l'analyseur lexical, l'analyseur syntaxique et la construction de l'arbre abstrait ;
3. tester avec le compilateur qui utilise des "passes NOP" ;
4. compléter la structure de l'arbre abstrait issu de la passe de gestion des identifiants ;
5. modifier la passe de gestion des identifiants ;
6. tester avec le compilateur qui ne réalise que la passe de gestion de identifiants ;
7. compléter la structure de l'arbre abstrait issu de la passe de typage ;
8. modifier la passe de typage ;
9. tester avec le compilateur qui réalise la passe de gestion de identifiants et celle de typage ;
10. compléter la structure de l'arbre abstrait issu de la passe de placement mémoire ;
11. modifier la passe de placement mémoire ;
12. tester avec le compilateur qui réalise la passe de gestion de identifiants, celle de typage et de placement mémoire ;
13. modifier la passe de génération de code ;
14. tester avec le compilateur complet et itam.

4 Critères d'évaluation

Une grille critériée sera utilisée pour évaluer votre projet. Elle décrit les critères évalués pour le style de programmation, le compilateur réalisé et le rapport. Pour chacun d'eux est précisé ce qui est inacceptable, ce qui est insuffisant, ce qui est attendu et ce qui est au-delà des attentes.

Programmation fonctionnelle (40%)					
		Inacceptable	Insuffisant	Attendu	Au-delà
Compilation		Ne compile pas	Compile avec des warnings	Compile sans warning	
Style de programmation fonctionnelle		Le code est dans un style impératif	Il y a des fonctions auxiliaires avec accumulateurs non nécessaires	Les effets de bords ne sont que sur les structures de données et il n'y a pas de fonctions auxiliaires avec accumulateur non nécessaire	
Représentation des données		Type non adapté à la représentation des données	Type partiellement adapté à la représentation des données	Type adapté à la représentation des données	Monade
Lisibilité	Code source documenté	Aucun commentaire n'est donné	Seuls des contrats succincts sont donnés	Des contrats complets sont donnés	Des contrats complets sont donnés et des commentaires explicatifs ajoutés dans les fonctions complexes
	Architecture claire	Mauvaise utilisation des modules / foncteurs et fonctions trop complexes	Mauvaise utilisation des modules / foncteurs ou fonctions trop complexes	Bonne utilisation des modules / foncteurs. Bon découpage en fonctions auxiliaires	Introduction, à bon escient, de nouveaux modules / foncteurs
	Utilisation des itérateurs	Aucun itérateur n'est utilisé	Seul List.map est utilisé	Une variété d'itérateurs est utilisé à bon escient dans la majorité des cas où c'est possible	Une variété d'itérateurs est utilisé à bon escient dans la totalité des cas où c'est possible
	Respects des bonnes pratiques de programmation	Code mal écrit rendant sa lecture et sa maintenabilité impossible (par exemple : failwith au lieu d'exceptions significatives, mauvaise manipulation des booléens, mauvais choix d'identifiants, mauvaise utilisation du filtrage...)	Code partiellement mal écrit rendant sa lecture et sa maintenabilité difficile	Code bien écrit facilitant sa lecture et sa maintenabilité	Code limpide
Fiabilité	Tests unitaires	Aucun test unitaire	Tests unitaires des fonctions hors *analyse_xxx*, ne couvrant pas tous les cas de bases et les cas généraux	Tests unitaires des fonctions hors *analyse_xxx*, couvrant les cas de bases et les cas généraux	Tests unitaires de toutes les fonctions, couvrant les cas de bases et les cas généraux
	Tests d'intégration	Aucun test d'intégration ou ne couvrant pas les quatre passes	Tests d'intégrations, des passes de gestion des identifiants, typage et génération de code, non complet	Tests d'intégration complets des passes de gestion des identifiants, typage et génération de code	Tests d'intégration complets des quatre passes

Traduction des langages (40%)				
	Inacceptable	Insuffisant	Attendu	Au-delà
Grammaire	Non conforme à la grammaire du sujet	Partiellement conforme à la grammaire du sujet	Conforme à la grammaire du sujet	Plus complète que la grammaire du sujet
Fonctionnalités traitées intégralement	Aucune	Quelques-unes	Toutes celles du sujet	Fonctionnalités non demandées dans le sujet traitées correctement

	Pointeurs	Surcharge	Types énumérés	Switch/Case
Gestion identifiants	☐ Inacceptable ☐ Insuffisant ☐ Attendu	☐ Inacceptable ☐ Insuffisant ☐ Attendu	☐ Inacceptable ☐ Insuffisant ☐ Attendu	☐ Inacceptable ☐ Insuffisant ☐ Attendu
Typage	☐ Inacceptable ☐ Insuffisant ☐ Attendu	☐ Inacceptable ☐ Insuffisant ☐ Attendu	☐ Inacceptable ☐ Insuffisant ☐ Attendu	☐ Inacceptable ☐ Insuffisant ☐ Attendu
Placement mémoire	☐ Inacceptable ☐ Insuffisant ☐ Attendu	☐ Inacceptable ☐ Insuffisant ☐ Attendu	☐ Inacceptable ☐ Insuffisant ☐ Attendu	☐ Inacceptable ☐ Insuffisant ☐ Attendu
Génération de code	☐ Inacceptable ☐ Insuffisant ☐ Attendu	☐ Inacceptable ☐ Insuffisant ☐ Attendu	☐ Inacceptable ☐ Insuffisant ☐ Attendu	☐ Inacceptable ☐ Insuffisant ☐ Attendu

— Inacceptable : Non traité
— Insuffisant : Partiellement traité ou erroné
— Attendu : Complètement traité et correct

Rapport (20%)				
	Inacceptable	Insuffisant	Attendu	Au-delà
Forme	Beaucoup d'erreurs de syntaxe et d'orthographe et mise en page non soignée.	Beaucoup d'erreurs de syntaxe et d'orthographe ou mise en page non soignée	Peu d'erreurs de syntaxe ou d'orthographe et mise en page soignée	Pas d'erreur de syntaxe ou d'orthographe et mise en page soignée
Introduction	Non présente	Copier / coller du sujet	Bonne description du sujet et des points abordés dans la suite du rapport	
Types	Aucune justification sur l'évolution de la structure des AST	Justifications non pertinentes ou non complètes sur l'évolution de la structure des AST	Justifications pertinentes et complètes sur l'évolution de la structure des AST	Justifications pertinentes et complètes sur l'évolution de la structure des AST. Comparaison avec d'autres choix de conception.
Pointeurs	Aucune explication sur leur traitement	Explications non pertinentes ou non complètes sur leur traitement	Explications pertinentes et complètes sur leur traitement (sans s'attarder sur les points déjà traités pour d'autres constructions)	Explications pertinentes et complètes sur leur traitement (sans...). Comparaison avec d'autres choix de conception.
Surcharge	Aucune explication sur leur traitement	Explications non pertinentes ou non complètes sur leur traitement	Explications pertinentes et complètes sur leur traitement (sans s'attarder sur les points déjà traités pour d'autres constructions)	Explications pertinentes et complètes sur leur traitement (sans...). Comparaison avec d'autres choix de conception.
Types énumérés	Aucune explication sur leur traitement	Explications non pertinentes ou non complètes sur leur traitement	Explications pertinentes et complètes sur leur traitement (sans s'attarder sur les points déjà traités pour d'autres constructions)	Explications pertinentes et complètes sur leur traitement (sans...). Comparaison avec d'autres choix de conception.
Switch/Case	Aucune explication sur leur traitement	Explications non pertinentes ou non complètes sur leur traitement	Explications pertinentes et complètes sur leur traitement (sans s'attarder sur les points déjà traités pour d'autres constructions)	Explications pertinentes et complètes sur leur traitement (sans...). Comparaison avec d'autres choix de conception.
Conclusion	Non présente	Creuse	Bon recul sur les difficultés rencontrées	Bon recul sur les difficultés rencontrées et améliorations éventuelles