DLP : Développement d'un langage de programmation

Master STL, Université Pierre et Marie Curie

Année 2020–2021

TME 1 – Installation et programmation en ILP1

Christian Queinnec & Antoine Miné

1 Prise en main d'ILP1

Objectif : obtenir un environnement de travail incorporant les outils et les programmes du cours.

Buts :
- installer l'environnement de travail ILP1 ;
- compiler ILP1 ;
- exécuter la base de tests intégrée à ILP1 afin de vérifier que tout fonctionne correctement.

1.1 Mise en place de l'environnement

Vous devez suivre les instructions de mise en place décrites dans le document *Environnement des TME* sur la page du cours. Vous devez notamment faire un *fork* du projet `ILP1` sur le serveur GitLab `https://stl.algo-prog.info`, puis importer le projet dans Eclipse. À l'issue de ces instructions, vous devriez avoir un projet `ILP1` dans le *Package Explorer* d'Eclipse. La mention `[ilp1 master]` associée au projet indique que le projet est bien géré par `git`.

En cas de difficulté pour vous connecter au serveur GitLab, vous trouverez sur la page de l'UE une archive ZIP contenant les sources d'ILP1. Vous pouvez l'utiliser pour commencer le TME en attendant que votre problème de connexion soit résolu. Notez cependant qu'il sera indispensable de faire tous vos rendus par le serveur GitLab.

1.2 Grammaires ANTLR

L'analyse syntaxique est la première étape de l'interprétation et de la compilation des programmes ILP. Elle transforme un programme texte à la syntaxe ILPML en arbre syntaxique Java. Nous utilisons pour cette analyse ANTLR 4, un outil de génération d'analyseurs syntaxiques à partir de grammaires. ANTLR 4 comporte plusieurs composants :

1. un outil qui convertit des fichiers grammaires `.g4` en fichiers sources Java ;
   - la grammaire d'ILP1 est contenue dans le fichier `ANTLRGrammars/ILPMLgrammar1.g4` ;
   - les fichiers Java sont générés dans le répertoire `target/generated-sources/antlr4/antlr4/` (il y a bien deux `antlr4` dans le chemin, ce n'est pas une erreur) ; ces fichiers Java seront compilés avec le reste des sources d'ILP (contenus dans `Java/src`) pour obtenir l'interprète et le compilateur ILP1 ;
2. une bibliothèque d'exécution, utilisée par les analyseurs syntaxiques Java générés ;
3. un greffon Eclipse qui automatise la conversion des `.g4` en `.java`.

Les composants 1 et 2 sont contenus dans l'archive JAR `antlr-4.4-complete.jar`.

Génération de l'analyseur syntaxique. Normalement, le greffon ANTLR 4 IDE se charge d'appeler automatiquement ANTLR pour générer les fichiers Java à chaque modification du fichier `.g4` source. Il fournit de plus une coloration syntaxique pour les fichiers `.g4`. Dans le cas où le fichier `.g4` n'est pas automatiquement compilé en Java, il est possible de lancer manuellement cette compilation grâce à un clic droit sur le nom du fichier `.g4` dans la fenêtre de gauche (*Package Explorer*), puis en sélectionnant dans le menu l'option *Run As*, puis *Generate ANTLR Recognizer*. Enfin, si le greffon ANTLR 4 pour Eclipse n'est pas installé, il est possible de lancer cette compilation en ligne de commande : dans un terminal, allez à la racine du projet (dans `~/git/ILP1`) et lancez le script « `./compile_ANTLR.sh` » qui compilera tous les fichiers `.g4` trouvés dans le projet.

Attention : il ne faut jamais modifier un fichier Java généré par ANTLR 4, seulement le fichier source `.g4` (sinon, vos modifications seront écrasées à la prochaine compilation du fichier `.g4`).

Travail à réaliser :

- Assurez-vous que le fichier `Java/jars/antlr-4.4-complete.jar` existe bien et est pris en compte dans le *classpath* : faites un clic droit sur le projet dans le *Package Explorer* d'Eclipse, puis *Properties* > *Java Build Path* > *Libraries*. Si `antlr-4.4-compelte.jar` n'est pas présent, ajoutez-le avec le bouton *Add JARs*.
- Assurez-vous que le greffon ANTLR 4 IDE du *Marketplace* d'Eclipse est bien installé : menu *Help* > *Eclipse Marketplace...*
- Vérifiez que le répertoire `target/generated-sources/antlr4/antlr4` est bien rempli de fichiers Java générés par ANTLR.
- Vérifiez que le répertoire `target/generated-sources` est configuré comme un des répertoires contenant les sources Java du projet : clic droit sur le projet `ILP1`, puis *Properties* > *Java Build Path* > *Source* et éventuellement *Add Folder...* s'il est nécessaire de l'ajouter.

1.3 Bibliothèque JUnit 4

Le projet ILP1 contient des tests unitaires utilisant la bibliothèque JUnit 4. Vous aurez par la suite à enrichir ces tests unitaires.

Travail à réaliser : assurez-vous que la bibliothèque JUnit 4 est bien référencée par le projet : clic droit sur le projet `ILP1`, puis *Properties* > *Java Build Path* > *Libraries* puis, si nécessaire, *Add Library...*> *JUnit* > *JUnit 4*.

1.4 Compilation des sources Java

Eclipse compile automatiquement les fichiers sources Java dès que le projet est créé, et à chaque fois qu'un fichier source est ajouté ou modifié. Si tout se passe bien, aucune croix rouge n'apparaît. Sinon, les messages d'erreur, qui apparaissent dans la fenêtre du bas, indiquent la nature du problème. Les causes les plus fréquentes d'erreur de compilation sont :

- une configuration incorrecte des bibliothèques utilisées (ANTLR ou JUnit, voir ci-dessus) ;
- une génération absente ou incorrecte des fichiers d'analyse syntaxique par ANTLR (greffon ANTLR 4 IDE non installé ou mal configuré) ;
- un *Java Build Path* incomplet (vérifiez les onglets *Source* et *Libraries*) ;
- un choix incorrect de *ComplianceLevel* (Java 8 ou supérieur est requis).

Travail à réaliser : vérifiez que la compilation s'est bien déroulée (pas de croix rouge, pas de message d'erreur).

1.5 Bibliothèque d'exécution C

Le compilateur ILP1 convertit un programme ILP en programme C, qui peut ensuite être compilé par un compilateur C standard (par exemple `gcc`). Le programme généré dépend néanmoins d'une bibliothèque d'exécution, programmée en C, qui fournit les types de données et fonctions communs à tous les programmes ILP1. Cette bibliothèque d'exécution se trouve dans le sous-répertoire `C` du projet Eclipse, dans les fichiers `ilp.c` et `ilp.h`.

Il n'est pas nécessaire de compiler la bibliothèque d'exécution séparément. Le script `C/compileThenRun.sh`, appelé automatiquement comme dernière étape de la compilation, se charge d'ajouter tous les fichiers `C/*.c` (dont notamment `C/ilp.c`) au code généré afin de créer un exécutable, qui est ensuite lancé.

La bibliothèque d'exécution peut utiliser le ramasse-miettes Boehm GC, ce qui est conseillé pour améliorer la gestion de la mémoire du code compilé. Le script `C/install_gc.sh` cherche si Boehm GC est déjà disponible sur votre système et, si ce n'est pas le cas, le compile à partir des sources `C/gc-7.2g.tgz` disponibles dans le projet.

Travail à réaliser : assurez-vous que Boehm GC est disponible en lançant le script `./install_gc.sh` dans un terminal depuis le sous-répertoire `C`.

1.6 Tests unitaires avec Eclipse et avec GitLab

Les tests unitaires JUnit 4 sont isolés dans des classes séparées, dans des packages dont le nom se termine en `.test`. La classe `com.paracamplus.ilp1.test.WholeTestSuite` est un méta-test, qui lance tous les tests unitaires des différents packages d'ILP1. Un test JUnit 4 se lance sous Eclipse avec un clic droit sur la classe de test, puis *Run as > JUnit test* (cette option n'apparaît que quand Eclipse détecte que la classe possède des points d'entrée JUnit, de la même manière que *Run as > Java application* n'est visible que si la classe possède une méthode `main` statique). Si tout se passe bien, le test affiche une barre totalement verte.

Travail à réaliser : lancez depuis Eclipse les tests JUnit 4 intégrés à la classe `WholeTestSuite` et vérifiez leur bonne exécution (barre totalement verte).

Un intérêt du test unitaire est de tester les régressions quand le code évolue. Vous serez donc amené à réutiliser ce test dans les prochains TME, alors que vous modifierez le langage ILP1. Pour gagner du temps, vous pourrez vous contenter de lancer un sous-ensemble des tests, en particulier `InterpreterTest` (pour tester l'interprète), ou `CompilerTest` (pour tester le compilateur).

Le serveur GitLab comporte une fonction d'intégration continue qui lance également des tests. C'est le fichier `.gitlab-ci.yml` qui précise les classes de test à exécuter. Normalement, ces tests sont lancés à chaque *push* des modifications du projet depuis Eclipse vers le serveur, mais il est également possible de lancer le jeu de tests manuellement, en cliquant, dans l'interface web du GitLab (`https://stl.algo-prog.info`), sur menu *CI / CD > Pipelines* (à gauche), puis sur le bouton vert *Run Pipeline*, et en confirmant en cliquant à nouveau sur *Run Pipeline* dans la nouvelle page. GitLab vous dirige sur une page indiquant l'état des tests. Au bout d'un certain temps, les tests passent d'un croissant bleu (en cours) ou de barres jaunes (en attente) à un V vert (réussi). Cliquez sur l'onglet *Tests* puis *ILP1* pour avoir des détails sur les tests effectués. Contrairement au lancement des tests depuis Eclipse, ces tests sont lancés sur le serveur ; ils peuvent prendre plus de temps à finir, et l'onglet *Tests* se remplira lentement (n'hésitez pas à rafraîchir la page).

Travail à réaliser : lancez l'intégration continue manuellement depuis l'interface web du serveur GitLab du cours et vérifiez leur bonne exécution (V vert).

1.7 Astuces pour le développement sous Eclipse

- Pour créer des classes Java, sélectionnez un nom de package sous `src` dans le *Package Explorer* avec un clic droit, et cliquez sur *New > Java Class* (idem pour créer des interfaces ou des packages).
- Si vous modifiez des fichiers en dehors d'Eclipse, la touche F5 indique à Eclipse de se resynchroniser avec le système de fichiers (attention, seul le répertoire actuellement sélectionné dans le *Package Explorer* et ses sous-répertoires sont rafraîchis).
- Lorsque le curseur est sur un nom, F3 permet d'ouvrir le fichier définissant ce nom.
- F4 permet de voir la hiérarchie des classes ou des interfaces du nom concerné.
- Ctrl + shift + O permet d'importer les classes ou interfaces qui manquent. Notez que des classes de même nom peuvent exister dans des packages différents. Il faut donc faire attention, en cas d'ambigüté, à importer la classe du bon package.
- Ctrl + espace propose des complétions.
- Le survol avec la souris d'une zone erronée propose souvent de bonnes solutions (*Quick Fix*).
- Augmentez votre espace de travail en déplaçant les vues, souvent inutiles, *Outline*, *Tasks* sous la vue *Package Explorer*. Pour cela faites glisser l'onglet depuis sa position actuelle à celle souhaitée. Vous pouvez aussi les fermer. En cas de problème, vous pouvez restaurer la vue d'origine dans le menu *Window > Reset Perspective...*
- Vous pouvez voir plusieurs fichiers côte-à-côte en prenant l'onglet de l'un et en le faisant glisser sur le bord horizontal (ou vertical) de la fenêtre. Vous pouvez même ouvrir deux fenêtres sur le même fichier (menu contextuel, *New Editor* sur l'onglet contenant le nom du fichier).

2 Exécution de programmes ILP1

Objectif : comprendre et réaliser toutes les étapes pour exécuter un programme ILP1.

Buts :
- créer une application en ligne de commande pour lancer l'interprète sur un fichier donné ;
- lancer l'interprète sur les programmes d'exemple et observer les résultats ;
- faire un *push* vers le serveur GitLab du cours.

La distribution d'ILP1 contient une classe `com.paracamplus.ilp1.interpreter.test.InterpreterTest` capable de lancer l'interprète sur une base de fichiers test. Cette classe a le format d'un test unitaire JUnit 4, et est donc lancée par un clic droit puis *Run as > JUnit test*. Elle va automatiquement tester tous les programmes contenus dans le répertoire `SamplesILP1` avec l'extension `ilpml`. Toutefois, la distribution n'inclue pas d'application pour lancer l'interprète sur un programme spécifié en ligne de commande, ce qui serait pourtant bien utile ! Nous allons réaliser une telle application ici, nommée `FileInterpreter`.

Attention : Toutes les extensions d'ILP, qu'elles soient vues en cours, réalisées en TME ou en examen, doivent impérativement respecter deux règles :

1. il est *interdit de modifier* une classe Java fournie ; l'extension doit être intégralement contenue dans de *nouvelles classes* ;
2. il est *interdit d'ajouter* une classe à un package existant dans les sources fournies ; toute nouvelle classe doit être contenue dans un *package dédié à l'extension* ; nous utiliserons par convention `com.paracamplus.ilpX.ilpXtmeY`, où `X` indique la version du langage étendu (ici 1, car nous étendons ILP1) et `Y` indique le numéro du TME (`1` également).

Suivant ces règles, nous demandons donc de :

1. créer un package, nommé `com.paracamplus.ilp1.ilp1tme1` ;
2. créer dans ce package une classe `FileInterpreter`.

La classe doit être exécutable, c'est à dire définir une méthode :

```
public static void main (String[] args)
```

et lancer l'interprétation des fichiers dont le nom est spécifié dans `args`. Le travail de `FileInterpreter` étant similaire à celui de `InterpreterTest`, nous vous conseillons de faire dériver `FileInterpreter` de `InterpreterTest`, et de réutiliser ses méthodes. En particulier :

— le constructeur de `InterpreterTest` prend en argument un objet `File` spécifiant le fichier à interpréter ; rappelons qu'en Java `new File(nom)` crée un objet `File` à partir d'un objet `nom` de type `String` dénotant un nom de fichier ;

— la méthode `processFile` de `InterpreterTest` permet de lancer l'interprète sur le fichier spécifié lors de la création de l'objet `InterpreterTest`.

Une fois la classe créée, elle peut être exécutée depuis un terminal, ou bien directement depuis Eclipse avec un clic droit suivi de *Run as > Java application*. Par défaut, sous Eclipse, la liste d'arguments d'un programme lancé ainsi est vide. Il sera donc nécessaire de spécifier en argument un nom de fichier à interpréter, ce qui ce fait avec l'option *Run as > Open run dialog...* onglet *Arguments*. Nous vous conseillons d'entrer une fois pour toutes comme nom de fichier `${file_prompt}`, ce qui vous permet de sélectionner le fichier dans une boîte de dialogue qui s'affichera à chaque nouvelle exécution de l'application.

Travail à réaliser :

— lisez le code de `InterpreterTest` dans `com.paracamplus.ilp1.interpreter.test` ;

— créez une classe `FileInterpreter` capable de lancer l'interprète sur un fichier spécifié en ligne de commande ;

— lancez l'interprète sur quelques exemples de fichiers `.ilpml` du répertoire `SamplesILP1` ;

— déterminez les étapes à suivre pour réaliser une application similaire, mais qui utiliserait le compilateur à la place de l'interprète ;

— faites un *push* de vos modifications vers le serveur GitLab par un clic droit dans Eclipse sur le projet, puis *Team > Commit...* (voir Sec. 5.2 du document « Environnement des TME » pour plus de détails) ; n'oubliez pas d'inclure un message de *commit* informatif ; vérifiez dans l'interface web de GitLab que ces modifications sont bien sur le serveur et que les tests d'intégration continue, lancés automatiquement par le *push*, continuent de fonctionner (nous n'ajoutons pas de test pour le moment).

3 Programmation en ILP1

Objectif : s'initier à la syntaxe et la sémantique du langage ILP1.

Buts :
- savoir écrire un programme ILP1 ;
- s'assurer que le programme est syntaxiquement correct ;
- comprendre l'exécution du programme.

Nous allons maintenant écrire des programmes dans le langage d'entrée d'ILP1, utilisant l'extension `.ilpml` et la syntaxe vue en cours.

Travail à réaliser :
- lisez le fichier de grammaire `ANTLRGrammars/ILPMLGrammar1.g4` qui décrit la syntaxe du langage en ANTLR 4 ;
- étudiez quelques exemples de fichiers `.ilpml` fournis dans `SamplesILP1` et faites la correspondance avec la grammaire ANTLR 4 ;
- créez une copie d'un des exemples fournis, modifiez-le de diverses manières et lancez l'interprète de la question précédente dessus pour tester votre compréhension du langage ILP1 ;
- créez en particulier des exemples à la syntaxe incorrecte ; quel est le résultat de l'interprétation ? Étudiez les messages d'erreur ;
- réalisez un programme ILP1 qui calcule le discriminant d'une équation du second degré ax^2+bx+c étant donnés les coefficients `a`, `b`, `c`. Votre programme doit d'abord lier les variables `a`, `b` et `c` aux valeurs choisies pour exécuter le calcul (avec des blocs unaires enchâssés). Ensuite, le programme calcule le discriminant et doit retourner l'une des chaînes suivantes :
  - `"discriminant négatif : aucune racine"`
  - `"discriminant positif : deux racines"`
  - `"discriminant nul : une seule racine"`

  Lancez l'interprète sur ce programme et vérifiez qu'il est syntaxiquement correct et donne le résultat attendu. Essayez d'éviter les calculs inutiles en ne calculant qu'une seule fois le discriminant.

4 Base de tests

Objectif : créer une base de tests.

Buts :
- comprendre les tests JUnit 4 intégrés à ILP1 ;
- ajouter des programmes aux tests de régression d'ILP1.

Les classes `InterpreterTest` et `CompilerTest` vérifient respectivement que l'interprète et le compilateur se comportent comme attendu, en se basant sur un jeu de programmes de test ILP1 fournis dans le répertoire `SamplesILP1`. Pour chaque test, il faut naturellement fournir :
- un fichier `.ilpml` contenant le programme ILP1 testé ;
- le résultat attendu, qui sera comparé au résultat effectif de l'interprétation ou de la compilation.

Comme un programme ILP1 fournit toujours une valeur de sortie (dernière expression évaluée) mais peut aussi afficher du texte arbitraire sur la console (primitive `print`), il faut fournir deux fichiers résultat :
- un fichier `.print` contenant l'affichage attendu sur la console (qui peut être vide) ;
- un fichier `.result` contenant la valeur attendue du résultat du programme.

Pour chaque extension d'ILP que vous serez amenés à réaliser, il vous sera nécessaire de développer une base de tests spécifique, avec deux buts :

1. vérifier que l'extension n'a pas cassé les fonctionnalités existantes (tests de regression) ;
2. vérifier que les nouveaux traits ajoutés au langage fonctionnent comme prévu.

La base de tests contiendra donc généralement des programmes ILP existants (pour vérifier le point 1) et des nouveaux programmes qui ne pouvaient pas être écrits dans les versions d'ILP antérieures (pour vérifier le point 2). Naturellement, la base doit être contenue dans un répertoire séparé des bases de tests existantes, et être lancée par une nouvelle classe de test JUnit 4, afin de ne pas « polluer » les versions antérieures d'ILP et les faire cohabiter harmonieusement.

Travail demandé :

- écrivez plusieurs programmes ILP1 de test au format `.ilpml` dans un nouveau répertoire `SamplesTME1` ; ajoutez les fichiers `.print` et `.result` associés ;
- créez, dans le package `com.paracamplus.ilp1.ilp1tme1.test`, des nouvelles classes de test JUnit 4 dérivant de `InterpreterTest` et de `CompilerTest` pour vérifier les programmes de ce nouveau répertoire `SamplesTME1` (vous changerez donc la valeur de l'attribut `samplesDirName`) ;
- vérifiez que ces tests sont correctement exécutés sous Eclipse, en ajustant au besoin les fichiers `.print` et `.result` (la comparaison entre la sortie effective et les fichiers est faite par des comparaisons de chaînes, qui sont assez sensibles aux espaces) ;
- ajoutez dans le fichier `.gitlab-ci.yml` une nouvelle règle `TME1`, sur modèle de la règle `ILP1` déjà présente, pour lancer ces classes de test lors de l'intégration continue sur le serveur GitLab ; vous ajouterez donc les lignes suivantes au fichier :

```
TME1:
  stage: tme
  script: >
    /home/dlp/run.sh -id TME1 -name TME1
    com.paracamplus.ilp1.ilp1tme1.test.InterpreterTest
    com.paracamplus.ilp1.ilp1tme1.test.CompilerTest
  artifacts:
    reports:
      junit: report.xml
```

- faites un *push* sur le serveur GitLab du cours et vérifiez que les tests fonctionnent également sur le serveur.

5 Première extension d'ILP1

Objectif : enrichir le nœud `ASTsequence` et vérifier par le test la correction de cette extension.

Buts :

- s'initier à l'extension du langage ILP ;
- apprendre à rédiger des tests JUnit 4.

Notre première extension consiste à ajouter une fonctionnalité simple à un type de nœuds déjà présent dans l'arbre syntaxique abstrait : le nœud séquence `ASTsequence`. Pour respecter la contrainte forte de ne pas modifier le code existant, nous devons créer une nouvelle version du nœud `ASTsequence` dans un nouveau package, tout en nous assurant que cette nouvelle version sera utilisée à la place de l'ancienne dans le code existant. Ceci sera facilité par l'héritage et par l'emploi dans ILP du motif Fabrique.

Plus précisément, nous voulons ajouter à la classe `ASTsequence` une méthode `getAllButLastInstructions` qui retourne la liste des instructions de la séquence, privée de la dernière instruction.

Commencez par créer un fichier `IASTsequence.java` dans le package `com.paracamplus.ilp1.ilp1tme1.sequence` contenant l'interface suivante :

```
package com.paracamplus.ilp1.ilp1tme1.sequence;

import com.paracamplus.ilp1.interfaces.IASTexpression;
import com.paracamplus.ilp1.interpreter.interfaces.EvaluationException;

public interface IASTsequence
extends com.paracamplus.ilp1.interfaces.IASTsequence
{
    IASTexpression [] getAllButLastInstructions () throws EvaluationException;
}
```

Notes :

- l'interface a le même nom, `IASTsequence`, qu'une interface existant déjà dans ILP1 ; il est donc parfois nécessaire de bien qualifier le nom d'interface souhaitée pour éviter les confusions (e.g., `extends com.paracamplus.ilp1.interfaces.IASTsequence`) ;
- la nouvelle interface dérive de l'ancienne interface ; ainsi, tout objet implantant la nouvelle interface peut être passé en argument à une méthode qui s'attend à un objet implantant l'ancienne interface, ce qui est le cas de l'ensemble du code ILP1 avant l'extension.

Travail demandé :

- créez dans `com.paracamplus.ilp1.ilp1tme1.sequence` une classe `ASTsequence` qui implante cette nouvelle interface (elle pourra bien sûr hériter de l'ancienne classe `ASTsequence`) ;
- créez également une nouvelle version de la fabrique `ASTfactory` qui, au lieu de créer des anciens nœuds `ASTsequence`, créera des nœuds de votre nouvelle classe ;
- créez une nouvelle version du test unitaire JUnit 4 `InterpreterTest` qui utilise la nouvelle fabrique au lieu de l'ancienne, et vérifiez que l'exécution des programmes ILP est inchangée ;
- créez un nouveau test unitaire en JUnit 4 qui teste les nouvelles fonctionnalités du nœud `ASTsequence`, en appelant la méthode `getAllButLastInstructions` et en vérifiant le résultat ;
- ajoutez votre classe de test au fichier `.gitlab-ci.yml`, faites un *push* et vérifiez l'exécution du test sur le serveur GitLab du cours.

6 Rendu

La procédure de rendu est détaillée en section 5.4 du document *Environnement des TME* sur la page du cours.

Normalement, vous avez effectué des *push* réguliers sur le serveur GitLab du cours après avoir réalisé chaque question, puis vérifié que le code visible dans l'interface web du serveur correspond bien à votre dernière version, et que le résultat des tests d'intégration continue est correct.

Pour finaliser votre rendu, il ne reste plus qu'à ajouter un *tag* : dans l'interface web du serveur GitLab, sélectionnez *Repository* > *Tags*, cliquez sur *New Tag*, choisissez pour nom « rendu-final-tme1 », et cliquez sur *Create tag*. Si vous n'avez pas fini toutes les questions à la fin de la séance TME, vous pouvez faire un *tag* « rendu-initial-tme1 » ; vous compléterez plus tard le TME et ferez un nouveau *tag* pour le rendu final. Il est possible d'ajouter dans le champ *Release Notes* des commentaires à destination du chargé de TME, par exemple pour expliquer les difficultés rencontrées ou les écarts par rapport à ce qui vous est demandé dans l'énoncé.