TP 2 : Interpréteur

1 Introduction

Pour le deuxième TP, vous allez construire un logiciel qui lit une phrase. Cette phrase sera interprétée selon trois interpréteurs différents. Les deux premiers interpréteurs sont décrits dans cet énoncé. Le troisième vous sera donné une semaine avant la remise. La phrase à interpréter représente l'architecture d'un projet d'informatique. Les interpréteurs seront utilisés pour vérifier l'architecture, pour générer le code de départ du projet et pour construire un schéma UML du projet.

La prochaine section décrit le logiciel à construire. La troisième section décrit une technique possible pour l'implémentation du logiciel. La dernière section décrit les attentes du devoir et les éléments d'évaluations.

2 Description

Un interpréteur est un logiciel qui exécute une suite de commandes. Cette exécution est dynamique : l'interpréteur va lire les commandes dans un fichier et ensuite les exécuter une après l'autre. Le terme commande désigne une unité de base pour l'interpréteur. Nous utilisons le terme langage pour désigner l'ensemble des commandes possible. Le terme programme désigne une suite de commande. Il est possible de construire plusieurs interpréteurs différents pour le même langage. C'est ce que nous allons explorer dans ce devoir.

Puisque ce texte doit parler de votre devoir (un programme) et du texte à interpréter (un programme), alors nous utiliserons le terme logiciel pour désigner le programme que vous devez construire et le terme programme pour le programme interprété par votre logiciel.

Ce logiciel doit interpréter le contenu d'un fichier. Votre logiciel doit faire deux validations du contenu du fichier. Une première validation va s'occuper de la syntaxe de base de chaque ligne du fichier. Cette validation est faite lors de la lecture du fichier (section 2.1). Lorsque la lecture est terminée, une deuxième validation sera faite par un des trois interpréteurs. Cette validation va vérifier l'ordre des commandes du fichier (section 2.2.1).

Dans cette section, nous allons décrire les entrées de notre interpréteur avec le langage qu'il utilise et ensuite nous allons décrire deux des interpréteurs qui seront exécutés sur le programme.

2.1 Entrées

Au lancement, votre logiciel doit demander à l'utilisateur d'entrer le nom du fichier qui contiendra le programme à interpréter. S'il y a des problèmes à ouvrir le fichier, alors le logiciel doit afficher un message d'erreur. Lorsque le fichier est ouvert, le logiciel doit lire et mémoriser le programme contenu dans le fichier. **Votre logiciel doit utiliser la classe Scanner pour lire la réponse de l'utilisateur et pour lire le contenu du fichier**.

2.1.1 Syntaxe du fichier en entrées

Voici la syntaxe de l'information que contiendra le fichier. Votre logiciel doit vérifier la syntaxe décrite dans cette section lors de la lecture. S'il y a une erreur, alors vous affichez un message d'erreur à l'aide de `System.err.println` et terminez l'exécution du logiciel à l'aide de `System.exit`.

Le fichier d'entrée contient la suite de commande à interpréter. Chaque ligne du fichier peut être vide ou contenir une (et une seule) commande. Une commande contient un ou des identificateurs et possiblement des parenthèses et virgules. Il peut y avoir des espaces autour des identificateurs, parenthèses et virgules. Un identificateur est une suite de caractères parmi : {a..z, A..Z, _, 0..9}. Le premier caractère ne peut pas être un chiffre (0..9).

Il y a sept (7) commandes différentes regroupées dans trois (3) catégories (nullaire, unaire et binaire). Chaque commande à un nom unique. Ce nom doit être en minuscule.

- Les commandes nullaires : ces commandes sont simplement composées de leur nom.
  - `classeFin`
  - `methodeFin`
  - `abstrait`
- La commande unaire : cette commande commence par son nom. Il est suivi d'une parenthèse ouvrante, d'un identificateur et se termine par une parenthèse fermante.
  - `classeDebut( identificateur )`
    - L'identificateur représente le nom de la classe.
- Les commandes binaires : ces commandes commencent par leur nom. Il est suivi d'une parenthèse ouvrante et de deux identificateurs séparés par une virgule. Elles se terminent par une parenthèse fermante.
  - `methodeDebut( identificateur, identificateur )`
    - Le premier identificateur représente le type de retour de la méthode.
    - Le deuxième identificateur représente le nom de la méthode.
  - `attribut( identificateur, identificateur )`
    - Le premier identificateur représente le type de l'attribut.
    - Le deuxième identificateur représente le nom l'attribut.
  - `parametre( identificateur, identificateur )`
    - Le premier identificateur représente le type du paramètre.
    - Le deuxième identificateur représente le nom du paramètre.

Voici un exemple de fichier valide.

```
abstrait
classeDebut( C )

classeDebut( D )
attribut( String, a1 )
attribut( String, a2 )

classeDebut( E )
methodeDebut( void, m1 )
methodeFin

abstrait
methodeDebut( void, m2 )
parametre( int, p1 )
parametre( int, p2 )
methodeFin
classeFin

classeFin

classeDebut( F )
classeFin

classeFin
```

2.2 TRAITEMENT

Votre logiciel devra lancer l'exécution des trois interpréteurs, un après l'autre.

Un interpréteur est une mécanique (machine) qui parcours une suite de commande, les exécutant une après l'autre. Pour ce faire, l'interpréteur doit maintenir un état interne. Un état interne est un ensemble d'information qui permet à l'interpréteur d'interpréter correctement une commande. Dans les descriptions qui suivent, nous allons proposer de l'information de base pour l'état de chaque interpréteur. Vous pouvez modifier cette information comme vous le souhaitez. L'important est que votre interpréteur réussisse à exécuter les commandes correctement.

Règle importante : votre interpréteur doit exécuter les commandes une après l'autre, sans revenir en arrière au sauter des commandes.

Voici la description de chaque interpréteur.

2.2.1 Premier interpréteur : cet interpréteur va vérifier l'ordre des commandes dans la suite.

Cet interpréteur va vérifier si l'ordre des commandes est correct dans le programme.

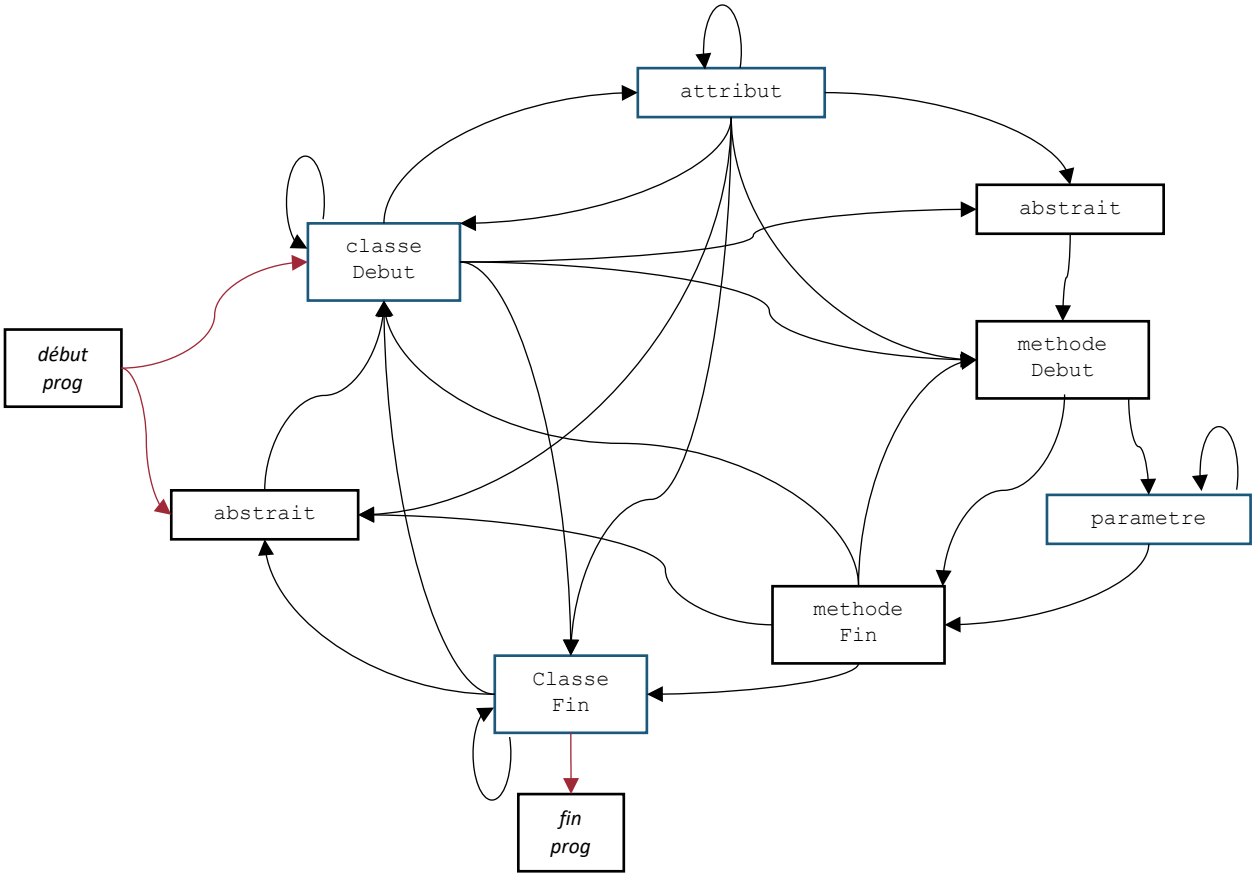
Ordre des commandes

Voici une description formelle de la syntaxe de notre langage de commande. Une explication de cette syntaxe va suivre la description.

f :
    (une fin de ligne)
fs :
    f { f }
lettreMinuscule :
    (une parmi) `a b c d e f g h i j k l m n o p q r s t u v w x y z`
lettreMajuscule :
    (une parmi) `A B C D E F G H I J K L M N O P Q R S T U V W X Y Z`
lettre :
    _
    lettreMinuscule
    lettreMajuscule
chiffre :
    (une parmi) `0 1 2 3 4 5 6 7 8 9`
caractere :
    lettre
    chiffre
identificateur :
    lettre { caractere }
declarationClasse :
    [`abstraite` *fs]* `classeDebut` `(` *identificateur* `)` *fs declaration* `classeFin` *fs*
déclaration :
    { declarationAttribut } { declarationMethode } { declarationClasse }
declarationAttribut :
    `attribut` (*identificateur* , *identificateur*) *fs*
declarationMethode :
    [`abstraite` *fs]* `methodeDebut` `(` *identificateur* , *identificateur* `)` *fs { declarationParametre }* `methodeFin` fs
declarationParametre :
    `parametre` `(` *identificateur* , *identificateur* `)` *fs*
programme :
    declarationClasse { declarationClasse }

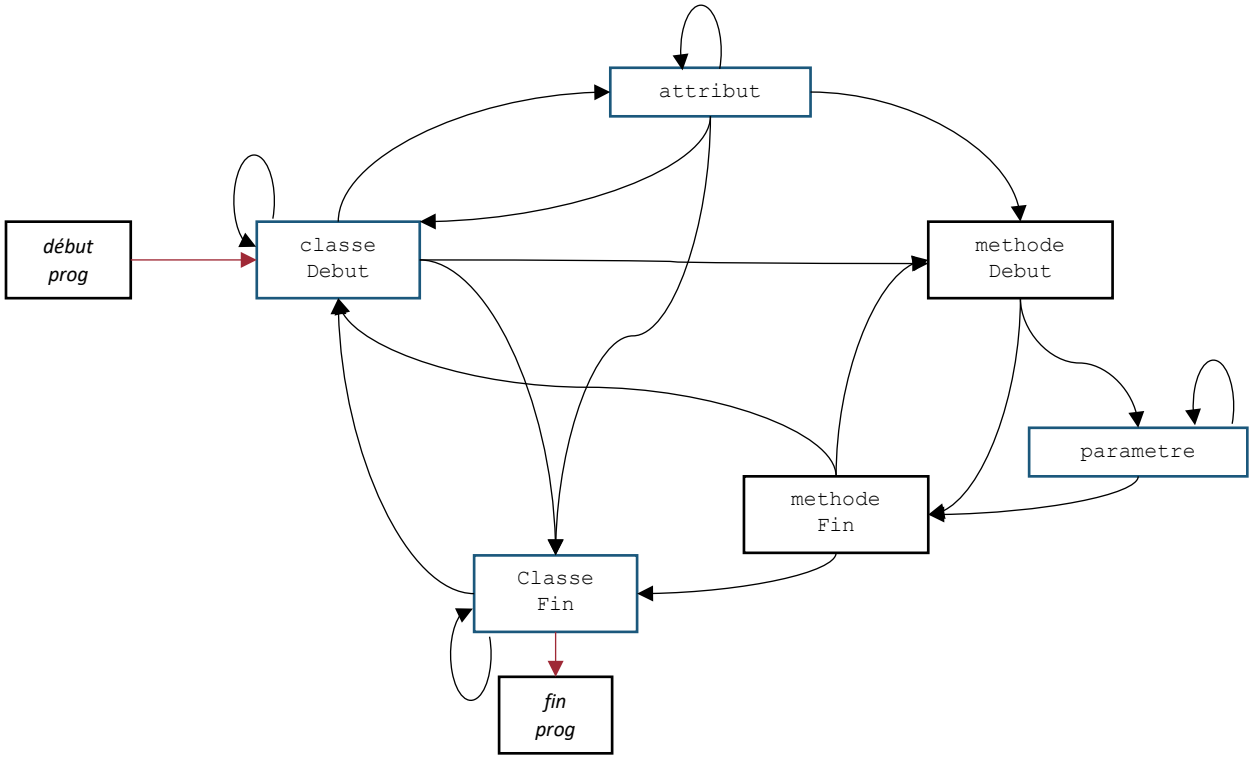
Dans cette description, les caractères écrits avec une police de caractère à espacement uniforme (`courier new`) représentent les caractères écrits dans le fichier. Les mots écrits en italique représentent des règles de productions. Une règle de production sert à décrire un élément syntaxique d'un langage (exemple : ***declarationAttribut :*** sers à décrire la syntaxe d'une commande `attribut`. Les éléments entre [crochet] représentent des éléments optionnels (pouvant apparaitre 0 ou 1 fois). Les éléments entre { accolade } représentent des éléments pouvant apparaitre zéro, une ou plusieurs fois. Lorsqu'une règle de production est sur plusieurs lignes alors chaque ligne représente un choix différent.

Cette description de la syntaxe des commandes nous permet de déduire l'ordre possible des commandes dans un programme. Par exemple, la commande `abstrait` ne peut apparaitre (optionnellement) qu'avant la commande `classeDebut` ou la commande `methodeDebut`. Nous pouvons déduire l'ordre des commandes et construire un graphe représentant cet ordre.



Ordre des commandes (suite)

Ce graphe ne tient pas compte du fait qu'il faut rencontrer le même nombre de `classeDebut` et de `classeFin`. Ce graphe est aussi complexe. Il est possible de le simplifier en retirant les commandes abstrait. Alors nous obtenons le graphe suivant qui permet de construire les interprétations



Chaque commande à une description de ce qui est correct (haut) et une description des changements qu'elle doit ensuite apporter à l'état (bas). Si une commande n'est pas correcte, alors l'interpréteur donne un message d'erreur sur `System.err` et termine l'exécution du logiciel (dans ce cas, les autres interpréteurs ne sont pas lancés).

État interne :

- mode : une valeur parmi {DClasse, DAttribut, DMethode, DParametre, FClasse, FMethode}, commence à FClasse.
- estAbstrait : un booléen, commence à faux.
- nbrOuverture : un entier, commence à zéro.

Commandes :

- `abstrait`
  - Il faut que estAbstrait soit à faux.
  - Placer estAbstrait à vrai.
- `classeDebut( identificateur )`
  - Cette commande est valide si mode ∈ {DClasse, FClasse, DAttribut, FMethode}.
  - Augmenter nbrOuverture de 1.
  - Le mode devient DClasse.
  - Placer estAbstrait à faux.
- `classeFin`
  - Cette commande est valide si mode ∈ {DClasse, FClasse, DAttribut, FMethode}.
  - Il faut que estAbstrait soit à faux.
  - Il faut aussi que nbrOuverture soit plus grand que zéro.
  - Diminuer nbrOuverture de 1.
  - Le mode devient FClasse.
- `attribut( identificateur, identificateur )`
  - Cette commande est valide si mode ∈ {DClasse, DAttribut}.
  - Il faut que estAbstrait soit à faux.
  - Le mode devient DAttribut.
- `methodeDebut( identificateur, identificateur )`
  - Cette commande est valide si mode ∈ {DClasse, DAttribut, FMethode}.
  - Le mode devient DMethode.
  - Placer estAbstrait à faux.
- `parametre( identificateur, identificateur )`
  - Cette commande est valide si mode ∈ {DMethode, DParametre}.
  - Il faut que estAbstrait soit à faux.
  - Le mode devient DParametre.
- `methodeFin`
  - Cette commande est valide si mode ∈ {DMethode, DParametre}.
  - Il faut que estAbstrait soit à faux.
  - Le mode devient FMethode.

Lorsque l'interpréteur à terminé, il faut que le mode soit FClasse, que estAbstrait soit à faux et nbrOuverture soit à 0.

2.2.2 Deuxième Interpréteur : générateur de code.

Ce deuxième interpréteur génére du code Java. Les commandes sont interprétées comme des déclarations :

1. classes (classeDebut, classeFin).

```
.
classeDebut( Nom )
.
classeFin
.
```

2. variables d'instance (attribut).

```
.
attribut( Type, Nom )
.
```

3. méthodes (methodeDebut, parametre, methodeFin).

```
.
methodeDebut( Type, Nom )
.
parametre( Type, Nom )
.
methodeFin
.
```

Lorsque qu'une méthode ou une classe est précédée de la commande `abstrait`, alors la méthode ou la classe est **abstract**. Si une commande de début de classe apparait dans une autre classe, alors la nouvelle classe hérite de l'autre. Par exemple.

```
.
classeDebut( Nom1 )
.
classeDebut( Nom2 )
.
classeFin
.
classeFin
.
```

Ici, la classe Nom2 hérite de la classe Nom1 puisque la déclaration de Nom2 est à l'intérieur de la déclaration de la classe Nom1. La première commande classeFin représente la fin de la déclaration de la classe Nom2 et la dernière commande classeFin représente la fin de la déclaration de la classe Nom1. Le code pour chaque classe devra être généré dans un fichier différent. Ce qui veut dire qu'à un moment donné, il y aura plusieurs fichiers ouverts. La meilleure stratégie est d'utiliser une structure de

Pile pour maintenir la liste des fichiers ouverts. Voici la description de l'effet de chaque commande. L'état interne de cette description contient une pile des fichiers ouverts (pileFichier). Les écritures dans la description des commandes vont toujours écrire dans le fichier qui est au sommet de la pileFichier. Ces commandes écrivent du code Java, formatez ce code proprement lors de la génération (indentation, espace, ligne…).

État interne :

- pileFichier : une pile contenant les fichiers ouverts.
- pileNom : une pile des noms de classes déclarés, mais non terminés (utilisé pour l'héritage).
- estAbstrait : un booléen, commence à faux (pour savoir si nous devons placer « abstract » devant une déclaration).
- estPremierParametre : un booléen, commence à faux (pour les virgules entre les paramètres).

Commandes :

- `abstrait`
  - Placer estAbstrait à vrai.
- `classeDebut( idNom )`
  - Ouvrir un nouveau fichier du nom « *idNom*.java » et placer ce fichier au sommet de la pileFichier.
  - Empiler le « *idNom* » de la classe au sommet de la pile pileNom.
  - Écrire « **public** »
  - Si estAbstrait alors écrire « **abstract** ».
  - Écrire « **classe** *idNom* ».
  - Si pileNom a plus d'un élément, alors écrire « **extends** » suivit du nom directement sous le sommet de pileNom.
  - Écrire « **{** ».
  - Placer estAbstrait à faux.
- `classeFin`
  - Écrire « **}** ».
  - Fermer le fichier au sommet de pileFichier.
  - Dépiler pileFichier.
  - Dépiler pileNom.
- `attribut( idType, idNom )`
  - Écrire « **private** *idType idNom* **;** ».
  - Écrire « **public** *idType* **get***idNom* **() {**».
  - Écrire « **return** *idNom* **;** ».
  - Écrire « **}** ».
  - Écrire « **public void set***idNom* **(** *idType idNom* **) {**».
  - Écrire « **this.***idNom* **=** *idNom* **;** ».
  - Écrire « **}** ».
- `methodeDebut( idType, idNom )`
  - Écrire « **public** ».
  - Si estAbstrait alors écrire « abstract ».
  - Écrire « *idType idNom* **(**» .
  - Placer estPremierParametre à vrai.

- `parametre( idType, idNom )`
  - si estPremierParametre est faux alors écrire « , ».
  - Écrire « *idType idNom* ».
  - Placer estPremierParametre à faux.
- `methodeFin`
  - Écrire « **)** ».
  - Si estAbstrait alors écrire « **;** » sinon écrire « **{}** ».
  - Placer estAbstrait à faux.

2.2.3 Troisième interpréteur : générateur de représentation UML.

Le troisième interpréteur va construire un fichier de type LaTeX contenant un diagramme UML décrivant le projet. LaTeX est un langage décrivant la disposition de texte et d'équation dans un document. Il est possible d'utiliser les librairies de ce langage pour décrire des dessins. Vous n'avez pas besoin d'apprendre ce langage pour réaliser cet interpréteur. UML est un standard pour décrire l'architecture d'un logiciel. Vous n'avez pas besoin de connaitre ce standard pour réaliser cet interpréteur.

Les écritures dans la description des commandes vont toujours écrire dans un fichier créer au début de l'interprétation. Ce fichier doit avoir le nom « uml.tex ». Certaines chaînes de caractères à écrire sont longues. Il y a un fichier disponible sur Moodle (DescriptionLatex.java) qui contient des constantes avec ces chaînes de caractères déjà construites. Il y a deux versions de ce fichier, la version '8' va fonctionner pour tout le monde (java 8 et plus). Il y a aussi la version '14', qui fonctionne seulement si vous utilisez Java 14 ou plus. Prenez un ou l'autre.

Pour cet interpréteur, nous avons besoin de construire une classe décrivant un état. Cette classe contient les variables d'instance suivantes :

État :

- public boolean premierAttribut = true;
- public boolean premiereMethode = true;
- public boolean premiereClasse = true;

État interne :

- nbrClasse : une valeur entière pour compter le nombre de classes ouvertes. Commence à 0.
- pileEtat : une pile des États.
- estAbstrait : un booléen, commence à faux (pour savoir si nous devons placer « abstract » devant une déclaration).
- estPremierParametre : un booléen, commence à faux (pour les virgules entre les paramètres).

Commandes :

- `abstrait`
  - Placer estAbstrait à vrai.
- `classeDebut( idNom )`
  - Si nbrClasse = 0 alors écrire DescriptionLatex.PAGE_DEBUT
    Sinon
    - Si pileEtat.sommet().premiereClasse alors
      - Écrire DescriptionLatex.CLASSE_FIN

                - Écrire DescriptionLatex.LISTE_CLASSE_DEBUT
                - Placer pileEtat.sommet().premiereClasse à faux.
            - Écrire DescriptionLatex.CLASSE_INTERNE_PREFIX
        - Empiler un nouvel État sur pileEtat.
        - Écrire DescriptionLatex.CLASSE_DEBUT
        - Si estAbstrait alors écrire DescriptionLatex.ABSTRAIT_DEBUT
        - Écrire *idNom*
        - Si estAbstrait alors
            - Écrire DescriptionLatex.ABSTRAIT_FIN
            - Placer estAbstrait à faux.
        - Augmenter nbrClasse de 1.

- `classeFin`
    - Diminuer nbrClasse de 1.
    - Si pileEtat.sommet().premiereClasse alors écrire DescriptionLatex.CLASSE_FIN
      Sinon écrire DescriptionLatex.LISTE_CLASSE_FIN
    - Si nbrClasse = 0 alors écrire DescriptionLatex.PAGE_FIN
      Sinon écrire DescriptionLatex.CLASSE_INTERNE_SUFFIX
    - Dépiler pileÉtat.

- `attribut( idType, idNom )`
    - Si pileEtat.sommet().premierAttribut alors
        - écrire DescriptionLatex.LISTE_ATTRIBUT_DEBUT
        - placer pileEtat.sommet().premierAttribut à faux
      Sinon écrire DescriptionLatex.LISTE_ATTRIBUT_SEP
    - écrire *idNom* + ':' + *idType* (important : **pas de fin de ligne**)

- `methodeDebut( idType, idNom )`
    - Si pileEtat.sommet().premiereMethode alors
        - Écrire DescriptionLatex.LISTE_METHODE_DEBUT
        - Placer pileEtat.sommet().premiereMethode à faux
      Sinon écrire DescriptionLatex.LISTE_METHODE_SEP
    - Si estAbstrait alors écrire DescriptionLatex.ABSTRAIT_DEBUT
    - Si *idType* != « void » alors écrire *idType* + ' '
    - Écrire *idNom*
    - Écrire DescriptionLatex.PARAMETRE_DEBUT
    - Placer estPremierParametre à vrai

- `parametre( idType, idNom )`
    - Si estPremierParametre alors placer estPremierParametre à faux.
      Sinon écrire DescriptionLatex.PARAMETRE_SEP
    - Écrire *idType*

- `methodeFin`
    - Écrire DescriptionLatex.PARAMETRE_FIN
    - Si estAbstrait alors
        - Écrire DescriptionLatex.ABSTRAIT_FIN
        - Placer estAbstrait à faux

Cet interpréteur va construire un fichier de type LaTeX. Si vous êtes intéressé à voir le résultat graphique (en format PDF), alors il suffit d'utiliser Overleaf (si vous avez un compte) ou un logiciel d'édition LaTeX comme TeXstudio.

3 CONSTRUCTION

Pour la construction du logiciel, vous pouvez utiliser la technique proposée dans cette section. Cette hiérarchie permet une bonne représentation d'un interpréteur et va nous donner une bonne flexibilité pour adapter différents contextes pour l'interprétation. Aussi, il faut utiliser cette technique pour pouvoir obtenir une note supérieure à 13/15. L'utilisation de cette technique ne garantit pas une note plus haute que 13.

3.1.1 Représentation de l'interpréteur

Pour représenter les commandes interprétables, nous allons utiliser une interface : `Expression`. Cette interface contient une méthode dont le contrat est de lancer l'interprétation d'une commande dans un contexte d'interprétation. C'est dans cette méthode que nous appliquons une fonction de rappel (callback).

```
void interprete( ContexteInterpretation contexte );
```

Un contexte d'interprétation est une interface qui représente le fonctionnement d'un interpréteur. Cette interface encapsule le fonctionnement des commandes sous un même type. Il va contenir une méthode pour chaque commande. Les fonctions de rappel seront dans cette classe.

Ensuite vous devez définir une classe abstraite pour regrouper les commandes du langage. Cette classe aura le nom `Commande` et implémentera la classe `Expression`. Chacune des commandes du langage sera représentée par une classe : `Abstrait`, `ClasseDebut`, `ClasseFin`, `Attribut`, `MethodeDebut`, `Parametre` et `MethodeFin`. Chacune de ces classes doit hériter de la classe `Commande`. Elles vont donc automatiquement implémenter l'interface `Expression`. Le code de la méthode `interprete` doit simplement faire appel à la méthode équivalente dans le contexte d'interprétation. Elle devra donner `this` en argument afin que le code appelé connaisse l'identité de l'appelant.

3.1.2 Contexte pour l'interprétation

Voici le code pour l'interface `ContexteInterpretation`.

```
public interface ContexteInterpretation {
   void genDebutClasse( MotDebutClasse motDebutClasse );
   void genFinClasse( MotFinClasse motFinClasse);
   void genDebutMethode( MotDebutMethode motDebutMethode);
   void genAttribut( MotAttribut motAttribut);
   void genAbstrait( MotAbstrait motAbstrait);
   void genParametre( MotParametre motParametre);
   void genFinMethode( MotFinMethode motFinMethode);
}
```

C'est cette interface qui sera implémentée par chaque interpréteur que vous voulez ajouter.

3.1.3 Lien avec le code

La méthode `interprete` dans les `Expression` de type `Commande` va simplement appeler les méthodes définies par le contexte d'interprétation reçu en argument. Pour l'interprétation d'un

programme, vous devez appeler la méthode `interprete` sur chaque `Commande`, une après l'autre, en utilisant le même contexte d'interprétation pour chacune.

Ensuite, il vous reste à construire une classe implémentant le `ContexteInterpretation` pour chaque interpréteur. Cette classe va contenir les variables d'instance représentant l'état et va implémenter les méthodes pour chacune des actions.

Finalement, pour lancer votre interpréteur, il suffit de construire une instance de la classe de contexte d'interprétation de l'interpréteur voulu. Ensuite, il reste à démarrer l'interprétation du programme avec le contexte d'interprétation.

3.1.4 Traitement des erreurs

Lors de l'exécution, il peut y avoir des erreurs :

- Un pop sur une pile vide.
- Impossible d'ouvrir un fichier.
- …

Si une de ces erreurs arrive, il faut afficher un message convenable en utilisant `System.err` et terminer l'exécution du logiciel avec un appel à `System.exit`.

4 DIRECTIVES

Les sections suivantes décrivent les attentes et les éléments d'évaluation pour le devoir.

4.1 DIRECTIVES POUR LA CONSTRUCTION DU PROJET.

- Placez vos noms au début du fichier `Principal.java`.
- Lorsque vous ajoutez une méthode, vous devez l'ajouter dans la classe appropriée.
- Des tests vous seront donnés avec les résultats attendus.

4.2 DIRECTIVES POUR L'ÉCRITURE DU CODE.

1. Le TP est à faire seul ou en équipe de deux.
2. Code :
    a. Pas de `goto`, `continue`.
    b. Les `break` ne peuvent apparaitre que dans les `switch`.
    c. Un seul `return` par méthode.
    d. Additionnez le nombre de `if`, `for`, `while`, `switch` et `try`. Ce nombre ne doit pas dépasser 5 pour une méthode.
3. Indentez votre code. Assurez-vous que l'indentation est faite avec des espaces.
4. Commentaires
    - Commentez l'entête de chaque classe et méthode.
    - Une ligne contient soit un commentaire, soit du code, pas les deux.
    - Utilisez des noms d'identificateur significatif.

- Une ligne de commentaire ou de code ne devrait pas dépasser 120 caractères. Continuez sur la ligne suivante au besoin.
- Nous utilisons Javadoc :
    - La première ligne d'un commentaire doit contenir une description courte (1 phrase) de la méthode ou la classe.
        - Courte.
        - Complète.
        - Commencez la description avec un verbe.
        - Assurez-vous de ne pas simplement répéter le nom de la méthode, donnez plus d'information.
    - Ensuite, au besoin, une description détaillée de la méthode ou classe va suivre.
        - Indépendant du code. Les commentaires d'entêtes décrivent ce que la méthode fait, ils ne décrivent pas comment c'est fait.
        - Si vous avez besoin de mentionner l'objet courant, utilisez le mot 'this'.
    - Ensuite, avant de placer les **tags,** placez une ligne vide.
    - Placez les **tag** @param, @return et @throws au besoin.
        - @param : décris les valeurs acceptées pour la méthode. Vous devez commenter les paramètres de vos méthodes.
    - Dans les commentaires, placer les noms de variable et autre ligne de code entre les tags {@ *du code ici* }.
    - Écrivez les commentaires à la troisième personne.

4.3 REMISE

Remettre le TP par l'entremise de Moodle. Placez vos fichiers `'*.java'` dans un dossier compressé de **Windows**, vous devez remettre l'archive. Le TP est à remettre avant le 8 avril 23 :55.

4.4 ÉVALUATION

- Fonctionnalité (9 pts) : des tests partiels vous seront remis. Un test plus complet sera appliqué à votre TP. Votre projet doit compiler sans erreur pour avoir ces points.
- Structure (4 pt) : découpez votre code en classe et méthode. Si votre TP utilise la technique proposée à la section 3, alors vous pouvez avoir jusqu'à 4 points pour la structure, sinon vous serez limité à 2 points pour la structure.
- Lisibilité (2 pts) : commentaire, indentation et noms d'identificateur significatif.