**Polytech' Paris-Saclay** **ET4-Info** **2023/2024**

# Projet de Compilation (à faire préférentiellement par groupes de 4 étudiants)

## Description générale

Il s'agit de réaliser un **compilateur** pour un micro-langage de programmation à objets. Un programme a la structure suivante :

*liste éventuellement vide de définitions de classes ou d'objets isolés*

*bloc d'instructions jouant le rôle de programme principal*

Une **classe** décrit les caractéristiques communes aux objets de cette classe : les **champs** mémorisent l'état interne d'un objet et les **méthodes** les actions qu'il est capable d'exécuter. Une classe peut être décrite comme extension ou spécialisation d'une (unique) classe existante, sa super-classe. Elle réunit alors l'ensemble des caractéristiques de sa super-classe et les siennes propres. Ses méthodes peuvent redéfinir celles de sa super-classe. La relation d'**héritage** est transitive et induit une relation de sous-type : un objet de la sous-classe est vu comme un objet de la super-classe.

Les objets communiquent par « envois de messages ». Un message est composé du nom d'une méthode avec ses arguments ; il est envoyé à l'objet destinataire qui exécute le corps de la méthode et peut renvoyer un résultat à l'appelant. La liaison de méthodes est **dynamique** : en cas d'appel d'une méthode redéfinie dans une sous-classe, la méthode exécutée dépend du type dynamique du destinataire, pas de son type apparent.

**Classes prédéfinies :** il existe deux classes prédéfinies. Les instances de `Integer` sont les constantes entières selon la syntaxe usuelle. Un `Integer` peut répondre aux opérateurs arithmétiques et de comparaison habituels, en notant `=` l'égalité et `<>` la non-égalité. Il peut aussi exécuter la méthode `toString()` qui renvoie une chaîne avec la représentation de l'entier. Les instances de `String` sont les chaînes de caractères selon les conventions du langage C. On ne peut pas modifier le contenu d'une chaîne. Les méthodes de `String` sont `print()` et `println()` qui impriment le contenu du destinataire et le renvoient en résultat, et l'opérateur binaire `&` qui renvoie une nouvelle instance de `String` formée de la concaténation de ses opérandes. **On ne peut pas ajouter des méthodes ou des sous-classes aux classes prédéfinies.**

## Description détaillée

### I Déclaration d'une classe

Elle a la forme suivante [1] : **`class`** `nom (parametre, …)` ***`[extends nom]`*** **`is`** `{ … }`

Une classe commence par le mot-clef `class` suivi du nom et, entre parenthèses, la liste éventuellement vide des paramètres de son unique constructeur. Les parenthèses sont obligatoires même si le constructeur ne prend pas de paramètre. La syntaxe d'un paramètre, d'un constructeur et des méthodes est donnée page suivante.

La clause optionnelle `extends` indique le nom de son éventuelle super-classe.

Après le mot-clé `is`, on trouve entre accolades la liste optionnelle des déclarations des champs, suivi de la liste optionnelle des méthodes de la classe et du constructeur éventuel, dans un ordre quelconque.

```
class Point(xc, yc : Integer) is {
     auto x, auto y, index: Integer;
     var auto name : String;
     static auto next: Integer;
     def Point(xc: Integer, yc: Integer) is {
       this.x := xc; this.y := yc; this.index := Point.incr();
      this.setName("Point_" & this.index.toString());
     }
     static def init() is { Point.next = 1; }
     static def incr() : Integer is { next := next +1; result := this; }
      def setName(newName: String) is { this.name := newName; }
}
```

1 *Les parties optionnelles dans la définition de la syntaxe sont incluses entre `[` et `]`.*

```
class PointColore(xc, yc: Integer, col: Couleur) extends Point is {
     auto c: Couleur;
     def PointColore(xc: Integer, yc: Integer, col: Couleur): Point(xc, yc) is {
       this.c := col; this.setName("PC_" & index.toString());
     }
 }
```

Les champs ne sont visibles que dans le corps des méthodes de la classe (modulo héritage). Un champ d'une sous-classe peut **masquer** un champ d'une de ses super-classes. Une méthode a accès aux champs des instances de sa classe (au sens large) et seulement à ceux-ci. Les noms des classes et des méthodes sont visibles partout.

## II Constructeur

Une classe a un unique constructeur dont l'en-tête correspond à celui de la classe (au regroupement près des paramètres de même type, auquel cas l'ordre et les noms des paramètres doivent être respectés). Si le constructeur a des paramètres, la classe doit explicitement définir son constructeur. Si le constructeur n'a pas de paramètres et que la classe ne comporte pas de constructeur, le compilateur engendre un constructeur par défaut qui se contente de renvoyer l'instance l'instance sur laquelle il a été appliqué. Dans le cas d'une sous-classe, le constructeur de la sous-classe doit appeler explicitement le constructeur de sa super-classe, sauf si celui-ci est un constructeur sans paramètre. La syntaxe de définition d'un constructeur est illustrée sur les exemples. Un constructeur renvoie l'instance sur laquelle il a été appliqué ; sa définition ne comporte pas de type de retour.

## III Déclaration d'un champ

Elle a la forme : *`[static][auto]`* `nom : classe;`

Les champs n'ont pas de valeur initiale et devront être initialisés par le constructeur ou une autre méthode. Le mot-clef optionnel **`static`** a le même sens qu'en Java. Si le mot-clef optionnel **`auto`** est présent, la classe définit automatiquement une méthode du nom du champ, qui en renvoie sa valeur. Il n'y a pas de méthode définie automatiquement pour modifier la valeur d'un champ. On peut regrouper plusieurs attributs de même type, comme dans l'exemple ci-dessus (le mot-clef **`auto`** étant répété devant chaque attribut si besoin).

## IV Déclaration d'une méthode

Elle prend l'une des deux formes suivantes :

**`def`** *`[override] [static]`* `nom (param, ...) : classe := expression`

**`def`** *`[override] [static]`* `nom (param, ...) [ : classe ]` **`is`** `bloc`

Une déclaration de paramètre formel a la forme `nom: Classe`. Comme pour les attributs, on peut regrouper des paramètres de même type. Le mot-clef **`static`** est présent si la méthodeest une méthode de classe. Le mot-clef **`override`** est présent *ssi* la méthode redéfinit une méthode d'une super-classe.

Si la partie optionnelle `: nomClasse` est présente, elle indique le type de la valeur renvoyée, sinon la méthode ne renvoie rien. La première syntaxe est adaptée aux méthode dont le corps se réduit à une unique expression. Une telle méthode renvoie le résultat de l'expression qui constitue le corps de la méthode; La seconde syntaxe permet de définir des méthodes avec un corps arbitrairement complexe ou ne renvoyant pas de résultat. Si la méthode renvoie un résultat, celui-ci est la valeur de la pseudo-variable `result`. L'identificateur réservé `result` correspond à une variable implicitement déclarée par la méthode. L'usage de `result` est interdit dans le corps d'un constructeur ou dans le programme principal.

## V Expressions et instructions

Les **expressions** ont une des formes ci-dessous. L'évaluation d'une expression produit une valeur à l'exécution:

*identificateur*

*constante*

(*expression*)

(*nomClasse expression*)
*accès à un champ*
*instanciation*
*envoi de message*
*expression avec opérateur*

Les **identificateurs** correspondent à des noms de paramètres ou de variables locales à un bloc (dont le programme principal), visibles compte-tenu des règles de portée du langage. Il existe de plus trois identificateurs réservés :

- `this` et `super` avec le même sens qu'en Java ;
- `result`, dont le rôle a déjà été décrit.

La forme (*nomClasse expression*) correspond à un "cast" : l'expression est typée statiquement comme une valeur de type *nomClasse*, qui doit forcément être une **superclasse** du type de l'expression (pas de cast "descendant"). Le seul intérêt pratique de cette construction consiste à la faire suivre de l'accès à un attribut masqué dans la classe courante: le "cast" est notamment sans effet sur la liaison dynamique de fonctions.

L'**accès à un champ** a la forme *expression*`.`*nom*. En particulier à l'intérieur d'une méthode d'instance, l'accès à un champ du destinataire s'écrit sous la form `this.`*nom*. L'accès à un champ `static` se fait via le nom de la classe exclusivement (il n'est pas considéré comme un champ implicite d'une instance de la classe).

Les **constantes** littérales sont les instances des classe prédéfinies `Integer`, `Void` et `String`.

Une **instanciation** a la forme **`new`** *nomClasse*`(`*arg*, ...`)`. Elle crée dynamiquement et renvoie un objet de la classe considérée après lui avoir appliqué le constructeur de la classe. La liste d'arguments doit être conforme au profil du constructeur de la classe (nombre et types des arguments).

Les **envois de message** correspondent à la notion habituelle en programmation objet : association d'un message et d'un destinataire qui doit être **explicite** (pas de `this` implicite). La méthode appelée doit être visible dans la classe du destinataire, la liaison de fonction est **dynamique**. Les envois peuvent être combinés comme dans `o.f().g(x.h()*2, z.k())`. L'ordre de traitement des arguments dans les envois de messages n'est pas précisé par le langage. Pour appeler une méthode `static`, on utilise le nom de la classe comme destinataire comme dans `Point.get()`

Les **expressions avec opérateur** sont construites à partir des opérateurs unaires et binaires classiques, avec leurs syntaxe d'appel, priorité et associativité habituelles; les opérateurs de comparaison **ne** sont **pas** associatifs. Les opérateurs arithmétiques ou de comparaison ne sont disponibles que pour la classe `Integer`. L'opérateur binaire & (associatif à gauche) est défini pour la classe `String`. Il correspond à la concaténation de chaînes.

Les **instructions** du langage sont les suivantes :

*expression* `;`
*bloc*
**`return`**`;`
*cible* `:=` *expression*`;`
**`if`** *expression* **`then`** *instruction* **`else`** *instruction*

Une **expression** suivie d'un `;` a le statut d'une instruction : on ignore le résultat fourni par l'expression.

Un **bloc** est délimité par des accolades et comprend soit une liste éventuellement vide d'instructions, soit une liste **non vide** de déclarations de variables locales suivie du mot-clef **`is`** et d'une liste **non vide** d'instructions. Une déclaration de variable locale a la forme `nom: Classe`. Comme pour les attributs, on peut regrouper des variables locales de même type :

```
{ v1, v2, v3 : Integer; p1, p2 : Point;
  is
  v1 := 2; v2 := v1+1; v3 := 2*v2+v1;
  p1 := new Point(v1*2, v2+v3);
  p2 := p1.clone().println();
}
```

L'instruction `return;` permet de quitter immédiatement l'exécution du corps d'une méthode. On rappelle que le résultat est par convention le contenu de la pseudo-variable `result` au moment du `return` ou de la fin du bloc. Les constructeurs sont la seule exception à cette règle et renvoient toujours l'objet sur lequel ils sont appliqués : leur corps ne doit **pas** comporter d'occurrence de `result`.

Dans une **affectation**, la cible est un identificateur de variable ou le nom d'un champ d'un objet qui peut être le résultat d'un calcul, comme par exemple : `x.f(y).z := 3;` Le type de la partie droite doit être conforme avec celui de la partie gauche. Il s'agit d'une **affectation de références** et non pas de valeur, sauf pour les classes prédéfinies. On notera que l'affectation est une instruction et ne renvoie donc pas de valeur.

L'expression de contrôle de la **conditionnelle** est de type `Integer`, interprétée comme « vrai » si et seulement si sa valeur est non nulle. Il n'y a ni booléens, ni opérateurs logiques.

## VI Aspects Contextuels :

Les aspects contextuelles sont ceux classiques dans les langages objets, aux précisions près ci-dessous. D'autres précisions pourront être fournies en réponse à vos questions.

- La surcharge de méthodes dans une classe ou entre une classe et super-classe n'est **pas** autorisée en dehors des redéfinitions; elle est autorisée entre méthodes de classes non reliées par héritage. La redéfinition doit respecter le profil de la méthode originelle (**pas** de covariance du type de retour).
- Tout contrôle de type est à effectuer modulo héritage ;
- Les méthodes peuvent être (mutuellement) récursives ;
- Le graphe d'héritage doit être sans circuit ; les classes peuvent apparaître dans un ordre quelconque.

## VII Aspects lexicaux spécifiques

Les noms de classes doivent débuter par une **majuscule** ; tous les autres identificateurs doivent débuter par une **minuscule**. Les mots-clefs sont en **minuscules**. La casse des caractères importe dans les comparaisons entre identificateurs. Les commentaires suivent les conventions du langage C.

## Déroulement du projet et fournitures associées

1. Écrire un analyseur lexical et un analyseur syntaxique de ce langage. Définition d'un format d'AST et construction de l'AST pour représenter le programme à compiler.

2. Écrire les fonctions nécessaires pour obtenir un **compilateur** de ce langage vers le langage de la machine abstraite dont la description vous sera fournie. Un interprète du code de cette machine abstraite sera mis à disposition pour que vous puissiez exécuter le code que vous produirez. Cette étape nécessite en préalable la mise en place des informations nécessaires pour pouvoir effectuer les vérifications contextuelles, puis la génération de code.

La fourniture associée à cette seconde étape sera un dossier comportant :

- Les sources commentés
- Un document (5 pages maximum) expliquant les choix d'implémentation principaux **et un état d'avancement clair** (ce qui marche, ce qui est incomplet, etc.)
- Un résumé clair de la contribution de chaque membre du groupe

- Un fichier type `makefile` produisant l'ensemble des exécutables nécessaires. Ce fichier devra avoir été testé de manière à être utilisable par un utilisateur arbitraire (pas de dépendance vis-à-vis de variables d'environnement). Votre exécutable doit prendre en paramètre le nom du fichier source et doit implémenter l'option **`-o`** pour pouvoir spécifier le nom du fichier qui contiendra le code engendré.
- Vos fichiers d'exemples (avec des exemples corrects et des exemples incorrects).

**Organisation à l'intérieur du groupe**

Il convient de répartir les forces du groupe et de paralléliser **dès le début** ce qui peut l'être entre les différentes aspects de la réalisation. **Anticipez** suffisamment à l'avance les étapes de réflexion sur la mise en place des vérifications contextuelles et la génération de code : de quelle information avez-vous besoin ? Où la trouverez-vous dans le source du programme ? Comment la représenter pour la retrouver facilement ? Quelles sont les principales fonctions nécessaires et quel est leur en-tête, etc. Définissez des exemples simples et pertinents pour appuyer vos réflexions et pour vos futurs tests de votre réalisation. **Prévoyez des exemples de complexité croissante et des exemples tant corrects que incorrects.** Réfléchissez aux aspects du langage qui peuvent éventuellement être ajoutés dans un second temps.

**Attention aux dépendances des étapes** dans la réalisation: par exemple, pas la peine de faire le contrôle de type si vous n'avez pas encore réglé les problèmes de portée. Pour chaque identificateur de variable, mémoriser les informations nécessaires : un champ ? Un paramètre ? Une variable locale (de quel bloc), etc ! Est-il stratique ou non ? Est-il visible à tel endroit du programme ? Idem pour un nom de méthode !