**Polytech' Paris-Saclay** **ET4-Info** **2022/2023**

# Projet de Compilation (à faire 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. **Dans ce langage il n'existe que des champs et méthodes « d'instances » : il n'existe pas l'équivalent des champs ou méthodes `static` de Java. Par contre on peut définir des objets isolés, sans leur associer une classe** (voir ci-dessous).

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 (arg, …)] [bloc] 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. Une classe a toujours un constructeur, dont le corps peut être vide mais qui renvoie toujours implicitement l'instance sur laquelle il a été appliqué.

La syntaxe d'un paramètre a la forme suivante : `nom : classe`

La clause optionnelle `extends` indique le nom de la super-classe avec, entre parenthèses, les arguments pour le constructeur de la superclasse. Les parenthèses sont obligatoires même en l'absence d'arguments.

Le bloc optionnel qui suit correspond au corps du constructeur et peut donc en référencer les paramètres ainsi que les attributs visibles dans la classe. Après le mot-clé `is`, on trouve entre accolades la liste optionnelle des déclarations des champs suivie de la liste optionnelle des méthodes. Si le bloc est absent, le constructeur renvoie l'instance sur laquelle il a été appliqué.

Des exemples d'en-têtes de classes (certains aspects sont expliqués plus bas) apparaissent page suivante :

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

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

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

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 ne peut accéder directement qu'aux champs de l'objet sur lequel elle est appliquée**, pas à ceux d'autres objets y compris ceux de la même classe (la visibilité n'est pas liée au type comme en Java). Les noms des classes, des objets isolés et des méthodes sont visibles partout.

## II Déclaration d'un objet isolé

Un objet isolé est le récepteur de ce qui serait en Java des champs ou des méthodes statiques, avec une syntaxe simplifiée. Un objet isolé n'a jamais de paramètres pour son constructeur et on omet le couple de parenthèses. Un objet isolé ne peut pas hériter d'une classe ou d'un autre objet. Sa syntaxe de déclaration se résume à la forme suivante :

```
object nom [bloc] is { … }
```

Un objet ne définit pas une classe et ne peut pas être instancié ou utilisé comme type. Il existe en un seul exemplaire et est automatiquement créé au lancement du programme, dans l'ordre de leur déclaration s'il en existe plusieurs. Exemple :

```
object CptPoint { this.next := 0; } is
   { var next: Integer;
     def incr() : Integer is
         { result = this.next; this.next = this.next + 1; }
     def get() : Integer := this.next - 1;
   }
```

## III Déclaration d'un champ

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

Les champs n'ont pas de valeur initiale et doivent être initialisés par le constructeur ou une autre méthode. 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.

## IV Déclaration d'une méthode

Elle prend l'une des deux formes suivantes :

```
def [override] nom (param, ...) : classe := expression
def [override] nom (param, ...) [ : classe ] is bloc
```

Une déclaration de paramètre formel a la forme `nom: Classe`.

Le mot-clef **`override`** est présent si et seulement si la méthode redéfinit une méthode d'une super-classe. Si la partie `: nomClasse` est présente, elle indique le type de la valeur retournée, sinon la méthode ne retourne aucune valeur. La première syntaxe est adaptée aux méthode dont le corps se réduit à une unique

expression. Une telle méthode renvoie une valeur qui est par définition 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. Dans ce cas, le résultat renvoyé est la valeur de la pseudo-variable `result`. L'identificateur réservé `result` correspond à une variable implicitement déclarée dans la méthode et qui ne peut être utilisée que comme cible d'affectations : elle ne peut pas être utilisée autrement dans une instruction ou une expression. 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 sans effet sur la liaison dynamique de fonctions.

L'**accès à un champ** a la forme *expression*`.`*nom*, mais on ne peut pas accéder directement à un champ d'un objet autre que le destinataire : *expression* doit forcément être `this`, `super` ou une expression parenthésée ou un « cast » construit au dessus de ces deux cas.

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 et avoir procédé aux initialisations éventuelles des champs. 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 et les appels aux constructeurs n'est pas précisé par le langage. Pour appeler une méthode d'un objet isolé, on utilise le nom de l'objet isolé comme destinataire. Exemple : `CptPoint.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`.

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 variable locale au bloc se déclare comme un champ sans le mot-clef **`var`**. Cependant, on peut regrouper plusieurs variables locales de même type en les séparant par une virgule.
Exemple de bloc avec des déclarations de variables locales :

```
{ 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).
- Les règles de portée sont les règles classiques des langages objets ;
- 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 et objets isolés peuvent apparaître dans un ordre quelconque.

## VII Aspects lexicaux spécifiques

Les noms de classes et d'objets isolés 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. Construction d'un arbre syntaxique, d'un AST ou de tout ensemble de structures C équivalent pour représenter le programme analysé.

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 de la contribution de chaque membre du groupe
- Un fichier `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 (tant corrects que 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 opeuvent éventuellement être ajoutés dans un second temps.

**Attention aux dépendances des étapes** dans la réalisation: par exemple, pas la peine d'espérer faire le contrôle de type si vous n'avez pas encore réglé les problèmes de portée. Pour chaque identificateur, mémoriser dans vos structures ce qu'il représente : un champ ? Un paramètre ? Une variable locale (de quel bloc), etc ! Est-il visible à tel endroit du programme ?