Projet de Compilation (à faire par groupes de ~4 étudiants)

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

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 : `Integer` et `String`. Les instances de `Integer` sont les constantes entières avec 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 seules méthodes de `String` sont `print` et `println` qui impriment le contenu du destinataire et le renvoient en résultat, ainsi que l'opérateur binaire `&` qui renvoie une nouvelle instance formée de la concaténation de ses opérandes. **On ne peut pas ajouter de méthode ou de sous-classe aux classes prédéfinies.**

Description détaillée

I Déclaration d'une classe

Elle a la forme suivante[1] :

`class` `nomClasse (param, …)` *`[`* ***`extends`*** *`nomClasse ]`* **`is`** `{ … }`

Une classe commence par le mot-clef `class` suivi du nom de la classe et, entre parenthèses, la liste éventuellement vide des paramètres de son unique constructeur. Les parenthèses sont obligatoires même s'il n'y a pas de paramètre.

La syntaxe d'un paramètre prend la forme suivante : *`[`* ***`var`*** *`]`* `nom : nomClasse`

S'il est précédé du mot-clef **`var`**, un paramètre définit implicitement un attribut d'instance de la classe qui sera automatiquement initialisé par la valeur de l'argument fourni à l'appel du constructeur de la classe. On peut regrouper plusieurs paramètres de même type en les séparant par une virgule. Si le mot-clef **`var`** est présent, il s'applique alors à tous les membres de la liste qui suit.

La clause optionnelle `extends`, si elle existe, indique le nom de la super-classe.

Après le mot-clé **`is`**, on trouve entre accolades les déclarations des attributs, des méthodes et du constructeur de la classe, dans un ordre quelconque.

Une classe doit toujours définir **un unique constructeur** dont la déclaration a la forme suivante:

`def` `nomClasse (param, …)` *`[ : superClasse(arg1, …) ]`* **`is`** `{ … }`

L'en-tête du constructeur doit correspondre à l'en-tête de la classe. Si la classe a une superclasse, le mot clef **`is`** est précédé de la partie optionnelle qui correspond à l'appel au constructeur de la superclasse. Le corps peut être vide mais un constructeur renvoie implicitement l'instance sur laquelle il a été appliqué. À la création d'une instance, on exécute le constructeur de la super-classe si elle existe, puis le corps du constructeur de la classe.

1 Les parties optionnelles dans la description de la syntaxe sont indiquées par *`[`* et *`]`*.

Exemples d'en-têtes de classes et constructeurs associés :

```
class Point(var xc, yc: Integer, var name: String) is
{ var index: Integer;
  var static cptPoint : Integer;
  def Point(var xc, yc: Integer, var name: String) is /* constructeur */
        { this.index := Point.incr(); }
  def static init() is { Point.cptPoint := 0; }
  def static getCpt() : Integer := Point.cptPoint
  def static incr() : Integer is
     { Point.cptPoint := Point.cptPoint + 1; result := Point.cptPoint; }
}
class PointColore(x, y: Integer, n: String, var col: Color)
      extends Point is
{ def PointColore(x, y: Integer, n: String, var col: Color)
      : Point(x, y, "PC-" & n)  is  { }
}
```

La classe `Point` définit 4 attributs d'instance et un attribut de classe. Elle définit son constructeur ainsi que plusieurs méthodes de classe. La classe `PointColore` ajoute un attribut d'instance à ceux hérités de sa classe. L'en-tête du constructeur de `PointColore` illustre l'appel au constructeur de la super classe.

II Déclaration d'un attribut

Elle a la forme suivante, qui ne permet pas d'initialiser l'attribut : **`var`** *`[static]`* *`nom`* `: nomClasse ;`
Comme pour les paramètres des en-têtes de classes, on peut factoriser la description des attributs de même type. Si le mot-clef **`static`** est présent, il décrit un attribut de classe (même notion qu'en Java) et s'applique à tous les éléments de la liste qui suit. Les attributs ne sont visibles que dans le corps des méthodes de la classe (modulo héritage). Un champ d'une classe peut **masquer** un attribut d'une super-classe.

Une méthode peut accéder aux attributs de l'objet sur lequel elle est appliquée, et à ceux de ses paramètres et variables locales de même type. Une méthode de classe ne peut pas accéder à un attribut d'instance, uniquement aux attributs de classe.

III Déclaration d'une méthode

Elle prend l'une des deux formes suivantes :

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

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 forme de 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 forme permet de définir des méthodes avec un corps arbitrairement complexe ou ne renvoyant pas de résultat.
Par convention, quand la méthode a un type de retour, le résultat renvoyé est la valeur d'une pseudo-variable `result`. Cette pseudo-variable est un identificateur réservé, correspondant à une variable implicitement déclarée dans la méthode, dont le type est le type de retour de la méthode. L'usage de `result` est interdit dans le corps d'une méthode qui ne renvoie pas de résultat, dans un constructeur et dans le corps du programme.

Les déclarations de paramètres des méthodes suivent la même syntaxe que les définitions d'attributs ou des paramètres de la classe, sauf qu'ils ne sont jamais précédés du mot-clef **`var`**.

IV 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*)
sélection
instanciation
envoi de message
expression avec opérateur

Les **identificateurs** correspondent à des noms de paramètres ou de variables locales visibles selon les règles de portée du langage. Il existe par ailleurs trois identificateurs réservés :

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

Une **sélection** a la forme *expression*`.`*nom* et a la valeur de l'attribut *nom* de l'objet qui est le résultat de l'évaluation de l'expression. L'attribut doit être visible dans le contexte dans lequel la sélection intervient. Le `this` doit être présent dans l'accès à un attribut d'instance du receveur (pas de `this` implicite) ou le nom de la classe pour un attribut de classe.

La forme (*nomClasse expression*) correspond à un "cast" : l'expression est typée statiquement comme une valeur de type *nomClasse*, qui doit forcément être une **super-classe** (au sens large) 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 fonction.

Les **constantes** littérales sont les instances des classe prédéfinies `Integer` 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())`. Les arguments dans les envois de messages et les appels aux constructeurs doivent être évalués **de la gauche vers la droite**. L'appel à une méthode de classe doit faire apparaître explicitement la classe concernée (exemple : `Point.getCpt()`).

Les **expressions avec opérateur** sont construites à partir des opérateurs unaires et binaires classiques, avec les syntaxe, priorité et associativité habituelles; les opérateurs de comparaison **ne** sont **pas** associatifs. Ces opérateurs binaires ou unaires ne sont disponibles que pour les éléments de la classe `Integer`. L'opérateur binaire & (associatif à gauche) est défini uniquement 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 au bloc suivie du mot-clef **`is`** et d'une liste non vide d'instructions. La syntaxe de définition des variables locales est identique à celle des paramètres de méthodes.
Exemple : `{ x, y: Integer; p1: Point; is x := 0; p1 := new Point(x, x); }`

L'instruction `return;` permet de quitter immédiatement l'exécution du corps d'une méthode. On rappelle que si une méthode renvoie un résultat, par convention celui-ci est le contenu de la pseudo-variable `result` au moment du `return` ou à la fin du corps de la méthode. Les constructeurs sont la seule exception à cette règle : ils renvoient toujours l'objet sur lequel ils sont appliqués (pas d'usage 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 pointeurs** 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.

V 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.

- Les définitions de classes peuvent apparaître dans un ordre arbitraire.
- La surcharge de méthodes dans une classe ou entre une classe et une 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 (« type conforme ») est à effectuer modulo héritage ;
- Les méthodes peuvent être (mutuellement) récursives ;
- Le graphe d'héritage doit être sans circuit.

VI Aspects lexicaux spécifiques

Les noms de classes débutent par une majuscule ; tous les autres identificateurs débutent par une minuscule. Les mots-clefs sont en minuscules. La casse des caractères importe dans les comparaisons entre identificateurs. Les commentaires et chaînes de caractères 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 AST pour représenter le programme analysé et impression sous forme non ambiguë du contenu de cet AST.

Cette étape fera l'objet d'une **remise à mi-parcours** du source de ces analyseurs ainsi que des tests effectués pour valider leur correction. Vos tests doivent permettre de s'assurer de la bonne prise en compte des précédences et associativités des constructions.

2. Écrire les fonctions nécessaires pour effectuer les vérifications contextuelles.

3. Ajouter une phase de génération de code pour obtenir un **compilateur** de ce langage vers le langage de la machine virtuelle dont la description vous a été fournie. Un émulateur de cette machine virtuelle est mis à disposition pour que vous puissiez exécuter le code que vous produisez.

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** 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 manipuler 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. **Prévoyez des exemples de complexité croissante et des exemples tant corrects que incorrects.** Réfléchissez aux aspects du langage qui peuvent éventuellement n'être ajoutés que dans un second temps.

Attention aux dépendances des étapes dans la réalisation: par exemple, le contrôle de type ne peut se faire qu'après avoir vérifié la portée des identificateurs. Pour chaque identificateur, mémoriser ce qu'il représente : un champ ? un paramètre ? une variable locale (de quel bloc) ? Dans quelle(s) partie(s) du programme cet identificateur est-il visible ? Pour une classe, combien de champs a-t-elle au total ? Combien de méthodes, compte-tenu des redéfinitions ?

La dernière partie du polycopié traite des **appels de fonctions** (en général) et de la gestion de la **liaison dynamique** dans les langages objet. **Lisez cette partie avant de réaliser cette partie de votre compilateur.**