Polytech' Paris-Sud 4ème Année **Compilation** **2015/2016**

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

bloc d'instructions jouant le rôle de programme principal

Une **classe** décrit les caractéristiques communes aux objets de cette classe : les **champs** stockent l'état interne d'un objet et les **méthodes** les actions qu'il peut exécuter. Une classe peut être décrite comme spécialisation d'une unique classe existante (sa superclasse) et réunit alors les caractéristiques de sa super-classe et les siennes propres. Elle peut redéfinir les méthodes de sa superclasse. Les champs de la super-classe sont visibles des méthodes de la sous-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 n'importe laquelle de ses superclasses.

Les objets communiquent par « **envois de messages** ». Un message est composé du nom d'une méthode avec ses éventuels arguments ; il est envoyé à l'objet destinataire qui exécute le corps de la méthode et renvoie le résultat à l'appelant. En cas de redéfinition d'une méthode par une sous-classe, la méthode exécutée dépend du type dynamique du destinataire comme en Java (**liaison dynamique** de fonctions).

Classes prédéfinies : la classe `Void` est non instantiable, contient l'unique valeur `void` et ne définit aucune méthode. Les instances de `Integer` sont les constantes entières décimales selon la syntaxe usuelle. `Integer` définit les quatre opérateurs arithmétiques (ainsi que les `+` et `−` unaires) et les six opérateurs de comparaison habituels, en notant `<>` la non-égalité. Elle définit aussi `toString` qui renvoie une représentation textuelle de l'entier. `String` définit les chaînes de caractères avec les conventions du langage C. On ne peut pas modifier le contenu d'une chaîne. Ses méthodes sont `print` et `println` qui impriment le contenu du destinataire et renvoient `void`, ainsi que l'opérateur binaire `&` qui renvoie la concaténation de ses opérandes. **Ces classes ne peuvent pas être redéfinies, modifiées ou dérivées.**

I Déclaration d'une classe

Elle a la forme suivante (les parties entre `[` et `]` sont optionnelles, les … indiquent une répétition) :

```
[final] class Nom([param,] …) [extends Nom([arg,] …)] is
                    { [champs…] [méthodes… ] [bloc] }
```

Une classe commence par le mot-clef `class` suivi de son nom et d'une liste des paramètres, définissant ainsi le prototype d'un (unique) constructeur. Les parenthèses sont obligatoires même en l'absence de paramètre. La présence du mot clef **`final`** indique que la classe n'est pas dérivable.

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

Après le mot-clé `is` on trouve entre accolades la liste optionnelle des déclarations de champs, suivie de la liste optionnelle des déclarations de méthodes. Le bloc optionnel final correspond au corps du constructeur et peut en référencer les paramètres.

Exemples d'en-tête :

```
class Point(xc: Integer, yc: Integer)  is
   { var x: Integer; var y : Integer; … { x := xc; y := yc; }}

class ColoredPoint(xc: Integer, yc: Integer, c: Color)
    extends Point(xc, yc)  is { var col : Color; … { col := c; }}

final class DefaultPoint() extends ColoredPoint(0,0,Color.white()) is { }
```

II Déclaration d'un champ

```
var [static] nom : type [:= expression];
```

La déclaration est précédée de `static` si le champ est défini au niveau de la classe.

Les expressions d'initialisation des champs `static` sont exécutées au début du programme, dans l'ordre de définition des classes et des champs. Le comportement d'un programme est indéfini si l'expression d'initialisation d'un champ `static` fait appel, directement ou non, à un objet de la même classe (mais elle peut référencer un champ `static` de la même classe s'il la précède).

Les expressions d'initialisation des champs non `static` sont exécutées **avant** le corps du constructeur de la classe et **après** l'appel au constructeur de la superclasse. Ces expressions ne sont pas dans la portée des paramètres du constructeur de la classe et ne peuvent donc pas les référencer. Il n'y a pas de valeur par défaut pour les champs ou, plus généralement, pour les variables locales des méthodes. Le comportement d'un programme qui référencerait à l'exécution une variable non initialisée est indéfini.

Les champs ne sont visibles que dans le corps des méthodes de la classe ou de ses sous-classes. Un champ peut **masquer** un champ d'une superclasse qui n'est alors plus visible dans la sous-classe (le langage ne définit pas de moyen syntaxique pour contourner ce masquage).

III Déclaration d'une méthode

```
def [static] [final] [override] nom ( [param, … ] ) returns Classe bloc
```

Le mot-clef `static` est présent si la méthode s'applique à la classe, **`final`** si la méthode ne peut plus être redéfinie et `override` doit être utilisé quand la classe définit une méthode de même nom qu'une méthode d'une superclasse ; on considère qu'il doit s'agir d'une redéfinition et elle doit en respecter strictement le profil : nombre et types des paramètres **et** du résultat. Il n'y a pas de contravariance des paramètres ou de covariance du type de retour. Un paramètre a la forme `nom : type`.

La visibilité des champs est liée à la classe : une méthode peut accéder aux champs de son destinataire et de ceux de ses paramètres et variables locales pourvu que les règles de visibilité des champs d'une classe soient respectées. Une méthode `static` ne peut référencer que des champs `static`. Une méthode non `static` peut référencer les champs `static`.

La surcharge de méthodes dans une classe ou entre classes et sous-classes n'est pas autorisée en dehors des redéfinitions mais est autorisée entre méthodes de classes non reliées par héritage.

Le corps d'une méthode est un bloc, tel que décrit ci-dessous. La valeur retournée par le corps doit être compatible avec le type de retour de la méthode. Les méthodes peuvent être récursives.

IV Les expressions

Les expressions ont une des formes ci-dessous. Toute expression renvoie une valeur.

Identificateur ou constante
expression arithmétique ou de comparaison
instanciation
sélection
envoi de message
affectation
`if-then-else` ou `if-then`
`do-while`
bloc nommé ou bloc anonyme
`yield` expression ou `yield nom : expression`

Les **identificateurs** correspondent à des noms de paramètres, de variables locales ou de champs visibles compte-tenu des règles du langage. Il existe trois **identificateurs réservés**: `this` a le même sens qu'en Java. ; `super` ne peut apparaître que comme **destinataire d'un appel à une méthode redéfinie** dans la classe, pour indiquer une liaison statique vers la méthode correspondante de la superclasse. Tout autre usage de `super` est interdit. L'identificateur `void` désigne l'unique valeur de `Void` ; on ne peut lui appliquer aucune méthode. Les **constantes** littérales sont les instances des classe prédéfinies `Integer` et `String`.

Les **expressions avec opérateur** sont les expressions arithmétiques et de comparaison classiques, avec syntaxe d'appel, priorité et associativité habituelles, sauf que les opérateurs de comparaison **ne** sont **pas** associatifs. Ces opérateurs ne sont disponibles que dans la classe `Integer`. On ne traite pas les opérateurs logiques. L'opérateur binaire `&` (associatif à gauche) n'est disponible que dans la classe `String`.

Une **instanciation** a la forme **`new`** `Classe( [arg, ...] )`. Elle crée dynamiquement et renvoie un objet de la classe considérée[1] après lui avoir appliqué le constructeur de la classe et procédé aux initialisations des champs. La liste d'arguments doit être conforme au profil du constructeur de la classe.

Une **sélection** a la forme `expression.nom` et prend la valeur du champ `nom` de l'objet correspondant au résultat de l'expression. Le champ doit exister dans l'objet en question et être visible dans le contexte dans lequel la sélection intervient. Pour accéder à un champ du receveur, le `this` peut être omis. L'accès à un champ `static` se fait sous la forme `nomClasse.nomChamp`.

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 `1 + o.f().g(x.h()*2, z.k())`. L'appel à une méthode `static` se fait sous la forme `nomClasse.nom(arguments...)`. L'ordre de traitement des arguments n'est pas défini dans le langage. Le passage des paramètres et du résultat se fait par pointeurs, sauf pour les types prédéfinis.

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 dans : `x.f(y).z := 3;` Le type de la partie droite doit être compatible avec celui de la partie gauche. Il s'agit d'une **affectation de pointeurs,** sauf pour les classes prédéfinies. La valeur retournée par l'affectation est celle de l'expression en partie droite. L'affectation est non associative.

La **conditionnelle** `if … then … else …` a le statut d'une expression et son exécution retourne donc une valeur. La condition doit être une expression à valeurs entières, interprétée comme « vrai » si elle est non nulle Les mots-clef `then` et `else` sont suivis d'une expression. Ces deux parties doivent renvoyer une valeur de même type, qui est par définition le type de la conditionnelle. On peut omettre la partie `else`, ce qui est équivalent à une clause `else void`, ou de façon équivalente, `else { }`.

La **boucle** `while … do …` a aussi le statut d'une expression. La partie condition suit les mêmes règles que pour la conditionnelle ; le mot-clef `do` est suivi d'une expression. Le type d'une boucle est `Void`.

Un **bloc nommé** a la syntaxe suivante : `{` *label* **`:`** *Classe* **`|`** *corps du bloc* `}`. Le *label* sert à donner un nom au bloc, pour pouvoir en sortir directement par une expression `yield` nommée. Le nom de classe *Classe* doit correspondre au type de la valeur retournée par ce bloc. Deux blocs d'une méthode ne peuvent pas avoir le même label. Le corps du bloc est constitué d'une liste de déclarations de variables locales ou d'expressions séparées par un `';'`, suivie d'un `';'` final optionnel. Une déclaration de variable locale a la syntaxe d'une déclaration de champ, hormis la clause `static`. Un même bloc ne peut pas contenir plusieurs déclarations avec un même nom de variable. Les variables locales ne sont pas forcément regroupées en début de bloc.

---

1 On ne se préoccupera pas de la gestion du « tas » en supposant implicitement l'existence d'un « garbage collector ».

À l'exécution, les déclarations et expressions sont exécutées en séquence et les valeurs des expressions intermédiaires ignorées. Si le corps du bloc est vide, ou termine par une déclaration de variable ou par un `';'` il renvoie `void`, sinon il renvoie la valeur de la dernière expression du corps. Le type d'un bloc est le type de la valeur qu'il renvoie.

Un **bloc anonyme** a la syntaxe allégée `{` *corps de bloc* `}` et est équivalent à un bloc nommé dont le nom aurait été engendré automatiquement et dont le type associé au label serait le type du bloc.

L'exécution de `yield` *expression* quitte l'exécution du bloc immédiatement englobant en renvoyant la valeur de l'expression. L'exécution de `yield` *nom* `:` *expression* est similaire sauf qu'elle sort du bloc nommé *nom* (qui doit être un bloc englobant) en quittant les éventuels blocs intermédiaires.

IV Vérifications contextuelles

Une classe ne peut référencer, en dehors d'elle-même, que des classes déjà définies. Un programme ne peut pas redéfinir une classe existante. Les noms des classes et des méthodes sont visibles partout. Dans une classe, un identificateur ne peut pas désigner à la fois une variable, un paramètre ou un champ et une méthode. Un identificateur peut désigner à la fois un bloc et une variable locale, un champ ou une méthode.

En dehors des règles spécifiques à chaque construction du langage, les règles de portée des paramètres et des variables locales sont celles habituelles en Java. La portée d'une variable locale va **de la fin** de sa déclaration à la fin du bloc englobant. Une expression ne peut référencer que des variables ou champs déjà déclarés.

Tout contrôle de type doit être fait modulo héritage.

Un constructeur renvoie systématiquement l'objet sur lequel il a été appliqué. Par convention, le corps d'un constructeur ne doit pas comporter d'instruction `yield` et doit être de type `Void`.

.

VI Aspects lexicaux spécifiques

Les **noms de classes** débutent par une **majuscule**, tous les autres identificateurs débutent par une **minuscule**. Les mots-clefs sont entièrement en minuscules. La casse importe dans les comparaisons entre identificateurs.

Déroulement du projet et fournitures associées

1. Écriture d'un analyseur lexical et d'un analyseur syntaxique de ce langage. Construction d'un arbre de syntaxe abstraite, ou tout ensemble de structures C équivalent, et réalisation de fonctions d'impression permettant de vérifier simplement la correction de ces analyseurs.

   Cette partie est à rendre le **mercredi 9 décembre à 18h**, accompagnée d'un ensemble de programmes de test que vous aurez conçus. À l'issue de cette étape des analyseurs lexical et syntaxique sera mis à disposition des groupes qui le souhaiteraient afin qu'un groupe ne reste pas bloqué sur cette partie.

2. Écriture des fonctions nécessaires pour effectuer les vérifications contextuelles adaptées à ce langage et définitions de programmes source sur lesquels tester vos fonctions.

3. Écriture d'un **compilateur** de ce langage vers le langage de la machine abstraite dont la description sera fournie ultérieurement. Un interprète du code de cette machine abstraite sera mis à disposition pour que vous puissiez exécuter le code que vous produirez.

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

- Les sources commentés
- Un document (5 pages maximum) expliquant les choix d'implémentation principaux et indiquant clairement l'état d'achèvement du projet.
- Un résumé de la contribution de chaque membre du groupe
- Un fichier `makefile` produisant les exécutables. Ce fichier ne devra pas avoir 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` qui spécifie le nom du fichier qui contiendra le code engendré.
- Vos fichiers d'exemples.

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 que nous détaillons ci-dessous. Anticipez les étapes de réflexion sur la mise en place des vérifications contextuelles. Définissez vos propres exemples simples et pertinents pour appuyer vos réflexions et servir ultérieurement de fichiers de test.

- Réalisation des analyseurs lexicaux et syntaxiques conformes à la définition du langage.
- Définition de structures de données ou d'arbres de syntaxe abstraite pour représenter un programme source (classe, champ, méthode, expression, etc). Écriture des fonctions associées (construction, parcours, …) et de fonctions d'impression pour vérifier la correction.
- Représentation des informations de portée, de type, etc. et écriture des vérifications contextuelles.
- Compréhension du fonctionnement de la machine virtuelle (fournie ultérieurement) et de son interprète.
- Réflexion sur l'organisation de la mémoire à mettre en place : calcul de la place nécessaire pour représenter un objet, représentation des objets en mémoire, organisation des « tableaux d'activation » pour les appels de fonctions, mise en place de la liaison dynamique de fonction.
- Actualisation des vérifications contextuelles pour calculer les nouvelles informations nécessaires.
- Génération de code en fonction de l'organisation de la mémoire mise en place.
- Implémentation des classes prédéfinies
- Test de ces différents aspects sur votre batterie d'exemples (des exemples supplémentaires seront mis à disposition mais il vous appartient de vous constituer votre propre base d'exemples pertinents).

Dans l'exemple fourni avec cet énoncé, **seules les vérifications lexicales et syntaxiques ont actuellement été effectuées**. En cas de doute sur le comportement attendu, n'hésitez pas à demander. N'extrapolez pas la syntaxe à partir de l'exemple : **respectez la définition du langage fournie par l'énoncé.** Comme pour le TP, un ensemble de fichiers est mis à disposition pour vous aider à démarrer.