Polytech' Paris-Sud 4ème Année **2017/2018**

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 trois classes prédéfinies. La classe `Void` est non instanciable, contient l'unique valeur `void` et ne définit aucune méthode. 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, ainsi que l'opérateur binaire `&` qui renvoie une nouvelle instance 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 (param, …) [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 : `[var] nom : classe [:= expression]`

On peut associer à un paramètre[2] une expression dont l'évaluation fournira à l'exécution une valeur par défaut pour ce paramètre si l'appel ne comporte pas l'argument correspondant. Pour qu'il n'y ait pas d'ambiguïté sur le paramètre concerné par une valeur par défaut, on impose que tous les paramètres avec valeur par défaut, soient situés **après** les paramètres qui n'ont pas de valeur par défaut. Le type de l'expression par défaut doit être conforme au type déclaré pour le paramètre.

Un paramètre du constructeur précédé du mot-clef **`var`** définit implicitement une variable d'instance de la classe qui sera donc automatiquement initialisée à la valeur de l'argument fourni à l'appel du constructeur.

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

2 Ceci ne s'applique pas uniquement aux paramètres des constructeurs mais à ceux de toute méthode.

La clause optionnelle `extends` indique le nom de la super-classe avec, entre parenthèses, les arguments à transmettre à son constructeur. 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. Après le mot-clé `is`, on trouve entre accolades la liste optionnelle des déclarations des champs suivie de la liste optionnelle des méthodes.

Quelques exemples d'en-têtes de classes :

```
class Point(xc : Integer := 0, yc : Integer := 0, name: String := "")
   { index := CptPoint.incr(); } /* corps du constructeur */
is { var index: Integer;         /* champs supplémentaires et méthodes */
     def f() …
   }

class ColoredPoint(xc: Integer, yc: Integer, c: Color)
    extends Point(xc, yc) is { … }
```

Un champ d'une sous-classe peut **masquer** un champ d'une de ses super-classes. Les champs ne sont visibles que dans le corps des méthodes de la classe (modulo héritage). Une méthode peut accéder aux champs de l'objet sur lequel elle est appliquée et à ceux de ses paramètres et variables locales de la même classe (la visibilité est liée au type). 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'ayant jamais de paramètres pour son constructeur, on omet le couple de parenthèses. Un objet isolé ne peut pas hériter d'une classe ou d'un autre objet et n'a pas de type associé. Sa syntaxe de déclaration se résume à la forme suivante :

```
object nom is { … }
```

Un tel 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 existe plusieurs déclarations d'objets isolés. Exemple :

```
object CptPoint is
   { var next: Integer := 1; /* index du prochain Point créé */
     def incr() : Integer is { result = next; next = next + 1; }
     def howMany() : Integer := next - 1;
   }
```

III Déclaration d'un champ

Elle a la forme suivante :

```
var nom : classe [:= expression];
```

La partie `:= expression` est optionnelle. Les expressions associées aux déclarations des champs sont exécutées, dans l'ordre de leurs déclarations, **avant** le corps du constructeur de la classe mais **après** l'appel au constructeur de la superclasse.

IV Déclaration d'une méthode

Elle prend l'une des deux formes suivantes :

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

Le mot-clef **`override`** est présent si et seulement si la méthode redéfinit une méthode d'une super-classe. Si la partie `: classe` est présente, elle indique le type de la valeur retournée, sinon la méthode ne retourne aucune valeur. La première forme de définition est adaptée aux méthode dont le corps se réduit à une unique expression. Une telle méthode renvoie forcément 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 de la pseudo-variable `result`. Celle-ci est un identificateur réservé, correspondant à une variable implicitement déclarée dans la méthode. L'usage de `result` est interdit dans le corps d'une méthode qui ne renvoie pas de résultat, dans un constructeur ou dans le corps du programme.

Une déclaration de paramètre formel a la forme `nom: classe [ := expression ]`. Les règles pour les valeurs par défaut sont les mêmes que pour les paramètres des constructeurs.

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*)
sélection
instanciation
envoi de message
expression avec opérateur

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

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

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

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

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 de fonctions.

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), compte-tenu d'éventuelles valeurs par défaut.

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 d'évaluation des arguments dans les envois de messages n'est pas précisé par le langage. Si un envoi de message nécessite l'usage d'une expression par défaut, et uniquement dans ce cas, celle-ci est évaluée pour fournir la valeur de l'argument manquant.

Les **expressions avec opérateur** sont construites à partir des opérateurs unaires et binaires classiques, avec syntaxe d'appel, 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 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 déclaration de variable locale au bloc a la syntaxe d'une déclaration de champ.

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 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 pointeur** 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). La méthode redéfinie bénéficie automatiquement des expressions par défaut pour les paramètres et ne doit **pas** les rappeler dans son en-tête. Elle ne peut pas modifier l'expression par défaut associée à un paramètre mais elle peut procurer une valeur par défaut à un paramètre qui n'en avait pas.
- Le contrôle de la portée des identificateurs qui interviennent dans une expression par défaut pour un paramètre est effectuée statiquement, en fonction du contexte qui existe dans la définition de méthode qui introduit cette valeur par défaut. Une telle expression ne peut pas mentionner `this` ou `super`, ni des variable d'instance.
- 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 ;

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.

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 obtenir un **compilateur** de ce langage vers le langage de la machine abstraite dont la description vous est 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 ?