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** 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. Les champs de la super-classe sont visibles par les 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 la super-classe. Un champ ou une méthode peut avoir un sens au niveau de la classe et est alors dit `static` (même notion qu'en Java).

Les objets communiquent par « envois de messages ». Un message est composé du nom d'une méthode avec ses éventuels arguments ; ce message est envoyé à un objet destinataire qui exécute le corps de la méthode demandée et renvoie un résultat à l'appelant. En cas d'appel d'une méthode redéfinie dans une sous-classe, la méthode à appliquer dépend du type dynamique du destinataire et non pas de son type apparent (**liaison dynamique** de fonctions).

Classes prédéfinies : il existe deux classes prédéfinies. Les instances de la classe `Integer` sont les nombres entiers, selon la syntaxe usuelle. Un `Integer` peut répondre aux opérateurs arithmétiques et de comparaison habituels. Il peut aussi répondre à la méthode `toString` qui retourne une chaîne avec la représentation de l'entier. Les instances de la classe `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 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 :

```
class nom (param, …) [extends nom (arg, …)] [bloc] is {decl, …}
```

Une classe commence par le mot-clef `class` suivi de son nom et, entre parenthèses, de 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 forcément un constructeur même si celui ne fait rien hormis retourner l'instance sur laquelle il a été appliqué.
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 si le constructeur ne prend pas de paramètre.
Le bloc optionnel qui suit correspond au corps du constructeur et peut donc référencer ses paramètres. Lors de l'initialisation d'un objet, il sera exécuté **après** l'appel au constructeur de la super-classe.
Après le mot-clé `is`, on trouve entre accolades la liste des déclarations des champs et des méthodes, dans un ordre arbitraire, éventuellement entrelacés.

La page suivante comporte quelques exemples d'en-têtes :

```
class Point(xc : Integer, yc : Integer) { x := xc; y := yc; } is
   { var x: Integer; var y : Integer; ... }
class ColoredPoint(xc : Integer, yc : Integer, c : Color)
    extends Point(xc, yc) { col := c; } is { var col : Color; … }
class DefaultPoint() extends ColoredPoint(0, 0, Color.white()) is { … }
```

Les champs ne sont visibles que dans le corps des méthodes de la classe (au sens large). Un champ d'une sous-classe **peut masquer** un champ d'une de ses super-classes.
Une méthode peut accéder aux champs de l'objet auquel elle est appliquée (seulement aux champs `static` si la méthode est `static`) ainsi qu'à ceux de ses paramètres et variables locales de la même classe (modulo héritage). Les méthodes peuvent être récursives. Les noms des classes et des méthodes sont visibles partout.

II Déclaration d'un champ

Elle a la forme suivante :

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

La partie `:= expression` est optionnelle. Une 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 lancement du programme, dans l'ordre de définition des classes et des champs. Les expressions associées aux déclarations des champs non `static` sont exécutées, dans l'ordre de leurs déclarations, **avant** le corps du constructeur de la classe et **après** l'appel au constructeur de la superclasse. Les expressions ne sont pas dans la portée des paramètres du constructeur de la classe et ne peuvent donc pas les référencer.

III Déclaration d'une méthode

Elle a la forme suivante :

```
[override | static] def nom (param, ...) [ returns type ] is bloc
```

Un paramètre a la forme `nom : type`. Le mot-clef def est précédé de `static` si la méthode s'applique à la classe. Si la clause **`returns`** `type` est présente, elle indique le type de la valeur retournée, sinon la méthode ne retourne rien. Par convention, le résultat renvoyé par une méthode est la valeur de la pseudo-variable `result`. Cette pseudo-variable est un identificateur réservé, correspondant à une variable implicitement déclarée (avec le bon type) dans chaque méthode qui renvoie une valeur. Cette pseudo-variable peut être la cible d'affectations mais ne peut pas être utilisée autrement dans une instruction.

IV Expressions et instructions

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

identificateur
sélection
constante
(*expression*)
(**as** *nomClasse* **:** *expression*)
instanciation
envoi de message
expression avec opérateur

Les **identificateurs** correspondent à des noms de paramètres, de variables locales à un bloc ou de champ visibles compte-tenu des règles du langage (les règles de portée sont celles des langages classiques). Il existe trois identificateurs réservés :

- `this` et `super` avec le même sens qu'en Java, sauf que `super` ne peut apparaître qu'en position de destinataire d'un message dont le sélecteur est une méthode redéfinie ;
- `result` qui a déjà été décrit et ne peut apparaître que dans le corps d'une méthode qui a un type de retour. Cet identificateur ne sert qu'à contenir la valeur de retour d'une fonction.

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.

La forme (**`as`** *nomClasse* **`:`** *expression*) correspond à un "cast" de l'expression: l'expression est typée statiquement comme une valeur de la classe *nomClasse*, qui doit forcément être une **superclasse** du type de l'expression (pas de "cast descendant" qui obligerait à tester dynamiquement le type à l'exécution). En pratique, le seul usage 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 l'aspect dynamique de la liaison de fonctions.

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

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

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 pour une méthode d'instance; le nom de classe est obligatoire pour une méthode `static`). 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 imposé par le langage.

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. 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*

Toute **expression** suivie d'un **`;`** a le statut d'une instruction.

Un **bloc** est délimité par `{` et `}` 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, hormis la clause `static`.

L'instruction **`return;`** permet de quitter immédiatement l'exécution d'un corps de méthode ou du programme principal. On rappelle que pour toute méthode qui renvoie une valeur, cette valeur est par convention le contenu de la pseudo-variable `result` au moment du `return` ou de la fin du bloc. La seule exception à cette règle est que les constructeurs retournent toujours l'objet sur lequel ils sont appliqués : en conséquence leur corps ne peut **pas** comporter d'occurrence de `result`. Le bloc qui constitue le programme principal ne renvoie pas de valeur.

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 compatible modulo héritage avec celui de la partie gauche. Il s'agit d'une **affectation de pointeurs** et non pas de valeur, sauf pour les classes prédéfinies.

L'expression de contrôle de la **conditionnelle** est de type `Integer`, interprétée comme « vrai » si et seulement si sa valeur est non nulle.

La surcharge de méthodes dans une classe ou entre une classe et super-classes n'est pas autorisée en dehors des redéfinitions. On autorise des méthodes homonymes dans des classes non reliées par héritage. On peut redéfinir une méthode non `static` d'une super-classe à l'aide du mot-clef `override` et en respectant le profil de la méthode originelle (nombre et types des paramètres **et** du résultat : pas de covariance du type de retour). Le corps d'une méthode est un « bloc » au sens ci-dessous. Tout **contrôle de type** est à effectuer modulo héritage.

V 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 importe dans les comparaisons entre identificateurs.

Déroulement du projet et fournitures associées

1. Écrire un analyseur lexical et un analyseur syntaxique de ce langage. Construction d'un arbre syntaxique, ou tout ensemble de structures C équivalent, et réalisation de fonctions d'impression pour vérifier facilement la correction de ces analyseurs.

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.

2. À partir des structures précédentes, écrire des fonctions nécessaires pour obtenir un **compilateur** de ce langage vers le langage d'une machine abstraite dont la description vous est fournie en annexe. Un interprète du code de cette machine abstraite sera mis à disposition pour que vous puissiez exécuter le code que vous produirez. Cette étape comporte 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
- 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, exécutables automatiquement (compilation avec sortie du code objet dans un fichier + exécution du code produit par votre compilateur par l'interprète de la machine abstraite) via le fichier `makefile`.

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 suffisamment à l'avance les étapes de réflexion sur la mise en place des vérifications contextuelles et la génération de code. Définissez des exemples simples et pertinents pour appuyer vos réflexions et servir ultérieurement de fichiers de test pour vérifier la correction de l'implémentation de tel ou tel aspect.

- Réalisation des analyseurs lexicaux et syntaxiques.
- Définition de structures C ou d'un arbre syntaxique pour représenter le programme source (classe, champ, méthode, expression, etc) avec leurs propriétés. Écriture des fonctions associées (construction, parcours, …) et de fonctions d'impression.
- Représentation des informations de portée, de type, etc. Réalisation de l'ensemble des vérifications contextuelles.
- Compréhension du fonctionnement de la machine virtuelle et de son interprète.
- Réflexion (notamment à partir d'exemples) sur l'organisation de la mémoire à mettre en place : représentation des objets en mémoire, calcul de la place nécessaire pour représenter un objet, organisation des « tableaux d'activation » pour les appels de fonctions, mise en place des mécanismes d'adressage, des « tables virtuelles de fonctions » et de la liaison dynamique de fonction). Intégration des informations nécessaires dans la représentation du programme source.
- Génération de code en fonction de l'organisation de la mémoire mise en place et des informations précédentes.
- Implémentation des classes prédéfinies
- Test de ces différents aspects sur votre batterie d'exemples (quelques 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 été complètement effectuées**. En cas de doute sur des aspects syntaxiques ou sur le comportement attendu, n'hésitez pas à demander.