Polytech' Paris-Sud 4me Année - Informatique 2010-2011

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 non vide de définitions de classes
bloc d'expressions (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. Un champ d'une sous-classe **ne peut pas masquer** un champ 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. 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 champs ne sont visibles que dans le corps des méthodes de la classe (au sens large). 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. Les noms des classes et des méthodes sont visibles partout, mais une classe ne peut mentionner que des classes déjà définies, en plus d'elle-même. Il en est de même pour les méthodes qui peuvent être récursives, mais pas mutuellement récursives.

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 réel 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 `Entier` sont les nombres entiers, selon la syntaxe usuelle. Un `Entier` peut répondre aux opérateurs arithmétiques et de comparaison habituels. Il peut aussi répondre à la méthode `imprimer` qui imprime l'entier et retourne sa valeur. Les instances de la classe `Chaîne` sont les chaînes de caractères selon les conventions du langage C. La seule méthode de `Chaîne` est `imprimer` qui imprime le contenu du destinataire et le renvoie en résultat. On ne peut pas modifier le contenu d'une chaîne. **On ne peut pas ajouter des méthodes ou des sous-classes aux classes prédéfinies.**

Description détaillée

Déclaration d'une classe : une classe commence par le mot-clef **`class`** suivi de son nom et, entre parenthèses, une liste éventuellement vide de paramètres, suivie éventuellement du nom de la super-classe avec, entre parenthèses, les arguments pour le constructeur de la super-classe. Le couple de parenthèses est obligatoire même si le constructeur ne prend pas de paramètre. L'en-tête de la classe correspond à la définition d'un constructeur appelé automatiquement lors de la création d'un objet et décrit le passage de paramètres entre les constructeurs de la sous-classe et de la super-classe. L'en-tête est suivi d'un bloc optionnel d'initialisation (correspondant au corps du constructeur de la nouvelle classe et exécuté **après** le constructeur de la super-classe), suivi du mot-clef **`is`** et du corps de la méthode (un bloc d'expressions).

```
class PointColore(xc: Entier, yc:Entier, c: Couleur:= Couleur.blanc())
      extends Point(xc, yc) { coul := c } is { … }

class DefaultPoint()
 extends PointColore(0, 0) is { … }
```

Le corps de la classe comprend la liste éventuellement vide de champs, suivie de la liste éventuellement vide de méthodes.

Déclaration d'un champ : elle peut avoir l'une des deux formes suivantes :

```
val nom : type := expression
var nom : type := expression
```

Le premier cas correspond à un champ constant : sa valeur ne peut être définie que dans les constructeurs ou par l'expression dans la déclaration. Cette restriction n'existe pas dans le second cas. Une déclaration peut être précédée de **`static`** si l'attribut est défini au niveau de la classe. Les champs **`static`** avec expression dans leur déclaration sont initialisés au début de l'exécution dans l'ordre de définition des classes. Les déclarations de champs sont **séparées** par '`;`'. Les expressions associées aux déclarations sont exécutées **après** le bloc d'initialisation.

Une **déclaration d'une méthode** a de la forme suivante :

```
def nom (paramètres éventuels) returns type is Bloc
```

Un paramètre a la forme `nom : type := expression`, la partie := `expression` est facultative et fournit une valeur par défaut au paramètre. Les paramètres avec valeur par défaut doivent apparaître **après** les paramètres sans valeurs par défaut. Le type de l'expression et du paramètre formel doivent être compatibles. Le mot clef **`def`** est **suivi** de **`static`** si la méthode s'applique à la classe. Une méthode retourne toujours un résultat. Il n'y a **pas** de séparateur entre les déclarations de méthodes.

La surcharge de méthodes dans une classe ou entre une classe et ses super-classes (au sens large) n'est pas autorisée en dehors des redéfinitions. Une méthode qui redéfinit une méthode doit ajouter le mot-clef **`override`** après **`def`** et doit conserver le profil de la méthode originelle (nombre et types des paramètres et du résultat, existence de valeurs par défaut). Un champ ne peut pas avoir le même nom qu'une méthode. Une méthode **`static`** ne peut pas surcharger une méthode, **`static`** ou non, d'une super-classe. La valeur de retour d'une méthode est celle de son bloc principal, qui doit être d'un type compatible avec le type de retour déclaré par la méthode.

Il existe des méthodes prédéfinies pour les opérateurs arithmétiques et de comparaison habituels (+, * , =, <=, etc.) mais leur usage est restreint à la classe prédéfinie `Entier`. Il n'est pas possible de définir de telles méthodes pour d'autres classes.

Expressions et Bloc d'expressions : Les expressions ont une des formes ci-dessous. Toutes renvoient une valeur (qui pourra éventuellement être ignorée, selon le contexte où cette expression est utilisée) :

Bloc
cible **:=** *Expr*
`if` *Expr* **`then`** *Expr* **`else`** *Expr*
Id
Sélection
Constante
(Expr)
`new` *nom-de-classe(arguments)*
envoi-de-message
expression arithmétique et de comparaison

Un **bloc** est délimité par `{` et `}` et comprend soit une liste non vide d'expressions, soit une liste non vide de déclarations de variables locales (avec la même syntaxe que pour les champs, sauf la clause **`static`**) suivie du mot-clef **`is`** et d'une liste non vide d'expressions. Les expressions du bloc sont

séparées par le symbole '`;`'. La valeur d'un bloc est celle de sa dernière expression. Un bloc ne peut pas être utilisée comme sous-expression, sauf comme partie **`then`** ou **`else`** d'une conditionnelle ou comme élément d'un autre bloc.

Dans une **affectation**, la cible est un identificateur de variable ou le nom d'un champ (non constant) d'un objet qui peut être le résultat d'un calcul : `x.f(y).val := 3`. Le type de la partie droite doit être compatible avec celui de la partie gauche. Il s'agit d'une affectation de pointeurs et non pas de valeur, sauf pour les classes `Entier` et `Chaine`. La valeur retournée par une affectation est celle de sa partie droite. Les affectations ne peuvent pas être faites en cascade.

L'expression de contrôle de la **conditionnelle** est de type `Entier`, interprétée comme « vrai » si et seulement si sa valeur est non nulle. Les parties `then` et `else` doivent être de types compatibles.

L'affectation a une priorité plus faible que la conditionnelle, elle-même moins prioritaire que les autres constructions.

Les **identificateurs** correspondent à des noms de paramètres, de variables locales ou de champ (`static` ou pas) visibles compte-tenu des règles du langage. Il existe deux 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.

Une **sélection** a la forme `objet.nom` et prend la valeur du champ `nom` de l'objet correspondant. Comme pour l'affectation, `objet` peut être en fait une expression dont la valeur sera une instance d'une classe dans laquelle le champ devra être visible

Les **constantes** littérales sont les instances des classe prédéfinies `Entier` et `Chaîne.`

La construction **`new`** *nom-de-classe(arguments)* crée dynamiquement et renvoie un objet de la classe considérée après lui avoir appliqué le constructeur de la classe. La liste d'arguments est obligatoire et doit être conforme au profil du constructeur de la classe.

Les **expressions arithmétiques et de comparaison** sont construites à partir des opérateurs unaires et binaires classiques, avec leurs précédences et associativités habituelles; les opérateurs de comparaison ne sont pas associatifs. Ces opérateurs binaires ou unaires ne sont définies que pour les éléments de la classe `Entier`. On note `<>` l'opérateur « différent de » entre entiers.

Les **envois de message** correspondent à la notion habituelle en programmation objet: association d'un message et 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 bien sûr se faire en cascade comme dans `o.f().g(x.h()*2, z.k())`. Un bloc ne peut pas être argument dans un envoi de message.

La **portée des variables** déclarées dans un bloc est restreinte à ce bloc. Plus généralement, les règles de portée sont celles des langages classiques. Pour une méthode, les paramètres ne sont visibles que dans le corps de cette méthode (un paramètre ne peut par exemple pas apparaitre dans l'expression qui définit la valeur par défaut d'un autre paramètre de la liste).
Le **contrôle de type** est à effectuer modulo héritage. La **conformité des appels de méthodes** ou de constructeurs tient compte de la présence d'éventuelles valeurs par défaut.

Aspects lexicaux : 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. A l'aide de `flex` et `bison`, écrire un analyseur lexical et un analyseur syntaxique de ce langage.
Dans cette première partie, on ne s'intéresse donc qu'à la correction lexicale et syntaxique du programme fourni en entrée (et non pas aux « vérifications contextuelles » ou à la génération de code). Il est par contre important que le résultat de l'analyse syntaxique, par exemple la construction et l'impression d'un arbre syntaxique, permette de contrôler la correction des analyseurs produits.

2. Dans une deuxième étape il vous faudra enrichir votre grammaire avec des fonctions associées aux règles de grammaire afin d'effectuer l'ensemble des vérifications contextuelles et la génération de code pour une machine abstraite dont une description vous sera fournie. Un interprète du code de cette machine vous sera aussi fourni pour tester le code que vous produirez.
La fourniture associée à cette étape sera un dossier expliquant les choix d'implémentation principaux (6 pages maximum !). Il devra aussi remettre une distribution comportant :

- les sources commentés
- 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 compilateur prend le nom du fichier source et le nom du fichier de code engendré sur la ligne de commande.
- Vos fichiers d'exemples, exécutables automatiquement (compilation + 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 intelligemment ce qui peut l'être entre les différentes aspects de la réalisation, que nous détaillons ci-dessous :

- construction des analyseurs lexicaux et syntaxiques et vérification de leur correction
- définition d'une structure pour stocker les éléments d'un programme source (classe, champ, méthode, expression, etc) avec leurs propriétés. Définition des structures d'arbres abstraits associées (ou toute forme équivalente).
- construction des arbres abstraits à partir de l'analyseur syntaxique
- annotation des arbres abstraits pour représenter les informations de portée, de type et gestion des vérifications associées
- compréhension du fonctionnement de la machine virtuelle et de son simulateur
- définition de l'organisation dynamique de la mémoire (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 et de la liaison dynamique).
- génération de code à partir des arbres abstraits annotés et de l'organisation de la mémoire mise en place
- implémentation des classes prédéfinies
- test de ces différents aspects sur votre batterie d'exemple (quelques exemples supplémentaires seront mis à disposition mais il vous appartient de vous constituer votre propre base d'exemples).

Quelques exemples de programmes seront fournis *Attention: il peut rester des erreurs résiduelles (syntaxiques ou contextuelles) ou des imprécisions. Donc en cas de doute, posez la question !*