Analyse sémantique

Mirabelle Nebut

Bureau 332 - M3
mirabelle.nebut at lifl.fr

2011-2012

On a vu...

Comment calculer des données à partir d'une grammaire algébrique :

- grammaire attribuée
- attributs
- actions associées aux productions

On a vu. . .

La grammaire attribuée ne dit pas :

- quand effectuer les actions
- dans quel ordre effectuer les actions

On a vu...

Ce n'est pas facile d'écrire une grammaire attribuée :

- l'information circule entre les noeuds via les attributs
- les attributs hérités ne sont pas intuitifs
- il n'y pas de notion de "variable globale".

Rapidement les actions rendent la grammaire illisible.

Ce qu'on va voir maintenant

Comment, en pratique, en utilisant un outil type Cup :

- on n'utilise que des attributs synthétisés
- on peut utiliser une variable globale
- on peut alléger la grammaire avec une bonne COO
- les actions construisent un modèle sémantique, les calculs seront faits dessus par la suite.

Et ce n'est pas si compliqué !

Analyse sémantique de INIT

Le contrôle de type et des déclarations de INIT :

- pas de double déclaration
- toute variable utilisée a été déclarée
- toute variable utilisée l'est en accord avec son type

Dans l'archive du TP4 : `syntaxePlanning.cup`

Utilisation d'une table des symboles

Structure de données centrale en compilation.

TableSymboles
ajoutIdentificateur(String, Type) contientIdentificateur(String) : boolean getTypeIdentificateur(String) : Type

Type
ENTIER LISTE PROGRAMME

L'exemple d'INIT

Cup et les grammaires attribuées

Un peu de COO

Attributs synthétisés, pour simplifier

CUP construit l'arbre syntaxique en ordre postfixe

= des feuilles vers l'axiome

Convient bien à des attributs synthétisés

En pratique, on peut utiliser des hérités, mais les effets de bords sont compliqués à expliquer.

Petits arrangements avec le formalisme

CUP permet d'écrire n'importe quelle grammaire S-attribuée.

Pour des raisons de Génie Log, l'approche est un peu différente de l'approche de théorie du langage.

Attributs

Un seul attribut par symbole, sous-type de `Object`.

Pour déclarer un attribut, on donne uniquement son type (objet) :

```
terminal String IDENT;
non terminal ArrayList<String> listeIdent;
```

L'attribut du non-terminal en partie gauche est par convention `RESULT`.

Variable globale à l'analyseur syntaxique

Puisque l'analyseur est une classe Java...

... on peut utiliser ses attributs de classe.

Permet de ne pas "ballader" la table des symboles associée à chaque non-tal.

```
// recopié ds la classe interne qui gère les actions
action code {:
private TableSymboles tableSymboles;
:}
```

Ne doit pas remplacer les attributs, mais bien pratique.

Actions

Les actions sémantiques sont écrites en Java.

Elles sont placées :

- entre {: :}
- en fin de production avant le ;
- mais avant un %prec.

```
entete ::= PROG IDENT:nom FININSTR {:
    this.tableSymboles = new TableSymboles();
    this.tableSymboles.ajoutIdentificateur(nom,
                                           Type.PROGRAMME);
:}
```

Exécution des actions

Une action est exécutée lors de la "réduction" de la production associée.

cad quand la partie droite de la production a été reconnue.

```
programme ::= entete listeDecl listeInstr {:
    System.out.println("*****\n" + this.tableSymboles);
:}
```

Cette action est la dernière à être effectuée.

Actions et accès aux attributs

Le nom des attributs est local à une production (= var locale)

Accès à l'attribut d'un symbole par " :"

```
listeIdent ::= IDENT:nom {:
   RESULT = new ArrayList<String>();
   RESULT.add(nom);
:}
              | IDENT:nom SEPVAR listeIdent:liste {:
   RESULT = liste;
   RESULT.add(nom);
:}
```

Attributs de l'axiome

L'analyseur syntaxique décoré fournit les attributs de l'axiome.

En supposant que l'axiome de la grammaire pour `Init` possède un attribut de type `Programme` :

```
import java_cup.runtime.Symbol;
...
ParserInit parser = new ParserInit(scanner);
...
Symbol s = parser.parse();
Programme p = (Programme) s.value;
...
```

Grammaire à opérateurs

CUP permet d'écrire des grammaires à opérateurs sous forme ambiguë. . .

en précisant la priorité et l'associativité des opérateurs

On peut donc aussi attribuer une grammaire ambiguë.

C'est très pratique !

L'exemple d'INIT

Cup et les grammaires attribuées

Un peu de COO

Constatation : la grammaire est illisible

Tout de même plus lisible que le TP1 !

Mais si le code des actions est gros, on ne s'y retrouve plus.

Solution : créer une classe qui fait le travail.

Exemple

Pour INIT : une classe `ControleurType`.

Le code d'une action devient une méthode.

On peut maintenant factoriser le code.

La grammaire est plus lisible.

Constatation : la grammaire reste illisible !

En effectuant toute l'analyse sémantique pendant l'an. syntaxique :

```
gauche ::= droite {: code module 1
                      code module 2
                      // code module supprimé
                      ...
                   :}
```

- analyseur syntaxique difficile à lire ;
- difficilement maintenable ;
- difficile à étendre.

Solution logicielle

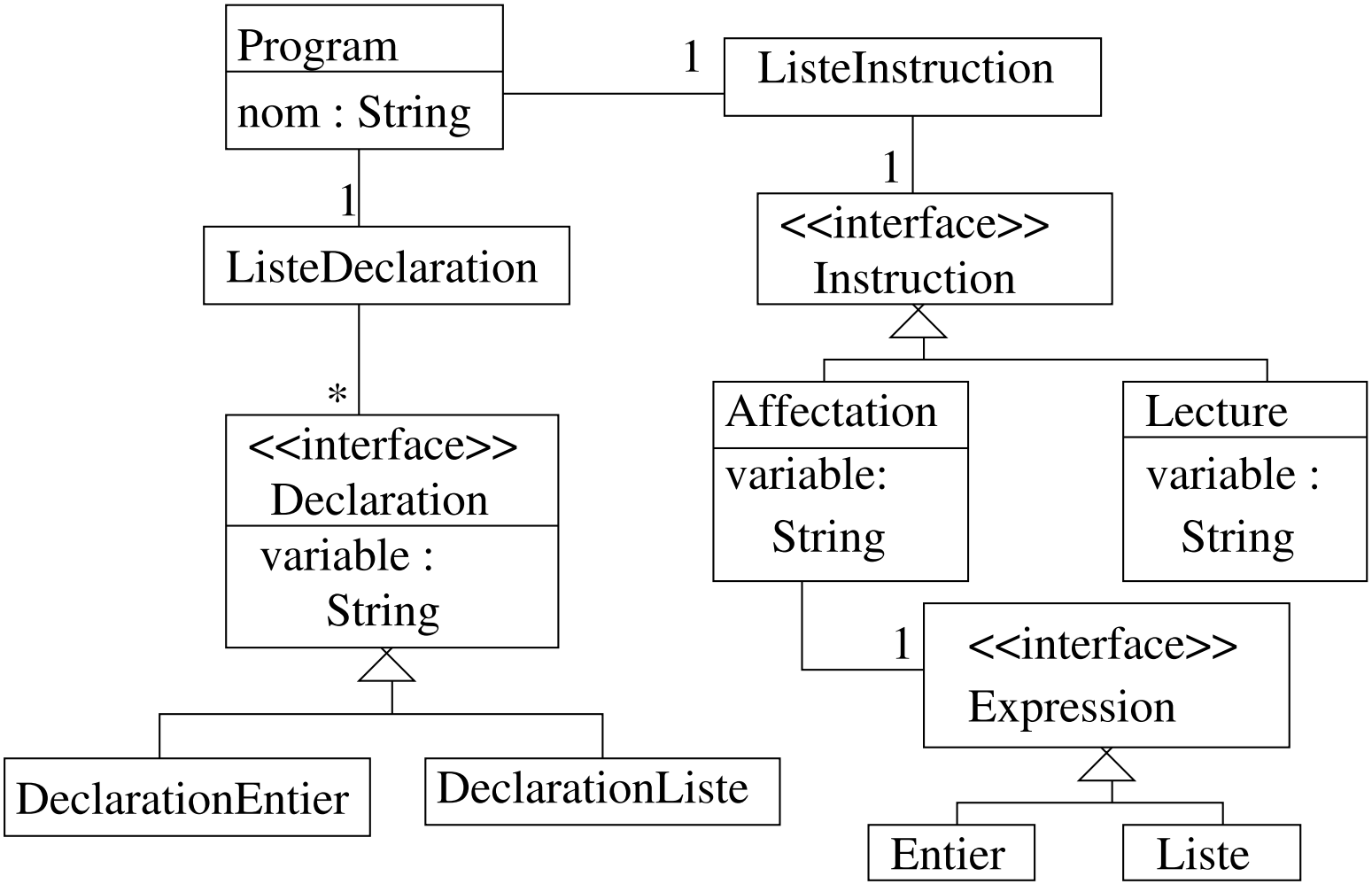
Construire une structure de données qui représente le texte source.

Cette structure fait l'interface entre partie avant et partie arrière.

Les analyses sémantiques s'appliqueront à cette structure.

[Fowler11] parle de modèle sémantique (= des classes Java).

Exemple pour Init



Pattern Visiteur

C'est **le** pattern utilisé en COMPIL.

Principe :

- coder chaque analyse dans une classe indépendante, un "Visiteur"
- chaque visiteur parcourt / "visite" la structure de données
- la structure de données est "Visitable"

Vous verrez ça en COO !