CSC4251_4252 : Arbre de Syntaxe Abstraite

Pascal Hennequin, J. Paul Gibson, Denis Conan

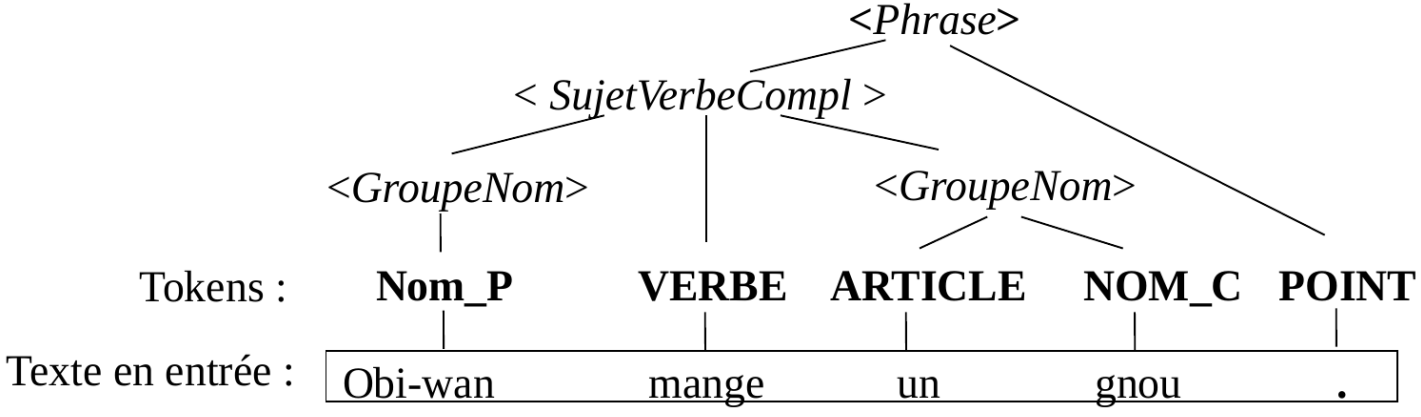
Novembre 2024

Sommaire

1. Arbre Syntaxique
2. CST vs AST
3. Arbre et Analyse LR
4. Arbre et Typage Objet
5. Visiteur
6. Classes JAVA du cours
7. (Bonus) Retour au source

1 Arbre Syntaxique I

- **L'Arbre Syntaxique est :**
  - La preuve de la validité syntaxique de l'entrée
  - Le support d'une part importante de la sémantique de l'entrée
  - La structure de données transmise dans les deux phases suivantes de la compilation : analyse sémantique et génération de code intermédiaire
- *Rappel :*
  - Nœud = règle de production
  - Feuille = *token*



1 Arbre Syntaxique II

Syntaxique

Sémantique

Bon pour génération

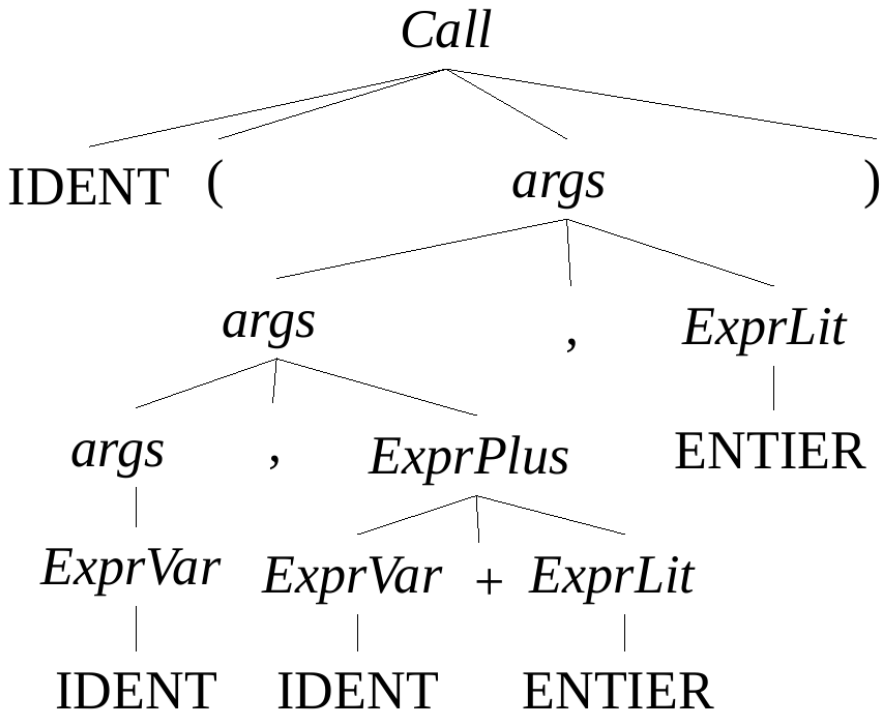
2 CST vs AST I

- ***Concrete Syntax Tree* (CST) :**
  - Forme adaptée pour la validation syntaxique
  - Pas de valeurs sémantiques pour les *tokens*
  - Les nœuds respectent strictement les règles de grammaire
  - Construction automatique avec les outils d'analyse syntaxique
- ***Abstract Syntax Tree* (AST) :**
  - Forme porteuse de la sémantique
  - Intègre les valeurs sémantiques des *tokens*
  - Supprime les *tokens* inutiles (séparateurs, parenthèses, etc.)
  - Se libère du « tyran algébrique »
    - « `OPBIN` » *vs.* « $\{$`PLUS,MOINS,...`$\}$ »
    - Liste « `1 2 3` » *vs.* « `List ( List ( List (List(`ϵ`), 1) , 2) , 3)` »

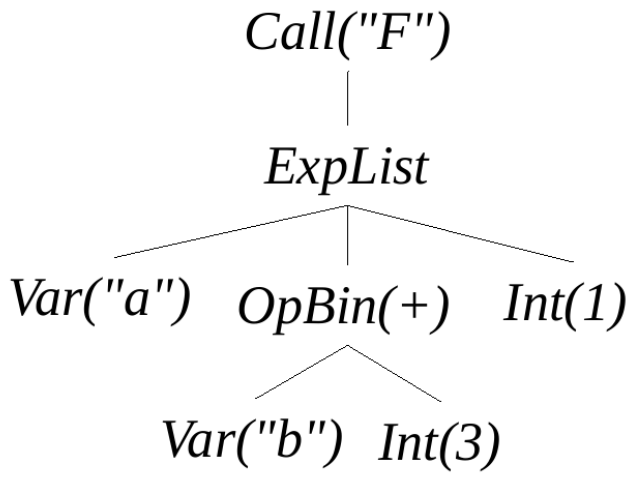
2 CST vs AST II

- Exemple : « `F(a, b + 3, 1)` »

CST



AST



3 Arbre et Analyse LR

Analyse LR

- Construction ascendante de l'arbre
- Action sémantique = nouvel arbre à partir des fils déjà existants

```
/* Exemple dans une spécification CUP */
nonterminal Arbre v, s1, s2, ... ;
v :=  s1 :x1 s2 : x2 ... /* RegleN */
      {: RESULT = new Arbre (« RegleN », x1, x2, ...) ; :}
;
/* Cas des feuilles */
temninal [type] Tj, ... ;
v :=  ...Tj... /* RegleM */
      {: RESULT = new Arbre (« RegleM », ... , new Arbre(« Tj »), ...) ; :}
;
```

4 Arbre et Typage Objet

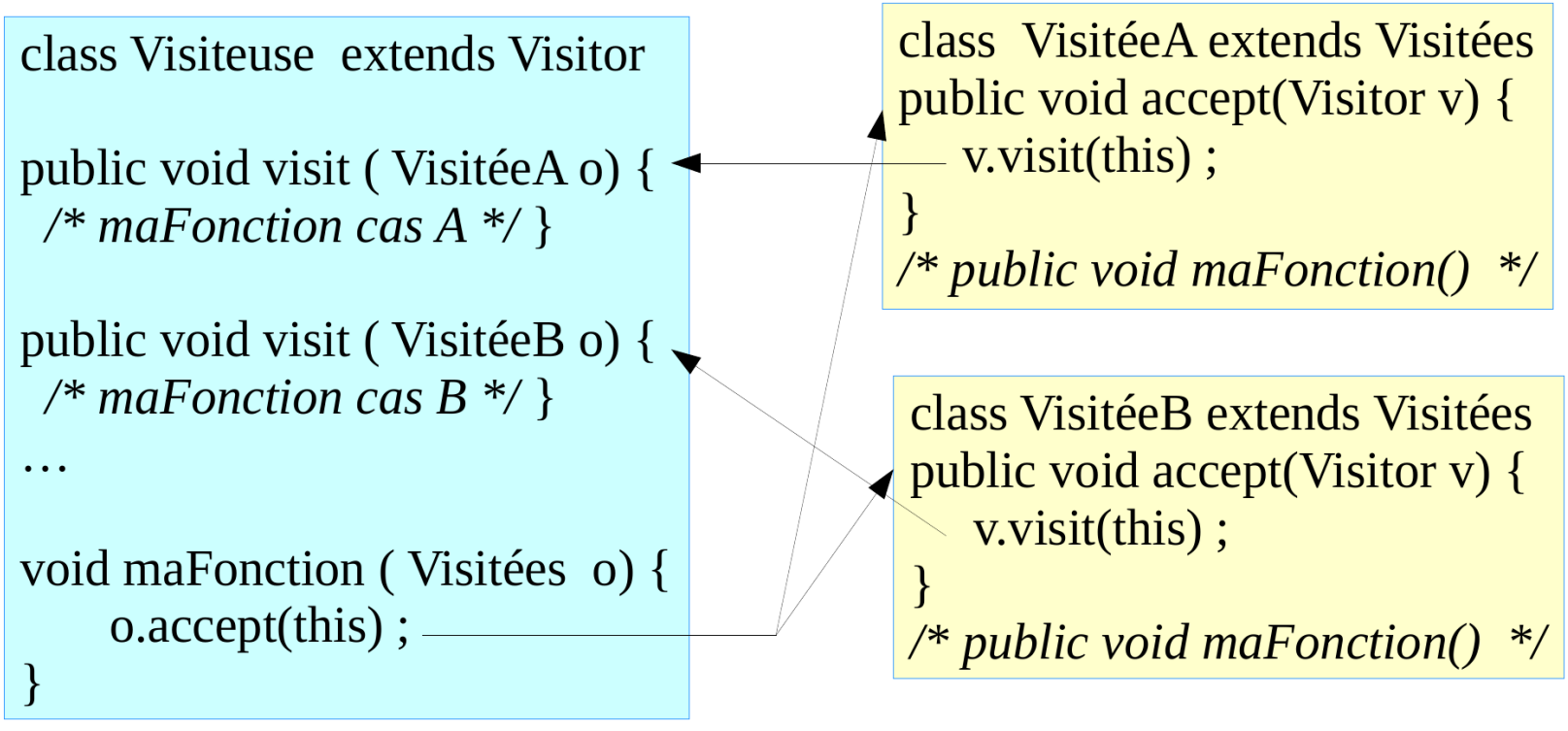
- **Utilisation d'un langage objet :**
  - Typage des nœuds
    - « new ArbreRegle$N$(x_1,...) » vs « new Arbre("Regle$N$", x_1,...) »
  - Héritage et productions alternatives dans la grammaire
    - « Arbre$V$ = classe abstraite » pour « ArbreRegleV_1, ArbreRegleV_2, ... »
  - Redéfinition (*overriding*) et liaison dynamique
    - ArbreV_T = new ArbreRegleV_i() ;
      T.xxx() ; /* méthode xxx de la classe concrète de T (ArbreRegleV_i) */
- **Réalisation d'une fonction sur l'arbre**
  - Méthodes d'objets dans les nœuds et parcours récursifs
  - Variante : patron de conception « Visiteur » (*Design pattern Visitor*)
    - But : obtenir le même résultat mais avec le code des méthodes d'objets de chaque nœud regroupé dans une seule classe

5 Visiteur I

- **Ajouter une fonction à une hiérarchie de classes (classes visitées)**
  - Non pas en utilisant le schéma naturel OO des méthodes d'instance dans chaque classe (héritage, redéfinition)
  - Mais en utilisant une classe externe (classe visiteuse)
- **Utilité**
  - Regrouper l'algorithme de la fonction dans une même classe
  - Ne pas modifier les classes visitées pour chaque nouvelle fonction
- **Problème de liaison dynamique : comment connaître dans la classe visiteuse, la classe concrète des objets visités ?**
  - Solution « pas très classe » : Imbrications de « `if (x instanceof X1)` »
  - Solution « plus classe » : définir une méthode dans les classes visitées qui sert uniquement à connaître la classe concrète

5 Visiteur II

- « *Accept* et *Visit* sont dans un bateau »



(Solution « plus classe » ! ?)

6 Classes JAVA du cours I

- `AstNode` : **Classe abstraite ancêtre des « visitées »**

```
/** Patron Visiteur */
public abstract void accept(AstVisitor v);

/** Constructeur varargs */
protected AstNode(AstNode ... fils )

/** Itérable avec for(AstNode f : node) */
public Iterator<AstNode> iterator()
public int size()

/** Impression d'un nœud */
public String toString()

/** Impression Arbre */
public String toPrint()
```

6 Classes JAVA du cours II

- `AstList<R>` : **fils homogènes et construction itérative**

```
class AstList<R extends AstNode> extends AstNode {
  public void accept(AstVisitor v) { v.visit(this); }
  /** Construction itérative avec ajout en fin de liste */
  public AstList<R> add(R node)...
}
```

- `Ast` : **Classe concrète de base (pour tests ou CST)**

```
class Ast extends AstNode {
  public void accept(AstVisitor v) { v.visit(this); }
  public final String label;
  public Ast(String label, AstNode... f) { super(f); this.label =
  public String toString() { return label; }
}
```

6 Classes JAVA du cours III

- `AstVisitor` : **Interface des classes « Visiteuses »**
  - Méthodes abstraites `visit()` pour chaque classe visitable de l'AST

```
interface AstVisitor {

  <T extends AstNode> void visit(AstList<T> n);

  void visit(Ast n);

  /* ... idem pour chaque classe visitable */

}
```

6 Classes JAVA du cours IV

- `AstVisitorDefault` : « visiteur » générique de l'AST
- Classe parente pour des « visiteurs » qui font des calculs par parcours en profondeur de l'arbre
- Les méthodes `visit()` sont alors à redéfinir (*override*)... Ou pas !

```
class AstVisitorDefault implements AstVisitor {
  public void defaultVisit(AstNode n) {
    for (AstNode f : n) f.accept(this);
  }

  public <T extends AstNode> void visit(AstList<T> n) {
    defaultVisit(n);
  }
  public void visit(Ast n) { defaultVisit(n); }
  /* ... idem pour chaque classe visitable */
}
```

7 (Bonus) Retour au source

- **Propager les informations de positions dans le fichier source**
  - JFlex vers CUP : transparent avec `JflexCup.include`
    - `ComplexSymbolFactory` et `TOKEN()`
- CUP vers AST :
  - Option « `-locations` » de CUP requise (Cf. fichier Maven `pom.xml`)
  - Méthode `addPosition()` de `AstNode` :

```
v   :=   s1:aa ... sn:zz
         {: RESULT= new Arbre(aa,...,zz);
            RESULT.addPosition(aaxleft, zzxright); :}
;
```

  - Par exemple : `AstNode.toString()` $\rightsquigarrow$ `RopBin[16/10/293-16/21/304](+)`