Utilisation de visiteurs pour l'analyse sémantique avec Tatoo

michel.chilowicz@univ-mlv.fr

Clyde Auditorium à Glasgow, photo par Ross Goodman (CC-By)

Analyse sémantique

Différentes phases :

- Découvertes des types (enter)
- Vérification ou inférence de types
- Analyse statique du flux d'exécution
- Désucrage syntaxique
- Optimisations diverses
- …

➔ Étape suivante : génération de code

Comment réaliser l'analyse sémantique ?

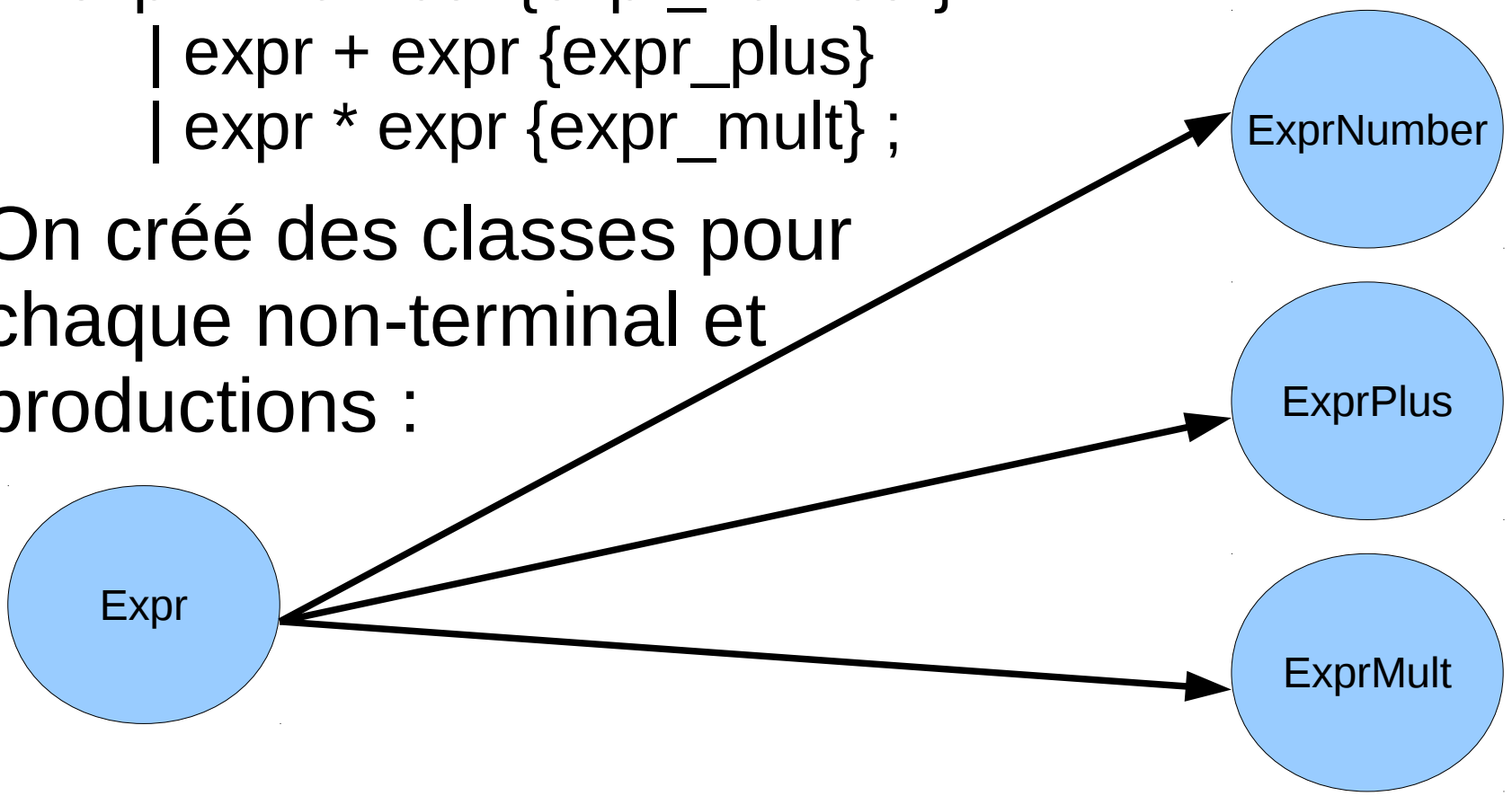
1) Construction de l'AST de l'unité de compilation
2) Visite de l'AST pour récolter les informations nécessaires et construire des structures :
   - Table de symboles (type, variable)
   - Graphe de flux d'exécution
   - AST réécrit (désucrage syntaxique)
   - ...

Visiter l'AST... avec un visiteur !

- Design pattern classique (défini dans le livre du « Gang of Four »)
- Permet de séparer proprement le code de traitement de la structure (l'AST) de la structure elle-même
- Un visiteur pour chaque opération d'analyse
- Traitement de chaque type de nœud dans une méthode distincte (types de nœuds organisés hiérarchiquement)

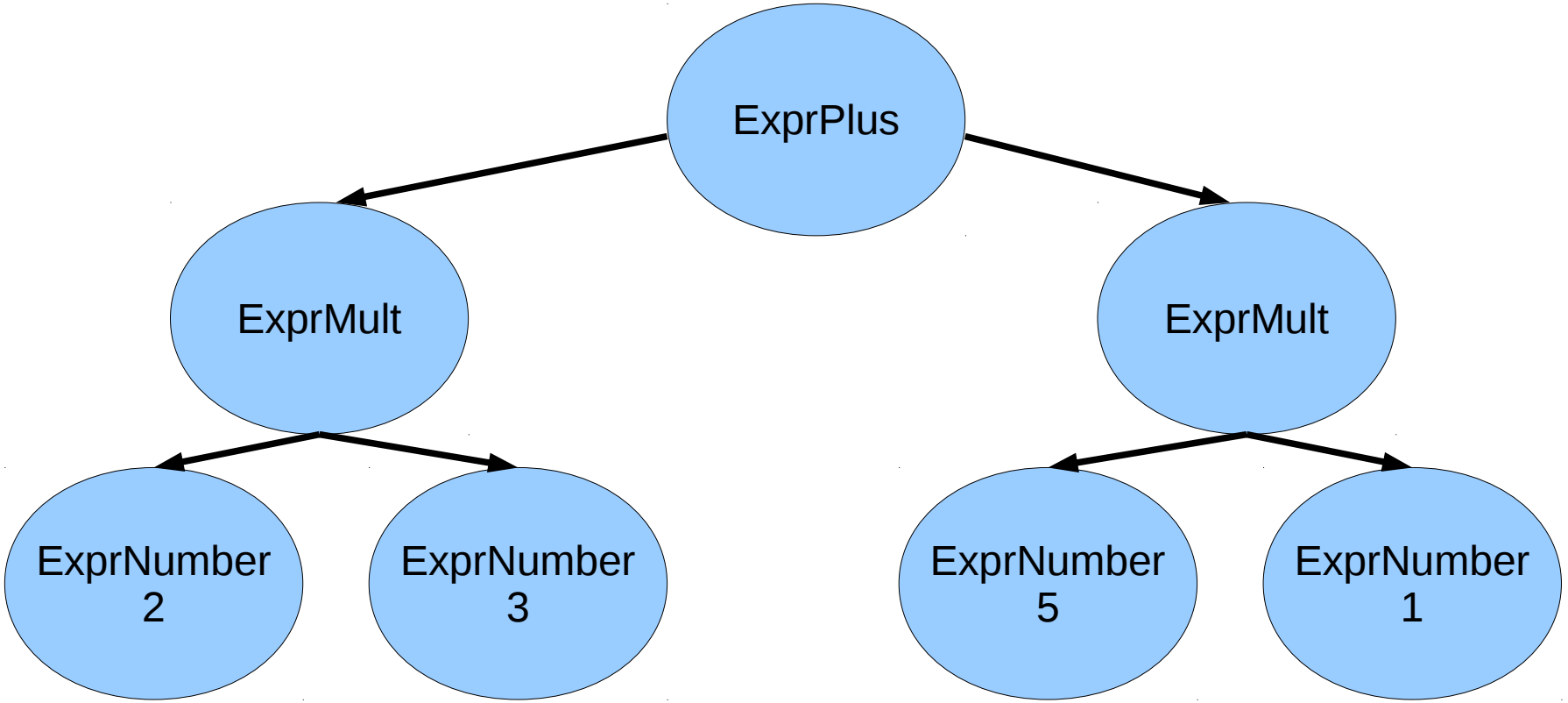
Un exemple de grammaire

- Grammaire d'expressions sur les entiers :
  - expr = number {expr_number}
    | expr + expr {expr_plus}
    | expr * expr {expr_mult} ;
- On créé des classes pour chaque non-terminal et productions :



Génération de l'AST

- Exemple d'expression :
  2 * 3 + 5 * 1
- AST généré par l'application de la grammaire :



Visiteur avec « instanceof »

- On souhaite construire un visiteur évaluant une expression (nœud Expr)

```
public class EvalVisitor extends Visitor<Integer>
{
	public Integer visit(Expr expr)
	{
		if (expr instanceof ExprNumber)
		{
			return ((ExprNumber)expr).getValue() ;
		} else if (expr instanceof ExprPlus)
		{
			ExprPlus exprPlus = (ExprPlus)expr ;
			return visit(exprPlus.getExpr()) + visit(exprPlus.getExpr2()) ;
		} else if (expr instanceof ExprMult)
		{
			ExprMult exprMult = (ExprMult)expr ;
			return visit(exprMult.getExpr()) * visit(exprMult.getExpr2()) ;
		}
	}
}
```

Inconvénients de l'utilisation d'*instanceof*

- Code peu élégant et peu orienté objet
- Difficilement maintenable en cas de changement de nœuds de l'AST
- Lenteur à l'exécution

➔ Solution : *double dispatch*

Visiteur avec *double dispatch*

- On écrit pour chaque type de nœud une méthode de visite :
  - public Integer visit(Expr expr) { … }
  - public Integer visit(ExprPlus expr) { … }
  - ...
- Pour un noeud déclaré statiquement *Expr n*, *visit(n)* appelle toujours *visit(Expr expr)*, or *n* peut avoir dynamiquement le type *ExprNumber*, *ExprPlus*...

➔ Il faut ajouter dans chaque nœud une méthode *accept(Visitor<R> v)* appelant la méthode *v.visit(this)*

Un visiteur d'évaluation

```
public class EvalVisitor extends Visitor<Integer>
{
	public Integer visit(Expr expr)
	{
		assert(false) : « Sould never be called »
	}

	public Integer visit(ExprNumber expr)
	{
		return expr.getValue() ;
	}

	public Integer visit(ExprPlus expr)
	{
		return expr.getExpr().accept(this) +
			expr.getExpr2().accept(this) ;
	}

	public Integer visit(ExprMult expr)
	{
		return expr.getExpr().accept(this) *
			expr.getExpr2().accept(this) ;
	}
}
```

```
public interface Visitable
{
	public <R> R accept(Visitor<R> v) ;
}


public class ExprPlus implements Visitable<R>
{
	private Expr expr ;
	private Expr expr2 ;

	public ExprPlus(Expr expr, Expr expr2)
	{
		this.expr = expr ;
		this.expr2  = expr2 ;
	}

	public Expr getExpr() { return expr ; }

	public Expr getExpr2() { return expr2 ; }

	public <R> accept(Visitor<R> v)
	{
		return v.visit(this) ;
	}
}
```

Code à dupliquer sur toutes les classes de noeuds

Interface ou classe générée Visitor

- Interface Visitor : on définit les méthodes *visit* pour chacun des types. Si un type de nœud est ajouté, il faut ajouter sa méthode *visit* et implanter cette méthode sur tous les visiteurs : fastidieux.
- Classe Visitor : générée par Tatoo, consiste à implanter par défaut sur une méthode *visit(B)* un appel à *visit(A)* (A étant ancêtre de B).

Visitor<R, P, E extends Exception> de Tatoo

- R : type de retour des méthodes *visit*
- P : type du paramètre passé en argument des méthodes *visit*
  - ➔ La méthode visit du noeud parent peut transmettre des données à la méthode visit des noeuds enfants (ex : table de symboles...)
- E : type de l'exception
  - ➔ Il est possible de lever une exception (court-circuit) propagée aux méthodes *visit* des parents

Exemple de visiteur généré par Tatoo

```
public class Visitor<_R,_P,_E extends Exception>
{

  public _R visit(Start start,_P _param) throws _E {
    return visit((Node)start,_param);
  }

  public _R visit(ExprNumber expr_number,_P _param) throws _E {
    return visit((Expr)expr_number,_param);
  }


  public _R visit(ExprPlus expr_plus,_P _param) throws _E {
    return visit((Expr)expr_plus,_param);
  }

  public _R visit(ExprMult expr_mult,_P _param) throws _E {
    return visit((Expr)expr_mult,_param);
  }

  protected _R visit(Expr expr,_P _param) throws _E {
    return visit((Node)expr,_param);
  }

 protected _R visit(Node node,_P _param) throws _E {
    throw new AssertionError("default visit not defined");
 }
}
```

Les visites sur les nœuds correspondant aux productions appellent par défaut une visite sur leur non-terminal (*Expr*)

Les non-terminaux à plusieurs productions sont *protected*

Visite sur type *Node* racine

Un visiteur compteur de nœuds

- Très simple à implanter en dérivant la classe Visitor générée par Tatoo : une méthode à redéfinir

```
public class NodeCounterVisitor extends Visitor<Integer, Void, RuntimeException>
{
	@Override
	protected Integer visit(Node node, Void param) throws RuntimeException
	{
		int counter = 1 ;
		for (Node child : node.nodeList()) counter += child.accept(this, param) ;
		return counter ;
	}
}
```