

# La syntaxe abstraite de Zwei

Martin Odersky

28 novembre 2005  
version 1.1

# La syntaxe abstraite de Zwei

```
P = Program { D } E
D = ClassDef name [ name ] { M }
M = FieldDecl name T
  | MethodDef name { name T } T E
T = ClassType name
  | IntType
  | NullType
S = While E { S }
  | Var name T E
  | Set name E
  | Do E
  | PrintInt E
  | PrintChar E

E = Ident name
  | New name { E }
  | Select E name
  | Call E name { E }
  | IntLit int
  | NullLit
  | Unop U E
  | Binop B E E
  | ReadInt
  | ReadChar
  | If E E E
  | Block { S } E

U = Not | Neg
B = Add | Sub | Mul
  | Div | Mod | Eq
  | Ne | Lt | Le
  | Gt | Ge | And
```

Quelques alternatives de la grammaire méritent un commentaire :

- dans `ClassDef name1 [ name2 ] { M }`, l'attribut `name2` est le nom de la super-classe, qui peut être vide le cas échéant ;
- `Select` est l'accès à un champ d'une instance, `Call` est l'appel d'une méthode d'une instance.
- `Var` est l'instruction (*statement*) qui déclare une variable, `Set` est celle qui redéfinit la valeur d'une variable et `Do` est une instruction sans définition ou création de variable.

# Obtenir un arbre de syntaxe abstraite (AST)

L'AST de Zwei est obtenu en définissant :

- une classe abstraite `Tree` ;
- une sous-classe abstraite de `Tree` pour chaque non-terminal `P`, `D`, etc. de la grammaire ;
- une sous-classe `case` concrète de la classe du non-terminal pour chaque alternative `Program`, `ClassDef`, `FieldDecl`, `MethodDef`, etc. de la grammaire.

# Définition de Tree pour Zwei

```
abstract class Tree {
  private var p: Int = Position.UNDEFINED;
  def pos: Int = p;
  def setPos(p: Int): this.type = { this.p = p; this }
}

/** P = Program { D } E */
case class Program (classes: List[ClassDef], main: Expr) extends Tree;

/** A common superclass for all definitions */
abstract class Def extends Tree;
// ...

/** A common superclass for tree nodes designating types */
abstract class TypeTree extends Tree;

/** T = Int */
case class IntType extends TypeTree;
// ...

/** A common superclass for all statements */
abstract class Stat extends Tree;

/** S = While E { S } */
case class While(cond: Expr, stats: List[Stat]) extends Stat;

/** S = Var name T E */
case class Var(varname: Name, vartype: TypeTree, init: Expr) extends Stat;
// ...
```

On notera dans le définition de `Tree` pour `Zwei` que :

- le constructeur de chaque classe se compose des différents sous-arbres de l'alternative correspondante dans la grammaire ;
- l'option est exprimée par la classe `Option` de Scala :  
[ `name` ] dans la grammaire devient `Option[Name]` dans l'arbre ;
- la répétition est exprimée par des listes : { `T` } dans la grammaire devient `List[T]` dans l'arbre ;
- les identifiants sont représentés par des instances de `Name` :  
pour le moment, vous pouvez considérer `Name` comme une simple chaîne de caractères — mais l'utilisation d'une classe dédiée sera très utile par la suite.

Le champ `pos` contient la position courante dans l'arbre — importante pour produire des messages d'erreur :

- `pos` est commun à tous les types d'arbres ; c'est pourquoi il est membre de la classe `Tree`.
- `pos` ne fait pas partie du constructeur des classes, ceci afin de ne pas «polluer» le filtrage de motif :
  - `setPos` doit être appelée juste après la construction de la classe — c'est une sorte de post-constructeur.
  - `setPos` retourne `this` : il est ainsi possible de chaîner l'appel à `setPos` directement au constructeur :

```
val prog = Program(classes, main) setPos pos;
```