Compilation

Y a-t-il une vie après le parsing ?

Jules Chouquet

SOM2IF15 – 2024

Où on en est :

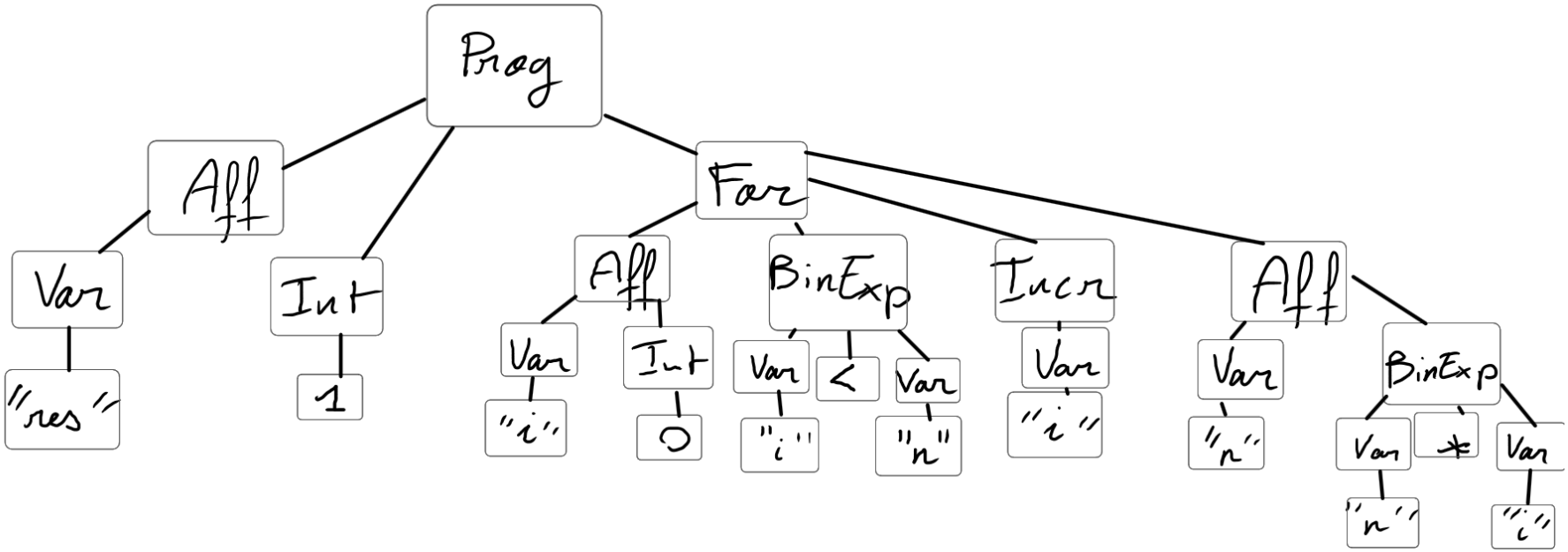
D'où l'on part : un programme P écrit dans un langage de haut niveau, et traduit en arbre syntaxique par ANTLR.
Où l'on va : toujours vers de l'assembleur, à terme.

Objet de ce cours

Comprendre comment manipuler concrètement des données structurées sous forme arborescente à l'aide d'un patron de conception (*design pattern*).
Examiner l'implémentation de ces concepts sur un arbre syntaxique généré par ANTLR, et voir comment appliquer plusieurs algorithmes à partir de cette conception.

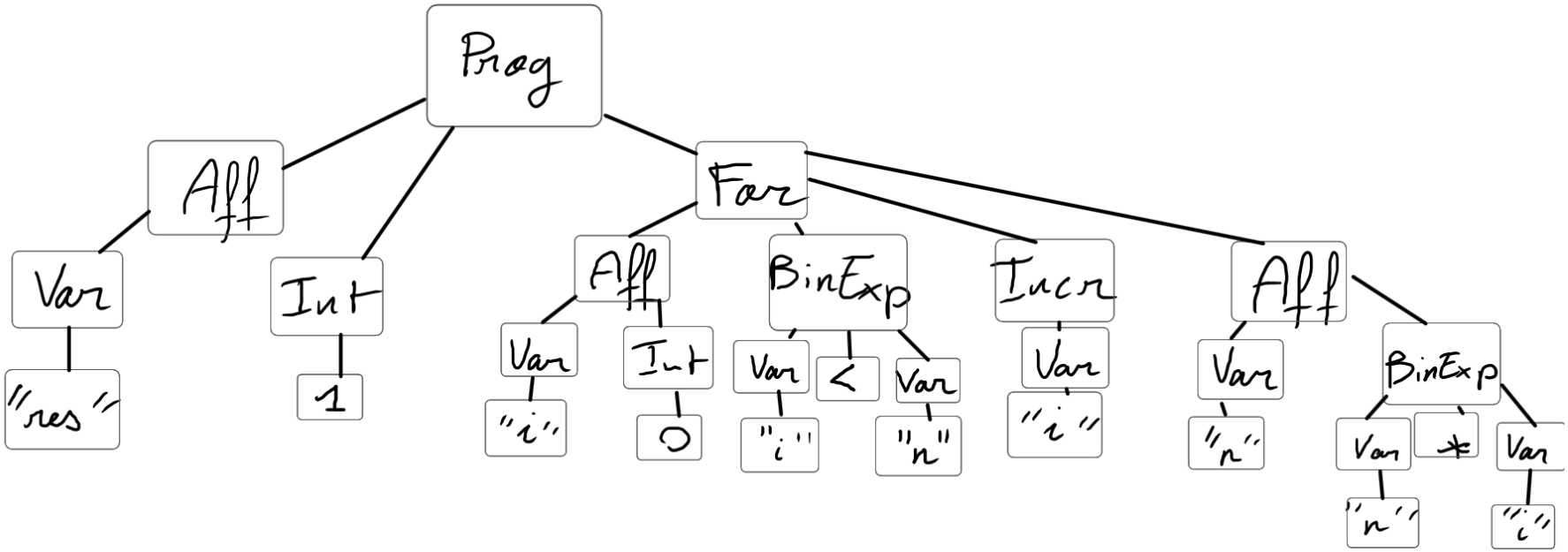
La difficulté

Les arbres de syntaxe pour des langages de programmation peuvent être lourdingues, pénibles à manipuler.



La difficulté

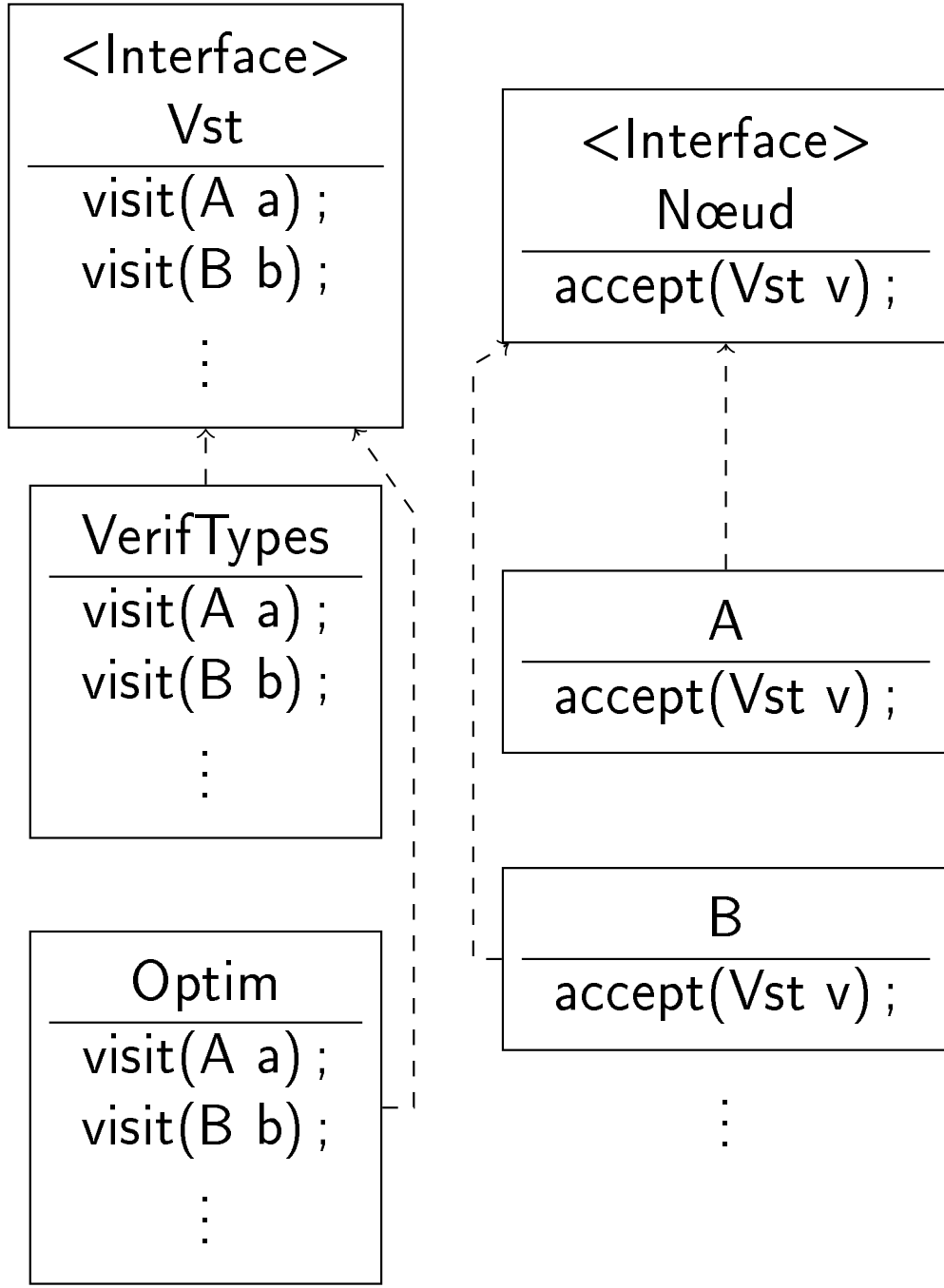
Les arbres de syntaxe pour des langages de programmation peuvent être lourdingues, pénibles à manipuler.



Heureusement, il existe une solution pour travailler avec de façon générique et efficace. . .

Les visiteurs !

Les visiteurs !



Idée générale

Gagner en souplesse et maintenabilité

- L'algorithme de parcours de l'arbre est découplé de l'implémentation de l'arbre.
- Chaque nœud appelle le parcours de ses fils et retourne une valeur du type approprié.
- Les types de nœuds différents sont visités sans faire appel à des cascades de `switch` ou `instanceof`.
- Écrire des algorithmes différents pour la même structure devient particulièrement commode.

Que visite-t-on ?

La génération de parseurs par ANTLR comporte une option `-visitor`, qui génère l'interface du visiteur pour l'arbre de syntaxe généré.

Regardons à quoi doivent ressembler les méthodes avant de voir l'implémentation de visiteurs.

[demo]

L'importance des annotations dans ANTLR

Annoter les règles alternatives par des noms pour chaque cas permet à ANTLR de générer des sous-classes de la règle générale, et ainsi d'avoir un comportement différent selon les cas : le contexte, le nombre de branches, ne sont pas toujours les mêmes.

On ne va écrire qu'un seul visiteur qui implémente celui généré par ANTLR, pour transformer l'arbre de syntaxe généré en un arbre de syntaxe abstraite.

Modifications

- Classes maison pour les nœuds de l'arbre abstrait, avec toutes les informations dont on aura besoin, et pas plus.
- En particulier, simplifications si possible

Quelles classes pour l'arbre de syntaxe abstraite (ast) ?

- Une super classe abstraite `Nœud`
- Une classe abstraite pour chaque catégorie syntaxique : `Expression`, `Statement`,...
- Une classe concrète pour chaque feuille de **l'ast générique**

Avant de commencer, il faut avoir déjà conçu l'ast générique de son langage, c'est-à-dire la hiérarchie de catégories syntaxiques décrivant le langage.
Il y a des choix à faire, par exemple : deux classes `ExpPlus` et `ExpMoins`, ou une seule classe `ExpBin` avec un champ `operator` ?[1]

Une fois ces choix faits, ils peuvent être implémentés par une hiérarchie de classes en java, et on peut écrire le visiteur pour transformer les arbres d'ANTLR en ast.

1. Le premier choix peut être fait, comme dans le manuel d'A.Appel, et peut sembler plus simple. Mais forcément, si on veut rajouter un opérateur au langage, il faut changer les classes de l'ast...

Exemple de génération d'ast

En créant nos sous-classe de Nœud, nous allons dans le même temps écrire une interface de visiteur pour chacune d'entre elles.
Les `Nœuds` de l'ast devront disposer d'une méthode `accept`, car bien entendu, nous utiliserons aussi des visiteurs sur les ast, donc autant s'y préparer dès maintenant. [2]

[Exemple avec la syntaxe d'arbres binaires]

2. Mais attention, c'est bien le visiteur fourni par ANTLR qui va nous permettre de construire l'ast. On se rapportera souvent au fichier (généré) `[langage]Parser.java` ainsi qu'à la documentation d'ANTLR

Quelques exemples

Quelques algorithmes implémentés avec un visiteur :

- retourner la profondeur maximale d'entrelacement des blocs.
- afficher le texte du programme de façon standard et lisible

[demos]

Localiser les nœuds dans le texte source

Pour ne traiter par la suite qu'avec l'ast que l'on va construire, on va d'ores et déjà enrichir notre arbre avec les informations de position pour chaque nœud, de façon à délivrer des messages d'erreurs précis par la suite.

Pour cela, on prendra soin de créer une classe dédiée `Position` avec deux champs entiers `ligne` et `colonne`. Chaque `Nœud` aura un champ position, qui sera récupéré à partir de l'arbre généré par ANTLR.

Les méthodes `ctx.start.getLine()`,`ctx.start.getPositioInLine()` appliquées aux contextes permettront de récupérer ces positions.

À partir de maintenant, on ne manipulera plus que l'arbre de syntaxe abstraite de notre programme.

Nous avons les outils nécessaires pour passer aux phases sémantiques de la compilation.

À partir de maintenant, on ne manipulera plus que l'arbre de syntaxe abstraite de notre programme.

Nous avons les outils nécessaires pour passer aux phases sémantiques de la compilation.

À suivre, deux programmes au moins qui devront parcourir l'AST :

- Un algorithme de vérification du typage
- La transformation en un AST de plus bas niveau (la représentation intermédiaire)