Analyse Contextuelle

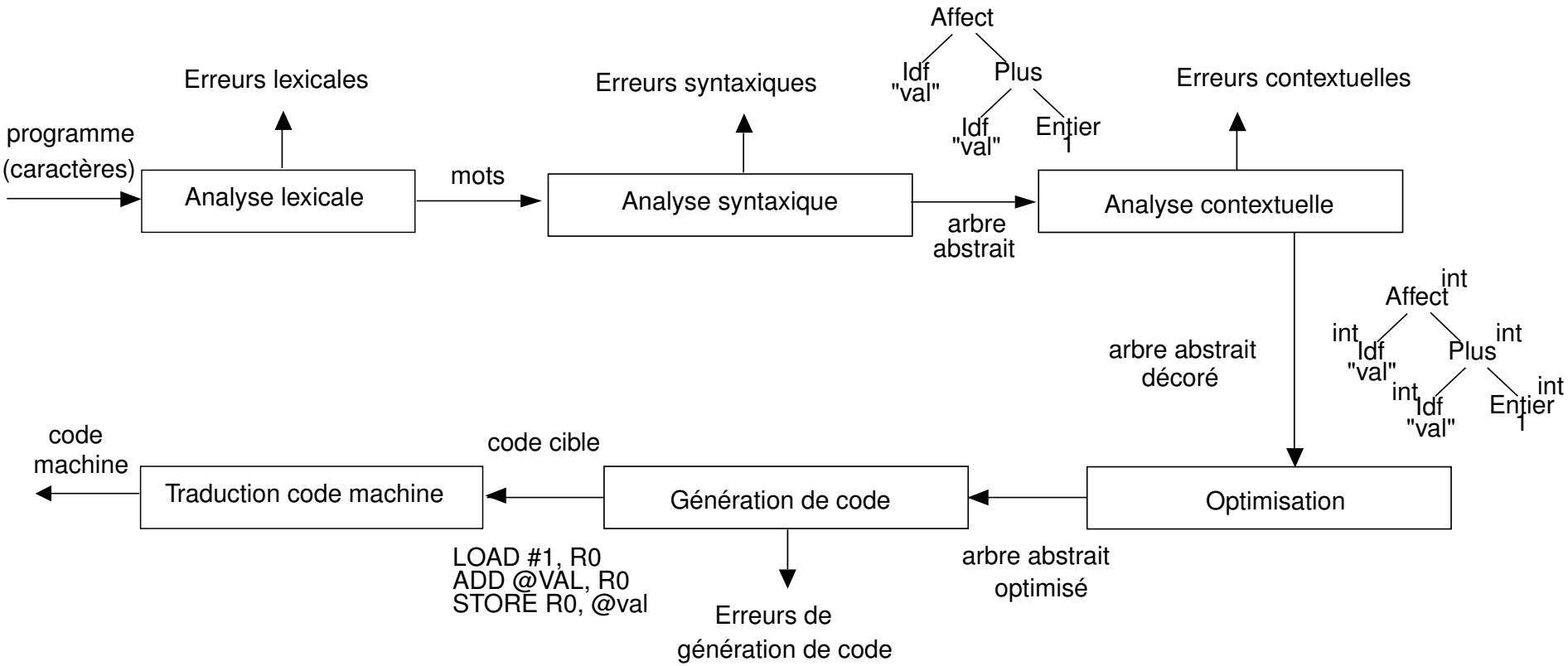
CS410 - Langages et Compilation

Julien Henry
Catherine Oriat

Grenoble-INP Esisar

2013-2014

Phases de la compilation



Rôle de l'analyse contextuelle

1. Vérifier les propriétés contextuelles du langage de programmation
   - Description sémantique du langage
   - Erreur si le programme est incorrect
2. Enrichir l'arbre de syntaxe abstraite du programme pour permettre la génération de code.
   - Un arbre abstrait décoré, résultat de la compréhension du programme

$\Rightarrow$ fonctions récursives sur les arbres de syntaxe abstraite.

Exemple

En Java, les identificateurs qui sont locaux à des méthodes doivent être initialisés avant d'être utilisés.

```
class A {
  void p() {
    int x = x+1;
    // erreur contextuelle : x doit etre initialisé
   }
}
```

```
class A {
  void p() {
    int x = (x=2)*x;
    // OK : opérateurs évalués de gauche à droite en Java
   }
}
```

Summary

Principe

- Repérer les utilisations d'identificateurs non déclarés
- Relier chaque utilisation d'un identificateur (variable, fonction, type, etc.) à sa déclaration (pour faciliter la génération de code)
- Décorer l'arbre de syntaxe abstraite avec ces nouvelles informations.

Table des symboles

- Des informations doivent être collectées pour chaque identificateur introduit :
  - Nature de l'objet : variable, fonction, type, variable globale, locale, paramètre, etc.
  - Type : le type de la variable, la signature de la fonction, etc.
  - Plus tard, pour la génération de code, l'allocation en mémoire...
- Ces informations servent à chaque utilisation de l'identificateur
- Pour chaque utilisation d'un identificateur, il faut un lien direct vers ses informations.

Arbres de syntaxe annotés

Plusieurs représentations possibles pour lier les utilisations et les déclarations :

- Les informations sont associées à la déclaration. Chaque utilisation a un pointeur vers la déclaration.
- On crée une table annexe, associant le nom de l'identificateur à toutes ses informations.

L'analyse de portée va permettre de décorer l'AST avec ces informations.

Vérifier la portée

Entrée : l'arbre de syntaxe abstraite généré par l'analyseur syntaxique

Sortie : Un arbre de syntaxe abstraite et une table qui stocke les informations sur les variables.

Environnement

Un **environnement** est une fonction qui associe à un identificateur une définition. La définition contient les informations nécessaires à la bonne utilisation de l'identificateur :

- Nature de l'objet
- Type

Exemple

```
public class A {
   Integer t;
   public Integer f(Integer x) {
      ...
   }
}
```

L'environnement dans le corps de la fonction *f* contient :

$$
\begin{aligned}
\mathit{Integer} &\rightarrow (\underline{\mathit{class}}, \mathit{Integer}) \\
A &\rightarrow (\underline{\mathit{class}}, A) \\
t &\rightarrow (\underline{\mathit{attr}}, \mathit{Integer}) \\
f &\rightarrow (\underline{\mathit{func}}, \mathit{Integer} \times \mathit{Integer}) \\
x &\rightarrow (\underline{\mathit{arg}}, \mathit{Integer}) \\
&\vdots
\end{aligned}
$$

Surcharge

Certains langages de programmation autorisent la surcharge : un nom peut correspondre à plusieurs objets différents du programme.

Dans ce cas, l'environnement associe à un identificateur un ensemble de définitions

Exemple : fonction "+" qui définit la somme de deux entiers, et une autre fonction "+" qui définit la somme de deux flottants.

Structure de blocs

Dans les langages de programmation usuels, on a une structure de blocs, qui peuvent être imbriqués à différents niveaux.
Exemple :

```
public class A {
  public void f() {
    int x;
    x = 0;
    {
      int y;
      x++;
    }
    y++;
  }
}
```

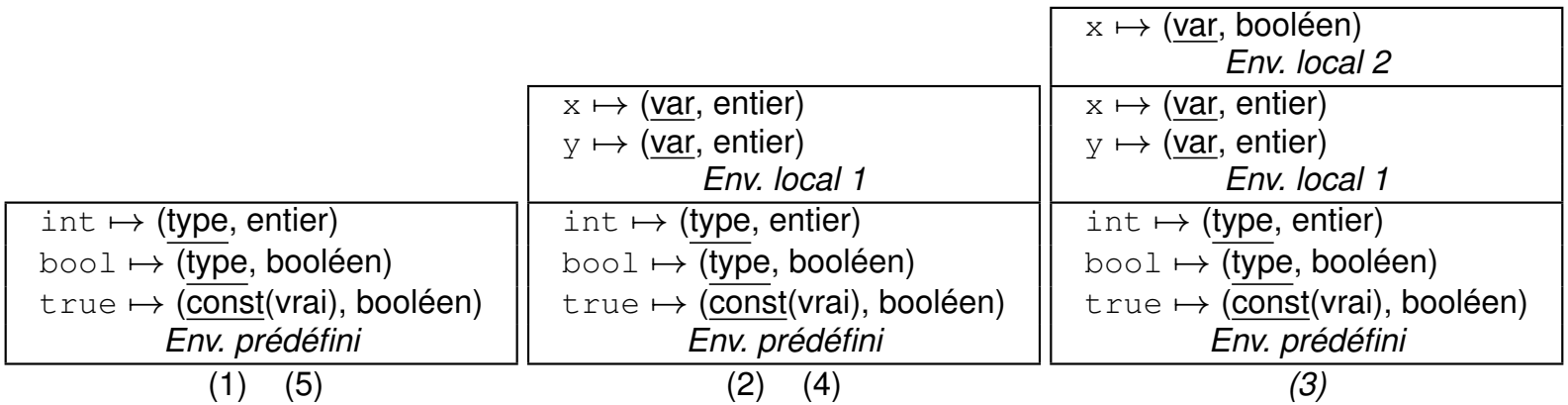
Structure de blocs

Pour gérer les blocs, on peut utiliser une pile d'environnements :

- Au départ, la pile d'environnement contient l'environnement des identificateurs prédéfinis (Integer, true, ...).
- Quand on rentre dans un bloc, on crée un nouvel environnement vide que l'on empile sur la pile d'environnement.
- Quand on voit une déclaration, on ajoute la définition dans l'environnement en tête.
- Quand on sort d'un bloc, on dépile l'environnement en tête de pile.
- Pour trouver les informations associés à un identificateur, on parcours les définitions de l'environnement en tête de pile :
  - Si l'identificateur existe, c'est OK.
  - Sinon, on recherche récursivement la définition dans l'environnement suivant dans la pile.

Exemple

```
// (1)
{ // (2)
   int y;
   int x = 1;
   { // (3)
      bool x = true;
      y = 1;
   } // (4)
   x = x+y;
}
// (5)
```



Classe Environnement

```
public class Environnement {
  // Empile un bloc
  void Entrer_bloc() {...}

  // Dépile un bloc
  void Sortir_bloc() {...}

  // Ajoute une définition dans le bloc de tête
  Boolean Enrichir_bloc(String idf, Definition def) {...}

  // renvoie true ssi l'identificateur est défini
  Boolean Est_defini(String idf) {...}

  // renvoie la définition associée à l'identificateur
  Definition Cherche_definition(String idf) {...}
}
```

Règles de portée

Selon la spécification du langage, les règles de portée sont différentes :

Exemple :

- En Java, il est incorrect de définir dans un sous bloc un identificateur déjà déclaré dans un bloc parent.
- En C, c'est autorisé

Exemple

Prog	$\rightarrow$	Bloc
Bloc	$\rightarrow$	`declare` Liste_Decl `begin` Liste_Inst `end` `';'`
Liste_Decl	$\rightarrow$	ε | Decl Liste_Decl
Decl	$\rightarrow$	Liste_Idf `':'` Type `';'`
Type	$\rightarrow$	`integer` | `boolean`
Liste_Idf	$\rightarrow$	`idf` | `idf` `','` Liste_Idf
Liste_Inst	$\rightarrow$	ε | Inst Liste_Inst
Inst	$\rightarrow$	Bloc | `idf` `':='` Exp `';'`
Exp	$\rightarrow$	...

Règles de portée

On peut spécifier les règles de portée d'un langage par des règles du type :

$$\frac{env \cup Env(Liste\_Decl) \vdash Liste\_Inst}{env \vdash declare\ Liste\_Decl\ begin\ Liste\_Inst\ end;}$$

$$\frac{idf \in env \quad env \vdash expr}{env \vdash idf := expr}$$

$$\frac{env \vdash Inst \quad env \vdash Liste\_Inst}{env \vdash Inst\ Liste\_Inst}$$

$$\frac{env_1 = Env(Liste\_Decl), \quad env_2 = Env(Decl), \quad env_1 \cap env_2 = \emptyset}{Env(Decl\ Liste\_Decl) = env_1 \cup env_2}$$

$$\frac{}{Env(Liste\_Idf : Type;) = Env(Liste\_Idf)}$$

$$\frac{}{Env(idf, Liste\_Idf) = Env(Liste\_Idf) \cup \{idf\}}$$

Summary

Notion de type

Un **type** est défini par :

- un ensemble de valeurs
- un ensemble d'opérations
- une représentation pour ces valeurs, ainsi qu'une spécification précise des opérations sur cette représentation. Ces spécifications sont plus ou moins complexes suivant le type (exemple : les types flottants)

Vérification de Type

Pendant l'analyse de portée, on fait également une vérification de type.

La vérification de type consiste à :

- vérifier que les opérations sont effectuées sur les bonnes valeurs
- vérifier que les paramètres des fonctions appartiennent au bon type
- trouver une opération unique applicable, dans le cas des langages autorisant la surcharge.

Equivalence de type

Selon les langage, la définition d'équivalence de type diffère. Deux types sont équivalents lorsque les deux types sont en fait considérés comme un seul type par le compilateur.

- **Equivalence de nom** : deux types sont équivalents si ils ont le même nom. (exemple : Java, Ada)
- **Equivalence structurelle** : deux types sont équivalents si la structure des deux types sont les même. (exemple : C)

Représentation des types

Un type est défini par deux choses :

- un nom
- une structure qui décrit la forme du type

Lorsqu'on définit un type, on stocke dans l'environnement le nom et la structure de ce type.

Exemple

On définit le langage qui permet de déclarer et d'affecter des variables de type entier ou pointeurs.
Un exemple de programme écrit dans ce langage :

```
program
   x :  int;
   p :  access int;
   pp : access access int;
begin
   x := 1;
   p.all := 2;
   pp.all := p;
   pp.all.all := 3;
end
```

Exemple

La syntaxe du langage est définie par la grammaire hors-contexte suivante :

Prog	→	`program` Liste_Decl `begin` Liste_Inst `end`
Liste_Decl	→	ε | Decl Liste_Decl
Decl	→	`idf` `':'` Type `';'`
Type	→	`int` | `access` Type
Liste_Inst	→	ε | Inst Liste_Inst
Inst	→	Place `':='` Exp `';'`
Place	→	`idf` | Place `'.'` `all`
Exp	→	Exp `'+'` Terme | Terme
Terme	→	Place | `num` | `'('` Exp `')'`

Exemple : règles contextuelles

Les règles contextuelles du langage sont les suivantes :

- Un identificateur peut être déclaré au plus une fois.
- Tout identificateur utilisé dans les instructions doit être déclaré.
- Dans une affectation, les types des parties gauche et droite doivent être équivalents (équivalence structurelle).
- On peut additionner uniquement des expressions de type entier.

Exemple : Grammaire de type

Les types du langage sont engendrés pas la grammaire de types suivante :

$$\text{Exp\_Type} \quad \rightarrow \quad \text{entier} \mid \text{ref(Exp\_Type)}$$

- Dans ce langage, on considère `int` comme étant un mot réservé, et non comme un identificateur de type. Les seuls identificateurs sont donc des identificateurs de variable.
- Un environnement associe donc à tout identificateur son type :

$$\text{Environ} \quad = \quad \text{Nom} \rightarrow \text{Exp\_Type}$$

Exemple : attributs des Déclarations

On décrit la construction de l'environnement sur la partie déclaration du langage. Pour cela, on définit les attributs suivants :

Liste_Decl $\uparrow^{env}$	env : Environ
Decl $\uparrow^{env}$	env : Environ
Type $\uparrow^{type}$	$type$: Exp_Type
`idf` $\uparrow^{nom}$	nom : String

Grammaire de la partie déclaration :

Prog	$\rightarrow$	`program` Liste_Decl $\uparrow^{env}$ `begin` Liste_Inst $\downarrow_{env}$ `end`
Liste_Decl $\uparrow^{\emptyset}$	$\rightarrow$	ε
Liste_Decl $\uparrow^{e_1+e_2}$	$\rightarrow$	Decl $\uparrow^{e_1}$ Liste_Decl $\uparrow^{e_2}$
		condition : $\text{Dom}(e_1) \cap \text{Dom}(e_2) = \emptyset$
Decl $\uparrow^{\{nom \mapsto t\}}$	$\rightarrow$	`idf` $\uparrow^{nom}$ `':'` Type $\uparrow^{t}$ `';'`
Type $\uparrow^{entier}$	$\rightarrow$	`int`
Type $\uparrow^{ref(t)}$	$\rightarrow$	`access` Type $\uparrow^{t}$

Exemple : attributs des Instructions

Pour la partie instructions du langage, on définit les attributs suivants :

Liste_Inst $\downarrow_{env}$	*env* : Environ	
Inst $\downarrow_{env}$	*env* : Environ	
Place $\downarrow_{env} \uparrow^{type}$	*env* : Environ	*type* : Exp_Type
Exp $\downarrow_{env} \uparrow^{type}$	*env* : Environ	*type* : Exp_Type

Exemple : attributs des Instructions

Liste_Inst $\downarrow_{env}$	$\rightarrow$	ε
Liste_Inst $\downarrow_{env}$	$\rightarrow$	Inst $\downarrow_{env}$ Liste_Inst $\downarrow_{env}$
Inst $\downarrow_{env}$	$\rightarrow$	Place $\downarrow_{env}\uparrow^{t_1}$ `':='` Exp $\downarrow_{env}\uparrow^{t_2}$ `';'`
		condition équivalent(t_1, t_2)
Place $\downarrow_{env}\uparrow^{t}$	$\rightarrow$	`idf` $\uparrow^{nom}$
		condition $nom \in$ Dom(env)
		affectation $t := env(nom)$
Place $\downarrow_{env}\uparrow^{t_2}$	$\rightarrow$	Place $\downarrow_{env}\uparrow^{t_1}$ `'.'` `all`
		condition t_1 = ref(t_2)
Exp $\downarrow_{env}\uparrow^{entier}$	$\rightarrow$	Exp $\downarrow_{env}\uparrow^{t_1}$ `'+'` Terme $\downarrow_{env}\uparrow^{t_2}$
		condition t_1 = entier et t_2 = entier
Exp $\downarrow_{env}\uparrow^{t}$	$\rightarrow$	Terme $\downarrow_{env}\uparrow^{t}$
Terme $\downarrow_{env}\uparrow^{t}$	$\rightarrow$	Place $\downarrow_{env}\uparrow^{t}$
Terme $\downarrow_{env}\uparrow^{entier}$	$\rightarrow$	`num`
Terme $\downarrow_{env}\uparrow^{t}$	$\rightarrow$	`'('` Exp $\downarrow_{env}\uparrow^{t}$ `')'`

Remarque : Typage statique / dynamique

Certains langages ne sont pas typés statiquement :

- le typage n'est pas vérifié pendant la compilation
- le typage est vérifié pendant l'exécution

On peut alors écrire du code de la forme :

```
(if B then 1 else 42+"a")
```

Si `B = true` pendant l'exécution, pas d'erreur...

Les langages typés **statiquement** peuvent être compilés plus efficacement :

- pas besoin de faire des tests de type à l'exécution.
- facilite les optimisations.

Remarque 2 : Inférence de type / Vérification de type

Ne pas confondre **vérification** de type et **inférence** de type :

- **Vérification** de type :
  - le programmeur associe à chaque déclaration d'identificateur un type. le compilateur vérifie la correction de ces types.
  - Exemples : C, C++, Pascal, Ada, etc.

```
int addition (int x1, int x2) {
  int temp = x1 + x2;
  return temp;
}
```

- **Inférence** de type :
  - Les types sont "devinés" (i.e synthétisés, inférés) par le compilateur par une analyse des contraintes d'utilisation des variables.
  - Exemple : Caml.

```
let addition x1 x2 =
  let temp = x1 + x2 in temp
val addition : int -> int -> int <fun>
```

Summary

Analyse Contextuelle

L'analyse contextuelle décore l'arbre abstrait des informations nécessaires à la génération de code :

- Les expressions sont décorées de leur type.
- Les affectations sont décorées de leur type.
- Les identificateurs sont décorés de leur définition. Dans cette définition, on retrouve la nature de l'identificateur et son type.

Pourquoi décorer l'AST ?

Exemple :

On veut générer en langage d'assemblage MIPS le code correspondant à l'AST représentant l'expression $x * 42$. Il existe plusieurs instructions MIPS pour effectuer des multiplications :

- `mult rt, rs` : multiplie les registres `rt` et `rs` en utilisant des entiers signés.
- `multu rt, rs` : multiplie les registres `rt` et `rs` en utilisant des entiers non signés.

Il faut connaître le type de l'expression $x * 42$ pour savoir quelle opération appliquer !

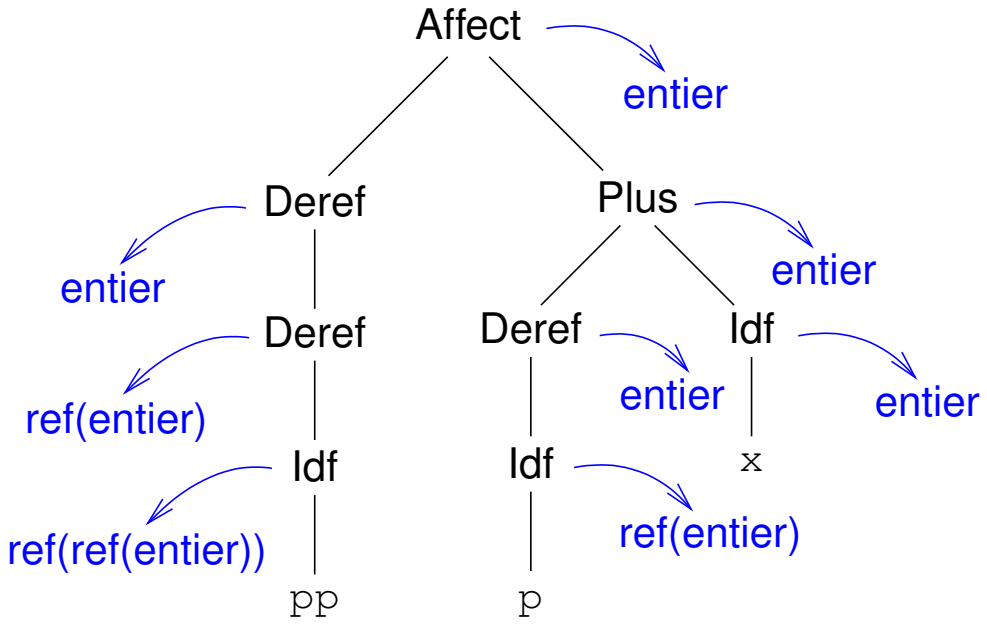
Pourquoi décorer l'AST ?

Plus généralement, les décorations de l'AST permettent :

- de distinguer un opération ou fonction qui a plusieurs définitions en fonction du type (par exemple, l'addition, la multiplications, ou des fonctions définies par l'utilisateur).
- Une affectation d'un entier ou d'un tableau, qui peut avoir des codage différents dans le langage cible.

Exemple

```
void f() {
   int     x;
   int*    p;
   int** pp;
   :
   **pp = *p + x;
}
```



Implémentation de l'analyse contextuelle

L'analyse contextuelle consiste en un parcours de l'arbre abstrait du programme. Pendant cette passe, on effectue les vérifications et on décore l'arbre abstrait.

- On parcourt l'arbre en utilisant la grammaire d'arbres (définie et propre au compilateur)
- On écrit une fonction par non terminal de la grammaire d'arbres. Chaque fonction est en charge de l'analyse contextuelle d'un sous arbre dont la racine est le non-terminal associé à cette fonction.

Exemple : JCas

La grammaire d'arbre définie dans le projet contient les règles suivantes :

```
PROGRAMME    →  Noeud.Programme(LISTE_DECL, LISTE_INST)
LISTE_DECL   →  Noeud.ListeDecl(LISTE_DECL, DECL)
             |  Noeud.Vide
DECL         →  Noeud.Decl(LISTE_IDENT, TYPE)
IDENT        →  Noeud.Ident
⋮
```

La passe de vérification contextuelle contient donc les fonctions suivantes :

```
verifier_PROGRAMME(Arbre a) throws ErreurVerif { ... }
verifier_LISTEDECL(Arbre a) throws ErreurVerif { ... }
verifier_DECL(Arbre a) throws ErreurVerif { ... }
verifier_IDENT(Arbre a) throws ErreurVerif { ... }
⋮
```

Exemple : JCas

L'environnement est construit lorsqu'on analyse les déclarations :

```
verifier_LISTEDECL(Arbre a) throws ErreurVerif { ... }
```

L'environnement est utilisé lorsqu'on analyse les instructions :

```
verifier_LISTEINST(Arbre a) throws ErreurVerif { ... }
```