Grenoble INP ESISAR

4A IR - CS410

TP Languages et Compilation
2013 / 2014

Jcas Compiler

Passe 2

Vérifications contextuelles

- Bassem Debbabi (TP1)
- Mouna Tka (TP2)

Plan

- **Contraintes contextuelles**
  - Les types du langage JCas
  - Règles de visibilité
  - Profils d'opérateurs
  - Vérifications de type
- **Enrichissement et décoration de l'arbre abstrait**
  - Ajouts de Noeud.Conversion
  - Décoration de l'arbre abstrait
- **Mise en oeuvre de la passe 2**
  - Types fournis
  - Implémentation de la passe 2
- **Tests**

But de la passe 2

- Vérifier qu'un programme JCas est contextuellement correct ;
- enrichir et décorer l'arbre abstrait pour préparer la passe 3.

1. Contraintes contextuelles

Les contraintes contextuelles du langage JCas sont définies dans `Context.txt` (page 6).

Contraintes contextuelles
– Les types du langage JCas

- Les types du langage JCas sont les suivants: *intervalle d'entiers, réel, booléen, string* et *tableau.*
- **Intervalle d'entier**
  - Exemple : `1..10` représente l'intervalle des entiers de 1 à 10, noté *Type.Interval(1,10)*.
  - `max_int` est une constante qui représente l'entier maximal du langage Jcas, de valeur valmax, où

    valmax = java.lang.Integer.MAX_VALUE.
  - Le type `integer` représente

    Type.Integer = Type.Interval(-valmax, valmax).

Contraintes contextuelles
– Les types du langage JCas

- **Réel**
  - Correspond à un sous-ensemble de $\mathbb{R}$, noté *Type.Real.*
- **Boolean**
  - Correspond à l'ensemble {*vrai, faux*}, noté *Type.Boolean*.
  - On a deux constantes de type booléen : `true` (valeur *vrai*) et `false` (valeur *faux*).
- **String**
  - Pas de syntaxe dans le langage JCas. On ne peut donc pas déclarer de variable de type string. On a uniquement des littéraux de type string comme dans l'instruction :

```
write("OK");
```

Contraintes contextuelles
– Les types du langage JCas

- **Tableau**
  - Syntaxe : `array[` *type_intervalle* `]` of *type*
  - Exemples :

```
array[1..10] of integer
Array[1..10] of array[1..5] of boolean
```

  - Notation : *Type.Array(...)*

Contraintes contextuelles
– Les types du langage JCas

- *Grammaire de types du langage JCas*

EXP_TYPE	→	INTERVALLE
	|	Type.Real
	|	Type.Boolean
	|	Type.String
	|	Type.Array(INTERVALLE, EXP_TYPE)
INTERVALLE	→	Type.Interval(entier, entier)

Contraintes contextuelles
– Les types du langage JCas

- *Equivalence de types*
  - Équivalence structurelle (≠ équivalence de nom).
  - Exemple :

```
v1 : array[1..10] of integer ;
m  : array[1..5] of array[1..10] of integer ;
v2 : array[1..10] of integer ;
```

`m[1]`, `m[2]`, … `v1` et `v2` sont de même type.

```
v1 := v2 ;    -- ok
m[1] := v1 ; -- ok
m := v1 ;     -- interdit
```

Contraintes contextuelles
– Règle de visibilité

- Les règle de visibilités du langage JCas sont les suivants :
  - On ne peut pas re-déclarer un identificateur déjà déclaré.
  - Tout identificateur apparaissant dans un programme JCas doit être déclaré, sauf les identificateurs prédéfinis.
  - Les identificateurs prédéfinis ne peuvent pas être redéfinis.
- Les identificateurs d'un programme JCas sont de différentes natures :
  - Identificateurs de constantes (de type intervalle, booléen, réel ou chaîne) ;
  - identificateurs de type ;
  - identificateurs de variable.

Nature = {const, type, var}.

Contraintes contextuelles
– Règle de visibilité

- Seuls des identificateurs de variables peuvent être déclarés dans un programme JCas. Les seuls identificateurs de constante et de type sont donc des identificateurs prédéfinis.
- **La nature des identificateurs doit être vérifiée.**
- L'environnement associe à chaque identificateur une définition.
  - Une définition est un couple (Nature, Type).
- Au début de l'analyse du programme, l'environnement contient uniquement les identificateurs prédéfinis.

Contraintes contextuelles
– Règle de visibilité

- Environnement prédéfini :

`"boolean"`	→	(type, *Type*.*Boolean*)
`"false"`	→	(const(*faux*), *Type.Boolean*)
`"true"`	→	(const(*vrai*), *Type.Boolean*)
`"integer"`	→	(type, *Type.Integer*)
`"max_int"`	→	(const(*valmax*), *Type.Integer*)
`"real"`	→	(type, *Type.Real*)

Contraintes contextuelles
– Profils d'opérateurs

Type.Integer : *type Type.Interval(-valmax, valmax)*

Type.Interval : un type intervalle quelconque *Type.Interval(a,b)*.

not :	Type.Boolean → Type.Boolean
and, or :	Type.Boolean, Type.Boolean → Type.Boolean
=, <, >, /=, <=, >= :	Type.Interval, Type.Interval → Type.Boolean
	Type.Interval, Type.Real → Type.Boolean
	Type.Real, Type.Interval → Type.Boolean
	Type.Real, Type.Real → Type.Boolean

Contraintes contextuelles

– Profils d'opérateurs

`+`, `-` :	Type.Interval → Type.Integer
	Type.Real → Type.Real
`+`, `-`, `*` :	Type.Interval, Type.Interval → Type.Integer
	Type.Interval, Type.Real → Type.Real
	Type.Real, Type.Interval → Type.Real
	Type.Real, Type.Real → Type.Real
`div`, `mod` :	Type.Interval, Type.Interval → Type.Integer
/ :	Type.Interval, Type.Interval → Type.Real
	Type.Interval, Type.Real → Type.Real
	Type.Real, Type.Interval → Type.Real
	Type.Real, Type.Real → Type.Real
`[]` (indexation)	Type.Array(Type.Interval, *type*), Type.Interval → *type*

Contraintes contextuelles
– Vérification de type

- Intervalles *exp_const1 .. exp_const2*
  - *exp_const1* et *exp_const2* doivent être de type *Type.Interval*.
- Affectations *place := expression*
  - Le type de *place* et le type de *expression* doivent être compatibles pour l'affectation, c'est-à-dire :
    - *place* et *expression* de type *Type.Interval* (pas forcément avec les mêmes bornes) ;
    - *place* et *expression* de type *Type.Real* ;
    - *place* et *expression* de type *Type.Boolean* ;
    - *place* de type real et *expression* de type *Type.Interval* ;
    - *place* et *expression* de type *Type.Array*, les types des indices étant identiques (de type *Type.Interval*, avec les mêmes bornes), et les types des élements compatibles pour l'affectation.

Contraintes contextuelles
– Vérification de type

- Instructions `if` et `while` : la condition doit être de type *Type.Boolean*.
- Instruction `for` : la variable de contrôle, ainsi que les deux expressions doivent être de type *Type.Interval*.
- Instruction `read` : la place doit être de type *Type.Interval* ou *Type.Real*.
- Instruction `write` : Les expressions doivent être de type *Type.Interval*, *Type.Real* ou *Type.String*.
- Les places et expressions doivent être bien typées vis-à-vis des déclarations et des profils des opérateurs.

2. Enrichissement et décoration de l'arbre abstrait

L'enrichissement et la décoration de l'arbre abstrait a pour but de préparer la passe 3 (génération de code).
ct. `ArbreEnrichi.Doc` (page 12).

Enrichissement et décoration de l'arbre abstrait – Ajouts de Noeud.Conversion

- Le langage JCas autorise l'ajout d'un entier et d'un réel, ou l'affectation d'un entier à un réel.
- On ne peut pas réaliser cela directement en assembleur : il faut commencer par convertir l'entier en réel.
- Un Noeud.Conversion représente la conversion d'un entier vers un réel (et non l'inverse !)

  Exemple

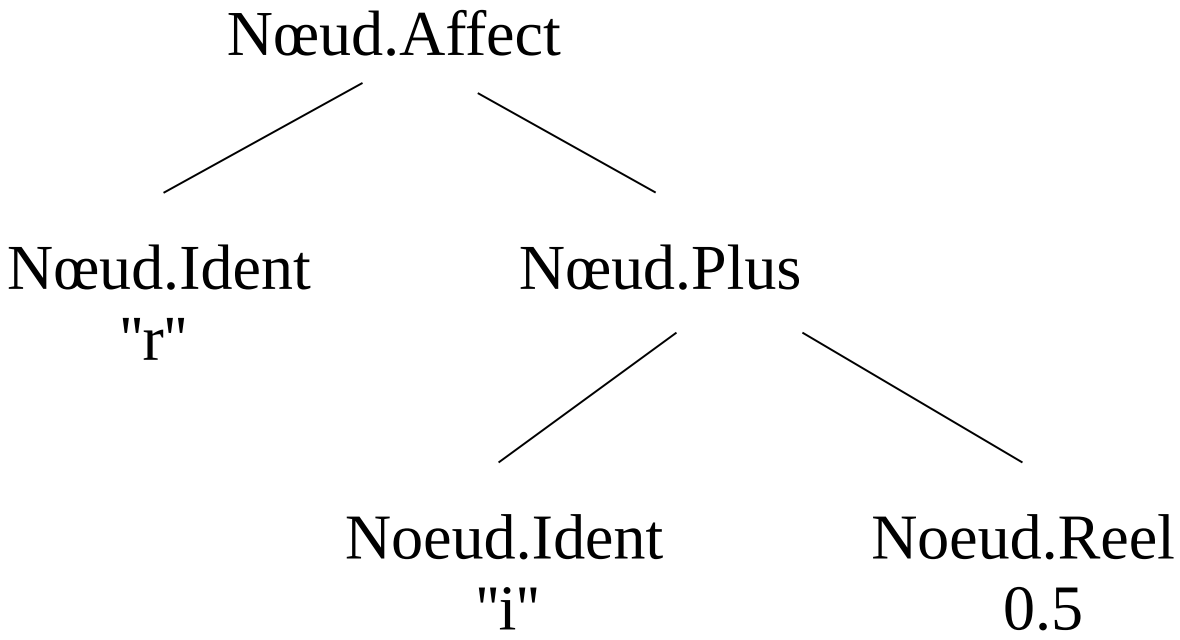
```
r : real ;
i : integer ;

r := i + 0.5 ;
r := i + 1 ;
```

  Pour ajouter les *Noeud.Conversion*, on utilise les procédure `setFils1` et `setFils2` de la classe `Arbre`.

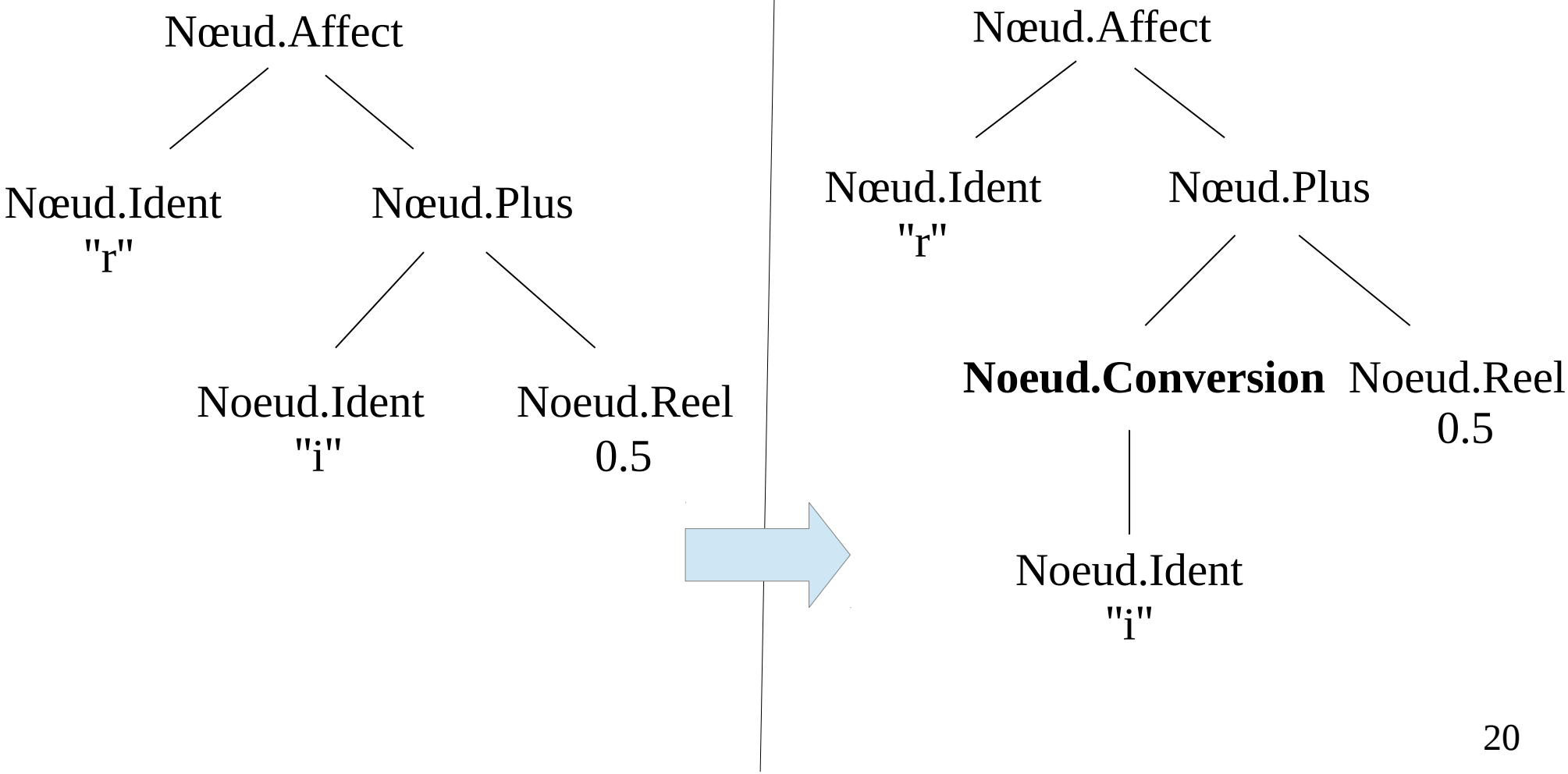
Enrichissement et décoration de l'arbre abstrait – Ajouts de Noeud.Conversion

```
r := i + 0.5 ;
```



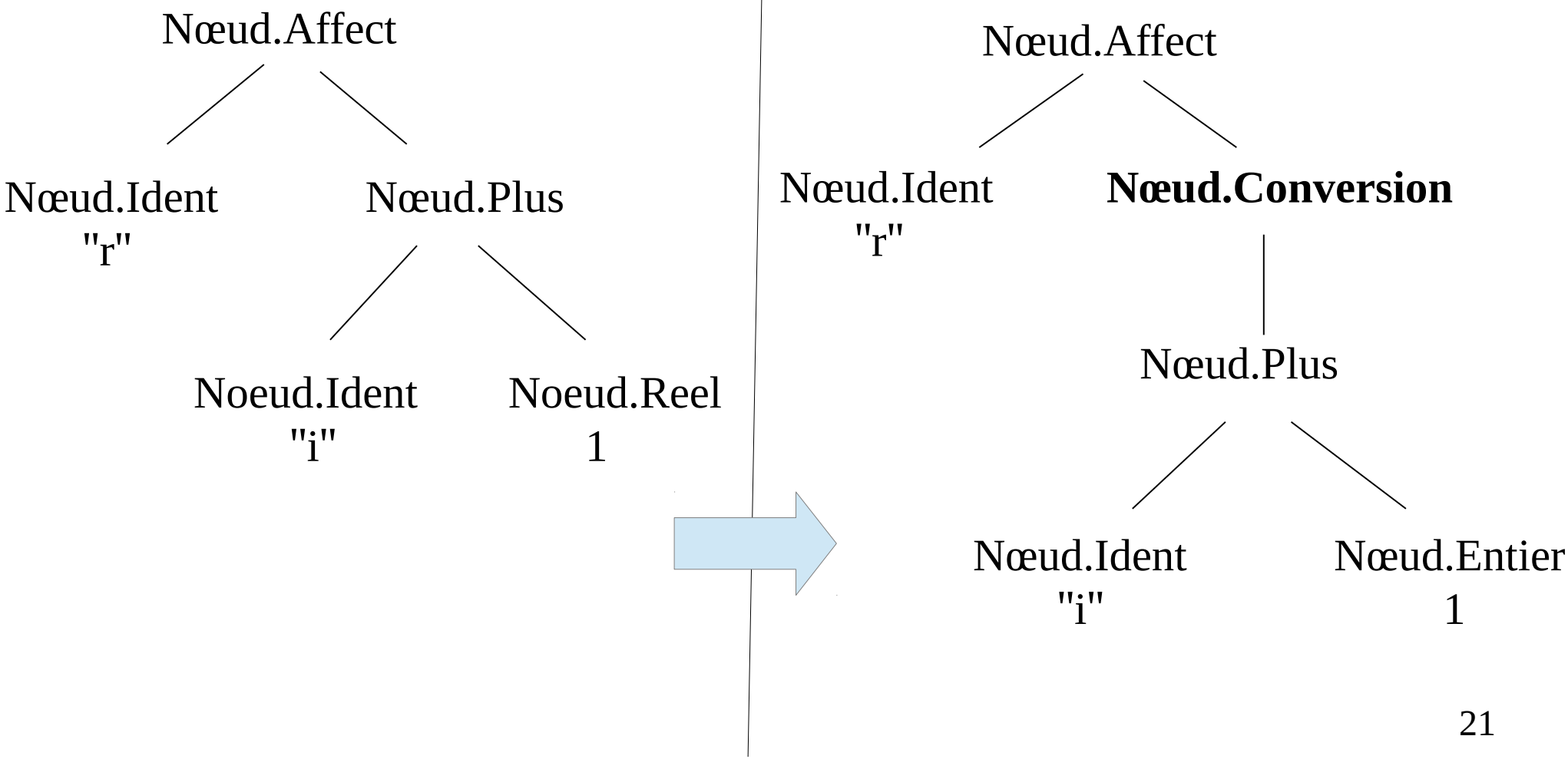
Enrichissement et décoration de l'arbre abstrait – Ajouts de Noeud.Conversion

```
r := i + 0.5 ;
```



Enrichissement et décoration de l'arbre abstrait – Ajouts de Noeud.Conversion

```
r := i + 1 ;
```

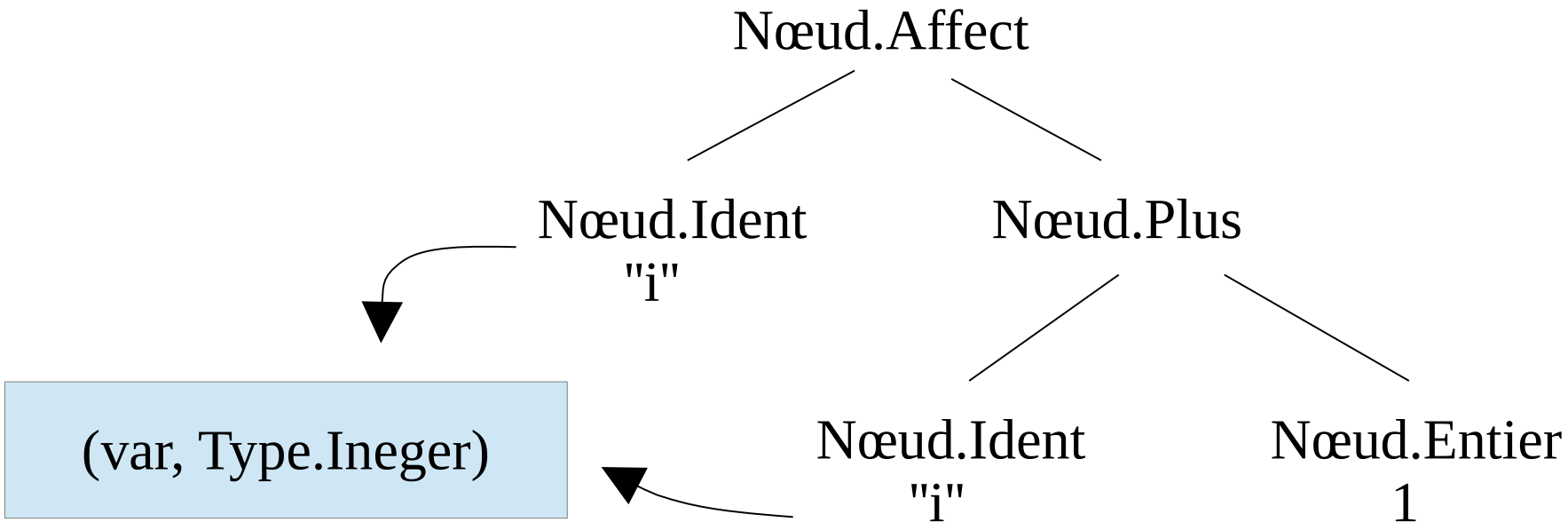


Enrichissement et décoration de l'arbre abstrait – Décoration de l'arbre abstrait

- A chaque noeud de l'arbre est associé un décor. Un décor est un triplet : (Defn defn, Type type, int infoCode)
  - `defn` est associé aux *Noeud.Ident*.
  - `type` est associé aux *Noeud.Affect*, *Noeud.Conversion* et à tous les noeuds qui dérivent de EXP dans la grammaire d'arbres.
  - `infoCode` peut servir en passe 3 pour calculer le nombre de registres nécessaires pour évaluer une expression.
- Sémantique de partage :
  - Les *defns* et *types* sont partagés.

Enrichissement et décoration de l'arbre abstrait – Décoration de l'arbre abstrait

- Par exemple, tous les *Noeud.Ident* **correspondant au même identificateur** sont décorés avec la même *defn*.



3. Mise en oeuvre de la passe 2

Mise en oeuvre de la passe 2 – Types fournis

- `NatureType` et `Type` : type énuméré et classe permettant de manipuler des types du langage JCas ;
- `NatureDefn` et `Defn` : type énuméré et classe permettant de manipuler des définitions.

  Constantes :

  - `NatureDefn.ConstInteger`, `NatureDefn.ConstBoolean`, `NatureDefn.Var`, `NatureDefn.Type`,
- Remarque : une `Defn` est un triplet
  - (String, NatureDefn, Type).
- `Environ` : Permet d'associer des Defn à des identificateurs.
- `ErreurType`, `ErreurDefn` : Exception levée en cas d'erreur sur un type ou une Defn.
- `Decor` : class permettant de manipuler des décors.

Mise en oeuvre de la passe 2

– Implémentation de la passe 2

- La passe 2 est un parcours de l'arbre abstrait du programme. Lors de ce parcours :
  - on vérifie que le programme JCas est contextuellement correct ;
  - on ajoute des Noeud.Conversion ;
  - on décore les différents noeuds de l'arbre.
- Pour implémenter ce parcours d'arbre, on suit exactement la grammaire d'arbre. On écrit (au minimum) une méthode par non-terminal de la gramme d'arbres.

Mise en oeuvre de la passe 2

– Implémentation de la passe 2

```
void verifier_PROGRAMME(Arbre a) throws ErreurVerif;
void verifier_LISTE_INST(Arbre a) throws ErreurVerif;
void verifier_LISTE_DECL(Arbre a) throws ErreurVerif;
...
```

- Pour les identificateurs, il faut distinguer les déclarations et les utilisations d'identificateurs.

```
void verifier_IDENT_Decl(Arbre a, ...) throws ErreurVerif;
void verifier_IDENT_Util(Arbre a; ...) throws ErreurVerif;
```

- On peut également définir d'autres méthodes pour les différents noeuds de l'arbre.

Mise en oeuvre de la passe 2
– Implémentation de la passe 2

- Exemple :

```
void verifier_LISTE_INST(Arbre a) throws ErreurVerif {
   switch (a.getNoeud()) {
      case Vide:
         break;
      case Liste_Inst:
         verifier_LISTE_INST(a.getFils1());
         verifier_INST(a.getFils2());
         break;
      default:
         throw new ErreurInterneVerif(
            "Arbre incorrect dans verifier_LISTE_INST")  ;
   }
}
```

Mise en oeuvre de la passe 2

– Implémentation de la passe 2

```
void verifier_INST(Arbre a) throws ErreurVerif {
   switch (a.getNoeud()) {
      case Nop:
         break;
      case Affect:
         verifier_Affect(a);
         break;
      case Pour:
         verifier_Pour(a);
         break;
      ... // Traiter tous les noeuds possibles
      default:
         throw new ErreurInterneVerif(
            "Arbre incorrect dans verifier_INST")  ;
   }
}
```

Mise en oeuvre de la passe 2
– Implémentation de la passe 2

- *Conseils*

  Pour cette étape, il est important de :

  - bien décomposer les problèmes en écrivant des méthodes **courtes** ;
  - factoriser les éléments communs (**pas de copié-collé !**) ;
  - compiler et tester au fur et à mesure ;
  - conserver et documenter tous les fichiers de test :
    - tests de non régression ; scripts permettant d'enchaîner les tests ;
    - commentaires indiquant le résultat du test (passe, erreur contextuelle ligne *n*)

- *Exercice*

  - Ecrire la méthode qui construit l'environnement prédéfini (dans `Verif.java`).

Mise en oeuvre de la passe 2
– Implémentation de la passe 2

- *A faire*
  - `ErreurContext` :
    - Type énuméré qui définit une constante par type d'erreur contextuelle, ainsi qu'une procédure qui affiche un message d'erreur pour chaque erreur contextuelle.
  - `ReglesTypage` :
    - classe définissant des prédicats indiquant si deux types sont compatibles (pour une affectation, pour un opérateur binaire, pour un opérateur unaire).

      Les classes `ResultatAffectCompatible`,

      `ResultatUnaireCompatible` et `ResultatBinaireCompatible` servent de type pour les méthodes de `ReglesTypage`.
  - `Verif` : classe principale de la vérification contextuelle.

Mise en oeuvre de la passe 2 – Tests

- On distingue les types de tests suivants :
  - Tests unitaires (test d'une méthode, d'une classe)

    Exemple : écrire une classe `TestReglesTypage` qui permet de tester les méthodes de la classe `ReglesTypage`.
  - Test d'intégration (teste l'intégration de plusieurs méthodes ou plusieurs classes).
  - **Test système** : test du compilateur dans les conditions normales d'utilisation

    => Ecrire des programmes JCas de test

    Ecrire des programmes JCas valides (contextuellement corrects) et invalides (contextuellement incorrects). Pour les programmes valides, vérifier que l'arbre est correctement décoré, Pour les programmes JCas incorrects, vérifier que le message d'erreur est pertinent.

Mise en oeuvre de la passe 2

- **A Rendre**
  - Programmes (dans `Verif/Src`)
  - Jeux de tests (dans `Verif/Test`)
  - Documentation (dans `Verif/Doc`) décrivant :
    - les messages d'erreurs,
    - l'architecture de la passe 2,
    - la méthodologie de test.