Projet Génie Logiciel

Étape B

Analyse contextuelle

Projet GL

Ensimag
Grenoble INP

3 décembre 2023

Grammaires attribuées

- Les grammaires attribuées
  - permettent de définir une classe de langages plus grande que les grammaires hors-contexte,
  - décrivent des calculs dirigés par la syntaxe.
  - permettent d'associer une interprétation à la syntaxe (sémantique dénotationnelle).
- On associe à chaque terminal et non terminal d'une grammaire hors-contexte des *attributs* (sortes de paramètres)
- Attributs *typés* (domaine de valeurs)
- Types : entier, réel, chaîne de caractères, ensemble, fonction... etc.

Attributs

- On distingue :
  - les attributs hérités X↓*att*
    Valeur dépendant du contexte dans lequel X est dérivé.
    Transmis du père vers le fils dans l'arbre de dérivation.
    cf. paramètres « in » en Ada.
  - les attributs synthétisés X↑*att*
    Valeur dépendant des règles appliquées pour dériver X
    Transmis du fils vers le père dans l'arbre de dérivation.
    cf. paramètres « out » en Ada, ou valeur(s) de retour d'une fonction.
- Description du calcul des attributs : pour chaque règle de la forme
  $$X \to Y_1 Y_2 \dots Y_n$$
  on définit :
  - les attributs hérités de Y_i en fonction des attributs de X, Y_1, ... Y_n ;
  - les attributs synthétisés de X en fonction des attributs de X, Y_1, ... Y_n.

Exemple : langage $L = \{a^n b^n c^n \ ; \ n \in \mathbb{N}\}$

- On considère $L_1 = \{a^n b^n c^p \ ; \ n, p \in \mathbb{N}\}$
- Grammaire qui engendre L_1 :
  $$\begin{array}{lcl} S & \to & A\ C \\ A & \to & \varepsilon \mid a\ A\ b \\ C & \to & \varepsilon \mid c\ C \end{array}$$
- Variante, avec style EBNF :
  $$\begin{array}{lcl} S & \to & A\ C \\ A & \to & \varepsilon \mid a\ A\ b \\ C & \to & c* \end{array}$$

Exemple : langage $L = \{a^n b^n c^n \ ; \ n \in \mathbb{N}\}$

- Définition de L à l'aide d'une grammaire attribuée
- Attributs synthétisés :
  $A{\uparrow}n$, $n : \mathbb{N}$ (nombre de 'a' de la chaîne)
  $C{\uparrow}n$, $n : \mathbb{N}$ (nombre de 'c' de la chaîne)
- Grammaire attribuée qui engendre L :
  $$\begin{array}{lcl} S & \to & A{\uparrow}n\ \ C{\uparrow}p \\ & & \text{condition } n = p \\ A{\uparrow}0 & \to & \varepsilon \\ A{\uparrow}n+1 & \to & a\ A{\uparrow}n\ b \\ C{\uparrow}0 & \to & \varepsilon \\ C{\uparrow}n+1 & \to & c\ C{\uparrow}n \end{array}$$

Exemple : langage $L = \{a^n b^n c^n \ ; \ n \in \mathbb{N}\}$

- Autre grammaire attribuée pour L
- Attributs synthétisé et hérité :
  $A{\uparrow}n$, $n : \mathbb{N}$ (nombre de 'a' de la chaîne)
  $C{\downarrow}n$, $n : \mathbb{Z}$ (nombre de 'c' que le contexte impose pour C)
- Grammaire attribuée qui engendre L :
  $$\begin{array}{lcl} S & \to & A{\uparrow}n\ \ C{\downarrow}n \\ A{\uparrow}0 & \to & \varepsilon \\ A{\uparrow}n+1 & \to & a\ A{\uparrow}n\ b \\ C{\downarrow}n & \to & \varepsilon \\ & & \text{condition } n = 0 \\ C{\downarrow}n & \to & c\ C{\downarrow}n-1 \end{array}$$

Exemple : langage $L = \{a^n b^n c^n \ ; \ n \in \mathbb{N}\}$

- Encore une autre grammaire attribuée pour L (avec petites extensions de syntaxe).
- Attributs synthétisé et hérité :
  $A{\downarrow}n$, $n : \mathbb{Z}$ (nombre de 'a' que le contexte impose pour A)
  $C{\uparrow}n$, $n : \mathbb{N}$ (nombre de 'c' de la chaîne C)
- Grammaire attribuée qui engendre L :
  $$\begin{array}{lcl} S & \to & A{\downarrow}n\ \ C{\uparrow}n \\ A{\downarrow}0 & \to & \varepsilon \\ A{\downarrow}n & \to & a\ A{\downarrow}n-1\ b \\ C{\uparrow}n & \to & \{\ n := 0\ \}\ (\mathrm{c}\ \{\ n := n+1\ \})* \end{array}$$

Propriétés contextuelles d'un programme

- Propriétés *contextuelles* d'un programme :
  ne peuvent pas être décrites par une grammaire hors-contexte
  - déclaration et utilisation des identificateurs ;
  - typage des expressions.
- Propriétés contextuelles : décrites par des règles contextuelles.
- Nécessaire pour définir la *sémantique statique* du langage (cf. slide suivant).
- Règle contextuelle non respectée ⇒ message d'erreur contextuelle

Sémantique Statique $\triangleq$ Syntaxe Contextuelle

- Syntaxe Contextuelle $\triangleq$ ensemble des programmes pour lesquels le compilateur doit produire un exécutable (sauf limitations).

  NB : la sémantique (dynamique) dépend souvent du *typage statique*. Exemple Java : la sémantique de "o.equals(x)" dépend du *type dynamique* de "o" **et** du *type statique* de "x".

- Nécessite une définition rigoureuse pour "compatibilité" des compilateurs. **Vous ne pouvez pas l'inventer !**

  Exemple : ce programme est-il accepté par les compilateurs Java ?

```
class A {
  void f(){
    A A = new A();
    A.f();
  }
}
```

  Oui

Exemple

- Langage à structure de bloc

```
                          -- niveau 1
declare                   -- niveau 2
   x, y : integer;
begin
   x := 1;
   declare                -- niveau 3
      x : boolean;
   begin
      x := true;
      y := 1;
   end;                   -- niveau 2
   x := x + y;
end;                      -- niveau 1
```


Règles contextuelles du langage

- Les identificateurs `integer` et `boolean` sont des identificateurs de type prédéfinis.
- Les identificateurs `true` et `false` sont des identificateurs de constantes booléennes prédéfinis.
- Un identificateur ne peut pas être déclaré plus d'une fois au même niveau.
- Tout identificateur utilisé dans les instructions doit :
  - être préalablement déclaré (soit dans le même bloc, soit dans un bloc englobant)
  - être utilisé conformément à sa déclaration.

Type, nature, définition et environnement

- Types du langage : entier ou booleen
  Type $\triangleq$ { entier, booleen }
- Nature des identificateurs : type, variable ou constante
  Nature $\triangleq$ { var, type, const }
- Définition d'un identificateur : nature et type
  Définition $\triangleq$ Nature $\times$ Type
- Symbol : domaine des identificateurs
- Environnement : associe à un identificateur sa définition
  Environnement $\triangleq$ Symbol $\rightarrow$ Définition (fonction partielle)
- $env(x)$: définition associée à $x \in$ Symbol dans env.
- $dom(env)$: domaine de l'environnement env (ensemble des identificateurs auquels est associée une définition)

Environnements

- Environnement prédéfini :

$$\begin{array}{rlll} \text{Predef} \triangleq \{ & \text{integer} & \mapsto & (\underline{\text{type}}, \underline{\text{entier}}), \\ & \text{boolean} & \mapsto & (\underline{\text{type}}, \underline{\text{booleen}}), \\ & \text{true} & \mapsto & (\underline{\text{const}}, \underline{\text{booleen}}), \\ & \text{false} & \mapsto & (\underline{\text{const}}, \underline{\text{booleen}}) \quad \} \end{array}$$

- Traitement de la structure de bloc : *empilement* d'environnements

env_3	niveau 3
env_2	niveau 2
Predef	niveau 1

  - $env_2 = \{ \text{x} \mapsto (\underline{\text{var}}, \underline{\text{entier}}), \text{y} \mapsto (\underline{\text{var}}, \underline{\text{entier}}) \}$
  - $env_3 = \{ \text{x} \mapsto (\underline{\text{var}}, \underline{\text{booleen}}) \}$

Opérations sur les environnements

Soit deux environnements env_1 et env_2.

- Union disjointe de deux environnements
  - $env_1 \oplus env_2$ n'est pas défini, si $dom(env_1) \cap dom(env_2) \neq \varnothing$
  - $(env_1 \oplus env_2)(x) \triangleq \begin{cases} env_1(x) \text{ si } x \in dom(env_1) \\ env_2(x) \text{ si } x \in dom(env_2) \end{cases}$

  Permet de traiter les déclarations d'identificateurs qui sont au même niveau.

- Empilement de deux environnements
  - $(env_1/env_2)(x) \triangleq \begin{cases} env_1(x) \text{ si } x \in dom(env_1) \\ env_2(x) \text{ si } x \notin dom(env_1) \text{ et } x \in dom(env_2) \\ \text{indéfini sinon} \end{cases}$

Syntaxe abstraite du mini-langage à blocs

PROG	$\rightarrow$	BLOC
BLOC	$\rightarrow$	`Bloc`[LIST_DECL LIST_INST]
LIST_DECL	$\rightarrow$	[DECL*]
DECL	$\rightarrow$	`Decl`[IDF TYPE]
TYPE	$\rightarrow$	IDF
LIST_INST	$\rightarrow$	[INST*]
INST	$\rightarrow$	BLOC | `Assign`[IDF EXP]
EXP	$\rightarrow$	IDF | `Num` | `Plus`[EXP EXP]
IDF	$\rightarrow$	`Idf`

Profils des symboles de la syntaxe contextuelle

- BLOC $\downarrow env$ $\quad env$: Environnement (environnement englobant)
- LIST_DECL $\downarrow env\_glob \uparrow env$
  env_glob : Environnement (environnement englobant)
  env : Environnement (environnement des déclarations)
- DECL $\downarrow env\_glob \uparrow env$
- IDF $\downarrow t \uparrow env$ $\quad t$: Type
- TYPE $\downarrow env\_glob \uparrow t$ $\quad t$: Type
- LIST_INST $\downarrow env$
- INST $\downarrow env$
- EXP $\downarrow env \uparrow t$ $\quad t$: TYPE
- `Idf` $\uparrow nom$ $\quad nom$: Symbol

Règles de la syntaxe contextuelle

PROG → BLOC $\downarrow$*Predef*
BLOC $\downarrow$*env_glob* → Bloc [LIST_DECL $\downarrow$*env_glob* $\uparrow$*env*
LIST_INST $\downarrow$*env*/*env_glob*]

LIST_DECL $\downarrow$*env_glob* $\uparrow$*env* →
{ *env* := ∅ }[(DECL $\downarrow$*env_glob* $\uparrow$*env*$_1$ { *env* := *env* ⊕ *env*$_1$ })*]
condition implicite : *dom*(*env*) initial et *dom*(*env*$_1$) disjoints

DECL $\downarrow$*env_glob* $\uparrow$*env* → Decl[IDF $\downarrow t$ $\uparrow$*env* TYPE $\downarrow$*env_glob* $\uparrow t$]

IDF $\downarrow t$ $\uparrow\{nom \mapsto (\text{var}, t)\}$ → Idf $\uparrow$*nom*

TYPE $\downarrow$*env_glob* $\uparrow t$ → Idf $\uparrow$*nom*
condition : $(\text{type}, t) \triangleq$ *env_glob*(*nom*)
condition implicite additionnelle : *nom* ∈ *dom*(*env_glob*)

Règles contextuelles (suite)

LIST_INST $\downarrow$*env* → [(INST $\downarrow$*env*)*]

INST $\downarrow$*env* → BLOC $\downarrow$*env*
→ Assign[Idf $\uparrow$*nom* EXP $\downarrow$*env* $\uparrow t$]
condition : *env*(*nom*) = (var, *t*)

EXP $\downarrow$*env* $\uparrow t$ → Idf $\uparrow$*nom*
condition : (*nat*, *t*) ≜ *env*(*nom*) et *nat* ∈ {var, const}
EXP $\downarrow$*env* $\uparrow$entier → Num
→ Plus[EXP $\downarrow$*env* $\uparrow t_1$ EXP $\downarrow$*env* $\uparrow t_2$]
condition : $t_1 = t_2 =$ entier

Les programmes suivants sont-ils acceptés ?

1.
```
begin
   false := true;
end;
```

2.
```
declare
   integer : integer;
begin
   integer := integer + 1;
end;
```

3.
```
declare
   integer : integer;
begin
   declare
      x : integer;
   begin
      x := 1;
   end;
end;
```


Les programmes suivants sont-ils acceptés ?

1.
```
begin
   false := true; -- erreur contextuelle :
end;              -- false identificateur de constante
```

2.
```
declare
   integer : integer; -- ok
begin
   integer := integer + 1; -- ok
end;
```

3.
```
declare
   integer : integer; -- ok
begin
   declare
      x : integer; -- erreur contextuelle : integer
   begin           -- identificateur de variable
      x := 1;
   end;
end;
```


Introduction

- cf. II-[SyntaxeContextuelle]
- Vérification contextuelle de Deca : nécessite trois passes sur le programme.
- Exemple

```
class A {
   B b;
}

class B {
   A a;
}
```

```
class Parcours {
   void parcoursA(A a) {
      if (a != null) {
         parcoursB(a.b);
      }
   }
   void parcoursB(B b) {
      if (b != null) {
         parcoursA(b.a);
      }
   }
}
```


Trois passes sur le programme

- Déclaration de champ ou méthode : référence à une classe qui apparait après. Ex : « `B b` » dans la classe « `A` »
  - Passe 1 : on vérifie le nom des classes et la hiérarchie de classes
- Remarque : Deca, contrairement à Java, impose que les super-classes soient déclarées avant les sous-classes. Par exemple

```
class D extends C { }
class C { }
```

  est un programme Deca incorrect.
- Méthodes mutuellement récursives (`parcoursA` et `parcoursB`)
  - Passe 2 : on vérifie les déclarations de champs et les signatures des méthodes.
  - Passe 3 : on vérifie le corps des méthodes, les expressions d'initialisation des champs, et le programme principal.

Exemple

- Les trois passes sur l'exemple :

```
class A {
   B b;
}

class B {
   A a;
}
```

```
class Parcours {
   void parcoursA(A a) {
      if (a != null) {
         parcoursB(a.b);
      }
   }
   void parcoursB(B b) {
      if (b != null) {
         parcoursA(b.a);
      }
   }
}
```

- Passe 1 : `class A ; class B ; class Parcours ;`
- Passe 2 : `B b ; A a ; parcoursA(A a) ; parcoursB(B b) ;`
- Passe 3 : corps de `parcoursA` et `parcoursB`

Domaines d'attributs de la grammaire attribuée de Deca

- Symbol : ensemble des identificateurs Deca
- Type ≜ {void, boolean, float, int, string, null}
  ∪ type_class(Symbol)
  À chaque classe *A* du programme correspond un type type_class(*A*).
- Visibility ≜ {protected, public}
- TypeNature ≜ {type} ∪ class(Profil)
- ExpNature ≜ {param, var} ∪ method(Signature)
  ∪ field(Visibility, Symbol)
  - field(public, *A*) : champ public d'une classe *A*
  - field(protected, *A*) : champ protégé d'une classe *A*
- Signature d'une méthode : liste (ordonnée) des types de ses paramètres
  Signature ≜ Type*

Domaines d'attributs de la grammaire attribuée de Deca

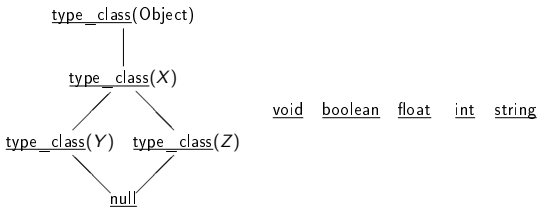
- Extension : nom de la super-classe, ou 0 pour Object
  Extension $\triangleq$ Symbol $\cup \{0\}$
- Profil d'une classe : nom de la super-classe et environnement des champs et méthodes de la classe
  Profil $\triangleq$ Extension $\times$ EnvironmentExp
- Definition d'un identificateur : sa nature et son type
  TypeDefinition $\triangleq$ TypeNature $\times$ Type
  ExpDefinition $\triangleq$ ExpNature $\times$ Type
  - type de l'objet pour un identificateur param, var ou field
  - type du résultat pour un identificateur de méthode
- Environnement : associe à un identificateur sa définition
  EnvironmentType $\triangleq$ Symbol $\rightarrow$ TypeDefinition (fonction partielle)
  EnvironmentExp $\triangleq$ Symbol $\rightarrow$ ExpDefinition (fonction partielle)

Relation de sous-typage

- *env* de EnvironmentType
- Relation de sous-typage relative à *env* :
  - Pour tout type T, T est un sous-type de T.
  - Pour toute classe A, type_class(A) est un sous-type de type_class(Object).
  - Si une classe B étend une classe A dans l'environnement *env*, alors type_class(B) est un sous-type de type_class(A).
  - Si une classe C étend une classe B dans l'environnement *env* et si type_class(B) est un sous-type de T, alors type_class(C) est un sous-type de T.
  - Pour toute classe A, null est un sous-type de type_class(A).
- Notation : subtype(*env*, T_1, T_2)
  T_1 est un sous-type de T_2 relativement à *env*

Relation de sous-typage

```
class X { };
class Y extends X { };
class Z extends X { };
```



Autres opérations

- cf. II-[SyntaxeContextuelle] (section 2)
- Compatibilité pour l'affectation
- Compatibilité pour la conversion
- Compatibilité des opérations unaires et binaires

Environnements prédéfinis

- env_types_predef : types prédéfinis (dans EnvironnementType)

  env_types_predef $\triangleq$ {
  void $\mapsto$ (type, void),
  boolean $\mapsto$ (type, boolean),
  float $\mapsto$ (type, float),
  int $\mapsto$ (type, int),
  Object $\mapsto$ (class(0, env_exp_object), type_class(Object))
  }
- env_exp_object : environnement des champs et méthodes de Object (dans EnvironnementExp)

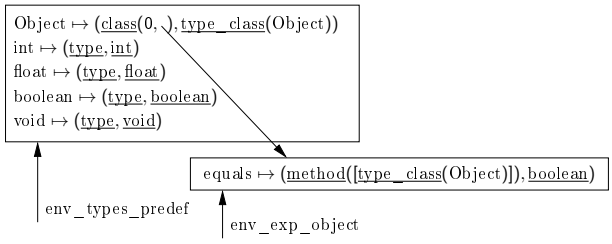
  env_exp_object $\triangleq$ {
  equals $\mapsto$ (method([type_class(Object)]), boolean) }

Implémentation des environnements

II-[SyntaxeContextuelle] section 10

- valeurs de EnvironmentExp comme listes chaînées de tables d'associations

  identificateur $\mapsto$ définition
- Environnements prédéfinis env_types_predef et env_exp_object.



Exemple

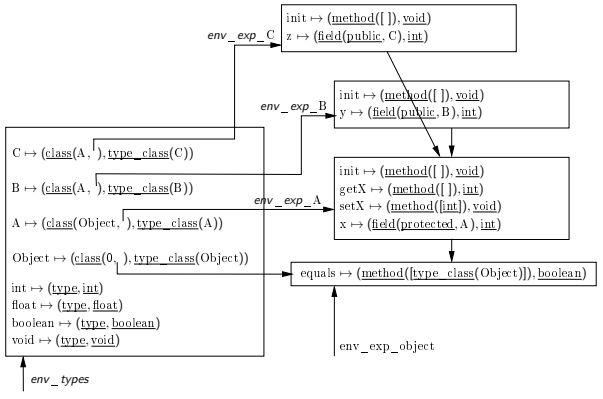
II-[SyntaxeContextuelle] section 10

```
class A {
   protected int x;
   void setX(int x) {
      this.x = x;
   }
   int getX() {
      return x;
   }
   void init() {
      x = 0;
   }
}
```

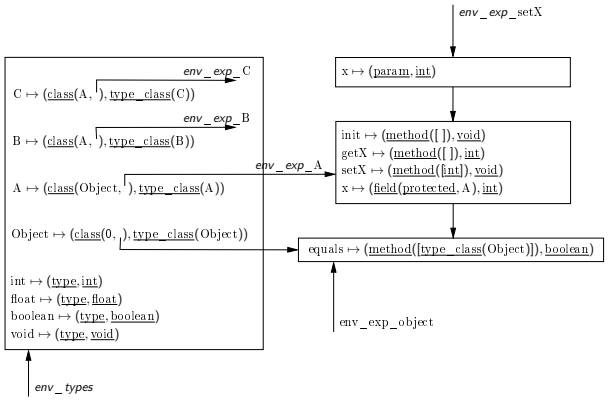
```
class B extends A {
   int y;
   void init() {
      setX(0);
      y = 0;
   }
}

class C extends A {
   int z;
   void init() {
      setX(0);
      z = 1;
   }
}
```


Environnement *env_exp* du corps des classes



Environnement d'analyse de la méthode *setX*



Conventions d'écriture dans la grammaire attribuée de Deca
Affectation des attributs

- Affectation explicite de la forme ***affectation*** *v* := *exp*.
  Règle (0.1)
  identifier ↓*env_exp* ↑*def*
  → Identifier ↑*name*
  affectation *def* := *env_exp*(*name*)
- Affectation implicite par expression fonctionnelle
  identifier ↓*env_exp* ↑*env_exp*(*name*)
  → Identifier ↑*name*

Conditions sur les attributs

- Clause condition
  Règle (3.28)
  rvalue ↓*env_types* ↓*env_exp* ↓*class* ↓$type_1$
  → **expr** ↓*env_types* ↓*env_exp* ↓*class* ↑$type_2$
  condition assign_compatible(*env_types*, $type_1$, $type_2$)
- Affectation : toute valeur d'attribut doit être définie
  identifier ↓*env_exp* ↑*def*
  → Identifier ↑*name*
  affectation *def* := *env_exp*(*name*)
  contraint *env_exp*(*name*) à être défini.

Conditions sur les attributs

- Filtrage d'un attribut synthétisé en partie droite
  Règle (3.29)
  condition ↓*env_types* ↓*env_exp* ↓*class*
  → **expr** ↓*env_types* ↓*env_exp* ↓*class* ↑boolean
  impose que la valeur de l'attribut synthétisé de **expr** soit le type boolean.
- Filtrage d'un attribut hérité en partie gauche
  Règle (3.73)
  rvalue_star ↓*env_types* ↓*env_exp* ↓*class* ↓[ ]
  → ε
  impose que la signature héritée en partie gauche soit la signature vide ([ ]).

Grammaires attribuées de la syntaxe contextuelle

- Vérifications contextuelles de Deca :
  trois passes sur le programme.
- Les règles contextuelles sont spécifiées à l'aide de trois grammaires attribuées.

Règles communes aux trois passes

- **identifier** ↓*env_exp* ↑*def* (0.1)
  → Identifier ↑*name*
  affectation *def* := *env_exp*(*name*)

  On doit trouver une définition associée au nom *name* dans l'environnement *env_exp*.

  type ↓*env_types* ↑*type* (0.2)
  → Identifier ↑*name*
  condition (_, *type*) ≜ *env_types*(*name*)

Passe 1

- Vérification du nom des classes et de la hiérarchie de classes
- Construction de l'environnement env_types, qui contient env_type_predef et les noms des différentes classes du programme
- **program** ↑*env_types* (1.1)
  → Program[
  list_decl_class ↓*env_types_predef* ↑*env_types*
  MAIN]

Passe 1

- **list_decl_class** ↓*env_types* ↑env_types_r (1.2)
  → {env_types_r := *env_types*}
  [(**decl_class** ↓*env_types* ↑env_types_r)*]

  À partir de l'environnement *env_types* hérité, on calcule l'environnement env_types_r résultant de la déclaration des classes.

  Si la liste de classes est vide, l'environnement résultant env_types_r est l'environnement *env_types* hérité.

Passe 1

- decl_class $\downarrow env\_types$ (1.3)
  $\uparrow\{name \mapsto (\underline{class}(super, \{\}), \underline{type\_class}(name))\} \oplus env\_types$
  $\rightarrow$ `DeclClass`[
  `Identifier` $\uparrow name$ `Identifier` $\uparrow super$
  LIST_DECL_FIELD LIST_DECL_METHOD
  ]
  condition $env\_types(super) = (\underline{class}(\_), \_)$
  - On récupère le nom de la super-classe *super*.
  - On vérifie que *super* fait partie de *env_types* et que c'est bien un nom de classe (condition).
  - L'environnement des types résultat est *env_types* auquel on ajoute la définition de la nouvelle classe.
  - Condition implicite (due à $\oplus$) : $env\_types(name)$ doit être non-défini.
  - Remarque : le profil de chaque classe est incomplet ; il ne contient que le nom de la super-classe mais pas l'environnement des champs et les méthodes (profil laissé vide {} et complété en passe 2).

Exercices

Question

Les programmes Deca suivants sont-ils corrects ?

1.
```
class A {
   B b;
}
```

2.
```
class A {
   void x;
}
```

3.
```
{ A a; }
```


Correction

1. `class A { B b; }`
   - Passe 1 : OK
     On a ajouté $A \mapsto (\underline{class}(\text{Object}, \{\}), \underline{type\_class}(A))$ dans *env_types*
   - Passe 2 : la classe *B* n'est pas déclarée
     Règles : (2.1 – 2.5) **(0.2)**
     $env\_types(B)$ n'est pas défini $\Rightarrow$ *B* n'est pas déclaré

2. `class A { void x; }`
   - Passe 1 : OK
   - Passe 2 : champ de type void interdit
     Règles : (2.1 – 2.4) (2.5 - 0.2) **(2.5)**
     condition $type \neq \underline{void}$

3. `{ A a; }`
   - Passe 1 : OK ; passe 2 : OK
   - Passe 3 : la classe A n'est pas déclarée
     Règles : (3.1) (3.4) (3.18) (3.16) (3.17) **(0.2)**
     $env\_types(A)$ n'est pas défini $\Rightarrow$ *A* n'est pas déclaré

Erreurs contextuelles

- Grammaires attribuées de Deca :
  spécification de la syntaxe contextuelle du langage
- Examen systématique de toutes les règles du langage :
  Identification de toutes les erreurs contextuelles possibles
- Contraintes potentielles
  - condition
  - filtrage d'un attribut hérité en partie gauche
  - filtrage d'un attribut synthétisé en partie droite
  - opération partielle

Examen systématique des règles

(0.1)
- raison : opération partielle
- message : « identificateur non déclaré »

(0.2)
- raison : opération partielle
- message : « identificateur de type non déclaré »

(1.3)
- raison : opération partielle *env_types*(*super*)
- message : « identificateur non déclaré »
- raison : condition
- message : « identificateur de classe attendu »
- raison : opération partielle $\oplus$
- message : « classe ou type déjà déclaré »

Examen systématique des règles

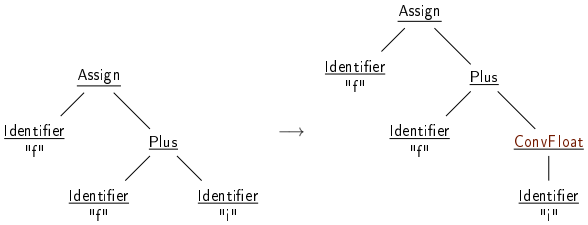
(2.3)
- raison : condition et opération partielle *env_types*(*super*)
- message : en fait erreur interne
- raison : opération partielle $\oplus$
- message : « un nom de méthode redéclare un nom de champ »

(2.4)
- raison : opération partielle $\oplus$
- message : « nom d'attribut déjà déclaré »

A continuer...

Enrichissement de l'arbre abstrait

- cf. IV-[ArbreEnrichi]
- Ajout de Nœud ConvFloat
- Exemple : f = f + i ;
  avec les déclarations “float f ;” et “int i ;”



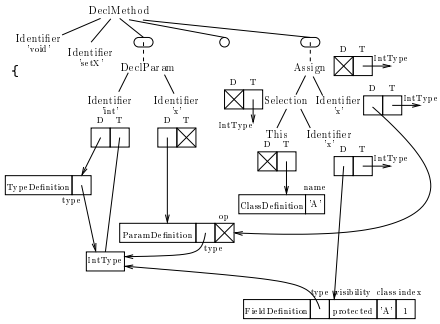
Décoration de l'arbre abstrait

- Décors : informations supplémentaires stockées dans l'arbre abstrait
- Permettent de faciliter la génération de code (étape C)
- Deux types de décors :
  - `Definition` : associés aux nœuds `Identifier`
    (Nature et type de l'identificateur)
  - `Type` : associés aux nœuds qui dérivent de EXPR dans la grammaire d'arbres
    (Type de l'expression)
- Remarque :
  - Il n'est pas nécessaire que les occurrences de déclaration d'un identificateur aient un `Type` associé.
  - Les occurrences d'utilisation d'un identificateur ont un `Type` associé en plus de sa `Definition`.

Décoration de l'arbre : Exemple

cf. IV-[Exemple] (section 2.3 : corps de la méthode setX)

```
void setX(int x) {
    this.x = x;
// this.x :
//    champ x
// x :
//    parametre x
}
```



Index des champs et des méthodes

- À chaque `Definition` de champ et de méthode est associé un index (numéro du champ ou de la méthode dans la classe).
- À chaque `Definition` de classe sont associés un nombre de champs (numberOfFields) et un nombre de méthodes (numberOfMethods).
- Ces informations doivent être mise à jour lors de l'étape B, en passe 2.

Index des champs et des méthodes

Exemple

```
class A {
   int x;
   void f() {}
   int y;
   void g() {}
}
```

```
class B extends A {
   int z;
   void f() {}
   void h() {}
   int x;
}
```

- Object :
  - 0 champ
  - 1 méthode : Object.equals (index 1)
- A :
  - 2 champs : A.x (index 1) A.y (index 2)
  - 3 méthodes : Object.equals (index 1) A.f (index 2) A.g (index 3)
- B :
  - 4 champs : A.x (index 1) A.y (index 2) B.z (index 3) B.x (index 4)
  - 4 méthodes : Object.equals (index 1) B.f (index 2) A.g (index 3) B.h (index 4)

Index des champs et des méthodes

Exemple

```
class A {
   int x;
   void f() {}
   int y;
   void g() {}
}
```

```
class B extends A {
   int z;
   void f() {}
   void h() {}
   int x;
}
```

Remarques

- Dans B, « void f() {} » redéfinit la méthode f
  - elle garde donc le même index que A.f
  - B a 4 méthodes et non 5
- Dans B « int x ; » déclare un *nouveau* champ B.x
  - B a 4 champs (A.x, A.y, B.z, B.x)
- Liaison dynamique sur les méthodes et non sur les champs.

Sémantique de partage

- Les `Definitions` sont partagées.
- Toutes les occurrences d'un même identificateur sont décorées avec la même `Definition`.
- cf. IV-[Exemple] (section 2.3)

Principaux répertoires concernés

cf. I-[Consignes]

- `src/main/java/fr/ensimag/deca/tree/`
  - Classes qui définissent les arbres
  - Méthodes de parcours de l'arbre abstrait
- `src/main/java/fr/ensimag/deca/context/`
  - Fichiers sources Java concernant l'étape B
- `src/test/java/fr/ensimag/deca/context/`
  - Fichiers Java de test concernant l'étape B
- `src/test/deca/context/`
  - Fichiers Deca de test

Classes fournies

cf. I-[Consignes]

- Classes pour les types :
  - Classe abstraite Type, dont dérivent les classes StringType, VoidType, BooleanType, IntType, FloatType, NullType et ClassType.

  Compléter le code des méthodes
  - boolean sameType(Type otherType) ;
  - boolean isSubClassOf(ClassType potentialSuperClass) ;
- Classes pour les définitions :
  - Classe abstraite Definition ;
  - Classes VariableDefinition, ParamDefinition, ClassDefinition, FieldDefinition, MethodDefinition...
- Classe Signature (pour la signature des méthodes)
- Classe EnvironmentExp (squelette fourni, à implémenter)
- Exception ContextualError, levée lorsqu'on détecte une erreur contextuelle (on s'arrête à la première erreur contextuelle).

Parcours de l'arbre abstrait

- Trois parcours à effectuer
- Parcours basés sur la syntaxe abstraite du langage II-[SyntaxeAbstraite]
- Une méthode abstraite verifyXyz pour chaque non terminal XYZ (ou classe AbstractXyz) de la grammaire d'arbres.
  - Dans la classe AbstractProgram :

```
abstract void verifyProgram(DecacCompiler compiler)
   throws ContextualError;
```

  - Dans la classe AbstractMain :

```
abstract void verifyMain(DecacCompiler compiler)
   throws ContextualError;
```

  - ...
- Ces méthodes abstraites sont ensuite implémentées dans les sous-classes (Program, EmptyMain, Main...)

Parcours de l'arbre abstrait

- Codage des attributs de la grammaire attribuée :
  - paramètres pour les attributs hérités ;
  - résultat de méthode pour un attribut synthétisé.
- Exceptions
  - env_exp : en général, dans la grammaire, on trouve un attribut hérité (pour la valeur « avant ») et un attribut synthétisé (pour la valeur « après » application de la règle). Dans l'implémentation, on utilise un seul objet de type EnvironmentExp, que l'on mute.
  - env_types : on peut stocker un environnement dans les objets de type DecacCompiler, de cette façon il n'est pas nécessaire de le passer en paramètres dans la plupart des méthodes.
  - On peut avoir besoin de paramètres supplémentaires pour les décorations.

Travail à effectuer pour l'étape B

cf. I-[Consignes]

- Compléter la classe EnvironmentExp.
- Classe de test de la classe EnvironmentExp
- Implantation des trois parcours de l'arbre
  - Implantation des méthodes verifyXyz dans les classes Java qui définissent l'arbre abstrait
  - Décoration et enrichissement de l'arbre
- On fournit le script `test_context`, qui appelle la classe `ManualTestContext` qui permet de tester l'analyse contextuelle.