

Projet Génie Logiciel

Étape B

Analyse contextuelle

Projet GL

Ensimag
Grenoble INP

9 décembre 2022



Sommaire

- 1 Grammaires attribuées
- 2 Syntaxe contextuelle de Deca
- 3 Implantation de l'analyse contextuelle

Sommaire de cette section

1 Grammaires attribuées

- Rappels
- Syntaxe contextuelle des langages à structure de bloc

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 $X \downarrow_{att}$

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 $X \uparrow_{att}$

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

$$X \rightarrow Y_1 Y_2 \dots Y_n$$

on définit :

- ▶ les attributs hérités de Y_i en fonction des attributs de X, Y_1, \dots, Y_n ;
- ▶ les attributs synthétisés de X en fonction des attributs de X, Y_1, \dots, 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 :

$$\begin{aligned} S &\rightarrow AC \\ A &\rightarrow \varepsilon \mid aAb \\ C &\rightarrow \varepsilon \mid cC \end{aligned}$$

- Variante, avec style EBNF :

$$\begin{aligned} S &\rightarrow AC \\ A &\rightarrow \varepsilon \mid aAb \\ C &\rightarrow c^* \end{aligned}$$

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 :

$$S \rightarrow A \quad C$$
$$A \rightarrow \varepsilon$$
$$A \rightarrow a \ A \quad b$$
$$C \rightarrow \varepsilon$$
$$C \rightarrow c \ C$$

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 :

$$S \rightarrow A \uparrow n \ C$$

$$A \uparrow 0 \rightarrow \varepsilon$$

$$A \uparrow n + 1 \rightarrow a \ A \uparrow n \ b$$

$$C \rightarrow \varepsilon$$

$$C \rightarrow c \ C$$

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 :

$$S \rightarrow A \uparrow n \ C \uparrow p$$

$$A \uparrow 0 \rightarrow \varepsilon$$

$$A \uparrow n + 1 \rightarrow a \ A \uparrow n \ b$$

$$C \uparrow 0 \rightarrow \varepsilon$$

$$C \uparrow n + 1 \rightarrow c \ C \uparrow n$$

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 :

$$S \rightarrow A \uparrow n \ C \uparrow p$$

condition $n = p$

$$A \uparrow 0 \rightarrow \varepsilon$$

$$A \uparrow n + 1 \rightarrow a \ A \uparrow n \ b$$

$$C \uparrow 0 \rightarrow \varepsilon$$

$$C \uparrow n + 1 \rightarrow c \ C \uparrow n$$

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 :

$$S \rightarrow A \uparrow n \ C \downarrow n$$

$$A \uparrow 0 \rightarrow \varepsilon$$

$$A \uparrow n + 1 \rightarrow a \ A \uparrow n \ b$$

$$C \downarrow n \rightarrow \varepsilon$$

condition $n = 0$

$$C \downarrow n \rightarrow c \ C \downarrow n - 1$$

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 :

$$S \rightarrow A \downarrow n \ C \uparrow n$$

$$A \downarrow 0 \rightarrow \varepsilon$$

$$A \downarrow n \rightarrow a \ A \downarrow n - 1 \ b$$

$$C \uparrow n \rightarrow \{ n := 0 \} (c \{ n := n + 1 \})^*$$

Sommaire de cette section

1 Grammaires attribuées

- Rappels
- Syntaxe contextuelle des langages à structure de bloc

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 \Rightarrow 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();  
    }  
}
```

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

```
declare                                -- niveau 1
  x, y : integer;                     -- niveau 2
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

$\text{Type} \triangleq \{ \text{entier}, \text{booleen} \}$

- Nature des identificateurs : type, variable ou constante

$\text{Nature} \triangleq \{ \text{var}, \text{type}, \text{const} \}$

- Définition d'un identificateur : nature et type

$\text{Définition} \triangleq \text{Nature} \times \text{Type}$

- Symbol : domaine des identificateurs

- Environnement : associe à un identificateur sa définition

$\text{Environnement} \triangleq \text{Symbol} \rightarrow \text{Définition}$ (fonction partielle)

- $\text{env}(x)$: définition associée à $x \in \text{Symbol}$ dans env .

- $\text{dom}(\text{env})$: domaine de l'environnement env (ensemble des identificateurs auxquels est associée une définition)

Environnements

- Environnement prédéfini :

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

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

env_3	niveau 3
env_2	niveau 2
Predef	niveau 1

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

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 $dom(env_1) \cap dom(env_2) \neq \emptyset$
- ▶ $(env_1 \oplus env_2)(x) \triangleq \begin{cases} env_1(x) & \text{si } x \in dom(env_1) \\ env_2(x) & \text{si } x \in dom(env_2) \end{cases}$

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

- Empilement de deux environnements

- ▶ $(env_1/ env_2)(x) \triangleq \begin{cases} env_1(x) & \text{si } x \in dom(env_1) \\ env_2(x) & \text{si } x \notin dom(env_1) \text{ et } x \in dom(env_2) \\ \text{indéfini} & \text{sinon} \end{cases}$

Syntaxe abstraite du mini-langage à blocs

PROG	→	BLOC
BLOC	→	<u>Bloc</u> [LIST_DECL LIST_INST]
LIST_DECL	→	[DECL*]
DECL	→	<u>Decl</u> [IDF TYPE]
TYPE	→	IDF
LIST_INST	→	[INST*]
INST	→	BLOC <u>Assign</u> [IDF EXP]
EXP	→	IDF <u>Num</u> <u>Plus</u> [EXP EXP]
IDF	→	<u>Idf</u>

Profils des symboles de la syntaxe contextuelle

- BLOC $\downarrow env$ 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 $\downarrow t \uparrow env$ t : Type
- TYPE $\downarrow env_glob \uparrow t$ t : Type

Profils des symboles de la syntaxe contextuelle

- BLOC $\downarrow env$ 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 $\downarrow t \uparrow env$ t : Type
- TYPE $\downarrow env_glob \uparrow t$ t : Type
- LIST_INST $\downarrow env$
- INST $\downarrow env$
- EXP $\downarrow env \uparrow t$ t : TYPE
- Idf $\uparrow nom$ nom : Symbol

Règles de la syntaxe contextuelle

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

Règles de la syntaxe contextuelle

PROG \rightarrow BLOC $\downarrow \text{Predef}$

BLOC $\downarrow \text{env_glob}$ \rightarrow Bloc [LIST_DECL $\downarrow \text{env_glob} \uparrow \text{env}$
LIST_INST $\downarrow \text{env} / \text{env_glob}$]

LIST_DECL $\downarrow \text{env_glob} \uparrow \text{env}$ \rightarrow
 $\{ \text{env} := \emptyset \} [(\text{DECL} \downarrow \text{env_glob} \uparrow \text{env}_1 \{ \text{env} := \text{env} \oplus \text{env}_1 \})^*]$

condition implicite : $\text{dom}(\text{env})$ initial et $\text{dom}(\text{env}_1)$ disjoints

Règles de la syntaxe contextuelle

PROG \rightarrow BLOC $\downarrow \text{Predef}$

BLOC $\downarrow \text{env_glob}$ \rightarrow Bloc [LIST_DECL $\downarrow \text{env_glob} \uparrow \text{env}$
LIST_INST $\downarrow \text{env} / \text{env_glob}$]

LIST_DECL $\downarrow \text{env_glob} \uparrow \text{env}$ \rightarrow
 $\{ \text{env} := \emptyset \} [(\text{DECL} \downarrow \text{env_glob} \uparrow \text{env}_1 \{ \text{env} := \text{env} \oplus \text{env}_1 \})^*]$
 condition implicite : $\text{dom}(\text{env})$ initial et $\text{dom}(\text{env}_1)$ disjoints

DECL $\downarrow \text{env_glob} \uparrow \text{env}$ \rightarrow Decl [IDF $\downarrow t \uparrow \text{env}$ TYPE $\downarrow \text{env_glob} \uparrow t$]

Règles de la syntaxe contextuelle

PROG \rightarrow BLOC $\downarrow \text{Predef}$

BLOC $\downarrow \text{env_glob}$ \rightarrow Bloc [LIST_DECL $\downarrow \text{env_glob} \uparrow \text{env}$
LIST_INST $\downarrow \text{env} / \text{env_glob}$]

LIST_DECL $\downarrow \text{env_glob} \uparrow \text{env}$ \rightarrow
 $\{ \text{env} := \emptyset \} [(\text{DECL} \downarrow \text{env_glob} \uparrow \text{env}_1 \{ \text{env} := \text{env} \oplus \text{env}_1 \})^*]$
 condition implicite : $\text{dom}(\text{env})$ initial et $\text{dom}(\text{env}_1)$ disjoints

DECL $\downarrow \text{env_glob} \uparrow \text{env}$ \rightarrow Decl [IDF $\downarrow t \uparrow \text{env}$ TYPE $\downarrow \text{env_glob} \uparrow t$]

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

Règles de la syntaxe contextuelle

PROG \rightarrow BLOC $\downarrow \text{Predef}$

BLOC $\downarrow \text{env_glob}$ \rightarrow Bloc [LIST_DECL $\downarrow \text{env_glob} \uparrow \text{env}$
LIST_INST $\downarrow \text{env} / \text{env_glob}$]

LIST_DECL $\downarrow \text{env_glob} \uparrow \text{env}$ \rightarrow
 $\{ \text{env} := \emptyset \} [(\text{DECL} \downarrow \text{env_glob} \uparrow \text{env}_1 \{ \text{env} := \text{env} \oplus \text{env}_1 \})^*]$
 condition implicite : $\text{dom}(\text{env})$ initial et $\text{dom}(\text{env}_1)$ disjoints

DECL $\downarrow \text{env_glob} \uparrow \text{env}$ \rightarrow Decl [IDF $\downarrow t \uparrow \text{env}$ TYPE $\downarrow \text{env_glob} \uparrow t$]

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

TYPE $\downarrow \text{env_glob} \uparrow t$ \rightarrow Idf $\uparrow \text{nom}$
 condition : $(\underline{\text{type}}, t) \triangleq \text{env_glob}(\text{nom})$
 condition implicite additionnelle : $\text{nom} \in \text{dom}(\text{env_glob})$

Règles contextuelles (suite)

$$\text{LIST_INST} \downarrow_{env} \rightarrow [(\text{INST} \downarrow_{env})^*]$$

Règles contextuelles (suite)

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

INST $\downarrow env$ \rightarrow BLOC $\downarrow env$

\rightarrow Assign[Idf $\uparrow nom$ EXP $\downarrow env \uparrow t$]

condition : $env(nom) = (\underline{var}, t)$

Règles contextuelles (suite)

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

INST $\downarrow env$ \rightarrow BLOC $\downarrow env$

\rightarrow Assign[Idf $\uparrow nom$ EXP $\downarrow env \uparrow t$]

condition : $env(nom) = (\underline{var}, t)$

EXP $\downarrow env \uparrow t$ \rightarrow Idf $\uparrow nom$

condition : $(nat, t) \triangleq env(nom)$ et $nat \in \{\underline{var}, \underline{const}\}$

EXP $\downarrow env \uparrow \underline{entier}$ \rightarrow Num

\rightarrow Plus[EXP $\downarrow env \uparrow t_1$ EXP $\downarrow env \uparrow t_2$]

condition : $t_1 = t_2 = \underline{entier}$

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
```

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;
```

Les programmes suivants sont-ils acceptés ?

- ❶

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

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

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

Sommaire

- 1 Grammaires attribuées
- 2 Syntaxe contextuelle de Deca
- 3 Implantation de l'analyse contextuelle

Sommaire de cette section

2 Syntaxe contextuelle de Deca

- Introduction
- Domaines d'attributs
- Opérations et prédicats sur les domaines d'attributs
- Environnements
- Conventions d'écriture
- Grammaires attribuées de la syntaxe contextuelle de Deca
- Erreurs contextuelles

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 apparaît 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`

Sommaire de cette section

2 Syntaxe contextuelle de Deca

- Introduction
- Domaines d'attributs
- Opérations et prédicats sur les domaines d'attributs
- Environnements
- Conventions d'écriture
- Grammaires attribuées de la syntaxe contextuelle de Deca
- Erreurs contextuelles

Domaines d'attributs de la grammaire attribuée de Deca

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

Domaines d'attributs de la grammaire attribuée de Deca

- Extension : nom de la super-classe, ou 0 pour Object
 $\text{Extension} \triangleq \text{Symbol} \cup \{0\}$
- Profil d'une classe : nom de la super-classe et environnement des champs et méthodes de la classe
 $\text{Profil} \triangleq \text{Extension} \times \text{EnvironmentExp}$
- Definition d'un identificateur : sa nature et son type
 $\text{TypeDefinition} \triangleq \text{TypeNature} \times \text{Type}$
 $\text{ExpDefinition} \triangleq \text{ExpNature} \times \text{Type}$
 - ▶ 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
 $\text{EnvironmentType} \triangleq \text{Symbol} \rightarrow \text{TypeDefinition}$ (fonction partielle)
 $\text{EnvironmentExp} \triangleq \text{Symbol} \rightarrow \text{ExpDefinition}$ (fonction partielle)

Sommaire de cette section

2 Syntaxe contextuelle de Deca

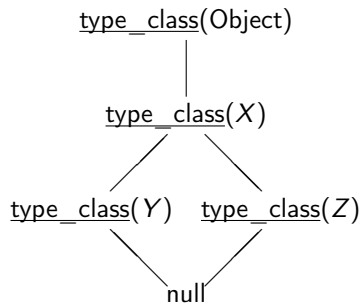
- Introduction
- Domaines d'attributs
- Opérations et prédicats sur les domaines d'attributs
- Environnements
- Conventions d'écriture
- Grammaires attribuées de la syntaxe contextuelle de Deca
- Erreurs contextuelles

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 : $\text{subtype}(\text{env}, T_1, T_2)$
 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 { };
```



void boolean float int string

Autres opérations

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

Sommaire de cette section

2 Syntaxe contextuelle de Deca

- Introduction
- Domaines d'attributs
- Opérations et prédicats sur les domaines d'attributs
- **Environnements**
- Conventions d'écriture
- Grammaires attribuées de la syntaxe contextuelle de Deca
- Erreurs contextuelles

Environnements prédéfinis

- `env_types_predef` : types prédéfinis (dans `EnvironnementType`)

$$\text{env_types_predef} \triangleq \{$$

<code>void</code>	\mapsto	$(\underline{\text{type}}, \underline{\text{void}}),$
<code>boolean</code>	\mapsto	$(\underline{\text{type}}, \underline{\text{boolean}}),$
<code>float</code>	\mapsto	$(\underline{\text{type}}, \underline{\text{float}}),$
<code>int</code>	\mapsto	$(\underline{\text{type}}, \underline{\text{int}}),$
<code>Object</code>	\mapsto	$(\underline{\text{class}}(0, \text{env_exp_object}), \underline{\text{type_class}}(\text{Object}))$

$$\}$$

- `env_exp_object` : environnement des champs et méthodes de `Object` (dans `EnvironnementExp`)

$$\text{env_exp_object} \triangleq \{$$

<code>equals</code>	\mapsto	$(\underline{\text{method}}([\underline{\text{type_class}}(\text{Object})]), \underline{\text{boolean}})$
---------------------	-----------	--

$$\}$$

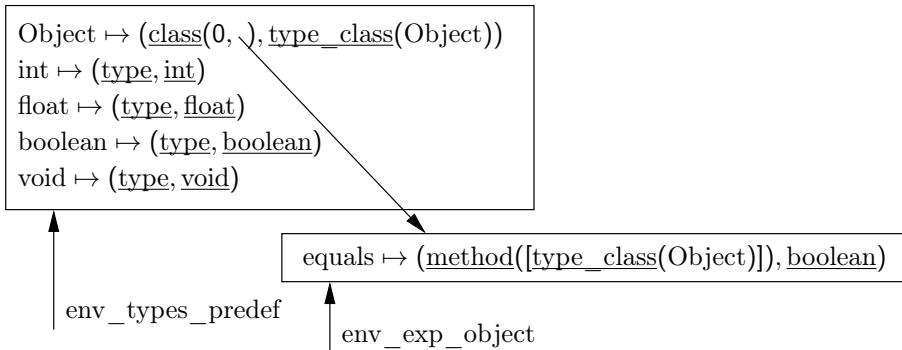
Implémentation des environnements

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.



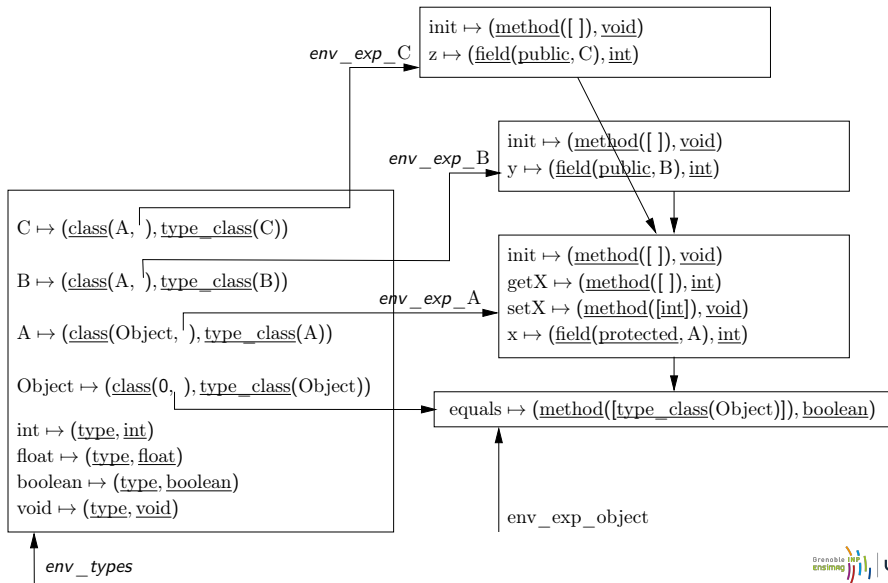
Exemple

II-[SyntaxeContextuelle] section 10

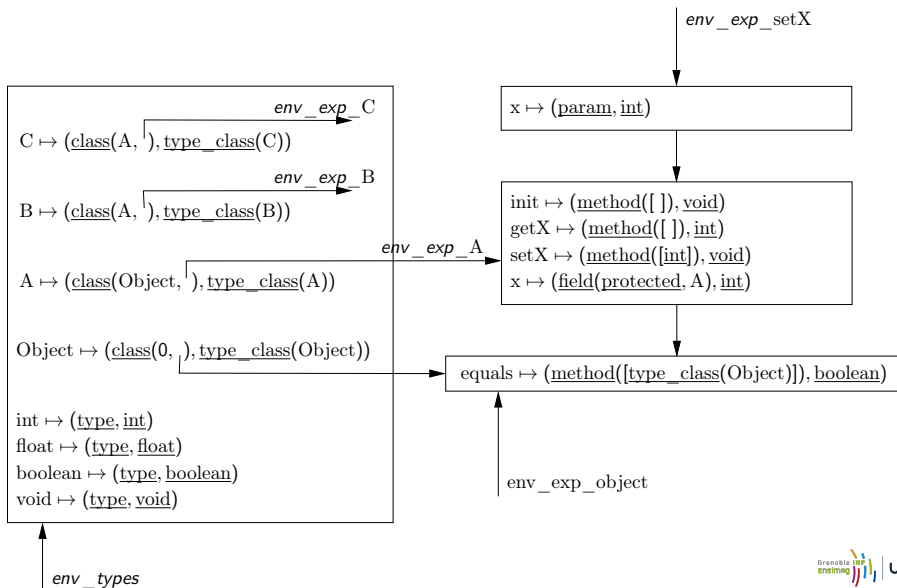
```
class A {  
    protected int x;  
    void setX(int x) {  
        this.x = x;  
    }  
    int getX() {  
        return x;  
    }  
    void init() {  
        x = 0;  
    }  
}
```

```
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*



Sommaire de cette section

2 Syntaxe contextuelle de Deca

- Introduction
- Domaines d'attributs
- Opérations et prédicats sur les domaines d'attributs
- Environnements
- **Conventions d'écriture**
- Grammaires attribuées de la syntaxe contextuelle de Deca
- Erreurs contextuelles

Conventions d'écriture dans la grammaire attribuée de Deca

Affectation des attributs

- Affectation explicite de la forme ***affectation*** $v := exp$.

Règle (0.1)

identifieur $\downarrow env_exp \uparrow def$
 $\rightarrow \underline{\text{Identifieur}} \uparrow name$
affectation $def := env_exp(name)$

- Affectation implicite par expression fonctionnelle

identifieur $\downarrow env_exp \uparrow env_exp(name)$
 $\rightarrow \underline{\text{Identifieur}} \uparrow name$

Conditions sur les attributs

- Clause condition

Règle (3.28)

rvalue $\downarrow env_types \downarrow env_exp \downarrow class \downarrow type_1$
 \rightarrow **expr** $\downarrow env_types \downarrow env_exp \downarrow class \uparrow type_2$
condition $assign_compatible(env_types, type_1, type_2)$

- Affectation : toute valeur d'attribut doit être définie

identifier $\downarrow env_exp \uparrow def$
 \rightarrow Identifier $\uparrow name$
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 $\downarrow env_types \downarrow env_exp \downarrow class$

\rightarrow **expr** $\downarrow env_types \downarrow env_exp \downarrow class \uparrow \underline{boolean}$

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 $\downarrow env_types \downarrow env_exp \downarrow class \downarrow []$

$\rightarrow \varepsilon$

impose que la signature héritée en partie gauche soit la signature vide ([]).

Sommaire de cette section

2 Syntaxe contextuelle de Deca

- Introduction
- Domaines d'attributs
- Opérations et prédicats sur les domaines d'attributs
- Environnements
- Conventions d'écriture
- Grammaires attribuées de la syntaxe contextuelle de Deca
- Erreurs contextuelles

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** $\downarrow env_exp \uparrow def$ (0.1)
→ Identifier $\uparrow name$
affectation $def := env_exp(name)$

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

- type** $\downarrow env_types \uparrow type$ (0.2)
→ Identifier $\uparrow name$
condition $(_, type) \triangleq env_types(name)$

Passé 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** $\uparrow env_types$ (1.1)
 \rightarrow Program[
 list_decl_class $\downarrow env_types_predef$ $\uparrow env_types$
 MAIN]

Passe 1

$$\bullet \text{ list_decl_class } \downarrow \text{env_types} \uparrow \text{env_types}_r \quad (1.2)$$
$$\rightarrow \{ \text{env_types}_r := \text{env_types} \}$$
$$[(\text{decl_class } \downarrow \text{env_types}_r \uparrow \text{env_types}_r)^*]$$

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.

Passé 1

• $\text{decl_class} \downarrow \text{env_types}$ (1.3)

$\uparrow \{ \text{name} \mapsto (\text{class}(\text{super}, \{\}), \text{type_class}(\text{name})) \} \oplus \text{env_types}$

$\rightarrow \text{DeclClass}[$

$\text{Identifier} \uparrow \text{name} \quad \text{Identifier} \uparrow \text{super}$

$\text{LIST_DECL_FIELD} \quad \text{LIST_DECL_METHOD}$

$]$

condition $\text{env_types}(\text{super}) = (\text{class}(_), _)$

- ▶ 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) : $\text{env_types}(\text{name})$ 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 $A \mapsto (\text{class}(\text{Object}, \{\}), \text{type_class}(A))$ 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

Correction

① `class A { B b; }`

- Passe 1 : OK

On a ajouté $A \mapsto (\text{class}(\text{Object}, \{\}), \text{type_class}(A))$ 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 void

Correction

❶ `class A { B b; }`

- ▶ Passe 1 : OK

On a ajouté $A \mapsto (\text{class}(\text{Object}, \{\}), \text{type_class}(A))$ 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

❷ `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 void

❸ `{ 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

Sommaire de cette section

2 Syntaxe contextuelle de Deca

- Introduction
- Domaines d'attributs
- Opérations et prédicats sur les domaines d'attributs
- Environnements
- Conventions d'écriture
- Grammaires attribuées de la syntaxe contextuelle de Deca
- Erreurs contextuelles

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 »

Examen systématique des règles

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

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

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 »

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 »

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

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 »

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 »

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...

Sommaire

- 1 Grammaires attribuées
- 2 Syntaxe contextuelle de Deca
- 3 **Implantation de l'analyse contextuelle**

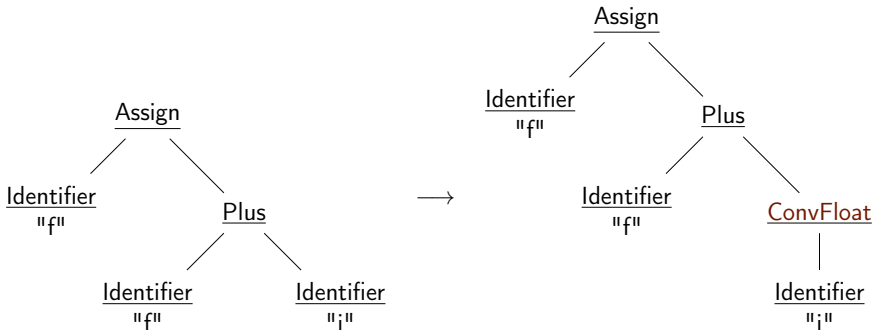
Sommaire de cette section

3 Implantation de l'analyse contextuelle

- Décoration et enrichissement de l'arbre abstrait
- Classes fournies
- Parcours de l'arbre abstrait
- Travail à effectuer pour l'étape B

Enrichissement de l'arbre abstrait

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

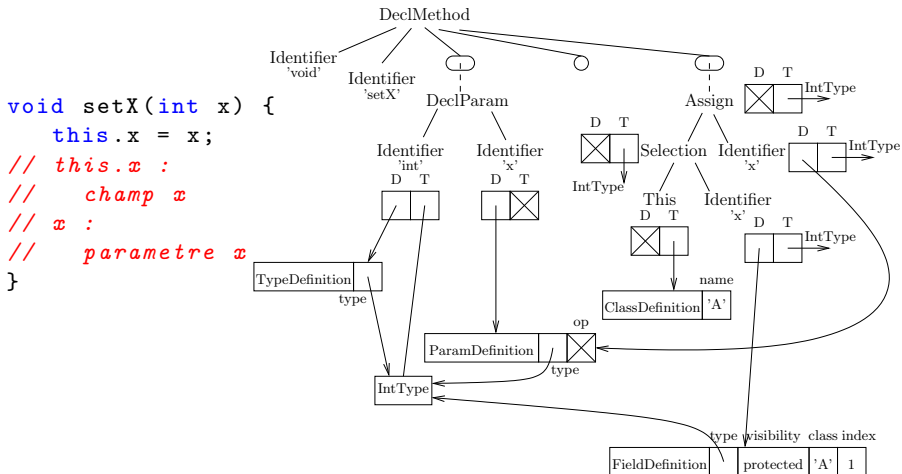


Décoration de l'arbre abstrait

- 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)



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)

Sommaire de cette section

3 Implantation de l'analyse contextuelle

- Décoration et enrichissement de l'arbre abstrait
- **Classes fournies**
- Parcours de l'arbre abstrait
- Travail à effectuer pour l'étape B

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).

Sommaire de cette section

3 Implantation de l'analyse contextuelle

- Décoration et enrichissement de l'arbre abstrait
- Classes fournies
- Parcours de l'arbre abstrait
- Travail à effectuer pour l'étape B

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.

Sommaire de cette section

3 Implantation de l'analyse contextuelle

- Décoration et enrichissement de l'arbre abstrait
- Classes fournies
- Parcours de l'arbre abstrait
- Travail à effectuer pour l'étape B

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.