Projet Génie Logiciel

Étape A

Analyse lexicale, syntaxique et construction de l'arbre abstrait

Projet GL

Ensimag
Grenoble INP

2 janvier 2023

Sommaire

1 Introduction

2 Spécifications

3 Analyse lexicale et syntaxique avec ANTLR
- Arbre abstrait représentant un programme Deca
- Écriture du Parser ANTLR et construction de l'arbre

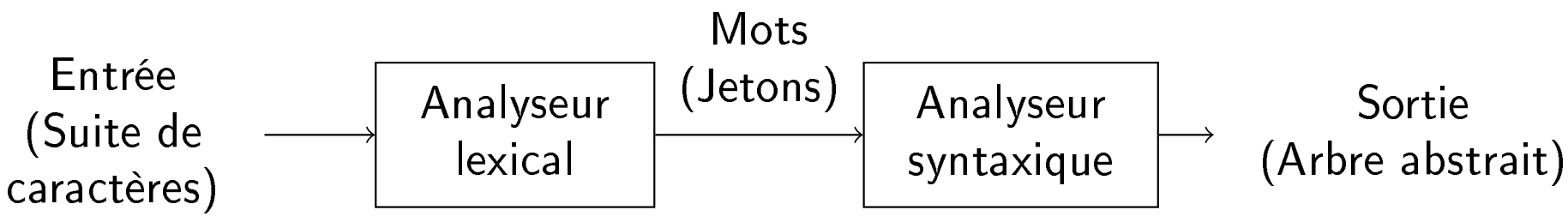
4 Notion d'« erreur »

Sommaire de cette section

1 Introduction
- Vue d'ensemble de l'étape A
- Analyse lexicale
- Analyse syntaxique
- ANTLR

Vue d'ensemble de l'étape A

- Étape A
  - Analyse lexicale
  - Analyse syntaxique
  - Construction de l'arbre abstrait
- Une passe sur le programme source Deca



Sommaire de cette section

1 Introduction
- Vue d'ensemble de l'étape A
- Analyse lexicale
- Analyse syntaxique
- ANTLR

Analyse lexicale

Partitionner une suite de caractères en une suite de mots.

- la suite de caractères :
  - = programme source en Deca
- les « mots » :
  - = unité lexicale
- à chaque unité lexicale est associé :
  - un « jeton » ou « token »

Analyse lexicale - Exemple

Prenons la suite de caractères : « $x = 2.5 * (y + 1);$ »
La suite de jetons correspondante est :

IDENT	['x']
EQUALS	['=']
FLOAT	['2.5']
TIMES	['*']
OPARENT	['(']
IDENT	['y']
PLUS	['+']
INT	['1']
CPARENT	[')']
SEMI	[';']

Sommaire de cette section

1 Introduction
- Vue d'ensemble de l'étape A
- Analyse lexicale
- Analyse syntaxique
- ANTLR

Analyse syntaxique

- L'analyse syntaxique permet de déterminer si une suite de mots est une phrase du langage.

$\Rightarrow$ Est-ce qu'une suite de jetons correspond à un programme Deca syntaxiquement correct ?

Outil
- La *grammaire hors-contexte* définit la syntaxe concrète du langage Deca, autrement dit l'ensemble des programmes syntaxiquement corrects (cf. II-[Syntaxe])

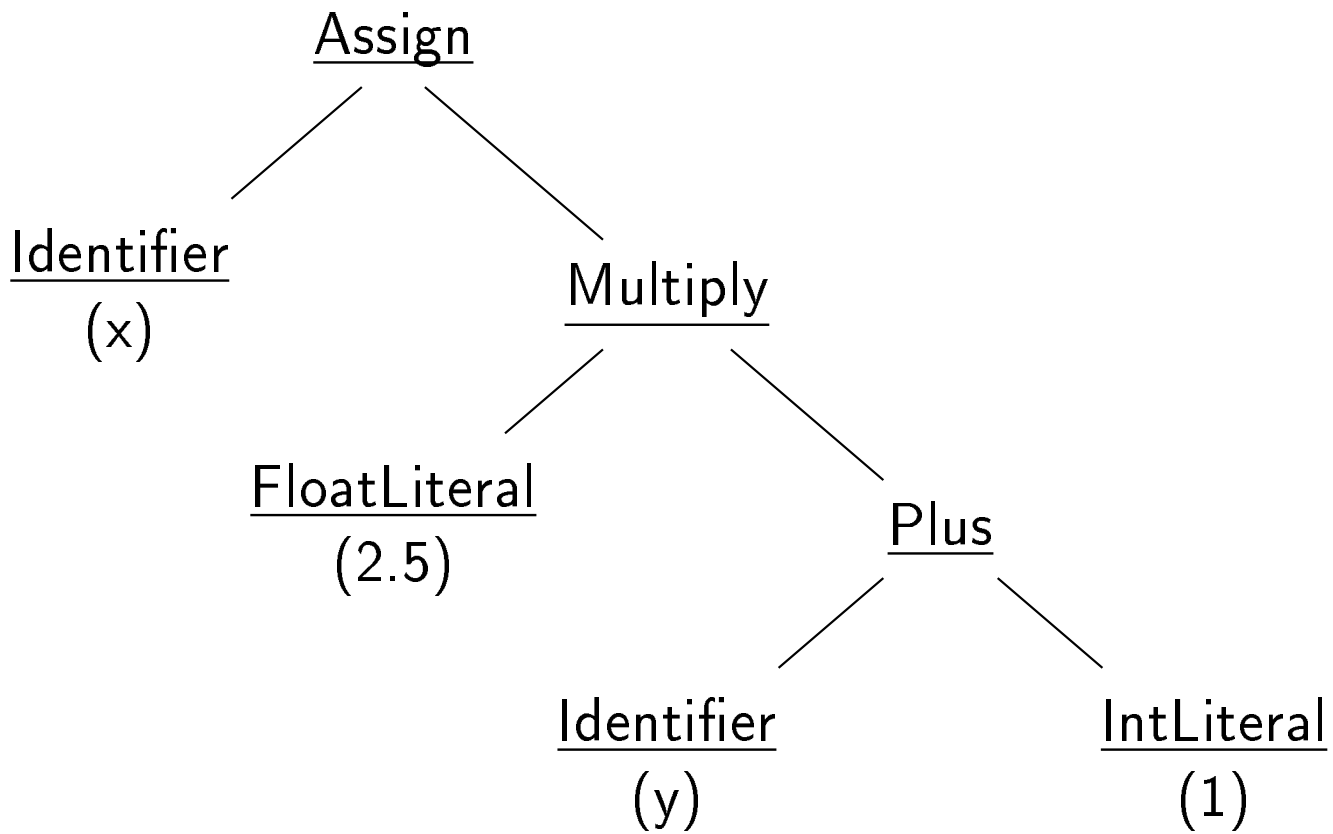
Construction de l'arbre abstrait

L'arbre abstrait est :

- une représentation structurée du programme Deca ;
- construit lors de l'analyse syntaxique.

Construction de l'arbre abstrait - Exemple

L'arbre abstrait correspondant à l'instruction : « $x = 2.5 * (y + 1);$ »



Sommaire de cette section

1 Introduction
- Vue d'ensemble de l'étape A
- Analyse lexicale
- Analyse syntaxique
- ANTLR

ANTLR : ANother Tool for Language Recognition

- ANTLR permet de générer des reconnaisseurs pour des langages :
  - analyseurs lexicaux (« *lexer* » en anglais),
  - analyseurs syntaxiques (« *parser* » en anglais).
- ANTLR est multi-langages : permet de générer des analyseurs en Ada 95, C, C#, Java, JavaScript, Perl, Python, Standard ML... etc.
- Analyse descendante ALL(*)
  Généralisation de l'analyse descendante LL(1)

Sommaire

1 Introduction

2 Spécifications

3 Analyse lexicale et syntaxique avec ANTLR
- Arbre abstrait représentant un programme Deca
- Écriture du Parser ANTLR et construction de l'arbre

4 Notion d'« erreur »

Sommaire de cette section

2 Spécifications
- Lexicographie
- Syntaxe

Lexicographie

- La lexicographie de Deca décrit l'ensemble des mots du langage Deca.
- Description complète : II-[Lexicographie]
- Chaînes de caractères
  - STRING_CAR est l'ensemble de tous les caractères, à l'exception des caractères '"', '\' et de la fin de ligne.
  - STRING = '"' (STRING_CAR + '\"' + '\\')* '"'
    NB : dans poly, caractère '\' doublé comme en ANTLR.
- Commentaires :
  - Un commentaire commence par '/*' et termine par '*/'. Le commentaire s'arrête au premier '*/' suivant le début du commentaire.
  - Un commentaire est une suite de caractères (autre qu'une fin de ligne) qui commence par '//' et s'étend jusqu'à la fin de la ligne ou du fichier.

Sommaire de cette section

2 Spécifications
- Lexicographie
- Syntaxe

Syntaxe

- La syntaxe concrète de Deca décrit l'ensemble des phrases (ou programmes) correctes du langage Deca.
- Description complète de la syntaxe : grammaire de Deca
  cf. II-[Syntaxe].
- Grammaire utilisant une syntaxe étendue proche des grammaires ANTLR[1] : grammaire hors-contexte classique, avec en plus
  - des répétitions : « * »,
  - des parties optionnelles : « ? ».

1. Style « EBNF » : Extended Backus-Naur Form

Syntaxe

- Règle particulière :
  - au lieu du très inefficace
    assign_expr
    → **or_expr**
    | **lvalue** '=' **assign_expr**
  - on utilise
    assign_expr
    → e=**or_expr** (
    { condition : expression e must be a "**lvalue**" }
    '=' **assign_expr**
    | ε)
  - e est l'expression correspondant à **or_expr**.
  - La condition sur e doit être vérifiée lors de l'analyse syntaxique (condition sur l'arbre construit pour la partie gauche).

Exercice

```
{
    int i = 1;
    println(i, " : ok");
}
```

Question

Le programme Deca est-il syntaxiquement correct ?

Correction

```
{
    int i = 1;
    println(i, " : ok");
}
```

- Oui, le programme est syntaxiquement correct.

Correction

```
{
    int i = 1;
    println(i, " : ok");
}
```

- Oui, le programme est syntaxiquement correct.
- prog → list_classes main EOF
  list_classes → (class_decl)*
  main → block
  block → '{' list_decl list_inst '}'
  list_decl → (decl_var_set)*
  decl_var_set → type list_decl_var ';'
  type → ident
  ident → IDENT
  list_decl_var → decl_var (',' decl_var)*
  decl_var → ident ('=' expr)?

Correction

```
{
    int i = 1;
    println(i, " : ok");
}
```

- expr → assign_expr
  assign_expr → or_expr
  or_expr → and_expr
  and_expr → eq_neq_expr
  ...
  primary_expr → literal
  literal → INT
  ...

Exercice

Question

Écrire un programme qui définit une classe Compteur, avec un champ x de type int, une méthode incr() et un programme principal qui fait fonctionner le compteur.

Correction

```
class Compteur {
    int x;
    void incr() {
        x = x + 1;
    }
}

{
    Compteur c = new Compteur();
    c.incr();
}
```

Sommaire

1 Introduction

2 Spécifications

3 Analyse lexicale et syntaxique avec ANTLR
- Arbre abstrait représentant un programme Deca
- Écriture du Parser ANTLR et construction de l'arbre

4 Notion d'« erreur »

Sommaire de cette section

3 Analyse lexicale et syntaxique avec ANTLR
- Structure d'un fichier source ANTLR
- Analyse lexicale
- Analyse syntaxique

Structure d'un fichier source ANTLR

- Analyseur lexical :
  lexer grammar *nomDeClasse*;
  - L'analyseur lexical généré est une classe de nom *nomDeClasse*;
- Analyseur syntaxique :
  parser grammar *nomDeClasse*;
  - L'analyseur syntaxique sera une classe de nom *nomDeClasse*;
- options {
  nom-option = *valeur*;
  nom-option2 = *valeur2*;
  ...
  }
  - language : le langage de programmation (défaut : Java)
  - superClass : le code généré est une classe qui hérite de cette classe
  - tokenVocab : le vocabulaire d'entrée à utiliser (le nom de l'analyseur lexical dans le cas d'un analyseur syntaxique)

Structure d'un fichier source ANTLR

- @header { ... }
  - Portion de code (Java) ajouté en tête du fichier généré
  - Exemple :
    @header {
      import *nom.du.paquetage*;
    }
- @members { ... }
  - Portion de code (Java) ajoutée dans la classe générée.
  - Peut être utilisé pour ajouter des champs et des méthodes dans la classe générée.
- La dernière partie du fichier est constituée de *règles* qui sont appliquées pour réaliser l'analyse lexicale ou syntaxique.

Sommaire de cette section

3 Analyse lexicale et syntaxique avec ANTLR
- Structure d'un fichier source ANTLR
- Analyse lexicale
- Analyse syntaxique

Règles pour l'analyse lexicale

- Une *règle* est de la forme :
  `NOM_REGLE : expression-régulière;`
- Le nom de la règle doit commencer par une majuscule.
- Exemple : `PLUS : '+';`
- Lorsque l'expression régulière est reconnue, un jeton correspondant au nom de la règle est renvoyé.
- Le jeton est un objet de type `org.antlr.runtime.Token`.
- `PLUS: [@6,13:13='+',<45>,1:13]`
  - 6 : 6ème jeton reconnu
  - 13:13 : la chaîne '+' commence au 13ème caractère du fichier source, et se termine sur ce même 13ème caractère.
  - <45> : code du jeton
  - 1 : numéro de ligne
  - 13 : numéro de colonne

Action associée à une règle

- Portion de code Java qui est effectuée avant de renvoyer le jeton.

  NOM_REGLE : *expression-régulière* { *action* } ;
  ESPACE : ' ' { System.out.println("espace reconnu"); };

- Fonction utile : `skip();`

  Permet de ne pas renvoyer le jeton correspondant
  ESPACE : ' ' { skip(); } ;

- Autre fonction utile : `getText();`

  Permet de récupérer le texte source correspondant

- Fragment de règle : règle qui peut être utilisée dans d'autres règles, et ne produisant pas de jeton (macros)
  fragment CHIFFRE : '0' .. '9';
  NOMBRE : CHIFFRE+;

Syntaxe des expressions régulières

`'c'`	le caractère *c*
`'\n'`	passage à la ligne (LF)
`'\r'`	retour chariot (CR)
`'\t'`	tabulation
`'\\'`	le caractère « \ »
`'\''`	le caractère « ' »
`'chaine'`	la chaîne de caractères
`.`	caractère quelconque (y compris une fin de ligne)
expr1 expr2	*expr1* suivie de *expr2*
'c1'..'c2'	caractères compris entre les caractères *c1* et *c2*
(expr)	expression *expr*
expr1 | expr2	expression *expr1* ou *expr2*
*expr**	expression *expr* répétée entre 0 et *n* fois
expr+	expression *expr* répétée entre 1 et *n* fois
~expr	tout caractère sauf *expr* (*expr* ne doit reconnaître que des caractères)

Résolution des ambiguïtés

Principe de la plus longue correspondance

- Pour chaque règle, l'analyseur lexical tente de reconnaître la chaîne la plus longue possible.
- Exemple

```
ELSE : 'else';
ELSEIF : 'elseif';
IF : 'if';
SPACE : ' ';
```

  - `else if` $\mapsto$ `ELSE SPACE IF`
  - `elseif` $\mapsto$ `ELSEIF` (et non `ELSE IF`)
- Si deux règles peuvent s'appliquer pour reconnaître deux chaînes de la même longueur : la première règle est prioritaire.
- Exemple :

```
DEFAULT : .;
```

  en fin de fichier pour reconnaître « tous les autres caractères ».

Traitement des commentaires Deca

- Si on définit les commentaires de la façon suivante :

  ```
  COMMENT : '/*' .* '*/' { skip(); } ;
  ```

  Si en entrée, on a « `/* foo */ bar */` », l'analyseur lexical va reconnaître « `/* foo */ bar */` », alors qu'il faudrait s'arrêter au premier « `*/` ».

- Une solution

  On utilise "`*?`" l'étoile « non gloutonne » en écrivant :

  ```
  COMMENT : '/*' .*? '*/'
      { skip(); } ;
  ```

Travail à réaliser pour l'analyse lexicale

- Pour l'analyse lexicale, travail à réaliser :
  - Compléter le fichier `src/main/antlr4/fr/ensimag/deca/syntax/DecaLexer.g4`
  - La classe `DecaLexer` hérite de la classe `AbstractDecaLexer`.
  - L'inclusion de fichier est traitée en analyse lexicale.
    Utiliser la méthode `doInclude` de `AbstractDecaLexer.java`.
- Utiliser le script `test_lex` pour faire des tests.

Sommaire de cette section

3 Analyse lexicale et syntaxique avec ANTLR
- Structure d'un fichier source ANTLR
- Analyse lexicale
- Analyse syntaxique
  - Arbre abstrait représentant un programme Deca
  - Écriture du Parser ANTLR et construction de l'arbre

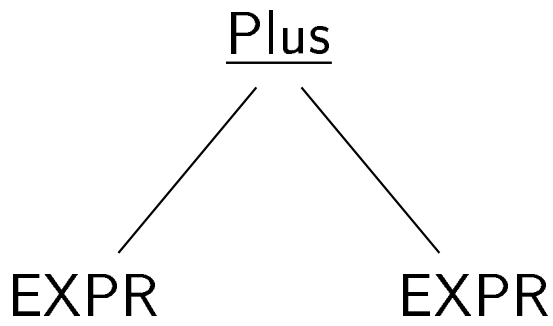
Arbre abstrait

- Arbre abstrait : représentation structurée du programme Deca
- Les constructions d'arbres sont décrites par une *grammaire d'arbres*,
  - définit la *syntaxique abstraite* du langage Deca.
- Référence : II-[SyntaxeAbstraite]

Grammaire d'arbres

- Vocabulaire terminal : nœuds de l'arbre
  Program, EmptyMain, Main, Identifier...
- Vocabulaire non terminal :
  PROGRAM, MAIN, DECL_VAR, IDENTIFIER...
- Règles :
  OP_ARITH → Plus [EXPR EXPR] ≡

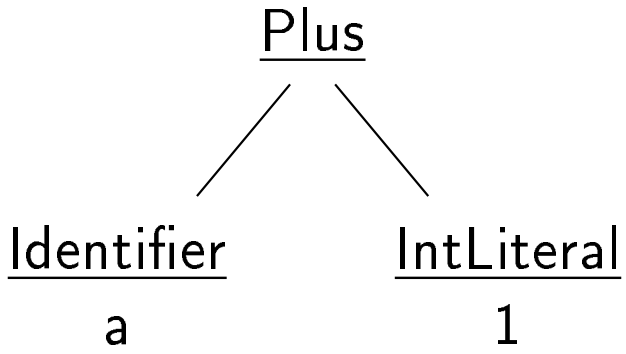
OP_ARITH →



IDENTIFIER → Identifier ↑ *Symbol*

Grammaire d'arbres

Par exemple, l'expression « $a + 1$ » correspond à l'arbre :



Codage des *listes d'arbres* avec une racine "anonyme" :

$$\begin{aligned} \text{PROGRAM} &\rightarrow \underline{\text{Program}}\ [\ \text{LIST\_DECL\_CLASS MAIN}\ ] \\ \text{LIST\_DECL\_CLASS} &\rightarrow [\ \text{DECL\_CLASS*}\ ] \end{aligned}$$

Exercices

Question

Construire l'arbre abstrait correspondant aux programmes suivants.

Exercices

- Programme 1 :

```
// rien !!!
```

- Programme 2 :

```
{}
```

- Programme 3 :

```
class A extends B
{ int x = 1; }
```

- Programme 4 :

```
void setX(int x) {
    this.x = x;
}
```

- Programme 5 :

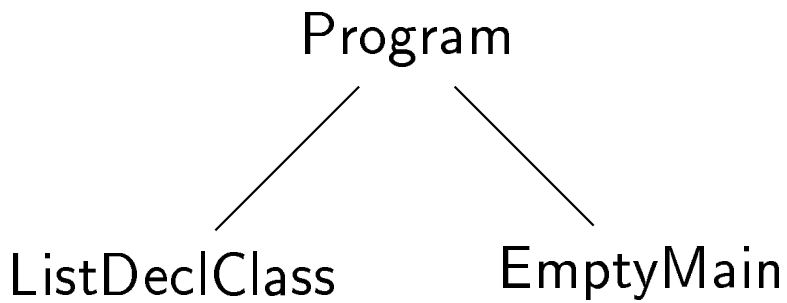
```
{
    A a = new A();
    println("a.getX() = ",
            a.getX());
}
```

- Programme 6 :

```
if (a == 1) {
    x = y;
} else if (a == 2) {
    x = t;
}
```

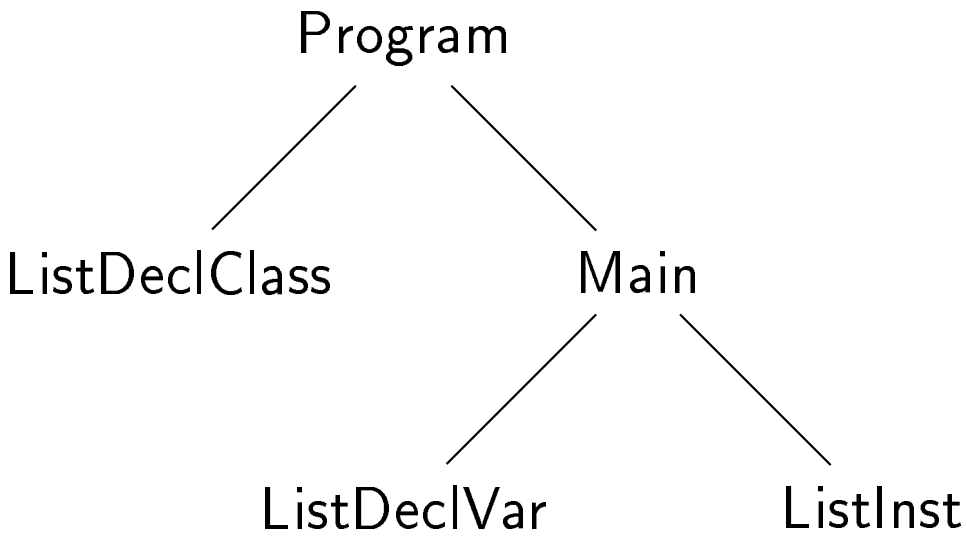
Programme 1

```
// rien !!!
```



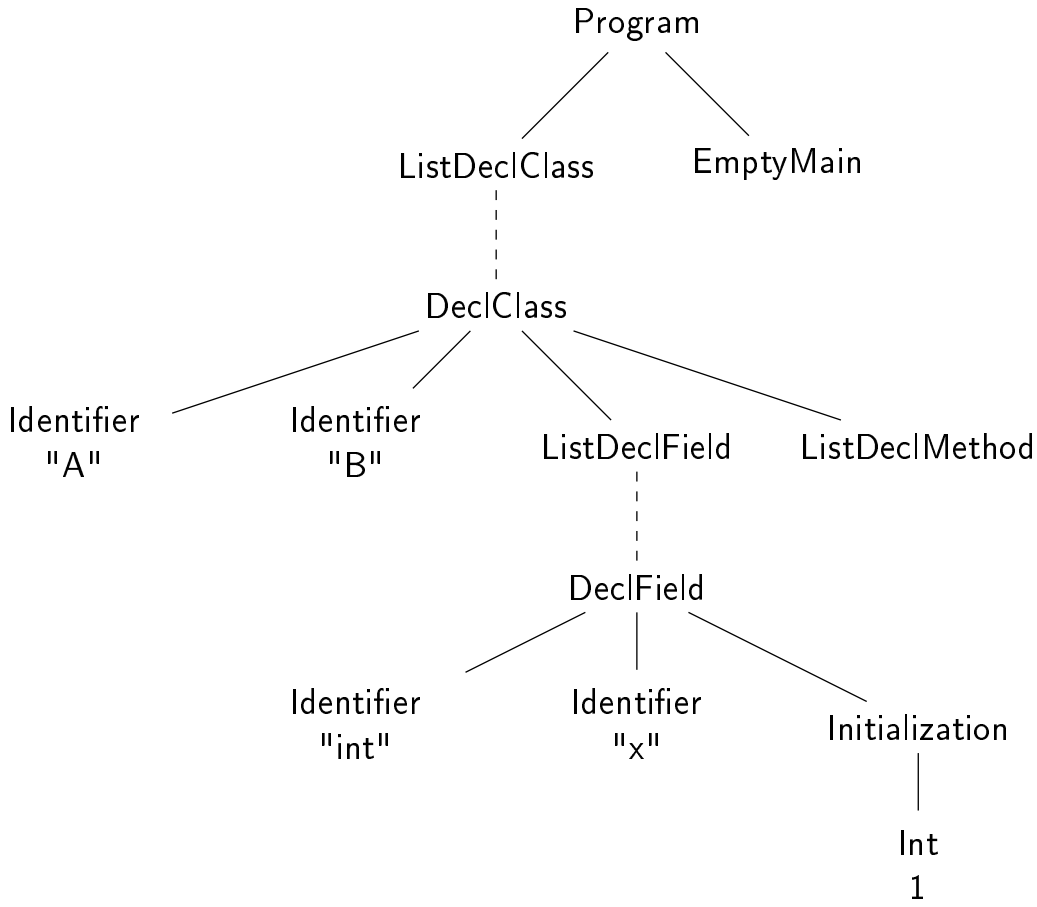
Programme 2

```
{}
```



Programme 3

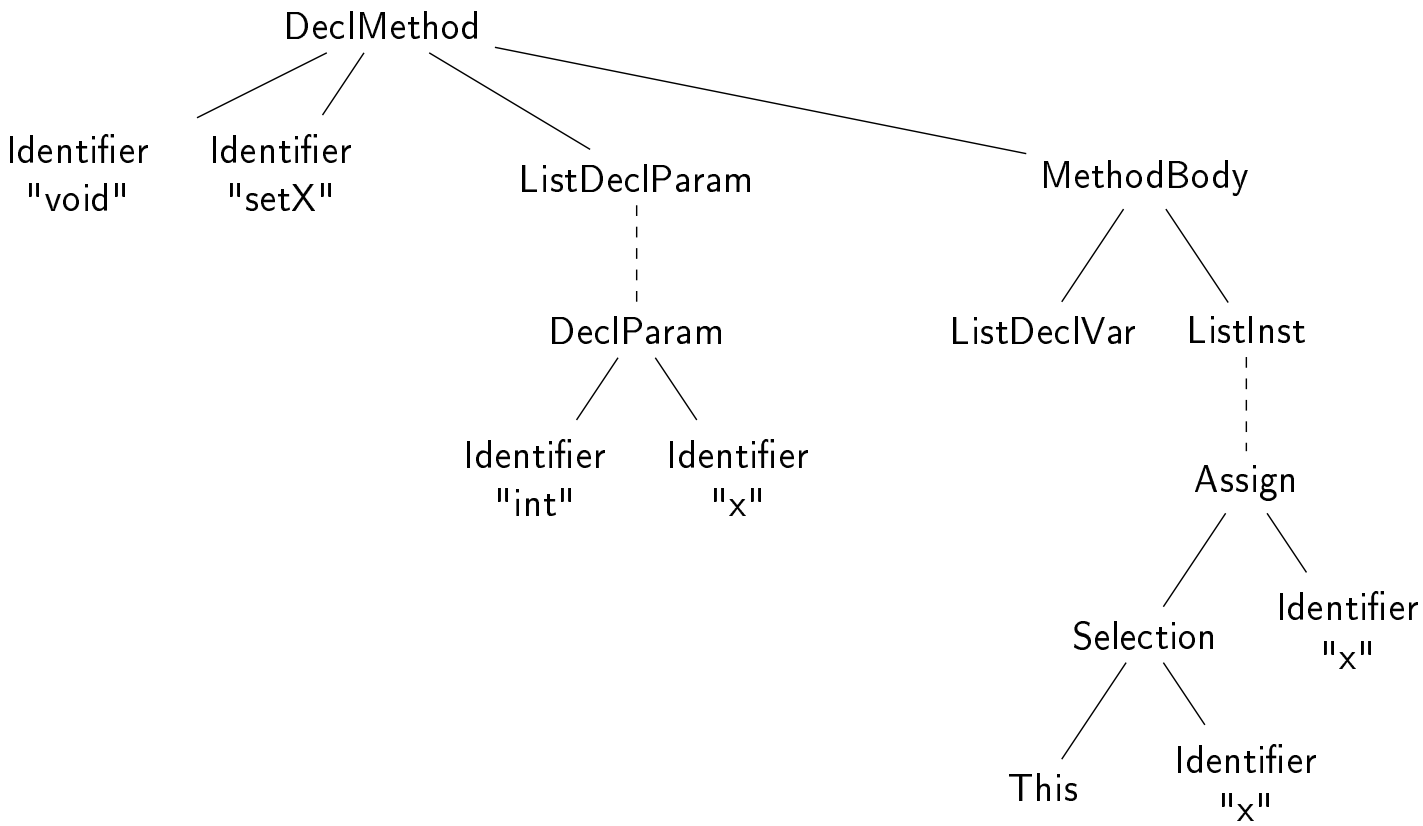
```
class A extends B {
  int x = 1;
}
```



Programme 4

```
void setX(int x) {
    this.x = x;
}
```

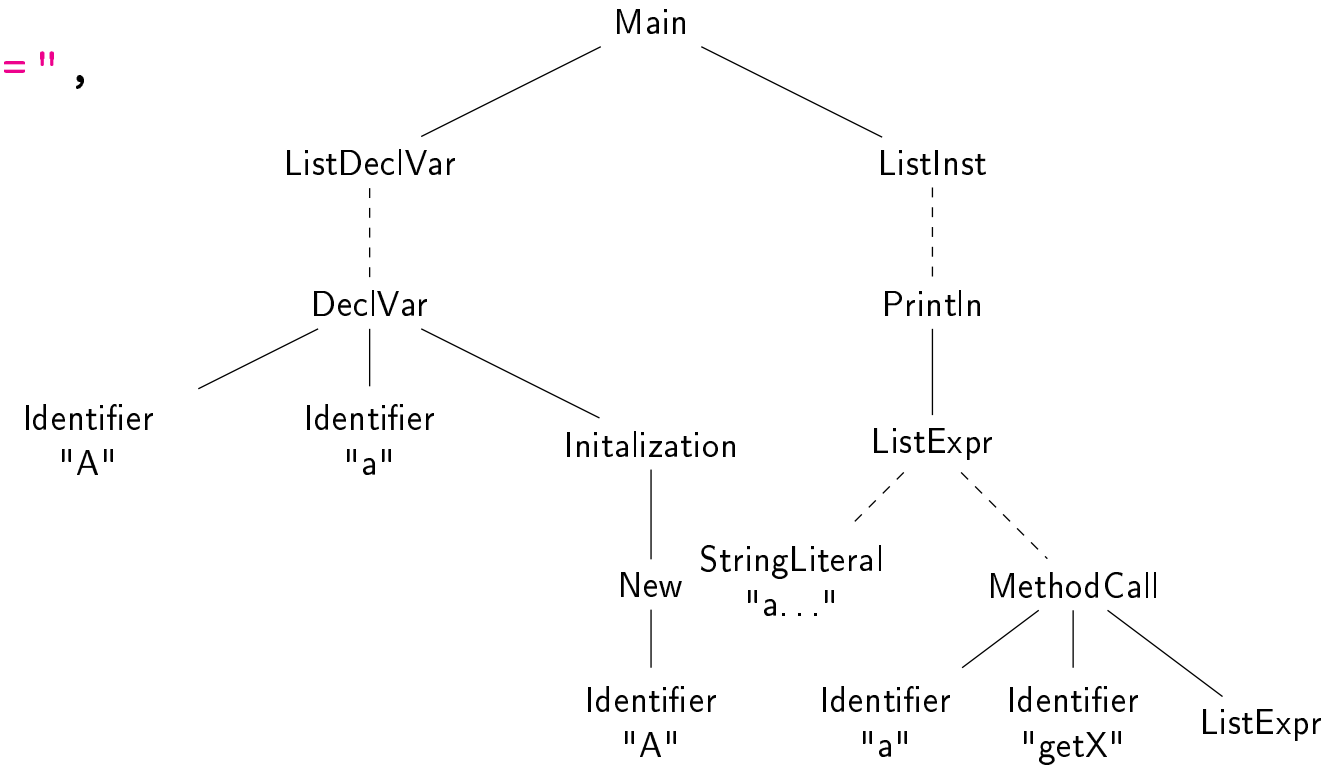
- Juste une déclaration d'une méthode.



Programme 5

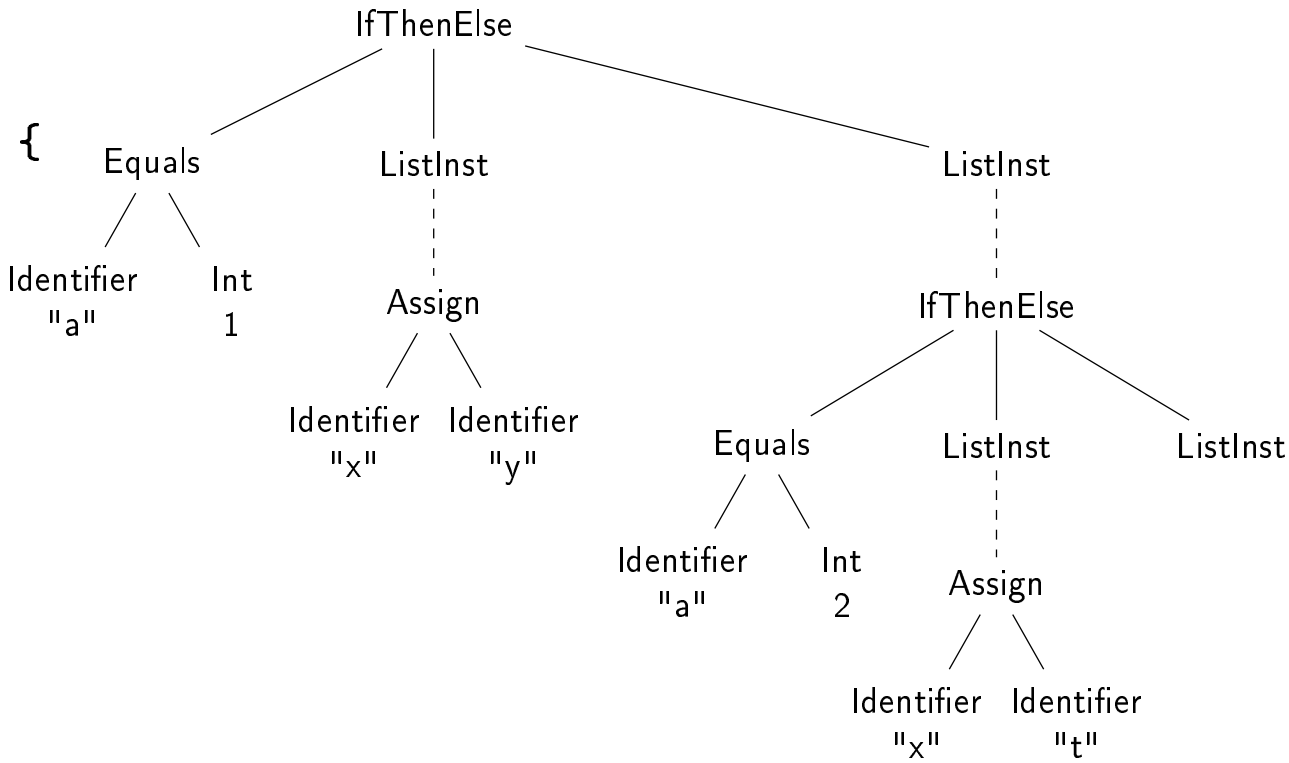
```
{
   A a = new A();
   println("a.getX()=",
       a.getX());
}
```

- Juste la partie principale



Programme 6

```
if (a == 1) {
    x = y;
} else if (a == 2) {
    x = t;
}
```



- Juste une instruction

Classes Java pour la grammaire d'arbres

- Classes définies dans `src/main/java/fr/ensimag/deca/tree/`.
- Patron de conception « *Interprète* » (cf. III-[ConventionsCodage])
- Non terminaux : classe abstraites Java

Non terminal	classe Java
PROGRAM	AbstractProgram
MAIN	AbstractMain
DECL_VAR	AbstractDeclVar
IDENTIFIER	AbstractIdentifier

- Terminaux : classes concrètes Java

terminal	classe Java
Program	Program
EmptyMain	EmptyMain
DeclVar	DeclVar
Identifier	Identifier

- Règle $X \to Y \quad \longmapsto \quad$ (abstract) class Y extends X
- Règle $X \to Y\,[A\ B] \quad \longmapsto \quad$ la classe Y (ou une classe de base) contient des champs de types A et B.

Classes Java pour la grammaire d'arbres

Exemple : les expressions

- Grammaire d'arbres :

  $\mathbf{EXPR} \rightarrow \mathbf{BINARY\_EXPR}$
  $\mathbf{BINARY\_EXPR} \rightarrow \mathbf{OP\_ARITH} \mid \ldots$
  $\mathbf{OP\_ARITH} \rightarrow \underline{\mathit{Plus}}[\mathbf{EXPR\ EXPR}] \mid \ldots$

- Classes :

```
abstract class AbstractExpr {...}

abstract class AbstractBinaryExpr extends AbstractExpr {
    AbstractExpr leftOperand;
    AbstractExpr rightOperand;
}

abstract class AbstractOpArith extends AbstractBinaryExpr {...}

class Plus extends AbstractOpArith {...}
```

Parcours d'arbre avec le patron interprète

Exemple : les expressions

```
abstract class AbstractBinaryExpr extends AbstractExpr {
    AbstractExpr leftOperand, rightOperand;
    public void decompile(IndentPrintStream s) {
        s.print("(");
        leftOperand.decompile(s);
        s.print(" " + getOperatorName() + " ");
        rightOperand.decompile(s);
        s.print(")");
    }
    abstract protected String getOperatorName();
}

class Plus extends AbstractOpArith {
    protected String getOperatorName() { return "+"; }
}
```

Factoriser le code « le plus haut possible »
dans la hiérarchie de classes

Classe abstraite `Tree`

- Toutes les classes pour les arbres sont définies dans le paquetage `fr.ensimag.deca.tree`.
- On a une classe abstraite `Tree` dont héritent toutes les autres classes.
- À chaque arbre est associé un objet de type `Location`, qui comporte
  - un nom de fichier (`fileName`),
  - un numéro de ligne (`line`),
  - un numéro de colonne (`positionInLine`).
- Permet de stocker la position d'un élément correspondant à un nœud de l'arbre dans le fichier source.
  (Utile pour les messages d'erreur).

Règles des grammaires ANTLR

- Exemple de règles

```
sum_expr : mult_expr (PLUS mult_expr)*;
mult_expr: INTEGER;
```

  - `mult_expr` dérive vers un entier ;
  - `sum_expr` dérive vers une suite de `mult_expr` séparées par des `+` ;
  - reconnaît par exemple `42`, `3+42`, `12+28+1`.

- Les non-terminaux commencent par une minuscule ; les terminaux (jetons produits par le lexer) sont en majuscule.

- Construction des règles
  - `(expr1 | expr2)` : `expr1` ou `expr2` ;
  - `(expr)*` : `expr`, répétée entre 0 et *n* fois ;
  - `(expr)?` : `expr`, 0 ou 1 fois.

Principe de l'analyse syntaxique

- `parser grammar DecaParser;`
  - ANTLR génère un analyseur syntaxique dans la classe `DecaParser`
- Pour chaque non-terminal de la grammaire, ANTLR génère une méthode dans la classe `DecaParser`.
  - Un appel à cette méthode reconnaît et consomme une séquence de jetons produits par l'analyse lexicale.
- Quand apparaît en partie droite de règle
  - un terminal : l'analyseur syntaxique vérifie que le jeton est correct et consomme ce jeton ;
  - un non-terminal : l'analyseur syntaxique effectue un appel récursif à la méthode correspondante.

Exemple 1

- Pour les règles

```
nombre_negatif : MINUS nombre;
nombre : INT;
```

ANTLR produit un code qui ressemble à

```
public final ... nombre_negatif()
                 throws RecognitionException {
    ...
    // NomDuFichier.g4:24:16: ( MINUS nombre )
    match(MINUS);
    nombre();
    ...
}

public final ... nombre()
                 throws RecognitionException { ... }
```

- L'exception RecognitionException est levée si une erreur de syntaxe est détectée.

Exemple 2

- Pour une règle de la forme `expr : expr1 | expr2;`
  ANTLR produit un code qui ressemble à

```
public final ... expr()
                    throws RecognitionException {
    switch (choisir_branche) {
    case 1:
        expr1(); break;
    case 2:
        expr2(); break;
    }
}
```

Actions associées à une règle

- Action associée à une règle : portion de code Java entre accolades
  - Ce code est inséré dans le corps des méthodes d'analyse
  - Exemple :

```
prog : debut { System.out.println("j'ai vu debut"); }
       suite { System.out.println("j'ai vu suite"); }
     ;
```

  - Après la reconnaissance de `debut`, affiche `"j'ai vu debut"` ; après la reconnaissance de `suite`, affiche `"j'ai vu suite"`.

Résultat de la méthode associée à un non-terminal

- La méthode associée au non-terminal `toto` retourne un objet de type `TotoContext`
- On peut ajouter des attributs synthétisés qui seront des champs de cet objet : *non_terminal* `returns[`*type1 nom1, type2 nom2*`]`
- Exemple

```
expr returns[int val] :
    sum_expr { $val = $sum_expr.val; }
    ;
sum_expr returns[int val] :
    e=mult_expr {$val = $e.val;}
    (PLUS e2=mult_expr {$val = $val + $e2.val;})*
    ;
mult_expr returns[int val] :
    INTEGER {$val = Integer.parseInt($INTEGER.text);}
    ;
```

- `$INTEGER.text` représente la chaîne de caractères qui correspond à l'entier reconnu.

Paramètres de la méthode associée à un non-terminal

- Exemple

```
sum_expr returns[int val] :
   mult_expr {$val = $mult_expr.val;}
   (plus_mult_expr[$val] {$val = $plus_mult_expr.after;})*
   ;

plus_mult_expr[int before] returns[int after] :
   PLUS mult_expr { $after = $before + $mult_expr.val; }
   ;
```

- Remarque

  L'objet en retour contient un uplet des *attributs synthétisés* ; les paramètres correspondent à des *attributs hérités* d'une *grammaire attribuée*.

  `plus_mult_expr[int before] returns[int after]`

  $\equiv$ $plus\_mult\_expr \downarrow before \uparrow after$

Section @init

- Déclaration de variables locales / initialisations
- Exemple

```
expr returns[int val]
@init {
   int i;
   $val = 0;
}
   : expr1 { i = 42; $val = $val + 1 }
   | expr2 { i = 43; $val = $val + 2 }
```

- Les déclarations/initialisations s'appliquent à toute la méthode `int expr()`.

Gestion des erreurs

- En cas d'erreur de syntaxe : l'analyseur lève l'exception `RecognitionException`.
- On peut également lever cette exception explicitement dans une règle.
- Par défaut, chaque méthode d'analyse générée rattrape l'exception avec la construction :

```
try {
    // corps de la regle
} catch (RecognitionException re) {
    _errHandler.reportError(this, re);
    _errHandler.recover(this, re);
}
```

Gestion des erreurs dans le cadre du projet

- Dans le cadre du projet, on s'arrête à la première erreur détectée (étapes B et C). En étape A, on laisse le rattrapage d'erreur d'ANTLR travailler sur les cas triviaux.

- Pour l'analyse syntaxique, pour s'arrêter à la première erreur de syntaxe non-triviale, on initialise l'objet `_errHandler` à `new DefaultErrorStrategy()` (dans le constructeur `AbstractDecaParser`).

Construction de l'arbre

- Au cours de l'analyse syntaxique, on construit l'arbre abstrait du programme Deca source.
- Fichier source : `DecaParser.g4`, qui génère `DecaParser.java`.
- La classe `DecaParser` étend la classe abstraite `AbstractDecaParser`.

Règles de DecaParser.g4

```
prog returns[AbstractProgram tree]
    : list_classes main EOF {
        // Verification des attributs
        // (pas indispensable, mais aide au debug)
        assert($list_classes.tree != null);
        assert($main.tree != null);

        // Construction de l'arbre
        $tree = new Program($list_classes.tree,
                            $main.tree);

        // Initialisation du champ 'location'
        // de l'arbre $tree a partir du jeton $main
        setLocation($tree, $main.start);
      }
    ;
```

Règles de DecaParser.g4

- ```
  prog returns[AbstractProgram tree]
      : list_classes main EOF { ... $tree = ...; };
  ```

- ```
  main returns[AbstractMain tree]
      : /* epsilon */ {
          $tree = new EmptyMain();
        }
      | block {
          assert($block.decls != null);
          assert($block.insts != null);
          $tree = new Main($block.decls, $block.insts);
          setLocation($tree, $block.start);
        }
      ;
  ```

Règles de DecaParser.g4

- ```
  block returns[ListDeclVar decls, ListInst insts]
      : OBRACE list_decl list_inst CBRACE {
          assert($list_decl.tree != null);
          assert($list_inst.tree != null);
          $decls = $list_decl.tree;
          $insts = $list_inst.tree;
        }
      ;
  ```

- ```
  list_decl returns[ListDeclVar tree]
          // tree est ici une liste d'arbres
  @init {
      $tree = new ListDeclVar();
  }
      : decl_var_set[$tree]*
      ;
  ```

⋮

Travail à réaliser pour l'analyse syntaxique

cf. I-[Consignes]

- Pour l'analyse syntaxique, travail à réaliser :
  - Compléter le fichier `src/main/antlr4/fr/ensimag/deca/syntax/DecaParser.g4`
  - Compléter le paquetage des constructeurs de l'arbre `src/main/java/fr/ensimag/deca/tree/*.java`
  - Utiliser le script `test_synt` pour faire des tests.

Sommaire

1 Introduction

2 Spécifications

3 Analyse lexicale et syntaxique avec ANTLR
- Arbre abstrait représentant un programme Deca
- Écriture du Parser ANTLR et construction de l'arbre

4 Notion d'« erreur »

Notion d'« erreur » dans le projet

On distingue plusieurs sortes d'erreurs :

- Erreurs du programme Deca
  - Programme Deca incorrect, doit être rejeté.
- Limitations du compilateur : programme Deca peut-être correct, mais le compilateur ne peut pas le compiler
  - limitation du langage ou des outils utilisés,
  - portion du langage non implémentée.
- Avertissements (« warnings ») : programme correct, mais le compilateur a détecté un problème.
  - Le compilateur DOIT générer du code.
  - Il PEUT afficher un message *UNIQUEMENT si l'option -w est utilisée*.
- Erreurs internes :
  - Le compilateur a détecté une erreur dans lui-même
  - (pas une erreur du programme Deca).

Erreurs du programme Deca

- Chaque étape du compilateur, ainsi que la ligne de commande, peut provoquer une erreur.
- On s'arrête à la première erreur détectée, on ne fait pas de récupération d'erreur (choix imposé).
- Programmation : utilisation d'exceptions Java
- Lorsqu'une erreur est détectée, un message d'erreur est affiché.
- Format des messages d'erreur (imposé : à RESPECTER STRICTEMENT)

  `<nom-fichier>:<no-ligne>:<no-colonne>: <description>`

  Exemple : `fichier.deca:12:4: caractère '#' non autorisé`
  - pas d'espace entre le nom du fichier et ':',
  - pas d'espace autour des numéros de ligne et de colonne.

Hiérarchie d'exceptions fournies

Exception
- ⟵ RuntimeException
  - ⟵ RecognitionException (erreurs étape A)
    - ⟵ IncludeFileNotFound
    - ⟵ InvalidLValue
    - ⟵ CircularInclude
  - ⟵ DecacInternalError (erreurs internes)
- ⟵ LocationException
  - ⟵ Contextual error (erreurs étape B)
- ⟵ CLIException (erreur ligne de commande)
- ⟵ DecacFatalError (erreur de lecture du fichier source ou d'écriture du fichier cible)

- D'autres exceptions peuvent être définies.