Théorie des langages

Analyse lexicale et syntaxique avec JavaCC et JJTree[1]

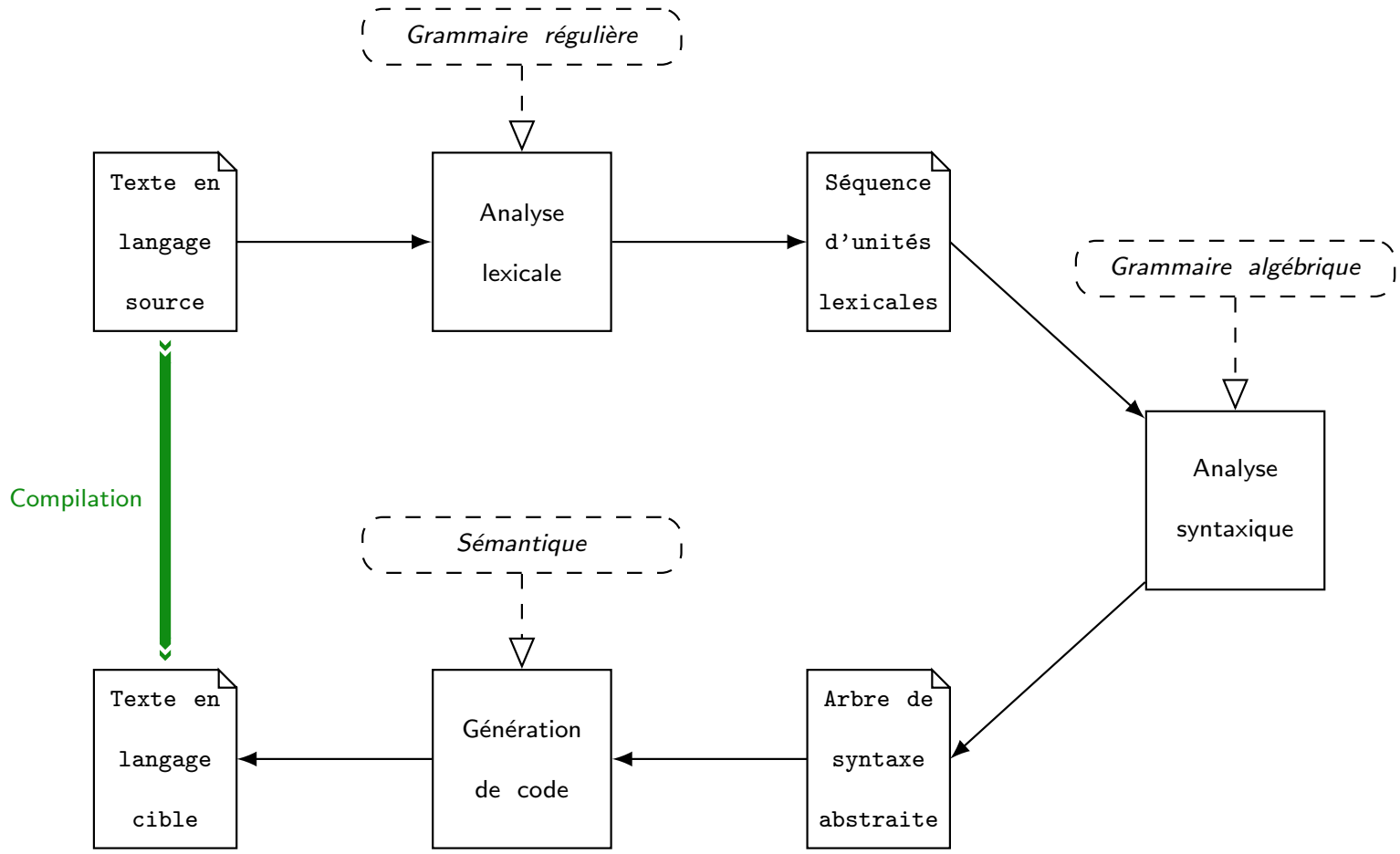
Jérôme Voinot
jerome.voinot@lifc.univ-fcomte.fr
http://lifc.univ-fcomte.fr/~voinot

Licence informatique 3ème année
Octobre 2007

Laboratoire d'Informatique de l'Université de Franche-Comté

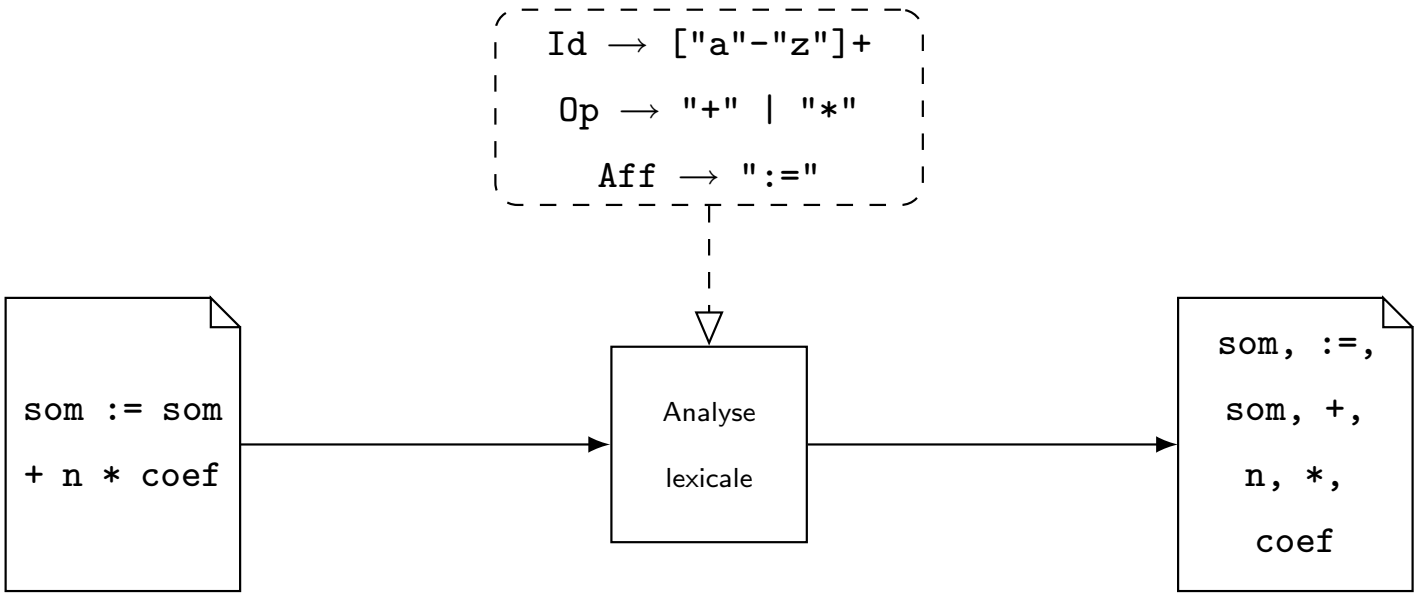
[1] Réalisé à partir de documents de A. Giorgetti

Principe de la compilation



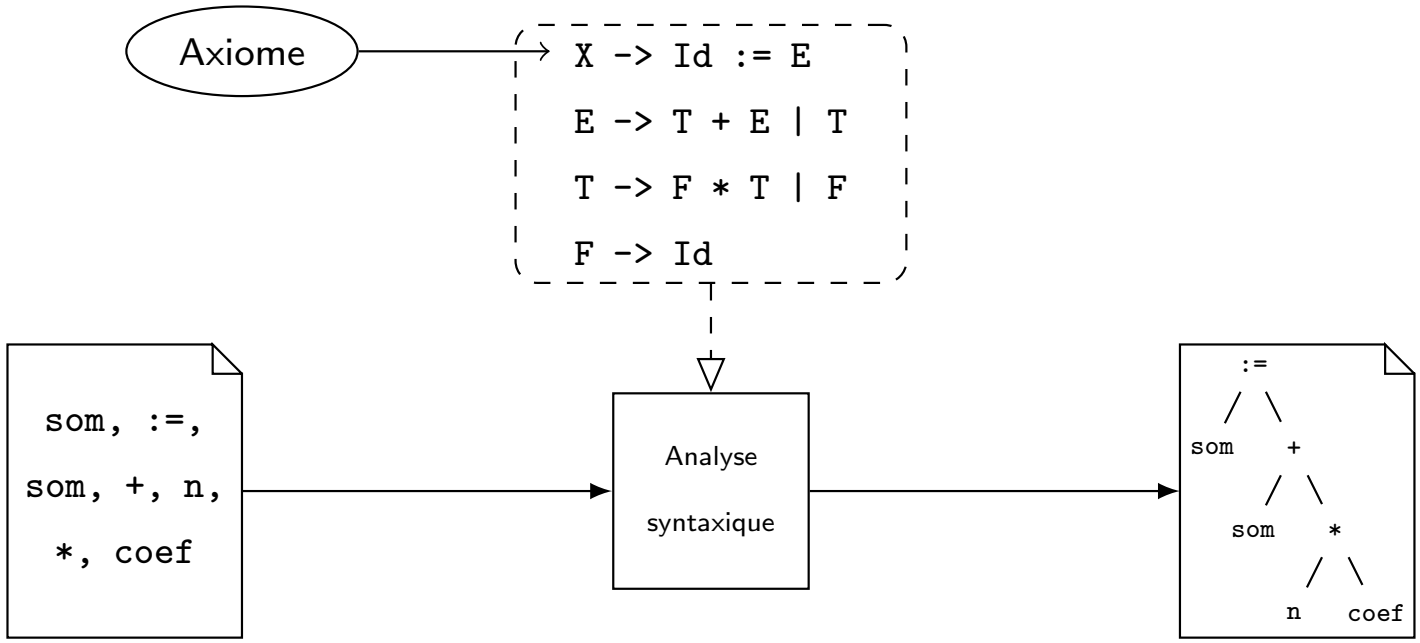
Analyse lexicale

Exemple



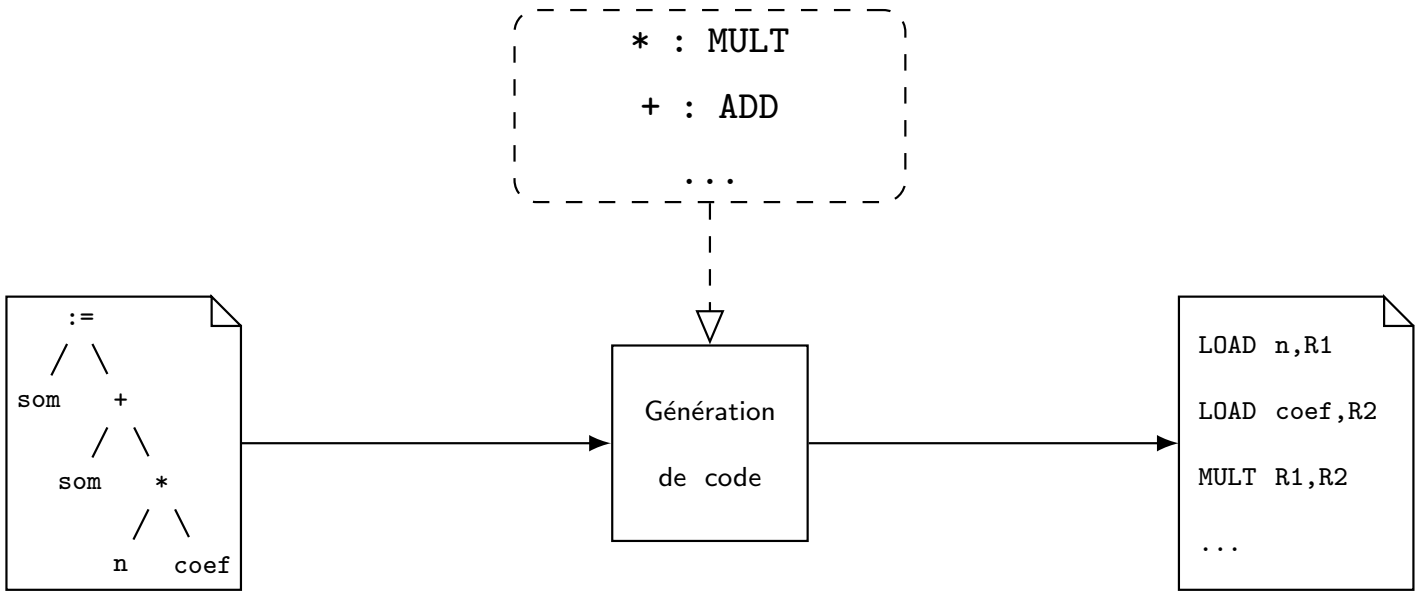
Analyse syntaxique

Exemple



Génération de code

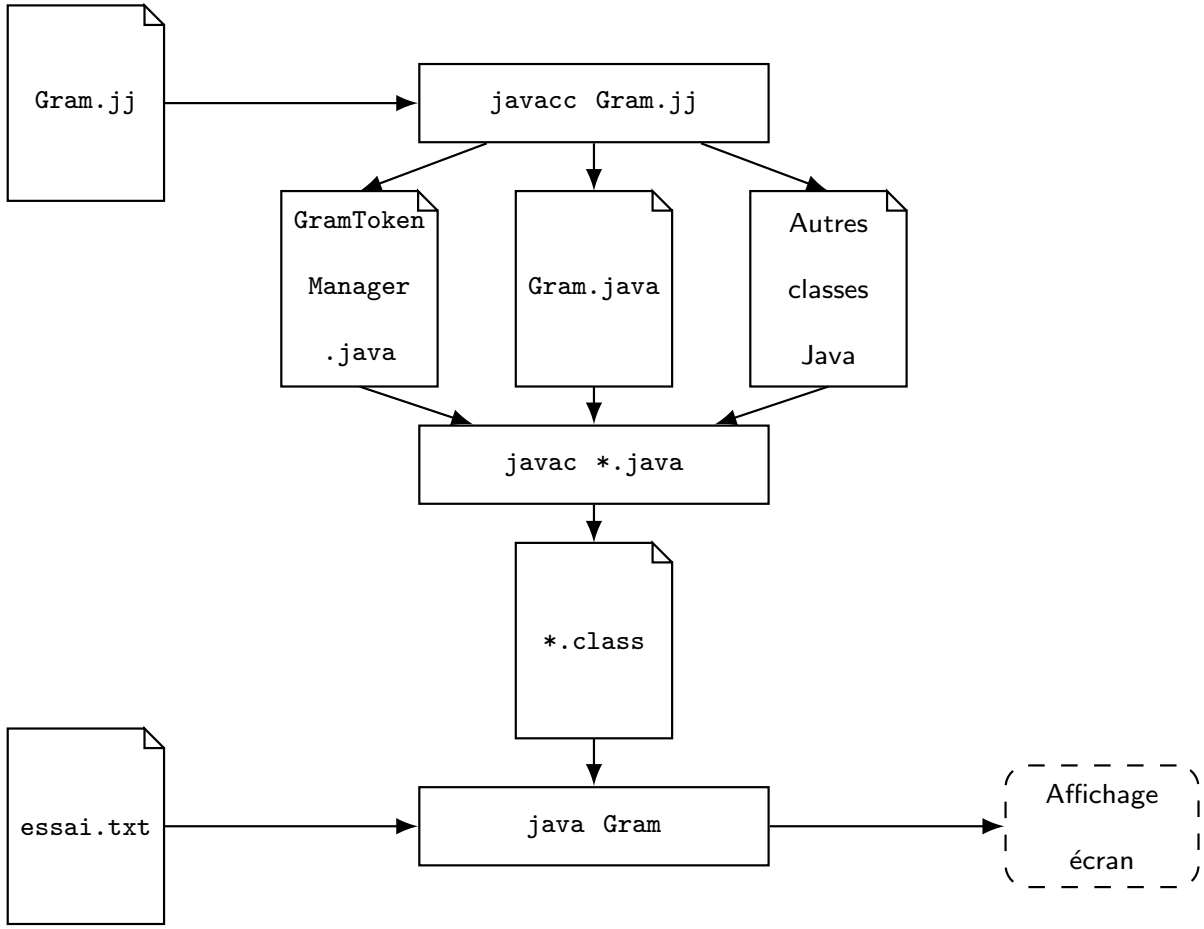
Exemple



JavaCC

- JavaCC signifie *Java Compiler Compiler*
- Générateur d'analyseur lexicaux et syntaxiques en Java
- Analyseur lexical (*lexer*, *scanner*) :
  - Découpe le code en unités lexicales (*token*)
  - Ignore le code non significatif (espaces, . . .)
- Analyseur syntaxique (*parser*) :
  - Vérifie que la structure du code est conforme aux règles d'une grammaire hors-contexte
  - Construit des représentations internes du code sous formes d'arbres syntaxiques
- Préprocesseur JJTree inclus dans JavaCC

Mise en œuvre de JavaCC



Structure du fichier `Gram.jj`

Copié dans `Gram.java`

```
PARSER_BEGIN(Gram)
public class Gram {
    public static void main (String arg[]) throws ParseException {
        Gram p = new Gram(System.in) ;
        p.axiome() ;
    }
}
PARSER_END(Gram)
```

Liste de règles lexicales et syntaxiques

```
TOKEN : {
    <SALUT : "Bonjour" | "Hello">
}

void axiome() : {
}{
<SALUT> unNonTerminal() ("\n"|"\r")* <EOF>
}
void unNonTerminal() : {
}{ ...
}
```

Analyse lexicale

- Objectif : découper le code en lexème (unité lexicale, *token*)

```
PROGRAM calcul ;
VAR i  : INTEGER ;
BEGIN
    i  := 1 ;
END
```

- Sortes de lexèmes :
  - Mots réservés du langage
  - Identificateurs choisis par le programmeur
- Non-lexèmes (*whitespaces*)
  - Espaces
  - Retours à la ligne
  - Tabulations
  - ...

Analyse lexicale avec JavaCC

- Déclarer des unités lexicales

```
TOKEN : {
    <"BEGIN"> | <";"> | <":=">
}
```

Méta-caractères (<, >, ", |, ...) utilisables comme lexèmes si entourer avec "

- Nommer un ensemble de lexèmes

```
TOKEN : {
    <VARID: ["a"-"z"] ["a"-"z"]*>
}
```

Non-terminaux lexicaux notés en majuscules par convention

Analyse lexicale avec JavaCC

- Associer des actions lexicales

```
TOKEN : {
    <DEBUT: "bEgIN"> {
        System.out.println(matchedToken.image.toUpperCase()) ;
    }
}
```

Affichage, remplacement, suppression, ...

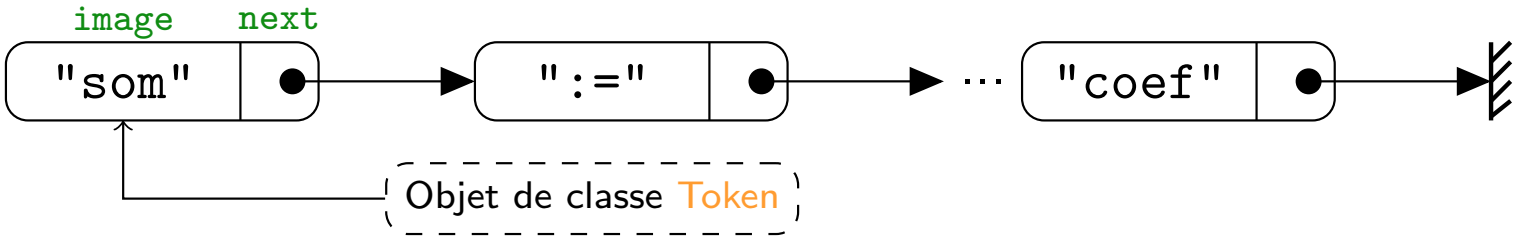
- Utiliser des expressions régulières

```
TOKEN : {
    <ID: ["a"-"z"] (["a"-"z"] | ["0"-"9"])*>
}
```

Langages réguliers (éventuellement infinis) décrit à l'aide des méta-caractères `[`, `-`, `]`, `|`, `+` et `*`

Effets de l'analyse lexicale avec JavaCC

- Construction d'une liste chaînée de lexèmes



- Objet qui stocke le lexème courant
  - `matchedToken` dans `GramTokenManager.java`
  - `token` dans `Gram.java`
- Classe `Token` décrite dans la documentation des classes JavaCC
  `https://javacc.dev.java.net/doc/apiroutines.html`

Lien avec l'analyse syntaxique

```
PARSER_BEGIN(Lexer)
public class Lexer {
    public static void main (String arg[]) throws ParseException {
        (new Lexer(System.in)).axiome() ;
    }
}
PARSER_END(Lexer)
```

Liste de règles lexicales

```
TOKEN : {
    <ID : ...> // non-terminal lexical
    |
    <AFF : ":=">
}
```

Liste de règles syntaxiques avec non-terminaux lexicaux

```
/* non-terminal syntaxique */
void axiome() : {
}{
<ID> <AFF> (<ID>,)*<ID>
}
```

Sortes de lexèmes

- 4 sortes de règles lexicales
  - `SKIP` : ignore la chaîne reconnue
    Exécute une action lexicale sans construire d'objet `Token`
  - `TOKEN` : chaîne reconnue comme un lexème
    Retourne cette chaîne à un analyseur syntaxique (ou à un autre extérieur) par le champ `image` de la classe `Token`
  - `MORE` : continue la reconnaissance
    La chaîne reconnue préfixe le résultat complet
  - `SPECIAL_TOKEN` : unité lexicale non retournée
    Essentiellement pour gérer les commentaires
- Pour plus de détails, voir le tutoriel *TokenManager*
  `https://javacc.dev.java.net/doc/tokenmanager.html`

Cas de SKIP et SPECIAL_TOKEN

- Pour les caractères “blancs”

```
SKIP : {
    " " | "\t" | "\n" | "\r"
}
```

Caractères non stockés

- Pour les commentaires

```
SPECIAL_TOKEN : {
    <LINE_COMM: "//" (~["\n","\r"])* (\n" | "\r" | "\r\n")>
}
```

Commentaires stockés dans le champ `specialToken` du lexème suivant

Principes de reconnaisance des unités lexicales

- La chaîne reconnue est la plus longue chaîne conforme à une expression régulière
- Si plusieurs expressions sont conformes, de même longueur maximale, seule la première dans l'ordre du fichier est appliquée
- La lecture de la fin de fichier provoque la création du lexème `EOF`

Notion d'état lexical

- Permet de simuler un automate
  - Associer un état à une règle lexicale
  - Indiquer l'état suivant en fin de règle

```
<S1> TOKEN : {
    ...
     : <S2>
}
```

  - Si aucun état n'est précisé, l'état lexicale suivant est le même
- Cas particuliers :
  - Pour désigner l'état initial, utiliser le mot réservé <DEFAULT>
  - Pour désigner une règle lexicale applicable dans tout état, utiliser <*>

Recapitulatif

- Contenu d'une règle lexicale
  - Au minimum des expressions régulières
  - Eventuellement :
    - Des non-terminaux lexicaux
    - Des actions à exécuter
    - L'état lexical suivant
- Pour aller plus loin sur ce sujet :
  - Pour la syntaxe des expression régulières, voir le cours "Détails d'analyse lexicale avec JavaCC"
  - Pour le reste, consulter le tutoriel *TokenManager*
    `https://javacc.dev.java.net/doc/tokenmanager.html`

Syntaxe d'une règle syntaxique

- En JavaCC, la règle $I \rightarrow V := E \mid B$ s'écrit :

```
void Instruction()  : {
}{
    <ID> <AFF> Expression()
  | Bloc()
}
```

- Différences :
  - `:` au lieu de $\rightarrow$
  - Indentation, non terminaux plus explicites
  - Blocs de code Java en plus
- Relations avec d'autres règles :
  - Analyse lexicale pour `<ID>` et `<AFF>`
  - Règles syntaxiques pour `Bloc()` et `Expression()`

Syntaxe d'une règle syntaxique

- Structure générale, format EBNF

```
String NTnom(...) : {
    String s1 = null;
}{
  (   s1 = NTautreNom(...) <VIRGULE>
      {
        String s2 = new String(s1);
        s2.toUpperCase();
      }
      s1 = NTnom2(...)              { ... }
    | s1 = NTnom3()
  )
  <EOF> { return(s1); }
}
```

Principe de l'analyse syntaxique avec JavaCC

- Analyse descendante LL(k)
  - Analyse = cherche si une grammaire engendre une séquence d'unités lexicales donnée
  - Descendante = part de l'axiome (L : *left to right*) et dérive toujours le non-terminal le plus à gauche (L : *left most*)
  - Condition : pas de récursivité gauche ($E \rightarrow E + T$)
- Conflits entre les règles (*choice points*)

$$\{X \rightarrow \text{“}b\text{”}\ \ \text{“}a\text{”} \mid Y, \quad Y \rightarrow \text{“}b\text{”}\ \ \text{“}c\text{”}\}$$

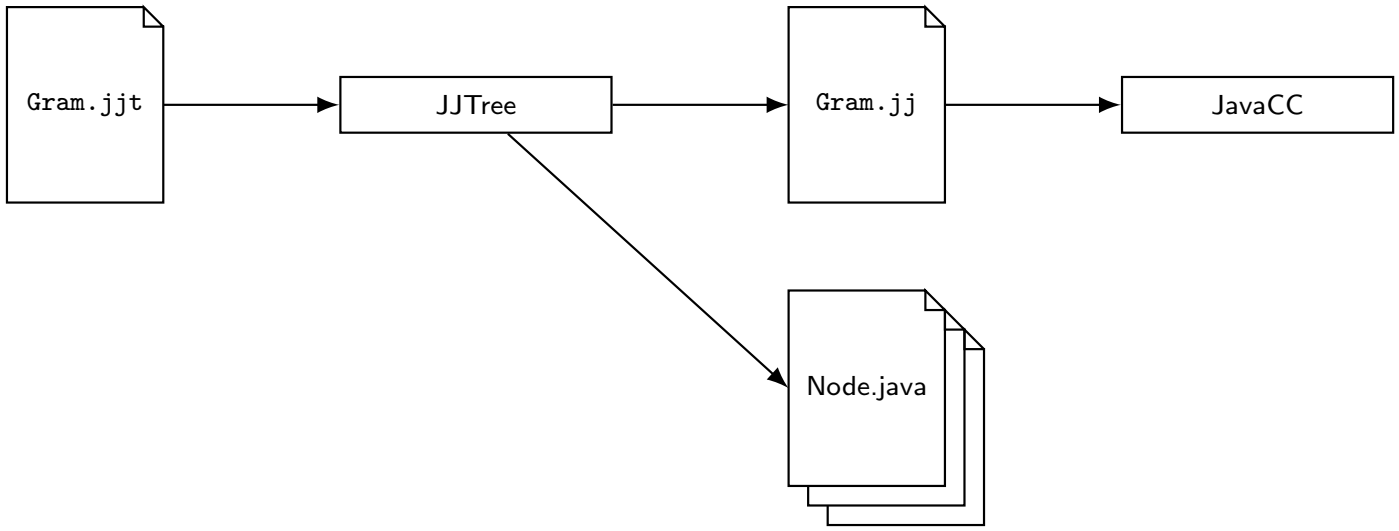
  - Levés en lisant k lexèmes à l'avance (*lookahead*, par défaut k = 1)
  - Coût lié à k seulement (augmenter k localement seulement)
  - Pas toujours possible $\Longrightarrow$ changer de grammaire

Construction d'arbres syntaxiques

- Arbre d'analyse syntaxique : un noeud par règle syntaxique concrète
- Arbre de syntaxe abstraite : nœuds selon la grammaire abstraite et les besoins de génération de code
- Origine des différences
  - Grammaire abstraite éventuellement ambiguë, pas LL(*)
  - Grammaire concrète LL(k), avec k minimal
    - plus longue, moins facile à lire
    - savoir améliorer la grammaire abstraite
    - élimination des récursions gauches
    - pratiquement toujours possible (↑ lookhead)

Arbres syntaxiques avec JavaCC

- Préprocesseur JJTree



- Plusieurs classes de noeuds (+ héritage) si l'option JJTree `MULTI` a pour valeur `true`

Annotations JJTree (#)

- Exemple

```
void Instruction() #void : {
}{
    Id() <AFF> Expression() #InsNode(2)
  | Bloc() #BlocNode(1)
}
```

- Effets :
  - Pas de noeud associé à `Instruction()`
  - Noeud binaire de classe `ASTInsNode`
  - Noeud unaire de classe `ASTBlocNode`
- Pour en savoir plus, voir TP2 et lire la documentation de JJTree `https://javacc.dev.java.net/doc/JJTree.html`

Génération de code

- Deux méthodes possibles
  - Distribuer une méthode `generer()` dans toutes les classes de noeuds (+ héritage)
  - Utiliser le modèle de conception `Visitor`
- Intérêts du modèle de conception
  - Plusieurs traitements différents (vérification de typage, estimation de la mémoire requise, ...) sans avoir à éditer toutes les classes de noeuds
  - Prévu dans JJTree (option `VISITOR=true`)
- Pour en savoir plus voir TP3 et suivants

Téléchargement de JavaCC

- Disponible sur le site `https://javacc.dev.java.net`
  - Pour Linux, télécharger le fichier `javacc-4.0.tar.gz`
  - Pour Windows, télécharger le fichier `javacc-4.0.zip`
- Installation
  - Disposer d'un environnement d'exécution Java 1.2 ou plus
  - Extraire dans un dossier (par exemple `C:\javacc`) et ajouter le sous-dossier `bin` à la variable système `PATH`
- L'accès à l'outil se fait en ligne de commande (`javacc` *`NomFichier`*`.jj`)