Université de Nice - Sophia Antipolis

Licence 3 Informatique

Techniques de compilation

Introduction à *JavaCC*

Jacques Farré

e-mail : Jacques.Farre@unice.fr

Table des matières

1 *JavaCC* : le générateur d'analyseur lexico-syntaxique en Java

On le trouve sur `http://www.webgain.com/products/metamata/java_doc.html`
Ces notes sont largement inspirées de la documentation originale (voir `~jf/Compil/Cours/02-Lexico-Syntaxique/JavaCC/javaccdocs/*`).

JavaCC (initialement *Jack*) est inspiré de *PCCTS*, qui a lui même donné *ANTLR* (`http://www.parr-research.com/antlr`).

Un fichier de description pour *JavaCC* est, par convention, suffixé par `.jj`. *JavaCC* est appelé par la commande :

```
javacc options fichier
```

Le résultat est un ensemble de classes *Java*, qui doivent être compilées par `javac`. En général, l'analyseur est la classe `MonAnalyseur` qui contient `main`, et il suffit de faire `javac MonAnalyseur.java` pour compiler l'ensemble des classes.

Pour un analyseur nommé *MonAnalyseur*, les `.java` qui sont produits sont

- `MonAnalyseur.java`, l'analyseur syntaxique
- `MonAnalyseurTokenManager.java`, l'analyseur lexical
- `MonAnalyseurConstants.java`, les constantes partagées par les deux analyseurs (analogue au `y.tab.c` de *YACC*)

plus des classes indépendantes de la grammaire : `Token.java`, `ParseError.java` ...

JavaCC : le générateur d'analyseur lexico-syntaxique en _Java_

- permet décrire un analyseur syntaxique récursif descendant; la grammaire – LL (1) – est décrite en EBNF, et inclut la définition de l'analyseur lexical
- permet d'indiquer comment traiter les aspects non LL (1) de la grammaire
- permet d'indiquer des actions sémantiques en *Java*
- gère des attributs hérités et synthétisés (grammaire attribuée à gauche – LAG)
- fournit des outils utiles (documentation de la grammaire, construction de l'arbre syntaxique, couverture de la grammaire, options de mise au point)

Slide 1

1.1 Définition d'une grammaire

L'***identificateur*** ci-contre est celui de l'analyseur généré, et il doit y avoir une classe de même nom :

```
PARSER_BEGIN (Exemple)
...
class Exemple {
  ...
}
...
PARSER_END (Exemple)
```

L'analyseur engendré par les productions sera formé d'un ensemble de méthodes publiques de la class `Exemple` (une méthode par non terminal).

Il n'y a aucune restriction à ce qui peut être mis dans l'unité de compilation : tout ce qui peut apparaître dans un fichier `.java` est accepté, mais *JavaCC* n'effectue aucune analyse de cette partie, donc les erreurs seront détectées lors de la compilation des classes engendrées.

Les options seront vues plus loin.

Dans les définitions qui suivent, les conventions suivantes sont adoptées :

- en ***italique gras***, ce qui est du *Java*
- en `courrier`, les symboles qui doivent apparaître tels quels dans la grammaire
- en fonte normale, les non terminaux

Forme d'une définition de grammaire pour *JavaCC*

- grammaire-*JavaCC* =

  options
  `PARSER_BEGIN (` ***identificateur*** `)`
  unité-de-compilation (une ou plusieurs classes)
  `PARSER_END (` ***identificateur*** `)`
  (production)*

- mots réservés

`EOF`	`IGNORE_CASE`	`JAVACODE`	`LOOKAHEAD`
`MORE`	`options`	`PARSER_BEGIN`	`PARSER_END`
`SKIP`	`SPECIAL_TOKEN`	`TOKEN`	`TOKEN_MGR_DECLS`

Slide 2

1.2 Création d'un analyseur

Voici une classe définissant un analyseur syntaxique de *Java* :

```
PARSER_BEGIN(JavaParser)
  public class JavaParser {
    public static void main(String args[]) {
      JavaParser parser;
      if (args.length == 1) {
        try {
          parser = new JavaParser(new java.io.FileInputStream(args[0]));
        } catch (java.io.FileNotFoundException e) {
          System.out.println("File " + args[0] + " not found.");
          return;
        }
      } else {
        System.out.println("Usage is : java JavaParser inputfile");
        return;
      }
      try {
        parser.CompilationUnit();
        System.out.println("Java program parsed successfully.");
      } catch (ParseException e) {
        System.out.println(e.getMessage());
        System.out.println("Encountered errors during parse.");
      }
    }
  }
PARSER_END(JavaParser)

// les productions, la racine est CompilationUnit
...
void CompilationUnit() :
{ }
{ // la partie droite de CompilationUnit
  ...
}
```

Définition de la classe de l'analyseur

```
PARSER_BEGIN(MonAnalyseur)
  public class MonAnalyseur {
    public static void main (String args []) {
      MonAnalyseur parser = new MonAnalyseur (System.in);
      try {
        parser.racine ();
      } catch (ParseException e) {
        System.out.println ("erreur : " + e.getMessage ());
      }
    }
  }
PARSER_END(MonAnalyseur)
// les productions
...
void racine () : { } { ... }
...
```

Slide 3

1.3 Les options

Pour le détail des options, voir la documentation :
`http://www.suntest.com/JavaCC/DOC/commandline.html`

Les options peuvent soit être passées à la commande `javacc`, soit données dans la définition de la grammaire. Majuscules et minuscules sont équivalentes pour les noms d'options.

Passées à la ligne de commande, elles ont l'une des formes suivantes

- `-option=valeur` : exemple `-static=true`
- `-option:valeur` : exemple `-debug_parser:true`
- `-option` : équivalent à `-option=true` pour les options à valeur booléenne
- `-NOoption` : équivalent à `-option=false`

Pour donner des options dans la grammaire, après le mot réservé `options`, on indique une liste analogue, entre accolades, par exemple :

```
options {static=true debug_parser=true}
```

Les options de la ligne de commande l'emportent sur les options définies dans la grammaire.

D'autres options en rapport avec la résolution des conflits sont :

- `choice_ambiguity_check` et `other_ambiguity_check` : aident à déterminer le choix du nombre de *lookahead*, respectivement pour les règles de la forme `A | B` (2 par défaut) et pour les règles de la forme `(A)*`, `(A)+`, `(A)?` (1 par défaut), mais n'influent pas l'analyseur
- `sanity_check` : par défaut, *JavaCC* effectue de nombreuses vérifications sur la grammaire; mettre cette option à faux a pour résultat d'accélérer la génération des analyseurs, mais aux risques et périls de l'utilisateur

Les options pour résoudre les conflits

- `LOOKAHEAD =` k
  - nombre maximum de symboles terminaux à examiner en général pour prendre une décision (1 par défaut – grammaire LL (1))
  - augmenter ce nombre diminue l'efficacité de l'analyseur et augmente sa taille
  - ce nombre peut être modifié localement, ce qui est un bon compromis
- `force_la_check` : permet d'effectuer les vérifications sur les conflits dans la grammaire (ce qui n'est pas fait par défaut quand l'utilisateur donne des indications pour résourdre les conflits – `LOOKAHEAD=2` par exemple – et se trompe)

Slide 4

D'autres options non mentionnées ci-contre :

- `debug_parser`, `debug_lookahead`, `debug_token_manager` : permettent d'avoir la trace de l'exécution des analyseurs syntaxique et lexical (défaut false)
- `error_reporting` : affichage détaillé des erreurs détectées par les analyseurs (défaut true)
- `common_token_action` : chaque fois qu'un terminal est reconnu, une méthode définie par l'utilisateur (`void CommonTokenAction(Token t)`) est appelée (défaut false)
- `user_token_manager` : permet de définir (programmer) son propre analyseur lexical (défaut false)
- `user_char_stream` : si l'option est vraie, la classe qui lit sur le flot d'entrée est définie par l'utilisateur; cette classe doit hériter de `CharStream` (defaut false)
- `build_parser` : si l'option est fausse, l'analyseur syntaxique n'est pas construit; on n'utilise donc que l'analyseur lexical (defaut true)
- `build_token_manager` : on ne construit pas d'analyseur lexical si l'option est fausse; cette option peut être utilisée lorsqu'on modifie uniquement la grammaire, mais pas les définitions des symboles terminaux : on récupère l'analyseur lexical produit auparavant (defaut true)

D'autres options utiles

- `static` : par défaut, l'analyseur produit est une classe avec des méthodes statiques; en ce cas, il ne peut y avoir qu'un seul analyseur; si l'option est fausse, les méthodes ne sont pas statiques et il est possible de créer plusieurs analyseurs
- `unicode_input` : l'analyseur lexical travaille par défaut sur des textes ASCII mais peut travailler sur des textes Unicode si cette option est vraie
- `java_unicode_escape` : permet d'obtenir un analyseur lexical reconnaissant les séquences unicode (défaut = false)
- `ignore_case` : l'analyseur lexical peut différencier les lettres minuscules des majuscules (option par défaut) ou non (si l'option est vraie)

Slide 5

2 Les productions

Il y a 4 types de productions

- les productions EBNF : ce sont les productions qui sont normalement utilisées pour décrire une grammaire (en ce sens, elles sont analogues aux règles de grammaire de *YACC*)
- les productions d'expressions régulières : elles servent à décrire les unités lexicales de la grammaire et sont donc assez proches de ce qui pourrait être décrit en *Lex*
- les déclarations de l'analyseur lexical : elles permettent de définir certaines propriétés de l'analyseur lexical
- les productions en code *Java* : utilisées lorsqu'une définition en EBNF n'est pas pratique; un faux ami, car *JavaCC* ne peut pas faire de vérification comme il peut le faire sur les production EBNF; elles doivent donc être employées avec précaution

Les productions

Il y a 4 types de productions

- les productions EBNF : la grammaire du langage proprement dite
- les productions d'expressions régulières : pour définir les unités lexicales du langage
- les déclarations de l'analyseur lexical : pour définir certaines propriétés de l'analyseur lexical
- les productions en code *Java* : permettent de "programmer" certaines règles de la grammaire (à ses risques et périls)

On voit en particulier que l'analyseur lexical et l'analyseur syntaxique sont définis ensemble

Slide 6

2.1 Les productions EBNF

Il faut se rappeler que *JavaCC* produit un analyseur récursif descendant : à chaque non terminal est associé une méthode chargée de reconnaître la partie droite. C'est pourquoi un non terminal se présente sous la forme d'un prototype de méthode (type de retour et paramètres). C'est une manière de définir des attributs hérités (paramètres) et des attributs synthétisés (paramètres et/ou type de retour); évidemment, on a une grammaire de nature LAG. La grammaire proprement dite doit être LL (1) [1], c'est à dire :

- ne doit pas avoir de règle récursive à gauche : $A = A\,\beta$ ou $A = B\,\beta$ et $B = A\,\alpha$
- dans une production de la forme : $R_1|R_2|...|R_k$, les premiers des R_i doivent être tous différents, de sorte que l'analyseur puisse produire un code du genre

```
si symbole_courant dans premiers (R1) analyser R1
sinon si symbole_courant dans premiers (R2) analyser R2
...
sinon si symbole_courant dans premiers (Rk) analyser Rk
sinon erreur
```

  Donc, s'il se trouve que des premiers d'un R_i sont aussi des premiers d'un R_j $(i < j)$, la règle analysée sera R_i pour les symboles communs

Chaque règle est une suite (éventuellement vide) de constructions élémentaires, qui sont soit des noms de non terminaux (sous forme d'appel de méthode), soit des terminaux (nommés ou directement sous forme d'expression régulière), soit des actions sémantiques (bloc entre accolades), soit une suite de sous-règles optionnelle (entre crochets `[...]` ou suivie de `?`), ou repétée 0, 1 ou plusieurs fois (`*` et `+`). Pour une règle de la forme $(R_1R_2...R_k)^*$, l'analyseur produit bouclera tant que le symbole courant est dans les premiers de R_1, ce qui posera problème si on a une production comme : $(R_1R_2...R_k)^*R_1$. Idem avec +, ? et les crochets.

1. On verra plus loin comment s'affranchir de cette contrainte.

Les productions EBNF

- production-EBNF = ***prototype-de-méthode*** : ***bloc*** { règles }
- règles = règle (| règle) *
- règle = (unité-de-règle) *
- unité-de-règle =
  - `(` règles `)` [`+` | `*` | `?`]
  - | [règles]
  - | [***variable*** `=`] expression-régulière
  - | [***variable*** `=`] ***appel-de-méthode***
  - | ***bloc***
  - | résolution-de-conflit

Slide 7

La grammaire ci-contre est extraite de la grammaire *Java* pour *JavaCC*

.

Noter que les symboles terminaux peuvent apparaître tels quels (`"abstract"`) ou sous forme d'expression régulière nommée (<`IDENTIFIER`>).

L'exemple ci-contre est assez lisible.

L'exemple de la page suivante, avec des actions et des attributs, est loin d'être aussi simple à lire ! Cet exemple aurait aussi pu être écrit :

```
void decls () :
{ String type; int t; }
{
  ( <INT> {t = 4;} | <FLOAT> {t = 8;} ) {type = token.image;}
  variable (type, t) ( "," variable (type, t) )* ";"
}

void variable (String type, int t) :
{ int nb, n; }
{
  <IDENT> { nb = 1; System.out.print (token.image + " : "); }
  (
    "["
    <CONST>
      { try {
          n = Integer.parseInt (token.image);
        } catch (NumberFormatException e) {
          n = 0;
        }
        System.out.print ("array of " + n + " ");
        nb = nb * n;
      }
    "]"
  )*
  { System.out.println (type + " --> taille = " + t * nb); }
}
```

ce qui est préférable à la récursivité droite.

Exemple de production EBNF

La déclaration de classes *Java*

```
void ClassDeclaration() :
{}
{
  ( "abstract" | "final" | "public" )*
  UnmodifiedClassDeclaration()
}

void UnmodifiedClassDeclaration() :
{}
{
  "class" <IDENTIFIER> [ "extends" Name() ]
  [ "implements" NameList() ]
  ClassBody()
}
```

Slide 8

Exemple de production EBNF avec actions et attributs

Des déclarations simples en C : on compte la taille des variables

```
void decls () :
{ String type; int t; }
{
  ( <INT> {t = 4;} | <FLOAT> {t = 8;} ) {type = token.image;}
  variable (type, t) ( "," variable (type, t) )* ";"
}

void variable (String type, int t) :
{ int nb = 1; }
{
  <IDENT> {System.out.print (token.image + " : ");}
  [ nb = dimensions (1) ]
  {System.out.println (type + " --> taille = " + t * nb);}
}
```

Slide 9

```
int dimensions (int nb) :
{ int v; int n; }
{
   "["
   <CONST>
     { try {
         v = Integer.parseInt (token.image);
       } catch (NumberFormatException e) {
         v = 0;
       }
       System.out.print ("array of " + v + " ");
       n = nb * v;
     }
   "]"
   [ n = dimensions (n) ]
   { return n; }
}
```

Slide 10

2.2 Les pièges des grammaires non LL (1)

La grammaire ci-contre n'est pas LL (1) : `ID` est dans les premiers de `declaration` et dans les premiers de `instruction`. *JavaCC* émet un message :

```
Warning: Choice conflict in (...)* construct at line 50, column 3.
         Expansion nested within construct and expansion following construct
         have common prefixes, one of which is: <IDENT>
         Consider using a lookahead of 2 or more for nested expansion.
Parser generated with 0 errors and 1 warnings.
```

La règle `( declaration() )*` sera appliquée tant que le symbole courant sera un identificateur. Par exemple :

```
a* b[10];
--> decl. de b
1+a;
--> expr.
a* b[10];
--> expr.
```

ou encore :

```
a* b[10];
--> decl. de b
a+b;
--> erreur
1+a;
--> expr.
a+b;
--> expr.
```

Ce conflit pourrait aussi être résolu de manière sémantique, selon que l'identificateur dénote un type ou une variable (voir plus loin).

Soit une grammaire inspirée de *C*

```
void bloc () : { }
{ ( declaration() )* ( instruction() )* }

void declaration () : { }
{ type() variable() ( "," variable() )* ";" }

void type () : { }
{ <ID> ("*")* }

void variable () : { String s; }
{ <ID> { s = token.image; } ( "[" <NB> "]" )*
  { print ("--> decl. de " + s); }
}

void instruction () : { }
{ expression() ";" { print ("--> expr."); }
}

void expression () : { }
{ operande() ( ("+" | "*") operande() )* }

void operande () : { }
{ <ID> ( "[" expression() "]" )* | <NB> }
```

Slide 11

2.3 Méthodes *Java* produites

Ci-dessous la méthode produite pour analyser la règle `bloc` de la grammaire précédente :

```
static final public void bloc() throws ParseException {
  label_1:
  while (true) {                          // boucler sur declarations
    if (jj_mask_0[getToken(1).kind]) {    // est-ce un premier de 'declaration' ?
      ;
    } else {                              // ce n'est un premier de type
      jj_expLA1[0] = jj_gen;
      break label_1;                      // sortir de la boucle
    }
    declaration();                        // analyser une declaration
    jj_consume_token(EOL);
  }
  label_2:
  while (true) {                          // boucler sur instructions
    if (jj_mask_1[getToken(1).kind]) {    // est un premier de 'instruction' ?
      ;
    } else {                              // non, sortir de la boucle
      jj_expLA1[1] = jj_gen;
      break label_2;
    }
    instruction();                        // analyser une instruction
    jj_consume_token(EOL);
  }
}

// les premiers de 'declaration
static boolean[] jj_mask_0 = new boolean[15];
static { jj_mask_0[ID] = true; }

// les premiers de 'instruction
static boolean[] jj_mask_1 = new boolean[15];
static { jj_mask_1[NB] = jj_mask_1[ID] = true;}
```

Méthode pour analyser la règle `variable`

```
void variable () : { String s; }
{ <ID> {s=token.image;} ( "[" <NB> "]" )* }
```

donne

```
static final public void variable()
                    throws ParseException {
  String s;
  jj_consume_token(ID);
          s = token.image; // action
  label_5:
  while (true) {          // analyser (...)*
    if (jj_mask_4[getToken(1).kind]) { ;
    } else {
      jj_expLA1[4] = jj_gen;
      break label_5;
    }
    jj_consume_token(12);      // 12 = '['
    jj_consume_token(NB);
    jj_consume_token(13);      // 13 = ']'
  }
}
static boolean[] jj_mask_4 = new boolean[15];
static { jj_mask_4[12] = true; }
```

Slide 12

2.4 Les productions d'expressions régulières

Les expressions régulières se présentent sous une forme plus conviviale que *Lex*. Au moins, les espaces ne sont pas significatifs.

Les productions d'expressions régulières sont utilisées pour définir les terminaux du langage.

Comme en *Lex* (*start conditions*), il est possible de définir différents états lexicaux; si la définition d'une expression régulière ne précise pas d'état lexical, elle ne sera "active" que dans l'état par défaut (`DEFAULT`), sinon elle ne sera active que dans les états indiqués. La forme avec `*` veut dire que l'expression régulière est active dans tous les états.

Une expression régulière "classique" peut utiliser les opérateurs $*$, $+$ et $?$ [1] Un élément d'une expression régulière "classique" est soit une chaîne de caractères, soit le nom d'une autre expression régulière (entre $<$ et $>$), soit un ensemble de caractères (dans ce dernier cas, à la différence de *Lex*, les caractères sont entre guillemets et séparés par des virgules. Par exemple :

```
< WHILE : "while" >
< #LETTRE : [ "a" - "z" ] >
< IDENT : ( <LETTRE> ) + >
```

Comme en *Lex*, c'est toujours la plus longue chaîne qui est reconnue, et si deux expressions régulières reconnaissent une même chaîne, c'est la première expression qui est prise en compte. Donc attention au piège classique quand les mots clés sont mis directement dans la grammaire :

```
...
< IDENT : ( <LETTRE> ) + >
...
void whileStat () : { } { "while" ... }
...
```

Comme `IDENT` apparaît avant `"while"`, le mot clé sera reconnu comme un identificateur !

1. `(A)?` est eqúivalent à `[A]`.

Les productions d'expressions régulières

production-expression-régulière =
[états-lexicaux]
genre [`[ IGNORE_CASE ]`] :
{ définition (| définition) * }

états-lexicaux =
<*>
| < ***identificateur*** (, ***identificateur***)* >

genre = `TOKEN` | `SPECIAL_TOKEN` | `SKIP` | `MORE`

définition = expression-régulière [***bloc***] [: ***identificateur***]

expression-régulière =
chaîne
| < [[#] ***identificateur*** :] expression-régulière-classique
| < ***identificateur*** >
| < `EOF` >

Slide 13

Le genre de l'expression régulière indique :

- `TOKEN` : ce sont des terminaux qui apparaîtront dans la grammaire; l'analyseur lexical crée un objet de type `Token` qui est retourné à l'analyseur syntaxique
- `SPECIAL_TOKEN` : ils ne sont pas retournés à l'analyseur lexical, mais chaînés au token suivant (qui peut-être aussi spécial) par le champ `specialToken` de la classe `Token`; ils sont utilisés par exemple pour conserver les commentaires
- `SKIP` : ce sont les chaînes qui ne sont pas retournées à l'analyseur syntaxique et dont on ne veut pas se souvenir; typiquement, les espaces, tabulations, fin de lignes ...
- `MORE` : analogue à `yymore()` de *Lex*; permet de reconnaître graduellement un symbole. Voir plus loin.

Le `#` indique qu'il ne s'agit pas d'un véritable symbole, mais seulement d'un nom qui sera utilisé dans d'autres expressions régulières.

Exemples de productions d'expressions régulières

- non mémorisé, non retourné à l'analyseur lexical

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

- mémorisé, et retourné à l'analyseur lexical

```
TOKEN : {
    < ABSTRACT: "abstract" >
  | < BOOLEAN: "boolean" >
  | < BREAK: "break" >
  ...
}
TOKEN : {
    < INTEGER_LITERAL :
        <DECIMAL> (["l","L"])?
      | <OCTAL> (["l","L"])?
    >
  | < #DECIMAL: ["1"-"9"] (["0"-"9"])* >
  | < #OCTAL: "0" (["0"-"7"])* >
  ...
}
```

Slide 14

Exemple extrait de la grammaire de *Java* montrant l'utilisation des états, des *more* et des symboles spéciaux pour traiter les trois types de commentaires *Java* :

```
// etat DEFAUT
  MORE : {
      "//" : IN_SINGLE_LINE_COMMENT
    |  <"/**" ~["/"]> : IN_FORMAL_COMMENT
    |  "/*" : IN_MULTI_LINE_COMMENT
  }

<IN_SINGLE_LINE_COMMENT>
  SPECIAL_TOKEN : {
    <SINGLE_LINE_COMMENT: "\n" | "\r" | "\r\n" > : DEFAULT
  }

<IN_FORMAL_COMMENT>
  SPECIAL_TOKEN : {
    <FORMAL_COMMENT: "*/" > : DEFAULT
  }

<IN_MULTI_LINE_COMMENT>
  SPECIAL_TOKEN : {
    <MULTI_LINE_COMMENT: "*/" > : DEFAULT
  }

<IN_SINGLE_LINE_COMMENT,IN_FORMAL_COMMENT,IN_MULTI_LINE_COMMENT>
  MORE : {
    < ~[] >
  }
```

Le champ `kind` de la classe `Token` permettra de savoir quel était le type du commentaire (`SINGLE_LINE_COMMENT`, `MULTI_LINE_COMMENT`, `FORMAL_COMMENT`).

`MORE`, `SPECIAL_TOKEN` et utilisation d'états

Pour reconnaître les commentaires de la forme `/* ... */`

- dans l'état par défaut, sur un début de commentaire, on passe dans un état ad-hoc

  ```
  MORE : { "/*" : IN_COMMENT }
  ```

- dans l'état `IN_COMMENT`, si on rencontre `*/`, c'est la fin du commentaire; on repasse en mode par défaut, en ayant cumulé tout le texte du commentaire (à cause des `MORE`).

  ```
  <IN_COMMENT> SPECIAL_TOKEN :
    { <COMMENT: "*/" > : DEFAULT }
  ```

- on reconnaîtra n'importe quel caractère jusqu'à la fin du commentaire (sauf `*/`)

  ```
  <IN_COMMENT> MORE : { < ~[] > }
  ```

- le commentaire n'est pas retourné comme symbole terminal à l'analyseur syntaxique, mais il sera chaîné au prochain terminal qui sera reconnu (`SPECIAL_TOKEN`)

Slide 15

2.5 Les déclarations pour l'analyseur lexical

L'analyseur lexical est une classe nommée `MonAnalyseurTokenManager`. Il est possible d'ajouter des déclarations (d'attribut ou de méthode) à celles produites automatiquement par *JavaCC* , et de les utiliser dans les actions lexicales (voir section 4).

Les déclarations pour l'analyseur lexical

- elles se présentent sous la forme d'un bloc *Java*

  déclarations-pour-l-analyseur-lexical =
  `TOKEN_MGR_DECLS` : ***bloc***
- ajoute de nouveaux attributs et/ou de nouvelles méthodes à la classe représentant l'analyseur lexical
- les attributs et les méthodes (produits par *JavaCC* ou définis par l'utilisateur) sont accessibles dans les actions des productions d'expressions régulières
- il ne peut y avoir qu'une seule règle de déclarations pour l'analyseur lexical

Slide 16

2.6 Les productions en code *Java*

Ces productions peuvent être utilisées par exemple lorsqu'on écrit un préprocesseur : à la rencontre d'un symbole particulier, on n'a pas envie d'analyser le texte qui suit, jusqu'à trouver un autre symbole. Un autre exemple est celui d'un compilateur qui dispose d'une directive `asm` pour insérer du code en assembleur : en ce cas, il saute tout le code assembleur jusqu'à rencontrer une directive qui en marque la fin.

On peut aussi imaginer des emplois de ces productions pour le rattrapage d'erreurs (dit en mode panique). C'est par exemple probablement ce que l'utilisateur a voulu écrire ci-contre : si la fenêtre n'est pas un des premiers de `UnAutreNonTerminal`, on saute jusqu'au prochain point-virgule. Mais il aurait dû écrire :

```
void UnNonTerminal () : { }
{   UnAutreNonTerminal () ... ";"
  | AllerAuPointVirgule ()
}
```

Les productions en code *Java*

- utilisées lorsque la définition en EBNF d'une partie de la grammaire n'est pas aisée

  production-en-code-*Java* =

  `JAVACODE`

  prototype et bloc de méthode
- exemple :

```
JAVACODE void AllerAuPointVirgule () {
// code pour avancer jusqu'au prochain ';'
}
```

- *JavaCC* ne peut pas analyser le code et considère ces productions comme une *boîte noire*
- mais il doit savoir comment les appeler :

```
void UnNonTerminal () : { }
{   AllerAuPointVirgule ()
  | UnAutreNonTerminal () ... ";"
}
```

  ne peut fonctionner car *JavaCC* ne sait pas sur quel symbole appeler une règle ou l'autre (et `AllerAuPointVirgule` est systématiquement invoquée)

Slide 17

3 La résolution des conflits

Par défaut *JavaCC* accepte des grammaires LL (1). Il est donc confronté à des conflits lorsque la grammaire qui lui est soumise n'est pas LL (1).

3.1 Points de conflit potentiels et résolution par défaut

Dans un analyseur prédictif, le symbole de fenêtre détermine la règle qui devra être reconnue. Les conflits interviennent lorsque la fenêtre ne permet pas de choisir entre deux règles (ou parties de règles). Ces points de conflit sont multiples. *JavaCC* détecte les récursivités à gauche et les conflits LL (1) :

```
void A () : { } { B () "a" }
void B () : { } { A () "b" C () }
void C () : { } { ( "c" | C () "c" ) }
```

donnera

```
Error: Line 32, Column 1: Left recursion detected: "A... --> B... --> A..."
Error: Line 34, Column 1: Left recursion detected: "C... --> C..."
```

et

```
void T () : { } { ( X () | Y () | "z" Z () ) }
void X () : { } { "t" "x" }
void Y () : { } { "t" "y" }
void Z () : { } { ( X () )+ Y () }
```

donnera

```
Warning: Choice conflict involving two expansions at
         line 32, column 21 and line 32, column 28 respectively.
         A common prefix is: "t"
         Consider using a lookahead of 2 for earlier expansion.
Warning: Choice conflict in (...)+ construct at line 35, column 19.
         Expansion nested within construct and expansion following construct
         have common prefixes, one of which is: "t"
         Consider using a lookahead of 2 or more for nested expansion.
```

Points de conflits potentiels et résolution par défaut

- règles $A = (\beta \mid \gamma)$: si $First(\beta) \cap First(\gamma)$ non vide
  - β est choisi pour tout symbole de fenêtre dans $First(\beta)$
- règles EBNF $A = \beta (\gamma)^{\oplus} \delta$: si $First(\gamma) \cap First(\delta)$ non vide ($\oplus = *$ ou `+`)
  - γ est choisi tant que le symbole de fenêtre est dans $First(\gamma)$
- règles EBNF $A = \beta (\gamma)^{?} \delta$ ou $A = \beta [\gamma] \delta$: si $First(\gamma) \cap First(\delta)$ non vide
  - γ est choisi si le symbole de fenêtre est dans $First(\gamma)$

Slide 18

3.2 Les moyens de résoudre les conflits

JavaCC offre différents moyens pour résoudre les conflits, qu'il est préférable d'utiliser à bon escient pour ne pas dégrader les performances des analyseurs produits.

3.2.1 Solution globale

Il est possible de demander à *JavaCC* de reconnaître une grammaire LL (k), avec k supérieur à 1 (option `lookahead=k`). Solution en général assez coûteuse et à éviter.

3.2.2 Solution locale

En général, une grammaire est LL (1) "presque partout". Une meilleure solution que la précédente est d'indiquer localement que la grammaire n'est pas LL (1). Par exemple, réécrire la grammaire non LL (1) comme c'est fait ci-contre fait disparaître le conflit de choix entre `X` et `Y` dans `T`, car le symbole qui suit le `t` permet de choisir. De même dans `Z`.

Attention. L'indication `lookahead` n'a de sens que dans un point de choix de l'analyseur.

Résolution des conflits

- soit la grammaire

```
void T () : { } { ( X () | Y () | "z" Z () ) }
void X () : { } { "t" "x" }
void Y () : { } { "t" "y" }
void Z () : { } { ( X () )+ Y () }
```

- c'est le symbole après `t` qui permet de choisir entre `X` et `Y` dans `T`, et permet de boucler ou non dans `Z` : la grammaire est LL (2)
- il est possible de donner l'option globale `lookahead=2` : solution coûteuse en général
- on peut indiquer que la grammaire est localement LL (2)

```
void T () : { } { ( LOOKAHEAD (2) X () | Y () | "z" Z () ) }
void Z () : { } { ( LOOKAHEAD (2) X () )+ Y () }
```

Slide 19

3.2.3 Solution lexicale

Elle permet d'avoir des grammaires plus générales que celles de la classe LL (voir exemple ci-contre). Mais avant de se résoudre à employer des moyens aussi puissants, il est préférable de réfléchir s'il n'est pas possible d'obtenir une grammaire LL (1), par exemple pour la grammaire ci-contre :

```
void T ()    : { } { ( SXY() | Y() | "z" Z() ) }
void X ()    : { } { "x" }
void Y ()    : { } { "y" }
void Z ()    : { } { S() X() SXPY() }
void S ()    : { } { ("t")+ }
void SXY ()  : { } { S() ( X() | Y() ) }
void SXPY () : { } { Y() | S() ( X() SXPY() | Y() ) }
```

Dans la grammaire suivante, le lookahead lexical permet de choisir entre `X` et `Y`, mais bien que la suite de `t` a été lue, ainsi que le symbole qui suit, pour faire le choix dans `Y` entre `Y1` et `Y2`, il est encore nécessaire de résoudre le conflit.

```
void T ()  : { } { ( LOOKAHEAD ( (("t")+ "x") ) X() | Y() | "z" Z() ) }
void X ()  : { } { ( X1() | X2() ) }
void Y ()  : { } { ( LOOKAHEAD (3) Y1() | Y2() ) }
void Z ()  : { } { ( LOOKAHEAD ( (("t")+ "x") ) X() )+ Y() }
void X1 () : { } { ("t")+ "x" }
void X2 () : { } { "x" "z" }
void Y1 () : { } { "t" "t" "z" }
void Y2 () : { } { ("t")* "y" }
```

Enfin, il est possible de mêler les deux possibilités :

```
void T ()  : { } { ( LOOKAHEAD (25, (("t")+ "x") ) X() | Y() | "z" Z() ) }
```

En ce cas, s'il y a plus de 25 `t`, ce sera `X` qui sera choisi !

Note. Les actions sémantiques sur les symboles terminaux ne sont pas effectuées lorsque l'analyseur avance pour résoudre le conflit. Elles ne seront effectués que lorsque l'analyse de la règle choisie aura lieu et que les symboles terminaux seront consommés.

Grammaires non LL (k)

- soit la grammaire non LL (k), pour k aussi grand que possible

```
void T () : { } { ( X () | Y () | "z" Z () ) }
void X () : { } { ("t")+ "x" }
void Y () : { } { ("t")* "y" }
void Z () : { } { ( X () )+ Y () }
```

- on pourrait indiquer un k très grand, en croisant les doigts pour ne jamais rencontrer de suite de `t` aussi longue
- ou on ne prévoit pas de limite à k

```
void T():{} {(LOOKAHEAD(("t")+ "x") X() | Y() | "z" Z())}
void Z():{} {(LOOKAHEAD(("t")+ "x") X() )+ Y()}
```

- l'analyseur lexical "avalera" la suite de `t` et si elle est suivi d'un `x`, fera le choix de `X` et sinon d'un `y`, puis l'analyse syntaxique reprendra sur le premier `t` de la suite

Slide 20

3.2.4 Solution syntaxique

Généré avec les options `-debug_parser -debug_lookahead`, l'analyseur donne à l'exécution (extraits de la sortie, beaucoup plus bavarde) :

```
i (i);
Call:   Stat
  Call:   Des(LOOKING AHEAD...)
****** FOUND A "i" MATCH (i) ******
    Visited token: <"i">; Expected token: <"i">
****** FOUND A "(" MATCH (() ******
    Visited token: <"(">; Expected token: <"(">
    Call:   Exp(LOOKING AHEAD...)
      Call:   Des(LOOKING AHEAD...)
****** FOUND A "i" MATCH (i) ******
        Visited token: <"i">; Expected token: <"i">
****** FOUND A ")" MATCH ()) ******
        Visited token: <")">; Expected token: <"(">
      Return: Des(LOOKAHEAD SUCCEEDED)
      Visited token: <")">; Expected token: <"+">
    Return: Exp(LOOKAHEAD SUCCEEDED)
    Visited token: <")">; Expected token: <")">
****** FOUND A ";" MATCH (;) ******
    Visited token: <";">; Expected token: <"(">
  Return: Des(LOOKAHEAD SUCCEEDED)
  Visited token: <";">; Expected token: <":=">     <=====
  Call:   Call
    Consumed token: <"i">
    Consumed token: <"(">
    Call:   Args
      Call:   Exp
        Call:   Des
          Consumed token: <"i">
-->Exp
        Return: Des
      Return: Exp
    Return: Args
    Consumed token: <")">
    Consumed token: <";">
  Return: Call
Return: Stat
```

Encore plus fort ! Résolution syntaxique

- les conflits ne peuvent pas toujours être résolus à l'aide d'expressions régulières
- par exemple un langage où `id (exp)` peut être une variable indicée ou un appel de fonction

```
void Stat(): {} { LOOKAHEAD ((Des() ":=")) Ass() | Call() }
void Ass() : {} { Des() ":=" Exp() ";" }
void Call(): {} { "i" "(" Args() ")" ";"  }
void Des() : {} { "i" ( "(" Exp() ")" )* }
void Exp() : {} { Des() ( "+" Des() )* {print("-->Exp");} }
void Args(): {} { Exp() ( "," Exp() )* }
```

- les action sémantiques ne sont pas effectuées lors de la résolution de conflit

Slide 21

3.2.5 Solution sémantique

Cette solution peut présenter l'avantage de ne pas avancer sur le texte.
La grammaire complète est la suivante :

```
PARSER_BEGIN(conflit5)
  import java.util.*;

  public class conflit5 {
    static Hashtable dict;

    public static void main (String args []) {
      conflit5 parser = new conflit5 (System.in); dict = new Hashtable (10);
      try {
        parser.Body ();
      } catch (ParseException x) {
        System.out.println ("erreur sur " + x.currentToken.image);
      }
    }
    private static boolean IsVariable () {
      Token t = getToken (1); Object o = dict.get (t.image);
      System.out.print ("--> examen de " + t.image);
      return (o != null && (String) o == "var");
    }
  }
PARSER_END(conflit5)

SKIP : { "\n" | " " }
TOKEN : { <VAR : "var"> | <FCT : "fct"> | <ID : (["a" - "z"])+ > }

void Body() : {} { ( Decl () ";" )+ ( Stat () ";" )+ <EOF> }
void Decl() : {} { ( DecV() | DecF() ) }
void DecV() : {} { <VAR> <ID> { dict.put (token.image, "var"); } }
void DecF() : {} { <FCT> <ID> { dict.put (token.image, "fct"); } }
void Stat() : {} { LOOKAHEAD({IsVariable()}) Ass() | Call() }
void Ass()  : {} { Var() ":=" Exp() }
void Call() : {} { <ID> "(" Args() ")" }
void Des()  : {} { LOOKAHEAD({IsVariable()}) Var () | Call () }
void Var()  : {} { <ID> ( "(" Exp() ")" )* }
void Exp()  : {} { Des() ( "+" Des() )* }
void Args() : {} { Exp() ( "," Exp() )* }
```

Toujours plus fort ! Résolution sémantique

- les conflits ne peuvent pas toujours être résolus par la syntaxe : si on étend le langage précédent

  ```
  void Des(): {} { ( Var() | Call() ) }
  void Var(): {} { <ID> ( "(" Exp() ")" )* }
  ```

  `v := x (i)` est ambigu : `x (i)` est une variable si `x` est une variable et un appel si `x` est une fonction, d'où

  ```
  void Stat(): {} { LOOKAHEAD({IsVariable()}) Ass() | Call() }
  void Des() : {} { LOOKAHEAD({IsVariable()}) Var() | Call() }
  ```

- ce qui suppose l'existence d'un dictionnaire

  ```
  private static boolean IsVariable () {
    Token t = getToken (1); Object o = dict.get (t.image);
    return (o != null && (String) o == "var");
  }
  ```

Slide 22

3.2.6 Forme générale de résolution des conflits

Forme générale de résolution des conflits

- les trois genres de résolution peut être présents simultanément

  résolution-des-conflits =

  `LOOKAHEAD (` [***constante-entière***] [`,`]
  [règles] [`,`]
  [{***expression-booléenne***}]
  `)`

- au moins une des trois composantes doit être présente, et s'il y en a plus d'une, elles doivent être séparées par des virgules
- la limite de symboles examinés n'est pas prise en compte pour la résolution sémantique
- la résolution sémantique semble emporter la décision sur les autres

Slide 23

4 L'analyseur lexical

L'analyseur lexical produit est très classique : c'est un automate à états finis.

L'analyseur lexical dispose du constructeur suivant :

```
MonAnalyseurTokenManager (CharStream stream)
```

Ce constructeur est en général appelé automatiquement par l'analyseur syntaxique, et ce constructeur n'est disponible qu'avec l'option `USER_TOKEN_MANAGER` **fausse** et l'option `USER_CHAR_STREAM` **vraie**. Dans la situation par défaut (les 2 options à faux), un constructeur semblable est produit, mais le type du paramètre change en fonction des options sur l'Unicode. Et bien sûr, il n'y a pas de classe `MonAnalyseurTokenManager` avec l'option `USER_TOKEN_MANAGER` **vraie** .

L'analyseur lexical dispose aussi des attributs et des méthodes déclarés par l'utilisateur (`TOKEN_MGR_DECLS`)

La classe `Token` dispose aussi de :

- `int beginLine, beginColumn, endLine, endColumn` : positions de début et de fin du symbole dans le texte source
- `Token next` : le chaînage avec le symbole suivant qui a été lu (en cas de lecture en avance lors d'une résolution de conflit)
- `Token specialToken` : les symboles spéciaux reconnus (mais non passés à l'analyseur syntaxique) avant ce symbole

L'analyseur lexical

- l'analyseur lexical est une classe appelée `MonAnalyseurTokenManager`
  - `Token getNextToken () throws TokenMgrError` : la méthode qui reconnaît un des symboles définis comme `TOKEN` dans la grammaire
- un symbole est un object de la classe `Token` qui possède notamment :
  - `int kind` : le code interne du symbole, manipulable au travers des noms symboliques dans les expressions régulières

    ```
    < PLUS : "+" >
    ```

    ces noms symboliques sont définis dans la classe `MonAnalyseurConstants`
  - `String image` : la chaîne qui a été reconnue

Slide 24

5 L'analyseur syntaxique

On donne ci-dessous quelques autres méthodes utiles non mentionnées ci-contre.

Toute règle de grammaire `type R (paramètres)` est traduite par une méthode de l'analyseur syntaxique :

```
type R (parametres) throws ParseException
```

Il est important de noter que contrairement aux analyseurs ascendants, on peut analyser une sous grammaire en appelant une méthode quelquonque `R`.

On dispose aussi des constructeurs (suivant les options)

- `MonAnalyseur (CharStream stream)` : idem celui ci-contre, mais quand `USER_TOKEN_MANAGER` est **faux** et `USER_CHAR_STREAM` est **vrai**
- `MonAnalyseur (TokenManager tm)` : quand l'utilisateur programme son propre analyseur lexical

Les deux méthodes suivantes :

```
void enable_tracing ()
void disable_tracing ()
```

permettent de tracer un peu plus finement l'exécution des analyseurs que les options `DEBUG_*`.

L'analyseur syntaxique

- `MonAnalyseur (java.io.InputStream stream)` : constructeur de l'analyseur, qui crée l'analyseur lexical qui lira sur `stream` (disponible si les options `USER_TOKEN_MANAGER` et `USER_CHAR_STREAM` sont **toutes deux fausses** – ce qui est le cas par défaut)
- `Token getNextToken () throws ParseException` : "avale" le prochain symbol, à ne pas confondre avec :
- `Token getToken (int x) throws ParseException`, qui retourne le x-ième symbole après le symbole courant, à condition que x symboles aient été lus (x ne peut pas être négatif)
- `Token token` : le symbole courant, équivalent à `getToken(0)`

Slide 25

6 Traitement des erreurs

La hiérarchie des classes d'exceptions servant à gèrer les erreurs n'est pas très figée.

Traitement des erreurs

- une erreur syntaxique lève une exception `ParseException`, tandis qu'une erreur lexicale lève une `TokenMgrError`
- il est possible d'indiquer une liste d'exceptions (en plus des deux précédentes) dans une règle :

  ```
  void R () throws X1, X2 : { ... } { ... }
  ```

- récupération en mode panique

  ```
  void R () {} { R1() | R2() | AvancerJusque(T) }
  ```

  où `AvancerJusque` est une production `JAVACODE` qui avance (par `getNextToken` jusqu'à lire le symbole `T`, et qui sera appelée si le symbole courant n'est pas dans les premiers de `R1` ou `R2`
- non satisfaisant si l'erreur a lieu dans `R1` ou `R2`

Slide 26

6.1 Un exemple plus complet de récupération d'erreur

```
PARSER_BEGIN(recup_erreur)

  import java.util.*;

  public class recup_erreur {
    static Hashtable dict;

    public static void main (String args []) {
      recup_erreur parser = new recup_erreur (System.in);
      dict = new Hashtable (10);
      try {
        parser.Body ();
      } catch (ParseException x) {
        System.out.println ("erreur sur " + x.currentToken.image);
      }
    }
    private static boolean IsVariable () {
      Token t = getToken (1);
      Object o = dict.get (t.image);
      System.out.println ("--> examen de " + t.image);
      return (o != null && (String) o == "var");
    }
    private static void SkipTo (int sy, boolean consume) {
      Token p = token; Token t = getNextToken ();
      while (t.kind != sy && t.kind != EOF) { p = t; t = getNextToken (); }
      if (consume) token = t; else token = p;
    }
}

PARSER_END(recup_erreur)

SKIP : { "\n" | " " }
TOKEN : { <VAR : "var"> | <FCT : "fct"> }
TOKEN : { <ID : (["a" - "z"])+ > }

/* nommer les symboles qui seront sautes */
TOKEN : { <ASS : ":="> | <SC : ";"> | <RP : ")" > }

void Body() : {} { ( Decl () )+ ( Stat () )+ <EOF> }

void Decl() : {}
  { try {
      ( DecV() | DecF() ) ";"
    } catch (ParseException e) {
      System.out.println (e.toString ()); SkipTo (SC, true);
    }
  }

void DecV() : {} { <VAR> ( <ID> { dict.put (token.image, "var"); } )+ }
void DecF() : {} { <FCT> ( <ID> { dict.put (token.image, "fct"); } )+ }

void Stat() : {}
  { try {
      ( LOOKAHEAD({IsVariable()}) Ass() | Call(SC) ) ";"
    } catch (ParseException e) {
      System.out.println (e.toString ()); SkipTo (SC, true);
    }
  }

void Ass()        : {} { Var(ASS) ":=" Exp(SC) }
void Des(int sy)  : {} { LOOKAHEAD({IsVariable()}) Var(sy) | Call(sy) }
void Call(int sy) : {} { <ID> "(" Args() ")" }
void Exp(int sy)  : {} { Des(sy) ( "+" Des(sy) )* }

void Var(int sy)  : {}
  { try {
      <ID> ( "(" Exp(RP) ")" )*
  } catch (ParseException e) {
      System.out.println (e.toString ()); SkipTo (sy, false);
  }
}

void Args() : {}
  { try {
      Exp(RP) ( "," Exp(RP) )*
    } catch (ParseException e) {
      System.out.println (e.toString ()); SkipTo (RP, false);
    }
  }
```

Récupération des erreurs “profondes”

– une meilleur solution est :

```
void R () {} { try {
                  ( R1() | R2() ) <T>
                } catch (ParseException e) {
                    AvancerJusque(T)
                }
              }
```

– avec

```
private static void AvancerJusque (int sy) {
  while (token.kind != sy && token.kind != EOF)
    token = getNextToken ();
}
```

Slide 27

7 Ce qu’on trouve encore en *JavaCC* et qui n’est pas mentionné dans ce cours.

La documentation des grammaires (*JJDoc*) et la construction d’un arbre abstrait (*JJTree*)

8 Exercices

8.1 Une calculette simple

Implémenter la grammaire (attribuée) et les actions sémantiques d’une calculette disposant des 4 opérations (+, -, *, /) et d’opérandes qui sont des constantes entières, par exemple 2 * (3 + 12 / 4).

8.2 Une calculette avec des variables

Étendre la calculette de manière à accepter

```
x = 2 * 3
y = x + 5 * 4
y * 4
```

8.3 Une calculette avec des tableaux

Étendre la calculette de manière à accepter en plus

```
x = {1, 2, 4, 7, 4}
y = {x (1), 9, 12, x (3), 2}
x (2) + y (1)
```

8.4 Rattrapage d’erreur

Munir l’analyseur syntaxique d’un rattrapage d’erreur “honorable” (c’est à dire faisant mieux que sauter en fin de ligne)