

Analyse lexicale

Analyse syntaxique

Évaluation

Génération de code - Cours 1

{Remi.Forax, Matthieu.Constant, Michel.Chilowicz}@univ-mlv.fr



Tatoo nain d'Argentine (© Cliff1066, CC-BY)

Génération de code

- Objectifs principaux
 - Comprendre le fonctionnement interne d'un compilateur et donc d'un programme
 - Écrire un compilateur
 - Utilisation du compilateur de compilateur Tatoo
- Séances :
 - Un cours d'introduction (celui-ci)
 - Des TPs d'introduction à Tatoo et de suivi de projet

Plan de cette séance

- Introduction : la compilation
- Contenu des cours et des travaux pratiques
- Analyse lexicale avec Tatoo (expressions rationnelles)
- Analyse syntaxique avec Tatoo (grammaire, analyse LR, priorité)
- Analyseur combinant analyses lexical et syntaxique
- Evaluation d'expressions booléennes avec une pile

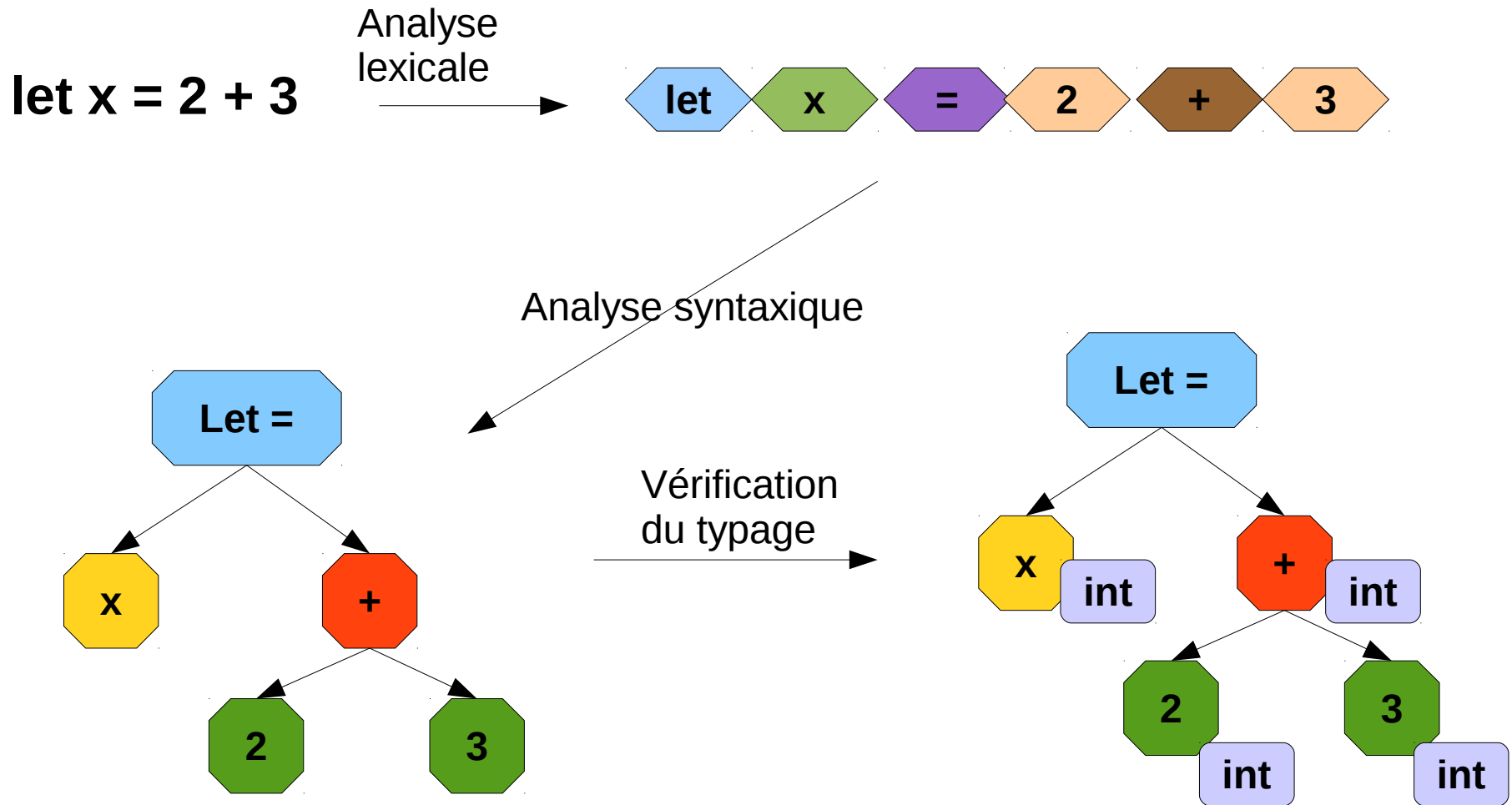
Un compilateur

- Un compilateur est un programme qui prend une description en entrée et la transforme en programme en sortie
- Un compilateur est un générateur de programmes

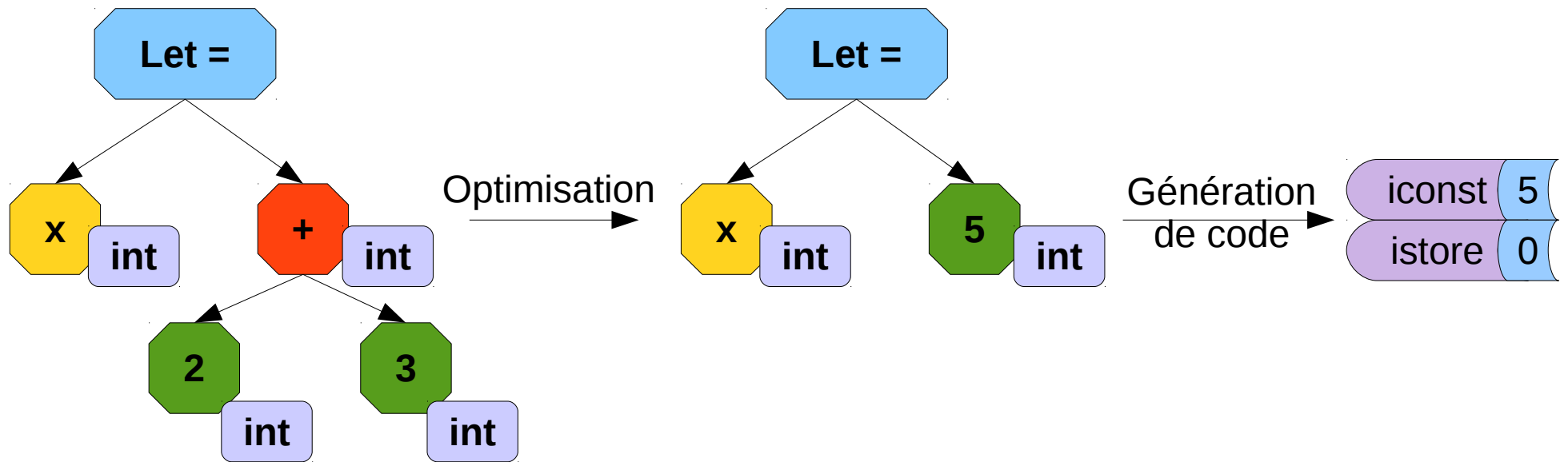
Etapes de la compilation

- **Analyse lexicale** (transforme un fichier en une séquence de lexèmes à l'aide d'un lexique)
- **Analyse syntaxique** (transforme une suite de lexèmes en arbre AST à l'aide d'une grammaire)
- **Analyse sémantique** (vérification du typage, optimisation...)
- **Génération de code** (transformation de l'AST en code exécutable)

Etapes de la compilation



Etapes de la compilation



Quelques générateurs de compilateurs

- Génération de compilateurs en langage C :
 - (F)lex pour l'analyse lexicale
 - Yacc, Bison pour l'analyse syntaxique
- Génération de compilateurs en Java :
 - JavaCC(LL) : compilateur historique de Sun
 - ANTLR(LL) : assez populaire
 - Jflex/JCup(LALR) : syntaxe proche de Flex/Yacc
 - SableCC(LR) : génération d'ASTs

Tatoo

- Générateur d'analyseurs lexical et syntaxique, développé au LIGM (R. Forax, J. Cervelle et G. Roussel)
- Utilisé en IR depuis ~5 ans et depuis plus longtemps en recherche
- Quelques caractéristiques :
 - Écrit en Java
 - Séparation propre entre le lexique et la grammaire
 - Construction automatique de l'AST

Techniques utilisées

Analyse lexicale	Expressions rationnelles
Analyse syntaxique	Grammaires – analyse LR
Analyse sémantique	Analyse procédurale - visiteur
Génération de code	Analyse procédurale - visiteur

Tatoo - principe

- Les expressions rationnelles et la grammaire sont définies dans un fichier EBNF.
 - EBNF : **Extended** Backus Naur Form
 - ✓ Extended : support aisé des répétitions (*, +)
- Tatoo génère automatiquement à partir de la spécification EBNF les classes Java implantant l'analyseur lexical et l'analyseur syntaxique correspondant.
- Il faut implanter en Java l'analyseur en combinant ces classes.

Analyse lexicale ou tokenisation

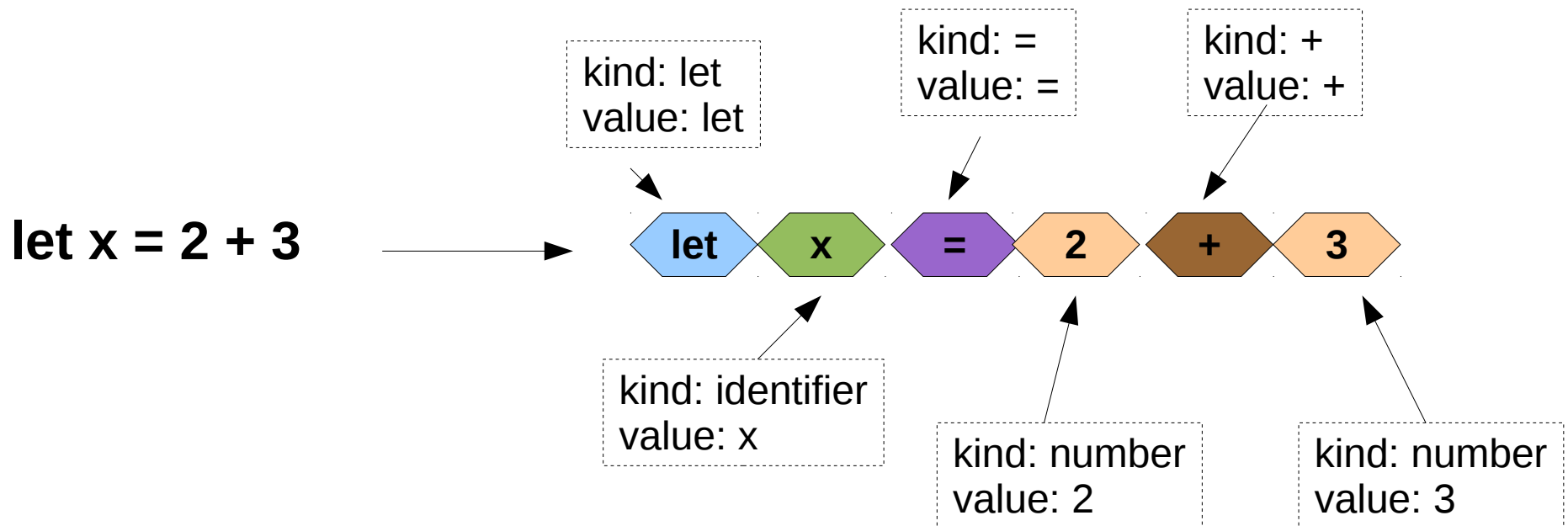
- L'analyseur lexical (ou lexer) transforme un fichier en une séquence de lexèmes (tokens) à l'aide d'un ensemble d'expressions rationnelles (transformées en automates)
- Un token contient deux informations :
 - la catégorie du token
 - sa valeur (la chaîne de caractères correspondante)

Analyse lexicale ou tokenisation

- Simplifie l'écriture de la grammaire
- L'analyseur lexical supprime
 - Les espaces
 - Les commentaires
 - Les retours à la ligne (suivant le langage), ...

Tokenisation

- Pour chaque *token*, l'automate qui a reconnu le motif fournit une catégorie (*kind*) et sa valeur (*value*)



Catégories

- Les catégories les plus fréquentes :
 - mots-clés du langage (types primitifs, introduction de structures, contrôle de flux...)
 - symboles de ponctuation (séparation d'arguments, de lignes)
 - opérateurs
 - littéraux (valeurs entières, réelles, ..., chaînes de caractères)
 - Identificateurs (noms des types, des variables, des fonctions, ...)

Tokenisation avec Tatoo

Configuration dans un fichier .ebnf

tokens: ← kind
let = 'let' ← expression rationnelle (*regex*)

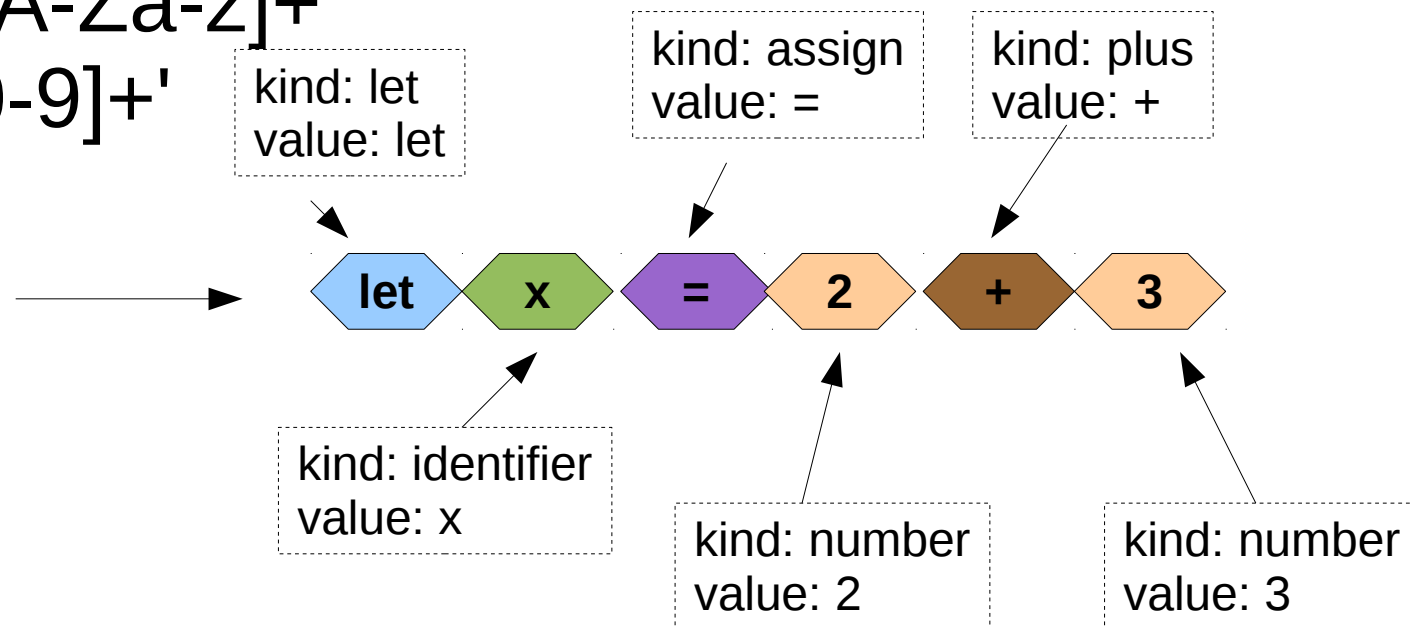
assign = '='

plus = '\+'

identfier = '[A-Za-z]+'

number = '[0-9]+'

let x = 2 + 3



Format des expressions rationnelles de Tatoo

- `foo` `f` suivi de `o` suivi de `o` (concaténation)
- `[a-z]` lettre entre `a` et `z`
- `[^a]` toute les lettres sauf `a` (complément)
- `.` n'importe quel caractère
- `a|b` `a` ou `b`
- `a?` `a` ou ϵ
- `a*` `a` répété 0 à n fois
- `a+` `a` répété 1 à n fois
- `\|` le caractère `|` (déspécialisation avec `\`)

Exemples d'expressions rationnelles

- Mots-clés : 'if', 'for', 'return'
- Opérateurs : '\+', '-', '*'
- Entiers : '[0-9]+|(0x[0-9A-Fa-f]+)'
- Caractères : "'[^']'"'
- Chaîne de caractères : "'[^"]*'"'
- Identificateurs : [A-Za-z_][0-9A-Za-z_]*

Procédure de tokenisation

- Application glissante
- L'analyseur lexical lance la reconnaissance sur tous les automates en parallèle
- En cas d'ambiguïtés entre plusieurs automates
 - On prend celui qui reconnaît le + long motif
 - Si 2 automates reconnaissent le même motif, on prend celui qui est déclaré en premier dans l'ordre du fichier EBNF

Tatoo a différents types de tokens

- Les tokens (**tokens:**) qui correspondent à des mots-clés
- Les blancs (**blanks:**) qui sont des espaces qui ne seront pas pris en compte
- Les commentaires (**comments:**) qui ne seront pas pris en compte par l'analyseur syntaxique

Exemple

- Attention, les sections doivent être déclarées dans cet ordre

tokens:

`id = [A-Za-z]([0-9A-Za-z])*`

blanks:

`space= "(\t|\r|\n)+"`

comments:

`comment="#"([^\r\n])*(\r)?\n"`

Tokenisation - exemple

l	e	t		x	=	2	+	3
---	---	---	--	---	---	---	---	---

let

← value

← kind

l	e	t		x	=	2	+	3
---	---	---	--	---	---	---	---	---

let

l	e	t		x	=	2	+	3
---	---	---	--	---	---	---	---	---

let

l	e	t		x	=	2	+	3
---	---	---	--	---	---	---	---	---

let

Tokenisation – exemple (suite)

l	e	t		x	=	2	+	3
---	---	---	--	---	---	---	---	---

let x = 2

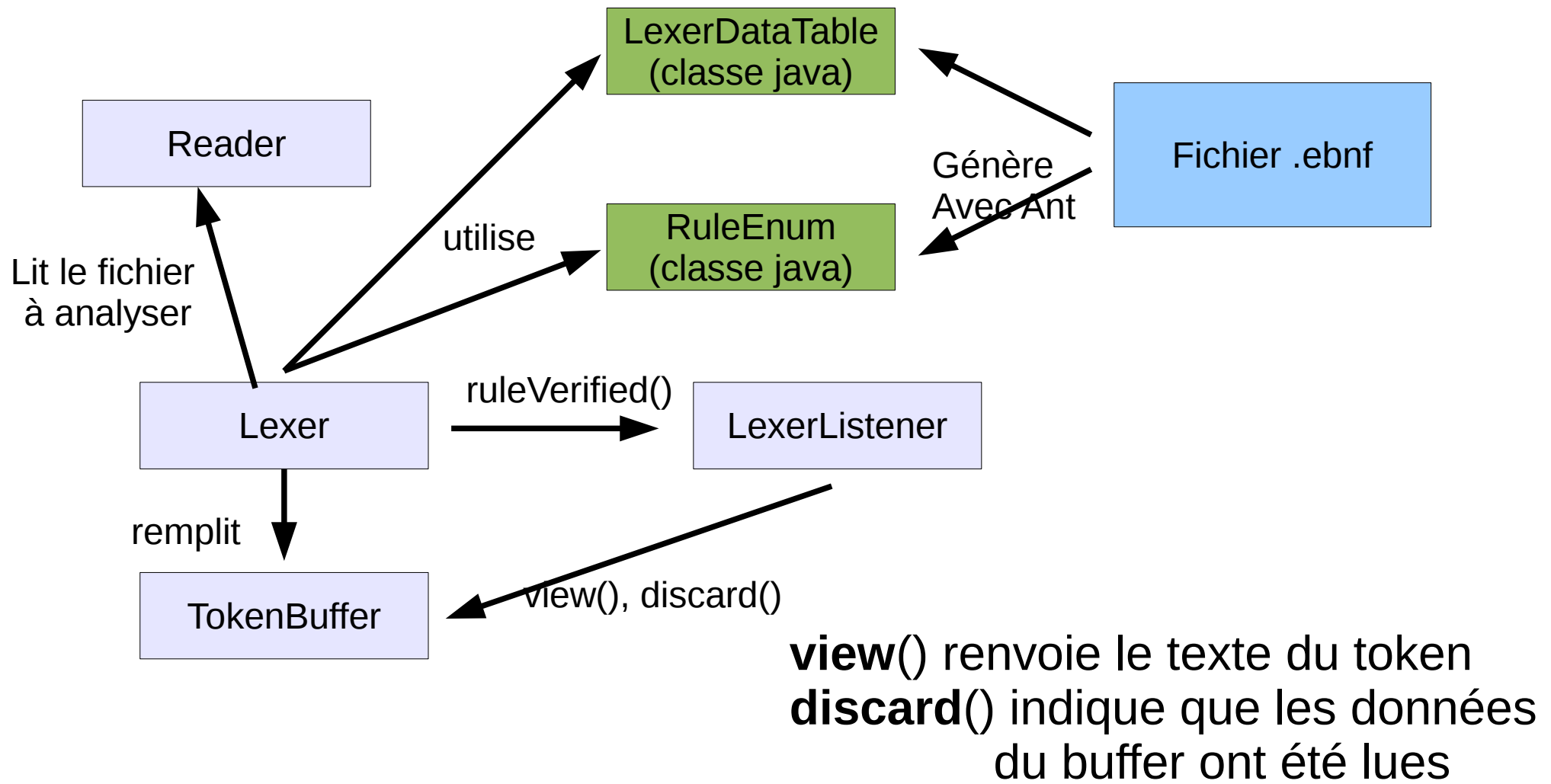
l	e	t		x	=	2	+	3
---	---	---	--	---	---	---	---	---

let x = 2 +

l	e	t		x	=	2	+	3
---	---	---	--	---	---	---	---	---

let x = 2 + 3

Fonctionnement du lexer de Tatoo



Génération des classes de base du lexer à l'aide de ANT et Tatoo

```
<?xml version="1.0"?>
<project name="eval" default="all" basedir=".">
  <property name="tatoo-build.dir" location="../build-lib"/>
  <property name="tatoo.jar" location="${tatoo-build.dir}/tatoo.jar"/>

  <property name="gen-src" value="gen-src"/>
  <property name="ebnf.file" value="file.ebnf"/>

  <property name="lexer.package" value="fr.umlv.compil.file.lexer"/>

  <target name="tasks">
    <taskdef name="ebnf" classname="fr.umlv.tatoo.cc.ebnf.main.EBNFTask"
      classpath="${tatoo.jar}"/>
  </target>

  <target name="ebnf" depends="tasks">
    <delete dir="${gen-src}"/>
    <ebnf destination="${gen-src}" ebnfFile="${ebnf.file}" parserType="lalr">
      <package lexer="${lexer.package}"/>
    </ebnf>
  </target>
```

Implémentation du lexer

```
Reader reader = ...
```

```
LexerListener<RuleEnum, TokenBuffer<CharSequence>> lexerListener =  
    new LexerListener<RuleEnum, TokenBuffer<CharSequence>>() {  
        @Override  
        public void ruleVerified(RuleEnum rule, int tokenLength,  
                                TokenBuffer<CharSequence> buffer) {  
            System.out.println("recognize token: "+rule+" "+buffer.view());  
            buffer.discard();  
        }  
    };
```

```
SimpleLexer lexer = Builder.lexer(LexerDataTable.createTable()).  
                        reader(reader).  
                        listener(lexerListener).  
                        create();  
  
lexer.run();
```

Buffer de Tatoo

- Tatoo utilise une interface *LexerBuffer* pour discuter avec les flux de caractères en entrée
- Cette interface permet de ne pas avoir le contenu total d'un fichier pour effectuer l'analyse lexicale.
- Seule la partie nécessaire à l'analyse lexicale réside dans un buffer qui s'agrandit automatiquement si nécessaire

Buffer de Tatoo

- Le *ReaderWrapper* est le buffer qui lit des données à partir d'un `java.io.Reader`

```
Reader reader = ...
LexerBuffer buffer = new ReaderWrapper(reader, null);

LexerListener<RuleEnum, TokenBuffer<CharSequence>> lexerListener =
    ...;
SimpleLexer lexer = Builder.lexer(LexerDataTable.createTable()).
    buffer(buffer).
    listener(lexerListener).
    create();

lexer.run();
```

Numéro de ligne/colonne

- Un *LocationTracker* permet de connaître les numéros de ligne/colonne actuelles

```
Reader reader = ...
LocationTracker tracker = new LocationTracker();
LexerBuffer buffer = new ReaderWrapper(reader, tracker);

LexerListener<RuleEnum, TokenBuffer<CharSequence>> lexerListener =
    ...;
SimpleLexer lexer = Builder.lexer(LexerDataTable.createTable()).
    buffer(buffer).
    listener(lexerListener).
    create();

try {
    lexer.run();
} catch (LexingException e) {
    System.out.println("erreur à la ligne " + tracker.getLineNumber());
}
```

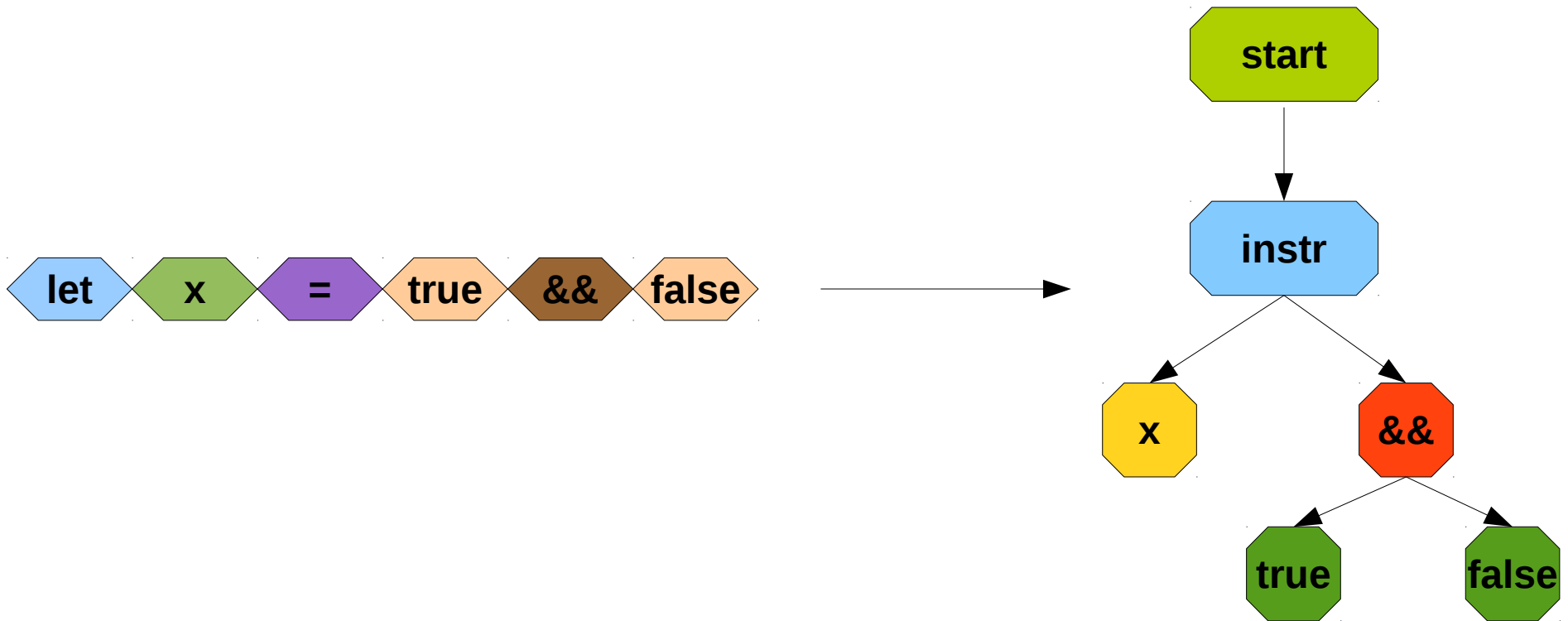
Analyse syntaxique

- Le but est de vérifier qu'un flux de tokens respecte une grammaire algébrique en parcourant l'arbre syntaxique
- Il existe plusieurs algorithmes, LL, LR, SLR, LALR. Tatoo est un analyseur LR et nous utiliserons LALR.

Analyse syntaxique

- A partir d'une grammaire, l'analyse LR produit un automate d'items (i.e. une table d'analyse)
- A partir de cette table, tout flux de tokens est converti en une suite d'opérations : décalage (shift), réduction (reduce) ou reconnaissance (accept)
- La suite d'opérations est alors transformée en arbre (réel ou d'appels de méthodes)

Analyse syntaxique



Grammaire

- Une grammaire est définie par un ensemble de productions
- Chaque production est de la forme
 $\text{NonTerminal} = (\text{Terminal} | \text{NonTerminal})^*$
- Un terminal correspond à la catégorie d'un token

productions:

expr = 'value'

| expr 'plus' expr

;

Terminal (entre ' ')

Non-terminal (sans ")

Grammaire

- Avec Tatoon, elle est spécifiée dans le fichier EBNF
- '*bar*' est un terminal et *baz* est un non-terminal
- Avec foo un terminal ou un non terminal
 - foo? Indique une répétition 0 à 1 fois
 - foo* indique une répétition 0 à n fois
 - foo+ indique une répétition 1 à n fois
 - foo/bar* indique 0..n foo séparés par des bar
 - foo/bar+ indique 1..n foo séparés par des bar

Grammaire dans le fichier ebnf

- **starts:** indique le(s) non-terminal(aux) de départ
- **productions:** indique l'ensemble des productions

Exemple de fichier ebnf

tokens:

```
_true='true'  
_false='false'  
let='let'  
and='&&'  
id='[a-zA-Z]+'
```

starts:

```
start
```

productions:

```
start = instr*  
      ;  
instr = 'let' 'id' 'assign' expr  
      ;  
expr = '_true'  
      | '_false'  
      | expr 'and' expr  
      ;
```

Astuces avec Tatoo

- Les terminaux et non-terminaux utilisés par Tatoo doivent être des identificateurs valides en Java (donc pas le droit à *if*, *else*, *true*, etc.)
- La directive *autoalias* permet d'utiliser l'expression rationnelle comme alias pour un terminal

directives:

autoalias

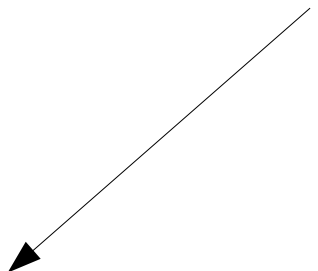
tokens:

plus = '+'

productions:

expr = expr '+' expr
;

On peut alors utiliser '+' à la place de 'plus'



Productions nommées

- Avec Tatoon, les productions sont nommées (entre accolades)
- Si un non-terminal n'a qu'une production, la production peut avoir le même nom que le non-terminal

- **productions:**

```
start = instr*           { start }  
;  
instr = 'let' 'id' '=' expr { instr }  
;  
expr = 'true'           { expr_true }  
      | 'false'          { expr_false }  
      | expr '&&' expr    { expr_and }  
;
```

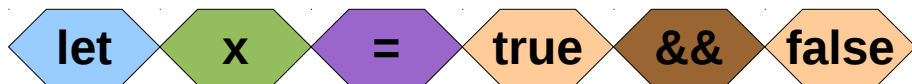
Analyse syntaxique avec productions nommées

- productions:

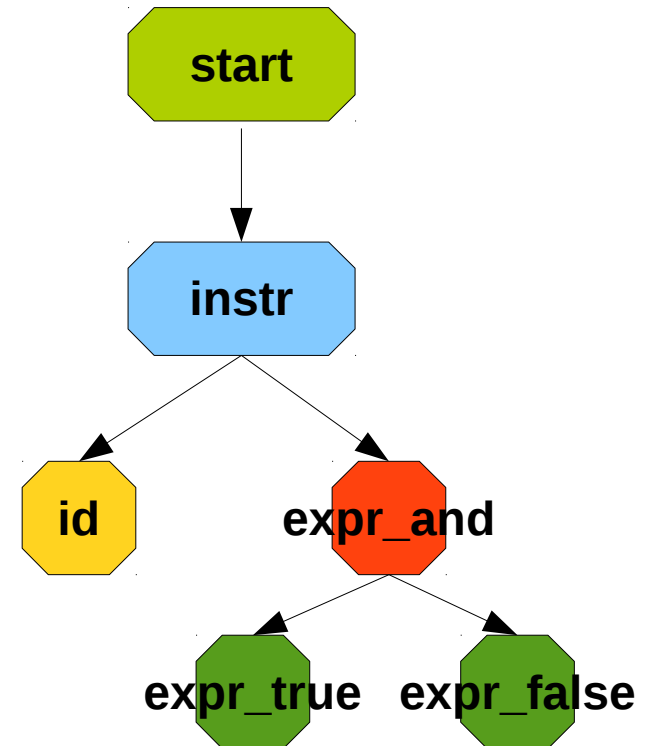
start = instr* { start }

instr = 'let' 'id' '=' expr { instr }

expr = 'true' { expr_true }
 | 'false' { expr_false }
 | expr '&&' expr { expr_and }



analyse



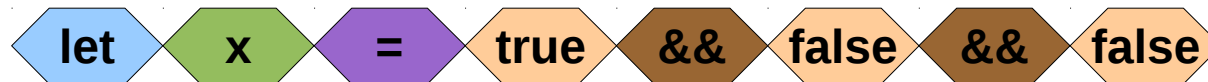
Analyse syntaxique et ambiguïté

- S'il y a deux arbres de dérivation possibles, la grammaire est ambiguë

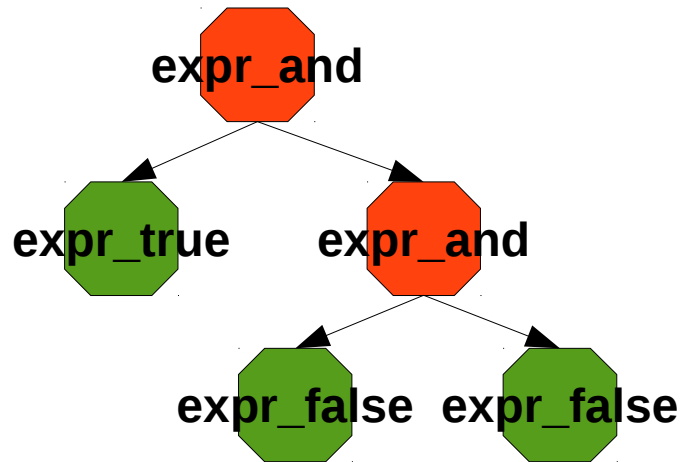
expr = 'true'
 | 'false'
 | expr '&&' expr
 ;

{ expr_true }
{ expr_false }
{ expr_and }

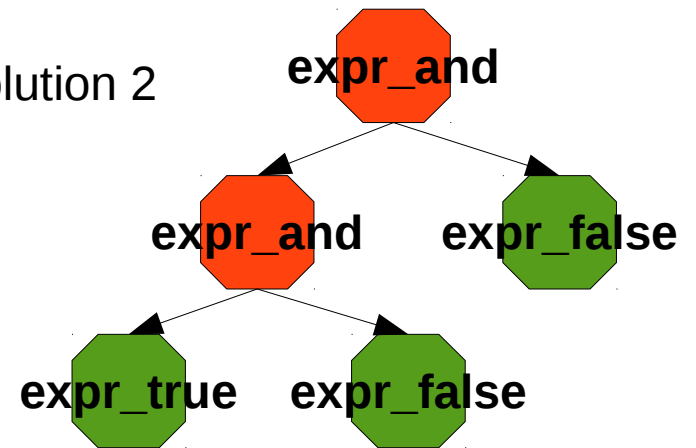
Exemple :



Solution 1



Solution 2



Analyse syntaxique et ambiguïté

- L'analyse LR détecte les ambiguïtés
- Avec Tatoon: shift/reduce conflict state 9:

state9 - state 0: let id = true && true

Compatible versions : DEFAULT

Kernel items	Actions
<u>expr</u> ::= <u>expr</u> && <u>expr</u> • <u>expr</u> ::= <u>expr</u> • && <u>expr</u>	let : <u>reduce</u> by <u>expr</u> ::= <u>expr</u> && <u>expr</u> __eof__ : <u>reduce</u> by <u>expr</u> ::= <u>expr</u> && <u>expr</u> and : shift to <u>state8</u> , <u>reduce</u> by <u>expr</u> ::= <u>expr</u> && <u>expr</u> branch : <u>reduce</u> by <u>expr</u> ::= <u>expr</u> && <u>expr</u>

- Deux solutions:
 - Ré-écrire la grammaire (cf. cours analyse syntaxique)
 - Mettre des priorités

Priorité et associativité avec tatoo

- Lorsqu'il y a un conflit décalage/réduction, on compare les priorités : la plus grande priorité est choisie
- Puis on applique l'associativité parmi
 - left $((\text{expr} \ \&\& \ \text{expr}) \ \&\& \ \text{expr})$
 - right $(\text{expr} \ \&\& \ (\text{expr} \ \&\& \ \text{expr}))$
 - nonassoc (non associatif)

Conflit entre opérateurs

- ```
expr = 'true'
 | 'false'
 | expr '&&' expr
 | expr '||' expr
 ;
```
- Exemple: `true || false && true`
- 2 arbres possibles:
  - `(true || false) && true`
  - `true || (false && true)`

# Conflit résolu par priorité

'&&' est plus prioritaire que '||'

## priorities:

boolean\_or = 1 left

boolean\_and = 2 left

## tokens:

and = '&&'

or = '||'

## productions:

expr = 'true'

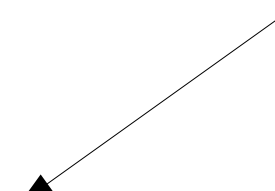
| 'false'

| expr '&&' expr [boolean\_and]

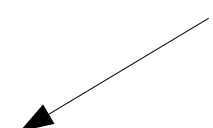
| expr '||' expr [boolean\_or]

;

Priorité donnée lors  
d'un décalage par  
&&

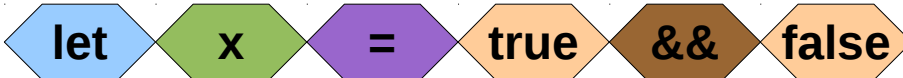


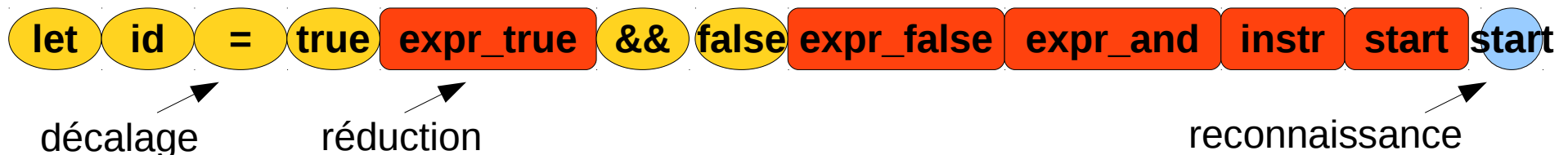
Priorité donnée lors  
d'une réduction par  
Expr = expr '&&' expr



# Sortie d'un analyseur syntaxique LR

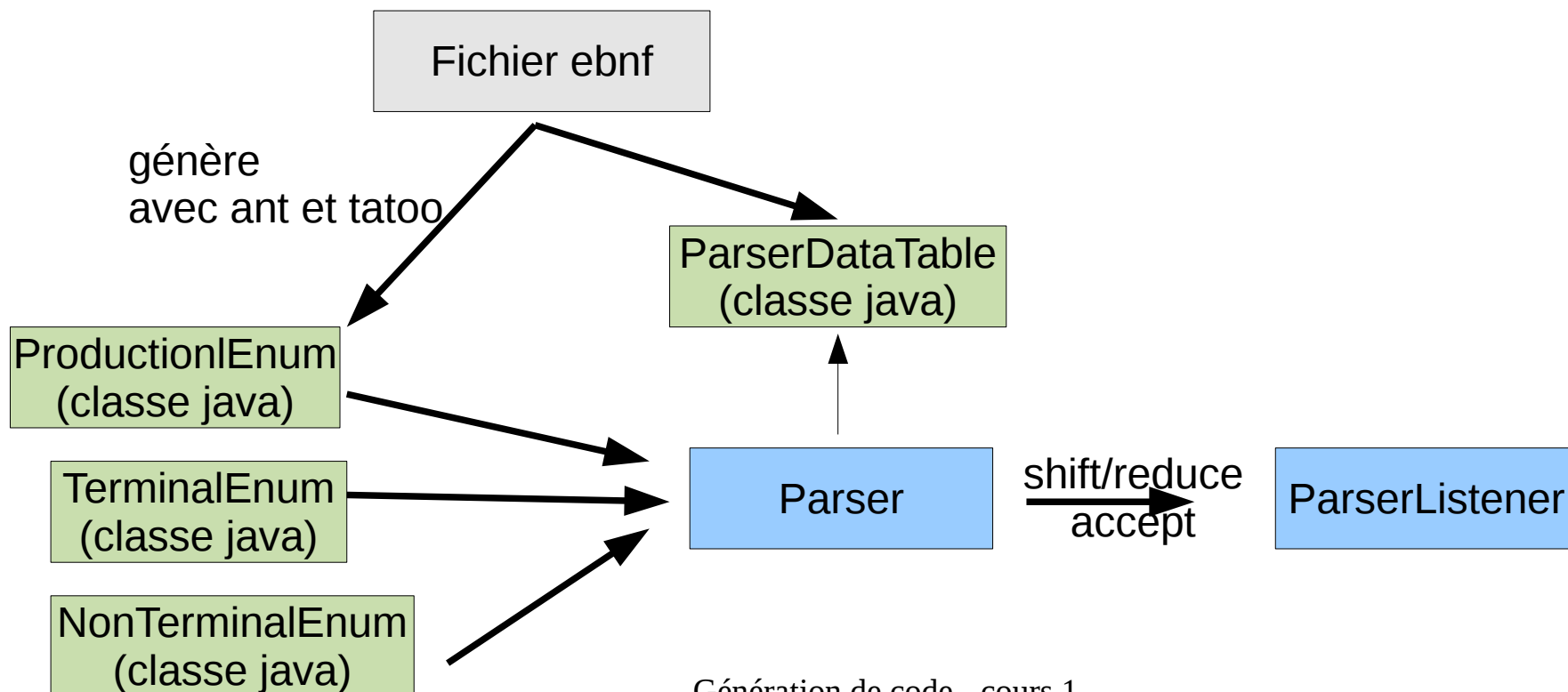
```
start = instr* { start }
;
instr = 'let' 'id' '=' expr { instr }
;
expr = 'true' { expr_true }
 | 'false' { expr_false }
 | expr '&&' expr [boolean_and] { expr_and }
;
```

Exemple: 



# L'analyseur syntaxique (Parser) de Tatoo

- L'automate LR est encodé dans la classe ParserDataTable.



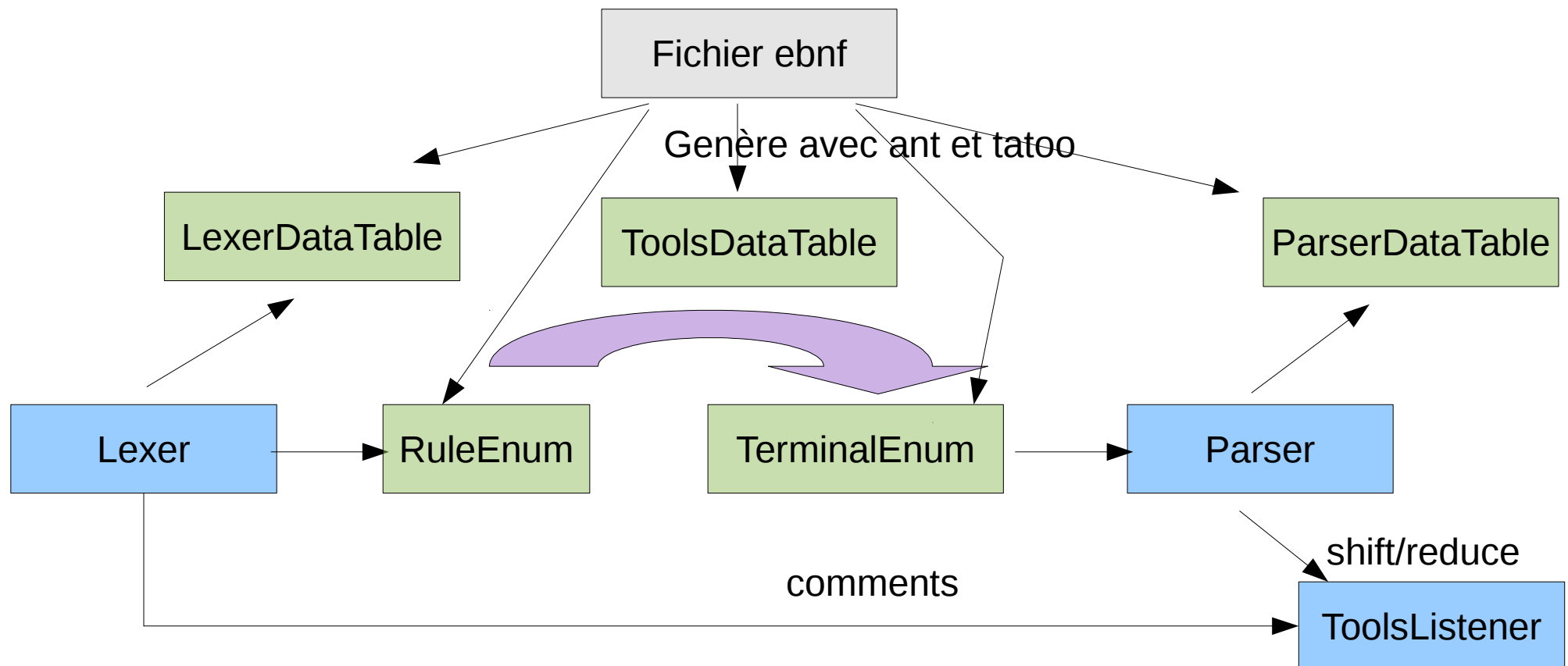
# Analyse syntaxique avec Tatoo

- On utilise un listener pour être averti des actions effectuées par l'analyseur

```
ParserListener<TerminalEnum, NonTerminalEnum, ProductionEnum>
parserListener =
new ParserListener<TerminalEnum, NonTerminalEnum, ProductionEnum>() {
 public void shift(TerminalEnum terminal) {
 // shift d'un terminal
 }
 public void reduce(ProductionEnum production) {
 // reduce d'une production
 }
 public void accept(NonTerminalEnum nonTerminal) {
 // accept d'un non terminal
 }
};
SimpleParser<TerminalEnum> parser =
 Builder.parser(ParserDataTable.createTable()).
 listener(parserListener).
 create();
```

# Analyseur = Lexer + Parser

- On associe les règles du lexer aux terminaux du parser



# Analyseur avec Tatoo

```
Reader reader = ...
```

```
ToolsListener<RuleEnum, TokenBuffer<CharSequence>, TerminalEnum,
NonTerminalEnum, ProductionEnum> toolsListener = ...;
```

```
Builder.analyzer(LexerDataTable.createTable(),
 ParserDataTable.createTable(),
 ToolsDataTable.createToolsTable()).
 reader(reader).
 listener(toolsListener).
 create().
 run();
```

# Analyseur avec Tatoo

```
ToolsListener<RuleEnum, TokenBuffer<CharSequence>, TerminalEnum,
NonTerminalEnum, ProductionEnum> toolsListener =
 new ToolsListener<RuleEnum, TokenBuffer<CharSequence>, TerminalEnum,
 NonTerminalEnum, ProductionEnum>() {
 public void comment(RuleEnum rule, TokenBuffer<CharSequence> buffer) {
 // commentaire
 }
 public void shift(TerminalEnum terminal, RuleEnum rule,
 TokenBuffer<CharSequence> buffer) {
 // shift
 }
 public void reduce(ProductionEnum production) {
 // reduce
 }
 public void accept(NonTerminalEnum nonTerminal) {
 // accept
 }
 };
```

# Évaluateur à la main

- Le ToolsListener permet d'écrire un évaluateur
- Besoin d'une pile de valeur (ici des booléens)
- Evaluation:
  - On empile *true* ou *false* lors d'un shift correspondant
  - Lors d'un reduce *expr\_and*, on dépile deux valeurs, on effectue l'opération et on empile le résultat
  - Lors d'un reduce de *instr*, on dépile la valeur et on affiche le résultat

# Evaluateur à la main

let id = true

|      |
|------|
| true |
|------|

let id = true **expr\_true** && false

|       |
|-------|
| false |
| true  |

let id = true **expr\_true** && false **expr\_false** **expr\_and**

|       |
|-------|
| false |
|-------|

← true && false

let id = true **expr\_true** && false **expr\_false** **expr\_and** **instr**

\_\_\_\_\_ print false

# Evaluateur à la main

```
new ToolsListener<RuleEnum, TokenBuffer<CharSequence>, TerminalEnum,
 NonTerminalEnum, ProductionEnum>() {
 public void shift(EvalTerminalEnum terminal, EvalRuleEnum rule,
 TokenBuffer<CharSequence> buffer) {
 if (rule == EvalRuleEnum._true || rule == EvalRuleEnum._false) {
 stack.push(Boolean.parseBoolean(buffer.view().toString()));
 }
 }
 public void reduce(EvalProductionEnum production) {
 switch(production) {
 case expr_and: {
 boolean secondValue = stack.pop();
 boolean firstValue = stack.pop();
 stack.push(firstValue && secondValue);
 break;
 }
 case instr:
 System.out.println("value: "+stack.pop());
 break;
 default:
 }
 } ... };
```