

# Lexing, Parsing & Evaluation

Rémi Forax

# Plan

- Intro
- Rappel
- Tatoo
- Regex
- Algo LR
- Priorité
- Token erreur
- Evaluation 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

# Langages Informatique

- Deux sortes de language informatique :
- GPL (general purpose language)
  - C, Java, Python, Perl, PHP, Ruby, etc.
- DSL (domain specific language)
  - Fichier ini
  - SQL

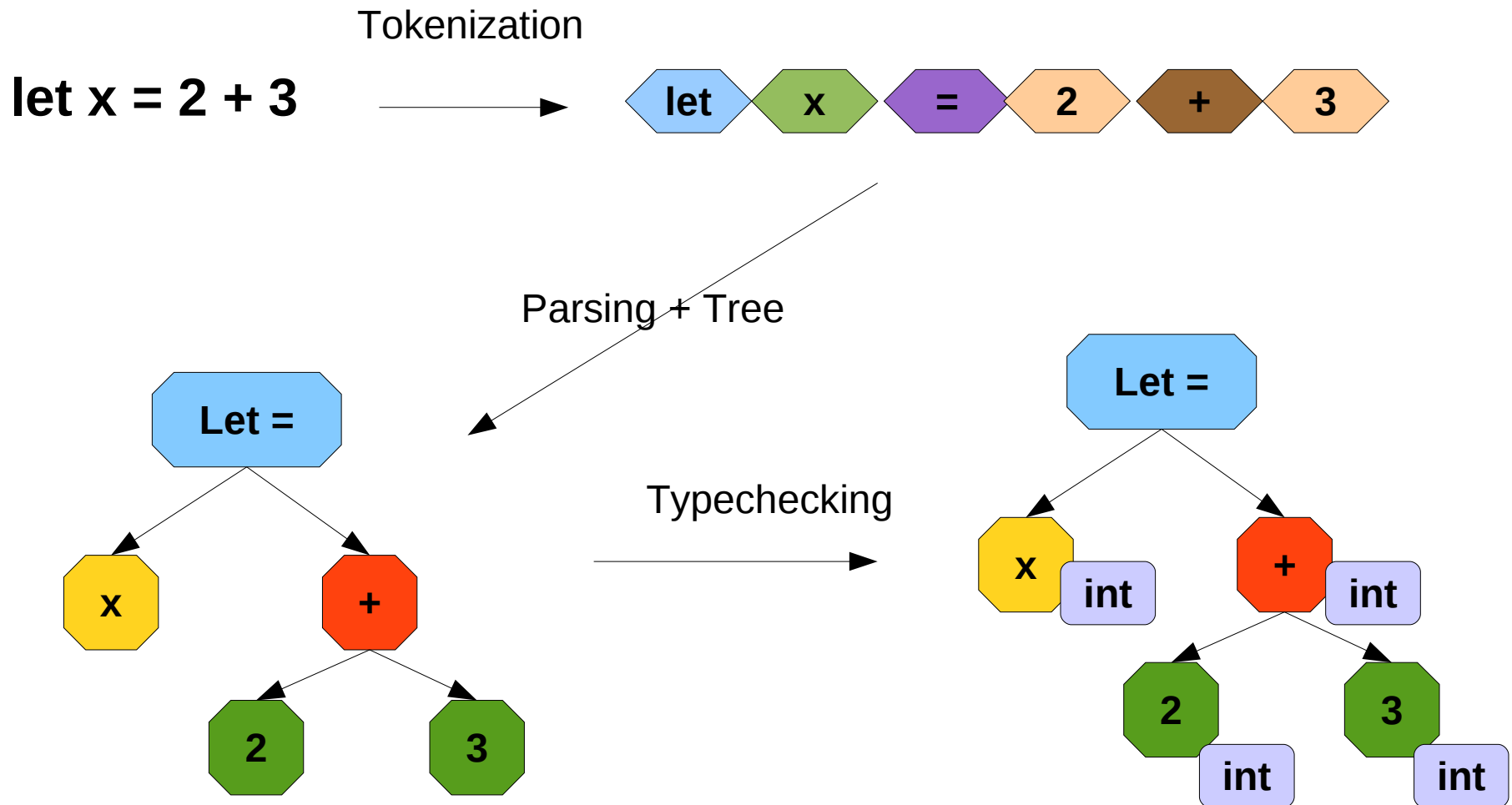
# Compilateurs de Languages

- Pour les GPL, le compilateur est souvent écrit avec le GPL (après bootstrap)  
Par ex: compilateur Java est écrit en Java
- Pour les DSL, le compilateur est écrit avec un GPL

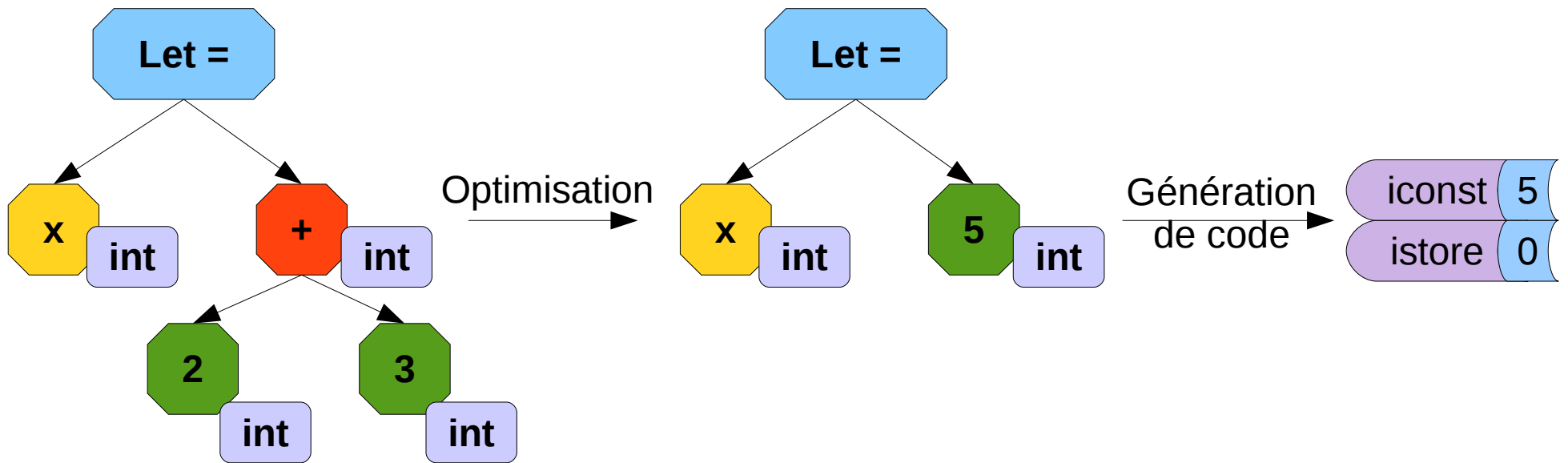
# Etapes lors de la compilation

- Un compilateur suit toujours les mêmes étapes
  - **Tokenisation** (transforme un fichier en mots token)
  - **Parsing** (transforme une suite de mots en flot de shift/reduce)
  - **Construction d'arbre** (le flot de shift/reduce crée un arbre AST)
  - **Passes sémantiques** (enter, attrib(typage), flow, desugar)
  - **Génération de code** (l'AST est transformé en code)
    - Passe intermédiaire en code 3 addresses

# Etapes lors de la compilation



# Etapes lors de la compilation





# Pourquoi étudier la compilation ?

- Savoir écrire des parseurs facilement
  - XML c'est bien mais écrire  $2 + 3 * 7$  en XML, c'est lourd
- Ecrire un compilateur est le meilleur moyen de comprendre les compilateurs
- Savoir comment écrire du code, quelles sont les optimisations qui sont couramment utilisées ?

# Générateur de parseur

- En C, lex/flex (lexer) et yacc/bison (parser).
- En Java, JavaCC(LL), ANTLR(LL) et JFlex/JCup(LR), SableCC(LR).
- ANTLR est le plus utilisé (de loin)
  - Backends dans plein de langages
  - Problèmes:
    - mix la grammaire et le langage cible
    - Analyse LL seulement

# Tatoo

- Générateur de lexer+parseur maison
- Ecrit en Java, la version courante est la 4.2
- Utilisé en IR depuis 4 ans et depuis plus longtemps en recherche
- Séparation propre entre les différentes informations pour le lexing et le parsing
- Construction de l'AST automatique

# Lexer

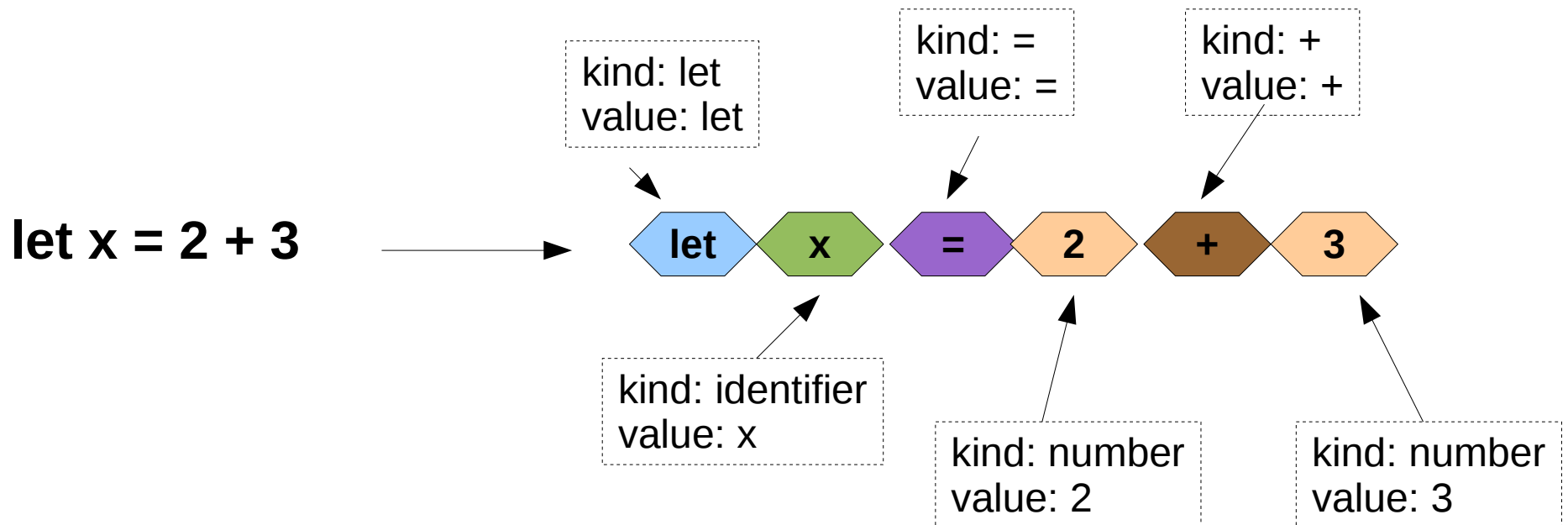
- On utilise un lexer qui à partir d'un ensemble d'expression régulière trouve quel est le token correspondant
- La tokenization fourni 2 informations par token
  - Le type du token reconnu
  - La valeur de ce token
    - la chaîne de caractère correspondant

# Analyse lexical

- Simplifie l'écriture de grammaire
- Le texte est découpé en lexèmes (*tokens*)
- Le lexer supprime
  - Les espaces
  - Les commentaires
  - Les retours à la ligne (suivant le langage), etc

# Tokenization

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



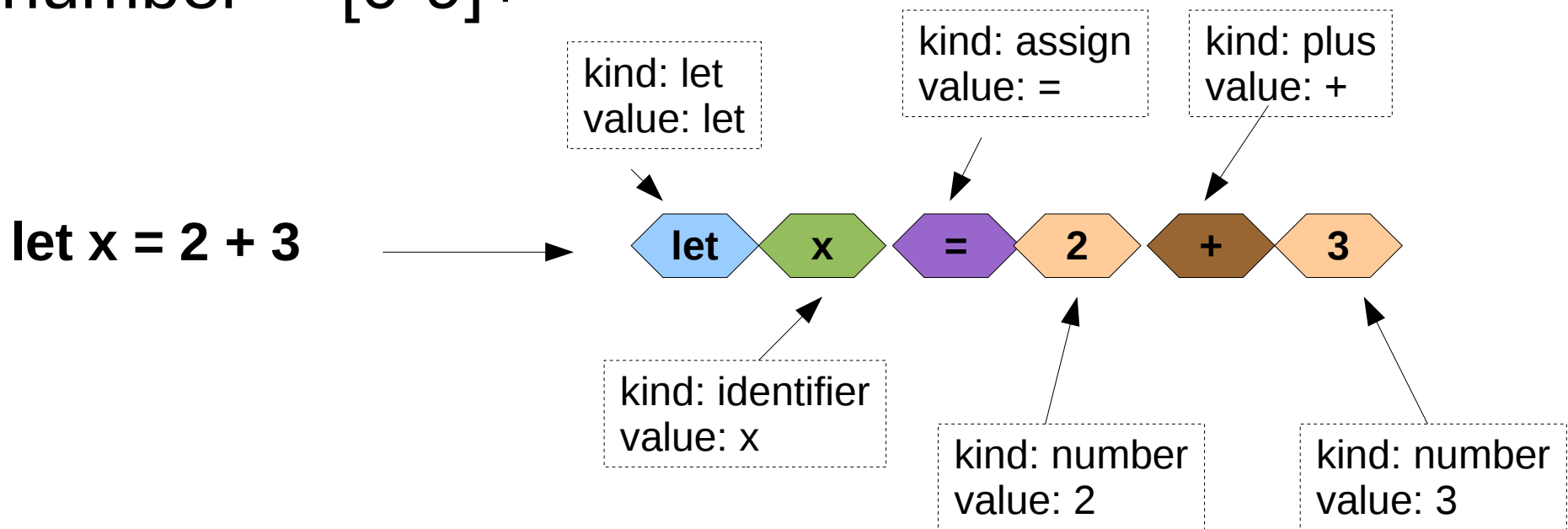
# Catégories

- Catégories les + fréquentes :
  - Chaque mot-clef du langage (*keyword*)
  - Chaque symbole de ponctuation ou opérateur
  - Chaque sortes de littérales:
    - Les entiers et autres valeurs numériques
    - Les chaines de caractères
  - Les identificateurs (*identifier*), c-a-d les noms des types, des variables, des fonctions etc.

# Tokenization avec Tatoon

**tokens:**

let  $\hat{=}$  'let' kind  
assign  $\hat{=}$  '=' expression rationnelle (regex)  
plus  $\hat{=}$  'plus'  
identiflier  $\hat{=}$  '[A-Za-z]+'





# Format des regex de Tatoo

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

# Exemples de regex

- Exemple de regex
  - Mot-clefs : 'if', 'for', 'return'
  - Opérateurs : '+', '-', '\\*'
  - Entiers : '[0-9]+|(0x[0-9A-Fa-f]+)'
  - Caractères : '"[^"]\*"'
  - Chaîne de caractères : '"[^"]\*"'
  - Identificateurs : '[A-Za-z\_][0-9A-Za-z\_]\*'

# Lexing

- Le lexer lance la reconnaissance sur tous les automates en parallèle
- Si plusieurs automates match, on continue jusqu'à ce que plus aucun ne match
  - On prend celui qui reconnaît le + long motif
  - Si 2 automates reconnaissent le motif de même longueur, on prend celui qui est déclaré en premier dans l'ordre du fichier

# Tatoo a différent types de tokens

- Les tokens (**tokens:**) qui correspondent à des mot-clefs
- Les blancs (**blanks:**) qui sont des espaces qui ne seront pris en compte ni par le parseur ni par l'analyser
- Les commentaires (**comments:**) qui seront envoyé à l'analyser et pas pris en compte par le parseur

# Exemple

- Attention, les sections doivent être déclarés dans cet ordre

## **tokens:**

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

## **blanks:**

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

## **comments:**

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

# Lexing

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
---	---	---	--	---	---	---	---	---

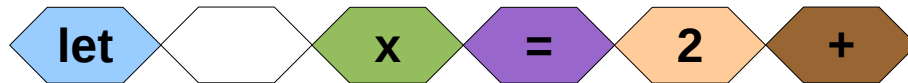
let

# Lexing (suite)

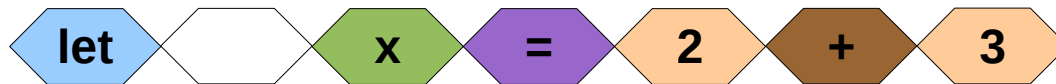
l	e	t		x	=	2	+	3
---	---	---	--	---	---	---	---	---



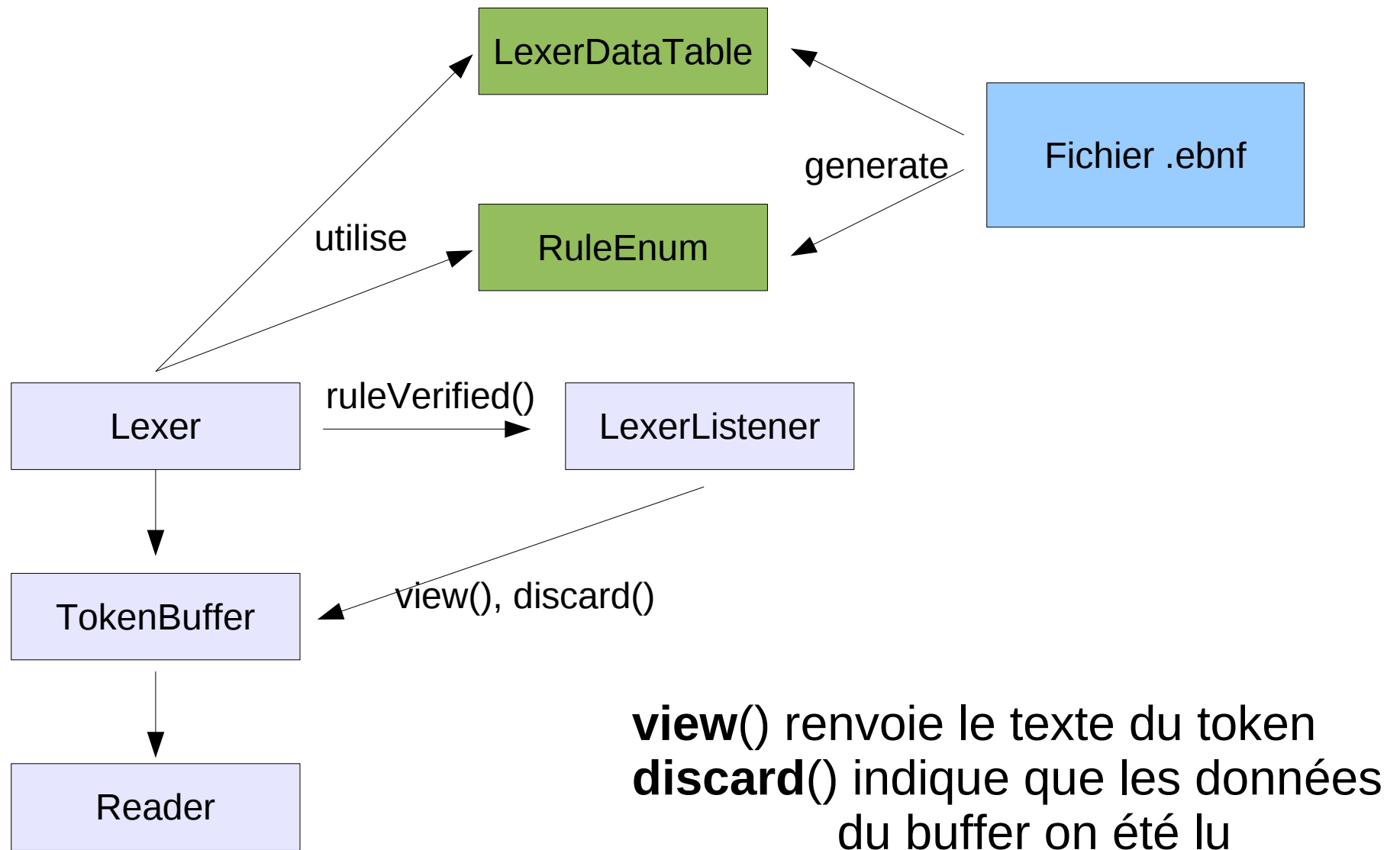
l	e	t		x	=	2	+	3
---	---	---	--	---	---	---	---	---



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



# Fonctionnement du lexer de Tatoo





# Lexer avec Tadoo

```
<?xml version="1.0"?>
<project name="eval" default="all" basedir=".">
  <property name="tadoo-build.dir" location="../build-lib"/>
  <property name="tadoo.jar" location="${tadoo-build.dir}/tadoo.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.tadoo.cc.ebnf.main.EBNFTask"
      classpath="${tadoo.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>
```



# Buffer de Tatoon

- Tatoon utilise une interface LexerBuffer pour discuter avec les flux de caractères ou d'entiers en entrée
- Cette interface permet de ne pas avoir le contenu totale d'un fichier pour effectuer le lexing.
- Seul la partie nécessaire au lexing 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());
}
```

# Pour aller plus loin

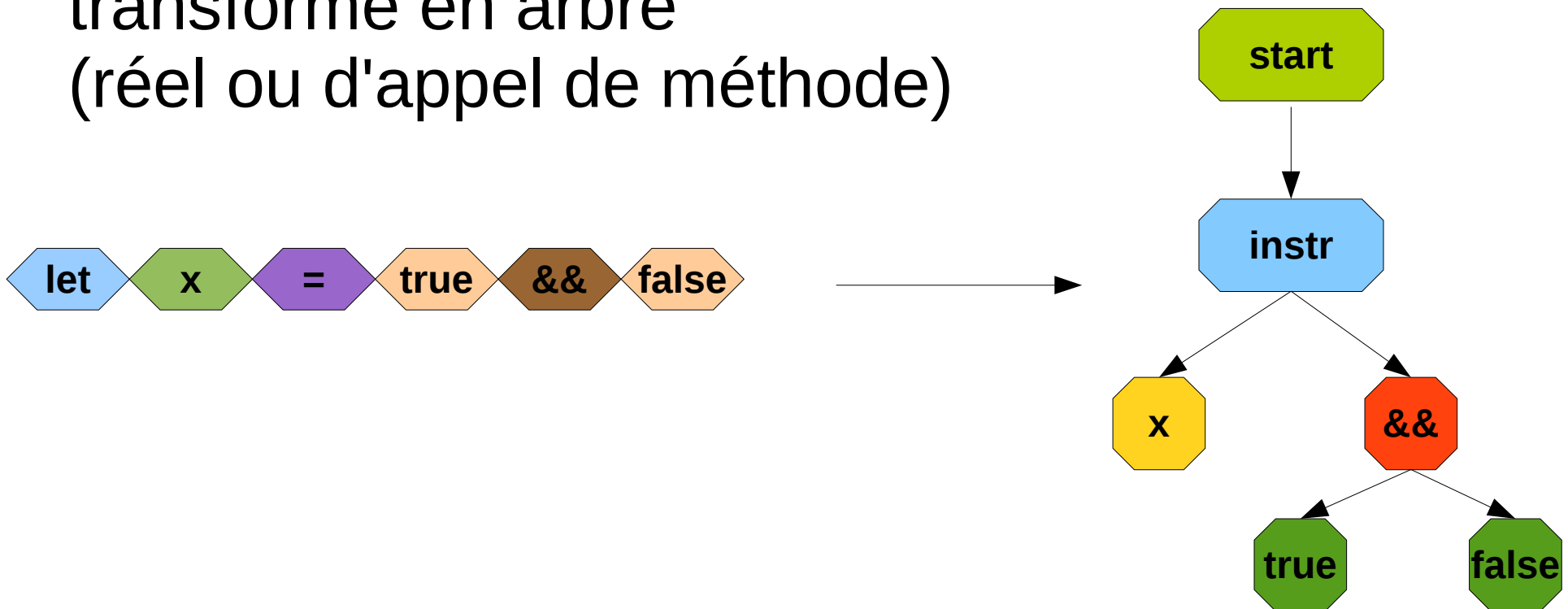
- Tatoon possède un mécanisme qui permet de restreindre les règles actives
- Si le lexer est branché sur un parseur, seuls les règles correspondants à des terminaux attendus par le parseur sont sélectionnés.
- Si il n'y a pas de parseur, toutes les règles sont sélectionnées
- Il est possible d'écrire un sélectionneur de règle en implantant l'interface RuleActivator.

# Analyse syntaxique

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

# Parsing

- On utilise un parseur LR qui transforme un flux de tokens en une suite de shift/reduce/accept
- La suite de shift/reduce/accept est alors transformé en arbre (réel ou d'appel de méthode)





# Le parsing

- A partir d'une grammaire, l'analyse LR produit un automate
- Pour faire fonctionner l'automate, il faut une pile qui stocke les états de l'automate
  - lors d'un shift, on empile l'état courant
  - Lors d'un reduce, on supprime de la pile tous les états correspondant à la production réduite et on shift en utilisant la table des gotos

# Grammaire

- Une grammaire est définie par un ensemble de productions
- Chaque production est de la forme  
 $\text{NonTerminal} = (\text{Terminal} | \text{NonTerminal})^*$

- **productions:**

expr = 'value'  
| expr '+' expr  
;

Terminal (entre ' ')

Non-terminal (sans ")

# Grammaire

- Le format de spécification de la grammaire avec Tatoon est ENBF (un peu modifié)
- 'bar' est un terminal
- 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 séparé par des bar
  - foo/bar+ indique 1..n foo séparé par des bar

# Grammaire

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

**starts:**

start

**productions:**

start = instr\*

;

instr = 'let' 'id' '=' expr

;

expr = 'true'

| 'false'

| expr '&&' expr

;

# Astuce avec Tatoo

- Les terminaux et non-terminaux utilisés par Tatoo doivent être des identificateurs valides en Java
  - donc pas le doit à if, else, true, etc
- La directive autoalias permet d'utiliser la regex comme alias pour un terminal

## **directives:**

autoalias

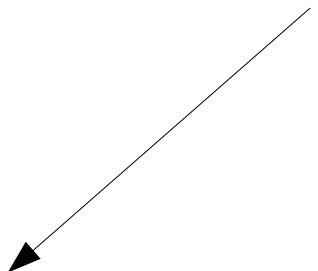
## **tokens:**

plus = '+'

## **productions:**

expr = expr '+' expr  
;

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



# Astuce avec Tatoon

- Par défaut, Tatoon requiert que les terminaux soit déclarés
- La directive autotoken déclare automatiquement les terminaux

## **directives:**

autotoken

## **productions:**

expr = expr 'plus' expr  
;

Pas besoin de déclarer le terminal plus



- Cela permet de tester des grammaires facilement

# Grammaire

- Avec Tatoon, les productions sont nommées (entre accolade)
- 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 }
;	

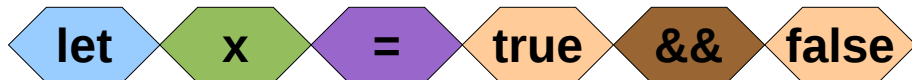
# Parsing

- productions:**

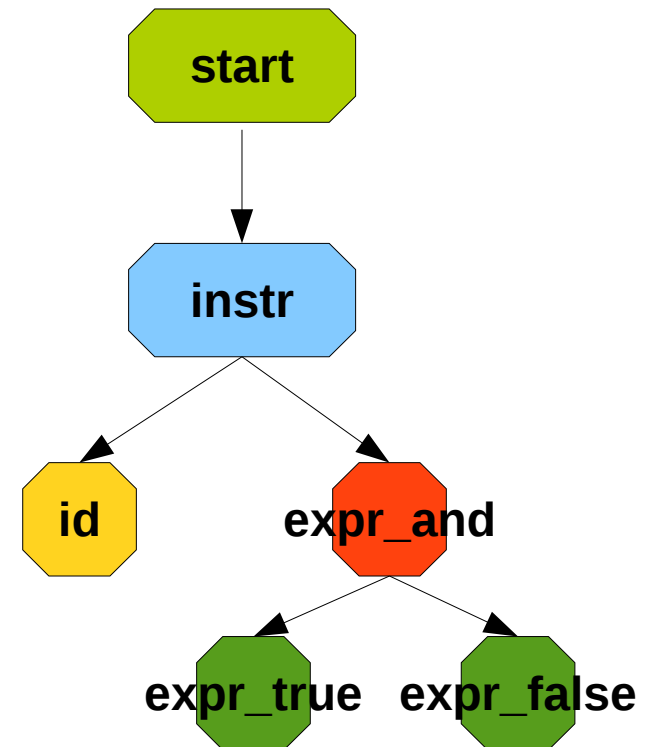
start = instr\*                   { start }

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

expr = 'true'                   { expr\_true }  
      | 'false'                 { expr\_false }  
      | expr '&&' expr         { expr\_and }



parsing →

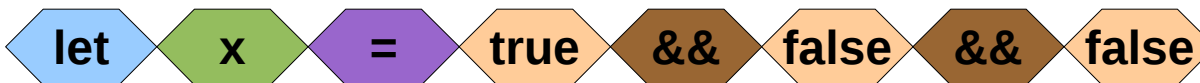




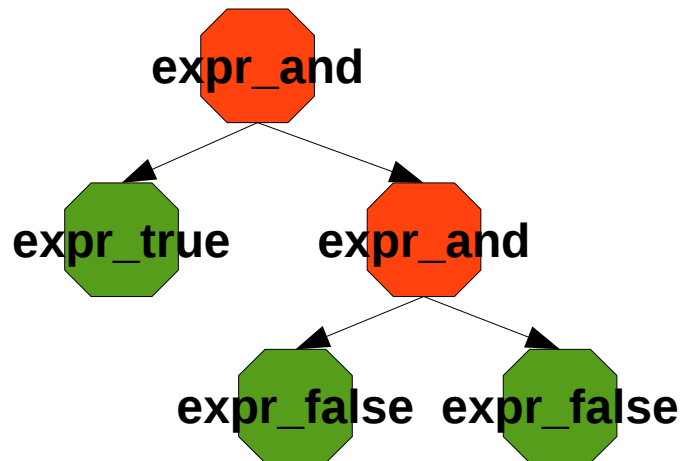
# Parsing & ambiguïté

- il y a deux arbres de dérivation possible => grammaire ambiguë

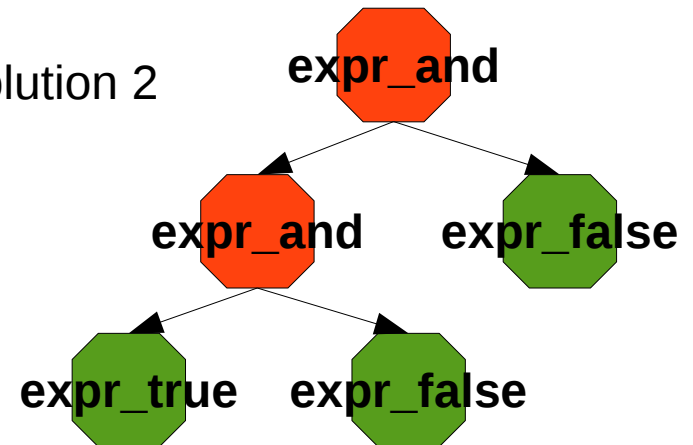
expr = 'true'	{ expr_true }
'false'	{ expr_false }
expr '&&' expr	{ expr_and }
;	

Exemple : 

Solution 1



Solution 2



# Parsing & ambiguïté

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

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

Compatible versions : DEFAULT

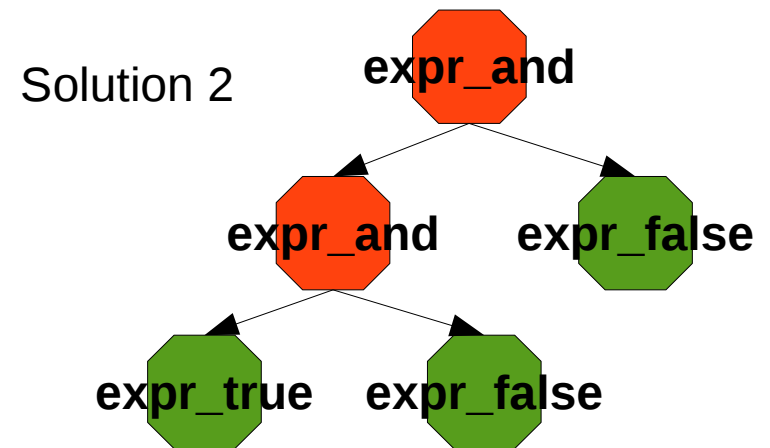
Kernel items	Actions
<u>expr</u> ::= <u>expr</u> && <u>expr</u> • <u>expr</u> ::= <u>expr</u> • && <u>expr</u>	let : <u>reduce</u> by <u>expr</u> ::= <u>expr</u> && <u>expr</u> __eof__ : <u>reduce</u> by <u>expr</u> ::= <u>expr</u> && <u>expr</u> and : shift to <u>state8</u> , <u>reduce</u> by <u>expr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

# Conflit LR shift/reduce

- Ré-écrire la grammaire:

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



- On indique dans la grammaire que la récursivité se fait vers la gauche  
=> Augmente le nombre d'état de l'automate LR

# Priorité et associativité

- Lorsqu'il y a un conflit shift/reduce
- On compare la priorité, 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 = '&&'

[boolean\_and]

or = '||'

[boolean\_or]

## **productions:**

expr = 'true'

| 'false'

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

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

;

# Conflicts if/else

- `instr = 'if' '(' expr ')' instr`  
    | `'if' '(' expr ')' instr 'else' instr`  
    ;  
  
• Exemple: `if (expr) if (expr) else ...`  
    (le else est celui du premier ou du second if ?)

state11 - state 0: `if ( id ) print id`

Compatible versions : DEFAULT

| Kernel items                                                                                                                                                              | Actions                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code><u>instr</u> ::= <i>if</i> ( <u>expr</u> ) <u>instr</u> • <i>else</i> <u>instr</u></code><br><code><u>instr</u> ::= <i>if</i> ( <u>expr</u> ) <u>instr</u> •</code> | <code>_else</code> : shift to <a href="#">state12</a> , <a href="#">reduce</a> by <code><u>instr</u> ::= <i>if</i> ( <u>expr</u> ) <u>instr</u></code><br><code>_eof_</code> : <a href="#">reduce</a> by <code><u>instr</u> ::= <i>if</i> ( <u>expr</u> ) <u>instr</u></code><br><code>print</code> : <a href="#">reduce</a> by <code><u>instr</u> ::= <i>if</i> ( <u>expr</u> ) <u>instr</u></code><br><code>_if</code> : <a href="#">reduce</a> by <code><u>instr</u> ::= <i>if</i> ( <u>expr</u> ) <u>instr</u></code><br><code>branch</code> : <a href="#">reduce</a> by <code><u>instr</u> ::= <i>if</i> ( <u>expr</u> ) <u>instr</u></code> |

# Resolution conflits if/else

- Conflit entre le shift de else et le reduce if (expr) instr
- if...else est associatif à gauche

- **priorities:**

ifelse = 1 left

**tokens:**

\_else = 'else'

[ifelse]

**productions:**

instr = 'if' '(' expr ')' instr

[ifelse]

| 'if' '(' expr ')' instr 'else' instr

;



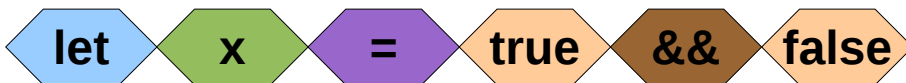
# Conflit Reduce/Reduce

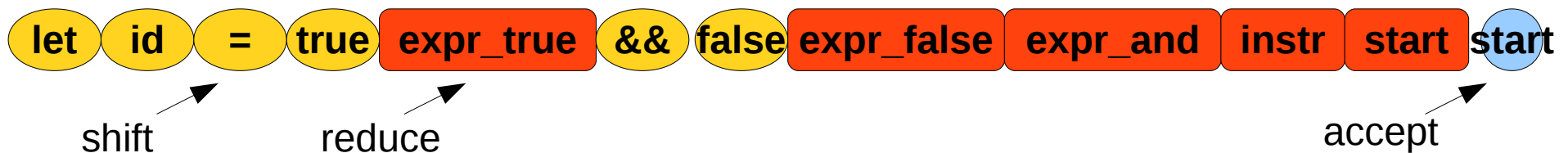
- Contrairement au conflit shift/reduce, l'erreur ne peut pas être résolu avec des priorités
- Il s'agit d'erreur qui montre que la grammaire est mal écrite, il y faut modifier la grammaire

# Parsing LR

- 3 instructions possibles: shift terminal, reduce production, accept non terminal

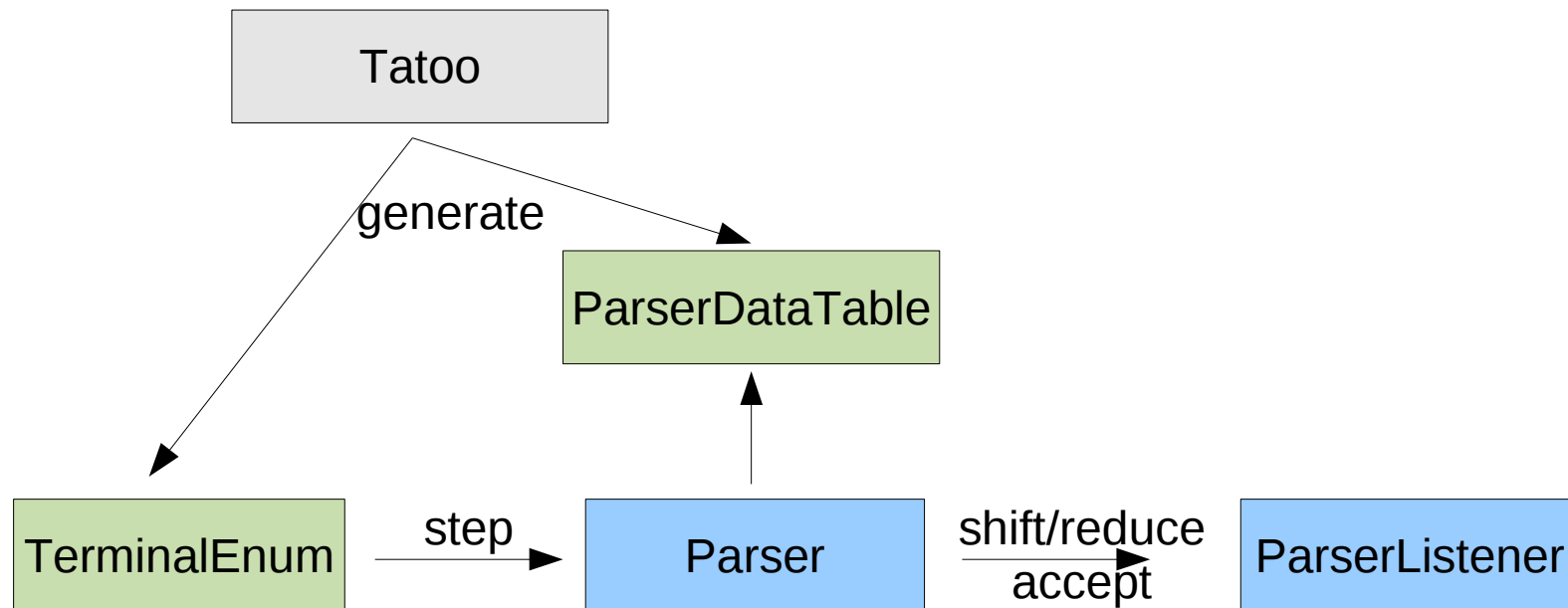
|                                  |                |
|----------------------------------|----------------|
| start = instr*                   | { start }      |
| ;                                |                |
| instr = 'let' 'id' '=' expr      | { instr }      |
| ;                           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

Exemple: 



# Parser de Tatoo

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



# Parseur avec Tatoo

- On utilise un listener pour être averti des actions effectuées par le parseur

```
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();
```

# Parseur avec Tatoo

- On envoie les terminaux au parseur avec la méthode `step`. On appelle `close` pour dire qu'il n'y aura plus d'autres terminaux.

```
SimpleParser<TerminalEnum> parser = ...  
parser.step(TerminalEnum.let);  
parser.step(TerminalEnum.id);  
parser.step(TerminalEnum.assign);  
parser.step(TerminalEnum._true);  
parser.step(TerminalEnum.and);  
parser.step(TerminalEnum._false);  
parser.close();
```

```
reduce instr_star_0_empty  
shift let  
shift id  
shift assign  
shift _true  
reduce expr_true  
shift and  
shift _false  
reduce expr_false  
reduce expr_and  
reduce instr  
reduce instr_star_0_rec  
reduce start  
accept start
```

# La reprise sur erreur

- Il y a deux façons de faire de la reprise sur erreur avec un analyseur LR
  - Prévoir un terminal spécifique dans la grammaire
  - Utiliser un algorithme automatique qui lors d'une erreur va tenter d'insérer/supprimer des terminaux
- Pour avoir une reprise sur erreur efficace il faut effectuer une action différentes en fonction de l'état LR (voir de la pile d'états)

# Le terminal 'error'

- On ajoute dans la grammaire des productions utilisant un faux terminal 'error'

**errors:**

error

**productions:**

```
expr = 'value'  
      | expr '&&' expr  
      | '(' error ')'  
      ;
```

Si une erreur se produit après '(', le parseur va supprimer les terminaux intermédiaire jusqu'à pouvoir shifter le ')' correspondant

# Les ';' dans un langage

- En C, on utilise des ';' à la fin des instructions pour faciliter la reprise sur erreur
  - ```
instr = print expr ';'
      | 'id' '=' expr ';'
      | 'error' ';'
      ;
```

Si il y a une erreur, on supprime les terminaux jusqu'à trouver un ';'.

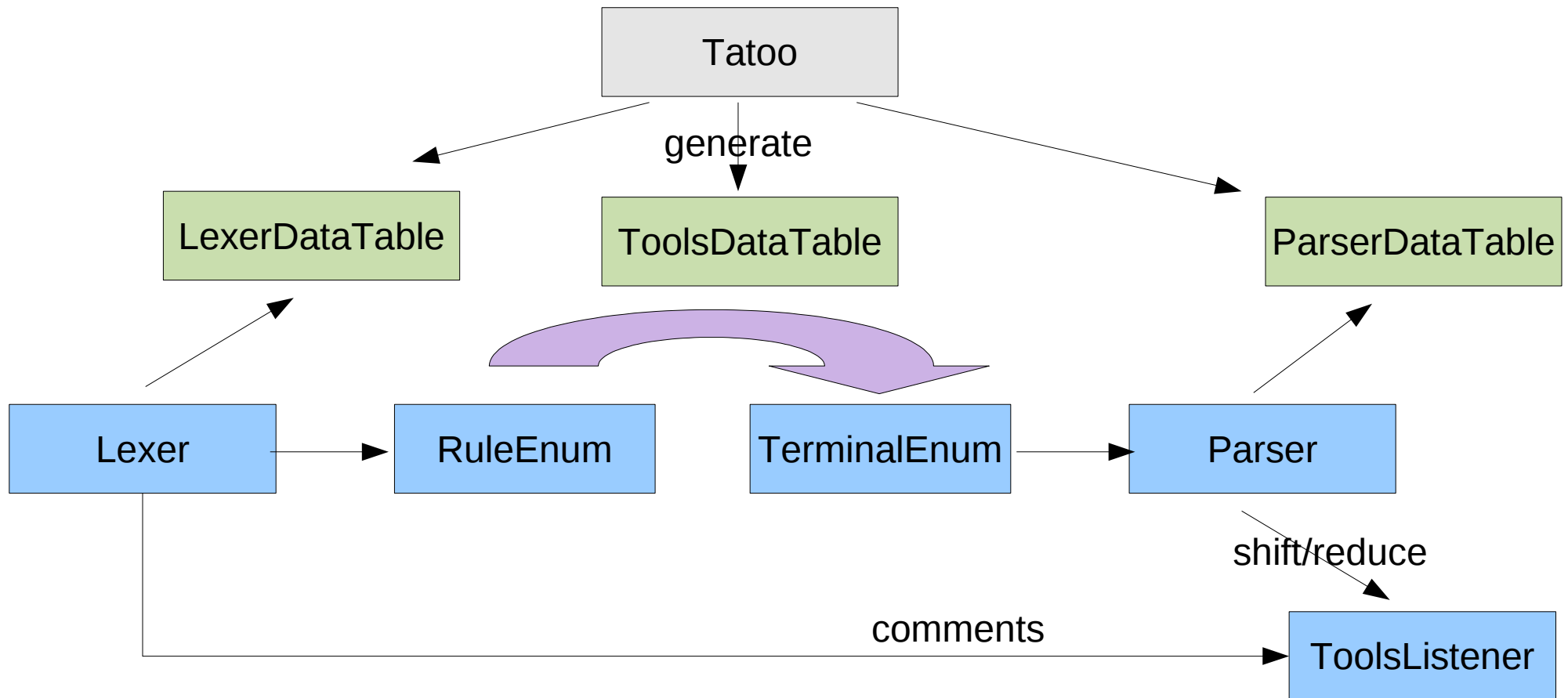


# Algorithme de reprise sur erreur

- Si une erreur arrive:
  - On dépile les états jusqu'à trouver un état qui accepte une action quand le lookahead est 'error'
    - Cela revient à supprimer tous les terminaux pris en compte jusqu'à '(' par exemple
  - On effectue les actions jusqu'à shifter 'error'
  - On lit les terminaux suivants jusqu'à voir le terminal après 'error'
    - par exemple ')' ou ';'

# Analyzer = Lexer + Parser

- On associe les rules du lexer aux terminaux du parser



# Analyzer 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();
```

# Analyzer avec Tatoon

```
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  
        }  
    };
```

# Evaluateur 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:
        }
    }
} ... };
```

# Evaluateur automatique

- Il est possible avec Tatoo d'indiquer un type pour les terminaux et les non terminaux
- Tatoo gère alors la pile des valeurs tout seul

types:

```
'true':boolean  
'false':boolean  
expr:boolean
```

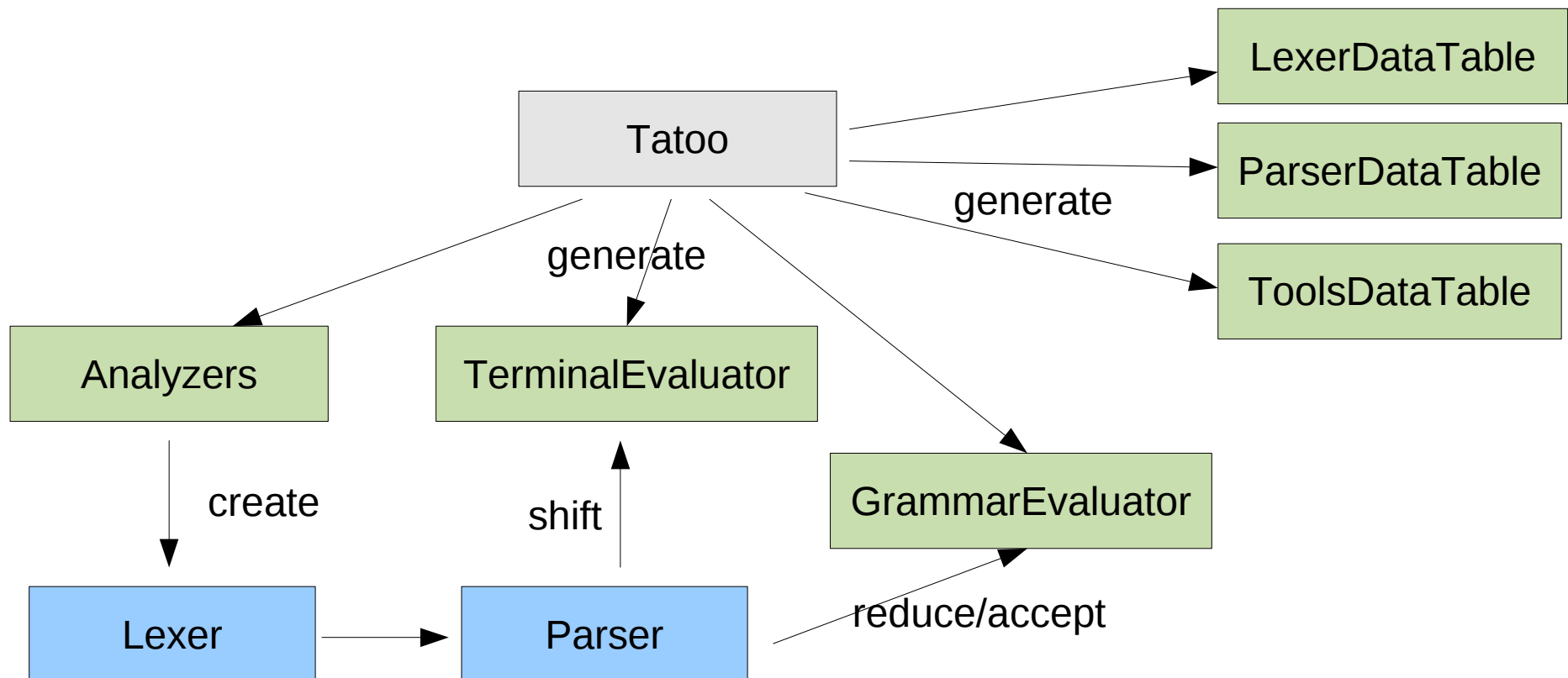
productions:

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



# Evaluateur de Tatoo

- Tatoo génère un évaluateur de terminaux et un évaluateur de productions



# Evaluateur automatique

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

```
TerminalEvaluator<CharSequence> terminalEvaluator = ...
```

```
GrammarEvaluator grammarEvaluator = ...
```

```
Analyzers.run(reader,  
               terminalEvaluator,  
               grammarEvaluator,  
               null, // start non terminal  
               null); // version
```

# TerminalEvaluator

- Indique la valeur de chaque terminal typé

```
TerminalEvaluator<CharSequence> terminalEvaluator =  
    new TerminalEvaluator<CharSequence>() {  
        public void comment(CharSequence data) {  
            // do nothing  
        }  
        public boolean _false(CharSequence data) {  
            return false;  
        }  
        public boolean _true(CharSequence data) {  
            return true;  
        }  
    };  
};
```

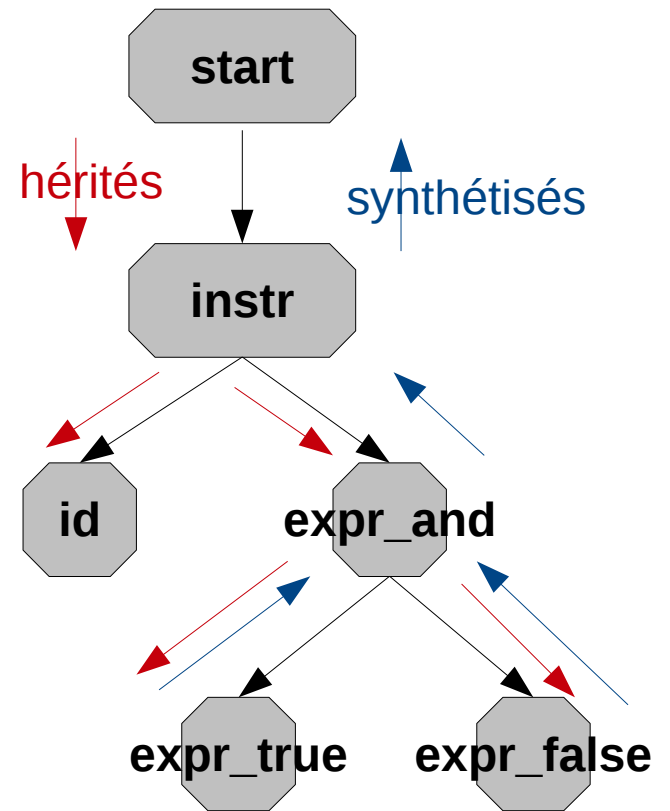
# GrammarEvaluator

- Indique la valeur de chaque production si la partie gauche ou la partie droite est typé

```
GrammarEvaluator grammarEvaluator =  
new GrammarEvaluator() {  
    public boolean expr_false(boolean expr) {  
        return expr;  
    }  
    public boolean expr_true(boolean expr) {  
        return expr;  
    }  
    public boolean expr_and(boolean expr, boolean expr2) {  
        return expr && expr2;  
    }  
    public void instr(boolean expr) {  
        System.out.println("value: "+expr);  
    }  
    public void acceptStart() {  
        // do nothing  
    }  
}
```

# Arbre & Attributs

- Sur un arbre de dérivation, on peut calculer deux sortes d'attributs
- Les attributs hérités qui vont de la racine vers les feuilles
- Les attributs synthétisés qui vont des feuilles vers la racine



# Evaluateur & Attributs

- Les évaluateur LR (comme ceux de Tatoon) permettent d'utiliser des attributs synthétisés sans construire réellement l'arbre
- Si l'on veut des attributs hérités, il faut construire l'arbre.  
On peut le faire grâce à un attribut synthétisé

