



# Langage Vyking

Lexing et parsing

---

**Compilateurs**

**INFO0085**

**PR. PIERRE GEURTS**

**CYRIL SOLDANI**

16 avril 2013

ROBIN KEUNEN s093137  
PIERRE VYNCKE s091918  
1<sup>ère</sup> Master Ingénieur Civil

---

# 1 Introduction

Pour séparer les difficultés dans la compilation du langage, nous travaillerons sur deux versions successives du langage. La première, *basic\_vyking* n'implémente que les fonctions de bases, le noyau du langage. La deuxième, *listed\_vyking* rajoute les listes et les opérateurs de liste. Nous nous attaquerons à cette version quand la première version du langage sera opérationnelle (du lexing à la génération de code).

Le parser est implémenté en Python 3.

## Structure des classes

La structure des classes est illustre à la figure 1

*Lexer* est une classe abstraite initiant les paramètres de base du lexer.  
*BasicVykingLexer* spécifie les règles pour un sous-ensemble du langage complet.  
*ListedVykingLexer* étend les règles pour ajouter les tokens correspondant aux listes. Le lexer ajoute un symbole ENDMARKER à la fin du flux de tokens.

*Parser*, comme *Lexer* est une classe abstraite initiant les paramètres de base du parser.  
*BasicVykingLexer* spécifie la grammaire pour un sous-ensemble du langage.  
*ListedVykingLexer* étend la grammaire pour y ajouter les listes.

*IndentFilter* se place entre le lexer et le parser pour générer des délimiteurs de bloc. Le filtre tire parti des générateur de python. Son mécanisme est expliqué à la section 3.

*Stack* implémente une pile pour le filtre. *ASTNodes* est une classe abstraite représentant les noeuds de l'arbre syntaxique. A chaque fois qu'un noeud doit retenir de de l'information, des actions ou des relations à d'autres noeuds, on crée une nouvelle classe héritant de *ASTNodes* pour le représenter. Se référer à la section 4.1.

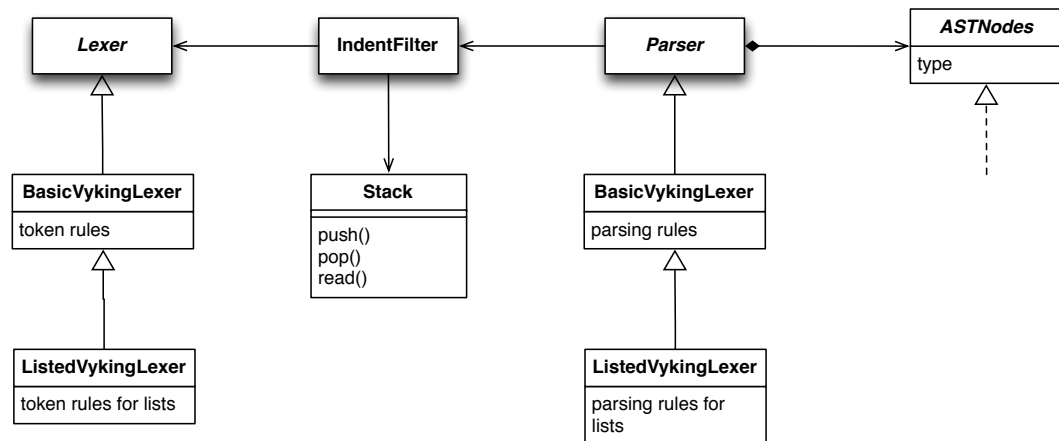


FIGURE 1 – Structure statique du compilateur (lexing et parsing)

# 2 Choix des outils

Nous avons isolé quelques lexer/parser dans la multitudes des outils disponibles.

**ANTLR** ANTLR est capable de générer du code Python pour le Parser, il dispose de beaucoup de documentation. L'analyse est top-down, ce qui rend la génération d'erreur plus facile. Cependant il vise les grammaires LL, nous ne voulions passer trop de temps à modifier la grammaire. De plus, en lisant sur le web, il apparait que le code généré cause beaucoup d'appels de fonctions[1]. Les appels sont bon marché en Java, le langage de base de ANTLR mais les appels sont chers en Python. D'autres outils sont plus efficaces et orientés Python.

**Plex** Plex est assez populaire mais cet outil ne fait que l'analyse lexicale. Nous préférons une solution intégrée pour ne pas devoir apprendre deux outils.

**PLY** PLY est une implémentation de lex et yacc pour Python. PLY est programmé en Python pur, il n'est pas construit sur un noyau C. Cette approche le rend plus lent que les outils construits en C mais il est plus efficace (en temps et espace)[2] que tous les autres bibliothèques Python. Il existe beaucoup de ressources en ligne et l'outil fournit des outils de diagnostic. Le report d'erreur était un des objectifs de développement pour PLY. D'un point de vue pédagogique, nous avons retenu ce choix car il nous permet de nous familiariser avec Lex et Yacc tout en conservant une approche *pythonesque*.

Nous avons retenu PLY. Une introduction est disponible ici <http://www.dabeaz.com/ply/ply.html>.

## 3 L'indentation

Dans une syntaxe à la python, les espaces ne sont significatifs qu'en début de ligne et servent à délimiter les blocs. L'indentation est représentée par les tokens WS dont la valeur est le nombre d'espaces en début de ligne.

### 3.1 Lexing de l'indentation

Le lexer peut être dans deux états : *initial* ou *bol* (beginning of line). L'état *bol* est enclenché quand le lexer lit un retour à la ligne. Une fois dans l'état *bol*, 3 actions sont possibles :

1. Lecture d'espace(s) : le token `INDENT` est émis, sa valeur est le nombre d'espaces.
2. Lecture d'un retour à la ligne, le lexer reste dans l'état *bol*.
3. Lecture d'un autre caractère, retour à l'état précédent.

Les transitions sont illustrées à la figure 2.

### 3.2 Délimitation des blocs

Dans la grammaire, les blocs sont délimités par les token `INDENT` et `DEDENT`. La classe *IndentFilter* se place entre le lexer et le parser, son rôle est d'insérer des `INDENT` et des `DEDENT` dans le flux de token. La grammaire requiert un bloc indenté après les instructions composées (*compound statement*). Une instruction composée et son bloc sont

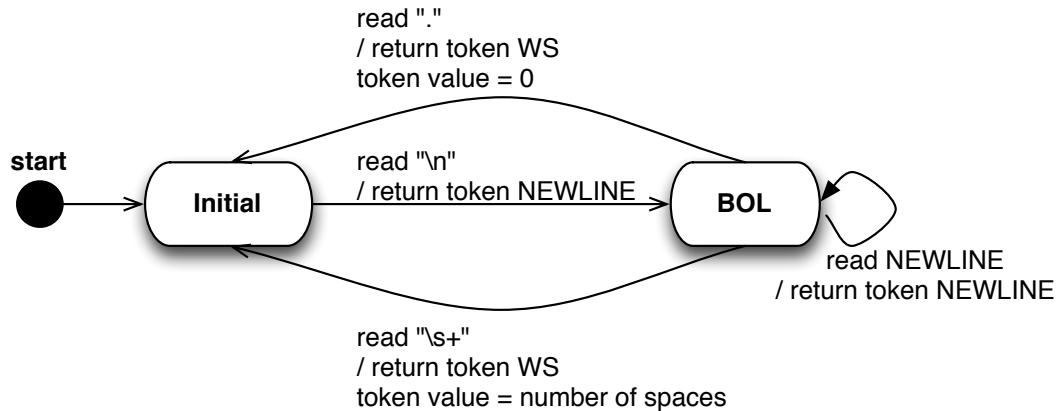


FIGURE 2 – Machine d'état du lexer

séparés par un ":" (COLON) La grammaire est la suivante :

```

compound_statement ::= statement_name COLON suite
suite               ::= simple_statement
                    | INDENT statement_sequence DEDENT
  
```

Les niveaux d'indentation sont mémorisés sur une pile "levels". Les machines d'états sont illustrées à la figure 3, page 5.

### Génération des INDENT

Le filtre peut être dans 3 états :

- NO\_INDENT : *IndentFilter* n'attend pas d'indentation. Si le niveau d'indentation augmente, c'est une erreur, si le niveau d'indentation diminue, il faut insérer des DEDENT. Quand le filtre reçoit un COLON, on passe à l'état MAY\_INDENT.
- MAY\_INDENT : si *suite* est une instruction simple (sur une seule ligne), on retourne à l'état NO\_INDENT. Si on lit un NEWLINE on passe à l'état MUST\_INDENT.
- MUST\_INDENT : *IndentFilter* attend un WS dont le niveau d'indentation est plus haut que le niveau courant, si c'est le cas, *IndentFilter* génère un INDENT et retourne à l'état NO\_INDENT. Si il n'y a pas de bloc indenté, on lève une erreur.

### Génération des DEDENT

Quand *IndentFilter* lit un WS dont la valeur est inférieure au niveau courant, il génère des DEDENT tant que le niveau au sommet de la pile n'est pas le même que celui de WS. Les états NO\_INDENT et NEED\_DEDENT ne sont pas explicites dans le code.

## 4 Parsing

### Instructions simples et composées

Les instructions peuvent être simples ou composées. Les instructions simples sont fermées par un NEWLINE, les instructions composées sont suivies d'un bloc *suite*. Le bloc *suite* est fermé par un DEDENT.

dangling statement dont need newline (dedent joue leur rôle)

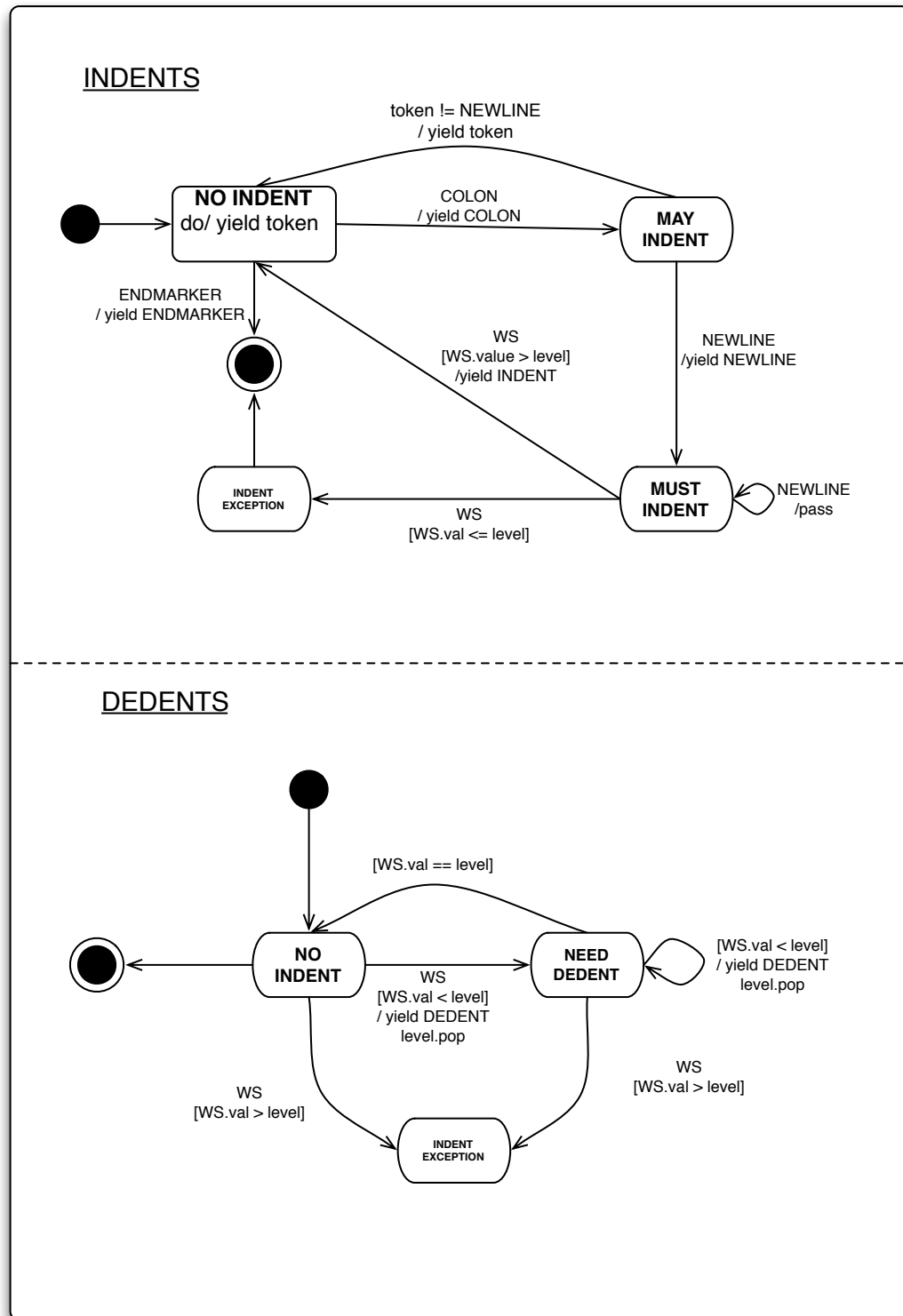


FIGURE 3 – Machine d'état de *IndentFilter*. "WS.val" représente le nombre d'espaces lu en début de ligne. "level" est le niveau courant d'indentation.

## if .. elif ... else

Nous cherchons à associer les tokens `ELSE` aux bons `IF`, c'est à dire à celui le plus proche. Le problème du "*dangling else*" a été résolu en spécifiant la précédence des règles de la grammaire. La grammaire d'une instruction "if" est la suivante :

```

if_statement ::= IF clause COLON suite if_closure
if_closure  ::= elif_statement
               | ELSE COLON suite
               | empty %prec unmatched_if

```

Le parser essaie d'associer une *if\_closure* (fermeture de if) à chaque instruction if. Une *if\_closure* peut être une instruction elif, un else ou vide. La liste de précédence indique au parser l'ordre correct de réduction des règles. La commande `%prec unmatched_if` permet d'associer une priorité de la table des priorités à une règle.

```

precedence = (
    ('nonassoc', 'unmatched_if'),
    ('nonassoc', 'ELSE'),
    ('nonassoc', 'ELIF'),
    ...
)

```

## 4.1 Arbre syntaxique

La classe *ASTNode* représente un noeud de l'arbre syntaxique. Pour chaque sorte de noeud, une nouvelle classe est implémentée. Ces classes héritent de *ASTNodes* et représente la structure du noeud. La figure 4 illustre le diagramme de structure statique de `ast.py`.

## 5 Remarques

Les 3 étapes lexing, filtre et parsing sont découplées. Mais grâce à l'utilisation de générateur/itérateurs pour construire les classes `Lexer` et `IndentFilter`, l'exécution du code ne nécessite qu'un parcours du texte. En effet, au lieu de passer tout le programme dans le `Lexer` et de conserver tous les tokens en mémoire, puis de faire un passe complet à travers le filtre, le `Lexergénère` les tokens un à un et le filtre ajoute les `INDENTS` et `DEDENTS` au vol.

## Références

- [1] more antlr - java, and comparisons to ply and pyparsing. [http://www.dalkescientific.com/writings/diary/archive/2007/11/03/antlr\\_java.html](http://www.dalkescientific.com/writings/diary/archive/2007/11/03/antlr_java.html), 2007.
- [2] David Beazley. Writing parsers and compilers with ply. <http://www.dabeaz.com/ply/PLYTalk.pdf>, February 2007.

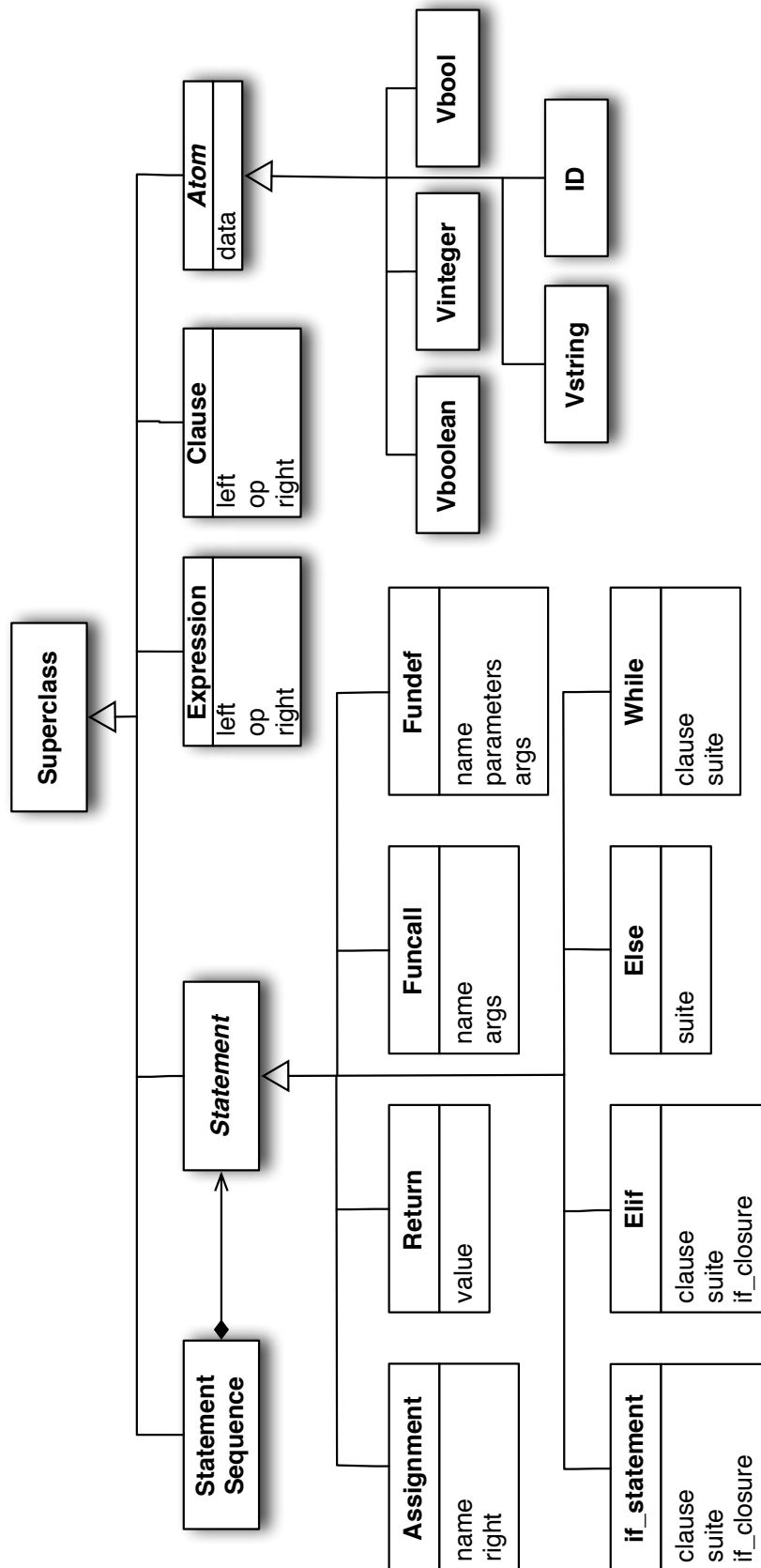


FIGURE 4 – Structure statique des noeuds de l’arbre syntaxique.