Langage Vyking

Lexing et parsing

Compilateurs

INFO0085

PR. PIERRE GEURTS

CYRIL SOLDANI

15 avril 2013

ROBIN KEUNEN s093137
PIERRE VYNCKE s091918
1ère Master Ingénieur Civil

1 Introduction

classes endmarker générateur

2 Trois versions de Vyking

Pour séparer les difficultés dans la compilation du langage, nous travaillerons sur trois 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. La troisième et dernière version *full_vyking* implémentera les closures.

3 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 construit en C mais il est plus efficace (en temps et espace)[2] que tous les autres librairies Python. Il existe beaucoup de ressources en ligne et l'outil fournit des outils de diagnostique. 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`.

4 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.

4.1 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 :

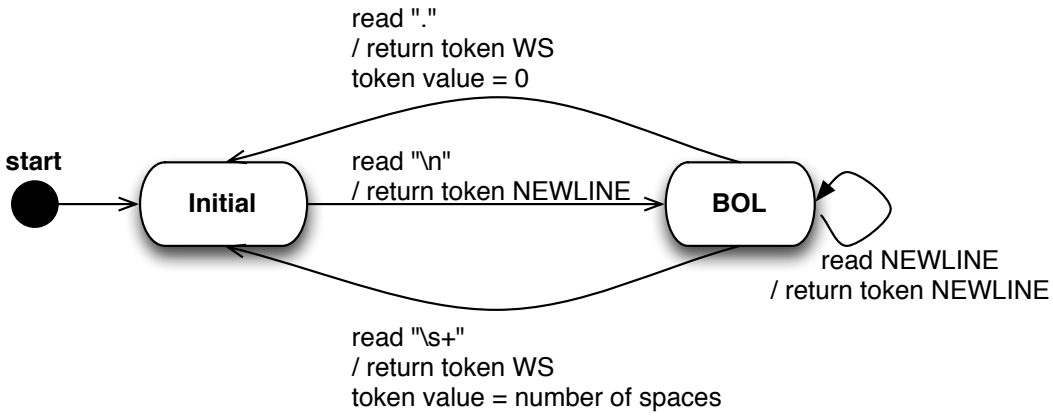

FIGURE 1 – Machine d'état du lexer

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 1.

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

$$
\begin{aligned}
compound\_statement &::= statement\_name\ COLON\ suite \\
suite &::= simple\_statement \\
&\mid INDENT\ statement\_sequence\ DEDENT
\end{aligned}
$$

Les niveaux d'indentation sont mémorisés sur une pile "*levels*". Les machines d'états sont illustrées à la figure 2, 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.

4.2.1 Génération des DEDENT

Quand *IndentFilter* lit un `WS` dont la valeur est inférieure au niveau courant, il génére 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.

5 Parsing

if ... elif ... else

dangling statement dont need mewline (dedent jour leur role)
precedence empty < else < elif

6 Gestion des erreurs

explication

detection

missing colon indentation error

fixing

missing line at end of file indentation problems

7 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 `Lexer`génère 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, 2007.

[2] David Beazley. Writing parsers and compilers with ply. http://www.dabeaz.com/ply/PLYTalk.pdf, February 2007.

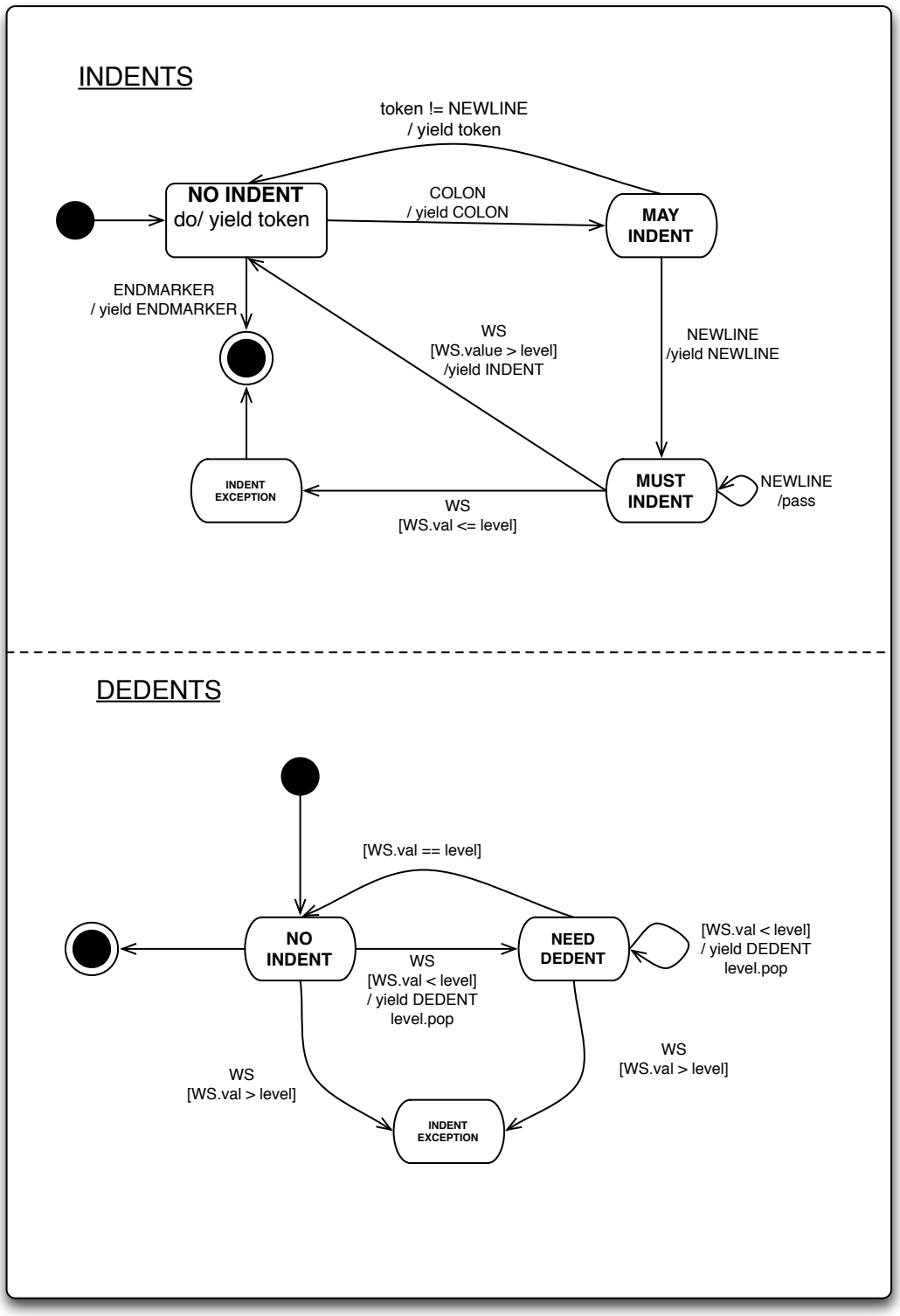

FIGURE 2 – Machine d'état de *IndentFilter*. "WS.val" représente le nombre d'espaces lu en début de ligne. "level" est le niveau courant d'indentation.