Chapitre 1

Conception du lexeur

I Conventions de notation

I.1 Types de *Token*

Dans cette partie, on désignera par i (avec $i \in \mathbb{N}^*$) le numéro de la ligne où le *Token* a été lu.

1. Retours chariot & délimitations de blocs
   - Token indiquant un saut de ligne : **Token(NEWLINE,** i**)** ;
   - Token signalant le début d'une indentation : **Token(BEGIN,** i**)** ;
   - Token marquant la fin d'une indentation : **Token(END,** i**)** ;
   - Token indiquant la fin du fichier source : **Token(EOF,** i**)**.

2. Identificateur, représentant une variable, une fonction ou un paramètre : **Token(IDENTIFIER, <identificateur>,** i**)**.

3. Mot-clé : **Token(KEYWORD, <mot clé>,** i**)**. On rappelle que les mots-clé, réservés par le langage et ne pouvant pas être utilisées comme identifiants, sont les suivants : *and*, *or*, *if*, *else*, *for*, *in*, *not*, *True*, *False*, *print*, *def*, *return* et *None*.

4. Opérateurs binaires (utilisés pour faire des opérations binaires) :
   - **Token(PLUS,** i**)** ;
   - **Token(MINUS,** i**)** ;
   - **Token(MULTIPLY,** i**)** ;

— **Token(FLOOR_DIVIDE,** i**)** ;
— **Token(MODULO,** i**)** [1] ;
— **Token(LESS_EQUAL,** i**)** ;
— **Token(GREATER_EQUAL,** i**)** ;
— **Token(LESS,** i**)** ;
— **Token(GREATER,** i**)** ;
— **Token(NOT_EQUAL,** i**)** ;
— **Token(EQUAL,** i**)** ;
— **Token(AND,** i**)** ;
— **Token(OR,** i**)**.

5. Opérateurs unaires (c'est-à-dire utilisés pour faire des opérations unaires) : **Token(UNARY_MINUS,** i**)** et **Token(NOT,** i**)**.

6. Opérateur d'assignation (":=") : **Token(ASSIGNMENT,** i**)**.

7. Types pris en charge par le langage : **Token(INTEGER, <int>,** i**)** et **Token(STRING, <str>,** i**)**.

8. Délimiteurs :
   — Crochet ouvrant ("[") : **Token(LBRACKET,**i**)** ;
   — Crochet fermant ("]") : **Token(RBRACKET,**i**)** ;
   — Parenthèse ouvrante ("(") : **Token(LPAREN,**i**)** ;
   — Parenthèse fermante (")") : **Token(RPAREN,**i**)** ;
   — Virgule : **Token(COMMA,**i**)** ;
   — Point-virgule : **Token(COLON,**i**)**.

I.2 Ensembles

Pour alléger les notations dans les automates qui suivent, on propose les notations suivantes :

- α désignera tout caractère alphanumérique (minuscule ou majuscule) :

$$\alpha = [a - z] \mid [A - Z].$$

1. On considère l'opérateur "*modulo*" comme étant l'opérateur binaire qui associe à deux entiers naturels le reste de la division euclidienne du premier par le second.

- $\mathcal{D}$ désignera tout chiffre :

$$\mathcal{D} = [0 - 9].$$

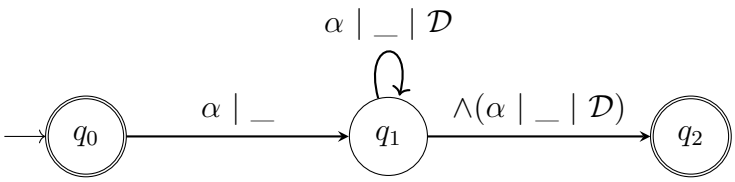
- $\mathcal{O}$ désignera tout opérateur formé d'un seul symbole :

$$\mathcal{O} = [+ \mid - \mid * \mid \%].$$

- $\mathcal{A}$ désignera n'importe quel symbole reconnu par la grammaire.

II Sous-automates

II.1 Identificateurs & mots-clés

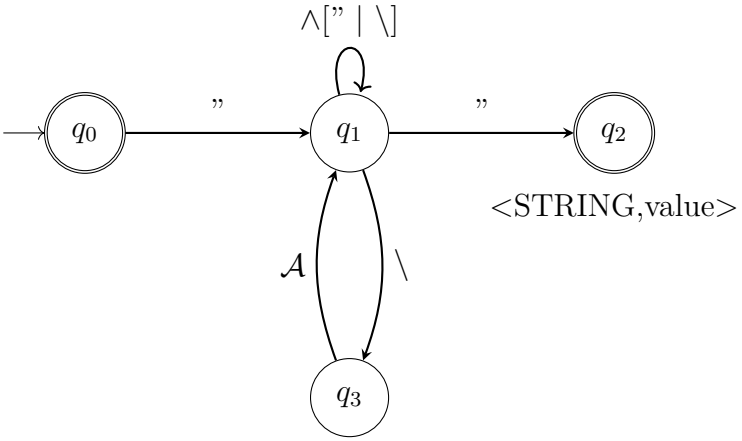


<IDENTIFIER,value>
ou <KEYWORD,value>

II.2 Délimiteurs

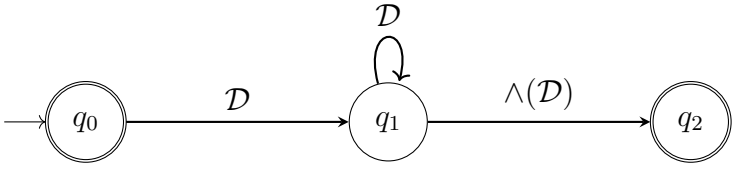
<value>

II.3 Chaînes de caractère



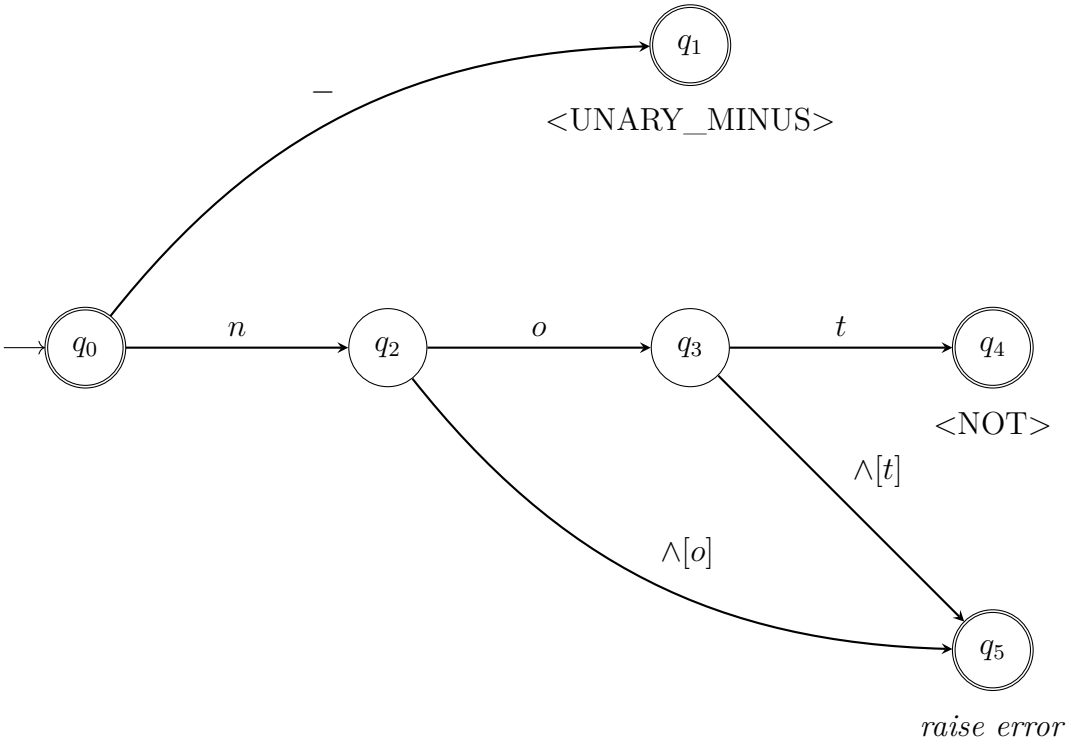
<STRING,value>

II.4 Entiers

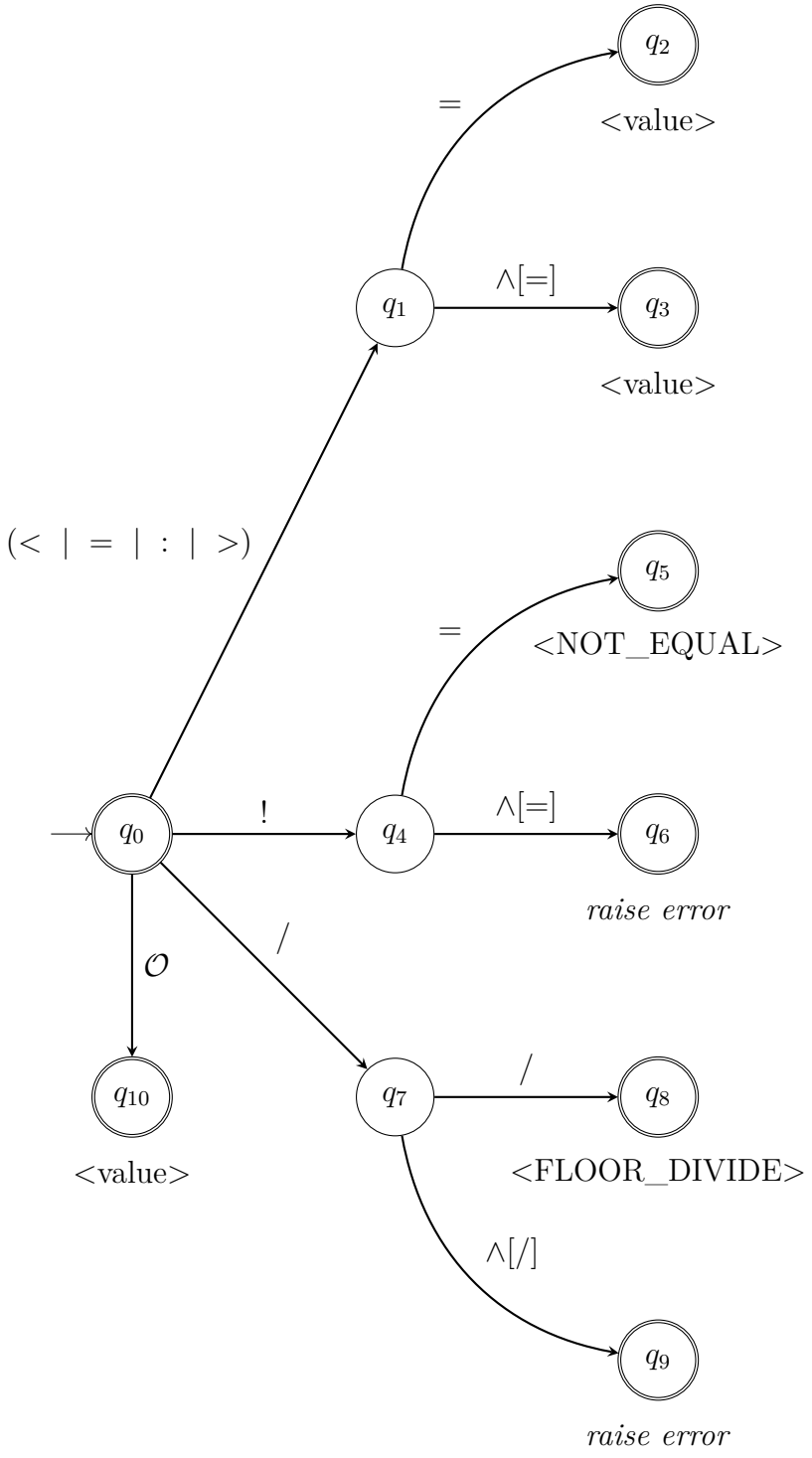


<INTEGER,value>

II.5 Opérateurs unaires

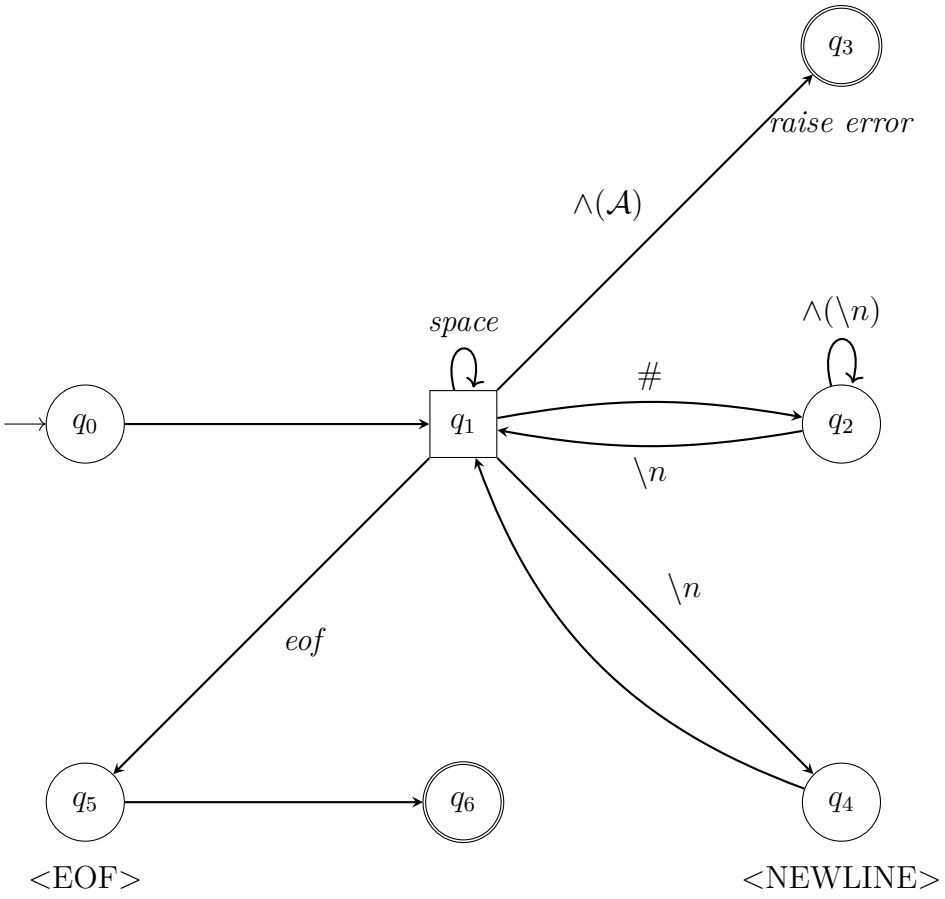


II.6 Opérateurs binaires



III Automate fini déterministe

Dans cette section, on va détailler l'automate fini déterministe correspondant au lexeur du "Mini-Python". Ce dernier implémentera les sous-automates détaillés précédemment, qui seront tous appelés simultanément à l'état 1 (représenté rectangulaire) de l'automate ci-dessous. Ainsi, n'importe quelle unité lexicale sera reconnue et verra son *Token* ajouté tant qu'aucun des autres symboles "$\backslash n$", "#" ou un caractère non-accepté par le langage.



Chapitre 2

Conception du parseur

I Grammaire considérée

On considère la grammaire (donnée dans le sujet) à la Figure 2.1, sous forme d'expressions régulières. Le premier travail est donc de la mettre sous une forme plus "conventionnelle".

II Construction de la grammaire LL(2)

On propose la grammaire suivante, obtenue à partir de celle-ci dessus, dont les règles figurent à la Figure 2.2 :

$$G = (\mathcal{N}, \mathcal{T}, \rightarrow, S), \quad \text{où}$$

$$\mathcal{N} = \{S, S', S'', A, B, B', C, C', D, D', E, E', E'', E''', F, G, G', H, H', I, I'\}$$

$$\text{et } \mathcal{T} = \alpha \cup \mathcal{D} \cup \{+; -; *; /; \%; (; [;);]; \backslash, ", :, !, =\}.$$

Il est à noter qu'elle n'est pas tout-à-fait LL(1) : l'ensemble de règles partant de E, E' et E'' rend la grammaire LL(2). Détaillons quelques étapes qui ont abouties à cette construction.

⟨ *file* ⟩	::=	**NEWLINE**? ⟨ *def* ⟩* ⟨ *stmt* ⟩+ **EOF**
⟨ *def* ⟩	::=	**def** ⟨ *ident* ⟩ (⟨ *ident* ⟩*) : ⟨ *suite* ⟩
⟨ *suite* ⟩	::=	⟨ *simple_stmt* ⟩ **NEWLINE**
	|	**NEWLINE BEGIN** ⟨ *stmt* ⟩+ **END**
⟨ *simple_stmt* ⟩	::=	**return** ⟨ *expr* ⟩
	|	⟨ *ident* ⟩ = ⟨ *expr* ⟩
	|	⟨ *expr* ⟩ [⟨ *expr* ⟩] = ⟨ *expr* ⟩
	|	**print**(⟨ *expr* ⟩)
	|	⟨ *expr* ⟩
⟨ *stmt* ⟩	::=	⟨ *simple_stmt* ⟩ **NEWLINE**
	|	**if** ⟨ *expr* ⟩ : ⟨ *suite* ⟩
	|	**if** ⟨ *expr* ⟩ : ⟨ *suite* ⟩ **else** : ⟨ *suite* ⟩
	|	**for** ⟨ *ident* ⟩ **in** ⟨ *expr* ⟩ : ⟨ *suite* ⟩
⟨ *expr* ⟩	::=	⟨ *const* ⟩
	|	⟨ *ident* ⟩
	|	⟨ *expr* ⟩ [⟨ *expr* ⟩]
	|	- ⟨ *expr* ⟩
	|	**not** ⟨ *expr* ⟩
	|	⟨ *expr* ⟩ ⟨ *binop* ⟩ ⟨ *expr* ⟩
	|	⟨ *ident* ⟩ (⟨ *expr* ⟩*)
	|	[⟨ *expr* ⟩*]
	|	(⟨ *expr* ⟩)
⟨ *binop* ⟩	::=	+ | - | * | // | % | <= | >= | < | > |!= | == | **and** | **or**
⟨ *const* ⟩	::=	⟨ *integer* ⟩ | ⟨ *string* ⟩ | **True** | **False** | **None**

FIGURE 2.1 – Grammaire proposée, sous forme de *regex*

$$
\begin{array}{lcl}
S & \rightarrow & \textbf{Newline}\ S' \mid S' \\
S' & \rightarrow & AS' \mid DS'' \\
S'' & \rightarrow & DS'' \mid \textbf{EOF} \\
A & \rightarrow & \textbf{def ident}\ (I) : B \\
I & \rightarrow & \textbf{ident}\ I' \\
I' & \rightarrow & \textbf{,}\ \textbf{ident}\ I' \mid \varepsilon \\
B & \rightarrow & C\ \textbf{Newline} \mid \textbf{Newline Begin}\ DB'\ \textbf{End} \\
B' & \rightarrow & \textbf{Newline}\ DB' \mid \varepsilon \\
C & \rightarrow & \textbf{return}\ E \mid \textbf{ident}\ C'' \mid EC' \mid \textbf{print}(E) \\
C' & \rightarrow & [E] = E \mid \varepsilon \\
C'' & \rightarrow & = E \mid (E''') \mid \varepsilon \\
D & \rightarrow & C\ \textbf{Newline} \mid \textbf{if}\ E : BD' \mid \textbf{for indent in}\ E : B \\
D' & \rightarrow & \textbf{else :}\ B \mid \varepsilon \\
E & \rightarrow & G \mid H \mid\ \textbf{True} \mid \textbf{False} \mid \textbf{None} \mid EE' \mid\ -E \mid \textbf{not}\ E \mid [E''] \mid (E) \\
E' & \rightarrow & FE \mid [E] \\
E'' & \rightarrow & EE''' \mid \varepsilon \\
E''' & \rightarrow & \textbf{,}\, EE''' \mid \varepsilon \\
F & \rightarrow & + \mid \text{-} \mid * \mid // \mid \% \mid <= \mid >= \mid < \mid > \mid != \mid == \mid \textbf{and} \mid \textbf{or} \\
G & \rightarrow & 0G' \mid 1G' \mid 2G' \mid 3G' \mid 4G' \mid 5G' \mid 6G' \mid 7G' \mid 8G' \mid 9G' \\
G' & \rightarrow & 0G' \mid 1G' \mid 2G' \mid 3G' \mid 4G' \mid 5G' \mid 6G' \mid 7G' \mid 8G' \mid 9G \mid \varepsilon \\
H & \rightarrow & \text{"}H'\text{"} \\
H' & \rightarrow & \beta H' \text{ où } \beta \in \alpha \cup \mathcal{D} \cup \{+; -; *; /; \%; (; [;);]; \backslash\} \cup \{\backslash\text{"}\}
\end{array}
$$

FIGURE 2.2 – Grammaire LL(2)