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.

Les tokens sont représentés par un numéro, une valeur optionnelle, et une ligne. Ils appartiennent à l'une des catégories suivantes : STRUCTURE, VALUE, KEYWORDS, OPERATORS, ou SYMBOLS.

1. Tokens de structure (1-4)
   - Token BEGIN (1) : signalant le début d'une indentation ;
   - Token END (2) : marquant la fin d'une indentation ;
   - Token NEWLINE (3) : indiquant un saut de ligne ;
   - Token EOF (4) : indiquant la fin du fichier source.

2. Tokens de valeur (10-12)
   - Token IDENTIFIER (10) : représentant une variable, une fonction ou un paramètre ;
   - Token INTEGER (11) : représentant une valeur entière ;
   - Token STRING (12) : représentant une chaîne de caractères.

3. Tokens mots-clés (20-32)
   - if (20)

— else (21)
— and (22)
— or (23)
— not (24)
— True (25)
— False (26)
— None (27)
— def (28)
— return (29)
— print (30)
— for (31)
— in (32)

4. Tokens opérateurs (40-53)
   — Addition "+" (40)
   — Soustraction "-" (41)
   — Multiplication "*" (42)
   — Division entière "//" (43)
   — Modulo "%" (44)
   — Inférieur ou égal "<=" (45)
   — Supérieur ou égal ">=" (46)
   — Supérieur ">" (47)
   — Inférieur "<" (48)
   — Différent "!=" (49)
   — Égal "==" (50)
   — Affectation "=" (51)
   — Division "/" (52)
   — Négation "!" (53)

5. Tokens symboles (60-65)
   — Parenthèse ouvrante "(" (60)
   — Parenthèse fermante ")" (61)
   — Crochet ouvrant "[" (62)

— Crochet fermant "]" (63)
— Deux-points " :" (64)
— Virgule "," (65)

La catégorie d'un token peut être déterminée par sa plage de numéros :
— 1-4 : STRUCTURE
— 10-12 : VALUE
— 20-32 : KEYWORDS
— 40-53 : OPERATORS
— 60-67 : SYMBOLS

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) :

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

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

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

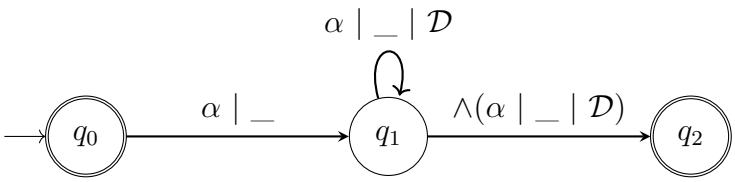
- $\mathcal{O}$ désignera tout opérateur formé d'un seul symbole :

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

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

II Sous-automates

II.1 Identificateurs & mots-clés

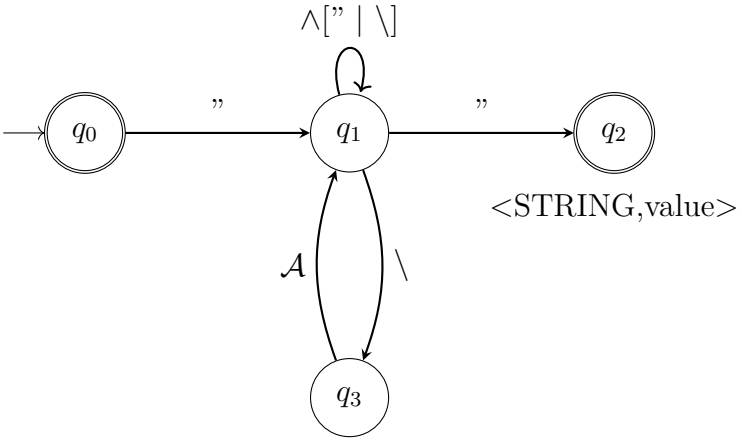


<IDENTIFIER,value>
ou <KEYWORD,value>

II.2 Délimiteurs

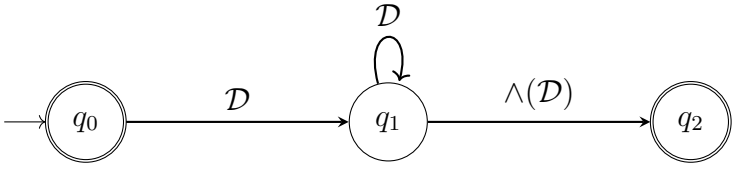
<value>

II.3 Chaînes de caractère



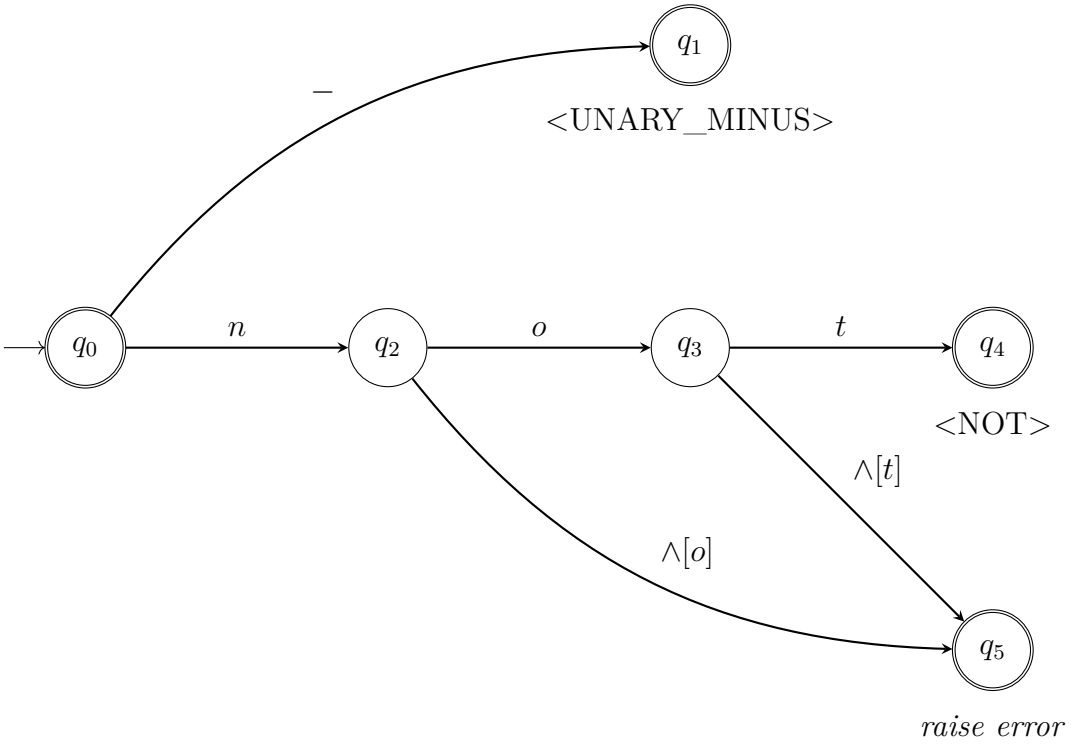
<STRING,value>

II.4 Entiers

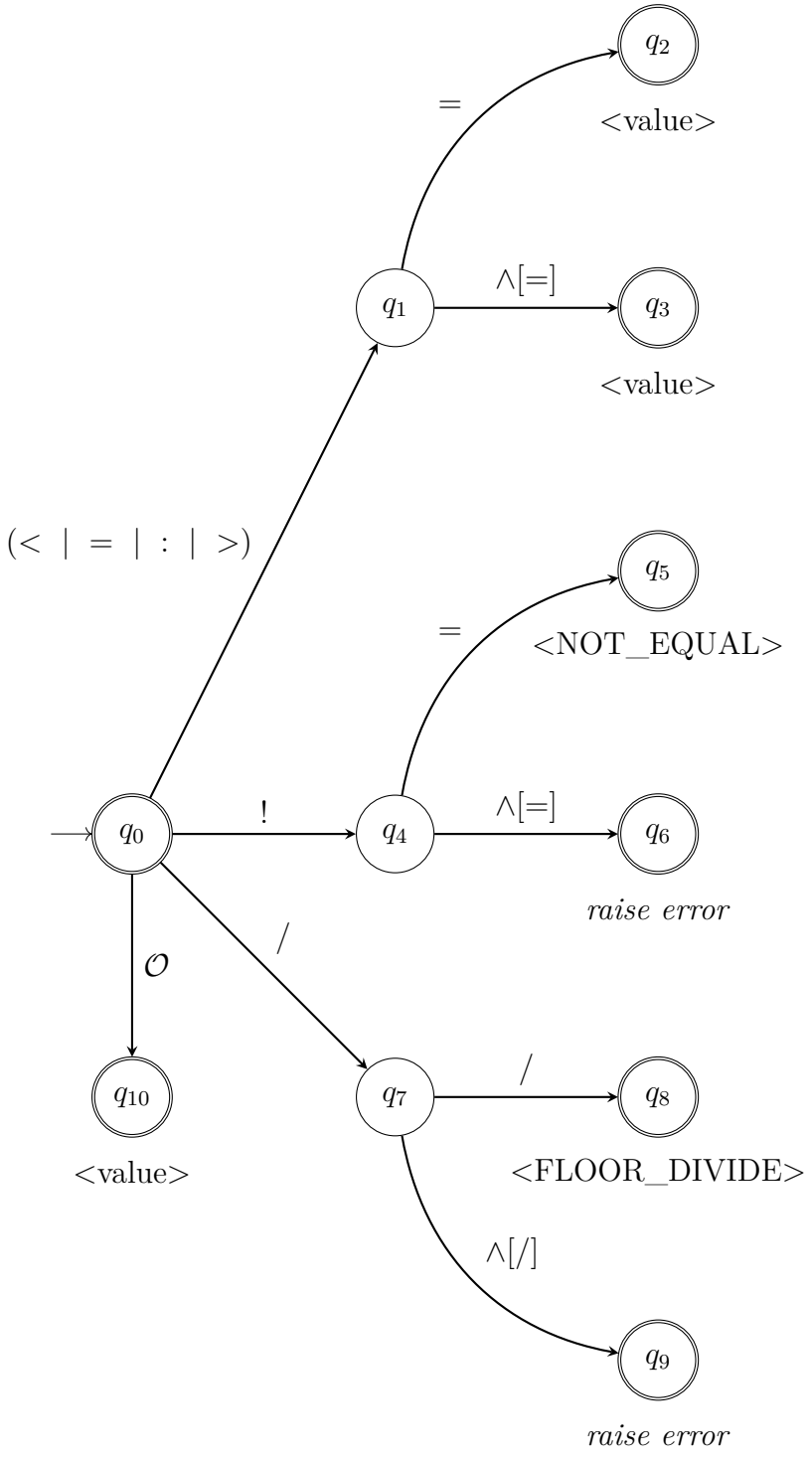


<INTEGER,value>

II.5 Opérateurs unaires

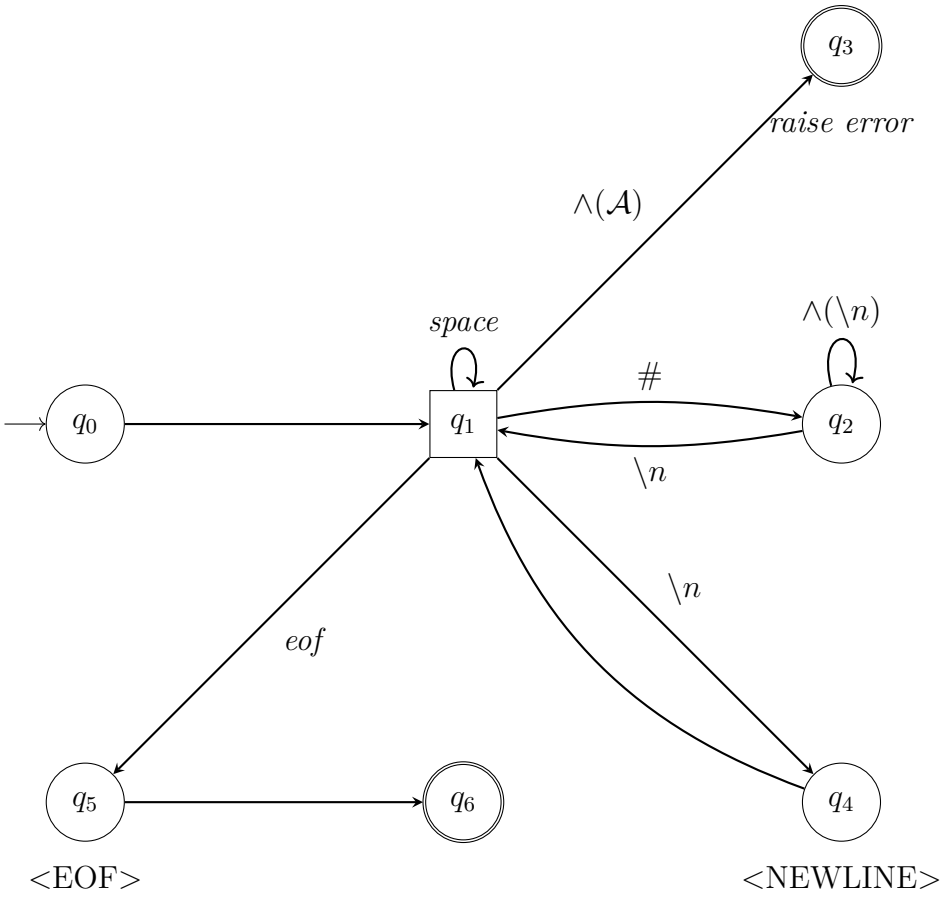


II.6 Opérateurs binaires



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 LL(2)

On propose la grammaire suivante, obtenue à partir de celle-ci dessus, dont les règles figurent en annexe (document "grammar.pdf") :

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

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

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

II Élagage de l'arbre

II.1 Suppression des éléments lexicaux non-essentiels

De nombreux *tokens* dans l'arbre à sa sortie du parseur sont purement syntaxiques et n'affectent pas réellement la logique sous-jacente. Il s'agit des *tokens* du types "**Newline**", "**def**" et "**EOF**", ainsi que les virgules et les deux-points. On va donc les éliminer en premier.

II.2 Modification des tuples et des listes

La prochaine étape que l'on propose est de s'occuper de reformatter les tuples et les listes du Mini-Python. Pour se simplifier la vie, on propose d'appliquer un algorithme récursif qui applique un formatage à chaque fois qu'il voit des nœuds de l'arbre entourés de *Tokens* de parenthèses / crochets ouvrant(e) et fermant(e).

Le formatage adopté est le suivant (on l'explique ici pour un tuple, mais le formatage est le même pour les listes) : dans la liste des enfants, à la place de l'ensemble "nœud parenthèse ouvrante + contenu du tuple + nœud parenthèse fermante, on place un nouveau nœud "tuple", dont les enfants sont le contenu du tuple.

Le problème avec un tel algorithme est le recopiage du contenu du tuple. En effet, même si les virgules séparant les différents éléments ont été supprimées à l'étape précédente de l'élagage, il reste tous les non-terminaux qui ont servi à la construction du tuple, or on aimerait que les enfants du nœud "tuple" soient directement le contenu du tuple (dans l'ordre). Il faut donc retirer ces non-terminaux (et seulement ceux-là, les autres peuvent encore servir à l'élagage). Dans notre grammaire, il s'agit des non-terminaux I, I_1, E_1 et E_2 (on peut noter que I et I_1 ne peuvent apparaître que dans les tuples).

II.3 Modification des fonctions

Pour les fonctions, on détecte leur présence dans l'arbre tout naturellement avec la présence du mot-clé "**def**". Dès lors, on le retire (car il n'est plus vraiment utile à l'analyse sémantique après cette étape), et on range dans un nouveau nœud "**function**" dans l'ordre l'identifiant contenant le nom de la fonction, un tuple (éventuellement vide) correspondant aux paramètres de la fonction, et un non-terminal contenant en enfants les lignes de code de la fonction.

II.4 Modification des opérateurs

Aussi, pour la plupart des opérateurs (somme, différences, opérateurs de comparaison, ...), leur arité est de 2. On propose donc de mettre leurs deux paramètres en enfant du nœud opérateur (on adopte en quelques sortes une notation préfixée pour ces opérateurs, à l'échelle de l'arbre).

II.5 Modifications des print et des return

Pour ce qui est des **print** et des **return**, le formalisme est le même : on les considère comme des opérateurs unaires, dont la valeur utilisée (à afficher ou à renvoyer) est passée en enfant du nœud **print** / **return**.

II.6 Modification des for et des if

À l'instar des fonctions, on a rassemblé les éléments caractéristiques des **for** en enfants du nœud "**for**". Pour une boucle de la forme "for i in L", on stockera dans les enfants du nœud "**for**" dans l'ordre l'identifiant contenant i, l'identifiant contenant L, et enfin le bloc à exécuter dans la boucle.

Pour ce qui est des **if**, on procède de même, en stockant dans les enfants du nœud **if** dans l'ordre : la condition à vérifier pour rentrer dans le **if**, le code a exécuter si la condition est vérifiée, et éventuellement un nœud "**else**" contenant le code a exécuter si cela n'est pas le cas.

II.7 Finalisations de l'AST

Enfin pour conclure l'élagage, on rassemble les chaînes "inutiles" de nœuds non-terminaux, de sorte que tous les nœuds représentant des lignes d'un même bloc de code soient tous les enfants du même nœud (généralement un non-terminal, comme S, A ou D).