

Calculabilité (TALA540B)

Corrigé « Analyseur syntaxique $a\ b\ [\]$ », assertions des instructions
de base des instructions de la machine de Turing de M. del Vigna

Patrick Paroubek

LISN (Laboratoire Interdisciplinaire des Sciences du Numérique) - CNRS - U. Paris-Saclay,
Campus universitaire bât 507, Rue du Belvédère, F - 91400 Orsay,
email:patrick.paroubek@lisn.upsaclay.fr

mercredi 25 octobre 2023 / Semestre 1 - Cours 6

Corrigé exercice

Énoncé

exercice (à rendre pour le prochain cours)

réaliser un parseur en descente recursive pour un langage défini comme suit:

```
alphabet = { 'a', 'b', '[', ']' }
```

Règles de grammaire:

- 1 on peut avoir des séquences de a de longueur quelconque, n'importe où
- 2 on peut avoir des séquences de b uniquement à l'intérieur de paires de [] et seulement si il y a au moins un a avant le premier b dans la portée des crochets courants
- 3 les crochets doivent être équilibrés et peuvent s'imbriquer

Des textes valides:

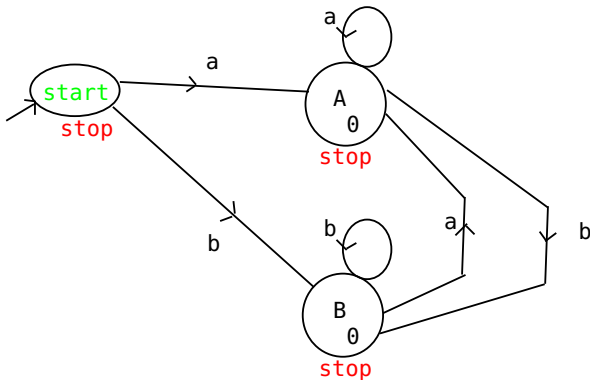
```
"a"  
"aaaaaaa"  
"aaaa[abbbbbbbbaaa]aaaa"  
"aa[aaa[abbbbb]bbb]aaa"
```

Des textes non-valides:

```
b  
[b]  
[  
aa[ab]aa]a  
aa[a[b]]  
a  
aa[ a[ab ] [ aa ] ]
```

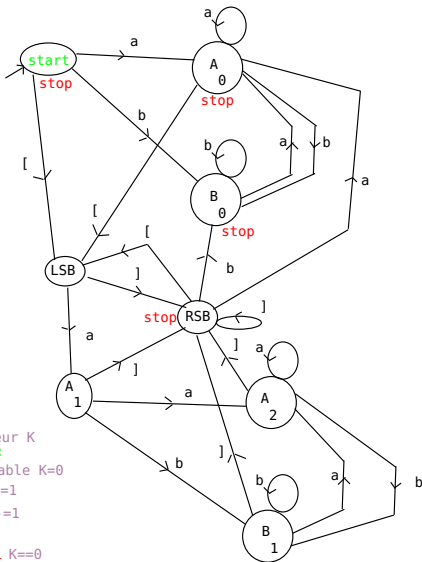
Corrigé exercice

début avec l'automate pour reconnaître $(a|b)^n$



Corrigé exercice

un peu de copier/coller et ajout de « [» et «] »



Compteur K

start:

variable K=0

[: K+=1

] : K-=1

stop:

OK si K==0

et fin de l'entrée ERREUR sinon

Au lieu de dessiner un automate, nous pouvons écrire une grammaire formelle pour le langage.

On peut définir une grammaire comme un système de réécriture qui part d'un symbole unique, source de tous les messages appartenant au langage que l'on peut générer en appliquant les règles de réécritures.

On distingue les symboles intermédiaires (non-terminaux), S, D et U dans l'exemple ci-dessous, et les symboles terminaux 0 et 1 dans l'exemple.

Par ex. la grammaire suivante génère une suite arbitraire de 1 se terminant toujours par un 0.

$$S \rightarrow 1D$$
$$D \rightarrow 1U$$
$$D \rightarrow 1S$$
$$U \rightarrow 0$$

Les langages peuvent être classés en fonction des caractéristiques des règles qui les définissent selon une hiérarchie dite de Chomsky (aussi appelée hiérarchie de Chomsky-Schützenberger), de la classe des plus complexe (dite de type 0), qui regroupe les langues naturelles, jusqu'à la classe la plus simple, celle des langages réguliers (dits de type 3), qui sont définis par les expressions régulières (et reconnus par les automates à états finis déterministes).

type 0	pas de restriction sur les règles	langues naturelles
type 1	la partie droite des règles doit contenir au moins autant de symboles que la partie gauche ¹	langages sensibles au contexte
type 2	toutes les règles doivent avoir un seul symbole non-terminal en partie gauche et il n'y a pas de restriction sur les parties droites.	langages hors-contextes
type 3	deux types de règles sont admises, tous deux avec un unique symbole non-terminal en partie gauche de règle : les règles dont la partie droite est uniquement composée de symboles terminaux et les règles qui contiennent un seul symbol non terminal soit en première soit en dernière position de partie droite sans contrainte sur le nombre de symboles terminaux présents de la partie droite	langages réguliers

1. Il est possible d'avoir une règle ne générant au symbole à condition que la partie gauche soit seulement le symbole initial et que ce dernier n'apparaisse jamais dans une partie droite de règle.

Corrigé exercice

Grammaire correspondant à l'automate

$S \rightarrow a A_0$	$B_1 \rightarrow b B_1$
$S \rightarrow b B_0$	$B_1 \rightarrow a A_2$
$S \rightarrow [\text{LSB}$	$B_1 \rightarrow] \text{RSB}$
$A_0 \rightarrow a A_0$	$A_2 \rightarrow] \text{RSB}$
$A_0 \rightarrow b B_0$	$A_2 \rightarrow a A_2$
$B_0 \rightarrow b B_0$	$A_2 \rightarrow b B_1$
$B_0 \rightarrow a A_0$	$\text{RSB} \rightarrow [\text{LSB}$
$\text{LSB} \rightarrow] \text{RSB}$	$\text{RSB} \rightarrow a A_0$
$\text{LSB} \rightarrow a A_1$	$\text{RSB} \rightarrow b B_0$
$A_1 \rightarrow] \text{RSB}$	$\text{RSB} \rightarrow] \text{RSB}$
$A_1 \rightarrow a A_2$	
$A_1 \rightarrow b B_1$	

- ▶ si une règle ayant un S en partie gauche est utilisée alors le compteur de crochets K reçoit la valeur 0
- ▶ la génération d'un $\ll [\gg$ en partie droite de règle provoque l'incréméntation de 1 du compteur K
- ▶ la génération d'un $\ll] \gg$ en partie droite de règle provoque la décrémentation de 1 du compteur K
- ▶ Si une règle ayant un S , A_0 , B_0 , ou RSB en partie gauche est utilisée ALORS, à la condition que le compteur K ait pour valeur 0 et qu'il n'y ait plus de caractère à lire sur l'entrée de l'automate, le texte est reconnu comme faisant partie du langage, SINON il est rejeté.

Corrigé exercice

Parseur implémentation

- ▶ à partir de la grammaire précédente il est facile d'implémenter un analyseur syntaxique en associant un appel de fonction récursive à chaque règle, en partant du squelette de programme d'analyse syntaxique récursif vu dans le précédent cours,
- ▶ des extraits du début du code et de la fin du code sont donnés dans les deux transparents suivants,
- ▶ le code complet est disponible à l'url :
https://perso.limsi.fr/pap/inalco/MASTER2_2023_2024/ab_parser.py

Corrigé exercice

Parser implémentation (début)

```
ab_parser.py - /home/pap/COURSES/inalco_23_24/M2/SEMESTER_1/course_51_6_20231025/ab_parser.py (3.8.10)
File Edit Format Run Options Window Help
1 #in file = 'ab_test.txt'
2 in_file = 'ab_test_bad.txt'
3 out_file = 'ab_test_parse.log'
4
5 all_parse_states = [ 'INIT ST', 'S', 'A0', 'A1', 'A2', 'B0', 'B1', 'LSB', 'RSB' ]
6 # manque les états intermédiaires dans cette liste
7
8 def main_parse():
9     with open( in_file, 'rt') as in_fdesc:
10         with open( out_file, 'wt' ) as out_fdesc:
11             return S( in_fdesc, out_fdesc )
12
13 def next_tokn( in_f ):
14     x = in_f.read( 1 )
15     return x
16
17 def S( in_f, out_f, parse_state = 'S', K = 0 ):
18     parse_state = 'S'
19     next_tok = next_tokn( in_f )
20     out_f.write( 'in state {0} with new input token {1} and K == {2}\n'.format( parse_state, next_tok, K ) )
21     if next_tok:
22         if next_tok == 'a':
23             return A0( in_f, out_f, parse_state, K )
24         elif next_tok == 'b':
25             return B0( in_f, out_f, parse_state, K )
26         elif next_tok == 'i':
27             return LSB( in_f, out_f, parse_state, K + 1 )
28         else:
29             print( "ERROR caractère {0} of code {1} non autorisé dans l'état {2}".format( next_tok, ord( next_tok ), parse_state ) )
30             print( 'aborting....' )
31             exit( 1 )
32     else:
33         if K == 0:
34             out_f.write( 'texte accepté comme valide' )
35             return True
36         else:
37             out_f.write( 'texte non valide [] non équilibrés' )
38             return False
39
40 def A0( in_f, out_f, parse_state = 'S', K = 0 ):
41     parse_state = 'A0'
42     next_tok = next_tokn( in_f )
43     out_f.write( 'in state {0} with new input token {1} and K == {2}\n'.format( parse_state, next_tok, K ) )
44     if next_tok:
45         if next_tok == 'a':
46             return A0( in_f, out_f, parse_state, K )
47         elif next_tok == 'b':
48             return B0( in_f, out_f, parse_state, K )
49         elif next_tok == 'i':
50             return LSB( in_f, out_f, parse_state, K + 1 )
51     else:
```

Ln: 2 Col: 27

Corrigé exercice

Parser implémentation (fin)

```
ab_parser.py - /home/pap/COURSES/inalco_23_24/M2/SEMESTER_1/course_51_6_20231025/ab_parser.py (3.8.10)
File Edit Format Run Options Window Help
130 return A2( in_f, out_f, parse_state, K + 1 )
131 else:
132     print( "ERROR caractère {0} of code {1} non autorisé dans l'état {2}".format( next_tok, ord( next_tok ), parse_state ))
133     print( 'aborting....' )
134     exit( 1 )
135 else:
136     return None
137
138 def A2( in_f, out_f, parse_state = 'S', K = 0 ):
139     parse_state = 'A2'
140     next_tok = next_token( in_f )
141     out_f.write( 'in state {0} with new input token {1} and K == {2}\n'.format( parse_state, next_tok, K ))
142     if next_tok:
143         if next_tok == 'a':
144             return A2( in_f, out_f, parse_state, K )
145         elif next_tok == 'b':
146             return B1( in_f, out_f, parse_state, K )
147         else:
148             print( "ERROR caractère {0} of code {1} non autorisé dans l'état {2}".format( next_tok, ord( next_tok ), parse_state ))
149             print( 'aborting....' )
150             exit( 1 )
151     else:
152         return None
153
154 def RSB( in_f, out_f, parse_state = 'S', K = 0 ):
155     parse_state = 'RSB'
156     next_tok = next_token( in_f )
157     out_f.write( 'in state {0} with new input token {1} and K == {2}\n'.format( parse_state, next_tok, K ))
158     if next_tok:
159         if next_tok == 'a':
160             return A0( in_f, out_f, parse_state, K )
161         elif next_tok == 'b':
162             return B0( in_f, out_f, parse_state, K )
163         elif next_tok == '[':
164             return LSB( in_f, out_f, parse_state, K + 1 )
165         elif next_tok == ']':
166             return RSB( in_f, out_f, parse_state, K - 1 )
167         else:
168             print( "ERROR caractère {0} of code {1} non autorisé dans l'état {2}".format( next_tok, ord( next_tok ), parse_state ))
169             print( 'aborting....' )
170             exit( 1 )
171     else:
172         if K == 0:
173             out_f.write( 'texte accepté comme valide' )
174             return True
175         else:
176             out_f.write( 'texte non valide [] non équilibrés' )
177             return False
178
179 main_parse()
180
```

Lrs 2 Col: 27

Corrigé exercice

Parser implémentation

- ▶ si l'implémentation récursive obtenue de cette façon n'est pas nécessairement optimale en termes algorithmique, elle présente l'avantage d'être obtenue de manière simple et intuitive à partir du graphe de l'automate ou bien des règles de la grammaire.
- ▶ la théorie algorithmique de ce type d'approche et de ses optimisations est expliquée en détails dans le fameux livre dit « du dragon² », dont on trouve des anciennes éditions en ligne³

2. https://fr.wikipedia.org/wiki/Dragon_book

3. <https://iitd-plos.github.io/col729/refs/ALSUdragonbook.pdf>

Corrigé devoir 1

un parseur pour le langage MTdV

La syntaxe et la sémantique des instructions du langage MTdV, vu au cours 3 de ce semestre, sont rappelées dans le transparent suivant.

instruction	sémantique
I	affiche l'état de la machine
P	« pause », interrompt temporairement l'exécution du programme; celle-ci reprend une fois que l'utilisateur a tapé sur une touche du clavier en réponse au prompt affiché.
G,D	déplace la tête de lecture respectivement d'une position à Gauche, à Droite
0,1	écrit respectivement un 0 (case vide) ou un 1 (baton) à l'emplacement de la tête de lecture
si (0) x1 x2 ... }	si la tête de lecture est sur une case vide alors les instructions x1, x2, ... sont exécutées en séquence, sinon exécute la première instruction qui suit l'accolade fermante }
si (1) x1 x2 ... }	même chose dans le cas où la tête de lecture est positionnée sur un baton
boucle x1 x2 ... }	répète la séquence d'instructions x1 x2 ..., jusqu'à ce que l'une d'elle soit l'instruction fin
fin	rompt le cycle de répétition d'une boucle en faisant exécuter la première instruction qui suit l'accolade fermante de la première boucle contenant l'instruction fin; si fin n'est contenue dans aucune boucle alors le programme s'arrête.
%	définit une ligne de commentaire non exécutable
#	marqueur de fin de fichier requis comme dernière instruction

Une première Grammaire du langage MTdV

Notez que « % » le marqueur de commentaire n'est pas décrit ici car son implémentation est triviale. Cette grammaire avec les contraintes sur le compteur K permet de vérifier la syntaxe d'un programme MTdV, y compris la complétude et la cohérence des éléments parenthétiques comme les instructions de début et de fin de blocs de code des boucles ou des branchements conditionnels.

$P_0 \rightarrow I P_0$ $P_0 \rightarrow P P_0$ $P_0 \rightarrow G P_0$ $P_0 \rightarrow D P_0$ $P_0 \rightarrow 0 P_0$ $P_0 \rightarrow 1 P_0$ $P_0 \rightarrow \text{fin } P_0$ $P_0 \rightarrow \} P_0$ $P_0 \rightarrow \#$ $P_0 \rightarrow \text{boucle } P_0$ $P_0 \rightarrow \text{si } (0) P_0$ $P_0 \rightarrow \text{si } (1) P_0$	$\text{assert}(k > 0)$ $K -= 1$ $\text{assert}(K == 0)$ et fin de l'entrée $K += 1$ $K += 1$ $K += 1$
---	---

Parser v1 pour MTdV (1/2)

Partie lexicale

L'implémentation est disponible à l'URL :

`https:`

`//perso.limsi.fr/pap/inalco/MASTER2_2023_2024/mdtv_parser_1.py`

Parser v1 pour MTdV (1/2)

Partie lexicale

```
*mdtv_parser_1.py - /home/pap/COURSES/inalco_23_24/M2/SEMESTER_1/course_S1_6_20231025/mdtv_parser_1.py [3.8.10]*

File Edit Format Run Options Window Help

import re
in_file = 'annule_arg.TS'
out_file = 'annule_arg.parse.log'
all_parse_states = [ 'p0' ]
terminals_re = {
    '#' : '^[\n]*#[\n]*',
    ')' : '^[\n]*)[\n]*',
    'I' : '^[\n]*I[\n]*',
    'P' : '^[\n]*P[\n]*',
    'G' : '^[\n]*G[\n]*',
    'D' : '^[\n]*D[\n]*',
    '0' : '^[\n]*0[\n]*',
    '1' : '^[\n]*1[\n]*',
    'fin' : '^[\n]*fin[\n]*',
    'boucle' : '^[\n]*boucle[\n]*',
    'si(0)' : '^[\n]*si[\n]*\^[^\n]*0[^\n]*[\n]*',
    'si(1)' : '^[\n]*si[\n]*\^[^\n]*1[^\n]*[\n]*' }

terminals_automata = { x : re.compile( terminals_re[ x ] ) for x in terminals_re.keys() }

def tokenize( in_f, terms_autom = {} ):
    txt = in_f.read()
    txt_sz = len( txt )
    tokens = []
    match = True
    while match and (len( txt ) != 0):
        for tok_nm in terms_autom.keys():
            #print( 'beginning of text == ', txt[ 0 : min(10, len( txt ) ) ] )
            #print( '\t searching for token ', tok_nm )
            match = terms_autom[ tok_nm ].search( txt )
            if match:
                #print( 'match found ', match )
                (b, e) = match.span( 0 )
                #print( '-----> ', (b, e) )
                if b == 0:
                    tokens.append( tok_nm )
                    txt = txt[ e : ]
                    break
            else:
                #print( tok_nm + ' match failed' )
                pass
        #print( tokens )
        if match or len( txt ) == 0:
            return tokens
        else:
            print( 'ERROR unknow token encountered in the input at position {0}'.format( txt_sz - len( txt ) ) )
            print( 'aborting' )
            exit( 1 )
```

Parser v1 pour MTdV (2/2)

Partie syntaxique

```
File Edit Format Run Options Window Help
"mdtv_parser_1.py - /home/pap/COURSES/inalco_23_24/M2/SEMESTER_1/course_51_6_20231023/mdtv_parser_1.py (5.8.10)"

def next_tok( tokens ):
    if tokens == []:
        print( 'ERROR in next_tok() no token available')
        return ''
    else:
        return tokens[ 0 ]

def main_parse( terms_autom ):
    with open( in_file, 'rt' ) as in_fdesc:
        with open( out_file, 'wt' ) as out_fdesc:
            tokens = Tokenize( in_fdesc, terms_autom )
            return P0( tokens, out_fdesc )
    |

def P0( tokens, out_f, parse_state = 'P0', K = 0 ):
    tok = next_tok( tokens )
    # out_f.write( 'in state {0} with new input token {1} and K == {2}\n'.format( parse_state, tok, K ) )
    if ( tok == '#' ) and ( K == 0 ):
        out_f.write( tok + '\n' )
        return True
    elif tok in [ 'I', 'P', 'G', 'D', 'O', 'I', 'fin' ]:
        out_f.write( tok + '\n' )
        return P0( tokens[ 1: ], out_f, 'P0', K )
    elif tok in [ 'boucle', 'si(0)', 'si(1)' ]:
        out_f.write( tok + '\n' )
        return P0( tokens[ 1: ], out_f, 'P0', K + 1 )
    elif tok == '}':
        if K < 1:
            print( 'ERROR unbalanced closing code block token "{}"' )
            return False
        else:
            out_f.write( tok + '\n' )
            return P0( tokens[ 1: ], out_f, 'P0', K - 1 )
    else:
        print( 'ERROR unexpected end of input' )
        return False

main_parse( terminals_automata )

Ln: 62 Col: 10
```

Parser v1 pour MTdV (1/2)

Partie lexicale

On peut améliorer cette première version en produisant un fichier de sortie au format csv avec un identifiant pour chaque token et sa profondeur d'emboîtement de blocs de code :

https://perso.limsi.fr/pap/inalco/MASTER2_2023_2024/mdtv_parser_1_with_token_id.py

Parser v1 pour MTdV (2/2)

Partie syntaxique

L'analyse syntaxique du programme MTdV *annule_arg.TS* peut être visualisée avec un tableur (LibreOffice) :

	A	B	C	D	E	F	G
1	#toki	toki	level				
2	1	si(0)	0				
3	2	1	1				
4	3	1	1				
5	4	si(1)	0				
6	5	D	1				
7	6	1	1				
8	7	boucle	1				
9	8	si(0)	2				
10	9	fin	3				
11	10	1	3				
12	11	si(1)	2				
13	12	0	3				
14	13	1	3				
15	14	D	3				
16	15	1	3				
17	16	1	3				
18	17	1	2				
19	18	1	1				
20	19	boucle	0				
21	20	si(0)	1				
22	21	G	2				
23	22	1	2				
24	23	si(1)	1				
25	24	fin	2				
26	25	1	2				
27	26	1	1				
28	27	1	0				
29	28	#	0				

Objectif Devoir Assertions

Générateur pre/post assertions MTdV en python

- ▶ L'objectif du devoir attendu pour le cours du 08 novembre est la génération automatique de pré et post assertions dans la syntaxe python pour n'importe quel programme MTdV.
- ▶ Pour chaque instruction MTdV, nous avons un minimum de pre et post assertions générales qui décrivent la sémantique des instructions en termes de :

1. déplacement de la tête de lecture,
2. modification de la bande de la machine de Turing,
3. affichage,
4. et progression dans l'exécution du programme.

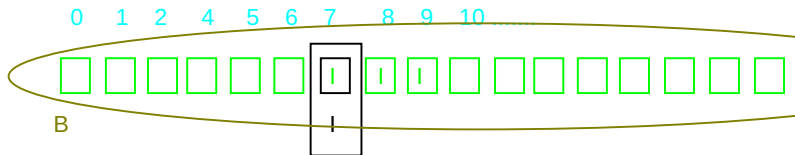
au moyen de quelques variables (vues au cours 4 du 11 octobre 202)
pour décrire l'état de la machine de Turing.

Les transparents qui suivent donnent ces pré et post condition pour les instructions de la machine de Turing.

Rappel MTdV

Liste d'instructions (syntaxe et sémantique)

instruction	sémantique
I	affiche l'état de la machine
G,D	déplace la tête de lecture respectivement d'une position à Gauche, à Droite
0,1	écrit respectivement un 0 (case vide) ou un 1 (baton) à l'emplacement de la tête de lecture
si (0) x1 x2 ... }	si la tête de lecture est sur une case vide alors les instructions x1, x2, ... sont exécutées en séquence, sinon exécute la première instruction qui suit l'accolade fermante }
si (1) x1 x2 ... }	même chose dans le cas où la tête de lecture est positionnée sur un baton
boucle x1 x2 ... }	répète la séquence d'instructions x1 x2 ..., jusqu'à ce que l'une d'elle soit l'instruction fin
fin	rompt le cycle de répétition d'une boucle en faisant exécuter la première instruction qui suit l'accolade fermante de la première boucle contenant l'instruction fin; si fin n'est contenue dans aucune boucle alors le programme s'arrête.
%	définit une ligne de commentaire non exécutable
#	marqueur de fin de fichier requis comme dernière instruction



```

0 |
1 D
2 boucle
3 0
4 D
  |
  si (0) fin }
G
|
boucle
|

```

$I = 1$ $P = 7$ $S = 1$

pré-condition

$I = 2$ $P = 8$ $S = 1$

post-condition

Notez que $B[P] = S$

on pourrait donc se passer de S

instruction	sémantique
I	affiche l'état de la machine

pré- et post- conditions

$$\begin{array}{l}
 \{ \quad (I = n) \wedge (B[P] \in \{0, 1\}) \wedge (P = k) \} \\
 \quad / \\
 \{ \quad (I = n + 1) \wedge (B[P] \in \{0, 1\}) \wedge (P = k) \}
 \end{array}
 \tag{1}$$

instruction	sémantique
G,D	déplace la tête de lecture respectivement d'une position à Gauche, à Droite

pré- et post- conditions

$$\begin{aligned}
 & \{ (I = n) \wedge (B[P] \in \{0, 1\}) \wedge (P = k) \} \\
 & \quad \textcolor{red}{D} \\
 & \{ (I = n + 1) \wedge (B[P] \in \{0, 1\}) \wedge (P = k + 1) \} \\
 & \quad \textcolor{red}{G} \\
 & \{ (I = n + 2) \wedge (B[P] \in \{0, 1\}) \wedge (P = (k + 1) - 1 = k) \}
 \end{aligned} \tag{2}$$

instruction	sémantique
0,1	écrit respectivement un 0 (case vide) ou un 1 (baton) à l'emplacement de la tête de lecture

pré- et post- conditions

$$\begin{aligned}
 & \{ \quad (I = n) \wedge (B[P] \in \{0, 1\}) \wedge (P = k) \} \\
 & \quad \quad \quad \textcolor{red}{0} \\
 & \{ \quad (I = n + 1) \wedge (B[P] = 0) \wedge (P = k) \} \\
 & \quad \quad \quad \textcolor{red}{1} \\
 & \{ \quad (I = n + 2) \wedge (B[P] = 1) \wedge (P = k) \}
 \end{aligned} \tag{3}$$

instruction	sémantique
<pre> si (0) % n x1 % n1+ ... % ... } % nj+ </pre>	si la tête de lecture est sur une case vide alors les instructions x1, x2, ... sont exécutées en séquence, sinon exécute la première instruction qui suit l'accolade fermante }

pré- et post- conditions

$$\begin{aligned}
 & \{ (I = n) \wedge (B[P] \in \{0, 1\}) \wedge (P = k) \} \\
 & \quad \textcolor{red}{si(0) \% n} \\
 & \{ (I = n + 1) \wedge (B[P] = 0) \wedge (P = k) \} \\
 & \quad \textcolor{red}{x1 \% n + 1} \\
 & \{ (I = n + 2) \wedge (B[P] \in \{0, 1\}) \wedge (P \in \mathbb{N}) \} \\
 & \quad \textcolor{red}{\dots} \\
 & \quad \textcolor{red}{\} \% n + j} \\
 & \{ (I = n + j + 1) \wedge (B[P] \in \{0, 1\}) \wedge (P \in \mathbb{N}) \}
 \end{aligned} \tag{4}$$

instruction	sémantique
$\text{si } (1) \quad \% \ n$ $x1 \ \% \ n1+$ $\dots \% \dots$ $\} \ \% \ nj+$	si la tête de lecture est sur une case vide alors les instructions $x1, x2, \dots$ sont exécutées en séquence, sinon exécute la première instruction qui suit l'accolade fermante $\}$

pré- et post- conditions

$$\begin{aligned}
 & \{ \quad (I = n) \wedge (B[P] \in \{0, 1\}) \wedge (P = k) \} \\
 & \quad \textcolor{red}{si(1) \% n} \\
 & \{ \quad (I = n + 1) \wedge (B[P] = 1) \wedge (P = k) \} \\
 & \quad \textcolor{red}{x1 \% n + 1} \\
 & \{ \quad (I = n + 2) \wedge (B[P] \in \{0, 1\}) \wedge (P \in \mathbb{N}) \} \\
 & \quad \dots \\
 & \quad \textcolor{red}{\} \% n + j} \\
 & \{ \quad (I = n + j + 1) \wedge (B[P] \in \{0, 1\}) \wedge (P \in \mathbb{N}) \}
 \end{aligned} \tag{5}$$

instruction	sémantique
<pre>boucle % n x1 % n1+ ... } % nj+</pre>	répète la séquence d'instructions <code>x1 x2 ...</code> , jusqu'à ce que l'une d'elle soit l'instruction fin

pré- et post- conditions

$$\begin{aligned}
 & \{ (I = n) \wedge (B[P] \in \{0, 1\}) \wedge (P = k) \wedge \text{len}(F) = m \} \\
 & \quad \text{boucle \% } n \\
 & \{ (I = n + 1) \wedge (B[P] \in \{0, 1\}) \wedge (P = k) \\
 & \quad \wedge ((\text{len}(F) = m + 1) \wedge F[m] = I = n + 1)) \} \\
 & \quad x1 \% n + 1 \\
 & \{ (I = n + 2) \wedge (B[P] \in \{0, 1\}) \wedge (P \in \mathbb{N}) \} \\
 & \quad \dots \\
 & \{ (I = n + 1) \wedge (B[P] \in \{0, 1\}) \wedge (P \in \mathbb{N}) \\
 & \quad \wedge ((\text{len}(F) = m + 1) \wedge F[m] = I = n + 1)) \} \\
 & \quad \} \% n + j \\
 & \{ (I = n + j + 1) \wedge (B[P] \in \{0, 1\}) \wedge (P \in \mathbb{N}) \\
 & \quad \text{len}(F) = m \}
 \end{aligned} \tag{6}$$

F est une pile qui contient l'adresse de la première instruction de la boucle tant que l'on exécute la boucle.

instruction	sémantique
<code>fin</code>	rompt le cycle de répétition d'une boucle en faisant exécuter la première instruction qui suit l'accolade fermante de la première boucle contenant l'instruction <code>fin</code> ; si <code>fin</code> n'est contenue dans aucune boucle alors le programme s'arrête.

pré- et post- conditions

$$\begin{aligned}
 & \{ (I = n) \wedge (B[P] \in \{0, 1\}) \wedge (P = k) \\
 & \quad ((len(F) = f) \wedge (f > 0) \wedge (F[f - 1] = x)) \} \\
 & \textcolor{red}{fin} \\
 & \{ (I = x) \wedge (B[P] \in \{0, 1\}) \wedge (P = k) \\
 & \quad \wedge (len(F) = f - 1) \}
 \end{aligned} \tag{7}$$

- ▶ La gestion des boucles implique la présence d'une pile pour gérer les numéro (adresse) des instructions pour retourner à la première instruction du corps de la boucle (rupture de séquence⁴).
- ▶ La pile obéit à une discipline LIFO=*Last In First Out* (dernier entré, premier sorti).
- ▶ Elle permet de gérer un nombre arbitraire de boucles imbriquées.
- ▶ Chaque entrée dans une boucle provoque l'empilement de l'adresse de la première instruction de la boucle et chaque sortie de boucle provoque le dépilement de l'adresse au sommet de la pile.

4. On parle de rupture de séquence lorsque l'ordre d'exécution des instructions d'un programme, qui est par défaut l'ordre de lecture des instructions, l'une après l'autre, par exemple lorsque l'on rencontre l'instruction *fin* la prochaine instruction exécutée sera l'instruction qui suit l'accolade fermante du bloc courant contenant l'instruction *fin*, qui est souvent différente de l'instruction située après l'instruction *fin* dans le programme.

C'est l'idée de base de **l'architecture de von Neumann**⁵, dans le cadre du projet EDVAC1 en juin 1945, il propose la première description d'un ordinateur qui stocke le programme à exécuter dans sa mémoire et a ainsi la possibilité de modifier le champ adresse des instructions pour gérer des boucles. Ce qui revient à considérer les instructions du programme comme des données.

5. https://fr.wikipedia.org/wiki/Architecture_de_von_Neumann

Example assertion python

MTdV

Voici les quatre premières lignes de la sortie de notre analyseur :
appliqué au programme annule_arg.TS :

```
#tokID tok    level
1      "si(0)" 0
2      "I"      1
3      "}"      1
4      "si(1)" 0

# Le modèles python de la machine de Turing :
B = [] # la bande (infinie à droite uniquement),
      # une liste de d'entiers 0 (vide) ou 1 (baton)
P = 0 # la variable entière positive pour la position
      # de la tête de lecture sur la bande
I = 0 # la variable entière positive pour le numéro
      # de l'instruction MTdV
      # qui est en train d'être exécutée
```

Les assertions python attendues par notre programme
de génération d'assertions en python

```
% #tokID tok    level
% assert( P==0 ) ; assert( I==1 ) ;
% assert( (B[ P ] == 0) or (B[ P ] == 1))
1      "si(0)" 0
% assert( P==0 ) ; assert( I==2 ) ; assert( B[ P ] == 0 )
2      "I"      1
% assert( P==0 ) ; assert( I==3 ) ; assert( B[ P ] == 0 )
3      "}"      1
% assert( P==0 ) ; assert( I==4 ) ;
% assert( (B[ P ] == 0) or (B[ P ] == 1))
4      "si(1)" 0
```

Pour savoir comment faire évoluer les assertions
à propos de l'état de la bande pour
ces première lignes de code, il faut identifier
le fait que le token 3 "}" est associé
au token 1 "si(0)".

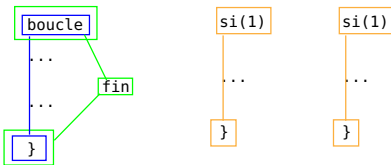
Il faut donc être capable d'identifier les emboîtements
d'instructions provoquant des ruptures de séquence
(sauts dans le programme), donc de gérer une pile
au lieu d'un compteur lors de l'analyse syntaxique.

- ▶ Quelles sont les instructions MTdV qui provoquent des ruptures de séquence (potentielles pour certaines) ?

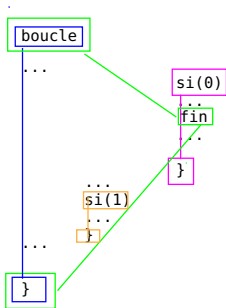
Réponse : `boucle`, `si(0)`, `si(1)`, `fin`

- ▶ Pourquoi avons-nous besoin d'une pile à tout prix ?

Réponse : parce qu'a priori nous n'avons aucune idée à l'avance du nombre d'emboîtement de blocs d'instructions que nous allons rencontrer dans le programme au fil de la lecture de ce dernier. La structure de pile permet de gérer les variations d'emboîtement des blocs d'instruction qui interviennent au fur et à mesure de la lecture du programme.



Influence des contextes dans une structure de boucle aux endroits stratégiques de rupture de séquence.



Parser v2 pour MTdV stack (1/2)

Partie lexicale

La nouvelle version de notre parseur avec une pile et une representation de la structure du programme⁶ est disponible à l'URL :

https://perso.limsi.fr/pap/inalco/MASTER2_2023_2024/mdtv_parser_2_stack.py

6. Ce sont les deux nouveaux arguments supplémentaire de la fonction récursive $P0()$: *stack* et *opdb*

Parser v2 pour MTdV stack (1/2)

Partie lexicale

```
mdtv_parser_3_stack.py - /home/pap/COURSES/inalco_23_24/M2/SEMESTER_1/course_S1_6_20231025/mdtv_parser_3_stack.py (3.8.10)
File Edit Format Run Options Window Help
import re
in_file = 'annule_arg.TS'
#out_file = 'annule_arg_parse.log'
out_file = 'annule_arg_parse.csv'
all_parse_states = [ 'P0' ]
terminals_re = {
    '#' : '^[\n]*#[\n]*',
    '}' : '^[\n]*#[\n]*',
    'I' : '^[\n]*I[\n]*',
    'P' : '^[\n]*P[\n]*',
    'G' : '^[\n]*G[\n]*',
    'D' : '^[\n]*D[\n]*',
    '0' : '^[\n]*0[\n]*',
    '1' : '^[\n]*1[\n]*',
    'fin' : '^[\n]*fin[\n]*',
    'boucle' : '^[\n]*boucle[\n]*',
    'si(0)' : '^[\n]*si[\n]*\([\n]*0[\n]*\)[\n]*',
    'si(1)' : '^[\n]*si[\n]*\([\n]*1[\n]*\)[\n]*'
}

terminals_automata = { x : re.compile( terminals_re[ x ] ) for x in terminals_re.keys() }

def tokenize( in_f, terms_autom = {} ):
    txt = in_f.read()
    txt_sz = len( txt )
    tokens = []
    match = True
    while match and (len( txt ) != 0):
        for tok_nm in terms_autom.keys():
            match = terms_autom[ tok_nm ].search( txt )
            if match:
                (b, e) = match.span( 0 )
                if b == 0:
                    tokens.append( tok_nm )
                    txt = txt[ e: ]
                    break
            else:
                pass
        if match or len( txt ) == 0:
            return tokens
        else:
            print( 'ERROR unknown token encountered in the input at position {}'.format( txt_sz - len( txt ) ) )
            print( 'aborting' )
            exit( 1 )

def next_tok( tokens ):
    if tokens == []:
        print( 'ERROR offset in next_tok() out of range (i.e. >= {})' .format( len( tokens ) ) )
        return ''
    else:
        return tokens[ 0 ]
```

Parser v2 pour MTdV stack (1/2)

Partie syntaxique

```
mdtv_parser_3_stack.py - /home/pap/COURSES/inalco_23_24/M2/SEMESTER_1/course_S1_6_20231025/mdtv_parser_3_stack.py (3.8.10)
File Edit Format Run Options Window Help

def main_parse( terms autom ):
    with open( in_file, 'rt' ) as in_fdesc:
        with open( out_file, 'wt' ) as out_fdesc:
            out_fdesc.write( '#tokID\ttok\tlevel\n' )
            tokens = tokenize( in_fdesc, terms_autom )
            return P0( tokens, out_fdesc )

def P0( tokens, out_f, parse_state = 'P0', K = 0, tokid = 0, stack = [], opdb = {} ):
    assert( stack is not None )
    tok = next tok( tokens )
    tokid += 1
    if ( tok == '#' ) and ( K == 0 ):
        out_f.write( '{0}\t{1}\t{2}\n'.format( tokid, tok, K ))
        opdb[ (tok, tokid) ] = None
        return True ##### stop recursion here
    elif tok in [ 'I', 'P', 'G', 'D', 'O', 'L' ]:
        out_f.write( '{0}\t{1}\t{2}\n'.format( tokid, tok, K ))
        opdb[ (tok, tokid) ] = None
        return P0( tokens[ 1: ], out_f, 'P0', K, tokid, stack, opdb ) ##### P0()
    elif tok in [ 'fin' ]:
        out_f.write( '{0}\t{1}\t{2}\n'.format( tokid, tok, K ))
        opdb[ (tok, tokid) ] = None
        return P0( tokens[ 1: ], out_f, 'P0', K, tokid, stack, opdb ) ##### P0()
    elif tok in [ 'boucle', 'si(0)', 'si(1)' ]:
        out_f.write( '{0}\t{1}\t{2}\n'.format( tokid, tok, K ))
        opdb[ (tok, tokid) ] = None
        return P0( tokens[ 1: ], out_f, 'P0', K + 1, tokid, stack + [ (tok, tokid) ], opdb ) ##### P0()
    elif tok == ' ':
        if K < 1:
            print( 'ERROR unbalanced closing code block token "' )
            return False ##### stop recursion here
        else:
            out_f.write( '{0}\t{1}\t{2}\n'.format( tokid, tok, K ))
            opdb[ (tok, tokid) ] = None
            opdb[ stack[ -1 ] ] = (tok, tokid)
            for i in range( len( stack ) - 1, 0, -1 ):
                if stack[ i ][ 0 ] == 'boucle': # on s'arrête au premier boucle rencontré en remontant dans la pile
                    break
            else:
                if stack[ i ][ 0 ] == 'fin': # il peut y avoir plusieurs "fin" dans le bloc d'une boucle
                    opdb[ stack[ i ] ] = (tok, tokid)
            return P0( tokens[ 1: ], out_f, 'P0', K - 1, tokid, stack[ 0:-1 ], opdb ) ##### P0()
    else:
        print( 'ERROR unexpected end of input' )
        return False ##### stop recursion here

main_parse( terminals_automata )

Ln: 61 Col: 15
```

Parser v3 pour MTdV stack

La nouvelle version finale de notre parseur qui contient maintenant des instructions supplémentaire pour relier les instructions “fin” aux instructions définissant les limites du bloc de code dans lequel elles sont incluse est disponible à l'URL :

https://perso.limsi.fr/pap/inalco/MASTER2_2023_2024/mdtv_parser_3_stack.py

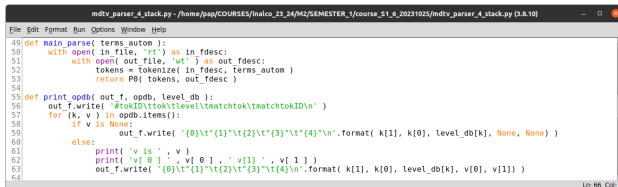
Parser v3 pour MTdV stack vfinale (1/3)

Partie lexicale

```
mdtv_parser_4_stack.py - /home/jap/COURSES/Inalco_23_24/M2/SEMESTER_1/course_S1_6_20231025/mdtv_parser_4_stack.py (3.8.10)
File Edit Format Run Options Window Help
1 import re
2 in_file = 'annule_arg.TS'
3 #out_file = 'annule_arg_parse.log'
4 out_file = 'annule_arg_parse.csv'
5 all_parse_states = [ 'P0' ]
6 terminals_re = {
7     '#' : '^[\n]*#[\n]*' ,
8     '.' : '^[\n]*.[\n]*' ,
9     'I' : '^[\n]*I[\n]*' ,
10    'P' : '^[\n]*P[\n]*' ,
11    'G' : '^[\n]*G[\n]*' ,
12    'D' : '^[\n]*D[\n]*' ,
13    '0' : '^[\n]*0[\n]*' ,
14    '1' : '^[\n]*1[\n]*' ,
15    'fin' : '^[\n]*fin[\n]*' ,
16    'boucle' : '^[\n]*boucle[\n]*' ,
17    'si(0)' : '^[\n]*si[\n]*\n{[\n]*0[\n]*}\n[\n]*' ,
18    'si(1)' : '^[\n]*si[\n]*\n{[\n]*1[\n]*}\n[\n]*' }
19 terminals_automata = { x : re.compile( terminals_re[ x ] ) for x in terminals_re.keys() }
20
21 def tokenize( in_f, terms_autom = {} ):
22     txt = in_f.read()
23     txt_sz = len( txt )
24     tokens = []
25     match = True
26     while match and (len( txt ) != 0):
27         for tok_nm in terms_autom.keys():
28             match = terms_autom[ tok_nm ].search( txt )
29             if match:
30                 (b, e) = match.span( 0 )
31                 if b == 0:
32                     tokens.append( tok_nm )
33                     txt = txt[ e : ]
34                     break
35             if match or len( txt ) == 0:
36                 return tokens
37         else:
38             print( 'ERROR unknown token encountered in the input at position {0}'.format( txt_sz - len( txt ) ) )
39             print( 'aborting' )
40             exit( 1 )
41
42 def next_tok( tokens ):
43     if tokens == []:
44         print( 'ERROR offset in next_tok() out of range (i.e. >= {0})'.format( len( tokens ) ) )
45         return ''
46     else:
47         return tokens[ 0 ]
48
49 def main_parse( terms_autom ):
50     with open( in_file, 'rt' ) as in_fdesc:
```


Parser v3 pour MTdV stack vfinale (2/3)

Fonction principale



```
mdtv_parser_4_stack.py - /home/pap/COURSES/inalco_23_24/M2/SEMESTER_1/course_51_6_20231025/mdtv_parser_4_stack.py (3.8.10)
File Edit Format Run Options Window Help
49 def main_parse( terms_autom ):
50     with open( in_file, 'rt' ) as in_fdesc:
51         with open( out_file, 'wt' ) as out_fdesc:
52             tokens = Tokenize( in_fdesc, terms_autom )
53             return P0( tokens, out_fdesc )
54
55 def print_opdb( out_f, opdb, level_db ):
56     out_f.write( '#tokID\ttok\tlevel\tmatchtok\tmatchtokID\n' )
57     for [k, v] in opdb.items():
58         if v is None:
59             out_f.write( '{0}\t{1}\t{2}\t{3}\t{4}\n'.format( k[1], k[0], level_db[k], None, None ) )
60         else:
61             print( 'v is ', v )
62             print( 'v[ 0 ] ', v[ 0 ], ' v[1] ', v[ 1 ] )
63             out_f.write( '{0}\t{1}\t{2}\t{3}\t{4}\n'.format( k[1], k[0], level_db[k], v[0], v[1] ) )
64
```

Parser v3 pour MTdV stack vfinale (3/3)

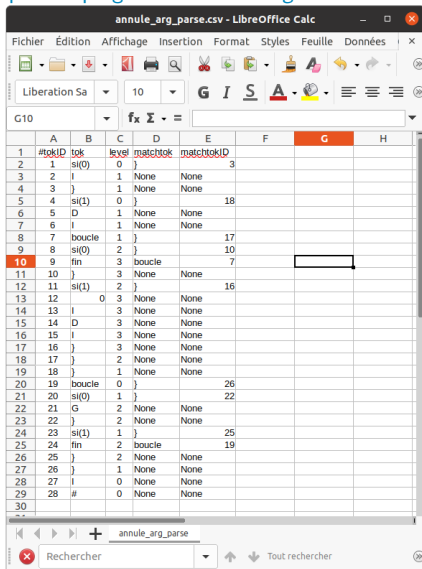
Partie syntaxique

```
"mdtv_parser_4_stack.py - /home/pap/COURSES/Inalco_23_24/M2/SEMESTER_1/course_S1_6_20231025/mdtv_parser_4_stack.py (3.8.10)"
File Edit Format Run Options Window Help
63 out_f.write( '{0}\t{1}\t{2}\t{3}\t{4}\n'.format( k[1], k[0], level_db[k], v[0], v[1] ) )
64
65 def P0( tokens, out_f, parse_state = 'P0', K = 0, tokid = 0, stack = [], opdb = {}, level_db = {} ):
66     assert( stack is not None )
67     print( '==> beg P0() stack == ', stack )
68     print( '==> beg P0() opdb == ', opdb )
69     tok = next_tok( tokens )
70     tokid += 1
71     print( 'in state {0} with new input token {1} and K == {2}\n'.format( parse_state, tok, K ) )
72     if ( tok == '#' ) and ( K == 0 ):
73         opdb[ (tok, tokid) ] = None ; level_db[ (tok, tokid) ] = K
74         print_opdb( out_f, opdb, level_db )
75         return True ##### stop recursion here
76     elif tok in [ 'I', 'P', 'G', 'D', 'O', '1' ]:
77         opdb[ (tok, tokid) ] = None ; level_db[ (tok, tokid) ] = K
78         return P0( tokens[ 1: ], out_f, 'P0', K, tokid, stack, opdb ) ##### P0()
79     elif tok in [ 'fin' ]:
80         opdb[ (tok, tokid) ] = None ; level_db[ (tok, tokid) ] = K
81         return P0( tokens[ 1: ], out_f, 'P0', K, tokid, stack, opdb ) ##### P0()
82     elif tok in [ 'boucle', 'si(0)', 'si(1)' ]:
83         out_f.write( '{0}\t{1}\t{2}\n'.format( tokid, tok, K ) )
84         opdb[ (tok, tokid) ] = None
85         level_db[ (tok, tokid) ] = K
86         return P0( tokens[ 1: ], out_f, 'P0', K + 1, tokid, stack + [ (tok, tokid) ], opdb ) ##### P0()
87     elif tok == '}':
88         if K < 1:
89             print( 'ERROR unbalanced closing code block token "{0}"' )
90             return False ##### stop recursion here
91         else:
92             level_db[ (tok, tokid) ] = K
93             opdb[ stack[ -1 ] ] = (tok, tokid)
94             opdb[ (tok, tokid) ] = None
95             if stack[ -1 ][ 0 ] == 'boucle':
96                 # on remonte dans les tokens, du dernier token au token boucle qui est en sommet de pile (dernière instruction boucle vue).
97                 # (notez que depuis cette dernière instruction boucle, le programme a pu voir passer beaucoup de blocs "si(0)" ou "si(1)"
98                 # qui depuis on été ensuite dépilées, mais dont les instructions "fin" qu'ils pouvaient contenir ne sont pas associés au numéro du
99                 # dernier token "boucle" car nous n'avions pas encore rencontré sa fin (le token courant "}").
100                 for i in range( tokid, stack[ -1 ][ -1 ], -1 ):
101                     # l'assert exprime le fait que nous avons déjà vus passer tous les tokens
102                     # entre le début du bloc et sa fin (le token courant '}')
103                     assert( i in [ k[1] for k in opdb.keys() ] )
104                     if ( 'fin', i ) in opdb.keys():
105                         opdb[ ( 'fin', i ) ] = stack[ -1 ]
106                         return P0( tokens[ 1: ], out_f, 'P0', K - 1, tokid, stack[ 0:-1 ], opdb ) ##### P0()
107             else:
108                 print( 'ERROR unexpected end of input' )
109                 return False ##### stop recursion here
110
111 main_parse( terminals_automata )
112
```

Ln: 89 Col: 0

Parser v3 pour MTdV stack vfinale

Sortie pour le programme *annule_arg.TS*



	A	B	C	D	E	F	G	H
1	#tokID	tok	level	matchtok	matchtokID			
2	1	si(0)	0	}	3			
3	2	I	1	None	None			
4	3	}	1	None	None			
5	4	si(1)	0	}	18			
6	5	D	1	None	None			
7	6	I	1	None	None			
8	7	boucle	1	}	17			
9	8	si(0)	2	}	10			
10	9	fin	3	boucle	7			
11	10	}	3	None	None			
12	11	si(1)	2	}	16			
13	12	0	3	None	None			
14	13	I	3	None	None			
15	14	D	3	None	None			
16	15	I	3	None	None			
17	16	}	3	None	None			
18	17	}	2	None	None			
19	18	}	1	None	None			
20	19	boucle	0	}	26			
21	20	si(0)	1	}	22			
22	21	G	2	None	None			
23	22	}	2	None	None			
24	23	si(1)	1	}	25			
25	24	fin	2	boucle	19			
26	25	}	2	None	None			
27	26	}	1	None	None			
28	27	I	0	None	None			
29	28	#	0	None	None			
30								