Chapitre 2

# TP2 : Analyse lexicale pour JSON

---

Dans ce chapitre, nous allons mettre en place l'outil d'analyse lexicale pour le protocole de communication entre le client et le serveur. Les échanges suivront le formalisme JSON (*JavaScript Object Notation*). Le chapitre suivant se focalisera sur l'analyse syntaxique du langage JSON.

## 2.1 Mise en place

❑ **Exercice 2.1**

1. Récupérer la grammaire du langage JSON sur le site `http://www.json.org`
2. Pour construire un analyseur à base d'un automate à pile, il faut déterminer ce qui sera reconnu par l'analyse lexicale et par l'analyse syntaxique.

☞**À faire** □ Relire le chapitre sur l'analyse lexicale du cours de compilation !

☞**À faire** □ Lister les éléments à reconnaître par l'analyseur lexical.

## 2.2 Analyse lexicale

### 2.2.1 Automates finis

Pour construire l'analyseur lexical, on peut envisager deux possibilités :

— on construit un seul automate à états finis qui contient toutes les entités lexicales à reconnaître. L'avantage de cette solution est que la programmation de cet unique automate est plus facile. Par contre, selon ce qu'on cherche à reconnaître, la mise au point d'un tel automate risque d'être fastidieuse.

— on construit un automate par entité (ou regroupement d'entités). L'avantage de cette solution est que l'on peut par la suite plus facilement ajouter de nouvelles entités à reconnaître. Par contre, la programmation sera plus complexe puisqu'il faudra gérer l'appel successif des automates puis arbitrer celui qui est valide pour la réponse de l'analyseur lexical.

Le langage JSON ne contient pas beaucoup d'entités, tout mettre dans un seul automate devrait être plus simple...

❑ **Exercice 2.2**

Construire (version papier) l'automate permettant de détecter les entités lexicales du langage JSON.

**Remarque 2.1**
Il faut bien sûr s'assurer que l'automate construit est déterministe. S'il est également minimal (en termes de nombre d'états), c'est encore mieux.

## 2.3 Implémentation

Il s'agit maintenant d'écrire le code C correspondant à l'automate.

**❑ Exercice 2.3**
Ecrire un programme C qui contient une fonction `int lex(TLex * _lexData)` qui :
- renvoie un entier qui précise le type de l'entité lexicale reconnue par l'analyseur (penser à utiliser des constantes pour ces types...) ;
- utilise une structure de données (ici appelée `lexData`) qui contient toute les informations nécessaires à l'analyseur : chaîne à analyser, position de l'analyseur dans la chaîne, valeurs des éléments lexicaux reconnus (ce qu'on a appelé « table des symboles » dans le cours).

Il faut également penser à :
- éliminer les espace inutiles de la chaîne d'entrée
- détecter les erreurs et les signaler
- vider la mémoire lorsque l'analyseur n'est plus utile...

**☞À faire ☐** Utiliser le modèle suivant (fichier `tp2_a_a_completer.c`) : les types de données et les prototypes de fonctions utiles sont fournis.

```
1  /**
2   * \file tp2_a.c
3   * \brief analyseur lexical pour le langage JSON
4   * \author NM
5   * \version 0.1
6   * \date 25/11/2015
7   *
8   */
9
10 #include <stdio.h>
11 #include <string.h>
12 #include <ctype.h>
13 #include <stdlib.h>
14 #include <assert.h>
15
16 #define JSON_LEX_ERROR -1 /**< code d'erreur lexicale */
17 #define JSON_TRUE 1 /**< entite lexicale true */
18 #define JSON_FALSE 2 /**< entite lexicale false */
19 #define JSON_NULL 3 /**< entite lexicale null */
20 #define JSON_LCB 4 /**< entite lexicale { */
21 #define JSON_RCB 5 /**< entite lexicale } */
22 #define JSON_LB 6 /**< entite lexicale [ */
23 #define JSON_RB 7 /**< entite lexicale ] */
24 #define JSON_COMMA 8 /**< entite lexicale , */
25 #define JSON_COLON 9 /**< entite lexicale : */
```

```
26 #define JSON_STRING 10 /**<entite lexicale chaine de caracteres
       */
27 #define JSON_INT_NUMBER 11 /**< entite lexicale nombre entier
      */
28 #define JSON_REAL_NUMBER 12 /**< entite lexicale nombre reel */
29
30 /**
31  * \union TSymbole
32  * \brief union permettant de  manipuler un entier/reel/chaine
       pour la table
33  * des symboles
34  */
35 typedef struct {
36         int type; /**< l'un des 3 types suivants : JSON_STRING/
             JSON_INT_NUMBER/JSON_REAL_NUMBER */
37         union {
38         int entier;
39         float reel;
40         char * chaine;
41     } val; /**< valeur associer a un element de la table des
           symboles */
42 } TSymbole;
43
44 /**
45  * \struct TLex
46  * \brief structure contenant tous les parametres/donnees pour
47  * l'analyse lexicale
48  */
49 typedef struct{
50         char * data; /**< chaine a parcourir */
51         char * startPos; /**< position de depart pour la
             prochaine analyse */
52         int nbLignes; /**< nb de lignes analysees */
53         TSymbole * tableSymboles; /**< tableau des symboles :
             chaines/entier/reel */
54         int nbSymboles; /**< taille du tableau tableSymboles */
55 } TLex;
56
57 /**
58  * \fn int isSep(char _symb)
59  * \brief fonction qui teste si un symbole fait partie des
       separateurs
60  *
61  * \param[in] _symb symbole a analyser
62  * \return 1 (vrai) si _symb est un separateur, 0 (faux) sinon
63  */
64 int isSep(const char _symb) {
65 /****** A ECRIRE *******/
66 }
67
```

```
68 /**
69  * \fn TLex * initLexData(char * _data)
70  * \brief fonction qui reserve la memoire et initialise les
71  * donnees pour l'analyseur lexical
72  *
73  * \param[in] _data chaine a analyser
74  * \return pointeur sur la structure de donnees creee
75  */
76 TLex * initLexData(char * _data) {
77 /****** A ECRIRE *******/
78 }
79
80 /**
81  * \fn void deleteLexData(TLex ** _lexData)
82  * \brief fonction qui supprime de la memoire les donnees pour
83  * l'analyseur lexical
84  *
85  * \param[in/out] _lexData donnees de l'analyseur lexical
86  * \return neant
87  */
88 void deleteLexData(TLex ** _lexData) {
89 /****** A ECRIRE *******/
90 }
91
92 /**
93  * \fn void printLexData(TLex * _lexData)
94  * \brief fonction qui affiche les donnees pour
95  * l'analyseur lexical
96  *
97  * \param[in] _lexData donnï¿½es de l'analyseur lexical
98  * \return neant
99  */
100 void printLexData(TLex * _lexData) {
101 /****** A ECRIRE *******/
102 }
103
104
105 /**
106  * \fn void addIntSymbolToLexData(TLex * _lexData, const int
        _val)
107  * \brief fonction qui ajoute un symbole entier a la table des
        symboles
108  *
109  * \param[in/out] _lexData donnees de l'analyseur lexical
110  * \param[in] _val valeur entiere e ajouter
111  * \return neant
112  */
113 void addIntSymbolToLexData(TLex * _lexData, const int _val) {
114 /****** A ECRIRE *******/
115 }
```

```
116
117 /**
118  * \fn void addRealSymbolToLexData(TLex * _lexData, const float
        _val)
119  * \brief fonction qui ajoute un symbole reel a la table des
       symboles
120  *
121  * \param[in/out] _lexData donnees de l'analyseur lexical
122  * \param[in] _val valeur reelle a ajouter
123  */
124 void addRealSymbolToLexData(TLex * _lexData, const float _val)
      {
125 /****** A ECRIRE *******/
126 }
127
128  /**
129  * \fn void addStringSymbolToLexData(TLex * _lexData, char *
       _val)
130  * \brief fonction qui ajoute une chaine de caracteres a la
       table des symboles
131  *
132  * \param[in/out] _lexData donnees de l'analyseur lexical
133  * \param[in] _val chaine a ajouter
134  */
135 void addStringSymbolToLexData(TLex * _lexData, char * _val) {
136 /****** A ECRIRE *******/
137 }
138
139 /**
140  * \fn int lex(const char * _entree, TLex * _lexData)
141  * \brief fonction qui effectue l'analyse lexicale (contient le
       code l'automate fini)
142  *
143  * \param[in/out] _lexData donnï¿½es de suivi de l'analyse
       lexicale
144  * \return code d'identification de l'entite lexicale trouvee
145 */
146 int lex(TLex * _lexData) {
147 /****** A ECRIRE *******/
148 }
149
150
151 /**
152  * \fn int main()
153  * \brief fonction principale
154  */
155 int main() {
156         char * test;
157         int i;
158         TLex * lex_data;
```

```
159
160         test = strdup("{\"obj1\": [ {\"obj2\": 12, \"obj3\":\"
               text1 \\\"and\\\" text2\"},\n {\"obj4\":314.32} ],
               \"obj5\": true }");
161         printf("%s",test);
162         printf("\n");
163
164         lex_data = initLexData(test);
165         i = lex(lex_data);
166         while (i!=JSON_LEX_ERROR) {
167                 printf("lex()=%d\n",i);
168                 i = lex(lex_data);
169         }
170   printLexData(lex_data);
171         deleteLexData(&lex_data);
172         free(test);
173         return 0;
174 }
```

## 2.4 Validation mémoire : Valgrind

### 2.4.1 Motivations

L'outil `valgrind` permet de détecter les erreurs de mémoire (mauvaise réservation/libération, accès à des zones non allouées, etc.) d'un programme pendant son fonctionnement. Si on ne détecte aucune erreur avec cet outil, cela ne garantit pas que le programme fonctionne correctement. Par contre, une erreur détectée par valgrind impliquera des soucis par la suite !

Il est donc prudent de vérifier régulièrement la bonne gestion mémoire de son code. En particulier, en C, lorsque l'on manipule des chaînes de caractères ! Par exemple, il faut s'assurer que les fonctions écrivent dans une zone allouée de taille suffisante (et penser au caractère `'\0'` qui termine les chaînes en C).

Si l'allocation de mémoire est importante, il en est de même de la libération. En effet, lors de la mise au point du code, les tests portent sur peu d'itérations (dans notre cas, on analyse une chaîne JSON et c'est tout), mais lorsque le code sera en production, plusieurs millions d'itérations seront faîtes et la moindre petite erreur de libération s'accumulera lentement mais surement.

### 2.4.2 Mode d'emploi

Pour utiliser correctement valgrind, il faut compiler en incluant les informations de debuggage : cela permet à valgrind, par exemple, de nous donner le nom de la fonction qui pose problème... L'option `-g` permet à `gcc` d'inclure ces informations dans l'exécutable produit.

Par exemple pour le programme de l'exercice 2.3, il faut lancer la commande :

```
gcc -g tp2_a.c -o tp2_a
```

L'outil valgrind se lance en ligne de commande : on lui donne le nom de l'exécutable et on ajoute une option permettant de repérer un maximum de problèmes :
`valgrind ./tp2_a -leak-check=full`

Ensuite, tout un tas d'infomations défilent à l'écran, en plus des messages du programme `tp2_a` mais la dernière ligne résume ce que l'on cherche à obtenir :
`ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)`

❑ **Exercice 2.4**
On peut ajouter un objectif `test` au fichier `makefile`. L'action de ce test serait de compiler le programme avec l'option `-g` puis de lancer `valgrind` sur l'exécutable produit.

☞**À faire** □ Construire un fichier `makefile` permettant de générer l'exécutable de l'exercice 2.3 (commande `make`) ou bien le test avec `valgrind` (commande `make test`).

**Remarque 2.2**
Tant que la ligne suivante n'est pas obtenue au test, il est conseiller de relire son code...
`ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)`

## 2.5 Livrables

— dans un dossier compressé `tp2.zip` : le code de l'analyseur lexical (fichier `tp2_a.c` et `makefile`), le fichier `refman.pdf`.

## 2.6 Conclusion

**Ce chapitre aura permis :**
— passer de la théorie (les automates finis) à la pratique en programmant leur fonctionnement ;
— être capable de reconnaître les entités lexicales du format JSON ;
— être capable de mettre en place des tests avec valgrind

**Pour aller plus loin :**
— La gestion des erreurs est utile dans la phase de mise au point. Malheureusement, on termine souvent par cette étape de programmation...
— pour tester un peu plus intensément le programme, il faudrait lire la chaîne JSON dans un fichier passé en paramètre du programme `tp2_a` pour ainsi lancer plus de tests sur des volumes plus importants..