

TP3 : Analyse syntaxique pour JSON

En reprenant la grammaire JSON, il s'agit, maintenant que l'analyse lexicale est capable de reconnaître les entités les plus simples, d'utiliser ces retours pour vérifier qu'une séquence est correcte. Puis, de stocker les données transmises dans une structure.

3.1 Préparation

□ Exercice 3.1

Reprendre la grammaire JSON et simplifier les notations pour n'avoir que des lettres (*O* pour object,...) et récrire la grammaire. Par exemple :

- object : *O*
- members : *M*
- pair : *P*
- string : *s*
- value : *V*
- array : *A*
- elements : *E*
- number : *n*
- true : *t*
- false : *f*
- null : *u*

□ Exercice 3.2

À partir de la grammaire, construire un analyseur syntaxique en utilisant la technique de construction d'une table SLR vue en cours :

1. augmenter la grammaire
2. construire l'automate (construction des collections canonique des ensembles d'items)
3. construire la table (avec les deux parties Goto et Action)

3.2 Mise en œuvre de l'analyse syntaxique

À partir de la table SLR précédemment construite, il s'agit de programmer l'automate à pile capable d'utiliser cette table pour mener à bien un analyse syntaxique.

Rappelons que c'est l'analyseur lexical, développé au chapitre précédent qui fournira tous les éléments de la grammaire, le rôle de l'analyseur syntaxique étant de vérifier que ces éléments sont correctement agencés entre eux.

Au delà de la vérification syntaxique, l'objectif de l'analyseur syntaxique est de stocker les informations lues dans le texte source sous une forme plus facilement exploitable par

la suite. Dans le cas de notre jeu, il s'agit de décoder le contenu d'un texte JSON pour en extraire des informations sur la situation du jeu (état du joueur courant, état des adversaires...). Ces informations seront utilisées par le bot pour élaborer sa réponse et le coup qu'il propose pour le tour suivant.

La structure d'un fichier JSON est hiérarchique (des éléments contiennent des éléments qui peuvent être des nombres... ou des éléments, etc.), il est donc pratique de stocker les informations lues dans une structure qui reprend cette hiérarchie tel qu'un arbre.

Avant d'écrire le code de l'automate à pile de l'analyseur syntaxique LR, il faut se munir d'outils :

- des structures de données et des fonctions de gestion d'une pile pour le fonctionnement de l'analyseur ;
- des structures de données et des fonctions de gestion d'un arbre pour stocker le contenu « JSON » reconnu par l'analyseur.

3.2.1 Gérer une pile

Comme son nom l'indique, un automate à pile utilise une pile ! L'exercice suivant permet de se constituer une bibliothèque de fonctions qui seront pratiques par la suite...

□ Exercice 3.3

On considère le fichier `pile.h` :

```

1  /**
2   * \file pile.h
3   * \brief gestion d'une pile
4   * \author NM
5   * \version 0.1
6   * \date 11/12/2015
7   *
8   */
9
10 #ifndef __PILE_H__
11 #define __PILE_H__
12
13
14 /** pile contenant des entiers */
15 /**
16  * \struct TIntPile
17  * \brief structure contenant une pile d'entiers
18  */
19 typedef struct {
20     int * data; /**< tableau d'entiers representant la pile */
21     int indexSommet; /**< indice du sommet */
22     int size; /**< taille en mémoire de la pile */
23 } TIntPile;
24
25
26 TIntPile * initIntPile();
27 void deleteIntPile(TIntPile ** _pile);
28 void printIntPile(TIntPile * _pile);

```

```
29 void empilerInt(TIntPile * _pile, int _val);
30 int depilerInt(TIntPile * _pile);
31 int sommetInt(TIntPile * _pile);
32
33 /** pile contenant des pointeurs sur des objets */
34
35 /**
36  * \struct TVoidPile
37  * \brief structure contenant une pile de pointeur void *
38  */
39 typedef struct {
40     void ** data; /**< tableau de pointeur void * */
41     int indexSommet; /**< indice du sommet */
42     int size; /**< taille en memoire de la pile */
43 } TVoidPile;
44
45
46 TVoidPile * initVoidPile();
47 void deleteVoidPile(TVoidPile ** _pile);
48 void printVoidPile(TVoidPile * _pile);
49 void empilerVoid(TVoidPile * _pile, void * _val);
50 void * depilerVoid(TVoidPile * _pile);
51 void * sommetVoid(TVoidPile * _pile);
52
53 #endif
```

Complétez le fichier `pile_a_completer.c` suivant :

```
1  /**
2  * \file pile.c
3  * \brief gestion d'une pile
4  * \author NM
5  * \version 0.1
6  * \date 11/12/2015
7  *
8  */
9  #include <stdio.h>
10 #include <stdlib.h>
11 #include <assert.h>
12 #include "pile.h"
13
14
15 /** constante pour la taille par default d'une pile (utilise
    pour
16  * la reservation memoire) */
17 #define _DEFAULT_PILE_SIZE 15
18
19
20
21 /** pile d'entier
    _____*/
```

```
22 /**
23  * \fn TPile * initPile()
24  * \brief fonction qui initialise une pile : reservation
      memoire+initialisation
25  * des champs
26  *
27  * \return pointeur sur TIntPile cree en memoire
28  */
29 TIntPile * initIntPile() {
30 /* A ECRIRE */
31 }
32
33
34 /**
35  * \fn void deleteIntPile(TIntPile ** _pile)
36  * \brief efface la memoire occupe par la pile
37  *
38  * \param[in/out] _pile : l'adresse du pointeur sur la pile
39  * \return neant
40  */
41 void deleteIntPile(TIntPile ** _pile) {
42 /* A ECRIRE */
43 }
44
45
46 /**
47  * \fn void printIntPile(TIntPile * _pile)
48  * \brief affichage du contenu d'une pile
49  *
50  * \param[in] _pile : la pile a afficher
51  * \return neant
52  */
53 void printIntPile(TIntPile * _pile) {
54 /* A ECRIRE */
55 }
56
57 /**
58  * \fn void empilerInt(TIntPile * _pile, int _val)
59  * \brief empiler un entier sur la pile, si la zone memoire
      reservee
60  * n'est pas suffisante, celle-ci est etendue
61  *
62  * \param[in/out] _pile : la pile a utiliser
63  * \param[in] _val : l'entier a empiler
64  * \return neant
65  */
66 void empilerInt(TIntPile * _pile, int _val) {
67 /* A ECRIRE */
68 }
69
```

```
70 /**
71  * \fn int depilerInt(TIntPile * _pile)
72  * \brief dÃ©piler un entier
73  *
74  * \param[in] _pile : la pile a depiler
75  * \return l'entier en sommet de pile (0 si la pile est vide)
76  */
77 int depilerInt(TIntPile * _pile) {
78     /* A ECRIRE */
79 }
80
81 /**
82  * \fn int sommetInt(TIntPile * _pile)
83  * \brief renvoie la valeur du sommet (sans depiler)
84  *
85  * \param[in] _pile : la pile a utiliser
86  * \return l'entier en sommet de pile (0 si la pile est vide)
87  */
88 int sommetInt(TIntPile * _pile) {
89     /* A ECRIRE */
90 }
91
92
93 /** pile de void *
94     _____*/
95
96 /**
97  * \fn TVoidPile * initVoidPile()
98  * \brief fonction qui initialise une pile de pointeur void *
99  *
100  * \return pointeur sur une pile TVoidPile
101  */
102 TVoidPile * initVoidPile() {
103     /* A ECRIRE */
104 }
105
106 /**
107  * \fn void deleteVoidPile(TVoidPile ** _pile)
108  * \brief libere la memoire occupee par la pile
109  *
110  * \param[in] _pile : adresse du pointeur sur la pile a liberer
111  * \return neant
112  */
113 void deleteVoidPile(TVoidPile ** _pile) {
114     /* A ECRIRE */
115 }
116
117
118 /**
```

```
119 * \fn void printVoidPile(TVoidPile * _pile)
120 * \brief affichage de la pile
121 *
122 * \param[in] _pile : pile a afficher
123 * \return neant
124 */
125 void printVoidPile(TVoidPile * _pile) {
126 /* A ECRIRE */
127 }
128
129 /**
130 * \fn void empilerVoid(TVoidPile * _pile, void * _val)
131 * \brief empile un void *
132 *
133 * \param[in/out] _pile : pile a utiliser pour empiler
134 * \param[in] _val : element de type void * a empiler
135 * \return neant
136 */
137 void empilerVoid(TVoidPile * _pile, void * _val) {
138 /* A ECRIRE */
139 }
140
141 /**
142 * \fn void * depilerVoid(TVoidPile * _pile)
143 * \brief dÃ©pile un Ã©lÃ©ment de type void *
144 *
145 * \param[in] _pile : pile a utiliser
146 * \return pointeur sur void (0 si la pile est vide)
147 */
148 void * depilerVoid(TVoidPile * _pile) {
149 /* A ECRIRE */
150 }
151
152 /**
153 * \fn void * sommetVoid(TVoidPile * _pile)
154 * \brief obtenir la valeur du sommet de type void *
155 *
156 * \param[in] _pile : pile a utiliser pour lire le sommet
157 * \return la valeur void * du sommet (0 si la pile est vide)
158 */
159 void * sommetVoid(TVoidPile * _pile) {
160 /* A ECRIRE */
161 }
162
163
164 x/** code pour test */
165 #ifndef TEST
166 /**
167 * \fn int main(void)
168 * \brief fonction principale utilisee uniquement en cas de
```

```
tests
169  *
170  */
171  int main(void) {
172      int i;
173      {
174          /* tests pour un pile d'entier */
175          TIntPile * p = NULL;
176
177          printf("-----\ntest pour
              une pile d'entier\n");
178          //empilerInt(p,99);
179          printIntPile(p);
180          p = initIntPile();
181          printIntPile(p);
182          for ( i=0;i<35;i++) {
183              empilerInt(p,sommetInt(p)+i);
184              printIntPile(p);
185          }
186          for ( i=0;i<40;i++) {
187              int r = depilerInt(p);
188              printf("r=%d\n",r);
189              printIntPile(p);
190          }
191          deleteIntPile(&p);
192      }
193      /* tests pour un pile de void * */
194      {
195          TVoidPile * q = NULL;
196          int a = 321;
197          char * b = "azerty";
198
199          printf("-----\ntest pour
              une pile de void *\n");
200          //empilerVoid(q,&a);
201          printVoidPile(q);
202          q = initVoidPile();
203          printVoidPile(q);
204          empilerVoid(q,&a);
205          printVoidPile(q);
206          empilerVoid(q,&b);
207          printVoidPile(q);
208          empilerVoid(q,&a);
209          printVoidPile(q);
210          empilerVoid(q,q);
211          printVoidPile(q);
212          printf("depiler : %p\n",depilerVoid(q));
213          printVoidPile(q);
214
215          printf("depiler : %p\n",depilerVoid(q));
```

```
216         printVoidPile(q);
217
218         printf("depiler : %p\n",depilerVoid(q));
219         printVoidPile(q);
220
221         printf("depiler : %p\n",depilerVoid(q));
222         printVoidPile(q);
223
224         printf("depiler : %p\n",depilerVoid(q));
225         printVoidPile(q);
226
227         deleteVoidPile(&q);
228
229     }
230 }
231 #endif
```

La pile d'entier servira à gérer les états de l'automate à pile alors que la pile de pointeurs `void *` servira à stocker les objets JSON créés au fur et à mesure de l'analyse.

❑ Exercice 3.4

À partir de la table SLR précédemment construite, programmer l'analyseur syntaxique en C :

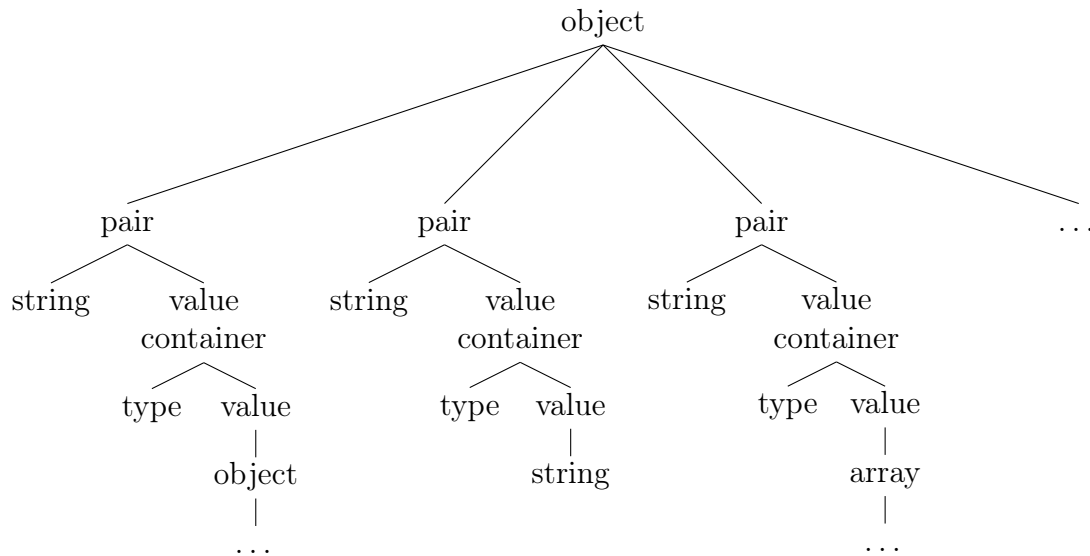
- il faut utiliser des structures de pile et les fonctions associées écrites pour l'exercice 3.3. La pile d'entiers sera utilisée pour stocker les états de la table d'analyse SLR;
- évidemment, le code de l'analyseur lexical, fabriqué au chapitre précédent, est utilisé à chaque étape où l'analyseur syntaxique cherche à lire une entité entrée.

Étapes :

1. Écrire l'algorithme qui correspond au fonctionnement de l'automate LR
2. Écrire un programme `tp3_a.c` qui implémente l'algorithme précédent dans une fonction `int analyseurLR(...)`. Cette fonction renvoie 1 si la chaîne passée en argument suit bien la grammaire JSon et 0 sinon.
3. Tester sur les fichiers `testX.json`

Tel que demandé à l'exercice précédent, l'analyseur syntaxique ne fait qu'analyser un code JSON et doit être capable de dire si la syntaxe JSON est respectée. L'objectif étant de récupérer les informations contenues dans le code JSON, il faut être capable d'enregistrer les données dans une structure adaptée. Le plus immédiat est d'utiliser une structure qui imite la structure hiérarchique du format JSON. Un arbre correspond bien à cette structure.

Le schéma suivant montre la structure de cette arbre en reprenant les termes utilisés pour décrire la syntaxe JSON :



Cet arbre est un exemple incomplet mais on remarque que la couche **value container** sert à encapsuler une valeur de n'importe quel type (un objet JSON, un tableau...).

Le code implémentant cette structure et les fonctions qui correspondent sont fournis dans le fichier `json_tree.c`.

□ Exercice 3.5

À partir de l'analyse LR développée à l'exercice précédent, écrire un programme `tp3_b.c` qui permet la construction en mémoire de l'arbre JSON.

1. Utiliser les structures et les fonctions contenues dans `json_tree.c`
2. Utiliser les fonctions permettant une sortie au format graphviz/dot pour permettre l'affichage de l'arbre JSON
3. L'exécutable obtenu doit prendre comme paramètre le nom du fichier à analyser et être appelé de la façon suivante :

```
./tp3_b -f test1.json
```

4. Tester sur les fichiers `testX.json` (5 jeux de test sont disponibles : `test1.json ... test5.json` avec les versions contenant des erreurs `test1_err.json ... test5_err.json`).

3.3 Livrables

- dans un dossier compressé TP3 : le code de l'analyseur syntaxique : les fichiers `tp3_b.c` (où `tp3_a.c` si vous n'avez pas pu traiter l'exercice 3.5). Attention, pensez à inclure tous les fichiers nécessaires à la compilation (les fichiers `.c` et `.h` pour l'analyseur lexical, la gestion des paramètres de la ligne de commande, la gestion des piles...ainsi que le `makefile`)
- La doc complète (format doxygen) doit être dans le fichier `refman.pdf`
- critères d'évaluation :
 - le code doit être commenté
 - le code sera testé avec Valgrind
 - le programme doit lire le fichier passé en paramètre et en faire l'analyse syntaxique

3.4 Conclusion

Ce chapitre aura permis :

- de mettre en place une analyse syntaxique
- de découvrir le format JSON d'un point de vue syntaxique
- de mettre en pratique la construction d'une table SLR
- de mettre en pratique la programmation d'un automate à pile en le connectant avec l'analyseur lexical construit au chapitre précédent.
- de comprendre l'intérêt de la structure de pile lorsque l'on travaille sur un arbre.

Pour aller plus loin :

- Une fois l'arbre JSON construit en mémoire, il faudra écrire des fonctions permettant d'accéder aux informations contenues dans l'arbre.
- Si le fichier passé en paramètre est correct (c'est-à-dire qu'il respecte la syntaxe JSON), le programme affiche par exemple le message « fichier valide » (exercice 3.4) ou bien l'arbre au format Graphviz/dot (exercice 3.5) mais si le fichier est incorrect, une version améliorée peut signaler le type d'erreur : lexicale/syntaxique. Mieux encore, le programme signale le type d'erreur et la position de l'erreur dans le fichier source) ou alors le programme signale le type, la position de l'erreur et précise ce qui est attendu ou comment corriger.