ESISAR

NE302 – Projet Réseau
Projet « HTTP Server » - part 2

Table des matières

1 Objectifs de l'étape 2:

1.1 Objectif de l'analyse syntaxique (étape 2)

L'analyse syntaxique consiste à vérifier la syntaxe de votre message, mais elle vous permettra aussi de déterminer tous les éléments de la syntaxe d'un message HTTP. Aussi vous recevrez une chaîne de caractères concernant la requête brute, et votre analyseur stockera le résultat de l'analyse dans une structure de données dans laquelle vous allez pouvoir faire des recherches plus facilement. Pour prendre un exemple simple, imaginons que la grammaire Française définisse une phrase comme

phrase = sujet verbe complément .

alors la chaîne de caractères : « le jeune lapin mange une carotte tendre.» sera stockée dans une structure de données tel que la recherche de :

sujet → « le jeune lapin »

verbe → « mange »

complément → « une carotte tendre »

1.2 Objectif de l'analyse syntaxique (étape 3)

Pour simplifier, on peut dire que l'analyse sémantique est de vérifier que la phrase à du sens. Ainsi si une règle sémantique indique qu'un animal mange un végétal, il suffira de vérifier que le verbe est une conjugaison de « manger », le sujet un animal, et le complément un végétal. L'analyse sémantique ne traitera plus la chaîne elle même, mais les éléments fournis par l'analyse syntaxique. Cette phrase donc aura un sens.

1.3 Développement d'un analyseur syntaxique de requêtes HTTP

Vous devrez prendre en compte la grammaire ABNF fournie dans les RFC723x et relatives Cette grammaire définit le format des messages HTTP échangés entre client et serveur.

L'objectif de cette partie est de réaliser un analyseur syntaxique permettant d'extraire certains champs (voire tous) d'une requête HTTP. (comme vous réalisez un serveur HTTP, l'analyse des réponses n'a pas d'intérêt ici.) En effet votre serveur va recevoir un flux d'octets correspondant à des requêtes, vous devrez pouvoir analyser ces requêtes pour en déduire la réponse à faire.

On parle de flux d'octets dans la requête car elle est potentiellement composée de deux parties :
- *une partie « textuelle » concernant la requête et les entêtes*
- *une partie « non textuelle » optionnelle contenant le message body.*

*La première étape consiste à faire **l'analyse syntaxique** de la requête (c'est à dire identifier les champs de la requête textuelle). La deuxième étape consistera (lors de la phase 3) à faire **l'analyse sémantique** de la requête pour en comprendre le sens à partir des éléments trouvés dans l'analyse syntaxique.*

Détails :

Les champs d'une requête seront indiqués par les « rulenames» de la grammaire ABNF fournie dans les Annexes des rfc723x.

Exemple :

*HTTP-message = start-line *(header-field CRLF) CRLF [message-body]*
start-line = request-line / status-line
request-line = method SP request-target SP HTTP-version CRLF

Transfer-Encoding-header = "Transfer-Encoding" ":" OWS Transfer-Encoding OWS
Transfer-Encoding = *("," OWS) transfer-coding *(OWS "," [OWS transfer-coding])
etc....

Ainsi si l'on considère la requête suivante :

```
GET /index.html HTTP/1.0
Host: www.esisar.grenoble-inp.fr
Transfer-Encoding: gzip,chunked
```

Si on recherche le champ start-line ou request-line, votre parseur devra trouver :

```
GET /index.html HTTP/1.0
```

Si on recherche le champ *method*, le parseur devra trouver
GET

Si on recherche le champ *request-target*, le parseur devra trouver
/index.html

Si on recherche le champ *header-field*, le parseur devra trouver les 2 éléments suivants :
Host: www.esisar.grenoble-inp.fr
et
Transfer-Encoding: gzip,chunked

Si on recherche le champ *transfer-coding*, le parseur devra trouver les 2 éléments suivants :
gzip
et
chunked

L'action de trouver des champs dans une requête est aussi appelée « parsing »

2 Détail du travail demandé

2.1 Familiarisez-vous avec la grammaire ABNF.

La grammaire ABNF est décrite dans la RFC5234, vous trouverez d'ailleurs sa description formelle dedans et devinez ? au format ABNF bien sûr (sinon ce n'est pas drôle) !! Lisez les chapitres 2 et 3 de cette RFC (7 pages).

2.2 Arbre de dérivation syntaxique.

Vous devrez réaliser un parseur qui effectue une validation syntaxique de votre message.

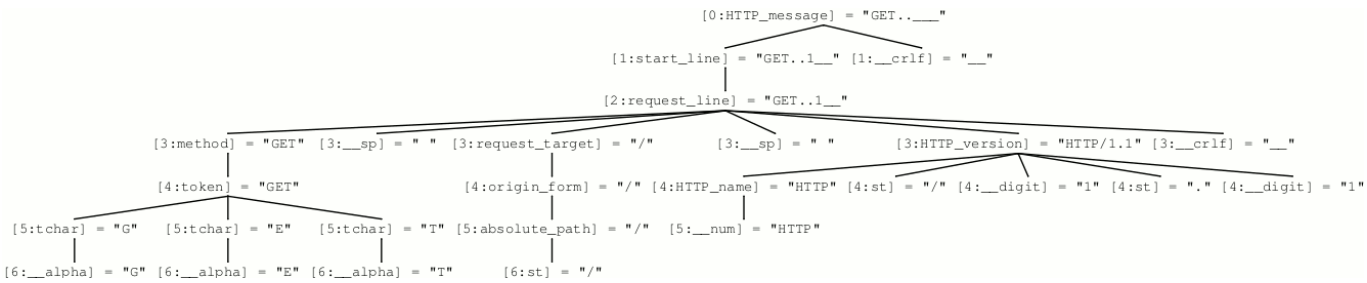
Comme dit plus haut, l'analyseur syntaxique va stocker dans uns structure adapté les différents éléments de la grammaire ABNF, cette structure sous forme arborescente est appelé arbre de dérivation syntaxique.

Afin de vous guider dans ce travail nous vous demandons de réaliser l'arbre de dérivation syntaxique de la requête HTTP.

Exemple d'arbres pour un requête simple GET / HTTP/1.1:

note1 : Le contenu est abrégé pour plus de lisibilité. Les CRLF sont remplacés par des __

note2 : Notez déjà la complexité de l'arbre pour une requête si petite…



Vous trouverez sur chamilo d'autres exemples dans l'étape 2.

On notera ici que :

- Cet arbre est constitué de nœuds intermédiaires, et de nœuds terminaux.
- Chaque nœud peut avoir un nombre différent de fils, il n'y a potentiellement pas de « limite » de nombre de fils….
- Un nœud intermédiaire ou terminal possède :
  - une étiquette (Tag) (nom de la « rulename » de la grammaire ABNF)
  - une valeur (Value) (partie (quelques caractères) du contenu de la requête HTTP)
- Certaines branches n'ont pas de ramifications, donc susceptibles d'être « simplifiées ».

2.3 Réalisation

Réaliser l'arbre complet de dérivation est un travail conséquent qui nécessite une réflexion préliminaire avant de se lancer dans l'aventure…. Voici quelques principes ou conseils qui vous permettront d'orienter votre réflexion. Pensez dès maintenant comment vous devrez/pourrez utiliser ce parseur dans l'intégralité de votre projet, notamment quand et comment votre code appellera ce parseur, que devra t-il vous retourner ? Il est nécessaire de réaliser l'algorithme de votre programme afin de définir l'interface de votre parseur. Encore une fois privilégier l'utilisabilité plutôt que la performance.

Réfléchissez-y. Réfléchissez-y encore !!

1 - L'analyse syntaxique peut se faire en descendant, en gardant l'indice de lecture dans la requête HTTP, par une technique appelée « recursive descent » chaque rulename de la grammaire peut être associée à une fonction retournant vrai ou faux. Pour retourner la valeur, un nœud terminal validera directement la validité ou non, un nœud intermédiaire procédera à l'appel d'une ou plusieurs autres fonctions correspondant aux nœuds fils (soit terminaux, soit eux mêmes intermédiaires…) et ainsi de suite….

On remarquera donc une similitude très grande entre la grammaire ABNF, et le code de la fonction.

par exemple, une alternative sera un « if », une répétition sera un « for ». Il en découlera donc des « pattern» de code simple à mettre en œuvre et répétitifs….

NOTE : Les plus aventureux pourront imaginer générer du code de manière automatique ou semi-automatique…

2- L'analyse se fait donc en descendant, et la validation en remontant, en descendant, le nœud s'insère dans l'arbre.

- Quand un nœud retourne vrai, il reste dans l'arbre (ça c'est facile)
- Quand un nœud retourne faux, il doit se supprimer de l'arbre (lui et tous ses fils s'il y en a) et faire revenir l'indice de lecture dans l'état initial d'avant son appel (on « backtrack ») pour éventuellement tenter une autre branche. (au fait !?? on pense à libérer la mémoire aussi….)
- On considère qu'il n'y a pas d'erreur de syntaxe si la règle racine retourne vrai.

3 - On considère que la requête HTTP est stockée en mémoire dans un espace mémoire contigu correspondant à la suite des octets de cette requête. Ainsi représenter un champ particulier dans la requête consiste à garder la référence d'un pointeur vers char et d'une longueur.Exemple d'une requête :

G	E	T		/	i	n	d	e	x	.	h	t	m	l		H	T	T	P	/	1	.	0	\r	\n	H	o	s	t	:	
w	w	w	.	t	o	t	o	.	c	o	m	\r	\n	\r	\n																

Le nœud intermédiaire *method* consiste à avoir :

un pointeur vers des char dont la valeur pointe sur et un entier de valeur 3

Le nœud intermédiaire *request-target consiste à avoir*

un pointeur vers des char dont la valeur pointe sur et un entier de valeur 11

Vous remarquerez donc qu'il n'est pas nécessaire de faire des recopies en mémoire !! Il y en a qui ont essayé, ils ont eu des problèmes !! Je répète, ils ont eu des PROBLEMES !!!

4 – Vous avez de la liberté concernant la manière dont vous allez implémenter en C votre code et votre arbre, mais réfléchissez-y bien….. Un nœud devra au minimum conserver une référence sur l'étiquette (le nom de la règle) et sur la valeur (le contenu pointé)

5- Vous pouvez simplifier la grammaire ABNF en considérant certains nœuds intermédiaires comme terminaux afin de réduire le nombre de fonctions à coder.

3 Périmètre du travail

3.1 Type de message

Vous pourrez faire quelques hypothèses concernant le type de message que vous allez recevoir, en effet en tant que serveur vous ne recevrez que des requêtes, vous pouvez donc supprimer la branche de la grammaire « status-line »

start-line = request-line

De même, en tant que serveur vous ne recevrez pas de request-target suivant la branche « absolute-form » ou «authority-form » Cf RFC7230 (5.3)

request-target = origin-form / asterisk-form

3.2 Entêtes HTTP

La RFC 7230 3.2.4 indique :

> *Messages are parsed using a generic algorithm, independent of the individual header field names. The contents within a given field value are not parsed until a later stage of message interpretation (usually after the message's entire header section has been processed). Consequently, this specification does not use ABNF rules to define each "Field-Name: Field Value" pair, as was done in previous editions. Instead, this specification uses ABNF rules that are named according to each registered field name, wherein the rule defines the valid grammar for that field's corresponding field values (i.e., after the field-value has been extracted from the header section by a generic field parser).*

Comme nous ne respecterons pas cette recommandations, dans notre cas nous utiliserons une autre définition de la règle header-field, cette définition indiquera tous les entêtes que votre parseur devra au minimum être en mesure de comprendre :

Il est donc obligatoire d'utiliser la RFC disponible sur chamilo allrfc.html (cliquable si téléchargée) plutôt que tout autre référence à une grammaire ABNF sur Internet.

```
Connection-header = "Connection" ":" OWS Connection OWS
Content-Length-header = "Content-Length" ":" OWS Content-Length OWS
Content-Type-header = "Content-Type" ":" OWS Content-Type OWS
Cookie-header = "Cookie" ":" OWS cookie-string OWS
Transfer-Encoding-header = "Transfer-Encoding" ":" OWS Transfer-Encoding OWS
Expect-header = "Expect" ":" OWS Expect OWS
Host-header = "Host" ":" OWS Host OWS


header-field =  Connection-header /
        Content-Length-header /
        Content-Type-header /
        Cookie-header /
        Transfer-Encoding-header /
        Expect-header /
        Host-header /
        ( field-name ":" OWS field-value OWS )
```

4 API pour la validation et l'évaluation de votre parseur

Cette partie est à lire à partir du moment ou vous avez bien compris les parties précédentes.

Comme nous ne connaîtrons pas le type de structure en C que vous utilisez pour votre arbre, nous utiliserons une API décrite ci-dessous utilisant des types « opaques » (on fournit le fichier .h, vous codez le fichier api.c correspondant à votre implémentation).

Nous allons devoir tester votre parseur. Pour cela nous allons rechercher dans votre arbre de dérivation, les nœuds avec une certaine étiquette. La fonction searchTree (que vous coderez, bah oui !! il n'y a que vous qui connaissez les choix d'implémentation) permet cette recherche :

- pour chaque nœud trouvé (à partir de la racine ou non), vous l'ajouterez dans une liste chaînée (de _Token) (champ void *node).

- la fonction « retourne » la liste chaînée une fois tout l'arbre parcouru.

- cette liste chaînée n'est finalement qu'une succession de pointeurs vers void...

- *« C'est bien M. Michu, mais maintenant j'en fais quoi des void * ??? »*

Bien je les passe à la fonction getElementTag et getElementValue pour que ces fonctions me retournent des char *

- *« Ah OK OK OK OK ! »*

fichier api.h (vous devez réaliser l'implémentation api.c)

```
// defini un type pour la liste chainee renvoyée en réponse de  la requete de recherche dans l'arbre.
typedef struct _token {
                        void *node;                 // node type opaque pointant vers un noeud de votre arbre.
                        struct _token *next;        // next pointe vers le prochain token.
                        } _Token;

// Fonction qui retourne un pointeur (type opaque) vers la racine de l'arbre construit.
void *getRootTree();

// Fonction qui recherche dans l'arbre tous les noeuds dont l'etiquette est egale à la chaine de caractères en argument.
// Par convention si start == NULL alors on commence à la racine
// sinon on effectue une recherche dans le sous-arbre à partir du noeud start
_Token *searchTree(void *start,char *name);

// fonction qui renvoie un pointeur vers char indiquant l'etiquette du noeud. (le nom de la rulename, intermediaire ou terminal)
// et indique (si len!=NULL) dans *len la longueur de cette chaine.
char *getElementTag(void *node,int *len);

// fonction qui renvoie un pointeur vers char indiquant la valeur du noeud. (la partie correspondnant à la rulename dans la requete HTTP )
// et indique (si len!=NULL) dans *len la longueur de cette chaine.
char *getElementValue(void *node,int *len);
```

```
// Fonction qui supprime et libere la liste chainée de reponse.
void purgeElement(_Token **r);

// Fonction qui supprime et libere toute la mémoire associée à l'arbre .
void purgeTree(void *root);

// L'appel à votre parser un char* et une longueur à parser.
int parseur(char *req, int len);
```

5 Recommandations

Le code que vous allez réalisé va être relativement important, voici donc quelques conseils concernant le développement en C de votre application:

- Il est obligatoire que le code fonctionne sur les stations de TP des salles B120,A131 ou B141. Une machine virtuelle est disponible en cas de doute.

- Il est indispensable d'avoir une maitrîse de quelques outils : éditeur, versionning.

- L'utilisation de gdb (option de compilation -g) est un premier point important, cependant vous allez rapidement voir que la plupart du temps les erreurs complexes de gestion de mémoire, overflow du tas ou de la pile, use after free, etc. deviennent très difficile à débugger avec cet outils.

- Il est conseillé d'utiliser deux autres outils :

1) l'instrumentation du code avec : https://github.com/google/sanitizers/wiki/AddressSanitizer

2) ou valgrind qui permet aussi de débugger ce genre de problèmes.

Le temps d'apprentissage de ces outils n'est pas une perte de temps , croyez-moi…

6 Exemple de code main.c

Code donné à titre d'exemple, vérifiez les mises à jour sur chamilo.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "api.h"

/// Incluez ici les fichiers d'entête nécessaires pour l'execution de ce programme.
/// La fonction parseur doit être dans un autre fichier .c
/// N'ajouter aucun autre code dans ce fichier.

int main(int argc, char *argv[])
{
        char req[]="GET / HTTP/1.0\r\nHost: www.google.com\r\nTransfer-Encoding: gzip\r\n\r\n";
        if (argc != 2 ) { printf("usage: %s <rulename>\n",argv[0]); return 0; }

        if (parseur(req,strlen(req))) {
                _Token *r,*tok;
```

```
                        void *root ;
                        root=getRootTree();
                        r=searchTree(root,argv[1]);
                        tok=r;
                        while (tok) {
                                int l;
                                char *s;
                                s=getElementValue(tok->node,&l);
                                printf("FOUND [%.*s]\n",l,s);
                                tok=tok->next;
                        }
                        purgeElement(&r);
                        purgeTree(root) ;
        }
        else {
                        return 0;
        }
        return 1;
}
```

7 Tests

Un jeu de test vous sera fourni (requêtes, champ à trouver, et résultats) afin de tester votre parseur, un autre jeu de test pourra être utilisé pour l'évaluation.