Groupe de Projet
Thomas Moulard
Taha Karim
Julien Assemat

Table des matières

Introduction

Présentation Générale du Projet [2 min]

Avancement Général du Projet

Avancement Général du Site Web

Avancement Prévu entre la Soutenance 1 et 2

Problèmes Rencontrés [1 min]

Groupe

Réalisation

Présentation des Tâches Individuelles [12 min]

Tâches effectuées par Taha

Tâches effectuées par Julien

Tâches effectuées par Thomas

Conclusion

1 Introduction

La première partie de notre projet est maintenant achevée. *EpiDatabase*, le système de gestion de base de données qui est l'objet de notre projet, est avant tout un développement basé sur les principes fondamentaux de ce genre d'applications. En effet, une telle application est destinée au final à fonctionner rapidement et surtout efficacement sur les plateformes Windows.

Pour cette première étape il a d'abord été question pour chacun de nous d'effectuer un travail de recherche afin de mener à bien nos tâches respectives. La conception d'applications de ce genre est loin d'être la plus courante, nous avons dû faire face à une bonne part d'inconnu pour poser les bases d'EpiDatabase.

2 Nature du Projet

Comme son nom le suggère, EpiDatabase n'est rien d'autre qu'un système de gestion de base de données (plus communément appelé SGBD). Plus concrêtement, il s'agit d'une application capable de stocker et de traiter des informations. A elle seule, une application d'un tel type n'aurait pas grand intérêt. En effet, elle ne représente que la partie immergée de l'iceberg, comme la plupart des SGBD. Pour réellement en voir l'efficacité, il faut en fait la coupler à une application qui elle, a besoin des données de la base de données.

Ainsi selon les actions demandées au SGBD, des ajouts, des suppressions, des modifications ou encore des séléctions d'enregistrements vont être effectuées et seront disponibles à tout instant pour l'application qui en dépend.

Pour arriver à réaliser notre SGBD, nous devons inclure divers éléments primordiaux, expliqués ci-dessous.

3 Présentation du Projet

3.1 Fonctionnement du Projet

3.2 Eléments constituant le Projet

La réalisation d'EpiDatabase nécessitera la mise en relation de plusieurs composants tels que la gestion des fichiers, la grammaire des requêtes, la reconnaissance de cette grammaire, une application réseau faisant le lien avec tous les programmes dépendant du SGBD.

3.2.1 Gestionnaire de Fichiers

Le premier constituant, et surtout le plus important, est le gestionnaire de fichiers. De part sa structure, il nous permettra de traiter et de réorganiser de la meilleure manière possible l'ensemble des informations stockées. C'est à ce niveau-là qu'interviendront le résultat des requêtes reçues par l'application, à savoir suppression, ajout, modification ou encore sélection d'enregistrements. La structure sera plus amplement développée dans les sections individuelles.

3.2.2 Requêtes

Les requêtes sont en quelque sorte les commandes permettant la mise en relation entre l'application qui demande les informations et le SGBD qui va les traiter et les lui envoyer. Plus grossièrement, les requêtes permettent de savoir ce que veut l'application.

Comme cela a déjà été précisé dans le cahier des charges, les requêtes suivront la grammaire SQL, avec plus ou moins de simplifications sur certaines fonctionnalités. Ce langage n'a pas été choisi par hasard, en effet, il a largement démontré son efficacité parmi les plus grands SGBD de par sa logique et sa simplicité, qui font de cette grammaire un langage efficace.

3.2.3 Parseur

Le parseur n'est rien d'autre qu'un analyseur syntaxique. Pour notre projet, il servira à reconnaître la grammaire SQL, utilisée principalement pour les requêtes : en effet, il faut que le SGBD soit en mesure de comprendre les *ordres* que lui envoie le programme qui veut les informations.

Pour la réalisation de ce composant, nous utiliserons le programme Bison, qui, à partir de la définition de la grammaire qu'on lui donne, génère automatiquement les fichiers .c permettant de parser n'importe quelle requête syntaxiquement correcte.

3.2.4 Lexeur

Le lexeur est le constituant indispensable au parseur pour une application telle qu'Epi-Database. Couplé au parseur, il permet de reconnaître les motifs de la grammaire dans une reqûete par exemple et à partir de là, il exécute du code selon le motif trouvé. De même que précédemment, nous utiliserons le meilleur allié de Bison, à savoir Flex, dont nous aborderons la programmation et les principes dans les sections individuelles.

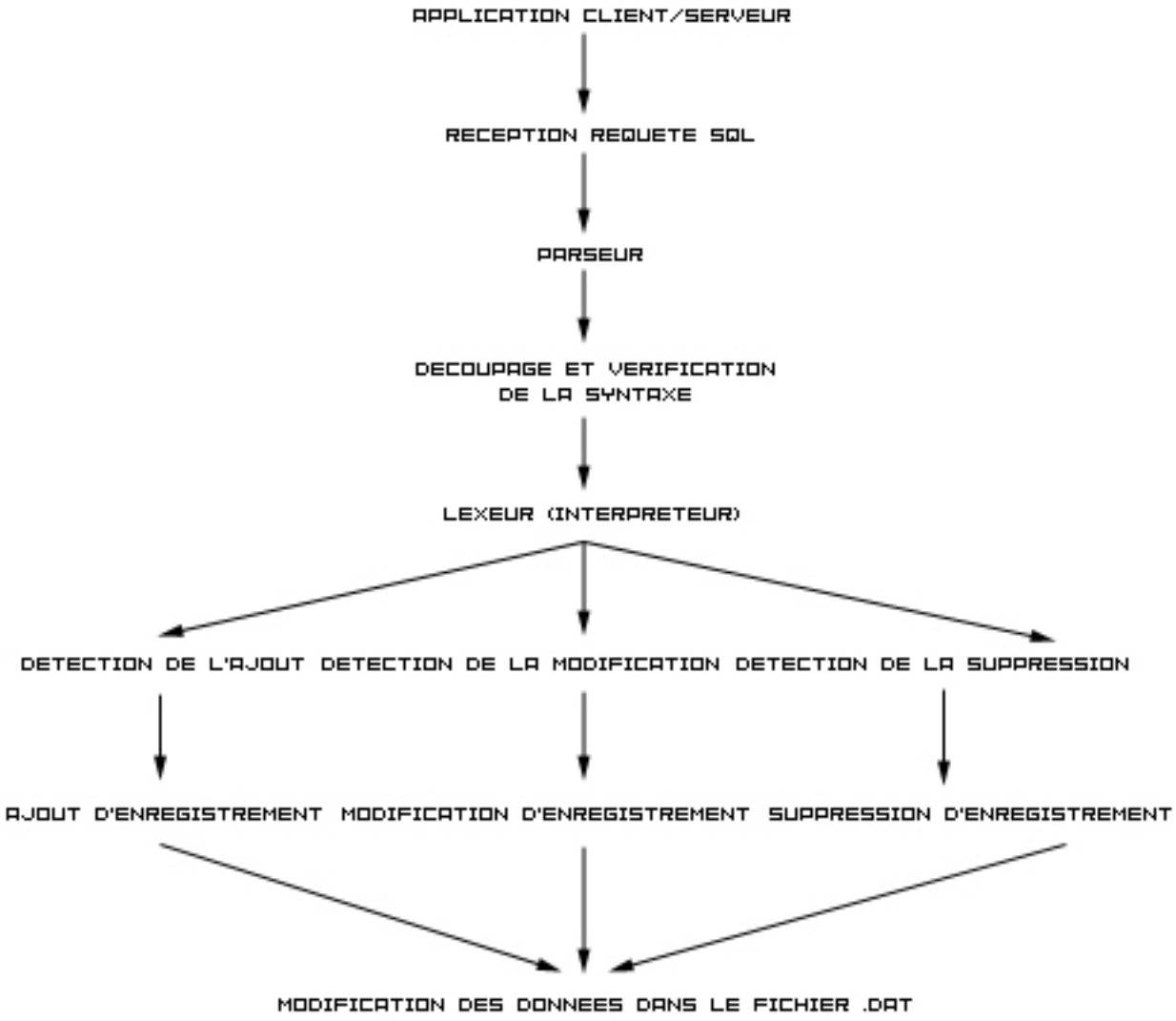
3.2.5 Application Client/Serveur

L'application Client/Serveur est chargée de faire le lien entre le SGBD et l'application qui demande les informations, le tout à l'aide de requêtes conformes à la grammaire SQL.

Dans un premier temps, le programme demandant des informations envoie une requête à l'application client/serveur qui elle, va rediriger vers le SGBD qui va effectuer les traitements.

Il est important de préciser qu'une base de données doit pouvoir traiter de nombreuses requêtes en quasi-simultanéité. Pour cela, une fois la requête reçue, l'accès a la base sera verrouillé le temps du traitement, puis déverrouillé une fois le traitement terminé et la requête suivante pourra être exécutée, tout cela afin d'éviter d'avoir des conflits au niveau des accès simultanés aux mêmes objets.

3.3 Schéma de Fonctionnement



4 Avancement Général

4.1 Avancement Général du Projet

Comme évoqué précédemment, la première étape d'un tel projet a nécessité avant tout beaucoup de recherche sur le fonctionnement d'un SGBD et plus précisément sur le parseur / lexeur et sur le système de fichiers.

Le parseur, bien qu'à ses débuts, est déjà capable de faire l'analyse syntaxique correcte de requêtes SQL puisque la définition de la grammaire a été suffisamment avancée pour cette première soutenance.

Par d'ailleurs, la conception du lexeur a dû être commencé dès cette première étape puisque, suite à nos recherches, il s'est avéré que le parseur et le lexeur nécessitaient d'être développés en parallèle.

De même, le système de fichiers, c'est-à-dire la structure du SGBD est achevé.

Cependant, en tant que première étape, la plupart de ses composants ont été dévoloppés indépendamment les uns des autres puis regroupés peu à peu.

4.2 Avancement Général du Site

Quant au Site relatif au projet, seule la maquette graphique n'aurait dû être présentée lors de cette première soutenance. Cependant dans un souci de développement plus rapide et d'information pour la suite, il a été développé un peu plus que prévu. En effet, en plus de la charte graphique, le squelette du site est en place et accessible depuis l'adresse Internet suivante :

http ://www.homerlan.com/epidatabase

4.3 Avancement Prévu pour la deuxième Soutenance

La tâche principale prévue pour la deuxième soutenance est la fin de la conception du parseur et l'avancement continu de la réalisation du lexeur. Il sera également question de recherches complémentaires pour pallier aux problèmes rencontrés et probablement une esquisse de l'application client/serveur.

Cela consistera également en l'avancement quat à la création du site avec la mise en place des scripts PHP ainsi qu'une base de données mySQL avec laquelle ils interfèreront.

5 Problèmes Rencontrés

5.1 Groupe

De ce côté-ci, le seul problème qui aurait pu nous déstabiliser vis-à-vis du projet de départ fut le désistement de notre quatrième membre.
Cependant, la réorganisation des tâches fut immédiate et paraît pour le moment faire face plus que convenablement à cette carence.

5.2 Projet

A ce niveau-ci, notre principal obstacle était en effet de ne pas connaître parfaitement le milieu des SGBD et donc leurs principes et leur fonctionnement. Cependant, nos recherches, bien qu'étant assez techniques, ne furent pas vaines et nous permirent de surmonter ces différents obstacles.

6 Tâches Individuelles

6.1 Tâches réalisées par Taha

6.1.1 Recherche de fonctionnement Sgbd

6.1.2 Introduction

Les bases de données sont actuellement au coeur du système d'information des entreprises. Les systèmes de gestion de bases de données, initialement disponibles uniquement sur des "mainframes", peuvent maintenant être installés sur tous les types d'ordinateurs y compris les ordinateurs personnels. Mais attention, souvent on désigne, par abus de langage, sous le nom "bases de données" des ensembles de données qui n'en sont pas.

Dans un premier temps, et de façon informelle, on peut considérer une Base de Données (BD) comme une grande quantité de données, centralisées ou non, servant pour les besoins d'une ou plusieurs applications, interrogeables et modifiables par un groupe d'utilisateurs travaillant en parallèle. Quant au Système de Gestion de Bases de Données (SGBD), il peut être vu comme le logiciel qui prend en charge la structuration, le stockage, la mise à jour et la maintenance des données ; c'est, en fait, l'interface entre la base de données et les utilisateurs ou leurs programmes.

6.1.3 Présentation de MyISAM

Il ya plusieurs moteurs de gestion de base de données , l'un des plus connus MyISAM, c'est le type par défaut de table MySQL.Il est basé sur ISAM et ajoute de nombreuses extensions pratiques. On ce qui concèrne le sytème de fichiers , chaque table MyISAM est stockée dans trois fichiers . Les fichiers portent le nom de la table et ont une extention qui spécifie le type de fichier .le fichier '.frm' stocke la définition de la table.L'index est stocké dans un fichier avec l'extension '.MYI', et les données sont stockées dans un fichier avec l'extention '.MYD'.De cette manière on a un grand choix de fonctionalitées, par example vérifier ou réparer une table , ou bien de la comprésser etc...

De cette facon , on a compris , que le meilleur moyen de représenter nos tables et nos champs c'est de les représenter sous forme de fichiers d'extentions différentes...

6.1.4 Mise en place du système de fichiers

Dans cette partie, le but est de donner une structure à notre base de données, on a choisi alors , de la représenter sous la forme suivante : (voir partie de Julien ASSEMAT).

6.1.5 Premières fonctions

Dans cette partie , on présentera quelques fonctions utiles , pour notre début du moteur de gestion.

6.1.6 Fonction d'ajout

Cette fonction sert à ajouter un champ quelconque, dans notre table, à l'aide d'un parcours itératif, avec un pointeur, on parcourt toute la table jusqu'à la fin , (car ca sera un ajout a la fin d'une liste chainée qui est notre table) tout en stockant le contenu des champs dans un tableau statique, qui servira à la fin comme source d'écriture dans notre fichier .epi !

6.1.7 Exemple de code

```
...
  while(Fields != NULL)
  {
    Temp.Name = Fields->Fields.Name;               /* parcours */
    Temp.BytesSize = Fields->Fields.BytesSize;
    *(Static + i*sizeof(t_Field)) = Temp;          /* stockage*/
    Fields = Fields->Next;
    i++;
  }
  Fields.Name= champ.name;
  Fields.BytesSize= champ.bytesSize;
  Temp.Name = Fields->Fields.Name;                /*ajout*/
    Temp.BytesSize = Fields->Fields.BytesSize;
    *(Static + (i+1)*sizeof(t_Field)) = Temp;
    return temp;

  f = fopen(Filename, "w");                      /*écriture*/
  fwrite(Static, sizeof(Static), 1, f);
  fclose(f);
```

6.1.8 Fonction de suppression

On a mis en place deux types de fonctions de suppression, une fonction toute simple qui permet la suppression du fichier .epi correspondant, une autre un peu plus compliquée qui permet la suppression du champ de l'endroit où il se trouve dans notre liste chainée, tout en gardant la tête de la liste... Pour cette fonction on utilisera, deux pointeurs comme vu en TD, le premier pointeur permettra de supprimer le champ sur lequel il pointe, le deuxième pointeur permmetra de récupérer le reste de la liste.

6.1.9 Exemple de code

```
temp = *fields;
temp0 = temp;
temp = temp->next;

while(temp != NULL)&&(temp != champ)
{
  temp = temp->next;
  temp0 = temp0->next;
}
 temp0->next = temp->next;
 free(temp);
 return (&(fields));
```

6.1.10 Debut réseau

Le réseau est prévu pour la prochaine soutenance , mais ca n'empêchera pas de faire quelques débuts, comme vu ci-dessous, la création d'un socket.

6.1.11 Exemple de code

```
#include <winsock2.h> #pragma comment(lib, "ws2_32.lib") void
main() { WSADATA WSAData; WSAStartup(MAKEWORD(2,0), &WSAData); }


void creer () {

SOCKET sock; SOCKADDR_IN sin; sin.sin_addr.s_addr =
inet_addr("127.0.0.1"); sin.sin_family      = AF_INET;
sin.sin_port        = htons(4148); sock =
socket(AF_INET,SOCK_STREAM,0); bind(sock, (SOCKADDR *)&sin,
sizeof(sin)); connect(sock, (SOCKADDR *)&sin, sizeof(sin));
```

6.1.12 Recherche parseur

Pour notre parseur , il sera vu en deuxième soutenance , et pour la recherche, voir partie (Thomas MOULARD).

6.2 Tâches réaslisées par Thomas

6.2.1 Mise en place du projet

Organisation du projet

Ma première tâche au sein du projet a été d'organiser notre travail à plusieurs niveaux. J'ai réparti le code du projet en répertoires et en fichiers distincts :

- epiEngine est le moteur de stockage de notre base de donnée. Il regroupe tout le code gérant la lecture et l'écriture des informations.
- sql contient l'interprêteur : il transforme les requêtes envoyées par le client en une structure directement utilisable par le moteur de stockage.

D'autres dossiers ont également été ajoutés : locale contient tous les messages d'erreurs et plus généralement les données qui diffèrent selon les langues. www contiendra le site web lorsqu'il sera réalisé. docs contient la documentation écrite par l'équipe (rapports, présentations, documentation collectée sur le net...) et enfin extra contient les programmes supplémentaires et les archives du projet.

Nous avons également convenu de respecter au maximum la norme EPITA, en particulier sur les points suivants :

- en-tête des fichiers
- nommage des variables
- découpage en un maximum de fichiers distincts

Même si ces points sont secondaires, il est important de se fixer dès le départ une ligne de conduite afin d'arriver à un code clair et lisible à la fin du projet.

Types de données

Mon second travail a consisté à définir les types de données que nous allons utiliser pour faire communiquer l'interpréteur, le moteur de stockage et l'interface client/serveur.
Désormais, chaque partie sait ce qu'il doit traiter et ce qu'il doit renvoyer, ainsi chaque membre de l'équipe peut avancer de manière quasiment indépendante.

Nous avons deux types principaux : s_sql_query qui contient la requête analysée et s_sql_result qui contient le résultat de la requête.

```
#define S_DATABASE      64
#define S_TABLE         64
#define S_COLUMN        64
#define S_INDEX         64
#define S_ALIAS         256
```

```
typedef char            t_dbname[S_DATABASE-1];
typedef char            t_tblname[S_TABLE-1];
typedef char            t_columnname[S_COLUMN-1];
typedef char            t_indexname[S_INDEX-1];
typedef char            t_aliasname[S_ALIAS-1];


typedef struct s_table  *p_table;


struct                  s_sql_query
{
   t_dbname             dbname;
   t_tblname            tblname;
   p_table              table;

   int                  if_exists;
   int                  if_not_exists;
};

struct s_sql_result
{
   int                  error_code;
};

struct                  s_column
{
   t_columnname         name;
   unsigned char        size;
   char                 flags;
};


struct                  s_table
{
   struct s_column      column;
   p_table              next;
};
```

Pour éviter d'allouer les chaînes dynamiquement, des longueurs maximums ont été définies pour les noms des base de données, tables, colonnes, etc.

On trouve ensuite s_sql_query notre type qui contient les requêtes analysées : il s'étoffera au fur et à mesure que l'interprêteur évoluera. Il gère actuellement les données suivantes :

- dbname est utilisé pour stocker le nom d'une base de donnée, par exemple lors d'une requête type "CREATE DATABASE".
- tblname est utilisé quasiment dans toutes les requêtes (CREATE TABLE, SELECT, INSERT, etc.), il indique sur quelle table les opérations s'appliquent.
- table est une liste chaînée de colonnes qui permet de décrire la structure d'une table. Il est utilisé dans CREATE TABLE. Chaque élément de la liste est une structure contenant toutes les informations à propos d'une colonne : nom, taille, et drapeaux.
- if_not_exists et if_exists sont des drapeaux permettant d'autoriser les créations et les suppressions à échouer sans émettre d'erreur ni stopper l'execution de la requête.

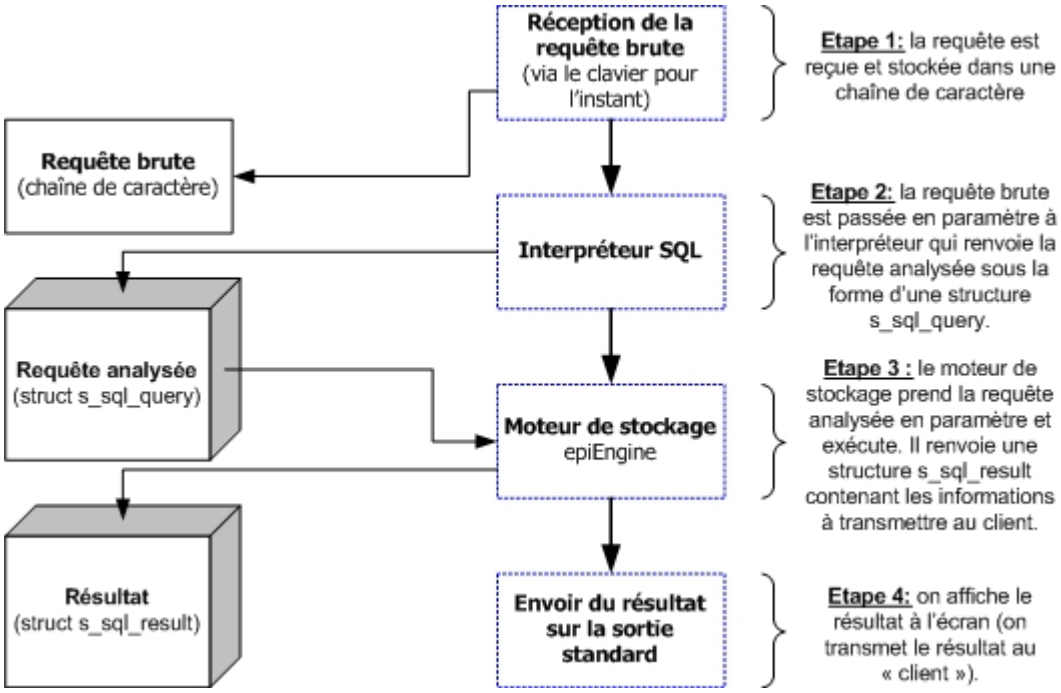
La seconde structure mise en place est s_sql_result qui gère le résultat renvoyé. Elle n'est composée pour l'instant que de l'entier error_code qui renvoit un code d'erreur : zéro s'il n'y a pas de problème ou un nombre supérieur à 0 sinon. Les erreurs sont indexées et sont liées à une chaîne de caractère décrivant le problème.

Ces deux structures ne sont instanciées qu'une seule fois dans tout le programme. Il s'agit des deux variables globales : gl_query et gl_result.

Ce système permet ainsi d'avoir un syntaxe commune à toutes les fonctions du moteur de stockage :

```
void FONCTION(s_sql_query *query, s_sql_result *result);
```

On peut schématiser le fonctionnement du programme de cette façon :


Réception de la requête brute (via le clavier pour l'instant)
Etape 1: la requête est reçue et stockée dans une chaîne de caractère
Requête brute (chaîne de caractère)
Interpréteur SQL
Etape 2: la requête brute est passée en paramètre à l'interpréteur qui renvoie la requête analysée sous la forme d'une structure s_sql_query.
Requête analysée (struct s_sql_query)
Moteur de stockage epiEngine
Etape 3 : le moteur de stockage prend la requête analysée en paramètre et exécute. Il renvoie une structure s_sql_result contenant les informations à transmettre au client.
Résultat (struct s_sql_result)
Envoir du résultat sur la sortie standard
Etape 4: on affiche le résultat à l'écran (on transmet le résultat au « client »).

6.2.2 Interprétation des requêtes SQL

Présentation : SQL, les outils flex et bison

Les requêtes envoyées à notre système de gestion de base de données utilisent le langage SQL. Même s'il peut gérer des variables, des boucles, des instructions conditionnelles, etc. Notre but est d'abord de gérer les requêtes de manipulation simples. Actuellement l'interpréteur reconnaît les syntaxes suivantes :

```
CREATE DATABASE [IF NOT EXISTS] ‘table‘ ;
DROP DATABASE   [IF EXISTS] ‘table‘ ;

CREATE TABLE    ‘table‘ ( ‘nom_de_colonne‘ type_de_colonne ,
...) ;

CHECK TABLE     ‘table‘ ;
OPTIMIZE TABLE  ‘table‘ ;

INSERT INTO ‘table‘ ( valeurs, ... ) ;
SELECT liste_champs FROM ‘table‘ ;
```

Les deux premières requêtes permettent respectivement de créer et détruire une base de donnée.

La troisième permet de créer une table.

La quatrième et la cinquième de réaliser des opérations de maintenance.

Enfin la cinquième insère des données dans une table et la dernière rapatrie toutes les données contenues dans une table.

Les parties entre crochets sont facultatives. Elles permettent de faire échouer silencieusement les deux premières requêtes.

Pour analyser les requêtes, nous avons choisi d'utiliser les deux outils flex et bison qui sont respectivement un analyseur lexical et syntaxique.
flex reconnaît via des expressions régulières les différents éléments du langage :

les mots-clés CREATE, DATABASE, TABLE...

les identifiants ma_table ou ‘ma_table‘

les autres types de donnée nombres (0, 17, 3.14), chaînes de caractères ("ma chaîne"), etc.

Chaque élément reconnu est ensuite envoyé à l'analyseur syntaxique qui reconnaît la grammaire du langage : c'est à dire toutes les règles qui indiquent si un langage est écrit de manière cohérente : "CREATE DATABASE ‘db‘ ;" a un sens mais pas "CREATE ‘db‘ DATABASE ;".

Analyseur lexical

Flex est notre analyseur lexical, il doit découper la requête de manière logique : il doit reconnaître les mots-clés du langage, les noms de table, de champs, les chaînes de caractères, etc.

Pour cela, on lui passe en paramètre un fichier comprenant un ensemble de règles. En voici un extrait :

```
%{
    #include <stdio.h>
    #include <stdlib.h>
    #include <strings.h>

    #include "../types.h"
    #include "sql_tab.h"

    extern YYSTYPE yylval;
%}

%%

    ADD                 return TOKADD;
    ALL                 return TOKALL;
    ALTER               return TOKALTER;
    ANALYZE             return TOKANALYSE;
    AND                 return TOKAND;
    AS                  return TOKAS;
    [...]

    ;                   return SEMICOLON;
    ,                   return VIRGULE;
    \(                  return PARENTHESEOPEN;
    \)                  return PARENTHESECLOSE;
    \"[^\"]\"+          yylval.string=(char*)strdup(yytext); return STRING;
```

```
    '[^']'+              yylval.string=(char*)strdup(yytext); return STRING;

    [0-9]+\.[0-9]*       yylval.floatNumber=atof(yytext); return FLOAT;
    [0-9]+               yylval.number=atoi(yytext); return INTEGER;

    \*                   return JOKER;

    [a-zA-Z0-9]{1,255}   yylval.string=(char*)strdup(yytext); return IDENT;
    `[a-zA-Z0-9]{1,255}` yylval.string=(char*)strdup(yytext);unquote(yylval.string);
                         return IDENT;

    \n                   /* ignore end of line */;
    [ \t]+               /* ignore whitespace */;
    .                    /* ignore */
%%
```

Il est divisé en 4 parties :

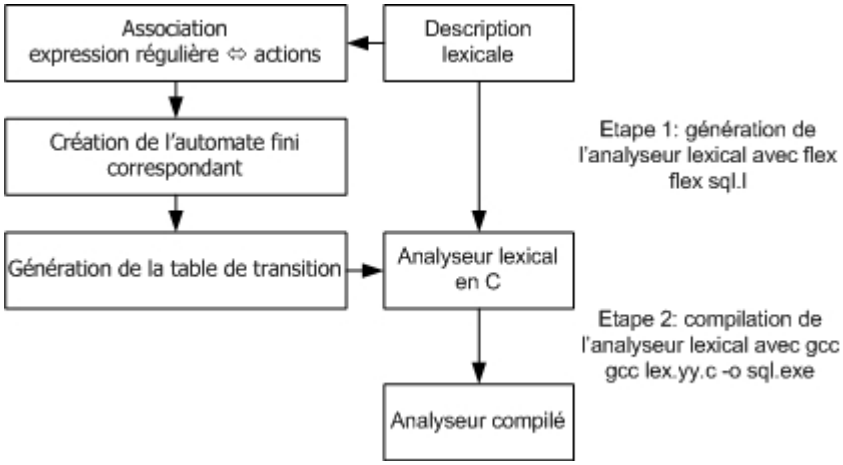
```
%{
DEFINITIONS
%}
%%
REGLES ET ACTIONS ASSOCIEES
%%
FONCTIONS
```

La première section contient du code C : les inclusions et les définitions de variables principalement.

La deuxième section est la plus intéressante, elle se présente sous la forme d'une expression régulière suivie d'un espace et du code C à exécuter lorsque l'on reconnaît ce type d'élément. On retourne la plupart du temps une constante qui permet d'identifier l'élément. Bison fournit également la variable yylval dans laquelle on peut stocker une valeur supplémentaire : lorsque l'on reconnaît un identifiant par exemple, on retourne une constante indiquant qu'il s'agit d'un identifiant et on mets sa valeur dans yylval.

Enfin la dernière section inclut des fonctions C supplémentaires.

Le fonctionnement de l'analyseur lexical peut donc se résumer sous cette forme :



Analyseur syntaxique

Bison est l'analyseur syntaxique que nous avons utilisé : à partir de la grammaire du langage, il reconnaît les requêtes qui ont un sens et exécute le code correspondant.

Voici un extrait de la grammaire utilisée dans le projet :

```
%{
[...]
extern struct s_sql_query  gl_query;
extern struct s_sql_result gl_result;
[...]
%}

%token TOKADD TOKALL TOKALTER TOKANALYSE TOKAND TOKAS TOKASC TOKASENSITIVE
[...]

%union
{
   double floatNumber;
```

```
   int number;
   char *string;
}

%token <string>      STRING
%token <floatNumber> FLOAT
%token <number>      INTEGER
%token <string>      IDENT

%start commands
%%

/* ROOT */
commands:
/* empty */
| commands command SEMICOLON
| commands command
;

/* LIST OF COMMANDS */
command:
check_table
| create_database
| create_table
| drop_database
| insert
| optimize_table
| select
;

[...]

/* === CREATE DATABASE === */
create_database:
TOKCREATE TOKDATABASE if_not_exists IDENT
{
   strcpy(gl_query.dbname,$4);
   createdatabase(&gl_query,&gl_result);
}
;
```

```
[...]

/* === SHARED STATES === */
[...]
if_not_exists
: /* empty */     { gl_query.if_not_exists = 0; }
| TOKIF TOKEXISTS { gl_query.if_not_exists = 1; }
;
%%
```

On peut remarquer que la grammaire se décompose en trois sections, tout comme la description lexicale. Elles ont également le même sens : définitions, règles et fonctions.

Cependant quelques points diffèrent : tout t'abord on trouve entre la première et la deuxième partie un ensemble de lignes qui ne sont pas des règles mais des indications pour bison.

%token permet d'indiquer la liste des tokens reconnus par flex, c'est à dire tous les types de données différents que comporte le langage à analyser.

%union est une union au sens du C : on définit ici le type de yylval utilisé par flex pour envoyer des informations à bison. Comme nous pouvons renvoyer plusieurs types d'informations, on définit ici une union, ce qui évacue le problème de typage de yylval.

%start indique l'état de départ de notre analyseur syntaxique.

La deuxième section définit la grammaire du langage sous la forme d'une énumération d'états. A chaque état est associé un motif et éventuellement des actions. Les motifs sont fortement récursifs. Prenons par exemples l'état de départ :

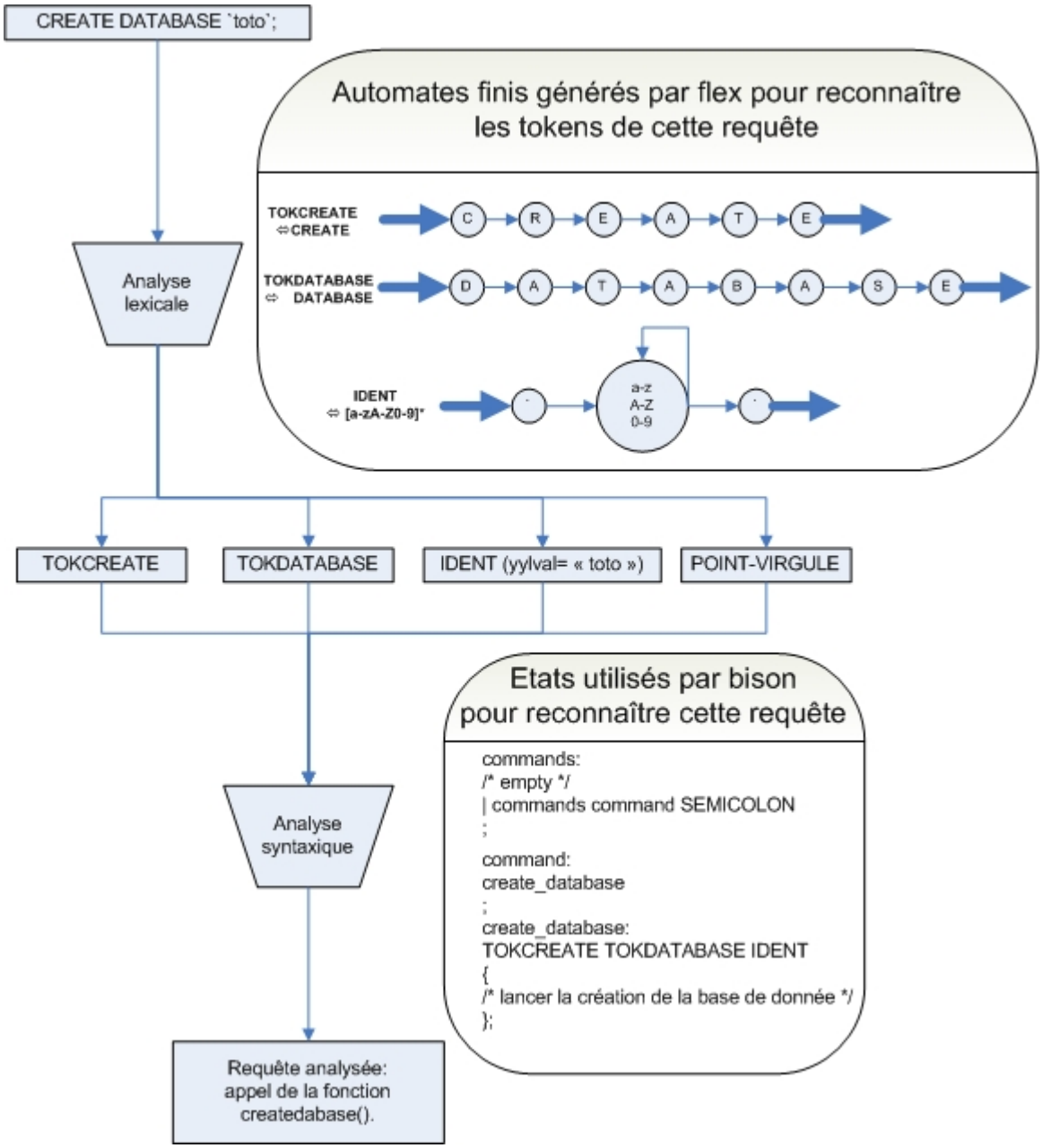
```
commands:
/* empty */
| commands command SEMICOLON
| commands command
;
```

Il indique que ce qui est analysé peut présenter trois formes :

1. rien
2. une commande et un point-virgule
3. une commande uniquement

Dans les deux derniers cas, le motif commence par "commands" soit un appel récursif à l'état actuel : il signifie que l'on peut avoir avant le motif les trois cas énumérés ci-dessus. Evidemment, on s'aperçoit ainsi que l'on peut ainsi analyser un nombre infini de commandes, séparées par un point-virgule ou pas.

De plus, à n'importe quel endroits des motifs, on peut ajouter entre accolade du code C qui s'executera lors de l'analyse du token précédent l'accolade.



6.2.3 Conclusion

Points positifs

L'un des points positifs à souligner pour cette première soutenance est le respect du planning. C'est d'autant plus réconfortant que l'on ne pouvait que très mal estimer le temps que nous allions mettre pour apprendre à manier flex et bison dans la mesure où aucun de nous ne connaissait ces outils. En d'autres termes, le cahier des charges est tout à fait réaliste et je pense que nous devrions arriver à atteindre les objectifs que nous nous sommes fixés.

Points négatifs

Cette première soutenance a également eu son lot d'imprévus : la redistribution du travail pour quatre sur les trois membres restants du groupe d'une part et la complexité du projet d'autre part. Flex et bison sont des outils difficiles à manier que je ne comprends pas encore totalement et il reste donc de nombreux problèmes à résoudre au niveau de l'analyse des requêtes. Les listes de valeurs séparées par des virgules sont mal analysées actuellement : valeur1, valeur2, valeur3 est certes reconnu, mais valeur1 valeur2, valeur3, l'est aussi ! De plus, les requêtes du type "SELECT * FROM ‘table‘ WHERE ..." semblent difficiles à analyser. Je pense même qu'il s'agira d'un problème majeur des prochaines soutenances.

Bilan de la première soutenance

Flex et bison étant compris dans les grandes lignes, un des gros problèmes du projet est résolu. Cela me permet donc au terme de cette première période d'être optimiste quant au déroulement du projet.

En effet, j'ai réussi à jetter les bases de notre analyseur lexical et syntaxique et l'organisation du projet a été facilitée par l'étude du code de mySQL. Nous avons donc de bases solides pour les prochaines soutenances et j'espère donc pouvoir tenir notre planning et remplir les objectifs que nous nous sommes fixés.

6.3 Tâches réaslisées par Julien

Pour ma part, durant cette première étape vers l'élaboration complète de notre projet, j'ai travaillé à plusieurs niveaux. Tout d'abord, il a fallu se documenter sur le fonctionnement classique d'un système de gestion de base de données (SGBD) afin que je puisse au mieux établir un système de fichiers rapide et efficace pour une application telle qu'EpiDatabase.

Dans un deuxième temps, une fois le système de fichiers mis en place, et une fois assez documenté dans le domaine des parseurs et des lexeurs, je me suis penché sur cette grosse tâche déjà bien entamée par Thomas, le tout afin de mieux en comprendre le fonctionnement et ensuite pour être capable de l'appuyer sur ce composant essentiel.

Enfin, dans un troisième et dernier temps, j'ai mis en place le site Web, support relationnel primordial entre l'équipe de développement et les utilisateurs.

6.3.1 Recherches

Comme cela a été dit précedemment, un projet tel que celui-ci demande de nombreuses recherches préalables sur autant de choses qu'il y a de composants.

En effet, en premier lieu, les recherches se sont portées sur le constituant essentiel d'un SGBD, à savoir la mise en place d'un système de fichiers, correspondant au squelette même de notre SGBD. Pour cela, et conseillé par Thomas, il a semblé que le site Web du très célèbre et performant SGBD mySQL (http ://www.mysql.com) disposait de toutes les informations dont j'avais besoin pour cette création.

Par ailleurs, ce même site propose une immense documentation très précise sur son parseur. Bien sûr il ne s'agit pas d'arriver à égaler les performances de mySQL mais de le prendre pour modèle pour EpiDatabase. Ainsi ses exemples sur la manière de concevoir le parseur et le lexeur nous ont grandement aidé et m'ont plus particulièrement appris que le développement simultané du lexeur et du parseur était primordial.

Cela nous a d'ailleurs convaincu d'utiliser les deux programmes Bison et Flex à ce niveau-là.

Pour ce qui est de l'aspect plus avancé de ces deux composants, il m'a fallu regarder quelques sources donnant des exemples à la fois simples et concrêts. Voici quelques unes de ces sources :

http ://comsci.liu.edu/ murali/lexyacc/lex1.txt
http ://comsci.liu.edu/ murali/lexyacc/lex2.txt
http ://comsci.liu.edu/ murali/lexyacc/yac1.txt

D'autres documents tels que des PDF rédigés par des développeurs ayant beaucoup d'expérience m'ont permis d'approfondir les principes et de passer à la pratique, l'un des plus complets que j'ai pu trouver étant *Compiler Construction using Flex and Bison* de *Anthony A. Aaby.*

6.3.2 Système de Fichiers

Comme je le disais donc, cette première étape du projet a été pour moi l'occasion de concevoir le système de fichiers. Il représente le squelette de système de gestion de base de données puisqu'il contient à lui seul absolument toutes les données relatives aux bases de données, aux tables et aux informations stockées. Ainsi, il me fallait une structure simple, rapide à exécuter et modulable très facilement.

Notre système de fichiers a été pensé et repensé pour, au final, arriver à quelque chose de performant. Une base de données correspondra à un répertoire. Les tables d'une certaine base de données seront stockées dans le répertoire de cette base de données et seront découpées en trois fichiers que l'on distinguera par leur extension :

- Un fichier *NomTable.epi* qui contient la structure de la table et rien d'autre. La structure est le nom des champs et la taille maximale en octets de chacun de ses champs.
- Un fichier *NomTable.key* qui contient toutes les clés relatives aux champs de la table.
- Un fichier *NomTable.dat* qui contient toutes les données, c'est-à-dire tous les enregistrements.

Du côté de la programmation, on utilise les types suivants :

```
struct                  s_column {
   t_columnname         name;
   unsigned char        size;
   char                 flags;
   int                  last;
};


struct                  s_table {
   struct s_column      column;
   p_table              next;
};
```

Note : la propriété permet de spécifier si oui ou non c'est le dernier champ.

Exemple : le fichier */base1/users.epi* donne accès à la structure de la table *users* située dans la base de données *base1*.

Lorsque la requête de création d'une table est détectée, tous les champs, spécifiés dans la requête, sont mis sous forme de liste chaînée avec la structure *s_table* et *p_table* (pointeur sur *s_table*).

Une fois cette structure établie, elle est convertie en tableau statique (sachant alors le nombre de champs et donc la taille du tableau). Ainsi, le tableau statique est enregistré dans le fichier *.epi* chargé de contenir la structure de la table.

La fonction d'enregistrement et de conversion est la suivante :

```
void createtable(struct s_sql_query *query, struct s_sql_result
*result) {
   FILE *f;
   char filename[S_TABLE+strlen(EXTENSION_TABLE)-1];
   p_table tbl;
   struct s_column Tab[query->nbcols], Temp;
   int i, j;

   *filename = 0;

   strcpy(filename, PATH_DATA);
   strcat(filename, query->dbname);
   strcat(filename, "\\");

    if(!opendir(filename))
    {
        result->error_code=MSG_DATABASE_NOT_EXISTS;
        return;
    }

   strcat(filename, query->tblname);
   strcat(filename, EXTENSION_TABLE);

   if(!(f = fopen(filename, "r")))
   {
        f = fopen(filename, "w");
```

```
        tbl = query->table;

        /* Conversion en tableau statique */
        i = 0;
        while((i < query->nbcols)&&(tbl != NULL))
        {
           strcpy(Tab[i].name, tbl->column.name);
           Tab[i].size = tbl->column.size;
           Tab[i].flags = tbl->column.flags;
           Tab[i].last = tbl->column.last;
           tbl = tbl->next;
           i++;
        }
        fwrite(Tab, sizeof(Tab), 1, f);
        fclose(f);

        /* Liberation du tableau */
        j = i - 1;
        while(j >= 0)
        {
                free(Tab + j*sizeof(struct s_column));
                j--;
        }

        result->error_code = MSG_OK;
   }
   else
   {
        fclose(f);

        if(query->if_not_exists)
            result->error_code=MSG_OK;
        else
            result->error_code=MSG_TABLE_EXISTS;
   }
}
```

Tout cela afin de pouvoir récupérer facilement la structure de la table. En effet, le fichier sera lu et le contenu sera directement stocké dans un tableau statique qui subira alors le traitement inverse, à savoir sa transformation en dynamique sur laquelle auront lieu les

transformations. Voici cette fonction :

```
void seecolumns(struct s_sql_query *query, struct s_sql_result
*result) {
    FILE *f;
    p_table tbl;
    char filename[S_TABLE+strlen(EXTENSION_TABLE)-1];
    struct s_column Tab[MAX_NUMBER_FIELDS];
    int i;

    *filename = 0;
    strcpy(filename, PATH_DATA);
    strcat(filename, query->dbname);
    strcat(filename, "\\");
    strcat(filename, query->tblname);
    strcat(filename, EXTENSION_TABLE);
    if(f = fopen(filename, "r"))
    {
        fread(&Tab, 1, sizeof(Tab), f);
        fclose(f);
        i = 0;
        while(Tab[i].last == 0)
        {
            tbl = query->table;
            while(tbl != NULL)
                tbl = tbl->next;
             addField(&tbl, &(query->table), Tab[i]);
             i++;
        }
        result->error_code = MSG_OK;
        addField(&tbl, &(query->table), Tab[i]);
    }
    else
        result->error_code = MSG_TABLE_NOT_EXISTS;
}
```

6.3.3 Parseur & Lexeur

Dans la deuxième partie des tâches qui m'étaient attribuées, je devais me charger de l'avancement du composant parseur/lexeur avec l'aide de Thomas. Comme évoqué précédemment, avec la documentation et les exemples trouvés, la nécessite de développer le

parseur et le lexeur en parallèle ne s'est pas faite attendre.

Pour le moment, je me suis consacré à la détection de deux requêtes ou commandes, la requête de création d'une table et celle retournant la structure d'une table c'est-à-dire le nom de ses champs.

Bien que ces deux requêtes ne soient pas exactement du langage SQL, elle sert ici d'exemple pour vérifier le bon fonctionnement du parseur et du lexeur.

La requête de création de table est de la forme suivante :

```
CREATE TABLE NomTable champ1;champ2;champ3 IN DATABASE Base1
```

Du côté du code interprêté par Bison, cela correspond à ceci :

```
create_table: TOKCREATE TOKTABLE IDENT fields TOKIN TOKDATABASE
IDENT {
    if(gl_query.nbcols > 0)
    {
        strcpy(gl_query.tblname, $3);
        strcpy(gl_query.dbname, $7);
        createtable(&gl_query,&gl_result);
    }
    else
        printf("ERROR : Aucun champ défini !\n");
};
```

De la même manière, on définit la commande reconnue pour retourner la structure d'une table. La reqûete est alors de la forme suivante :

```
TABLE NomTable IN DATABASE Base1
```

Cela est obtenu avec le code suivant :

```
structure_table: TOKTABLE IDENT TOKIN TOKDATABASE IDENT {
    strcpy(gl_query.tblname, $2);
    strcpy(gl_query.dbname, $5);
    seecolumns(&gl_query, &gl_result);
    /* Affichage des champs */
    ParserTemp = gl_query.table;
    printf("STRUCTURE - %s -\n", gl_query.tblname);
    while(ParserTemp != NULL)
```

```
    {
        printf(" %s ", ParserTemp->column.name);
        ParserTemp = ParserTemp->next;
    }
    printf("\n");
};
```

Voilà donc comment fonctionne grossièrement le système de règles utilisé pour définir la grammaire que l'on désire appliquer à nos requêtes SQL.

6.3.4 Site Web

La troisième de mes tâches fut le début de la conception du site Web relatif à notre projet. Seule la maquette graphique aurait dû être montrée pour cette première soutenance, mais je suis allé un peu plus loin que prévu dans son avancement et ai mis la maquette graphique en HTML. La maquette a été réalisée à l'aide du logiciel *Photoshop* d'Abode.

Le site, pour le moment, est prévu pour accueillir les sections suivantes :

- *Accueil* : page principale sur laquelle figurent toutes les nouveautés majeures concernant l'état d'avancement du projet.
- *Projet* : documentation relative au projet et détaillant avec précision les objectifs et les points essentiels de ce dernier.
- *Avancement* : détail complet graphique de l'état d'avancement des différentes tâches du projet.
- *Téléchargements* : liste de tous les fichiers en rapport avec le projet : cahier des charges, rapports...
- *Contact* : formulaire pour entrer en contact avec l'équipe de développement.

Le site est accessible sur Internet depuis l'adresse suivante : *http ://www.homerlan.com/epidatabase*. Comme on peut le voir, la navigation se fait par un système d'onglets implémentés directement en PHP :

```
<?
    // Liste des onglets
    class Onglet
    {
```

```
        var $Libelle;
        var $Page;
    }

    $Tabs = Array();

    $Tabs[0] = new Onglet();
    $Tabs[0]->Libelle = "Accueil";
    $Tabs[0]->Page = "home.php";

    $Tabs[1] = new Onglet();
    $Tabs[1]->Libelle = "Projet";
    $Tabs[1]->Page = "projet.php";

    $Tabs[2] = new Onglet();
    $Tabs[2]->Libelle = "Avancement";
    $Tabs[2]->Page = "avancement.php";

    $Tabs[3] = new Onglet();
    $Tabs[3]->Libelle = "Téléchargements";
    $Tabs[3]->Page = "telechargements.php";

    $Tabs[4] = new Onglet();
    $Tabs[4]->Libelle = "Contact";
    $Tabs[4]->Page = "contact.php";
?>
```

L'id de l'onglet est passé en paramètre dans l'adresse, ce qui permet, toujours sur la page *index.php* d'insérer le code de la page correspondant à l'id, grâce à la fonction PHP emphinclude().

Comme je viens de le dire, le site sera basé sur du code PHP couplé à une base de données mySQL, par exemple, pour gérer les fichiers, les bulletins d'actualité de la page d'accueil ou autre.

6.3.5 Conclusion

D'excellentes impressions en ce qui me concerne pour cette première étape de la conception d'EpiDatabase, que ce soit au niveau de l'avancement qu'au niveau des thèmes abordés : le choix d'un projet sur un système de gestion de base de données s'avère ici très intéressant. En effet, l'apprentissage, la manipulation et la conception d'outils comme le parseur ou le lexeur sont pour moi quelque chose de totalement nouveau en terme de fonctionnement

et de principe algorithmique. Par ailleurs, ce projet représente réellement une excellente application des notions du langage C vues en cours telles que les directives de compilations, les *makefile* ou encore les pointeurs.

7 Conclusion

Pour clôturer cette première étape dans le développement du système de gestion de base de données EpiDatabase, nous pouvons dire que le travail fourni pour arriver à traiter les tâches qui nous étaient attribuées n'était pas des moindres. Les recherches menées en amont nous on permis d'avancer rapidement tout en manipulant des principes nouveaux comme ceux d'un parseur, d'un lexeur ou plus généralement d'un SGBD quelconque. Dès ce premier pallier, ce projet s'avère complètement intéressant tant au niveau de l'algorithmique qu'au niveau des nouveautés abordées par chacun de nous.

Enfin, bien que le groupe de projet ait été diminué dès le départ, nous avons réussi à mener à bien l'ensemble des tâches prévues dans le cahier des charges pour cette première soutenance et nous pouvons même dire que nous avons pris un peu d'avance que nous tâcherons de maintenir par la suite.