1. Mise en place du projet

1.1. **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.

1.2. **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.

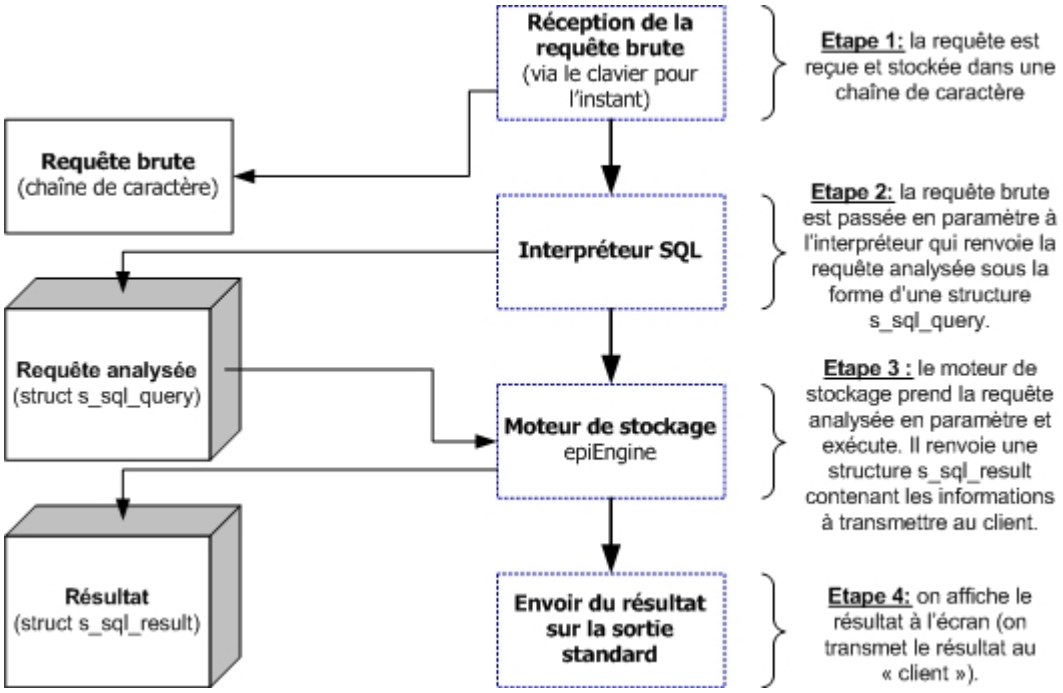
La seconde structure mise en place est s_sql_result qui gère le résultat renvoyé. Elle n'est composée pour l'instant que de l'entier error_code qui renvoit un code d'erreur: zéro s'il n'y a pas de problème ou un nombre supérieur à 0 sinon. Les erreurs sont indexées et sont liées à une chaîne de caractère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



2. Interprétation des requêtes SQL

2.1. **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 ;".

2.2. **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

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

Il est divisé en 4 parties:

```
%{
DEFINITIONS
```

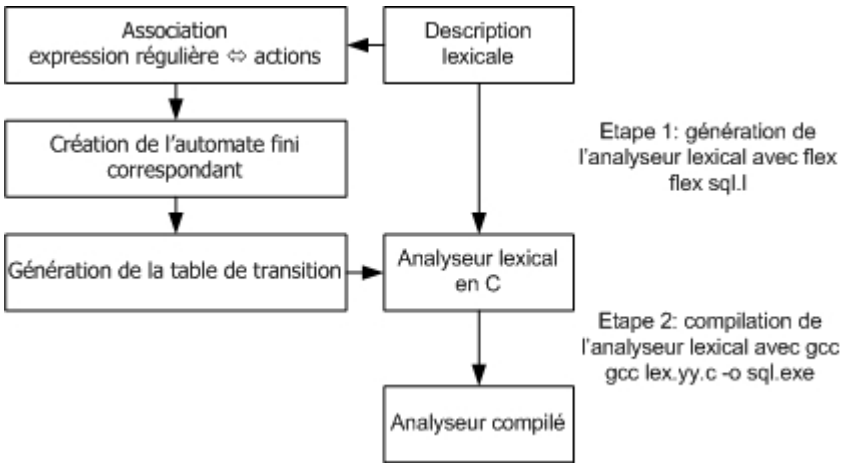
```
%}
%%
REGLES ET ACTIONS ASSOCIEES
%%
FONCTIONS
```

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:



2.3. **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:

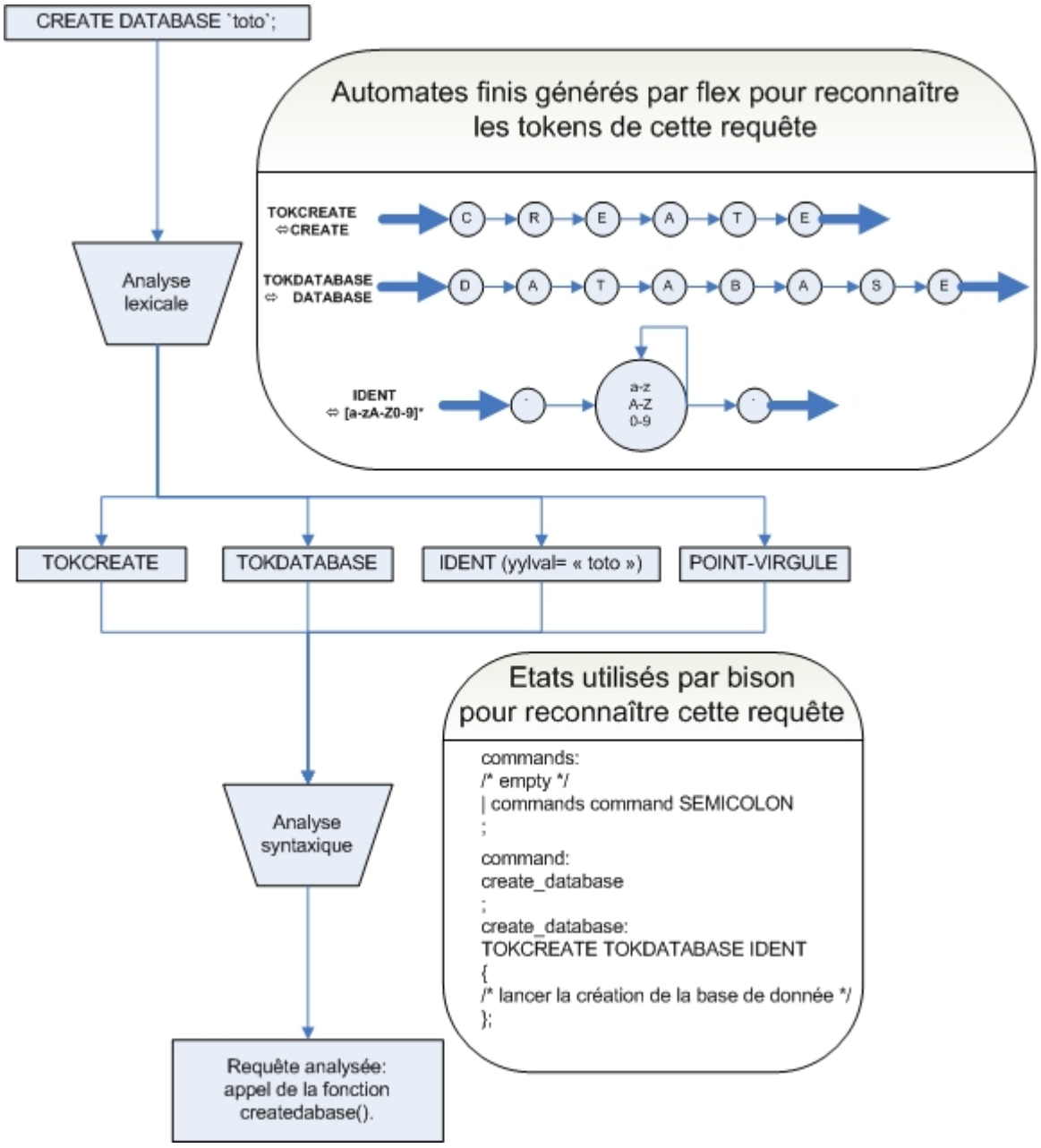
```
commands:
/* empty */
| commands command SEMICOLON
| commands command
;
```

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.


CREATE DATABASE `toto`;
Automates finis générés par flex pour reconnaître les tokens de cette requête
TOKCREATE
⇔CREATE
C
R
E
A
T
E
TOKDATABASE
⇔ DATABASE
D
A
T
A
B
A
S
E
IDENT
⇔ [a-zA-Z0-9]*
a-z
A-Z
0-9
Analyse lexicale
TOKCREATE
TOKDATABASE
IDENT (yylval= « toto »)
POINT-VIRGULE
Etats utilisés par bison pour reconnaître cette requête
commands:
/* empty */
| commands command SEMICOLON
;
command:
create_database
;
create_database:
TOKCREATE TOKDATABASE IDENT
{
/* lancer la création de la base de donnée */
};
Analyse syntaxique
Requête analysée: appel de la fonction createdabase().

3. Conclusion

3.1. **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.

3.2. **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.

3.3. **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.