Ecole Nationale Supérieure d'Informatique et d'Analyse des Systèmes - RABAT

Rapport de Projet de Compilation : QuickSQL

Réalisé par :

Hamza Benyazid
Chaimaa Najim
Meriem Ait ouaziz
Mimoun ghordou
Somaya fourtassi

Encadré par :

Pr. Rachid oulad haj thami
Pr. Youness tabii

Année académique 2021/2022

Remerciements :

Avant d'entamer tout développement de cette expérience fructueuse, nous aimerons tout d'abord exprimer notre gratitude à un nombre de personnes qui ont contribué à la réalisation et au bon déroulement de notre projet.

Toute notre reconnaissance s'adressent à nos encadrants Monsieur. TABII Youness et Monsieur. Rachid OULAD HAJ THAMI de nous avoir garanti les meilleures conditions et facilités pour mener à bien ce type de projets. Nous les remerçions également de leurs disponibilités, de leurs confiance, et du partage de leurs vastes connaissances tout au long de la réalisation de ce projet.

Table des matières

Introduction générale

Les langages de programmation ont évolué en parallèle des techniques de développement. Certains sont complémentaires, mais tous répondent à des objectifs différents. Cependant, tous langage est construit à partir d'une grammaire formelle, qui inclut des règles syntaxiques, auxquels on associe des règles sémantiques. Ainsi, l'implémentation d'un nouveau langage de programmation nécessite une étude et une analyse approfondie qui fait l'objet de ce projet.

Afin d'appliquer les méthodologies et les notions enseignées durant le cours compilation, nous sommes invités à réaliser un projet qui va nous permettre d'appliquer nos connaissances théoriques sur le champ pratique. Ce projet de compilation a pour objet l'élaboration d'un langage de programmation ainsi qu'un compilateur pour l'analyse lexicale et syntaxique. Pour ce faire, nous nous sommes basés, d'une part, sur la structure du langage Quick SQL qui est une méthode rapide de génération du code SQL requis pour créer un modèle de données relationnel à partir d'un document de texte avec retraits, et d'autre part sur la logique des langages de programmation enseignés durant les deux années écoulées afin de proposer un sous langage de Quick SQL.
Le présent travail est destiné à la présentation du langage réalisé et sera structuré comme suit :

- Le premier chapitre est consacré à la présentation du projet, la problématique et nos objectifs.
- Le deuxième chapitre est dédié à la présentaion des éléments principaux de notre language, à savoir la syntaxe et la grammaire.
- Le troisième chapitre se focalise sur l'analyse lexicale.
- Le quatrième chapitre aborde l'analyse syntaxique.
- Le cinquième chapitre entamme l'analyse sémantique.
- Le sixième chapitre est dédié à la génération du code.
- Finalement, le septième chapitre est consacré aux résultats et des exemples d'éxécution.

Chapitre 1

Présentation du projet

1.1 Contexte

Ce projet s'inscrit dans le module "Compilation" de la deuxième année du cycle ingénieur à l'ENSIAS. Nous sommes appelés à réaliser un compilateur en langage C d'un langage de notre propre choix.

1.2 Problématique

La création d'une base de donnée relationnelle avec le langage de création de données est souvent une tâche fastidieuse dû à la syntaxe "lourde" et quelquefois compliquée de SQL, en plus des différences syntaxiques existantes entre outils de base de donnée. Il s'agit d'un processus long et entâché d'erreurs.

1.3 Objectifs

Pour répondre à la problématique évoquée ci-dessus, nous proposons le langage "QuickSQL" permettant la génération de scripts SQL en utilisant une syntaxe simplifiée et intuitive.

1.4 Présentation du langage QuickSQL

L'objectif du langage QuickSQL est la simplification et l'accélération du processus de création de tables et de vues. A partir d'une syntaxe assez simple et intuitive, il est possible de générer des scripts SQL respectant les spécification du langage.

Nous présentons ci dessus un exemple d'un fichier écrit en QuickSQL :

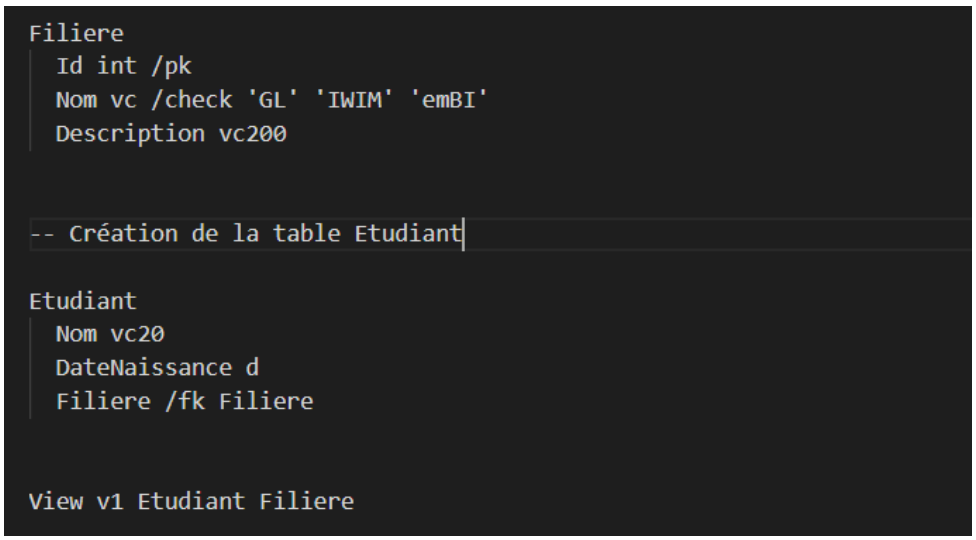

FIGURE 1.1 – Exemple de QuickSQL

Le fichier ci-dessus décrit la création de deux tables : Etudiant et Filiere, et d'une vue "v1". Les colonnes sont simplement indiquées par une indentation uniforme. L'ajout des contraintes est aussi simplifié : "/fk" désigne la contrainte foreign key par exemple. Les types de données sont également simplifiés : "d" désigne "Date".

Chapitre 2

Présentation de la grammaire

2.1 Syntaxe de QuickSQL

Le langage de programmation “QuickSQL” nous permet principalement de générer le code SQL nécessaire à la création des tables et des vues.

Nous allons par la suite présenter la syntaxe de notre langage suivant deux parties : création des tables et création des vues.

2.1.1 Création des tables

```
NomDeLaTable    [directivesDeLaTable1, directivesDeLaTable2...]


        // les directives des tables servent a definir des prefixes aux noms des colones ou
             faire une requete sql simple de select apres la creation de la table


        nomColonne1 [type] [contrainte1, contrainte2 ...]


        nomColonne2 [type] [contrainte1, contrainte2 ...]


        ...


        // declaration des colones
```

2.1.2 Création des vues

```
view nomDeLaVue nomTable1 nomTable2 ...
```

2.2 Unités lexicales

Dans cette partie, nous allons présenter l'ensemble des unités lexicales nécessaires pour l'implémentation de notre grammaire.

2.2.1 Type de données

Token Type	**Désignation**
num	TOKEN_NUM
int	TOKEN_INT
d	TOKEN_D
ts	TOKEN_TS
tstz	TOKEN_TSTZ
vc	TOKEN_VC
vcNNN	TOKEN_VCNNN
clob	TOKEN_CLOB
blob	TOKEN_BLOB
json	TOKEN_JSON

2.2.2 Directives de colonne (les contraintes)

Token Type	**Désignation**
/pk	TOKEN_PK
/fk	TOKEN_FK
/check	TOKEN_CHECK
/nn	TOKEN_NN
/between	TOKEN_BETWEEN
/index	TOKEN_INDEX
/default	TOKEN_DEFAULT
/unique	TOKEN_UNIQUE

2.2.3 Directives de table

Token Type	**Désignation**
/colprefix	TOKEN_TD_COLPREFIX
/select	TOKEN_TD_SELECT

2.2.4 Autres unités lexicales

Token Type	**Désignation**
ID	TOKEN_ID
view	TOKEN_VIEW
number	TOKEN_NUMBER
string	TOKEN_STRING
valeur quelconque	TOKEN_VALUE
espace	TOKEN_WHITESPACE
retour à la ligne	TOKEN_NL

2.3 Grammaire

On a bien fait toutes les vérifications nécessaires (récursivité à gauche et on factorisa tant qu'il est nécessaire) et on a conclue que notre langage est bien un langage LL1.

```
program : (view | table | new_line | comment | eps) { view | table | new_line | comment |
    eps }

table : ID table_directives {table_directives} new_line column { column }

view : "view" ID ID {ID} new_line

table_directives : colprefix_directive | select_directive | eps

colprefix_directive : /colprefix (ID|eps)

select_directive : /select

column : white_space ID type constraint new_line

type :  "num"
        | "int"
        | "d"
        | "ts"
        | "tstz"
        | "vc"
        | "vc" entier
        | "clob"
        | "blob"
        | "json"
        | eps

constraint : pk_constraint
            | fk_constraint
            | check_constraint
            | not_null_constraint
            | between_constraint
            | index_constraint
            | default_constraint
            | unique_constraint
            | eps

unique_constrant: /unique
```

```
pk_constraint : /pk

fk_constraint : /fk ID

check_constraint : /check value{,value}

not_null_constraint : /nn

between_constraint : /between (number number | string string)

index_constraint : /index

default_constraint : /default value

comment : -- text new_line| [ text ]
```

Chapitre 3

Analyse lexicale

3.1 Définition

L'analyse lexicale se trouve tout au début de la chaîne de compilation. La mission de l'analyse lexicale est de transformer une suite de caractères en une suite de mots dit aussi lexèmes (tokens) : l'analyseur lexical lit les caractères d'entrée jusqu'à ce qu'il puisse identifier la prochaine unité lexicale, qu'il transmet à l'analyseur syntaxique selon le schèma ci-dessous.

L'analyseur lexical detecte également les erreurs lexicales.

FIGURE 3.1 – Schèma descriptif de l'analyseur lexical

3.2 Les fonctions de l'analyseur

3.2.1 Description globale

L'analyseur lexical réalise principalemnt 3 opérations :

— Lecture du caractère suivant : La lecture se fait à partir d'un fichier à l'aide de la fonction **get_next_car()**.

— Ignorer les espaces : comformément à la syntaxe de notre langage, nous n'ignorons pas les types d'espaces suivant :

-Les espaces ' ' en début de ligne.

-Les retours à la ligne.

Ceci est réalisé à l'aide de la fonction **skip_whitespace()**.

— Reconnaitre le token lu : ceci est réalisé à l'aide de la fonction **lexer_get_next_token()**, qui fait appel aux fonctions de reconnaissance de tokens particulier(**lexer_get_number()**, **lexer_get_id()** ...).

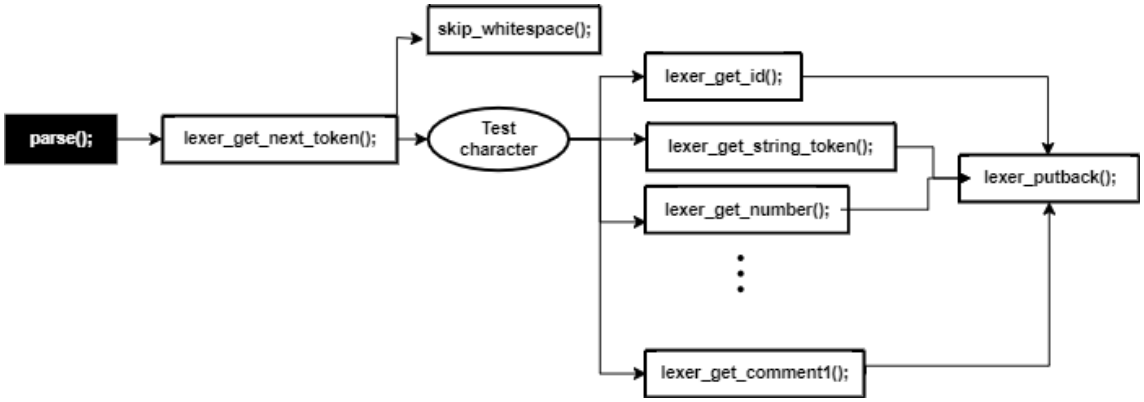

FIGURE 3.2 – Hiérarchie des fonctions de l'analyseur lexical

A noter que la fonction en haut de la hiérarchie est une fonction de l'analyseur syntaxique "parse()", car celui-ci "demande" répetitivement des tokens au lexer jusqu'a épuisation(fin de fichier) ou détection d'erreur, comme décrit sue le schèma de la figure 3.1.

3.2.2 Exemples de fonctions

— La fonction **lexer_get_next_token()** :

Elle fait appel à **skip_whitespace()** qui ignore les espaces et s'arrête sur un caractère à analyser.

Selon la valeur de ce caractère **lexer_get_next_token()** effectue le traitement adéquat : reconnaissance du caractère(cas d'un token composé d'un seul caractère), ou appel à une fonction spécifique reconaissant le token(**lexer_get_number()**, **lexer_get_id()**, **lexer_get_string()**....)

— La fonction **lexer_get_string_token()** : Les chaînes de cractères sont à l'interieur de 2 cractères " ' " : 'chaine de caractère'.

Cette fonction consomme les caractère à l'aide de **get_next_car()** jusqu'à trouver le caractère de fin de string " ' " ou fin de fichier EOF (une erreur est levée par la suite dans ce cas).

3.3 Le traitement des erreurs

Dans cette phase de compilation, les erreurs lexicals possibles commises par l'utilisateur sont détectées et signalées à l'utilisateur sous forme de messages d'erreur. Ce message contient la localisation de l'erreur et son type. Une fois une erreur est détectée, elle est signalée et le processus se termine par suite.

Les erreurs lexicales typiques sont :

— Mot non valide. Exemple : /foreignkey

— String non valide. Exemple : 'ceci est un string. (tous string doit être englobé entre ' ').

— Commentaire non valide. Exemple : [ceci est un commentaire .(Le commentaire est ouvert mais il n'est pas fermé .)

Chapitre 4

Analyse syntaxique

4.1 Définition

L'analyseur syntaxique est utilisé pour décomposer les données en tokens les plus petits possibles(terminaux) provenant de l'analyseur lexical. il reçoit une séquence de tokens et produit un arbre d'analyse.

4.2 Analyseur LL1

Le premier L signifie que le balayage de l'entrée se fera de gauche à droite (Left to Right) et le deuxième L montre que dans cette technique Left most Derivation Tree.

4.3 Principe de fonctionnement

Notre analyseur syntaxique prend en input un token(symbole terminal) généré par l'analyseur lexical par la fonction **lexer_get_next_token**. Ce token contient le code lexical et la valeur correspondante. On utilise le code pour vérifier si le token est bien placé selon notre grammaire.
Pour chaque non-terminal, on a créé une fonction sans paramètres et sans valeurs de retour, dans cette façon, on peut tester le positionnement des non-terminaux selon notre grammaire

4.4 Les fonctions de l'analyseur syntaxique

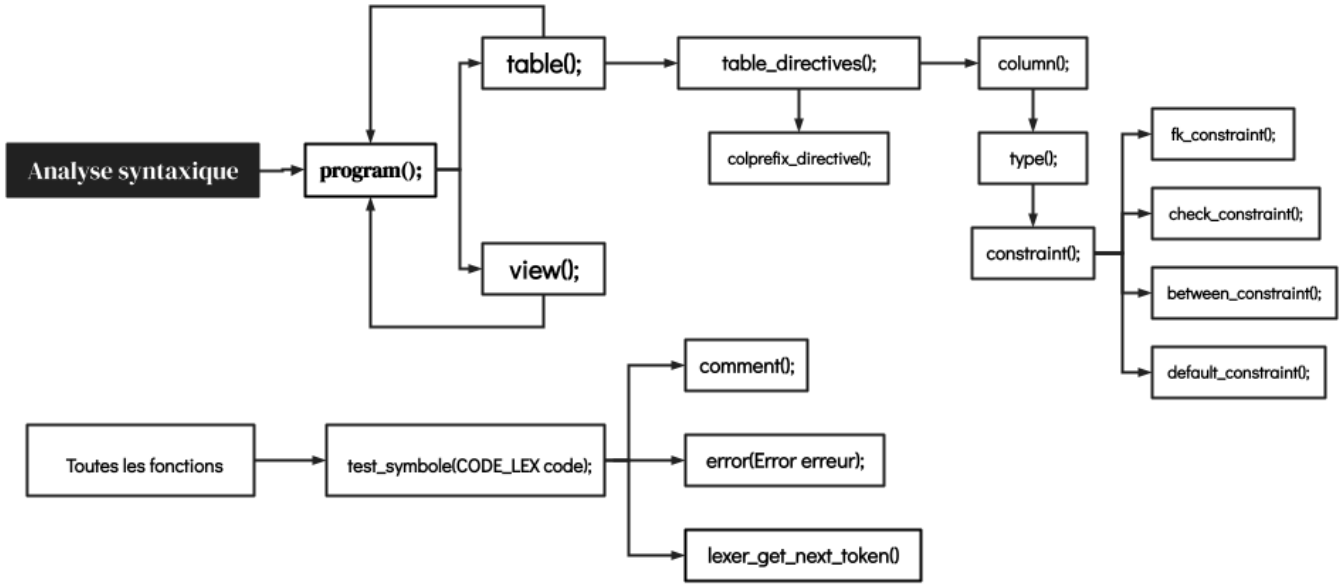

FIGURE 4.1 – Fonctions du parser

4.5 La détection des erreurs

la fonction test_symbole sert à vérifier si un token est bien placé, sinon le programme sera arrêté avec un message d'erreur qui contient le code du token erroné.

Pour traiter les erreurs dans le compilateur, on utilise cette fonction pour vérifier si un symbole est bien positionné selon l'algorithme LL1. S'il s'agit d'un non-terminal A, on effectue un test sur les éléments de first(A), puis on passe à next(A) dans le cas ou un epsilon peut être dérivé de A ...

```
ghmaimon@ghmaimon-GS72-6QD:~/Documents/Projects/QuickSQLCompiler/QuickSQL$
./main in.sql output.sql
< TOKEN_ID , Teacher >
< TOKEN_NL ,
 >
< TOKEN_WHITESPACE , 4 >
< TOKEN_ID , nom >
< TOKEN_INT , int >
........
< TOKEN_NL ,
 >
< TOKEN_WHITESPACE , 4 >
< TOKEN_ID , prenom >
< TOKEN_VC , vc >
........
< TOKEN_ID , age >

Syntax error in line 3: expected TOKEN_NL found TOKEN_ID
```

Chapitre 5

Analyse sémantique

5.1 Définition

L'analyse sémantique est une technique proche de l'analyse lexicale, mais au lieu de se faire au niveau des mots, l'analyse se fait sur le sens des phrases pour déterminer le sens des écrits, et c'est ce qui en fait toute la différence.

5.2 Règles sémantique

— Règle 1 : Les noms des tables doivent être uniques.
— Règle 2 : Les noms de colonnes dans une table doivent être uniques.
— Règle 3 : Toute directive de table ne doit pas être utilisée plus d'une fois dans une table.
— Règle 4 : Toute directive de colonne ne doit pas être utilisée plus d'une fois dans une colonne.
— Règle 5 : La longueur des noms de tables, de colonnes et de vues ne doit pas dépasser 128 caractères.
— Règle 6 : Les tables référencées dans les vues doivent être déclarées dans le même fichier.

5.3 Principe de fonctionnement

L'analyseur sémantique utilise la liste des tables et la liste des vues, que l'analyseur syntaxique prépare lors de son analyse, pour vérifier le respect des règles sémantiques. Voici ci-dessous les structures utilisées pour mémoriser toutes les informations.

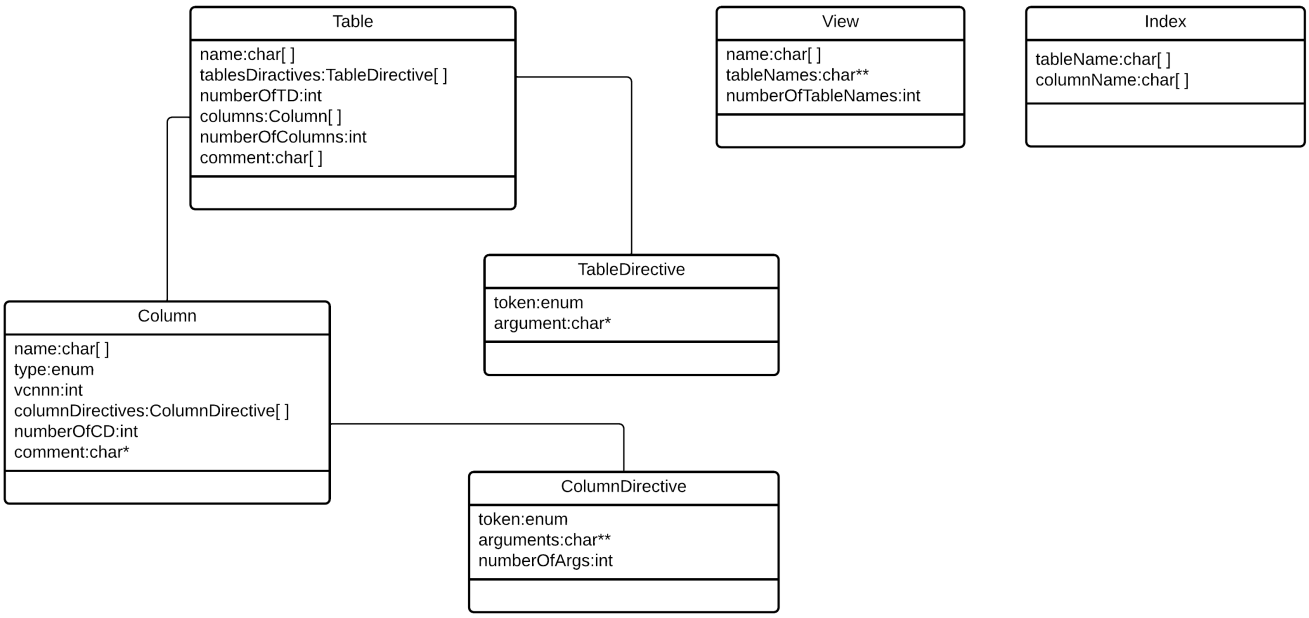

FIGURE 5.1 – les structures utilisées

L'appel de la fonction *sem*() vérifie le respect des règles sémantique d'une manière séquentiel, sauf la règle 5, cette règle est vérifiée quand les noms sont lus (au niveau de l'analyseur syntaxique).

```
void sem(){
    //Rule 1
    checkTableNamesDeclarations();
    //Rule 2
    checkColumnNamesDeclarations();
    //Rule 3
    checkTableDirectives();
    //Rule 4
    checkColumnDirectives();
    //Rule 6
    checkViewsTables();
}
```

FIGURE 5.2 – sem()

Chapitre 6

Génération du code

6.1 Définition

La génération de code est le processus par lequel le générateur de code d'un compilateur convertit une représentation intermédiaire du code source en un autre code.

6.2 Principe de fonctionnement

Le programme crée un fichier sql avec le nom spécifié par l'utilisateur et ensuite le programme crée les tables, les vues, les indexes, les commentaires et les sélections en suivant les listes des structures (voir la figure 5.1) préparé par l'analyseur syntaxique. Voici l'algorithme de fonction qui génère le code SQL :

Algorithm 1 Générer le code SQL

Function generer_code_SQL{$file\_name$}
    $file \leftarrow creer\_le\_fichier(file\_name)$
    $ouvrir\_le\_fichier(file)$
    $generer\_les\_tables(file)$
    $generer\_les\_commentaires(file)$
    $generer\_les\_indexes(file)$
    $generer\_les\_vues(file)$
    $generer\_les\_selections(file)$
    $fermer\_le\_fichier(file)$
EndFunction

Chapitre 7

Résultats

Introduction

Afin d'illustrer le résultat d'execution, nous prenons l'exemple d'une petite base de donnée de trois tables : Departement ,Employee, et Project.

7.1 Diagramme de classe de la base de donnés

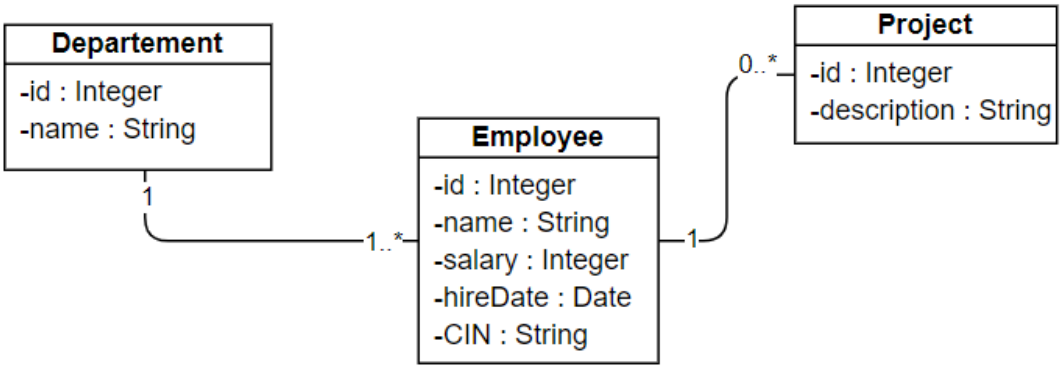

FIGURE 7.1 – Diagramme de classe de la BD

7.2 Exemple de compilation

7.2.1 Input :

```
--Création de table Departement
Departement
  name vc20 /check 'IT','Finance','Accounting'

--Création de la table Employee
Employee /select
  name vc20
  CIN vc10 /unique --numéro de la carte d'identité nationale
  hireDate d
  salary int
  departement_id /fk Departement

--Création de la table Project
Project
  description vc200
  Employee_id /fk Employee


view V1 Employee Departement
```

FIGURE 7.2 – Input du compilateur

7.2.2 Output :

L'Output du compilateur est un fichier ".sql".

Création des tables :

A noter que :

— Des colomnes "id" avec la contrainte "primary key" sont générées pour toute table où aucune colomne ne porte la contrainte "/pk".

— Les commentaires sont inclus dans l'output.

```
-- create tables
create table Departement (
    id      number generated by default on null as identity constraint Departement_id_pk primary key,
    name        varchar(20 char) constraint Departement_name_cc check (name in ('IT','Finance','Accounting'))
)
;

create table Employee (
    id      number generated by default on null as identity constraint Employee_id_pk primary key,
    name        varchar(20 char),
    CIN     varchar(10 char) constraint Employee_CIN_unq unique,
    hireDate        date,
    salary      integer,
    departement_id      number constraint Employee_departement_id_fk references Departement on delete cascade
)
;

--comments
comment on column Employee.CIN is 'numéro de la carte d identité nationale ';
create table Project (
    id      number generated by default on null as identity constraint Project_id_pk primary key,
    description     varchar(200 char),
    Employee_id     number constraint Project_Employee_id_fk references Employee on delete cascade
)
;
```

FIGURE 7.3 – Output : création des tables

Création des vues :

A noter que la clause "where" du code généré est vide, l'utilisateur est amené à modifier le fichier pour y indiquer les conditions de la jointure.

```
-- create views
create or replace view V1 as
select
    Employee.name                    Employee_name,
    Employee.CIN                     Employee_CIN,
    Employee.hireDate                    Employee_hireDate,
    Employee.salary                  Employee_salary,
    Employee.departement_id                  Employee_departement_id,
    Departement.name                     Departement_name
from
    Employee,   Departement
where

/
```

FIGURE 7.4 – Output : création de vues

Selection de données :

La directive de colonne "/select" conduit à la génération du code suivant :

```
-- load data
select
    name,
    CIN,
    hireDate,
    salary,
    departement_id
from Employee;
```

FIGURE 7.5 – Output : Selection

Conclusion générale

Dans ce présent rapport , nous avons présenté en premier lieu le contexte général du projet, puis nous avons décrit le langage adopté(syntaxe,grammaire...), pour ensuite enchaîner par une description détaillée des 4 phases de compilation implémentées à savoir : l'analyse lexicale, l'analyse syntaxique, l'analyse sémantique, et la génération du code. Finalement, nous avons présenté les résultats du travail réalisé.

Ce projet a été une bonne occasion pour consolider nos connaissances en matière de programmation en langage c et cerner les principes de fonctionnement de la chaine de compilation d'un code. Durant le travail sur ce projet, nous avons appris l'importance d'une bonne répartition des tâches et la communication entre les membres de l'équipe, ainsi que l'importance d'une analyse complète et détaillée des spécifications.

Bibliographie

[1] <https://www.libsdl.org/>, 2020. [Online ; accessed 16-January-2020].

[2] <https://www.overleaf.com/project>, 2020. [Online ; accessed 18-January-2020].

[3] <https://www.overleaf.com/project>, 2020. [Online ; accessed 18-January-2020].

[4] <https://www.overleaf.com/project>, 2020. [Online ; accessed 18-January-2020].

[5] Lexbook. Mancala. <http://lexbook.net/en/mancala>, 2015.