目　　录

1. 项目概况

1.1 基本目标

利用 Lex Yacc 工具，设计并实现一个 DBMS 原型系统，可以接受基本的 SQL 语句，对其进行词法分析、语法分析，然后解释执行 SQL 语句，完成对数据库物理文件的相应操作，实现 DBMS 的基本功能。

1.2 完成情况

目前我已经实现《测试数据》中列写出的所有 SQL 语句，主要包括：

1. 创建 CREATE 语句,创建数据库、创建表

   CREATE DATABASE 数据库名;

   CREATE TABLE 表名(列 1 类型,列 2 类型 。。。。。。);

2. SHOW 语句

   SHOW DATABASES; 显示当前的所有数据库

   SHOW TABLES ; 显示当前数据库里的所有表

3. USE 语句

   USE 数据库名; 使用某一数据库指明接下来的表级操作所属的作用域。

4. DROP 语句 删除表或数据库

   DROP DATABASE 数据库名;

   DROP TABLE 表名;

5. SELECT 语句

   无条件查询、条件查询、单表查询、多表查询，带*的查询，指明列结

果的查询，这些查询功能均已实现,

6. UPDATE 语句

UPDATE STUDENT SET SAGE=21 WHERE SSEX=1;

按照 where 后的条件以及 SET 动作更新某个表的某些列

7. DELETE 语句

按照 where 后的条件删除某个表中的某些列

附加功能：SQL 命令、表名、数据库名等字段支持大小写不敏感输入

2. 项目实现方案

2.1 逻辑结构与物理结构

我对数据库物理结构的实现参照了老师给的实验指导中的实现方法：

(1)数据格式为文本，便于调试；

(2)元数据文件为 sys.txt，整个 DBMS 的 sys.txt 存放用户数据库的名字，每个用户数据库的 sys.txt 存放该数据库中每个表的字段类型信息；

(3)基本数据文件为文本文件，每个表的数据分开存放，如 student 表的数据存放在以 student 命名的文本文件中

(4)一个用户数据库的所有表文件和元数据文件放在同一个文件夹里面，这个文件夹以该数据库的名字命名。

(5)所有的字段、数据信息均用空格分隔

表 1 DBMS 元数据的逻辑结构

数据库名
XJGL
IVAN
…

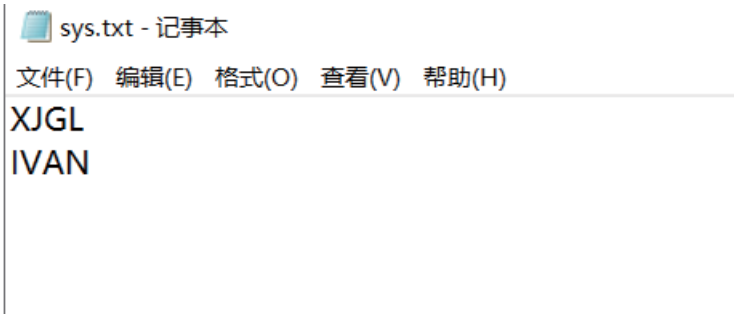

图 1 DBMS 的物理结构示意图

如上图所示，DBMS 包含两个数据库：IVAN 和 XJGL

表 2 用户元数据的逻辑结构

表名	列下标	列名	类型	类型长度
STUDENT	1	SNAME	CHAR	20
STUDENT	2	SAGE	INT	
…	…	…	…	…

图 2 用户数据库的物理结构示意图

如上图所示，为 XJGL 数据库的内容，它包含一个元数据文件（sys.txt）和 3 个表文件 STUDENT、CS 和 COURSE

该物理结构的优点、缺点分析：

优点：易于实现，因为数据皆为文本，具有可读性，测试起来比较方便，直接打开文本就能查看程序的运行结果是否符合预期。每个表对应一个物理文本文件，从人的直观角度来看比较清晰易懂。

缺点：和实际的数据库产品的物理结构有很大的差距，现实当中的数据库肯定不是用这种方式去实现的。表的每一行数据所占用的实际存储空间都是不一样的，存储不够紧凑，以文本形式存储数据的方式相比以二进制形式存储数据的方式来看，文本存储方式的空间利用率太低；此外，此物理结构对参照完整性难以实现，这种实现方法从读取效率、存储空间等角度来看，都不是很好。

2.2 语法结构与数据结构

我的设计思路参考了老师给的指导方案，并且结合自己的实际情况进行了一些修改。

CREATE TABLE 语句的产生式语法结构：

createsql: CREATE TABLE tablename　'('feildefinition')";'

feildefinition: feildefinition','feild_type　|　feild_type

feild_type: feildname type

非终结符 createsql 的属性定义如下结构说明：

```
struct CreateRootValue {
  char *tablename;//表名称
  vector<Col> * feildefinition; //指向一个由列（Col）组成的向量，此向量中的元素信息用于创建表的列，这个向量的每个元素在规约过程中由非终结符 feildefinition 获得
};
```

其中 Col 结构的定义如下：

```
typedef struct {
  int index; //该列处于表中的下标值
```

```
    string name;//列名称，C++STL string 类型
    enum Type type;//枚举  列的类型、INT CHAR…
    int length;//当列的类型为 CHAR 时  标识 CHAR 的长度限制
    union Value value;//value 成员用于暂存当前行的数据
}Col;
```

非终结符 feildefinition 的属性为 Col 类型,已在上文中给出，Col 结构可以存放列的类型信息。非终结符 fieldname 的属性为标识符（字符串），type 的类型为自定义的 type 结构（类型+长度）：

```
struct struct_type {
  enum Type type; //类型
  int char_length;//长度
};
```

注：这里我用了一个向量 vector<Col> colsList 用以代替实验指导中的链表结构，故我的 Col 结构体中没有出现 Col * next 域等字段。

在每次非终结符 feildefinition 被规约出来时就往这个全局向量里面 push 一个新的 Col 类型的元素进去，这个 Col 变量存储着之前规约过程中获得的列结构信息（列名称、类型）。整个 create 语句规约出来以后，这个向量中就存储了所有的列创建信息。举例：

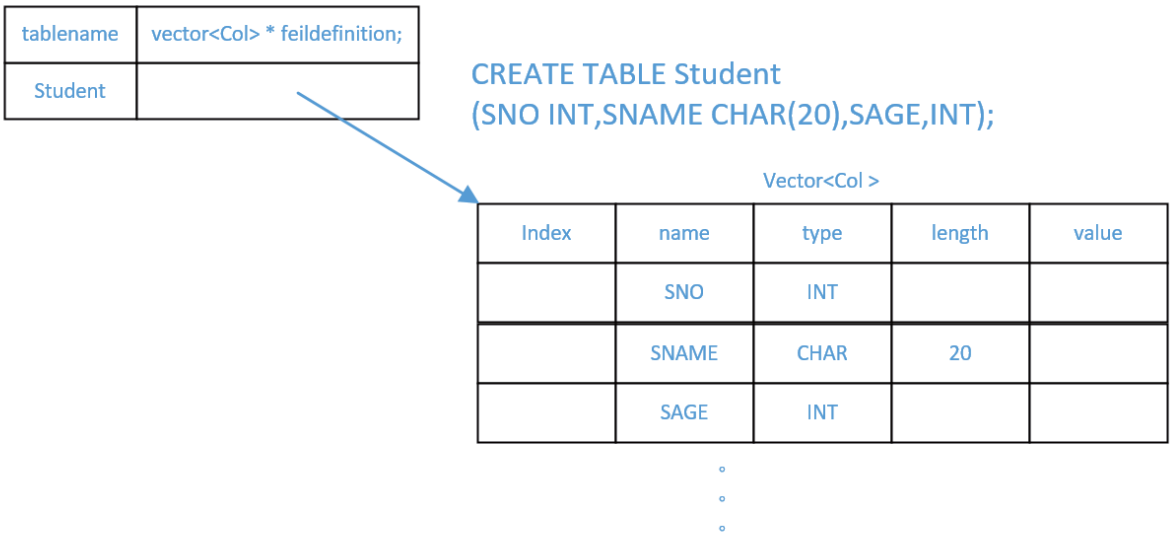

图 3 Create 数据结构图

Select 语句的产生式：

selectsql: SELECT fields_star FROM tables ';'

|SELECT fields_star FROM tables WHERE conditions ';'

fields_star: table_fields | '*'

table_fields: table_fields ',' table_field | table_field

table_field: fieldname | tablename '.' fieldname

非终结符 selectsql 的属性定义如下：

struct SelectRootValue {

vector<SelectCol> * cols; //fields_star 字段的列选取结果，所有的 select 结果列名存放在一个向量里面 这里的实现方法同上文的 Create 语句

vector<TableName> * tables; //tables 字段结果，存放要选取表的表名。

struct Conditions *condition;//condition 字段，存放条件信息

};

其中 TableName 为字符串类型的数据，SelectCol 也为字符串类型的数据，它表示要选取的列名字。非终结符 table_field 的属性定义为两个字符串，分别表示表名和列名比如“Student.Sname”。非终结符 tablename 和 feildname 属性均为字符串类型。非终结符 table_fields 的属性类型为 SelectCol,其相关说明已在上文给出。

对于 Condition 的设计方式参考了实验指导中的设计思路：其数据结构如下

```
struct Conditions {/*条件*/
    struct Conditions *left; //左部条件
    struct Conditions *right; //右部条件
    char * comp_op; /* 'a'是 and, 'o'是 or, '<' , '>' , '=',  '!=' */
    enum Type type; /* 0 是字段，1 是字符串，2 是整数 */
    union Value value;;/* 根据 type 存放字段名、字符串或整数 */
    char * table;/* NULL 或表名  */
};
```

Where 后面的复合条件构成了一棵条件树

举例

SELECT SNO,SNAME FROM Student Where Sno=1 AND Sname=“Ivan”;

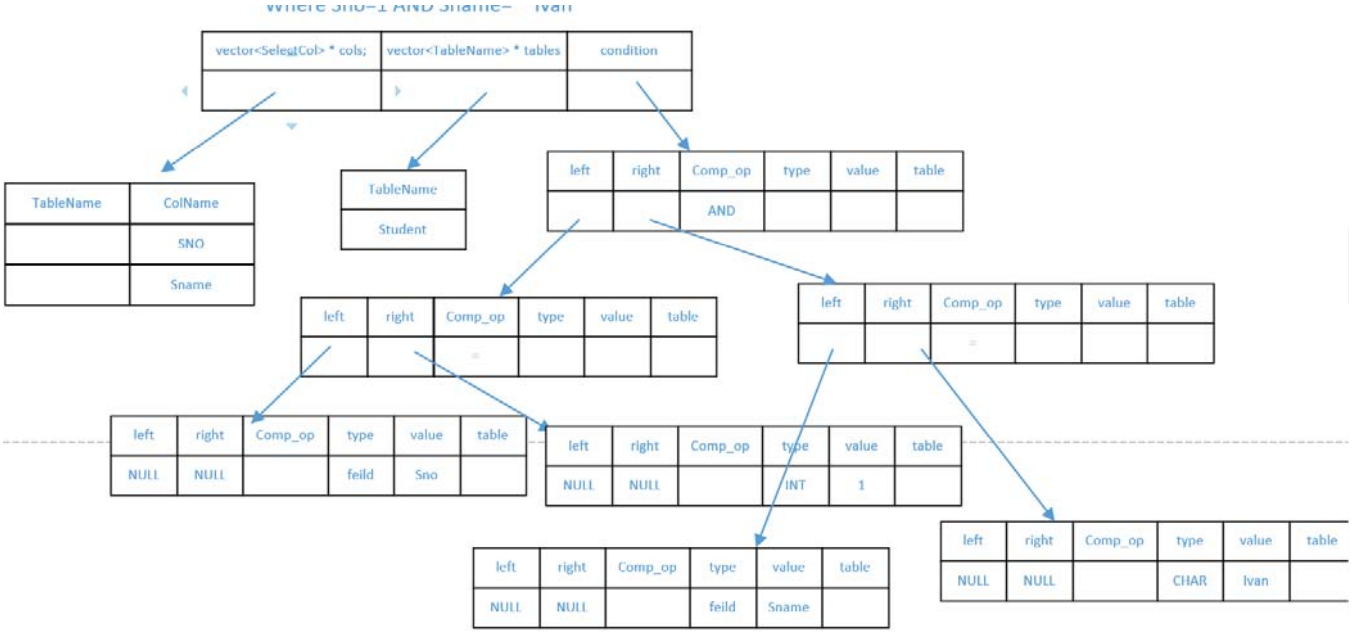

图 4 Select 语句数据结构图

Select 语句和 Create 语句中的数据结构定义方法涵盖了本实验大多数的情况，下面介绍的其他语句的实现思路与之大体类似，许多地方也都借鉴了这两个语句的设计思路。

Insert 语句产生式如下：

insertsql: INSERT INTO tablename '(' insert_feild ')' VALUES '(' insert_values ')' ';'

| INSERT INTO tablename VALUES '(' insert_values ')' ';'

insert_feild: insert_feild ',' feildname | feildname

insert_values: insert_values','value | value

value: STRING | INTEGER | FLOATE

非终结符 insertsql 的数据结构如下：

struct InsertRootValue {

TableName table; //表名称，字符串

vector<ColName> * cols;//一个存放 insert_field 字段信息的向量

vector<yyValue> * values;//存放 insert_value 字段信息的向量

};

其中 ColName 为字符串类型 yyValue 为自定义结构体：

```
struct yyValue {
    union Value value; //可以是整形，char *  和浮点类型
    struct struct_type type_length;//类型+长度
};
```

非终结符 tablename 的类型为字符串 insert_field 的类型为 ColName（字符串）insert_values 的类型为 yyValue 类型（类型+长度）

举例：

INSERT INTO STUDENT (Sno,Sname) VALUES (123,”Jason”);

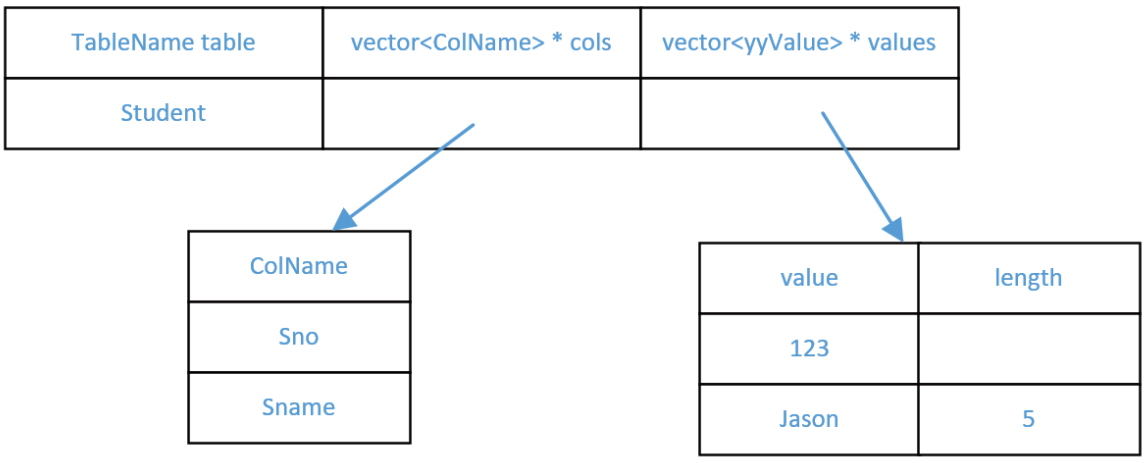

图 5 Insert 语句数据结构图

Update 产生式：

updatesql : UPDATE tablename SET setinfo WHERE conditions ';'

setinfo: setinfo ',' feild OPERATOR value　| feild OPERATOR value

updatesql 的数据结构：

```
struct UpdateRootValue {
    char * tablename;//tablename 字段，表名称，字符串
    Conditions * conditions;//conditions 字段，条件树
    vector<struct Feild_Value> *setList;//setinfo 字段，存放 setinfo 动作信息的一个向量
};
```

Tablename 的属性为字符串。Setinfo 的属性为自定义结构 struct Feild_Value 类型：

```
struct Feild_Value {
    struct Feild feild;
    union Value value;
    enum Type type;
    int length;
};
```

Field 的属性为 Field 类型

```
struct Feild {
    char * tablename;
    char * feildname;
};
```

举例：　UPDATE Student SET Sno=2,Sage=15 WHERE conditions ;

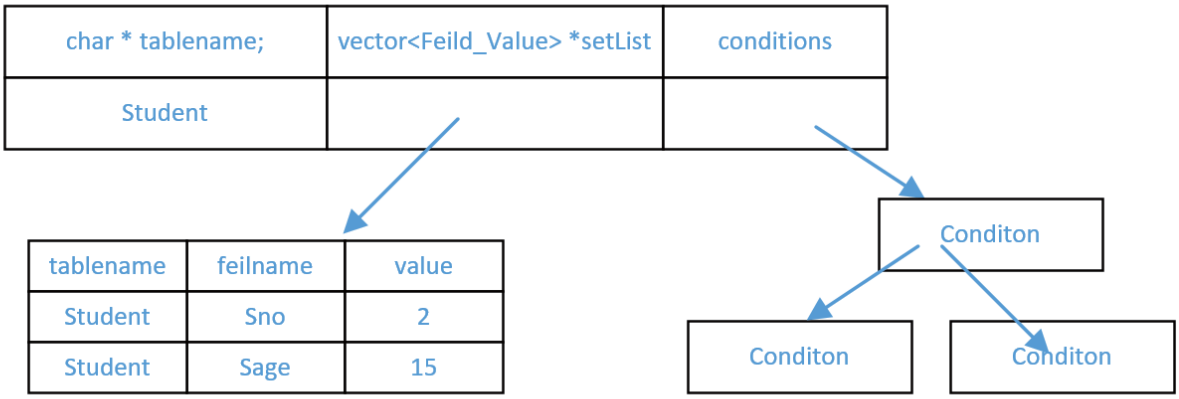

图 6 Update 语句数据结构图

Delete 语句：

产生式：deletesql: DELETE FROM tablename WHERE conditions ';'

Deletesql 的数据结构：

```
struct DeleteRootValue {
  char * tablename;//tablename 字段，表名称，字符串
  Conditions *conditions;//条件树
};
```

举例：DELETE FROM Student WHERE Sno=1;

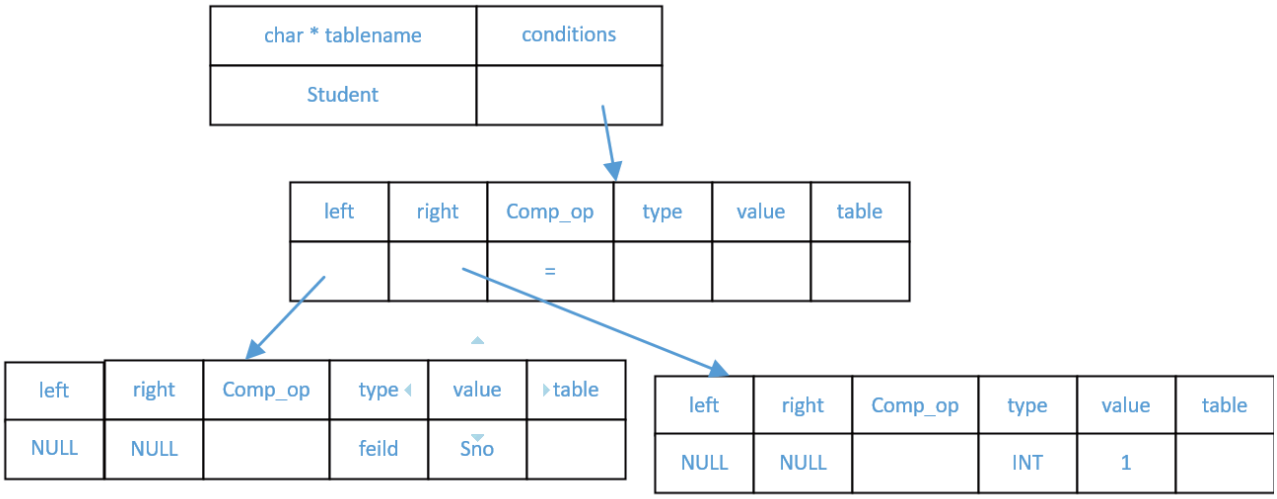

图 7 Delete 语句数据结构图

Drop 语句产生式：

dropsql: DROP TABLE tablename ';'

非终结符 dropsql 的数据结构：一个串类型的变量，存放表的名称。

Show 语句：

产生式：

showdb : SHOW DATABASES ';'

showsql: SHOW TABLES ';'

DataBase 相关的语句产生式：

createdb: CREATE DATABASE databasename ';' //非终结符 createdb 属性为字符串

dropdb: DROP DATABASE databasename ';' //非终结符 dropdb 属性为字符串

usedb : USE databasename ';' /非终结符/usedb 属性为字符串

databasename: ID;

2.3 执行流程

每次当一个 Sql 语句被完整的规约出来的时候，就把上述语法树根节点的属性值作为相应的函数参数传递给我写的语义函数，执行对应的数据库操作，下面一一介绍我编写的主要的语义函数：

函数名称：bool ZYF_CreateTable(CreateRootValue * createInfo);

函数说明：根据输入参数中包含的表信息构建一个物理表

输入参数： createInfo：非终结符 createsql 的属性值

输出参数：true 表示操作成功，flase 表示操作失败

执行流程：

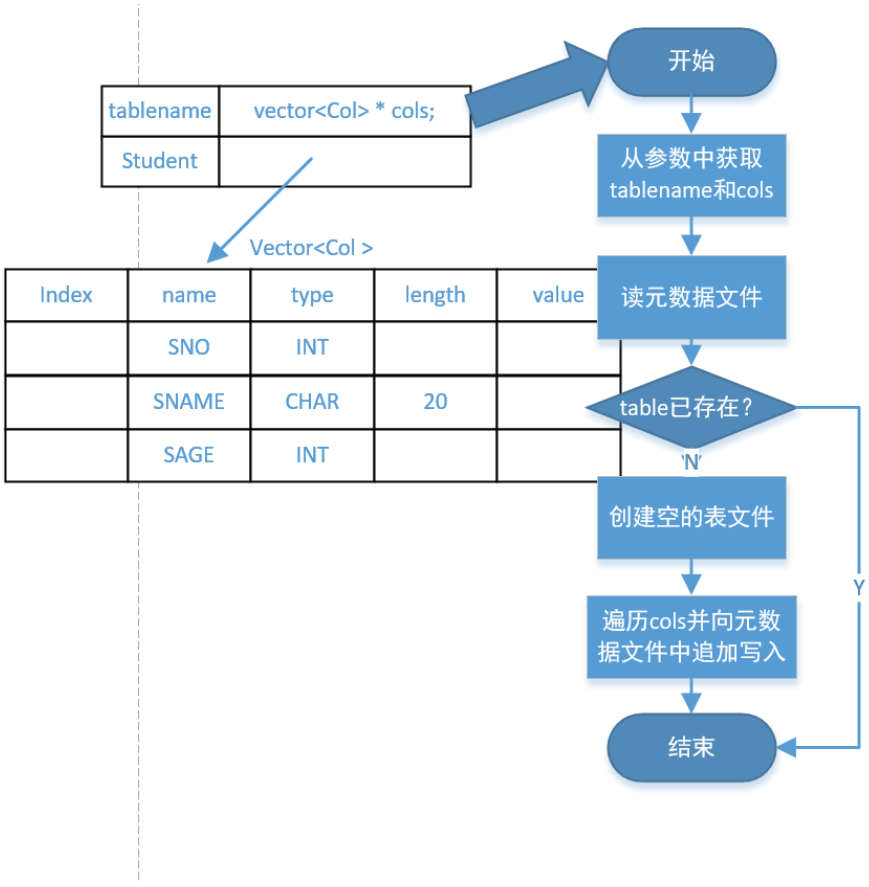

图 8 Create 语义函数流程图

Create 语句执行成功后会在数据库目录下新建一个空的表文件，并将其信息

填写到元数据文件里面。

函数名称：bool ZYF_Select(SelectRootValue * selectInfo);

函数说明： 根据输入参数中包含的 select 信息将查询结果打印出来

输入参数： selectInfo：非终结符 selectsql 的属性值

输出参数：true 表示操作成功，flase 表示操作失败

执行流程：

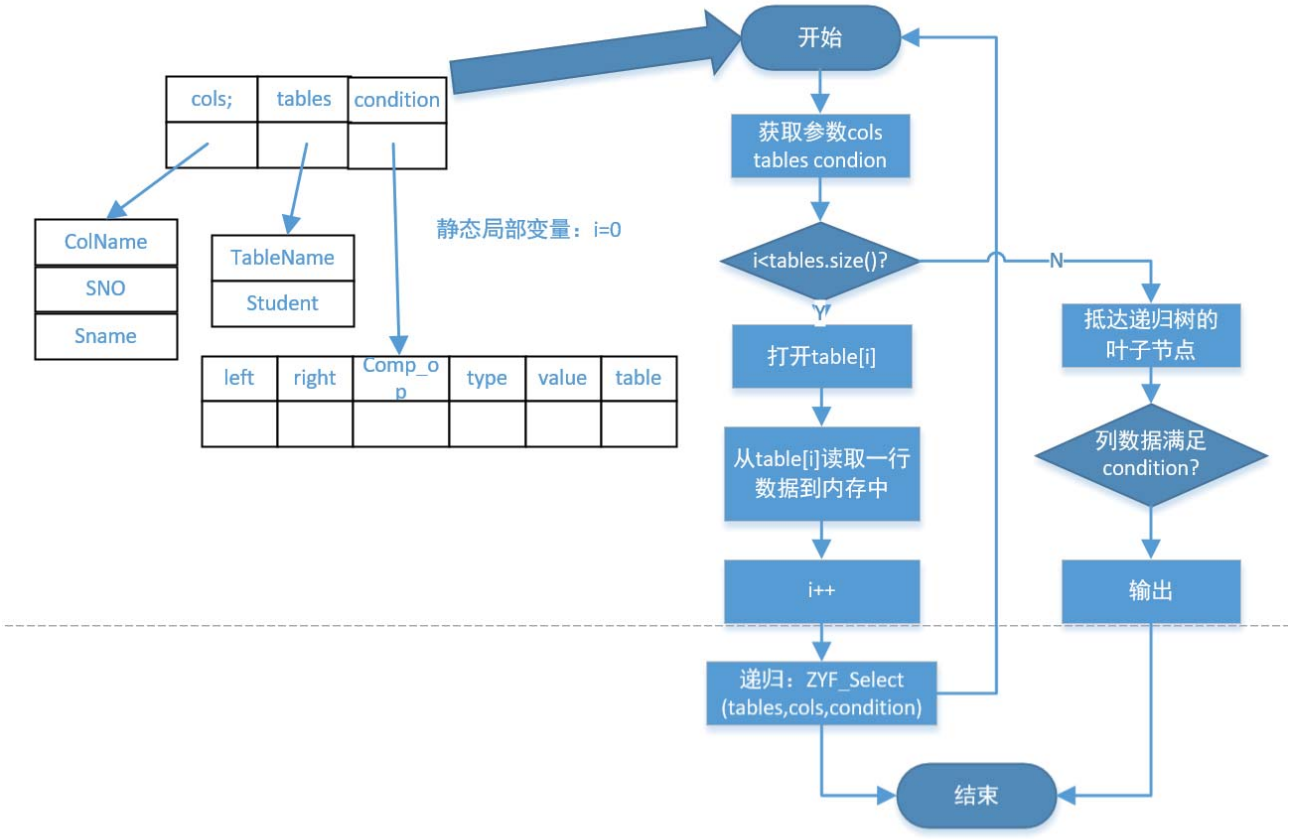

图 9 Select 语义函数流程图

前面提到过 Col 这个数据结构

```
typedef struct {
    int index; //该列处于表中的下标值
```

string name;//列名称，C++STL string 类型

enum Type type;//枚举 列的类型、INT CHAR…

int length;//当列的类型为 CHAR 时 标识 CHAR 的长度限制

union Value value;//value 成员用于暂存当前行的数据

};Col;

之前是作为一个非终结符的属性介绍的，现在我要介绍一下它的更重要的功能：在 Select Update Delete 等语句 ，存储表的一行数据。

比如这样的三个 Col 类型的变量可以存储如下数据：

Index	name	type	length	value
	SNO	INT		123
	SNAME	CHAR	20	Ivan
	SAGE	INT		15

图 10 Col 数据结构图

那么这三个 Col 数据就构成了 Student 表的一行数据：

Sno Sname Sage

123 Ivan 15

在 select 语句中就能用于判断是否满足条件。

对于足 condition 条件的计算 calculate(condtion)，算法流程如下：

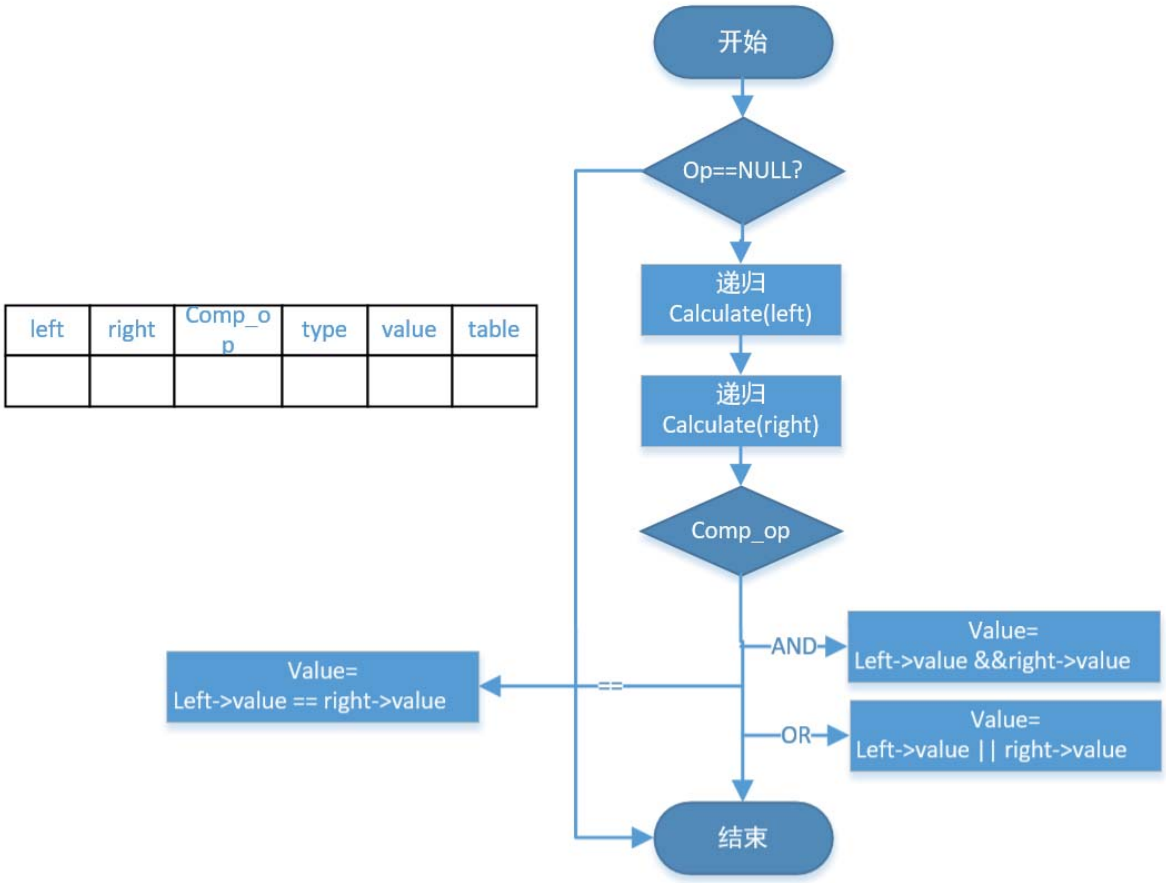

图 11 条件表达式求解函数流程图

表达式值计算函数的实现关键点是，要递归地计算 left 和 right 两个子表达式的值，然后根据操作符的类型进行 left OP right 运算，并将结果存在 value 属性里面，最后根据计算结果的类型修改 type 属性的值，表示这一表达式已经计算出结果。在具体实现的时候，我加入了操作数类型检查机制，不兼容的两种类型的变量进行二元运算是被禁止的。

函数名称：bool ZYF_Insert(InsertRootValue * insertInfo);

函数说明： 根据输入参数中包含的 insert 信息对一个表进行插入操作

输入参数： insertInfo：非终结符 insertsql 的属性值

输出参数：true 表示操作成功，flase 表示操作失败

执行流程：

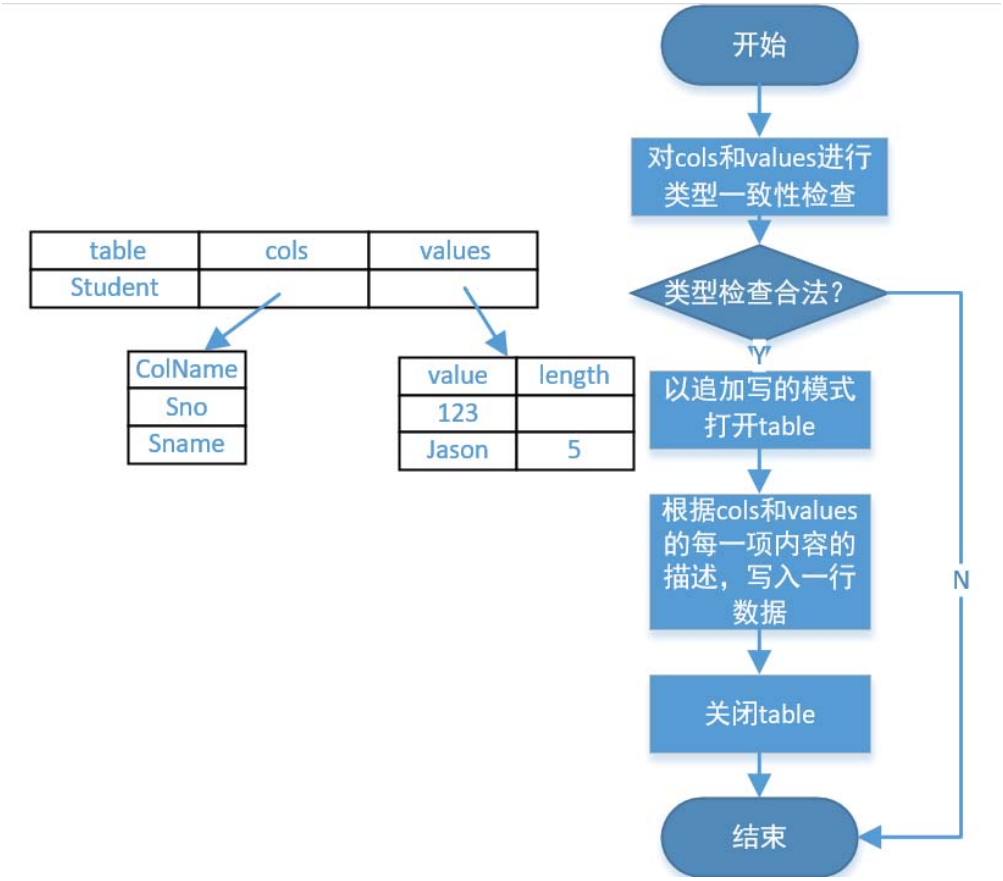

图 12 Insert 语义函数流程图

我对 Insert 语句做了类型检查和插入合法性检查，比如

Insert into STUDENT (Sno,Same) values(123,“ivan”,123)；

这样的语句是被禁止执行的，因为待插入的列数少于值的数量，同理 Insert into STUDENT (Sno, Same，Ssex) values(123,123)也是被禁止的。除此之外，如果待插入列和值的类型不匹配也是禁止的，比如 Insert into STUDENT (Sno,Same) values(123,123)。

函数名称：bool ZYF_Update(UpdateRootValue * updateInfo);

函数说明： 根据输入参数中包含的 update 信息对一个表中的符合条件的列进行更新

输入参数： updateInfo：非终结符 updatesql 的属性值

输出参数：true 表示操作成功，flase 表示操作失败

执行流程：

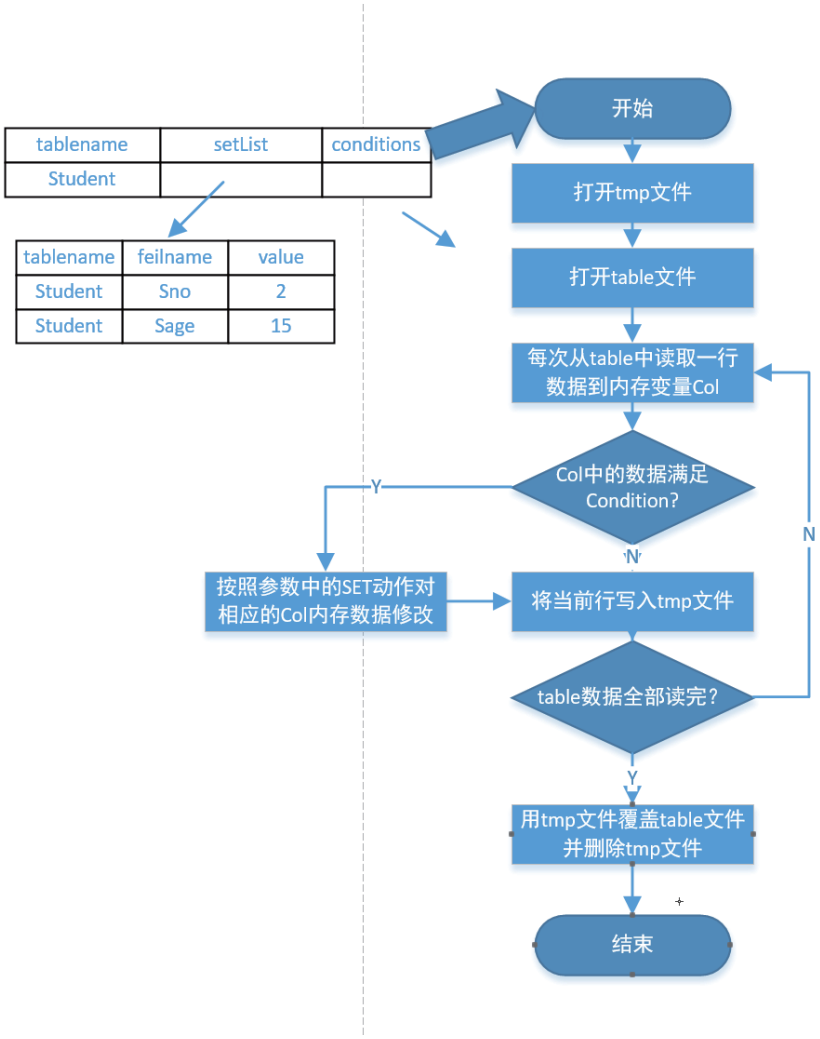

图 13 Update 语义函数流程图

Update 语句也会进行类型检查，包括 Set 动作中左操作数和右操作数类型是否匹配，比如 Set Sname=123 就是不合法的；因为物理结构设计的限制，这里我没有想出很高效的更新算法，我的处理方式是先一条条读出来，然后判断满足条件，往新文件里面写入数据，最后用新文件代替旧的表文件，总感觉这种做法太差了，IO 操作太频繁，降低了程序的效率，除了 Update 以外，Delete 语句也是如此，删除数据也会导致大量的 IO 操作，这里有待改进，如果我的物理

存储方法采用二进制存储，同时根据列的类型信息，限制每一条数据的存储长度，让整个表的数据存储紧凑有规律的话，就能通过读写指针直接定位修改了，可以省去不必要的 IO 操作。

函数名称：bool ZYF_Delete(DeleteRootValue * deleteInfo);

函数说明： 根据输入参数中包含的 delete 信息对一个表中的符合条件的列进行删除

输入参数： deleteInfo：非终结符 deletesql 的属性值

输出参数：true 表示操作成功，flase 表示操作失败

执行流程：

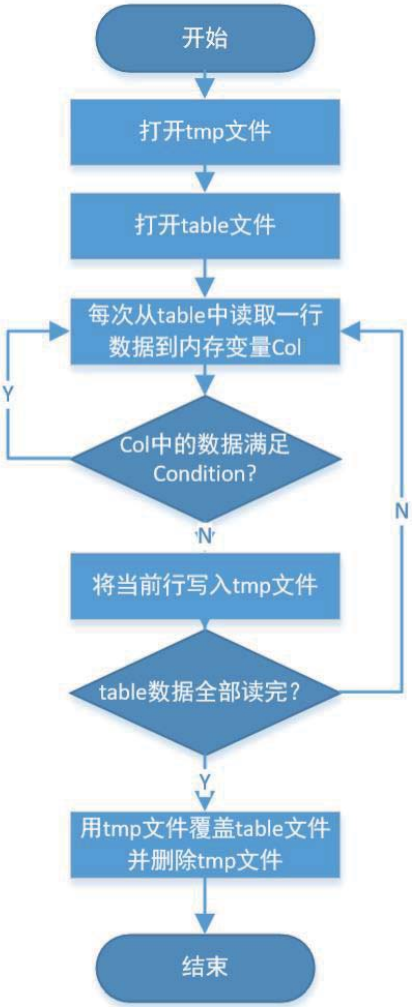

图 14 Delete 语义函数流程图

函数名称：bool ZYF_DropTable(char * tablename);

函数说明： 删除一个表

输入参数： tablename：要删除的表名

输出参数：true 表示操作成功，flase 表示操作失败

执行流程：

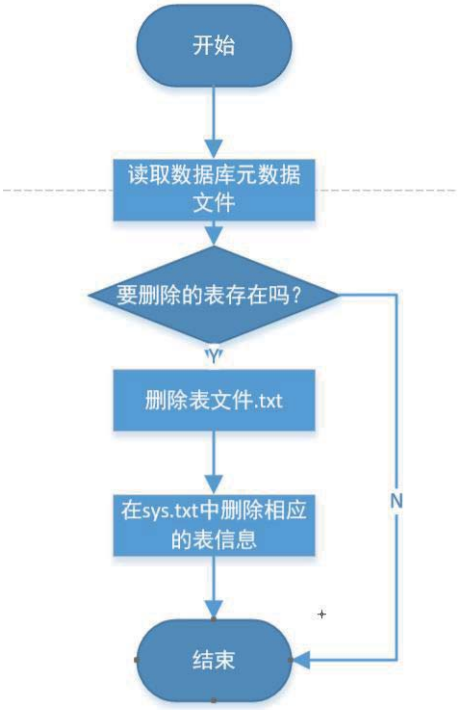

图 15 DropTable 语义函数流程图

函数名称：bool ZYF_DropDataBase(char * database);

函数说明： 删除一个数据库。同时删除其所含的所有表

输入参数： database:要删除的数据库名字

输出参数：true 表示操作成功，flase 表示操作失败

执行流程：

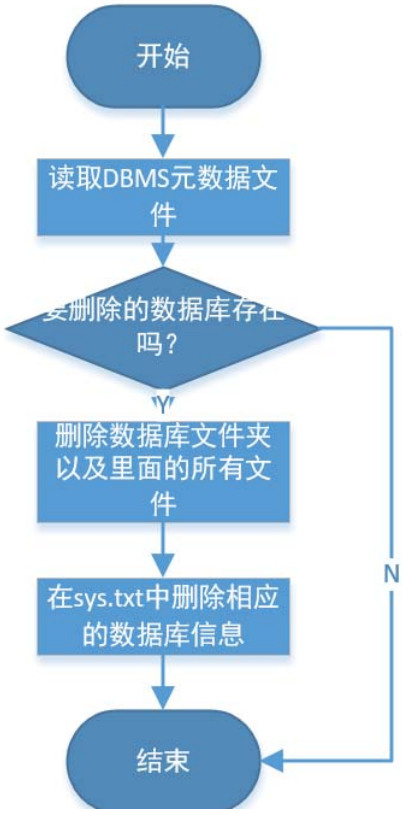

图 16 DropDataBase 语义函数流程图

函数名称：bool ZYF_CreateDataBase(char * database);

函数说明： 创建一个数据库

输入参数： database:要创建的数据库名字

输出参数：true 表示操作成功，flase 表示操作失败

执行流程：

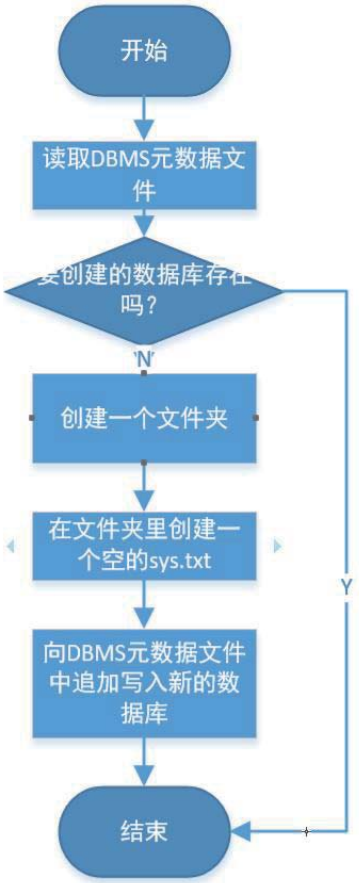

图 17 CreateDataBase 语义函数流程图

函数名称：bool ZYF_UseDataBase(char * database);

函数说明：使用数据库

输入参数： database:要使用的数据库名字

输出参数：true 表示操作成功，flase 表示操作失败

执行流程：

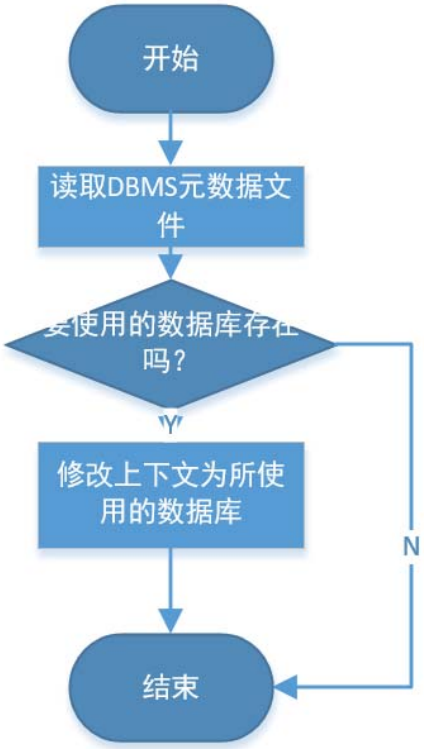

图 18　UseDataBase 语义函数流程图

函数名称：void ZYF_ShowTables();

函数说明：　打印出当前用户数据库下的所有表信息

执行流程：

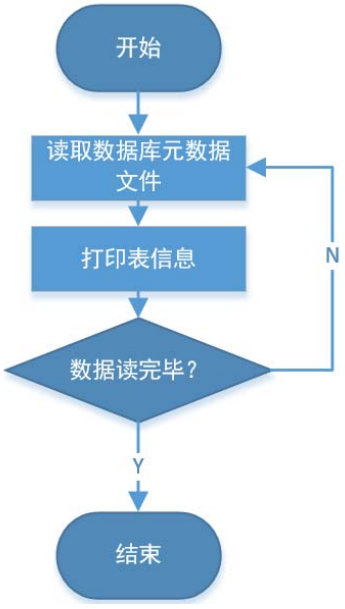

图 19　ShowTables 语义函数流程图

函数名称：void ZYF_ShowDataBase();

函数说明： 打印出 DBMS 所具有的所有数据库信息

执行流程：

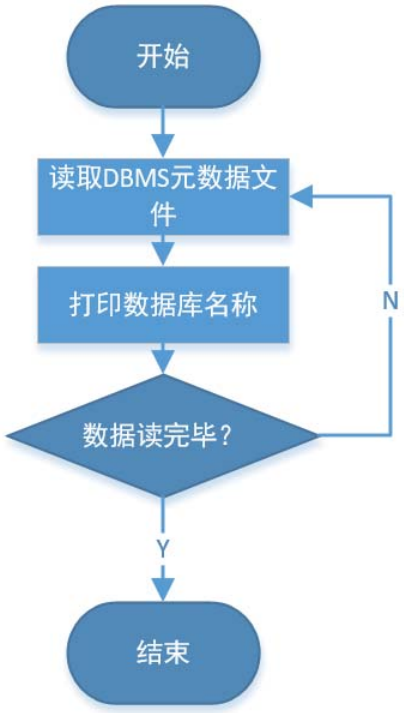

图 20　ShowDatabases 语义函数流程图

2.4 功能测试

测试 1

输入：CREATE DATABASE XJGL;

输出：create database success

测试 2

输入：CREATE DATABASE XJGL;

输出：CreateDataBase Error : database XJGL has existed

create database fail!

测试 3

输入：CREATE DATABASE JUST_FOR_TEST;

输出：create database success

测试 4

输入：CREATE DATABASE JUST_FOR_TEST;

输出：CreateDataBase Error : database JUST_FOR_TEST has existed

create database fail!

测试 5

输入：SHOW DATABASES;

输出：

databases:

XJGL

JUST_FOR_TEST

测试 6

输入：DROP DATABASE JUST_FOR_TEST;

输出：drop database sucess

测试 7

输入：SHOW DATABASES;

输出：

databases:

XJGL

测试 8

输入：USE XJGL;

输出：use database success

//测试 CREATE TABLE SHOW TABLES DROP TABLE

测试 9

输入：

CREATE TABLE STUDENT(SNAME CHAR(20),SAGE INT,SSEX INT);

CREATE TABLE COURSE(CNAME CHAR(20),CID INT);

输出：Create Table Success Create Table Success

测试 10

输入：CREATE TABLE CS(SNAME CHAR(20),CID INT);

输出：Create Table Success

测试 11

输入：

CREATE TABLE TEST_TABLE(COL1 CHAR(22),COL2 INT,COL3 CHAR(22);

输出：syntax error in line 1: ;

测试 12

输入：

CREATE TABLE TEST_TABLE(COL1 CHAR(22),COL2 INT,COL3 CHAR(22));

输出：Create Table Success

测试 13

输入：SHOW TABLES;

输出：

STUDENT :SNAME SAGE SSEX
COURSE :CNAME CID
CS :SNAME CID
TEST_TABLE :COL1 COL2 COL3 COL1 COL2 COL3

测试 14

输入：DROP TABLE TEST_TABLE;

输出：drop Success

测试 15

输入：SHOW TABLES;

输出：

STUDENT :SNAME SAGE SSEX
COURSE :CNAME CID
CS :SNAME CID

//测试 INSERT INTO VALUES

测试 16

输入：

INSERT INTO STUDENT(SNAME,SAGE,SSEX) VALUES ('ZHANGSAN',22,1);

输出：insert Success

测试 17

输入：

INSERT INTO STUDENT(SNAME,SAGE,SSEX) VALUES ('ZHANGSAN',22,1);

INSERT INTO STUDENT VALUES ('LISI',23,0);

INSERT INTO STUDENT(SNAME,SAGE) VALUES ('WANGWU',21);

INSERT INTO STUDENT VALUES ('ZHAOLIU',22,1);

INSERT INTO STUDENT VALUES ('XIAOBAI',23,0);

INSERT INTO STUDENT VALUES ('XIAOHEI',19,0);

INSERT INTO COURSE(CNAME,CID) VALUES ('DB',1);

INSERT INTO COURSE (CNAME,CID) VALUES('COMPILER',2);

insert into course (CNAME,CID) VALUES('C',3);

输出：insert Success

//测试单表查询

测试 18

输入：SELECT SNAME,SAGE,SSEX FROM STUDENT;

输出：

```
ZYF_SQL>>SELECT SNAME,SAGE,SSEX FROM STUDENT;
+----------+----------+----------+
|SNAME     |SAGE      |SSEX      |
+----------+----------+----------+
|ZHANGSAN  |22        |1         |
|LISI      |23        |0         |
|WANGWU    |21        |0         |
|ZHAOLIU   |22        |1         |
|XIAOBAI   |23        |0         |
|XIAOHEI   |19        |0         |
+----------+----------+----------+
```

图 21 测试 18 结果图

测试 19

输入：SELECT SNAME,SAGE FROM STUDENT;

输出：

```
ZYF_SQL>>SELECT SNAME,SAGE FROM STUDENT;
SNAME                SAGE
-------------------- --------------------
ZHANGSAN             22
LISI                 23
WANGWU               21
ZHAOLIU              22
XIAOBAI              23
XIAOHEI              19
```

图 22 测试 19 结果图

测试 20

输入：SELECT * FROM STUDENT;

输出：

```
ZYF_SQL>>SELECT * FROM STUDENT;
SNAME                SAGE                 SSEX
-------------------- -------------------- --------------------
ZHANGSAN             22                   1
LISI                 23                   0
WANGWU               21                   0
ZHAOLIU              22                   1
XIAOBAI              23                   0
XIAOHEI              19                   0
```

图 23 测试 20 结果图

测试 21

输入：SELECT SNAME,SAGE FROM STUDENT WHERE SAGE=21;

输出：

```
ZYF_SQL>>SELECT SNAME,SAGE FROM STUDENT WHERE SAGE=21;
SNAME                |SAGE                |
---------------------|--------------------|
WANGWU               |21                  |
```

图 24 测试 21 结果图

测试 22

输入：SELECT SNAME,SAGE FROM STUDENT WHERE (((SAGE=21)));

输出：

```
ZYF_SQL>>SELECT SNAME,SAGE FROM STUDENT WHERE (((SAGE=21)));
SNAME                |SAGE                |
---------------------|--------------------|
WANGWU               |21                  |
```

图 25 测试 22 结果图

测试 23

输入：SELECT SNAME,SAGE FROM STUDENT WHERE (SAGE>21) AND (SSEX=0);

输出：

```
ZYF_SQL>>SELECT SNAME,SAGE FROM STUDENT WHERE (SAGE>21) AND (SSEX=0);
SNAME                |SAGE                |
---------------------|--------------------|
LISI                 |23                  |
XIAOBAI              |23                  |
```

图 26 测试 23 结果图

测试 24

输入：

SELECT SNAME,SAGE FROM STUDENT WHERE (SAGE>21) OR (SSEX=0);

输出：

```
ZYF_SQL>>SELECT SNAME,SAGE FROM STUDENT WHERE (SAGE>21) OR (SSEX=0)
SNAME           |SAGE
----------------|----------
ZHANGSAN        |22
LISI            |23
WANGWU          |21
ZHAOLIU         |22
XIAOBAI         |23
XIAOHEI         |19
```

图 27 测试 24 结果图

测试 25

输入：SELECT * FROM STUDENT WHERE SSEX!=1;

输出：

```
ZYF_SQL>>SELECT * FROM STUDENT WHERE SSEX!=1;
SNAME           |SAGE      |SSEX
----------------|----------|----------
LISI            |23        |0
WANGWU          |21        |0
XIAOBAI         |23        |0
XIAOHEI         |19        |0
```

图 28 测试 25 结果图

// 测试多表查询

测试 26

输入：select * from student,course;

输出：

```
ZYF_SQL>>select * from student,course;
```

SNAME	SAGE	SSEX	CNAME	CID
ZHANGSAN	22	1	DB	1
ZHANGSAN	22	1	COMPILER	2
ZHANGSAN	22	1	C	3
LISI	23	0	DB	1
LISI	23	0	COMPILER	2
LISI	23	0	C	3
WANGWU	21	0	DB	1
WANGWU	21	0	COMPILER	2
WANGWU	21	0	C	3
ZHAOLIU	22	1	DB	1
ZHAOLIU	22	1	COMPILER	2
ZHAOLIU	22	1	C	3
XIAOBAI	23	0	DB	1
XIAOBAI	23	0	COMPILER	2
XIAOBAI	23	0	C	3
XIAOHEI	19	0	DB	1
XIAOHEI	19	0	COMPILER	2
XIAOHEI	19	0	C	3

图 29 测试 26 结果图

测试 27

输入：SELECT * FROM STUDENT,COURSE WHERE (SSEX=0) AND (CID=1);

输出：

```
ZYF_SQL>>SELECT * FROM STUDENT,COURSE WHERE (SSEX=0) AND (CID=1);
```

SNAME	SAGE	SSEX	CNAME	CID
LISI	23	0	DB	1
WANGWU	21	0	DB	1
XIAOBAI	23	0	DB	1
XIAOHEI	19	0	DB	1

图 30 测试 27 结果图

//测试 DELETE 语句

测试 28

输入：SELECT * FROM STUDENT;

输出：

```
ZYF_SQL>>SELECT * FROM STUDENT;
```

SNAME	SAGE	SSEX
ZHANGSAN	22	1
LISI	23	0
WANGWU	21	0
ZHAOLIU	22	1
XIAOBAI	23	0
XIAOHEI	19	0

图 31 测试 28 结果图

测试 29

输入：DELETE FROM STUDENT WHERE (SAGE>21) AND (SSEX=0);

输出：delete Success

测试 30

输入：SELECT * FROM STUDENT;

输出：

```
ZYF_SQL>>SELECT * FROM STUDENT;
```

SNAME	SAGE	SSEX
ZHANGSAN	22	1
WANGWU	21	0
ZHAOLIU	22	1
XIAOHEI	19	0

图 32 测试 29 结果图

测试 31

输入：UPDATE STUDENT SET SAGE=21 WHERE SSEX=1;

输出：update Success

测试 32

输入：

UPDATE STUDENT SET SAGE=27,SSEX=1 WHERE SNAME='ZHANGSAN';

输出：update Success

3. 总结与未来工作

3.1 未完成功能

在没有上实验课之前，其实我最初的实现想法是这样的：按照理论课上学的方法，赋值语句的翻译、布尔条件语句的翻译、控制语句的翻译，把 SQL 语句全部翻译成三地址码，然后写一个三地址码解释模块，对生成的三地址码序列进行解释执行，完成实验任务。这样做就是从生成中间代码的角度出发了。

3.2 未来实现方案

我觉得我的想法是可行的，方法在理论课上也都学过了，主要是 SQL 语句如何转换成对应的控制语句、赋值语句等等，比如 select 语句就能转成一个循环操作，循环读取表中的数据，然后判断是否符合条件，根据我在理论课上学到的知识，最初的这种想法需要用到大量的拉链回填技术，同时也需要注意维护好符号表，设计一个能进行高效查找的符号表数据结构。

此外，我的实现代码里面，并不支持 select 的嵌套查询，这是一个改进的地方，如何判断每一条组合数据是否符合条件，以及条件语句中出现的列名字的作用域问题。我完成的代码中对 select 的处理方法是用递归的方式逐条读取外存数据到内存的一个固定大小的数据结中，然后判断是否满足条件；我想如果是嵌套查询的话，因为有作用域问题，我想应该也是需要一个类似嵌套型的符号

表这样的结构来确定引用的作用域，此外我需要具体深入地了解一下实际的 DBMS 对这些语句的处理算法，因为我总感觉自己想出的方法不是特别好，特别合适。

总而言之，这学期我在编译原理这门课上投入了很多的时间，在验收的过程中也发现了许多不足之处和一些需要改进的编程漏洞，深知一个合格的 DBMS 是多么的严格，多么的一丝不苟。我还要继续学习，对以后的工作要更加认真严谨，十分感谢老师给我这个锻炼的机会，暑假里我要再完善这个大作业，把它做得更好。