编译实验实验报告

姓名: xxx

学号：xxx

2021 年 6 月 30 日

目录

1 实验内容

本次编译原理实验中，我们借助 Lex/Yacc 等分析工具，实现了一个较为简单的数据库系统。该数据库系统为磁盘数据库，将数据存储在本地，程序结束后数据不会消失。用户可以在终端输入 SQL 语句操作数据库，执行相应的 SQL 功能。

2 实验环境

Ubuntu 20.04.2 LTS, flex 2.6.4，bison (GNU Bison) 3.5.1 以及 C++ 开发环境。

3 内容分析

首先，我们考虑希望实现的所有数据库功能及对应的 SQL 语句格式，罗列如下：

操作描述	**SQL 语句**
创建数据库	create database {数据库名}
删除数据库	drop database {数据库名}
使用数据库	use {数据库名}
显示所有数据库	show databases
创建表单	create table {列名 列类型, ...}
删除表单	delete table {表单名}
展示所有表单	show tables
展示某表单的属性	desc {表单名}
插入	insert into {表单名} ... [where ...]
查找	select ... from {表单名, ...}
删除	delete from {表单名} [where ...]
更新	update {表单名} set ... [where ...]
删除表单	drop table {表单名}
退出	exit

综合考虑后，我使用 Lex 生成文法记号，Yacc 对语法进行分析，当识别出不同的命令时，再分不同的情况进行功能实现：

- 对于仅涉及到一、两个字符串的命令 (如创建数据库、删除表单、退出等)，Yacc 直接将该字符串传给功能实现接口进行处理。

- 对于涉及到以列表形式给出指令 (如创建数据表、插入元组、查找元组等) 的情况时，Yacc 在对语法树归结时，直接将 SQL 命令中的列表信息以 {内容所在的指针 + 内容有效字节} 的格式一条条

插入预先给出的全局链表中；当整条语句解析完成后，Yacc 通知功能实现模块，功能实现模块访问该全局链表，按照预先商定好的顺序对其进行解析，进而完成 SQL 功能。

4 具体实现

考虑到在本套系统中，Lex 与其余部分的联系稍微弱一些，因而在本章节我们主要分三部分进行阐述。第一部分介绍 Lex 处理单词的方式，第二部分介绍数据库物理结构与逻辑结构，第三部分介绍 Yacc 对记号流的解析以及对应 C++ 数据库操作接口的设计。

4.1 Lex 词法分析

Lex 负责解析输入的字符串，并将其转换为记号流传达给 Yacc。

4.1.1 解析 SQL 关键字

用户可能输入 SQL 关键字，此时 Lex 不区分大小写的解析出他们，并输出对应的宏定义值。以几个 SQL 关键字为例，给出代码详细说明 Lex 如何处理关键字：

定义部分

```
create [cC][rR][eE][aA][tT][eE]
database [dD][aA][tT][aA][bB][aA][sS][eE]
databases [dD][aA][tT][aA][bB][aA][sS][eE][sS]
use [uU][sS][eE]
table [tT][aA][bB][lL][eE]
```

识别规则部分

全部大写的返回值为 Yacc 中预先宏定义的常量值。

```
{create} {return CREATE;}
{database} {return DATABASE;}
{databases} {return DATABASES;}
{use} {return USE;}
{table} {return TABLE;}
```

4.1.2 解析自定义的字串

用户可能输入如下三种自定义的类型：

- 整数 (由 0-9 构成的数，允许前导 0)
- 字符串 (在本实验中我们认为字符串必须以**双引号**为首尾，暂不与标准 SQL 对应)
- 变量名 (包含数字、52 个大小写字母与下划线，且不以数字开头)。

定义部分

两个基本的类型 charType 和 digitType 分别表示可用的字符以及 10 个数码。 integer 是由若干个基本数码组成的一个合法的允许前导 0 的整数。 id 以一个字符起头，随后跟任意数量（可以为 0）的数码或字符。 string 以双引号为首尾，中间有任意数量（可以为 0）的数码或字符。

a.l

```
charType [a-zA-Z_]
digitType [0-9]

integer {digitType}+
id {charType}({charType}|{digitType})*
string \"({charType}|{digitType})*\"
```

识别规则部分

当识别到了这三种记号时，Lex 将其以字符串指针的形式存储在 yylval 中以供 Yacc 使用。**特别地**，整数类型也以字符串存储，解析的工作交给后人进行。

a.l

```
{id} {
    yylval.str = strdup(yytext);
    return ID;
}
{integer} {
    yylval.str = strdup(yytext);
    return INTEGER;
}

{string} {
    yylval.str = strdup(yytext);
    return STRING;
}
```

4.2 解析标点符号

下面给出所用到的全部标点信息。

a.l

```
; {return FIN;}
\( {return LB;}
\) {return RB;}
, {return COMMA;}
\" {return QM;}
\< {return BELOW;}
= {return EQU;}
\* {return STAR;}
```

4.3 数据库系统的结构设计

4.3.1 物理结构

在本地磁盘上，数据库系统文件全部存放在`DBMS/`文件夹下。在 DBMS 下有一份元数据`.dbinfo`，用于说明该数据库系统有哪些数据库。其中记载了明文形式的所有数据库名字，以供系统访问对应的次级文件夹。而在某一个数据库下，以`Database1`为例，其所有的表单以及元数据全部存储在文件夹`Database1/`下。在该文件夹下首先有一份元数据`.tableinfo`，用于说明该数据库下有哪些表单。其中记载了明文形式的所有表单名，以供系统进一步访问表单文件。每一份表单文件的文件名即为表单名，在其内部，数据按照如下的字节顺序排列：

- 前 32 位为一个 32 位整数 n，表明该表单有多少个属性域。
- 接下来跟 n 块数据代表属性域的详细属性。每一块数据前 32 位为为一个 32 位整数 l_i，表示该属性域的名称长度，随后跟 l_i 字节的信息，代表该属性域的名称，再后跟 2 个 32 位整数，分别为该属性域的占用字节长度和属性类型。
- 然后 32 位为一个 32 位的整数 m, 表明该表单有多少个元组。
- 接下来跟 m 块数据代表表单的具体内容。每一块数据又分成 n 小块，这每一小块就代表了该元组在对应属性域上的分量。在每一小块中，首先给出 32 位的一个整数 k_i，代表该元组在该属性上拥有 k_i 字节的信息，随后跟出 k_i 字节的数据，表明其具体的取值。

尽管在上述具体阐述中，说明字节块的逻辑意义看起来可能较为费力，但是实际在对物理结构读写时只需要`循环 + 字节读写`即可，比较方便。

数据库系统的层次物理结构，如图1所示。

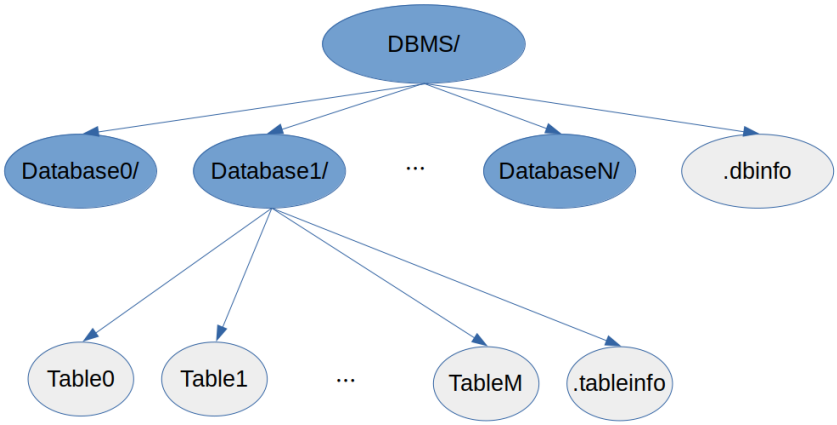


图 1: SQL 系统文件层次结构

4.3.2 逻辑结构

对应于4.3.1给出的物理结构，我们对该数据库系统的逻辑存储结构进行说明。

4.3.2.1 基本链表 本数据库系统使用自定义的链表结构，该链表的值域为**一个空指针与该空指针指向内容的有效字节长度**。有了这样两条信息，我们就可以使用该链表存储任何信息，十分方便。数据结构与函数接口如下所示。

下面给出链表的数据接口与函数接口信息。

list.h

```
struct ListElement{
    int bytes;
    void *val;
    ListElement *pre, *next;
};

struct List{
    ListElement *front, *back;
    int length;

    List();
    void push_back(void *val, int L);
    void erase(ListElement*);
    void copy(List *from); //将from链表拥有所有元素信息拷贝到当前链表
    void pop_front();
    void pop_back();
    void clear();
    void save(const char *file, int append); //向文件file写入字节流，append非0则以附加形式缀到文件尾
};
```

4.3.2.2 数据库系统结构

数据库系统的结构较为简单，是一个链表元素为数据库的链表。

数据库系统提供如下的全局函数接口：创建数据库、使用数据库、删除数据库、显示所有的数据库、从本地读所有数据库、将所有数据库写入本地。

数据结构与函数接口如下所示。

sql.h

```
extern List dbs;
void read();
void write();
int create_database(const char* databaseName, int bytes);
int use_database(const char* databaseName, int bytes);
int drop_database(const char* databaseName, int bytes);
void show_databases();
```

4.3.2.3 数据库结构

数据库结构中有该数据库的名称，还有该数据库下所有表单形成的链表。

对某个数据库而言，其提供如下的成员函数接口：查找某表单、显示所有表单、展示某表单的属性域、删除某表单、删除所有表单、从本地读数据库、将数据库写入本地。

数据结构与函数接口如下所示。

sql.h

```
struct DATABASE{
    char *name;
    List table;

    ListElement* find_table(const char* tableName, int bytes);
    void show_tables();
    int desc_table(const char *tableName, int bytes);
    int drop_table(const char *tableName, int bytes);
    void drop_all_table();

    void read();
    void write();
};
```

4.3.2.4 表单结构

表单结构中有该表单属于哪个数据库、该表单的名称、该表单拥有的属性数量、该表单的元组数量，以及所有属性构成的链表和所有元组构成的链表。

注意，在这里，一条元组本身就是一个链表，因而在该结构中，存储的元组链表其实是链表的链表。

对于某个表单而言，其提供如下的成员函数接口：检查元组信息合法性（在 where 子句中用到，检查 where 子句约束的属性是否都确实是存在的一个合法属性）、展示表单所有属性域、从本地读表单、将表单写入本地。

数据结构与函数接口如下所示。

sql.h

```
struct TABLE{
    char *belong_to_db;
    char *name;
    int fieldNumber, turpleNumber;
    List field, turpleEntry;

    List* check_all_turple();
    int check_turple(List *turple, ListElement *conElem);
    void desc();

    void read();
    void write();
};
```

4.3.2.5 属性域结构

`属性域`结构记录了某属性的名称、占用的字节数以及其类型。属性域的类型是我们提前宏定义的整数类型，不同的整数代表不同的数据类型，目前暂时只支持`INT`和`CHAR()`。

在属性域中有一个特别的值域`selected`，该属性仅在进行 select 操作时使用。当调用 select 语句时，首先检查所有的属性，如果 select 语句要求展示该属性则将`selected`设置为真，不然则设置为假，最终在展示时我们就可以根据该值域的真假决定是否输出。

数据结构与函数接口如下所示。

sql.h

```
struct FIELD{
    char *name;
    int bytes;
    int type;

    //仅仅在执行select操作时使用
    int selected;
};
```

4.3.2.6 条件结构

条件结构应用于 where 子句，用于记录某一个原子条件的信息。**注意**, 该结构体只能表示二元比较符，并且要求左操作数必须为数据属性，右操作数必须是属性值 (如 Age<20, Name < "ZhangSan")。其内部有左操作数的指针及字节长度、右操作数的指针及字节长度、比较符的类型，比较符的类型预先进行了宏定义，是一个整数类型。

数据结构如下所示。

sql.h

```
struct CONDITION{
    void *lval, *rval;
    int lbytes, rbytes;
    int cmp;
};
```

4.4 Yacc 的语法分析并与接口的交互

考虑到 Yacc 在对语法树归结时会相应的进行语义翻译，二者联系十分密切，所以这里我们将两部分放在一起进行说明，以期能有更好的效果。在逐一说明各个功能的 Yacc 语法分析与 C++ 接口使用方式之前，我们先给出 Yacc 用到的一些设置选项。

a.y

```
%union{
    char *str;
    int num;
    struct {
        char *p;
        int bytes;
    } M;
}

%token CREATE DATABASE DATABASES USE TABLE TABLES SHOW DESC INSERT INTO VALUES SELECT UPDATE DELETE
     DROP EXIT INT DOUBLE CHAR AND OR FROM WHERE SET
%token FIN LB RB COMMA BELOW EQU STAR QM
%token <str> ID STRING INTEGER
%type <str> create_database use_database desc_table drop_table drop_database value
%type <M> datatype
%type <num> create_table insert select

%left OR
%left AND
```

结合 Yacc 文件的部分相关代码，我们做如下说明：

• union 联合体有三种可能的情况：字符串指针、32 位整数、指针 + 字节数构成的结构体，在之后的程序中会根据不同的情况分配不同的数据类型。

• %token 给出了可能 Lex 可能用到的宏定义，自动生成相应的存放在 y.tab.h 中。

• 由于版本不同，在本实验环境中我们使用%type 关键字来为文法中所有的终结符、非终结符分配类型。

• 使用%left 指定左结合的优先级，解决多个逻辑表达式之间的组合问题。

4.4.1 总体语句框架

statements 由一条或多条的 statement 组成。而每一个 statement 则分别可能产生各种不同的非终结符，对应不同的功能；根据其返回值打印一定的提示信息。

a.y

```
statements:
    statement
    |statements statement
    ;

statement:
    create_database {
        if(create_database($1, strlen($1))) puts("create_database ok.");
        else puts("create_database fail.");
    }
    |use_database {
        if(use_database($1, strlen($1))) puts("use_database ok.");
        else puts("use_database fail.");
    }
    |create_table {
        if($1) puts("create_table ok.");
        else puts("create_table fail.");
    }
    |show_databases {
        show_databases();
    }
    |show_tables {
        if(current_database == NULL) puts("No database selected!");
        else current_database->show_tables();
    }
    |desc_table {
        if(current_database == NULL) puts("No database selected!");
        else current_database->desc_table($1, strlen($1));
    }
    |insert {
        if($1) puts("insert_into ok.");
```

```
32          else puts("insert_into fail.");
33      }
34      |delete {printf("Delete something ok\n");}
35      |update {printf("Update ok\n");}
36      |select {
37          if($1 == 1) puts("Select ok");
38          else puts("Select fail");
39      }
40      |drop_table {
41          if(current_database == NULL) puts("No database selected!");
42          else if(current_database->drop_table($1, strlen($1)) == 0) puts("drop_table fail.");
43          else puts("drop_table ok.");
44      }
45      |drop_database {
46          if(drop_database($1, strlen($1))) puts("drop_database ok.");
47          else puts("drop_database fail.");
48      }
49      |exit {return 0;}
50      ;
```

4.4.2 创建数据库

为了创建数据库，只需要将数据库的名称作为当前语句的内容向上传，statement 调用 create_database 进行创建。

create_database 函数接受一个字符串及其字节数，首先检查当前数据库系统下是否存在一个同名的数据库，如果不存在，那么先在文件系统中创建一个对应的数据库文件夹，并在文件夹内创建一个数据表元数据（初始为空），再更新内存中的信息，创建一个新的数据库元素，将对应的值初始化后插入到数据库系统链表尾。至此全部操作完成。

a.y

```
1  create_database:
2      CREATE DATABASE ID FIN {$$ = $3;}
```

4.4.3 使用数据库

使用数据库也类似地只需要将数据库的名称作为当前语句的内容向上传，statement 调用 use_database 进行选择。

ues_database 函数接受一个字符串及其字节数，检查该字符串对应的数据库是否存在，存在则操作成功。

a.y

```
use_database:
   USE ID FIN {$$ = $2;}
```

4.4.4 显示所有数据库

显示数据库不需要任何用户自行输入的参数，只要归结成功 statement 就调用 show_databases 展示所有的数据库。

show_databases 函数顺序遍历数据库系统链表中所有的数据表，并美观地将所有表名显示。

a.y

```
show_databases: SHOW DATABASES FIN
```

4.4.5 删除数据库

删除数据库也类似地将数据库名称向上传，statement 调用 statement 调用 drop_database 进行删除。

drop_database 函数接受一个字符串及其字节数，首先检查当前数据库系统下是否存在一个同名的数据库，如果存在，那么先在文件系统中删除对应的数据库文件夹，再更新内存中的信息，将该数据库对应的链表元素删去。至此全部操作完成。

a.y

```
drop_database:
   DROP DATABASE ID FIN {$$ = $3;}
```

4.4.6 创建表单

创建表单不仅需要表单名，还需要该表单各种属性的名称、类型、长度等信息，这些额外信息的数量未知，因而我们使用一个全局的**链表**来存储所有的这些信息。

归结进行时首先肯定先读取到表单名时，将该字符串加入到链表，成为链表的第一个元素。

接下来当归结到某一个属性域时，首先插入一个链表元素表示该属性的名称；接下来插入一个链表元素该元素的第一个字节表示该属性的类型，是一个提前约定好的整数，如果该类型是字符串数组，则该链表元素后面的所有字节则给出该数组的长度，以字符串形式给出，交由后人解析成整数。

按照 Yacc 归结的顺序，可以保证该链表中所有的信息均是保持原有顺序的。

当创建表单整条语句全部读取完成后，接口函数 temp_create_table 就对该全局链表进行解析，创建相应的表单，这里就不予赘述了。

注意！我们定义了一个全局的数据库指针 current_database 表明当前使用的数据库，在使用该创建表单功能前请先使用 use 指令选定数据库，不然该函数将运行失败并给出相应的报错。在本报告之后也有一些 SQL 功能需要提前使用 use 指令，之后便不再提及了，但是每一个之后对应的接口函数都对此做了错误处理，会产生相应的报错。

a.y

```
create_table:
    CREATE TABLE ID LB {
        temp.clear();
        temp.push_back($3, strlen($3));
    }
    fields
    RB FIN {
        if(temp_create_table() == 0) $$ = 0;
        else $$ = 1;
    }
fields:
    field
    |fields COMMA field
field:
    ID{
        temp.push_back($1, strlen($1));
    }
    datatype {
        temp.push_back($3.p, $3.bytes);
    }
datatype:
    INT {
        $$.bytes = 1;
        $$.p = (char*)malloc($$.bytes);
        $$.p[0] = 1;
    }
    |CHAR LB INTEGER RB {
        $$.bytes = 1 + strlen($3);
```

```
29          $$.p = (char*)malloc($$.bytes);
30          $$.p[0] = 2;
31          for(int i = 1; i < $$.bytes; i++) $$.p[i] = $3 [i - 1];
32      }
```

4.4.7 删除表单

删除表单跟之前的删除数据库很类似，上传表单名后 statement 使用`drop_table`指令。

a.y

```
1  drop_table:
2      DROP TABLE ID FIN {$$ = $3;}
```

4.4.8 显示所有表单

显示所有表单不需要额外的字符串参数，直接调用`show_tables`即可。

a.y

```
1  show_tables: SHOW TABLES FIN
```

4.4.9 展示表单所有属性

展示表单上传表单名称，调用`desc`成员函数即可。

a.y

```
1  desc_table: DESC ID FIN {$$ = $2;}
```

4.4.10 插入元组

在 insert 功能下，我们支持两种插入模式。一种是不提供属性名，但是必须按顺序给全所有属性值；另一种提供属性名，相应地给出对应的属性值。

`insert_values`对应解析出`VALUE (属性值 1，属性值 2，...)`的结构。

而 insert_method 则允许用户给出自定义的属性列，属性列之间用逗号分别，所有属性列用括号包围（哪怕是一个属性也需要括号）。

根据插入模式的不同，信息在全局链表中的组成方式也不同。

- 如果插入一条拥有全属性的元组，那么链表第一个元素记录了该表单的名称，用于函数索引出将要操作的表单；随后就给出所有属性域的内容以及长度直到结尾。接口函数会检查插入的属性值的数量是否与该表单拥有的属性属性匹配，但是暂时无法检测属性值的类型也匹配。

- 如果插入一条只拥有部分属性的元组，那么链表第一个元素同样记录了该表单的名称，用于函数索引出将要操作的表单；随后跟一个有效字节长度为 1、指针内容第一个字节为 0 的特征链表元素，表示即将开始属性名的插入。接下来跟着若干个链表元素，存储了将要插入元素拥有的属性名，以一个相同的特征 0 元素结束。当读取到第二个特征 0 元素后，就类似全属性元组结构一样，跟着所有对应的属性值直到文件结尾。

接口函数 temp_insert 解析该链表，并进行相应的插入工作，并会对一些常见错误进行汇报。当遇到一个不匹配的属性名后，该函数并不会报错，但也不会进行任何插入工作（此处没有与标准 DBMS 接轨，日后改进），那些没有初始值的属性值均被设置为空，展示时显示为 (null).

a.y

```
insert:
   INSERT INTO ID {
      temp.clear();
      temp.push_back($3, strlen($3));
   }
   insert_method
   FIN {
      if(temp_insert()) $$ = 1;
      else $$ = 0;
   }

insert_method:
   LB {
      char *p = (char*)malloc(1); p[0] = 0;
      temp.push_back(p, 1);
   }
   cols RB {
      char *p = (char*)malloc(1); p[0] = 0;
      temp.push_back(p, 1);
   }
   insert_values
   |insert_values
```

```
cols:
    ID {
        temp.push_back($1, strlen($1));
    }
    |cols COMMA ID {
        temp.push_back($3, strlen($3));
    }
insert_values:
    VALUES LB values RB
values:
    value {
        temp.push_back($1, strlen($1));
    }
    |values COMMA value {
        temp.push_back($3, strlen($3));
    }
value:
    INTEGER {
        $$ = $1;
    }
    |STRING {
        $$ = $1;
    }
```

4.4.11 查询元组

查询元组涉及到 where 条件子句，这里我们先对该类子句做一个说明。

对于任意涉及到 where 字句的其余 SQL 语句，where 都是可选的，因而在 Yacc 中 conditions 的解析也分为了有条件 with_conditions 和无条件。

那么有条件可以归约成若干条原子条件（即不包含逻辑与、逻辑或等，只有单独的一个逻辑表达式）的逻辑组合，正如 with_conditions 所示。

在本实验中，暂只支持 OR, AND 作为逻辑组合方式，<,= 作为逻辑比较符；尽管 >, <=, >= 等其余运算符都能较快的通过 < 来转换得到，但是在测试时仅采用了小于、等于这两个逻辑比较符，之后由于时间紧促，便没有来得及新加功能。

在对条件进行归约的时候，我们使用**后缀表达式**解决优先级问题。使用后缀表达式不会产生优先级问题，使用一个栈线性地扫描就可以无歧义地完成解析工作。可以看到，在归约时，归约到 value 时会向全局链表中加入数值/属性名，但归约到比较符时不进行任何操作，当整条逻辑比较语句全部读取完

成后才将比较符加入到链表中。通过这样的归约方式，我们就可以获得所有条件的后缀表达式。

因为字节长度的不同，所以接口函数可以通过分析字节长度来判断当前链表元素对应的是一个原子条件，还是一个逻辑组合符号。

接口函数根据该全局链表解析条件，遍历查询的表单下的所有元组，如果当前元组符合所有的条件，则将该元组加入到一个候选元组链表中，该候选元组链表用于随后的输出。

需要注意，尽管这套 DBMS 的实现十分朴素，许多地方的时间复杂度、空间复杂度都没有仔细优化，但是这里由于涉及到了元组的复制，我们还是给出了一个较好的方法，以免出现复杂度爆炸：候选元组链表中不照搬复制原有的元组内容，而是存原有元组的指针信息。由于候选元组链表存的只是一个指针，那么其空间消耗至多也就是一个指针的大小加一个 32 位的整数，不会因为元组数据内容的巨大而消耗过多的时间空间。

a.y

```
conditions:
    WHERE with_conditions
    |
with_conditions:
    with_conditions AND with_conditions {
        char *p = (char*)malloc(1); p[0] = 1;
        cons.push_back(p, 1);
    }
    |with_conditions OR with_conditions {
        char *p = (char*)malloc(1); p[0] = 2;
        cons.push_back(p, 1);
    }
    |LB with_conditions RB
    |ID BELOW value {
        CONDITION *now = new CONDITION;
        now->lval = $1; now->lbytes = strlen($1);
        now->rval = $3; now->rbytes = strlen($3);
        now->cmp = 1;
        cons.push_back(now, sizeof(CONDITION));
    }
    |ID EQU value {
        CONDITION *now = new CONDITION;
        now->lval = $1; now->lbytes = strlen($1);
        now->rval = $3; now->rbytes = strlen($3);
        now->cmp = 2;
        cons.push_back(now, sizeof(CONDITION));
    }
```

说明完了 where 子句对应的模块，剩余的部分就很简洁了。select 语句要么使用`*`要求显示所有的属性，要么给出需要的属性列表，以逗号分隔。本系统暂时只支持单表查询，不支持多表查询、子表查

询等操作。

在数据解析方面，由于有区分数据块归属的困难，我们再使用另一个全局的链表，与 where 子句使用的链表不同。这里使用的全局链表第一个元素给出表单的名称用于索引，随后给出所有希望查询的属性名；而 where 字句的链表专职处理逻辑语句的后缀表达式。

接口函数 temp_select 会检查属性名是否正确，并对一些常见错误进行汇报，随后将美观地展示出所有元组的信息。

a.y

```
select:
    SELECT{
        clear_cons();
        temp.clear();
    }
    select_what
    FROM ID{
        temp.push_back($5, strlen($5));
    }
    conditions FIN{
        if(temp_select() == 1) $$ = 1;
        else $$ = 0;
    }
ids:
    ID {
        temp.push_back($1, strlen($1));
    }
    |ids COMMA ID {
        temp.push_back($3, strlen($3));
    }
select_what:
    STAR
    |ids
```

4.4.12 退出系统

读取到退出指令，系统先妥善地将内存数据库存储到本地，再安全地退出。

a.y

```
exit:
    EXIT
```

4.4.13 删除元组与更新元组

由于时间紧张等原因，delete 和 update 操作并没有完成语义翻译，该系统暂时只能用 Yacc 解析出两种语句的结构，不能执行实际的操作，十分遗憾！

a.y delete

```
delete:
    DELETE FROM ID conditions FIN;
```

a.y update

```
update:
    UPDATE ID
    SET set_values
    conditions FIN;
set_values:
    set_value
    |set_values COMMA set_value
set_value:
    ID EQU value
```

5 结果展示

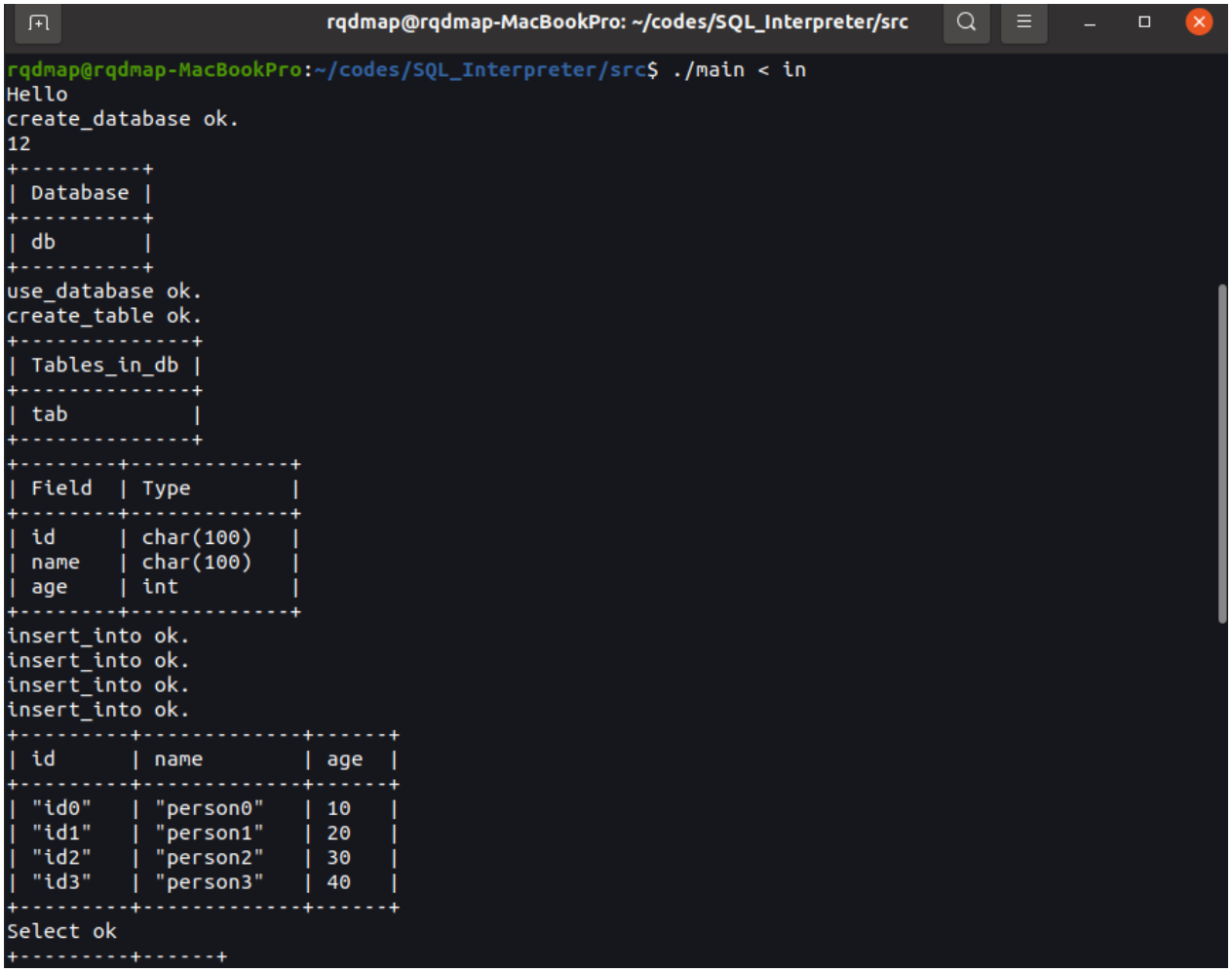
使用如下的测试数据作为标准输入。

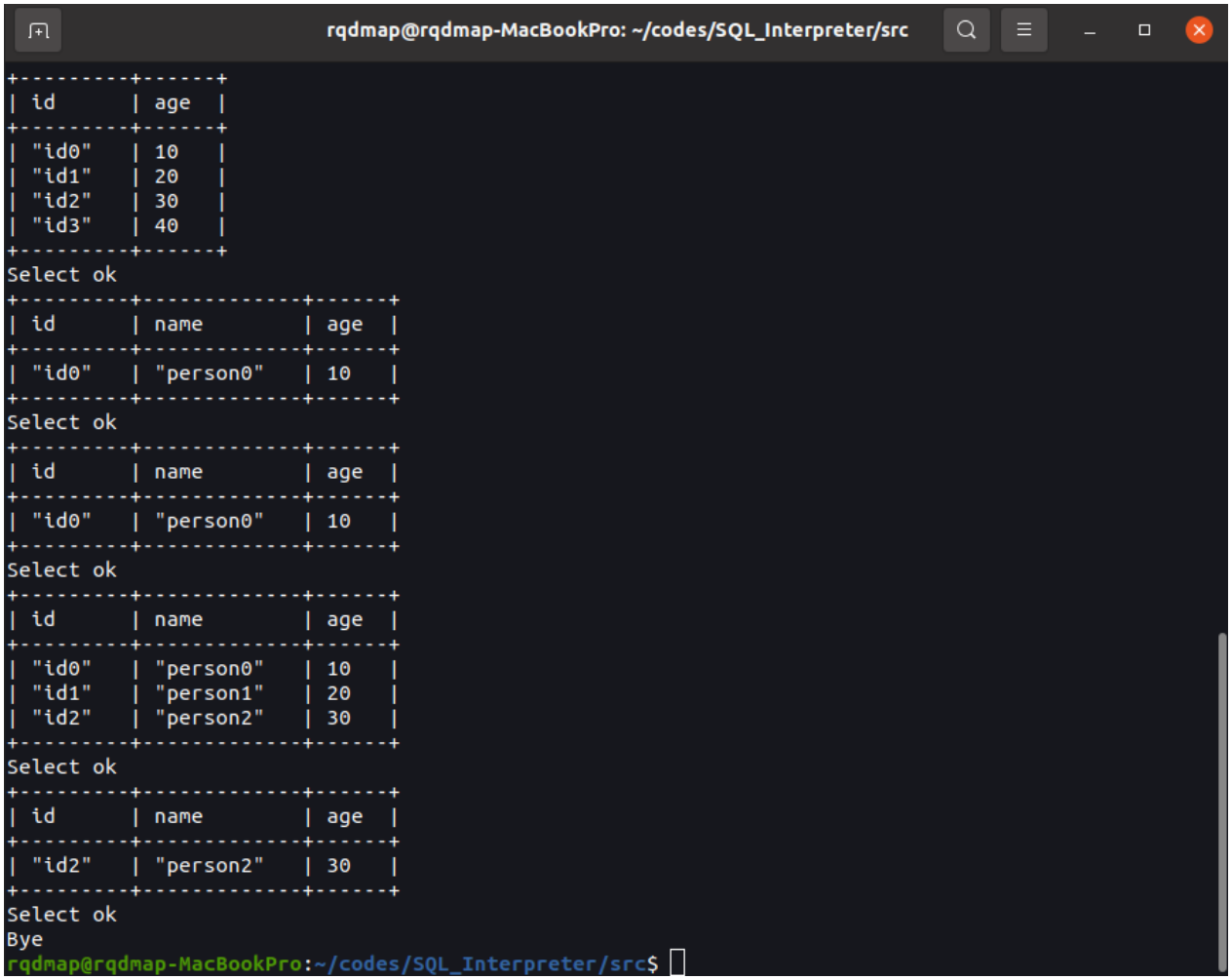
in

```
create database db;
show databases;
use db;

create table tab(
    id char(100),
    name char(100),
    age int
);

show tables;
desc tab;

insert into tab values ("id0", "person0", 10);
insert into tab values ("id1", "person1", 20);
insert into tab values ("id2", "person2", 30);
insert into tab values ("id3", "person3", 40);

select * from tab;
```

```
select id, age from tab;

select * from tab where age < 20;
select * from tab where (age < 20);
select * from tab where age < 20 OR id < "id3";
select * from tab where (age < 20 OR id < "id3") AND name = "person2";

update tab set id = "idx";
update tab set id = "idx" where id < "id2";

exit
```

测试结果如下所示，该系统能正确地执行上述语句。





6 局限与展望

- 该系统功能尚十分不完善，没有实现更新与删除操作，支持的数据类型、操作符、比较符不全面，不支持多表查询、子表查询，不支持主码、外码，不支持 check 语句，不支持数值表达式运算等。
- 该系统某些实现并没有与标准 DBMS 匹配，不能执行所有的标准 SQL 语句。
- 支持的条件类型十分单薄，不支持左值和右值的任意选择，可通过在条件结构体中添加数据类型进一步进行划分。
- 涉及到指定属性的函数操作并没有对该属性对应的数据类型进行合法性检查，默认用户遵循规范。
- 时空复杂度仍可以大力优化，并对代码和程序结构进一步美化。

7 个人体会

本次实验我受益匪浅，熟悉了 Lex/Yacc 分析工具的使用，扎实了 C++ 编程的功底，也提高了 Latex 书写文档的能力。半年来这是第三份千行代码以上的程序项目，从第一份 RQD-FTP 开始甚至不会写 C++ 多文件编译、乱在 h 文件中写实现，到第二份二进制表示的高精度整数类，再到现在这份垃圾的 DBMS 全套系统，果然大工程使人提高，在神秘的方面提高了退役 ACM 选手的代码能力。正如小兵老师所说，这次实验项目令人难忘，自己实现 DBMS 十分有趣，但是实现出来的东西却差强人意。尽管有一千行左右但也就是我 2 天日夜肝出来的结果，日后如果有时间还希望进一步优化，能真正媲美一个标准的 DBMS。~~当然，时空复杂度方面就不妄想与之比肩了，功能完备即心满意足了~~

8 程序清单

程序名	描述
MakeFile	
rqdmap.h	个人编程惯用的一些宏定义
list.h, list.cpp	自定义链表结构接口及定义
a.l	Lex 源文件
lex.yy.c	Lex 编译结果
a.y	Yacc 文件
y.tab.h, y.tab.c	Yacc 编译结果
sql.h, sql.cpp	SQL 函数接口及定义
main	可执行程序，数据库系统
in	测试数据
init	bash 脚本，初始化数据库
DBMS	数据库本地文件