How minirel works?
+ dbcreate foobar
+ minirel
+ dbdestroy

minirel: runs a loop
+ show prompt
+ get a user command
+ call parse() to parse it into an internal format
+ call interp() to understand what the query wants to do, then call the appropriate backend procedure
+ show results

commands
+ ddl statement: create table, destroy table, load table, print table, help, quit
+ dml statement: query or update
- query: select ... from ... where ... (can do selection or join)
- update: delete from table where ..., insert into table ...

You should try to look at minirel.C, parser/parse.y, interp.C to get a sense on how the code works and how backend procedures are called.

qu.1
---
```
/*
 * test 1 tests QU_Select without indices
 */


/* create relations */
create table soaps(soapid int, name char(28), network char(4), rating real);
load table soaps from ("../data/soaps.data");

create table stars(starid int, real_name char(20), plays char(12), soapid int);
load table stars from ("../data/stars.data");

/*
 * various selections without indices
 */

/* simple selection (should be the same as just printing the relation) */
select soapid, name, network, rating from soaps;
print table soaps;

/* names, ratings, and networks of soaps on NBC */
select name, rating, network from soaps where network = "NBC";

/* print character name, real name, and ids of stars with id's < 12 */
select plays, real_name, starid from stars where starid < 12;

/* ratings, networks, and names of soaps with ratings of 5 or greater */
select rating, network, name from soaps where rating >= 5.0;

/* selection that doesn't find anything */
select real_name, starid from stars where starid > 567;
```

```
/* select into a non-existent relation */
select network, soapid, name into ned
from soaps
where network = "CBS";
print table ned;

/* select into a existing relation */
select network, soapid, name into ned
from soaps
where network = "NBC";
print table ned;

/* select into a relation with strings of different sizes */
create table ted (plays char(12), soapid int);
select plays, soapid into ted from stars where plays < "L";
print table ted;
```

---

Stage 5: dbcreate, dbdestroy, backend procedures to support ddl commands
Stage 6: backend procedures to support dml commands (selection/projection, insertion, deletion, NO JOIN)

To do Stage 6, should really understand two tables that form the catalog: relcat and attrcat (see Stage 5)
+ relcat: one tuple for each relation (including relcat): relName, attrCnt
+ attrcat: one tuple for each attribute of every relation: relName, attrName, attrType, attrLen, *attrValue

select.C

---

```
const Status QU_Select(const string & result, ==> table to store the output in
                   const int projCnt,
                   const attrInfo projNames[],
                   const attrInfo *attr, ==> this and below are the selection condition
                   const Operator op,
                   const char *attrValue)
{
   // Qu_Select sets up things and then calls ScanSelect to do the actual work
    cout << "Doing QU_Select " << endl;

}


const Status ScanSelect(const string & result, ==> table to store output
                  const int projCnt,
                  const AttrDesc projNames[],
                  const AttrDesc *attrDesc, ==> attr for selection
                  const Operator op,
                  const char *filter, ==> *attrValue
                  const int reclen) ==> length of output tuple
{
    cout << "Doing HeapFileScan Selection using ScanSelect()" << endl;


}
```

---

```
// define attrInfo
typedef struct {
      char relName[MAXNAME]; // relation name
      char attrName[MAXNAME]; // attribute name
      int attrType; // INTEGER, FLOAT, or STRING
      int attrLen; // length of attribute in bytes
      void *attrValue; // ptr to binary value
} attrInfo;

// schema of tuples in the attribute catalog:
typedef struct {
char relName[MAXNAME]; // relation name
char attrName[MAXNAME]; // attribute name
int attrOffset; // attribute offset
int attrType; // attribute type
int attrLen; // attribute length
} AttrDesc;
```

QU_Select
+ Make sure to give ScanSelect the proper input
+ To go from attrInfo to attrDesc, need to consult the catalog (attrCat and relCat, global variables)

ScanSelect
+ have a temporary record for output table
+ open "result" as an InsertFileScan object
+ open current table (to be scanned) as a HeapFileScan object
+ check if an unconditional scan is required
+ check attrType: INTEGER, FLOAT, STRING
+ scan the current table
+ if find a record, then copy stuff over to the temporary record (memcpy)
+ insert into the output table

==========

```
/*
 * Deletes records from a specified relation.
 *
 * Returns:
 *    OK on success
 *    an error code otherwise
 */

const Status QU_Delete(const string & relation,
                  const string & attrName,
                  const Operator op,
                  const Datatype type,
                  const char *attrValue)
{
// part 6
return OK;
```

```
}

+ make sure to handle cases such as when an input argument is NULL
+ scan depends on attr type: INTEGER, etc.

================================

/*
 * Inserts a record into the specified relation.
 *
 * Returns:
 *    OK on success
 *    an error code otherwise
 */

const Status QU_Insert(const string & relation,
      const int attrCnt,
      const attrInfo attrList[])
{
// part 6
return OK;

}
```

Insert a tuple with the given attribute values (in *attrList*) in relation. The value of the attribute is supplied in the *attrValue* member of the attrInfo structure. Since the order of the attributes in attrList[] may not be the same as in the relation, you might have to rearrange them before insertion. If no value is specified for an attribute, you should reject the insertion as Minirel does not implement NULLs.