**Advanced Programming Final Term Paper**

No. : 533772

Name : Lanpei Li

**Exercise 1:** *For each Annotation I defined a class (Table,Column, Id,Relation,Many2One,One2Many) inheriting from the base class Annotation (Many2One,One2Many inherit from class Relation):*

class Annotation { public string Name { get; set; } }

class Column:Annotation { public int Length { get; set; } }

class Id:Annotation { }

class Many2One:Relation { }

class One2Many:Relation { public string MappedBy { get; set; } }

class Relation:Annotation { public string Target { get; set; } }

class Table:Annotation { public Id id { get; set; } public List<Column> columns { get; set; }

public Relation relation { get; set; } public string type { get; set; } }

**Exercise 2 :** *First, I tokenized the strings from a text file into tokens, then parse the tokens and return a list of Table object which has been bulid in exercise1:*

class AnnotationTokenizer{

private StringReader \_reader;

private string text;

public AnnotationTokenizer(string text){ this.text = text; \_reader = new StringReader(text); }

public IEnumerable<Token> Tokenize(){

var tokens = new List<Token>();

while (\_reader.Peek() != -1){

while (Char.IsWhiteSpace((char)\_reader.Peek())) \_reader.Read();

if (\_reader.Peek() == -1) break;

if (Char.IsLetter((char)\_reader.Peek())){

var token = ParseKeyword(); tokens.Add(token);

}else if (Char.IsDigit(c)) { var nr = ParseNumber(); tokens.Add(new IntToken(nr));

}else if (c == '@') {

tokens.Add(new SymbolToken { symbol = SymbolEnum.at, str = c.ToString() }); \_reader.Read();

}else if (c == ',') {

tokens.Add(new SymbolToken { symbol = SymbolEnum.comma, str = c.ToString() }); \_reader.Read();

}else if (c == ';'){

tokens.Add(new SymbolToken { symbol = SymbolEnum.semicolon, str = c.ToString() }); \_reader.Read();

}else if (c == '=') {

tokens.Add(new SymbolToken { symbol = SymbolEnum.equal, str = c.ToString() }); \_reader.Read();

}else if (c == '(') {

tokens.Add(new SymbolToken { symbol = SymbolEnum.leftparenthesis, str = c.ToString() });

\_reader.Read();

}else if (c == ')'){

tokens.Add(new SymbolToken { symbol = SymbolEnum.rightparenthesis, str = c.ToString() });

\_reader.Read();

}else if (c == '{') {

tokens.Add(new SymbolToken { symbol = SymbolEnum.leftbrace, str = c.ToString() }); \_reader.Read();

} else if (c == '}') {

tokens.Add(new SymbolToken { symbol = SymbolEnum.rightbrace, str = c.ToString() }); \_reader.Read();

}else if (c == '<'){

tokens.Add(new SymbolToken { symbol = SymbolEnum.leftanglebracket, str = c.ToString() });

\_reader.Read();

}else if (c == '>') {

tokens.Add(new SymbolToken { symbol = SymbolEnum.rightanglebracket, str = c.ToString() });

\_reader.Read();

} else if (c == '"') {

tokens.Add(new SymbolToken { symbol = SymbolEnum.doublequotation, str = c.ToString() });

\_reader.Read();

}else throw new Exception("Unknown character in text: " + c);

} return tokens; }

private int ParseNumber(){

var digits = new List<int>();

while (Char.IsDigit((char)\_reader.Peek())){

var digit = (char)\_reader.Read(); int i;

if (int.TryParse(Char.ToString(digit), out i)) digits.Add(i);

else throw new Exception("Could not parse integer number when parsing digit: " + digit);

} var nr = 0; var mul = 1; digits.Reverse(); digits.ForEach(d => { nr += d \* mul; mul \*= 10; }); return nr; }

private Token ParseKeyword(){

var text = new StringBuilder(); bool IsVar = false;

while (Char.IsLetter((char)\_reader.Peek()) || Char.IsDigit((char)\_reader.Peek())){

text.Append((char)\_reader.Read());

if (Char.IsDigit((char)\_reader.Peek())) IsVar = true;

else IsVar = false;

}

var potentialKeyword = text.ToString();

if (!IsVar){ var kw = new KeywordToken { str = potentialKeyword };

switch (potentialKeyword){

case "public": kw.keyword = KeywordEnum.\_public; break;

case "interface": kw.keyword = KeywordEnum.\_interface; break;

case "Table": kw.keyword = KeywordEnum.Table; break;

case "Id": kw.keyword = KeywordEnum.Id; break;

case "Column": kw.keyword = KeywordEnum.Column; break;

case "Many2One": kw.keyword = KeywordEnum.Many2One; break;

case "One2Many": kw.keyword = KeywordEnum.One2Many; break;

case "name": kw.keyword = KeywordEnum.name; break;

case "length": kw.keyword = KeywordEnum.length; break;

case "target": kw.keyword = KeywordEnum.target; break;

case "mappedBy": kw.keyword = KeywordEnum.mappedBy; break;

case "Integer": kw.keyword = KeywordEnum.Integer; break;

case "String": kw.keyword = KeywordEnum.String; break;

case "List": kw.keyword = KeywordEnum.List; break;

default: return new CharsToken { str = text.ToString() };

} return kw;

}else{ if (char.IsUpper(text[0])) return new IdentifierToken { str = text.ToString() };

else return new VariableToken { str = text.ToString() };}

}

}

class AnnotationParse{

private readonly IEnumerator<Token> \_tokens;

private List<Table> tbs;

public AnnotationParse(IEnumerable<Token> tokens) { \_tokens = tokens.GetEnumerator();

tbs = new List<Table>();}

public List<Table> Parse() {

while (\_tokens.MoveNext()) {

var t = ParseAnnotation(); \_tokens.MoveNext();

ParseKeyword(KeywordEnum.\_public); \_tokens.MoveNext();

ParseKeyword(KeywordEnum.\_interface); \_tokens.MoveNext();

var idf = ParseChars<StringToken>();

if (!(t is Table)) throw new Exception("Syntax error with unexpect Object Type");

((Table)t).type = idf; \_tokens.MoveNext();

ParseBody((Table)t);

} return tbs; }

private void ParseBody(Table t){

ParseSymbol(SymbolEnum.leftbrace); \_tokens.MoveNext();

do { var anno = ParseAnnotation();

if (anno is Id) t.id = (Id)anno;

else if (anno is Column)

t.columns = new List<Column> { (Column)anno };

else if (anno is Many2One) t.relation = (Many2One)anno;

else if (anno is One2Many) t.relation = (One2Many)anno;

else throw new Exception(" syntax error with unexpect object type");

\_tokens.MoveNext(); ParseDefinition(anno); \_tokens.MoveNext();

} while ((\_tokens.Current is SymbolToken) && (((SymbolToken)\_tokens.Current).symbol == SymbolEnum.at));

if ((\_tokens.Current is SymbolToken) && ((SymbolToken)\_tokens.Current).symbol == SymbolEnum.rightbrace){

ParseSymbol(SymbolEnum.rightbrace); return;

} throw new Exception("Syntax error with unexpect token :" + \_tokens.Current.str);}

private void ParseDefinition(Annotation annobody) {

if (\_tokens.Current is KeywordToken) {

if (((KeywordToken)\_tokens.Current).keyword == KeywordEnum.Integer){

ParseKeyword(KeywordEnum.Integer);

\_tokens.MoveNext(); string variable = ParseChars<StringToken>();

\_tokens.MoveNext(); ParseSymbol(SymbolEnum.semicolon);

}else if (((KeywordToken)\_tokens.Current).keyword == KeywordEnum.String){

ParseKeyword(KeywordEnum.String); \_tokens.MoveNext();

string variable = ParseChars<StringToken>(); \_tokens.MoveNext();

ParseSymbol(SymbolEnum.semicolon);

}else if (((KeywordToken)\_tokens.Current).keyword == KeywordEnum.List) {

ParseComplexType(); \_tokens.MoveNext();

string variable = ParseChars<StringToken>(); \_tokens.MoveNext();

ParseSymbol(SymbolEnum.semicolon);

} else throw new Exception("Syntax error with unexpect Keyword :" + \_tokens.Current.str);

}else if (\_tokens.Current is StringToken) {

var idf = ParseChars<StringToken>(); \_tokens.MoveNext();

string variable = ParseChars<StringToken>(); \_tokens.MoveNext(); ParseSymbol(SymbolEnum.semicolon);

}

}

private void ParseComplexType(){

ParseKeyword(KeywordEnum.List); \_tokens.MoveNext();

if (\_tokens.Current is SymbolToken){

if (((SymbolToken)\_tokens.Current).symbol == SymbolEnum.leftanglebracket){

ParseSymbol(SymbolEnum.leftanglebracket); \_tokens.MoveNext();

}

var idf = ParseChars<StringToken>(); \_tokens.MoveNext(); ParseSymbol(SymbolEnum.rightanglebracket);

} else throw new Exception("Syntax error with unexpect token :" + \_tokens.Current.str);

}

public string ParseChars<T>(){

if (\_tokens.Current is T) return \_tokens.Current.str;

throw new Exception("Syntax error with unexpect string :" + \_tokens.Current.str);

}

public void ParseKeyword(KeywordEnum keyword){

if (((KeywordToken)\_tokens.Current).keyword == keyword) return;

throw new Exception("Syntax error with unexpect keyword :" + \_tokens.Current.str);

}

public Annotation ParseAnnotation(){

ParseSymbol(SymbolEnum.at); \_tokens.MoveNext();

var anno = ParseAnnotator(); \_tokens.MoveNext();

ParseSymbol(SymbolEnum.leftparenthesis); \_tokens.MoveNext();

ParsePair(anno); \_tokens.MoveNext();

if (\_tokens.Current is SymbolToken){

while (((SymbolToken)\_tokens.Current).symbol == SymbolEnum.comma){

ParseSymbol(SymbolEnum.comma); \_tokens.MoveNext();

ParsePair(anno); \_tokens.MoveNext();

}

if (((SymbolToken)\_tokens.Current).symbol == SymbolEnum.rightparenthesis) {

ParseSymbol(SymbolEnum.rightparenthesis);

return anno;

} throw new Exception("Syntax error with unexpect symbol :" + \_tokens.Current.str);

}else throw new Exception("Syntax error with unexpect keyword :" + \_tokens.Current.str);

}

private void ParsePair(Annotation anno) {

if (\_tokens.Current is KeywordToken){

if (((KeywordToken)\_tokens.Current).keyword == KeywordEnum.name) {

ParseKeyword(KeywordEnum.name); \_tokens.MoveNext();

ParseSymbol(SymbolEnum.equal); \_tokens.MoveNext();

ParseSymbol(SymbolEnum.doublequotation); \_tokens.MoveNext();

anno.Name = ParseChars<StringToken>(); \_tokens.MoveNext();

ParseSymbol(SymbolEnum.doublequotation);

} else if (((KeywordToken)\_tokens.Current).keyword == KeywordEnum.length) {

ParseKeyword(KeywordEnum.length); \_tokens.MoveNext();

ParseSymbol(SymbolEnum.equal); \_tokens.MoveNext();

ParseSymbol(SymbolEnum.doublequotation); \_tokens.MoveNext();

if (anno is Column) ((Column)anno).Length = ParseNumber();

else throw new Exception(" syntax error with unexpect object type");

\_tokens.MoveNext(); ParseSymbol(SymbolEnum.doublequotation);

}else if (((KeywordToken)\_tokens.Current).keyword == KeywordEnum.target){

ParseKeyword(KeywordEnum.target); \_tokens.MoveNext();

ParseSymbol(SymbolEnum.equal); \_tokens.MoveNext();

ParseSymbol(SymbolEnum.doublequotation); \_tokens.MoveNext();

var varible = ParseChars<StringToken>();

if (anno is Many2One) ((Many2One)anno).Target = varible;

else if (anno is One2Many) ((One2Many)anno).Target = varible;

else throw new Exception(" syntax error with unexpect object type");

\_tokens.MoveNext(); ParseSymbol(SymbolEnum.doublequotation);

}else if (((KeywordToken)\_tokens.Current).keyword == KeywordEnum.mappedBy){

ParseKeyword(KeywordEnum.mappedBy); \_tokens.MoveNext();

ParseSymbol(SymbolEnum.equal); \_tokens.MoveNext();

ParseSymbol(SymbolEnum.doublequotation); \_tokens.MoveNext();

var varible = ParseChars<StringToken>();

if (anno is One2Many) ((One2Many)anno).MappedBy = varible;

else throw new Exception(" syntax error with unexpect object type");

\_tokens.MoveNext(); ParseSymbol(SymbolEnum.doublequotation);

}else throw new Exception("Syntax error with unexpect keyword :" + \_tokens.Current.str);

}else throw new Exception("Syntax error with unexpect string :" + \_tokens.Current.str);

}

private int ParseNumber() { return(\_tokens.Current as IntToken).Value; }

private Annotation ParseAnnotator(){

if (\_tokens.Current is KeywordToken) {

if (((KeywordToken)\_tokens.Current).keyword == KeywordEnum.Table) {

ParseKeyword(KeywordEnum.Table);

var tb = new Table { }; tbs.Add(tb); return tb;

}else if (((KeywordToken)\_tokens.Current).keyword == KeywordEnum.Id){

ParseKeyword(KeywordEnum.Id); return new Id { };

} else if (((KeywordToken)\_tokens.Current).keyword == KeywordEnum.Column){

ParseKeyword(KeywordEnum.Column); return new Column { };

}else if (((KeywordToken)\_tokens.Current).keyword == KeywordEnum.Many2One){

ParseKeyword(KeywordEnum.Many2One); return new Many2One { };

}else if (((KeywordToken)\_tokens.Current).keyword == KeywordEnum.One2Many){

ParseKeyword(KeywordEnum.One2Many); sreturn new One2Many { };

}else throw new Exception("Syntax error with unexpect keyword :" + \_tokens.Current.str);

}else throw new Exception("Syntax error with unexpect string :" + \_tokens.Current.str);

}

public void ParseSymbol(SymbolEnum symbol){

if (\_tokens.Current is SymbolToken) {

if (((SymbolToken)\_tokens.Current).symbol == symbol) return;

throw new Exception("Syntax error with unexpect symbol :" + \_tokens.Current.str);

} throw new Exception("Syntax error with unexpect token :" + \_tokens.Current.str);

}

}

class CharsToken:StringToken { }

class IdentifierToken:StringToken { }

class IntToken:Token { private readonly int \_value;

public IntToken(int value) { \_value = value; }

public int Value { get { return \_value; } } }

class KeywordToken:StringToken { public KeywordEnum keyword { get; set; } }

class StringToken: Token { }

class SymbolToken:Token { public SymbolEnum symbol { get; set; } }

class Token { public string str { get; set; } }

class VariableToken:StringToken {}

class TestParser { public List<Table> Test() {

string text = System.IO.File.ReadAllText(@"test.txt");

var tokens = new AnnotationTokenizer(text).Tokenize();

var parser = new AnnotationParse(tokens);

return parser.Parse(); } }

enum KeywordEnum{ \_public, \_interface, Table, Id, Column, Many2One, One2Many, name, length, target, mappedBy, Integer, String, List, }

enum SymbolEnum { at, comma, semicolon, equal, leftparenthesis, rightparenthesis, leftbrace, rightbrace, leftanglebracket, rightanglebracket, doublequotation, }

**Exercise 3 :** *Classes CsharpCodeGenerator and SQLCodeGenerator (which are inherited from CodeGenerator ) generate the specified language code from the result of exercise 2 (a list of object) and stored the new codes to the specified file.*

class CodeGenerator{

public IEnumerator<Table> \_tables;

public IEnumerable<Table> tables;

public string output;

public virtual string path { get; set; }

public CodeGenerator() { }

public string GenerateCode(){

while (\_tables.MoveNext()) output = string.Format("{0} {1} \n", output, Generate());

File.WriteAllText(path, output); return output;

}

public string Generate(){ return string.Format("{0} {1}", GenerateTitle(), GenerateBody()); }

public virtual string GenerateBody() { return ""; }

public virtual string GenerateTitle() { return ""; }

public virtual string GenerateId() { return ""; }

public virtual string GenerateColumn() { return ""; }

public virtual string GenerateRelation() { return ""; }

}

class CsharpCodeGenerator : CodeGenerator{

public override string path { get; set; }

public CsharpCodeGenerator(IEnumerable<Table> tb){

\_tables = tb.GetEnumerator(); tables = tb; path = @"CsharpCode.txt"; }

public override string GenerateBody(){

return string.Format("{{ \n {0} \n {1} {2} \n }}", GenerateId(), GenerateColumn(), GenerateRelation()); }

public override string GenerateRelation(){

if (\_tables.Current.relation is Many2One){

return string.Format(" public {0} {1};", \_tables.Current.relation.Target, \_tables.Current.relation.Name);

}else if (\_tables.Current.relation is One2Many) {

return string.Format(" public List<{0}> {1};", \_tables.Current.relation.Target, \_tables.Current.relation.Name);

} else throw new Exception(" Invaild relation Type"); }

public override string GenerateColumn(){

int i = \_tables.Current.columns.Count(); string cols = "";

while (i > 0) { i--; cols = string.Format(" {0} public string {1};\n", cols, \_tables.Current.columns[i].Name); }

return cols; }

public override string GenerateId(){ return string.Format(" public int {0};", \_tables.Current.id.Name); }

public override string GenerateTitle(){ return string.Format("class {0}", \_tables.Current.type); }

}

class SQLCodeGenerator : CodeGenerator{

public override string path { get; set; }

public SQLCodeGenerator(IEnumerable<Table> tb){

\_tables = tb.GetEnumerator(); tables = new List<Table>(tb); path = @"SQLCode.txt"; }

public override string GenerateBody(){

return string.Format("( \n {0} \n {1} {2} );", GenerateId(), GenerateColumn(), GenerateRelation()); }

public override string GenerateRelation(){

if (\_tables.Current.relation is One2Many)

return string.Format(" HasBook VARCHAR(80), \n PRIMARY KEY({0}) \n", \_tables.Current.id.Name);

else if (\_tables.Current.relation is Many2One) {

var ele = Array.FindAll(tables.ToArray(), x => x.type == \_tables.Current.relation.Target).ToList();

var r = ele.GetEnumerator(); string rel = "";

string pr = string.Format(" PRIMARY KEY({0}) \n", \_tables.Current.id.Name);

while (r.MoveNext())

rel = string.Format(" {0} Fk\_Id INT, \n FOREIGN KEY(Fk\_Id) REFERENCES {1}({2}), \n", rel, r.Current.Name, r.Current.id.Name);

return string.Format("{0} {1}", rel, pr);

} else throw new Exception("Invalid relation type");

}

public override string GenerateColumn() {

int i = \_tables.Current.columns.Count(); string cols = "";

while (i > 0){ i--;

cols = string.Format(" {0} {1} VARCHAR({2}),\n", cols, \_tables.Current.columns[i].Name, \_tables.Current.columns[i].Length);

} return cols;

}

public override string GenerateId(){ return string.Format(" {0} INT,", \_tables.Current.id.Name); }

public override string GenerateTitle() { return string.Format("CREATE TABLE {0}", \_tables.Current.Name); }

}

**Exercise 4 :** *I serialized the object to the suitable SQL query ,then used SQLite NuGet Package to execute the commands , get the results , deserialize to the corresponding object.*

class EntityManager<T> : IEntityManager<T> {

public T find(object primaryKey){

using (SQLiteConnection sqlite = new SQLiteConnection("Data Source=annotation.sqlite")){

sqlite.Open(); string tableName = ""; string selectQuery = "";

if (typeof(T).Name == "Book"){

tableName = "Book";

selectQuery = string.Format("Select \* From {0} \n Where id = {1};", tableName, primaryKey.ToString());

}else if (typeof(T).Name == "Publisher"){

tableName = "Publisher";

selectQuery = string.Format("Select \* From {0} \n Where id = {1};", tableName, primaryKey.ToString());

}else throw new Exception(" T type error");

SQLiteCommand selectCommand = new SQLiteCommand(selectQuery, sqlite);

SQLiteDataReader r = selectCommand.ExecuteReader();

T res = (T)Activator.CreateInstance(typeof(T));

while (r.Read()){

FieldInfo[] fields = typeof(T).GetFields();

fields[0].SetValue(res, r.GetInt32(0));

if (typeof(T).Name == "Book") {

fields[1].SetValue(res, r.GetString(1));

var ep = new EntityManager<Publisher>();

fields[2].SetValue(res, ep.find(r.GetInt32(2)));

}else if (typeof(T).Name == "Publisher"){

fields[1].SetValue(res, r.GetString(1));

if (r.GetString(2) == "true") {

var eb = new EntityManager<Book>();

List<Book> lb = new List<Book>();

lb = eb.findList(new Key("Fk\_Id", r.GetInt32(0).ToString()));

fields[2].SetValue(res, lb);

}

}else throw new Exception(" T type error");

} sqlite.Close(); return (T)res; }

}

private List<T> findList(Key keyObject){

using (SQLiteConnection sqlite = new SQLiteConnection("Data Source=annotation.sqlite")) {

sqlite.Open(); string tableName = ""; string selectQuery = "";

if (typeof(T).Name == "Book"){

tableName = "Book";

selectQuery = string.Format("Select \* From {0} \n Where {1} = {2};", tableName, keyObject.name, keyObject.value);

SQLiteCommand selectCommand = new SQLiteCommand(selectQuery, sqlite);

SQLiteDataReader r = selectCommand.ExecuteReader();

List<T> ListObj = new List<T>();

while (r.Read()){

T res = (T)Activator.CreateInstance(typeof(T)); ListObj.Add(res); FieldInfo[] fields = typeof(T).GetFields();

fields[0].SetValue(res, r.GetInt32(0)); fields[1].SetValue(res, r.GetString(1));

}sqlite.Close(); return (List<T>)ListObj;

}else throw new Exception(" T type error");

}

}

public void persist(T entity){

using (SQLiteConnection sqlite = new SQLiteConnection("Data Source=annotation.sqlite")){

sqlite.Open(); string tableName = ""; string insertQuery = "";

FieldInfo[] fields = typeof(T).GetFields(); string values = fields[0].GetValue(entity).ToString();

if (entity is Book){

tableName = "Book";

values = string.Format("{0},'{1}'", values, fields[1].GetValue(entity).ToString());

values = string.Format("{0},'{1}'", values, ((Publisher)fields[fields.Length - 1].GetValue(entity)).id);

insertQuery = string.Format("Insert Into {0} \n Values({1});", tableName, values);

}else if (entity is Publisher){

tableName = "Publisher";

values = string.Format("{0},'{1}'", values, fields[1].GetValue(entity).ToString());

if ((bool)fields[fields.Length-1].GetValue(entity)) values = string.Format("{0},'true'", values);

else values = string.Format("{0},'false'", values);

insertQuery = string.Format("Insert Into {0} \n Values({1});", tableName, values);

}else throw new Exception(" entity type error");

SQLiteCommand insertCommand = new SQLiteCommand(insertQuery, sqlite);

insertCommand.ExecuteNonQuery(); sqlite.Close();

}

}

public void remove(T entity){

using (SQLiteConnection sqlite = new SQLiteConnection("Data Source=annotation.sqlite")){

sqlite.Open(); string tableName = ""; string deleteQuery = "";

FieldInfo[] fields = typeof(T).GetFields();

string values = fields[0].GetValue(entity).ToString();

if (entity is Book) { tableName = "Book";

deleteQuery = string.Format("Delete From {0} \n Where id = {1};", tableName, values);

}else if (entity is Publisher) { tableName = "Publisher";

deleteQuery = string.Format("Delete From {0} \n Where id = {1};", tableName, values);

}

SQLiteCommand deleteCommand = new SQLiteCommand(deleteQuery, sqlite);

deleteCommand.ExecuteNonQuery(); sqlite.Close();

}

}

}

class Book { public int id; public string title; public Publisher publisher; }

class Publisher { public int id; public string name; public List<Book> books; public bool hasbook;

public Publisher() { hasbook = false; }}

**Exercise 5 :** *I used the SQLite NuGet Package to execute the query from the given string and then deserialized the result to the corresponding object.*

public Query<T> createQuery(string query){ return new Query<T>(query); }

class Query<T> : IQuery<T>{

public string query;

public Query(string \_query){ query = \_query; }

public void execute(){

var QueryArry = query.Split();

if (QueryArry[0].ToLower() == "update" || QueryArry[0].ToLower() == "delete"){

using (SQLiteConnection sqlite = new SQLiteConnection("Data Source=annotation.sqlite")) {

sqlite.Open(); SQLiteCommand Command = new SQLiteCommand(query, sqlite);

Command.ExecuteNonQuery(); sqlite.Close();

}

}

}

public List<T> getResultList(){

List<T> ObjList = new List<T>();

using (SQLiteConnection sqlite = new SQLiteConnection("Data Source=annotation.sqlite")) {

sqlite.Open(); SQLiteCommand Command = new SQLiteCommand(query, sqlite);

SQLiteDataReader r = Command.ExecuteReader();

while (r.Read()){

T Obj = (T)Activator.CreateInstance(typeof(T)); ObjList.Add(Obj);

fieldInfo[] fields = typeof(T).GetFields(); fields[0].SetValue(Obj, r.GetInt32(0));

if (typeof(T).Name == "Book"){

fields[0].SetValue(Obj, r.GetInt32(0)); fields[1].SetValue(Obj, r.GetString(1));

var en = new EntityManager<Publisher>(); fields[2].SetValue(Obj, en.find(r.GetInt32(2)));

}else if (typeof(T).Name == "Publisher") {

fields[0].SetValue(Obj, r.GetInt32(0)); fields[1].SetValue(Obj, r.GetString(1));

}else throw new Exception(" T type error");

} sqlite.Close();

} return ObjList;

}

}

class Key{ public string name; public string value;

public Key(string \_name, string \_value) { name = \_name; value = \_value; } }

**A simple test example for Exercise 1 to 5:**

class Program{

static void Main(string[] args){

var res = new TestParser().Test();

new CsharpCodeGenerator(res).GenerateCode();

new SQLCodeGenerator(res).GenerateCode();

if (!System.IO.File.Exists("annotation.sqlite")){

Console.WriteLine("Just entered to create Sync DB");

SQLiteConnection.CreateFile("annotation.sqlite");

using (SQLiteConnection sqlite = new SQLiteConnection("Data Source=annotation.sqlite")) {

sqlite.Open(); string createTableQuery = File.ReadAllText(@"SQLcode.txt");

SQLiteCommand command = new SQLiteCommand(createTableQuery, sqlite);

command.ExecuteNonQuery(); sqlite.Close();

}

}

Publisher p = new Publisher(); p.id = 1; p.name = "BBC";

Publisher q = new Publisher(); q.id = 2; q.name = "CNN";

Book b = new Book(); b.id = 1; b.title = "CS"; b.publisher = p;

Book c = new Book(); c.id = 2; c.title = "AI"; c.publisher = p;

p.books = new List<Book>(); p.books.Add(b); p.hasbook = true;

var ep = new EntityManager<Publisher>(); ep.persist(p); ep.remove(p); ep.persist(q); ep.persist(p);

var eb = new EntityManager<Book>(); eb.persist(b); eb.remove(b); eb.persist(c); eb.persist(b);

Book rb = eb.find(1); Publisher rp = ep.find(1);

string query1 = string.Format("Update book Set title = 'CNN' \n Where id = 1; ");

string query2 = string.Format("Select \* From book \n Where id = 2; ");

List<Book> ObjList1 = new List<Book>(); List<Book> ObjList2 = new List<Book>();

Query<Book> q1 = eb.createQuery(query1); Query<Book> q2 = eb.createQuery(query2);

q1.execute(); ObjList2 = q2.getResultList(); return;

}

}

**Exercise 6:**

Object relation mapping (ORM) is a technique that allows you to address, access and manipulate data from a database using an object-oriented paradigm. An ORM library is a library written in your object oriented language that encapsulate the code needed to manipulate the data, in such a way that you do not have to use SQL anymore, instead in can interact directly with an object in your language.

Linq gives developers the ability to query and transform data using their own language of choice, moreover, Linq incorporates a new way to query data using strongly-typed queries and strongly-typed results, common across a number of disparate data types including relational databases, .NET objects, and XML. By using strongly-typed queries and results, LINQ improves developer productivity with the benefits of IntelliSense and compile-time error checking.

JPA uses Java classes to represent relational database tables which can contain data read from a database or have data which will be written to a database. The mapping of the Java class and its member variables to a database table and fields can be fulfilled by Java annotations to the class or by an XML configuration file.

JPA supports the large data sets, data consistency, concurrent use, transparent persistence, persistent/Transient instances, automatic dirty checking, lazy fetching, outer Join fetching, runtime SQL generation, and inheritance mapping strategies.

In LINQ Query Operators, Quantifier operations return a [Boolean](https://msdn.microsoft.com/en-us/library/system.boolean(v=vs.110).aspx) value that indicates whether some or all of the elements in a sequence satisfy a condition. Operator “All” Returns a value ‘True’ if all elements of a sequence satisfy a predicate condition.Operator “Any” Determines by searching a sequence that whether any element of the some satisfy a specified condition. Operator “Contains” Returns a ‘True’ value if finds that a specific element is there in a sequence, if the sequence does not contains that specific element , ‘false’ value is returned. Equality Operator “SequenceEqual” Results a Boolean value if two sequences are found to be identical to each other.

**Reference:**

1.[Introduction to Recursive Descent Parsers with C#](http://blog.roboblob.com/2014/12/12/introduction-to-recursive-descent-parsers-with-csharp/)

2.[SQL Tutorial](https://www.w3schools.com/sql/)

3.[Getting started with SQLite in C#](http://blog.tigrangasparian.com/2012/02/09/getting-started-with-sqlite-in-c-part-one/)

4.[System.Data.SQLite](https://www.nuget.org/packages/System.Data.SQLite)

5.[Introducing LINQ to Relational Data](https://msdn.microsoft.com/en-us/library/cc161164.aspx)

6.[Introduction To LINQ](http://odetocode.com/Articles/737.aspx)

7.[JPA Persistence and Hibernate:](http://www.yolinux.com/TUTORIALS/Java-JPA.html)

8.[Chapter 2. Why JPA?](https://openjpa.apache.org/builds/1.2.3/apache-openjpa/docs/jpa_overview_why.html)

9[.LINQ - Query Operators](https://www.tutorialspoint.com/linq/linq_query_operators.htm)