Course Project: An Interpreter for SSQL
Design Document

Kuang Yuanyuan

January 2, 2015

CHAPTER 1

INFORMATION

1.1 Team Information

Name	Student ID	E-mail
Sun Jiacheng	12330285	291624707@qq.com
Kuang Yuanyuan	12330153	596755905@qq.com
Qiu Zhilin	12330268	847300960@qq.com
Sun Dongliang	12330284	1255993541@qq.com
Wang Kaibin	12330305	wkbpluto@qq.com

1.2 Tarball Structure

/doc -- documents of project

/src -- source code

1.3 Compile

cd ./src

make -- compile SimpleDB

make test -- compile and run sample test

make tar -- tar the project.

CHAPTER 2

PROJECT

2.1 Project Website

https://github.com/thomasking0529/SimpleDB

2.2 Code Style

K&R(CDT*s default style)

2.3 APIs

2.3.1 Lexer

enum TokenType:

KEYWORD -- select from where delete table create int values primary key default into insert case insensitive
ID -- id (identifier) is a sequence of digits, underline and letters. All identifiers should start with a letter or an underline. The maximum length of an identifier is 64.
NUM -- num (number) is a sequence of digits. (of 32-bits) only integers.
OP -- Arithmetical operators: +, -, *, /, unary -, unary +
Relational operators: <, >, <>, ==, >=, <=
Logical operators: &&, ||, !
Assignment operator: =.
Basic punctuation("(", ")", ",")

struct Token:

TokeType type -- token type.
std::string value -- token value, store the original string of token.

class Lexer:

std::list<Token> GetTokens(const std::string& s) -- get a list of tokens from input string

2.3.2 Parser

enum Action:

CREATE -- create table
INSERT -- insert one row to table
DELETE -- delete row(s) from table
SELECT -- query row(s) from table
INVALID -- if unknown keywords detected

enum Op:

PLUS, // +, both unary and binary
MINUS, // -, both unary and binary
MULTIPLY, // *
DIVIDE, // /
LT, // <
GT, // >
NE, // <>
E, // ==
GTE, // >=
LTE, // <=
AND, // &&
OR, // ||
NOT, // !
EQ, // =
LB, // (
RB, //)
COMMA, // ,

struct Condition:

Condition* lop-- left subtree
Condition* rop-- right subtree
Op op -- operator
std::string opd – operand

struct Property:

std::string id -- property id
int default_value -- default value
operator== -- operator overloading

struct Statement:

Action act -- action.
std::string table -- table to operate on.
std::list<Property> prop_list --property to return or add, for create and select
std::vector<std::string> key_idx -- list of keys
std::vector<int> value_list -- list of values
Conditio* cond -- where clauses, for select and delete

class Polish:

Polish() -- Constructor, initialize private variables
void neglect() -- set neglect flag, for neglect numbers
void Insert(const std::string& item) -- inset a token
int Calculate() -- calculate the expression
Condition* buildTree -- build a condition tree

class Parser:

Parser() -- Constructor, initialize private variables
Statement Parse(const std::string& s) -- main parser

2.3.3 Core

struct Row:

int key -- key value, for identify row
std::vector<int> -- key indices
std::vector<int> cols – values

struct Table:

std::string id -- table name
std::vector<Property> -- properties of table
std::vector<int> -- key indices of table
unsigned long long count -- if no primary key, use this
std::set<Row> rows -- set of rows, use Row.key for key
void Insert(const std::list<int>& record) -- insert
void Delete(const Condition* cond) -- delete
std::vector<int> Query(const Condition* cond) – query

class SimpleDB:

SimpleDB() -- Constructor, initialize private variables
void Execute(const std::string& stmt) -- execute raw input

2.4 Design

2.4.1 Lexer

Lexer reads a single string of statement and extracts tokens from it.

Finite State Automata:
read a charactor from input string;
check charactor for:
a-z *A-Z* *0-9*: concat to temp string
split characters: space, \t, \n, save temp string
symbols: save temp string

2.4.2 Parser

Design

Use LL(1) parser.
LL Parser:

An LL parser is called an LL(k) parser if it uses k tokens of look ahead when parsing a sentence. If such a parser exists for a certain grammar and it can parse sentences of this grammar without backtracking then it is called an LL(k) grammar. LL parser can only parse languages that have LL(k) grammars without ε-rules. LL(k) grammars without ε-rules can generate more languages the higher the number k of look ahead tokens. A corollary of this is that not all context-free languages can be recognized by an LL(k) parser. An LL parser is called an LL(*) parser (an LL-regular parser) if it is not restricted to a finite k tokens of look ahead, but can make parsing decisions by recognizing whether the following tokens belong to a regular language (for example by means of a Deterministic Finite Automaton).

Using the method introduced in the textbook, we construct the following First and Follow table:

	First	Follow
ssql_stmt	create insert delete select	$
create_stmt	create	$
decl_list	id primary	)
decl_list*	, ε	)
decl	id primary	,)
default_spec	defaultε	,)
column_list	id	) from
column_list*	, ε	) from
insert_stmt	insert	$
value_list	(-+ num	)

value_list*	, ε	)
simple_expr	(-+ num	,)
simple_expr*	+ -ε	,)
simple_term	(-+ num	+ -,)
simple_term*	ε /*	+ -,)
simple_unary	(-+ num	/ + -,) *
delete_stmt	delete	$
where_clause	whereε	;
disjunct	(!-+ id num	;)
disjunct*	||ε	;)
conjunct	(! -+ id num	|| ;)
conjunct*	&&ε	|| ;)
bool	(! -+ id num	&& || ;)
rop	== > < >= <= <>	(-+ id num
query_list	select	$
select_list	id *	From
expr	(-+ id num	&& || == > < >= <= <>;)
expr*	+ -ε	&& || == > < >= <= <>;)
term	(+ -id num	;) || && <> == > < >= <= + -
term*	ε / *	;) || && <> == > < >= <= + -
unary	(-+ id num	;) || && <> == > < >=<= + -* /
comp	(+ -id num	&& || ;)

And with the help of the method in the textbook, we generate the following Prediction table:

M[ssql_stmt, create]= ssql_stmt create_stmt
M[ssql_stmt, insert] = ssql_stmt insert_stmt
M[ssql_stmt, delete] = ssql_stmt delete_stmt
M[ssql_stmt, select] = ssql_stmt query_stmt
M[create_stmt, create] = create_stmt create table id (decl_list);
M[create_stmt, create] = create_stmt create table id (decl_list);
M[decl_list, id] = decl_list decl decl_list*
M[decl_list, primary] = decl_list decl decl_list*
…

The remainder is in the Appendix at the end of the document.
The table is stored and read for parsing in the file “Rule.txt” and terminals are stored in the file “Terminals”.

Implementation

- Basic Algorithm

```
Initialization:
    List token = lexer.getTokens();
    Map table = readFile"Rule.txt"
    Map action;
    Stack procedure;
    Stack father;
    procedure.push($);
    procedure.push(ssql_stmt);
    father.push($);
    father.push(ssql_stmt);
    ip = token.first;
    X = procedure.top;
    F = father.top;
Loop
    father.pop;
    procedure.pop;
    If (X match ip) {
        ip = token.next;
        action[ip].translate(ip,    F);
    } else if (ip is a terminals but no match X) {
        Error();
    } else if (X is ε) {
        do nothing
    } else if (table[X, ip] = Y1Y2…..Yk) {
        procedure.push(Yk, Yk-1…..Y1);
        father.push(X, X…..X);
    }
    X = procedure.top;
    F = father.top;
Until X = $
if token is not empty, Error;
```

- Implement detail

  The implement of the parser is as the pseudocode abode, which only use one more stack to store the Grammar Productions' left part, it's still an LL(1) grammar as we still only look one more token to decide which production to use. The father stack store the Grammar productions' left part of the corresponding sysmbol in the procedure stack, just to help with the translation. Then we can translate by tokens seeing it's father.

- Translation problem
  We generate a structure Statement when parsing and the SimpleDB's database manager will use the Statement to do query create etc. As was mentioned before, with the help of the Grammar productions' left part which the token is matched on, we could get the meaning of the token and generate the statement. However, when it comes to the expression, we use the algorithm of reverse polish expression, which implements as class Polish. When a token is seem as part of a expression, we insert it to a object of Polish, and for the math expression, calculate the result, for bool expression, build a condition tree in the Statement.

- Detail of translation:
  Use a object of Polish to reduce the where clause. Use reverse polish to check the priority of braces. Set a flag for neglect numbers, if unary plus encountered, discard it. If a unary minus encountered, if a number followed, flag = !flag, this help to reduce expression like ---1; if an id followed, expand the expression to (0-id); if a left brace followed, expand to ”(0-”, count the left braces, count unary braces followed by left brace and insert the “)” in correct place. If unit operator ! found, make it become a binary one and always give another operand "!!", and when building the condition tree, recognize the “!!” and do something.

2.4.3 Core

Core part of SimpleDB is supposed to manage the database in memory. Get the Statement instance from parser to Execute order.

SimpleDB a set of Table, Table manage a set of Row. Do query, insert, delete on Table.

CHAPTER 3

PROJECT SCHEDULE

3.1 Stage 1

3.1.1 Project architecture and APIs design

Author Deadline

Sun Jiacheng 12.5.2014

3.1.2 Lexer implementation

Author Deadline

Sun Jiacheng 12.5.2014

3.1.3 Group Discussion

Contents:

Stage 2 and Stage 3 work distribution.

Date:

12.6.2014

3.2 Stage 2

3.2.1 Parser

Author Deadline

Qiu Zhilin 12.23.2014

3.2.2 Core

Author Deadline

Sun Dongliang 12.23.2014

3.3 Stage 3

3.3.1 Test Document

Author Deadline

Wang Kaibin 1.2.2014

3.3.2 Design Document

Author Deadline

Kuang Yuanyuan 1.2.2014

CHAPTER 4

APPENDIX

4.1 Prediction table

M[ssql_stmt, create]= ssql_stmt →create_stmt
M[ssql_stmt, insert] = ssql_stmt→ insert_stmt
M[ssql_stmt, delete] = ssql_stmt →delete_stmt
M[ssql_stmt, select] = ssql_stmt →query_stmt
M[create_stmt, create] = create_stmt→ create table id (decl_list);
M[decl_list, id] = decl_list →decl decl_list*
M[decl_list, primary] = decl_list →decl decl_list*
M[decl_list*, ,] = decl_list* →, decl decl_list*
M[decl_list*,)] = decl_list* →ε
M[decl, id] = decl→id int default_spec
M[decl, primary] = decl→primary key (column_list);
M[column_list, id] = column_list →id column_list*
M[column_list*, ,] = column_list*→, id column_list*
M[column_list*,)] = column_list* →ε
M[column_list*, from] = column_list* →ε
M[default_spec, default] = default_spec default = simple_expr
M[default_spec, ,] = default_spec→ε
M[default_spec,)] = default_spec→ε
M[simple_expr, (]= simple_expr →simple_term simple_expr*
M[simple_expr, +]= simple_expr →simple_term simple_expr*
M[simple_expr, -]= simple_expr →simple_term simple_expr*
M[simple_expr, num]= simple_expr →simple_term simple_expr*
M[simple_expr*, +]= simple_expr*→+ simple_term simple_expr*
M[simple_expr*, -]= simple_expr*→-simple_term simple_expr*
M[simple_expr*,)]= simple_expr*→ε
M[simple_expr*, ,]= simple_expr*→ε
M[simple_term, (] = simple_term→simple_unary simple_term*
M[simple_term, -] = simple_term→simple_unary simple_term*
M[simple_term, +] = simple_term→simple_unary simple_term*
M[simple_term, num] = simple_term→simple_unary simple_term*
M[simple_term*,*] = simple_term*→* simple_unary simple_term*
M[simple_term*, /] = simple_term*→/ simple_unary simple_term*
M[simple_term*,)] = simple_term*→ε
M[simple_term*, ,] = simple_term*→ε
M[simple_term*, +] = simple_term*→ε
M[simple_term*, -] = simple_term*→ε
M[simple_unary, (] = simple_unary→ (simple_expr)
M[simple_unary, -] = simple_unary→-simple_unary

M[simple_unary, +] = simple_unary→+simple_unary
M[simple_unary, num] = simple_unary→num
M[insert_stmt, insert] = insert_stmt→insert into id (column_list) values (value_list);
M[value_list, (] = value_list→simple_expr value_list*
M[value_list, +] = value_list→simple_expr value_list*
M[value_list, -] = value_list→simple_expr value_list*
M[value_list, num] = value_list→simple_expr value_list*
M[value_list*, ,] = value_list*→, simple_expr value_list*
M[value_list*,)] = value_list*→ε
M[delete_stmt, delete] = delete_stmt→delete from id where_clause;
M[where_clause, where] = where_clause→where disjunct
M[where_clause, ;]= where_clause→ε
M[disjunct, (] = disjucnt→conjunct disjunct*
M[disjunct, -] = disjucnt→conjunct disjunct*
M[disjunct, +] = disjucnt→conjunct disjunct*
M[disjunct, id] = disjucnt→conjunct disjunct*
M[disjunct, num] = disjucnt→conjunct disjunct*
M[disjunct, !] = disjucnt→conjunct disjunct*
M[disjunct*, ||]= disjunct*→|| conjunct disjunct*
M[disjunct*,)] = disjunct*→ε
M[disjunct*, ;] = disjunct*→ε
M[conjunct, -] = conjunct→bool conjunct*
M[conjunct, +] = conjunct→bool conjunct*
M[conjunct, id] = conjunct→bool conjunct*
M[conjunct, num] = conjunct→bool conjunct*
M[conjunct, (] = conjunct→bool conjunct*
M[conjunct, !] = conjunct→bool conjunct*
M[conjucnt*, &&]= conjunct*→&& bool conjunct*
M[conjunct*,)] = conjunct*→ε
M[conjunct*, ||] = conjunct*→ε
M[conjunct*, ;] = conjunct*→ε
M[bool, (] = bool→ (disjunct)
M[bool, !] = bool→!bool
M[bool, -]= bool→comp
M[bool, +]= bool→comp
M[bool, id]= bool→comp
M[bool, num]= bool→comp
M[comp, -]= comp→expr rop expr
M[comp, +]= comp→expr rop expr
M[comp, id]= comp→expr rop expr
M[comp, num]= comp→expr rop expr
M[rop, <>] = rop→<>
M[rop, ==] = rop→==
M[rop, <] = rop→<

M[rop, >] = rop→>
M[rop, >=] = rop→>=
M[rop, <=] = rop→ <=
M[expr, -]= expr→term expr*
M[expr, +]= expr→term expr*
M[expr, id]= expr→term expr*
M[expr, num]= expr→term expr*
M[expr*, +] = expr*→+ term expr*
M[expr*, -] = expr*→-term expr*
M[expr*,)] = expr*→ε
M[expr*, ;] = expr*→ε
M[expr*, ||] = expr*→ε
M[expr*, &&] = expr*→ε
M[expr*, <>] = expr*→ε
M[expr*, ==] = expr*→ε
M[expr*, >] = expr*→ε
M[expr*, <] = expr*→ε
M[expr*, >=] = expr*→ε
M[expr*, <=] = expr*→ε
M[term, -] = term→unary term*
M[term, +] = term→unary term*
M[term, id] = term→unary term*
M[term, num] = term→unary term*
M[term*,)] = term*→ε
M[term*, ;] = term*→ε
M[term*, ||] = term*→ε
M[term*, &&] = term*→ε
M[term*, <>] = term*→ε
M[term*, ==] = term*→ε
M[term*, >] = term*→ε
M[term*, <] = term*→ε
M[term*, >=] = term*→ε
M[term*, <=] = term*→ε
M[term*, +] = term*→ε
M[term*, -] = term*→ε
M[term*, *] = term*→* unary term*
M[term*, /] = term*→/ unary term*
M[unary, -] = unary→-unary
M[unary, +] = unary→+unary
M[unary, id] = unary→id
M[unary, num] = unary→num
M[query_list , select] = query_list→select select_list from id where_clause;
M[select_list, *] = select_list→*
M[select_list, id] = select_list→column_list