code.md

```
# Group: Gideon Walker and Ryan Leas
# Project: SQL Compiler Part 1
# Date: 10/26/17
# Class: CS 5300 Databases

###############################################################################
#                                                                             #
#  LEXER                                                                      #
#                                                                             #
###############################################################################

# Token types
#
# EOF (end-of-file) token is used to indicate that
# there is no more input left for lexical analysis
INTEGER         = 'INTEGER'
STRING          = 'STRING'
PLUS            = 'PLUS'
MINUS           = 'MINUS'
MUL             = 'MUL'
LPAREN          = 'LPAREN'
RPAREN          = 'RPAREN'
ID              = 'ID'
ASSIGN          = 'ASSIGN'
SEMI            = 'SEMI'
DOT             = 'DOT'
COLON           = 'COLON'
COMMA           = 'COMMA'
EOF             = 'EOF'
KEYWORD         = 'KEYWORD'
SELECT          = 'SELECT'
FROM            = 'FROM'
WHERE           = 'WHERE'
AS              = 'AS'
IN              = 'IN'
CONTAINS        = 'CONTAINS'
INTERSECT       = 'INTERSECT'
UNION           = 'UNION'
EXCEPT          = 'EXCEPT'
HAVING          = 'HAVING'
GROUPBY         = 'GROUPBY'
AND             = 'AND'
OR              = 'OR'
EQUAL           = 'EQUAL'
GREATER         = 'GREATER'
LESSER          = 'LESSER'
GREATEREQUAL    = 'GREATEREQUAL'
LESSEREQUAL     = 'LESSEREQUAL'
MIN             = 'MIN'
MAX             = 'MAX'
SUM             = 'SUM'
COUNT           = 'COUNT'
AVG             = 'AVG'

SPACES = 8
# Helper Function
def flatten(S):
    if S == []:
        return S
    if isinstance(S[0], list):
        return flatten(S[0]) + flatten(S[1:])
    return S[:1] + flatten(S[1:])
```

```
# structure of tree node class set up here, used later on in
# Interpreter section
class Tree_Node(object):
    def __init__(self, left, right, value):
        self.left = left
        self.right = right
        self.value = value

# structure of token established here, used throughout program
class Token(object):
    def __init__(self, type, value):
        self.type = type
        self.value = value

    def __str__(self):
        """String representation of the class instance.

        Examples:
            Token(INTEGER, 3)
            Token(PLUS, '+')
            Token(MUL, '*')
        """
        return 'Token({type}, {value})'.format(
            type=self.type,
            value=repr(self.value)
        )

    def __repr__(self):
        return self.__str__()

# list of reserved keyword tokens
RESERVED_KEYWORDS = {
    'BEGIN': Token('BEGIN', 'BEGIN'),
    'END': Token('END', 'END'),
    'SELECT': Token('SELECT', 'SELECT'),
    'FROM': Token('FROM', 'FROM'),
    'WHERE': Token('WHERE', 'WHERE'),
    'AS': Token('AS', 'AS'),
    'AND': Token('AND', 'AND'),
    'OR': Token('OR', 'OR'),
    'IN': Token('IN', 'IN'),
    'CONTAINS': Token('CONTAINS', 'CONTAINS'),
    'INTERSECT': Token('INTERSECT', 'INTERSECT'),
    'UNION': Token('UNION', 'UNION'),
    'EXCEPT': Token('EXCEPT', 'EXCEPT'),
    'HAVING': Token('HAVING', 'HAVING'),
    'GROUPBY': Token('GROUPBY', 'GROUPBY'),
    'MIN': Token('MIN', 'MIN'),
    'MAX': Token('MAX', 'MAX'),
    'COUNT': Token('COUNT', 'COUNT'),
    'SUM': Token('SUM', 'SUM'),
    'AVG': Token('AVG', 'AVG'),

}

# first major component of this program, breaks everything into tokens
class Lexer(object):
    def __init__(self, text):
        # client string input, e.g. "4 + 2 * 3 - 6 / 2"
        self.text = text
        # self.pos is an index into self.text
        self.pos = 0
        self.current_char = self.text[self.pos]

    # raises exception if a character cannot be converted to token
    def error(self):
        raise Exception('Invalid character near or at "{}"'.format(self.current_char))

    def advance(self):
        """Advance the `pos` pointer and set the `current_char` variable."""
```

```
        self.pos += 1
        if self.pos > len(self.text) - 1:
            self.current_char = None  # Indicates end of input
        else:
            self.current_char = self.text[self.pos]

    # looks at the next character without incrementing pos
    def peek(self):
        peek_pos = self.pos + 1
        if peek_pos > len(self.text) - 1:
            return None
        else:
            return self.text[peek_pos]

    def skip_whitespace(self):
        while self.current_char is not None and self.current_char.isspace():
            self.advance()

    def integer(self):
        """Return a (multidigit) integer consumed from the input."""
        result = ''
        while self.current_char is not None and self.current_char.isdigit():
            result += self.current_char
            self.advance()
        return int(result)

    def string(self):
        """ Return a string consumed from the input """
        result = ''
        if self.current_char == "'":
            self.advance()
        # import ipdb; ipdb.set_trace()
        while self.current_char != "'":
            result += str(self.current_char)
            self.advance()
        if self.current_char == "'":
            self.advance()
        else:
            self.error()
        return result

    def _id(self):
        """Handle identifiers and reserved keywords"""
        result = ''
        while self.current_char is not None and self.current_char.isalnum():
            result += self.current_char
            self.advance()

        token = RESERVED_KEYWORDS.get(result, Token(ID, result)) # Gets the keyword or returns identifier token
        return token

    def get_next_token(self):
        """Lexical analyzer (also known as scanner or tokenizer)

        This method is responsible for breaking a sentence
        apart into tokens. One token at a time.
        """
        while self.current_char is not None:

            if self.current_char.isspace():
                self.skip_whitespace()
                continue

            if self.current_char.isalpha():
                return self._id()

            if self.current_char.isdigit():
                return Token(INTEGER, self.integer())

            if self.current_char == "'":
                return Token(STRING, self.string())
```

```
            if self.current_char == ';':
                self.advance()
                return Token(SEMI, ';')

            if self.current_char == '*':
                self.advance()
                return Token(MUL, '*')

            if self.current_char == '(':
                self.advance()
                return Token(LPAREN, '(')

            if self.current_char == ')':
                self.advance()
                return Token(RPAREN, ')')

            if self.current_char == '.':
                self.advance()
                return Token(DOT, '.')

            if self.current_char == '=':
                self.advance()
                return Token(EQUAL, '=')

            if self.current_char == '>' and self.peek() == '=':
                self.advance()
                self.advance()
                return Token(GREATEREQUAL, '>=')

            if self.current_char == '<' and self.peek() == '=':
                self.advance()
                self.advance()
                return Token(LESSEREQUAL, '<=')

            if self.current_char == '>':
                self.advance()
                return Token(GREATER, '>')

            if self.current_char == '<':
                self.advance()
                return Token(LESSER, '<')

            if self.current_char == ',':
                self.advance()
                return Token(COMMA, ',')

            self.error()

        return Token(EOF, None)


################################################################################
#                                                                              #
#  PARSER                                                                      #
#                                                                              #
################################################################################

# abstract-syntax tree base class, not much here but added onto by more
# specific AST nodes that inherit from this
class AST(object):
    pass

# binary opreator
class BinOp(AST):
    def __init__(self, left, op, right):
        self.left = left
        self.token = self.op = op
        self.right = right

# number (integer)
```

```
class Num(AST):
    def __init__(self, token):
        self.token = token
        self.value = token.value

# unary operation
class UnaryOp(AST):
    def __init__(self, op, expr):
        self.token = self.op = op
        self.expr = expr


class Compound(AST):
    """Represents a 'BEGIN ... END' block"""
    def __init__(self):
        self.children = []

# assignment statement
class Assign(AST):
    def __init__(self, left, op, right):
        self.left = left
        self.token = self.op = op
        self.right = right


class Var(AST):
    """The Var node is constructed out of ID token."""
    def __init__(self, token):
        self.token = token
        self.value = token.value

# relational algebra select operation
class Rel_Alg_Select(AST):
    def __init__(self, left, op, right):
        self.left = left
        self.token = self.op = op
        self.right = right

# attributes
class Attr(AST):
    def __init__(self, attribute, relation=None):
        self.attribute = attribute.value
        if relation:
            self.relation = relation.value
        else:
            self.relation = None

class Ag_Function(AST):
    def __init__(self, function, attribute, alias=None):
        self.function = function
        self.attribute = attribute
        self.alias = alias

# realtions
class Rel(AST):
    def __init__(self, relation, alias=None):
        self.relation = relation.value
        if alias:
            self.alias = alias.value
        else:
            self.alias = None


# if there is no operation in tree, just passes over
class NoOp(AST):
    pass

class Query(AST):
    def __init__(self, projects, relations, selects=None, groupby=None, having=None, nested=None):
        self.selects = selects
        self.projects = projects
```

```
            self.relations = relations
            self.groupby = groupby
            self.having = having
            self.nested = nested

    class Nest_Query(AST):
        def __init__(self, attribute, op, query):
            self.attribute = attribute
            self.op = op
            self.query = query

    class Set_Op(AST):
        def __init__(self, left=None, right=None, op=None):
            self.left = left
            self.right = right
            self.op = op

    # main Parser class
    class Parser(object):
        def __init__(self, lexer):
            self.lexer = lexer
            # set current token to the first token taken from the input
            self.current_token = self.lexer.get_next_token()

        def error(self):
            raise Exception('Invalid syntax near or at "{}"'.format(self.current_token.value))

        def eat(self, token_type):
            # compare the current token type with the passed token
            # type and if they match then "eat" the current token
            # and assign the next token to the self.current_token,
            # otherwise raise an exception.
            if self.current_token.type == token_type:
                print(self.current_token)
                self.current_token = self.lexer.get_next_token()
            else:
                self.error()

        def query(self):
            # query: compound statement
            #      | (? compound statement )?
            if self.current_token.type == LPAREN:
                self.eat(LPAREN)
            node = self.sql_compound_statement()
            if self.current_token.type == RPAREN:
                self.eat(RPAREN)
            # self.eat(SEMI)
            return node

        def sql_compound_statement(self):
            """
            note: ? means 0 or 1 instances
            sql_compound_statement: SELECT attribute_list
                                    FROM relation_list
                                    (WHERE condition_list)?
                                    (GROUP BY attribute_list)?
                                    (HAVING condition_list)?
                                    (INTERSECT | UNION | EXCEPT | CONTAINS sql_compound_statement)?
            """
            cond_nodes = list()
            group_by_list = list()
            having_list = list()
            compound_statement = None
            set_op = ''
            self.eat(SELECT)
            attr_nodes = self.attribute_list()
            self.eat(FROM)
            rel_nodes = self.relation_list()
            if self.current_token.type == WHERE:
                self.eat(WHERE)
                cond_nodes = self.condition_list()
```

```
        if self.current_token.type == GROUPBY:
            self.eat(GROUPBY)
            group_by_list = self.attribute_list()
        if self.current_token.type == HAVING:
            self.eat(HAVING)
            having_list = self.condition_list()
        if self.current_token.type in (INTERSECT, UNION, EXCEPT, CONTAINS):
            set_op = self.current_token.type
            if self.current_token.type == INTERSECT:
                self.eat(INTERSECT)
            elif self.current_token.type == UNION:
                self.eat(UNION)
            elif self.current_token.type == EXCEPT:
                self.eat(EXCEPT)
            elif self.current_token.type == CONTAINS:
                self.eat(CONTAINS)
            compound_statement = self.query()

        query = Query(attr_nodes, rel_nodes, cond_nodes, group_by_list, having_list)
        if compound_statement:
            return Set_Op(query, compound_statement, set_op)
        else:
            return query

    def attribute_list(self):
        """
        attribute_list : (attribute | ag_function) (COMMA attribute_list)*
        """
        if self.current_token.type == ID:
            node = self.attribute()
        else:
            node = self.ag_function()
        results = [node]
        while self.current_token.type == COMMA:
            self.eat(COMMA)
            if self.current_token.type == ID:
                next = self.attribute()
            else:
                next = self.ag_function()
            results.append(next)
        return results

    def ag_function(self):
        """ag_function: (MIN | MAX | SUM | COUNT | AVG) (attribute) (AS alias):"""
        function = self.current_token.value
        if self.current_token.type == MAX:
            self.eat(MAX)
        elif self.current_token.type == MIN:
            self.eat(MIN)
        elif self.current_token.type == SUM:
            self.eat(SUM)
        elif self.current_token.type == COUNT:
            self.eat(COUNT)
        elif self.current_token.type == AVG:
            self.eat(AVG)
        else:
            self.error()

        self.eat(LPAREN)
        attribute = self.attribute()
        self.eat(RPAREN)

        if self.current_token.type == AS:
            self.eat(AS)
            alias = self.current_token.value
            self.eat(ID)
            return Ag_Function(function, attribute, alias)

        return Ag_Function(function, attribute)

    def attribute(self):
```

```
        """
        attribute : identifier
                  | identifier DOT identifier
                  | STAR aka MUL

        """
        node = Attr(self.current_token)
        if self.current_token.type == MUL:
            self.eat(MUL)
        else:
            self.eat(ID)
            if self.current_token.type == DOT:
                self.eat(DOT)
                node.relation = node.attribute
                node.attribute = self.current_token.value
                self.eat(ID)
        return node

    def relation_list(self):
        """
        relation_list : relation
                      | relation COMMA relation_list
        """
        node = self.relation()
        results = [node]
        while self.current_token.type == COMMA:
            self.eat(COMMA)
            results.append(self.relation())
        return results

    def relation(self):
        """
        relation : identifier
                 | identifier AS identifier
        """
        node = Rel(self.current_token)
        self.eat(ID)
        if self.current_token.type == AS:
            self.eat(AS)
            node.alias = self.current_token.value
            self.eat(ID)
        return node

    def condition_list(self):
        """
        condition_list : condition
                       | condition (AND | OR) condition_list
        """
        node = self.condition()
        results = [node]
        while self.current_token.type in (AND, OR):
            if self.current_token.type == AND:
                self.eat(AND)
            else:
                self.eat(OR)
            results.append(self.condition())
        return results

    def condition(self):
        """
        condition : attribute (EQUAL | GREATER | LESSER | GREATEREQUAL | LESSEREQUAL) (attribute | INTEGER | STRING)
                  | attribute IN LPAREN sql_compound_statement RPAREN
        """
        # Left is always attribute
        left = self.attribute()
        if self.current_token.type in (IN,EQUAL, GREATER, LESSER, GREATEREQUAL, LESSEREQUAL):
            # Comparison
            token = self.current_token
            if self.current_token.type == EQUAL:
                self.eat(EQUAL)
            elif self.current_token.type == GREATER:
```

```
                self.eat(GREATER)
            elif self.current_token.type == LESSER:
                self.eat(LESSER)
            elif self.current_token.type == GREATEREQUAL:
                self.eat(GREATEREQUAL)
            elif self.current_token.type == LESSEREQUAL:
                self.eat(LESSEREQUAL)
            elif self.current_token.type == IN:
                self.eat(IN)

            # Right: integer, string, or attribute
            if self.current_token.type == INTEGER:
                right = self.current_token
                self.eat(INTEGER)
            elif self.current_token.type == STRING:
                right = self.current_token
                self.eat(STRING)
            elif self.current_token.type == LPAREN:
                self.eat(LPAREN)
                node = self.query()
                if self.current_token.type == RPAREN:
                    self.eat(RPAREN)
                sub_query = Nest_Query(left, token.value, node)
                return sub_query
            else: # attribute
                right = self.attribute()
            return Rel_Alg_Select(left, token, right)



    def parse_sql(self):
        """
        query: sql_compound_statement
        sql_compound_statement: SELECT attributes FROM (relations | query) WHERE (conditions | attributes IN query)
        """
        # import ipdb; ipdb.set_trace()
        node = self.query()
        if self.current_token.type != EOF:
            self.error()
        self.eat(EOF)
        return node


###############################################################################
#                                                                             #
#  INTERPRETER                                                                #
#                                                                             #
###############################################################################

# base Node visitor class, other more specific visits use these methods
class NodeVisitor(object):
    def visit(self, node):
        method_name = 'visit_' + type(node).__name__
        visitor = getattr(self, method_name, self.generic_visit)
        return visitor(node)

    def generic_visit(self, node):
        raise Exception('No visit_{} method'.format(type(node).__name__))

# main interpreter class
class Interpreter(NodeVisitor):

    # declares lists for selects, projects, and cross products.
    # lists used to generate relational algebra and query trees
    GLOBAL_SCOPE = {}
    QUERIES = list()
    SET_OPS = list()

    def __init__(self, parser):
        self.parser = parser
```

```
    def visit_Set_Op(self, set_op):
        left = self.visit(set_op.left)
        op = set_op.op
        right = self.visit(set_op.right)
        return Set_Op(left, right, op)

    def visit_Nest_Query(self, nest_query):
        # import ipdb; ipdb.set_trace()
        left = nest_query.attribute
        op = Token(EQUAL, '=')
        if isinstance(nest_query.query, Query):
            right = nest_query.query.projects.pop(0) #Only one ever
            condition = Rel_Alg_Select(left, op, right)
            nest_query.query.selects.append(condition)
        return self.visit(nest_query.query)

    def visit_Query(self, query):
        selects = list()
        projects = list()
        relations = list()
        for item in query.projects:
            projects.append(self.visit(item))
        for item in query.relations:
            relations.append(self.visit(item))
        new_query = Query(projects, relations)
        for item in query.selects:
            if isinstance(item, Nest_Query):
                nested_query = self.visit(item)
                if isinstance(nested_query, Query):
                    for itemx in nested_query.relations:
                        relations.append(itemx)
                    for itemx in nested_query.selects:
                        selects.append(itemx)
                else:
                    new_query.nested = nested_query
            else:
                selects.append(self.visit(item))
        new_query.selects = selects
        return new_query


    def visit_Rel_Alg_Select(self, node):
        if node.left.relation: # always attribute
            left = node.left.relation + '.' + node.left.attribute
        else:
            left = node.left.attribute

        if isinstance(node.right, Attr):
            if node.right.relation:
                right = node.right.relation + '.' + node.right.attribute
            else:
                right = node.right.attribute
        else:
            right = str(node.right.value)
        result = left +' '+ node.op.value +' '+ right
        return result

    def visit_list(self, node):
        for item in node:
            self.visit(item)

    def visit_Num(self, node):
        return node.value


    def visit_Compound(self, node):
        for child in node.children:
            self.visit(child)

    def visit_Attr(self, node):
        atr_name = node.attribute
```

```
        if node.relation:
            rel_name = node.relation
            atr_name = rel_name + '.' + atr_name
        return atr_name

    def visit_Ag_Function(self, node):
        ag_function = node.function
        attribute = self.visit(node.attribute)
        ag_function += '(' + attribute + ')'
        if node.alias:
            ag_function += ' AS ' + node.alias
        return ag_function

    def visit_Rel(self, node):
        rel_name = list()
        rel_name.append(node.relation)
        if node.alias:
            rel_name.append(node.alias)
        return rel_name

    def visit_NoOp(self, node):
        pass

    # function that calls the first visit and starts the interpretation
    def interpret(self):
        tree = self.parser.parse_sql()
        if tree is None:
            return ''
        return self.visit(tree)

# prints the relational algebra using the lists
def print_rel_alg(interpreter, end=''):

    print('PROJECT [', end='')
    for idx, item in enumerate(interpreter.projects):
        if idx == len(interpreter.projects) - 1:
            print(item, end='')
        else:
            print('{}, '.format(item), end='')
    print('] (SELECT [', end='')
    for idx, item in enumerate(interpreter.selects):
        if idx == len(interpreter.selects) - 1:
            print(item, end='')
        else:
            print('{} AND '.format(item), end='')
    print('] (', end='')
    for idx, list in enumerate(interpreter.relations):
        if idx == len(interpreter.relations) - 1:
            if len(list) == 1:
                print(list[0], end='')
            else:
                print('{} AS {}'.format(list[0], list[1]), end='')
        else:
            if len(list) == 1:
                print('{} X '.format(list[0]), end='')
            else:
                print('{} AS {} X '.format(list[0], list[1]), end='')
    print(')))', end=end)

def build_set_op_tree(set_op):
    return Tree_Node(build_query_tree(set_op.left), build_query_tree(set_op.right), set_op.op)

# builds the query tree using lists generated from visits
def build_query_tree(interpreter):
    project = 'PROJECT ['
    for idx, item in enumerate(interpreter.projects):
        if idx == len(interpreter.projects) - 1:
            project += item
        else:
            project += '{}, '.format(item)
    project += ']'
```

```
    tree = Tree_Node(None, None, project)
    select = 'SELECT ['
    for idx, item in enumerate(interpreter.selects):
        if idx == len(interpreter.selects) - 1:
            select += item
        else:
            select += '{} AND '.format(item)
    select += ']'
    select_node = Tree_Node(None, None, select)
    tree.left = select_node
    cross_node = build_cross_tree(interpreter.relations)
    tree.left.left = cross_node
    return tree

# separate frunction for building the cross product trees
def build_cross_tree(cross_prods):
    node = Tree_Node(None, None, None)
    if len(cross_prods) == 1:
        node.value = cross_prods[0]
        return node
    elif len(cross_prods) == 2:
        node.left = Tree_Node(None, None, cross_prods[0])
        node.right = Tree_Node(None, None, cross_prods[1])
        node.value = 'X'
        return node
    else:
        node.right = Tree_Node(None, None, cross_prods.pop(0))
        node.left = build_cross_tree(cross_prods)
        node.value = 'X'
        return node

# function to actually print/format the query tree
def print_query_tree(tree, spaces):
    if tree:
        spaces += SPACES
        print_query_tree(tree.right, spaces)
        spaces -= SPACES
        if tree.right != None:
            print(' ' * spaces, end='')
            print('/')
        if spaces != 0:
            print(' '*(spaces - SPACES), end='')
            print(' |' + '-'*(SPACES-2), end='')
        print(tree.value)
        if tree.left != None:
            print(' ' * spaces, end='')
            print('\\')
        spaces += SPACES
        print_query_tree(tree.left, spaces)
        spaces -= SPACES
    return

def main():

    # brings in the SQL query and the tables that represent the relation
    import sys
    text = open(sys.argv[1], 'r').read()
    tables = open(sys.argv[2], 'r').read()

    text = text.upper()
    # lexer called first to break into tokens
    lexer = Lexer(text)
    # parser called to generate tree for interpreter
    parser = Parser(lexer)
    # interpreter visits nodes on tree and generates relational algebra
    # and trees from it
    interpreter = Interpreter(parser)
    result = interpreter.interpret()
    if isinstance(result, Query):
        print(result.projects)
        print(result.selects)
```

```
            print(result.relations)
        elif isinstance(result, Set_Op):
            print(result.left.projects)
            print(result.left.selects)
            print(result.left.relations)
            print(result.op)
            print(result.right.projects)
            print(result.right.selects)
            print(result.right.relations)
        print('######################################')
        print('#          Relation Algebra          #')
        print('######################################\n')
        if isinstance(result, Query):
            print_rel_alg(result, end='\n')
        elif isinstance(result, Set_Op):
            print_rel_alg(result.left)
            print(' {} '.format(result.op), end='')
            print_rel_alg(result.right, end='\n')
        print('######################################')
        print('#            Query Tree              #')
        print('######################################\n')
        tree = None
        if isinstance(result, Query):
            tree = build_query_tree(result)
        elif isinstance(result, Set_Op):
            tree = build_set_op_tree(result)
        print_query_tree(tree, 0)


    if __name__ == '__main__':
        main()
```