Compilers project fall2019

# Project Title: ETL compiler framework

## Purpose:

The purpose of this project is to practice the need and implementation for scanning and parsing. On the other hand link the old theories of compilers to serve a hot and demanding area in data science namely the extract, transform and load (ETL).

## Motivation:

The SQL syntax has the power of a fourth-generation language where the programmer specifies only what is required without specifying details of how. The use of the “Select” SQL-construct allows explicit extract of data. Each selected expression allows rich string and mathematical range of operations, hence giving space for lite yet rich transform process. Finally the “Insert” SQL-construct together with “Select … into …” allows the process of loading data.

A framework that can deal with multiple data sources is very useful if it has a compiler that can scan, and parse SQL-like statements compile into a portable language such as python. An ETL task can be as easy as a SQL query.

The architecture of the framework consists of loosely coupled layers that allow plugging in of drivers that implement predefined interfaces to allow the framework to interact with new data sources.

The use of a compiler in the core of the framework allows future extension of new grammar constructs beyond the standard SQL grammar constructs.

# Project definition:

Design and build a scalable compiler that can process SQL-like statements that can Extract, transform and load from and into different data sources. The scope of this project is CSV files, MS Access, MS SQL-server.

# 1) Select: a statement that selects columns by name or index

Test cases:

1. Select \*

into [“Data Source=SQL;Initial Catalog=TestDB;Persist Security Info=True;User ID=sa;Password=P@ssw0rd”]

from [D:\file1.csv]

1. Select [1],[3],[4]

From [D:\file1.csv]

1. Select studentName, Phone

From [“Data Source=SQL;Initial Catalog=TestDB;Persist Security Info=True;User ID=sa;Password=P@ssw0rd”]

1. Select Phone, StudentName

From [D:\file1.csv]

Where DOB>”2019-12-31”

1. Select Phone, Studentname, city

[D:\file1.csv]

Where Phone like “065%”

Group by Phone, StudenName, city

1. Select Phone, Studentname, city, State, zipcode

From [D:\file1.csv] as file1 Inner join [D:\file2.csv] as file2

on file1.Phone=file2.phone

where city=”Houston”

order by zipcode

# 2) Insert : a statement that inserts line(s) into a file

Test cases:

1. Insert into [D:\file1.csv]

Select “value1”, “Value2”,…..

1. Insert into [D:\file1.csv]

Select \* from [D:\file3.csv]

-Note: Assume file1 and file2 should have the same number of fields in each line

# 3)Update: a statement that updates a line(s) in a file

Test cases:

1. Update [“Data Source=SQL;Initial Catalog=TestDB;Persist Security Info=True;User ID=sa;Password=P@ssw0rd”]

Set city =“value1”, phone=“Value2”,…..

1. Update [D:\file1.csv]

Set city =“value1”, phone=“Value2”,…..

Where …….

# 4)Delete: a statement that deletes a line(s) from a file

Test cases:

1. Delete From [D:\file1.csv]

Where …….

## Milestones

## 1) A class cls\_DS\_studentName that can read a zipped csv/txt file or from a given connection string to a sqlite DB ; and filters (i.e. for phone numbers that start with 805); and also selects named/numbered columns.

Test cases, get the count of rows for :

1. All columns all rows and give a rows and columns count.

i.e. select \*

1. All columns and filters on some values

i.e. DOB>”2019-12-31”

1. All named/numbered columns

i.e. Select [1],[3],[4]

From [D:\file1.csv]

1. Select from two different files and get the count of rows whose values match

Select Phone, Studentname, city, State, zipcode

From [D:\file1.csv] as file1 Inner join [D:\file2.csv] as file2

on file1.Phone=file2.phone

where city=”Houston”

order by zipcode

-Use pandas or dask libraries.

i.e.

import pandas

chunks = pd.read\_csv(“random.csv.zip”, chunksize=100000)  
data = pd.concat(chunks)

The above code snippet loads the contents of a whole csv file into memory. This will break the input file into chunks instead of loading the whole file into memory. This will reduce the pressure on memory for large input files and given an optimal chunksize found through trial and error, there can be significant increase in efficiency. The code below will split the input file into chunks of 100 000 lines

dask.dataframe is superior to the two above methods, can you

import dask.data

framedata = dask.dataframe.read\_csv(“random.csv.zip”)

<https://medium.com/casual-inference/the-most-time-efficient-ways-to-import-csv-data-in-python-cc159b44063d>

Another mature scenario for handling large files is:

<https://pythondata.com/working-large-csv-files-python/>

it creates a sqlite file and loads frames of data into it then you can execute any sql query!

import pandas as pd

import zipfile as zf #library for reading from a zip file

fPath= 'C:/Users/Teba/PycharmProjects/untitled/Test\_3500000.zip'

df = pd.read\_csv(zf.open(fPath),encoding="latin",chunksize=10\*\*5,iterator=True)

def process\_chunk(chunk,qu): #process each line of the chunk until it ends

print(chunk)

df = chunk[chunk.PHONE.str.startswith ('805',na=False)]

qu.put((len(df.index),len(chunk.index) ))

## 2) Design and implement a lex that can accept select statements.

## 3) Build a CFG for the above statements on paper.

## 4) Build a CSV parser class that loads pages of a large CSV into memory

## 5) Build a compiler class that receives a SQL like statement and executes it on the three different data sources (CSV, Access/Sqlite , SQLserver/Mysql)

You should allow different queries to be executed in-memory. You are encouraged to use ready made frameworks like pandas in python. Don’t write your own code for this part.

i.e.

import csv with open(“random.csv.zip”) as csvfile:   
 data = csv.DictReader(csvfile)  
 for row in data:  
 print(row['A'])

The above code snippet loads the contents of a whole csv into a collection of rows in-memory where each row is a dictionary. This will block your RAM if given a large CSV

i.e.

import pandas

chunks = pd.read\_csv(“random.csv.zip”, chunksize=100000)  
data = pd.concat(chunks)

The above code snippet loads the contents of a whole csv file into memory. This will break the input file into chunks instead of loading the whole file into memory. This will reduce the pressure on memory for large input files and given an optimal chunksize found through trial and error, there can be significant increase in efficiency. The code below will split the input file into chunks of 100 000 lines

dask.dataframe is superior to the two above methods, can you

import dask.data

framedata = dask.dataframe.read\_csv(“random.csv.zip”)

<https://medium.com/casual-inference/the-most-time-efficient-ways-to-import-csv-data-in-python-cc159b44063d>

Another mature scenario for handling large files is:

<https://pythondata.com/working-large-csv-files-python/>

it creates a sqlite file and loads frames of data into it then you can execute any sql query!

## 3) Use your CFG to parse the command and attach an action for each CFG rule.

The actions should be creating the appropriate translation into Python code in form of a string.

## 4) The strings generated from the previous phase are passed to the sematic module to annotate each string with annotations that help code optimization and generation.

The sematic analysis decides whether to open two data sources or one data source and return the result as a list of strings.

## 5) code optimization: Using annotations from the previous phase some strings could be merged, splited or removed from the output.

The result should be one single string containing valid python code.

## 6) code generation:

The input string is first written into a .py file then the string is compiled and executed.

Appendix

# A1) Use of Lex and YACC “Yet Another Compiler Compiler”

you are encouraged to use PLY for LEX and YACC to make tokenizing and parsing much easier for you

To install PLY on your machine for python2/3, follow the steps outlined below:

1. Download the source code from [here](http://www.dabeaz.com/ply/ply-3.10.tar.gz).
2. Unzip the downloaded zip file
3. Navigate into the unzipped ply-3.10 folder
4. Run the following command in your terminal: python setup.py install

If you completed all the above, you should now be able to use the PLY module. You can test it out by opening a python interpreter and typing import ply.lex.

Note: Do not use pip to install PLY, it will install a broken distribution on your machine.

[ <https://www.dabeaz.com/ply/ply.html> ]

[<https://my.eng.utah.edu/~cs3100/lectures/l14/ply-3.4/doc/ply.html#ply_nn2>]

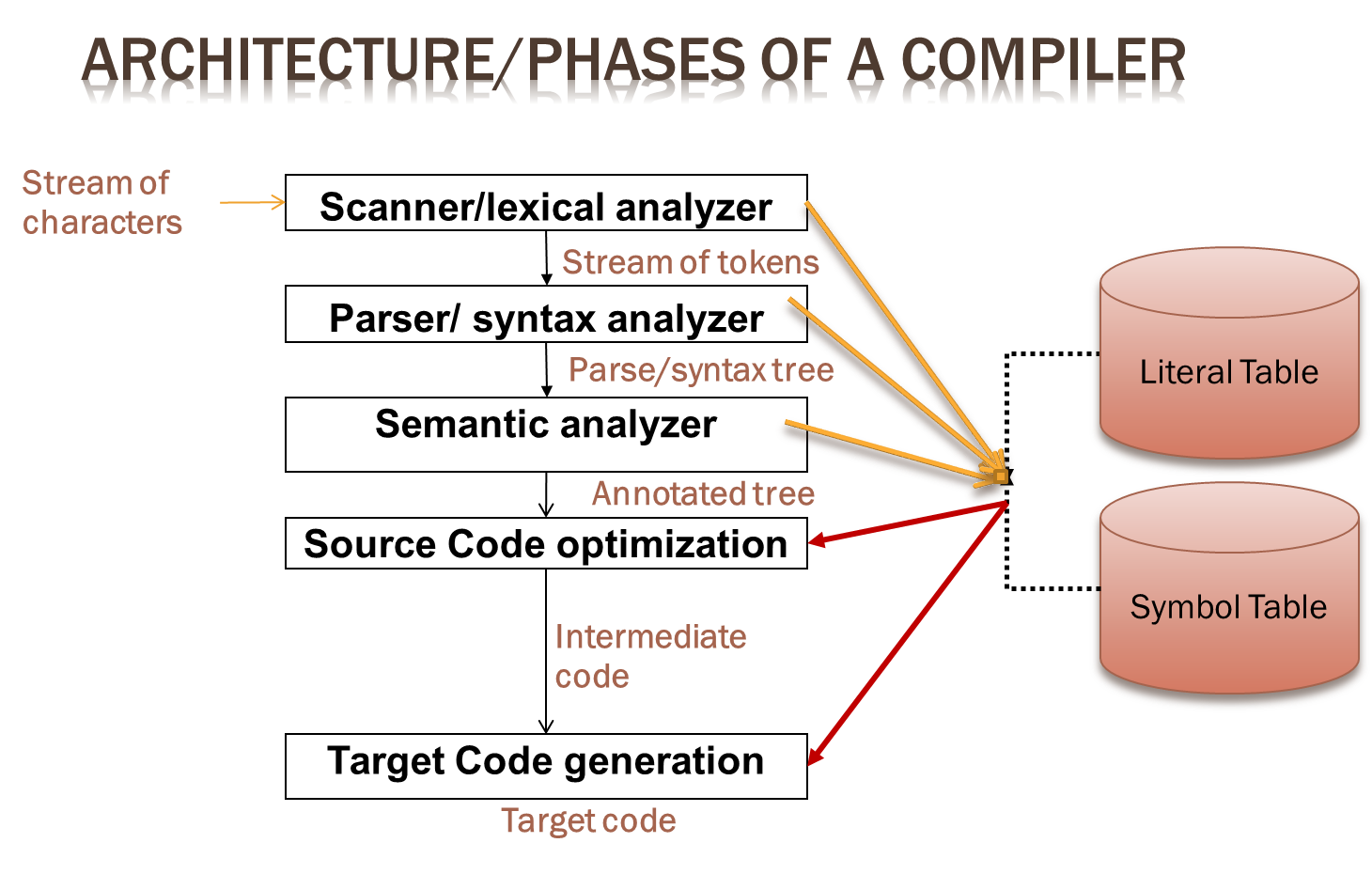
[<http://www.dabeaz.com/ply/example.html>]

[<https://jindongpu.wordpress.com/2016/03/13/lex-and-yacc-example-in-python/>]

# A2) suggested design pattern for the framework

The framework should allow for more datasources drivers to be defined and plugged. It also allows for more grammar to be defined and plugged into the framework. The most useful technique would be the use of an abstract class that is inherited for any newly driver.

A typical compiler consists of the of the following phases:



# Compiler.py

abstract class cls\_Compiler{

Dictionary literal\_hashTable( string, Token);

Dictionary Symbol\_hashTable( string, Token);

public compile(string inputProgram){

lexer=cls\_Lexer()

parser=cls\_Parser()

Semantics = cls\_Semantic()

codeOptimizer = cls\_codeOptimizer()

pythonCodeGenerator= cls\_codeGenerator()

stream\_of\_Tokens = lexer.tokenize(inputProgram)

parseTree = parser.parse(stream\_of\_Tokens)

AnnotatedTree = Semantics.parse(parseTree)

Intermediate\_CodeString = codeOptimizer.optimize(AnnotatedTree)

pythonCodeString = pythonCodeGenerator.generate(Intermediate\_CodeString)

}

public Excute(string python\_Code);

}

# Lexer.py

import ply.lex as lex

abstract class cls\_Lexer(lex.lex)

{

def \_\_init\_\_(self ):

super().\_\_init\_\_(self)

self.streamofTokens = []

**def build(self,\*\*kwargs):**

**self.lexer = lex.lex(module=self, \*\*kwargs)**

def tokenize(self,data):

self.lexer.input(data)

while True:

tok = lexer.token()

if not tok: break

print tok

return self.streamofTokens

/\*

Each token is specified by writing a regular expression rule compatible with Python's re module. Each of these rules are defined by making declarations with a special prefix t\_ to indicate that it defines a token.

\*/

def t\_NUMBER(t):

r'\d+'

t.value = int(t.value)

self.streamofTokens.append(t)

return t

identifier = r'(' + nondigit + r'(' + digit + r'|' + nondigit + r')\*)'

from ply.lex import TOKEN

@TOKEN(identifier)

def t\_ID(t):

self.streamofTokens.append(t)

/\*

To handle reserved words, you should write a single rule to match an identifier and do a special name lookup in a function like this:

reserved = {

'if' : 'IF',

'then' : 'THEN',

'else' : 'ELSE',

'while' : 'WHILE',

...

}

\*/

}

* The tokens returned by lexer.token() are instances of LexToken. This object has attributes tok.type, tok.value, tok.lineno, and tok.lexpos. The following code shows an example of accessing these attributes:

# Tokenize

while True:

tok = lexer.token()

if not tok:

break # No more input

print(tok.type, tok.value, tok.lineno, tok.lexpos)

# Parser using YACC from PLY

import ply.yacc as yacc

#import the work you did in the lexer

from **lex.py** import tokens

abstract class cls\_Parser(yacc.yacc)

{

/\*

each grammar rule is defined by a Python function where the docstring to that function contains the appropriate context-free grammar specification. The statements that make up the function body implement the semantic actions of the rule. Each function accepts a single argument p that is a sequence containing the values of each grammar symbol in the corresponding rule.

def p\_expression\_plus(p):

'expression : expression PLUS term'

# ^ ^ ^ ^

# p[0] p[1] p[2] p[3]

p[0] = p[1] + p[3]

\*/

def p\_Select(p):

‘Select : SELECT columnList FROM dataSource WHERE filterCriteria’

pythoncode\_obj = pythoncode ()

pythoncodeString = pythoncode\_obj.Select\_Action(p[2], p[5] ,p[7])

}

# Business layer

abstract class cls\_pythoncode(object):

Select\_Action( columnList =nothing, dataSource, filterCriteria =nothing, groupBy=nothing, sortBy =nothing)