5/26/2019

End of Course Project

(20 credits)

BHARATH REDDY NAGASETTY-60027
VISHAL KUMAR SINGH -60030

Contents

Introduction:

Web Application Framework or simply Web Framework represents a collection of libraries and modules that enables a web application developer to write applications without having to bother about low-level details such as protocols, thread management etc.

Flask is a small framework by most standards, small enough to be called a “microframework.” But being small does not mean that it does less than other frameworks. Flask was designed as an extensible framework from the ground up; it provides a solid core with the basic services, while extensions provide the rest. Flask has two main dependencies. The routing, debugging, and Web Server Gateway Interface (WSGI) subsystems come from Werkzeug, while template support is provided by Jinja2. Werkzeug and Jinja2 are authored by the core developer of Flask. There is no native support in Flask for accessing databases, validating web forms, authenticating users, or other high-level tasks. These and many other key services most web applications need are available through extensions that integrate with the core packages. The developer has the power to cherry-pick the extensions that work best for the project.

TextX is a meta-language and a tool for building Domain-Specific Languages in Python. It's built on top of the Arpeggio PEG parser and takes away the burden of converting parse trees to abstract representations from language designers. From a single grammar description, textX constructs Arpeggio parser and a meta-model in run-time. The meta-model contains all the information about the language and a set of Python classes inferred from grammar rules. The parser will parse programs/models written in the new language and construct Python object graph a.k.a. the model conforming to the meta-model. The textX tool has support for error reporting, debugging, and meta-model and model visualization.

Virtiual Environments:

The most convenient way to install Flask is to use a virtual environment. A virtual environment is a private copy of the Python interpreter onto which one can install packages privately, without affecting the global Python interpreter installed in your system. Virtual environments are very useful because they prevent package clutter and version conflicts in the system's Python interpreter.

Creating a virtual environment for each application ensures that applications have access to only the packages that they use, while the global interpreter remains neat and clean and serves only as a source from which more virtual environments can be created. As an added benefit, virtual environ- ments don't require administrator rights. Virtual environments are created with the third-party virtualenv utility.

conda create -- name <env name> To activate the Virtual environment just type **activate <env name>.**

Basic Application Structure:

Initialization:

All Flask applications must create an application instance. The web server passes all requests it receives from clients to this object for handling, using a protocol called Web Server Gateway Interface (WSGI). The application instance is an object of class Flask, usually created as follows:

```
from flask import Flask
app = Flask(__name__)
```

The only required argument to the Flask class constructor is the name of the main module or package of the application. For most applications, Python's __name__ variable is the correct value.

The name argument that is passed to the Flask application constructor is a source of confusion among new Flask developers. Flask uses this argument to determine the root path of the application so that it later can find resource files relative to the location of the application.

Routes and View Functions:

Clients such as web browsers send requests to the web server, which in turn sends them to the Flask application instance. The application instance needs to know what code needs to run for each URL requested, so it keeps a mapping of URLs to Python functions. The association between a URL and the function that handles it is called a route. The most convenient way to define a route in a Flask application is through the app.route decorator exposed by the application instance, which registers the decorated function as a route. The following example shows how a route is declared using this decorator:

```
@app.route('/')
def index():
 return '<Statement>'
```

The previous example registers the function index() as the handler for the application's root URL. If this application were deployed on a server associated with the www.ex-ample.com domain name, then navigating to http://www.example.com on your browser would trigger index() to run on the server. The return value of this function, called the response, is what the client receives. If the client is a web browser, the response is the document that is displayed to the user.Functions like index() are called view functions. A response returned by a view function can be a simple string with HTML content, but it can also take more complex forms.

Server Startup:

The application instance has a run method that launches Flask's integrated development web server:

```
if __name__ == '__main__':
app.run(debug=True)
```

The __name__ == '__main__' Python idiom is used here to ensure that the develop- ment web server is started only when the script is executed directly. When the script is imported by another script, it is assumed that the parent script will launch a different server, so the app.run() call is skipped. Once the server starts up, it goes into a loop that waits for requests and services them. This loop continues until the application is stopped, for example by hitting Ctrl-C. During development, it is convenient to enable debug mode, which among other things activates the debugger and the reloader. This is done by passing the argument debug set to True.

Designing the Interface:

Creating Webpages:

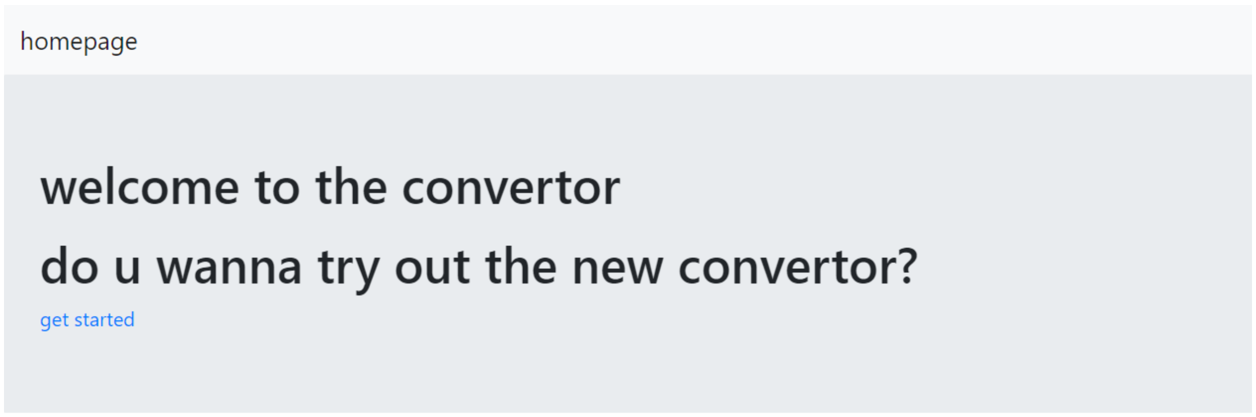
Two webpages namely home.html and display.html are made to ensure the user a visualization experience to translate the pseudocode to actual python grammar.

1. Home.html

```
<!DOCTYPE html>
<html lang="en" dir="ltr">
 <head>
<meta charset="utf-8">
<title>website base</title>
<link=rel="stylesheet"
ref="https://stackpath.bootstrapcdn.com/bootstrap/4.3.1/css/bootstrap.min.css" integrity="sha384-
ggOyR0iXCbMQv3Xipma34MD+dH/1fQ784/j6cY/iJTQUOhcWr7x9JvoRxT2MZw1T"
crossorigin="anonymous">
   <script src="https://stackpath.bootstrapcdn.com/bootstrap/4.3.1/js/bootstrap.min.js"
integrity="sha384-JjSmVgyd0p3pXB1rRibZUAYoIIy6OrQ6VrjIEaFf/nJGzIxFDsf4x0xIM+B07jRM"
crossorigin="anonymous"></script>
   <script src="https://code.jquery.com/jquery-1.12.0.min.js"></script>
   <script src="https://code.jquery.com/jquery-3.2.1.min.js"integrity="sha256-
hwg4gsxgFZhOsEEamdOYGBf13FyQuiTwlAQgxVSNgt4="crossorigin="anonymous"></script>
  </head>
<body>
   <nav class="navbar navbar-expand-lg navbar-light bg-light">
   <a class="navbar-brand" href="{{url_for('index')}}">homepage</a>
  </nav>
  <div class="jumbotron">
   <h1>welcome to the convertor</h1>
   <p></p>
   <h1>do u wanna try out the new convertor?</h1>
   <a href="{{url_for('convertor')}}">get started</a>
  </div>
</body>
</html>
```

The html document starts with a start tag Doctype to specify that it is a html page and followed by a CDN of java script and Bootstrap. BootstrapCDN is a public content delivery network where users of BootstrapCDN can load CSS, JavaScript and images remotely, from its servers.

The nav class is used for the navigation with the bootstrap effects and a hyperlink to home page is created where the user on clicking the homepage icon will be redirected to the main page. A style called jumbotron is used to enhance the quality of the webpage. A button ‘get started’ is present in the bottom of the page which is indeed linked to another page which displays the converted code.


homepage

welcome to the convertor

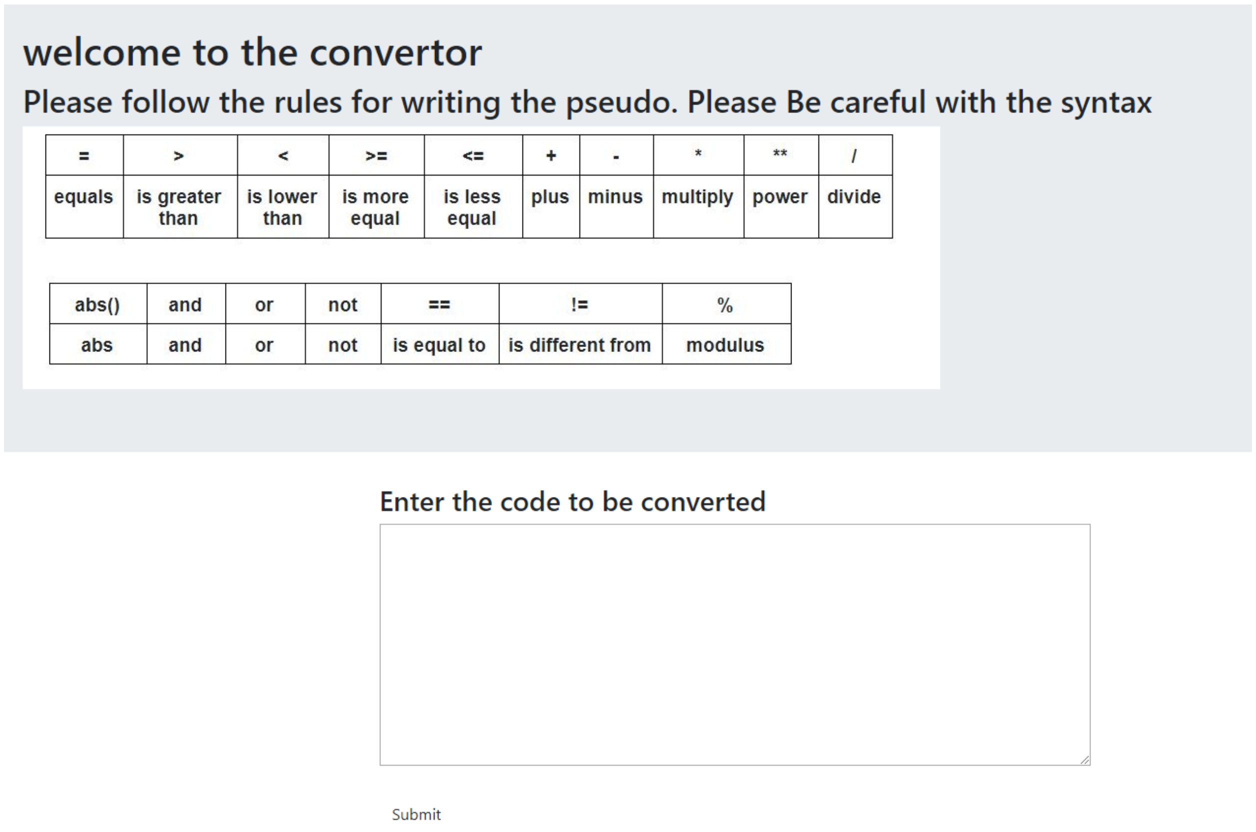
do u wanna try out the new convertor?

get started

2. Display.html:

```
<!DOCTYPE html>
<html lang="en" dir="ltr">
  <head>
    <meta charset="utf-8">
    <title>website base</title>
    <link rel="stylesheet"
href="https://stackpath.bootstrapcdn.com/bootstrap/4.3.1/css/bootstrap.min.css"
integrity="sha384-ggOyR0iXCbMQv3Xipma34MD+dH/1fQ784/j6cY/iJTQUOhcWr7x9JvoRxT2MZw1T"
crossorigin="anonymous">
    <script src="https://stackpath.bootstrapcdn.com/bootstrap/4.3.1/js/bootstrap.min.js"
integrity="sha384-JjSmVgyd0p3pXB1rRibZUAYoIIy6OrQ6VrjIEaFf/nJGzIxFDsf4x0xIM+B07jRM"
crossorigin="anonymous"></script>
    <link rel= "stylesheet" type= "text/css" href= "{{ url_for('static',filename='main.css') }}">
    <script src="https://code.jquery.com/jquery-1.12.0.min.js"></script>
    <script src="https://code.jquery.com/jquery-3.2.1.min.js"integrity="sha256-
hwg4gsxgFZhOsEEamdOYGBf13FyQuiTwlAQgxVSNgt4="crossorigin="anonymous"></script>
    <script type=text/javascript>
                                        $(function() {
                                          $('a#process_input').bind('click', function() {
                                                    $.getJSON('/background_process', {
                                                      proglang: $('textarea[id="proglang"]').val(),
                                                    }, function(data) {
                                                      $("#result").text(data.result);
                                                    });
                                                    return false;
                                          });
                                        });
</script>
  </head>
  <body>
    <nav class="navbar navbar-expand-lg navbar-light bg-light">
    <a class="navbar-brand" href="{{url_for('index')}}">homepage</a>
    </nav>
    <div class="jumbotron" >
     <h1>welcome to the convertor</h1>
     <h2>Please follow the rules for writing the pseudo. Please Be careful with the syntax</h2>
     <img src="../static/rules.JPG" alt="">
    </div>
    <div class='container'>
                  <h3>Enter the code to be converted</h3>
                        <form>
      <textarea id="proglang" rows="10" cols="100"></textarea>
      <br><br>
         <a href=# id=process_input><button class='btn btn-default'>Submit</button></a>
     </form>
     <br><br>
     <pre>
     <p id=result></p>
     </pre
     </div>
  </body>
</html>
```

The display page has the same visual effects as the display page but the processing of the display.html is what sets apart from the homepage. Here a JavaScript snippet is included for the asynchronous transfer of data between user and the server without the necessity for reloading the page. The display page consists of a textarea where the input is fed by the user and its passed through the java script which in turn passes to the convertor and then the results are taken from the file and displayed on the same page.


welcome to the convertor

Please follow the rules for writing the pseudo. Please Be careful with the syntax

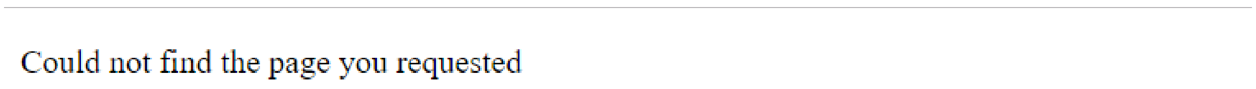
=	>	<	>=	<=	+	-	*	**	/
equals	is greater than	is lower than	is more equal	is less equal	plus	minus	multiply	power	divide

abs()	and	or	not	==	!=	%
abs	and	or	not	is equal to	is different from	modulus

Enter the code to be converted

3. Error Page:

An error page is created for just in case if the user enters different URL which is not a part of the web application.


Could not find the page you requested

Controller Flask Program:

```
from flask import Flask,render_template,request,jsonify
from source import inpo,clear
import subprocess
import os, sys
from subprocess import check_output
import time
import json
app= Flask(__name__)

@app.route('/')
def index():
  return render_template('home.html')

@app.route('/background_process')
def background_process():
  try:
    inp = request.args.get('proglang', type=str)
    inpo(inp)
    process()
    text=open('output.txt','r+')
    res=text.read()
    text.close()
    return jsonify(result=res)
  except Exception as e:
    return str(e)

def process():
  subprocess.call(['python','pseudo.py',],
    stdout=open('output.txt','wb'))



@app.route('/convertor',methods=['GET','POST'])
def convertor():
    return render_template('display.html')


@app.errorhandler(404)
def page_not_found(e):
    return render_template('404.html'),404


if __name__ == '__main__':
    app.run(debug=True)
```

The flask controller follows the same conventions mentioned in the flask documentation. The main aim of the program is to fetch the data from the webpage and store it in a text file so that the TextX file can use it for the text processing and pass the output to a text file from which it is passed to the display page.

A few packages needs to be imported like render_template,request and jsonify for the smooth processing of data. The function index acts a default page for the server. The main process happens in function background_process, where the input is being fetched from the textarea from html page and a subprocess is run with the python code. Two methods Get and Post are used in the program which are act as the connectors for connecting the data in the web server and the flask program.

TEXTX

Installation:

To install textx open the python terminal and run the following statement:

```
$ pip install textX
```

Note: Previous command requires pip to be installed.

To verify that textX is properly installed run:

```
$ textx
```

You should get output like this:

```
error: the following arguments are required: cmd, metamodel

usage: textx [-h] [-i] [-d] cmd metamodel [model]

textX checker and visualizer.
```

Visualization

TextX provides a great way to visualize the grammar created and it contains a inbuilt parser which automatically creates an Abstract syntax Tree, thus reduces the time to build a parser.

To get basic help type the following :

```
$ textx –help
```

You can check and visualize (generate a .dot file) your meta-model or model using TextX.

For example, to check and visualize a metamodel one could issue:

```
$ textx visualize <filename>.tx
```

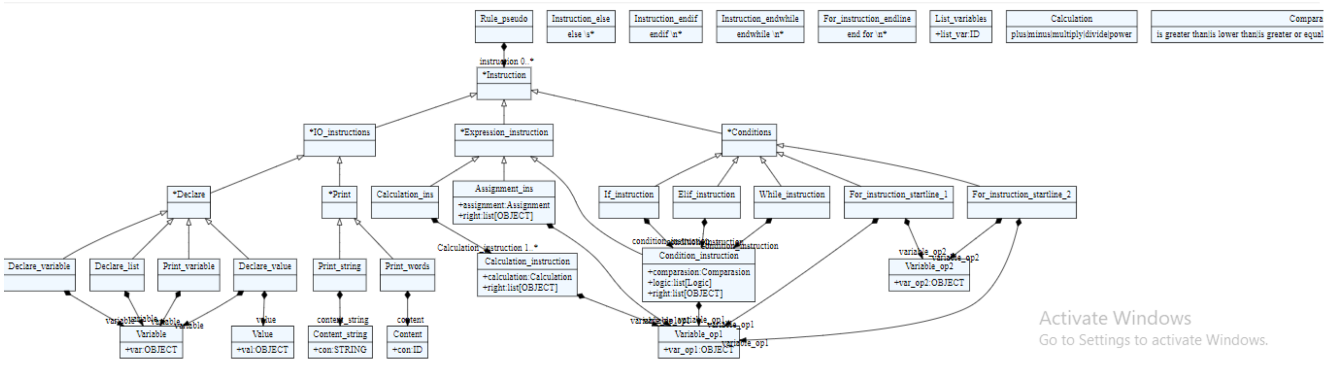
Example :

```
$ textx visualize pseu.tx

Meta-model OK.

Generating 'pseu.tx.dot' file for meta-model.

To convert to png run 'dot -Tpng -O pseu.tx.dot'
```

Create an image from the .dot file:

```
$ dot -Tpng -O robot.tx.dot
```

visit http://www.webgraphviz.com/ and paste the contents of the dot file on the website, it generates a tree like structure for the grammar. For example the tree would like below for the Pseudocode Project:



Visualizing the created model:

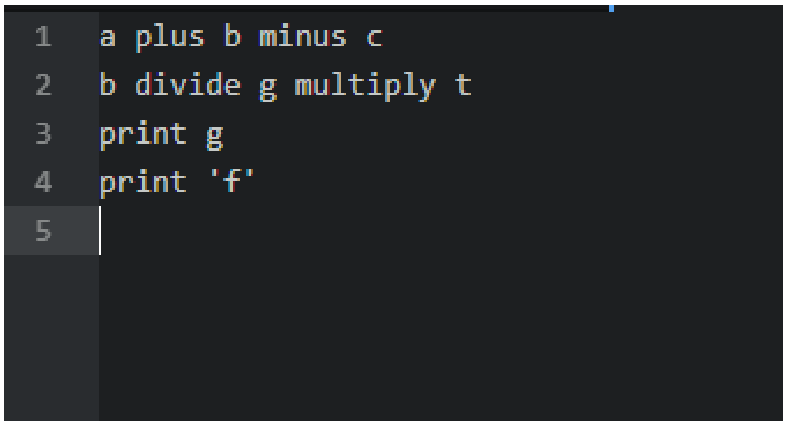
To check if the user input is processed by the grammar and to visualize the AST for the input put the following line into the terminal:

$ textx visualize pseu.tx input.txt

This will also create a dot file for the input file named:

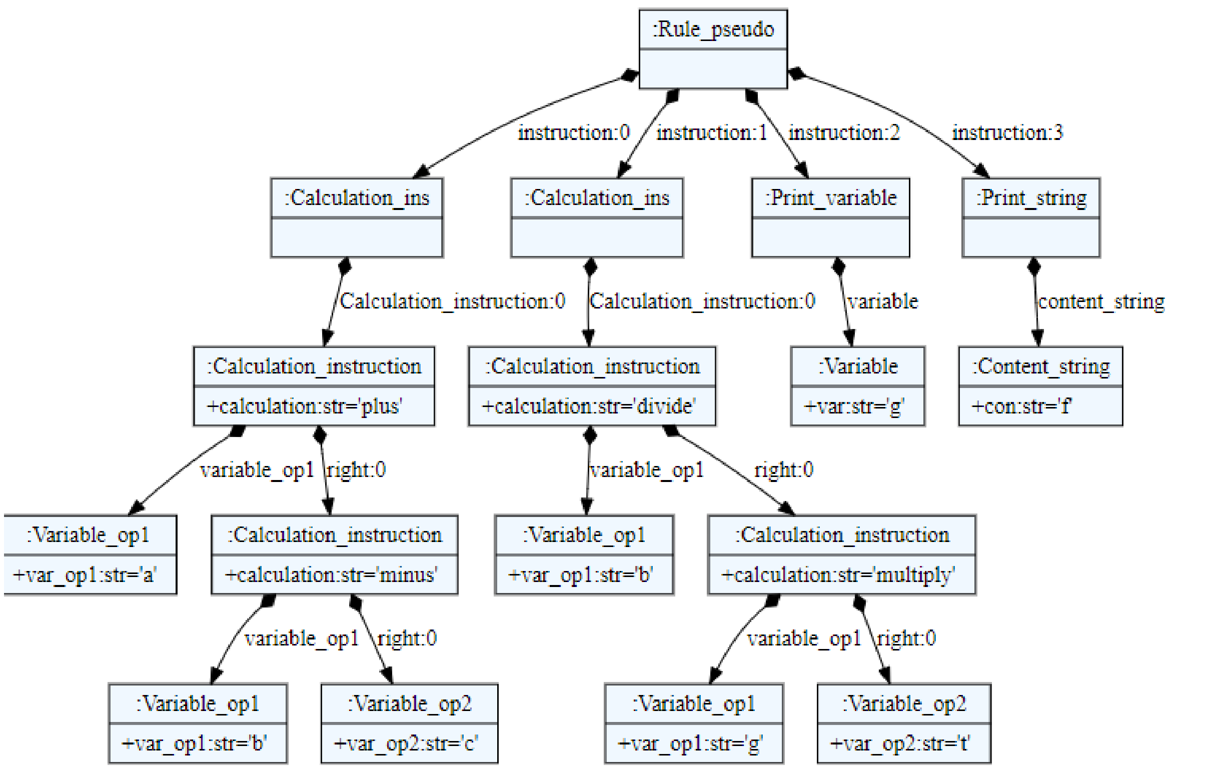
input.tx.dot

The contents of the input file are:-



```
a plus b minus c
b divide g multiply t
print g
print 'f'

```

The syntax tree can now be visualized using the input.tx.dot file .For the above input in would look like this:



To only check (meta)models use ‘**check**’ command:

$ textx check pseu.tx input.txt

If there is an error you will get a nice error report:

$ textx check pseu.tx input.txt

Meta-model OK.

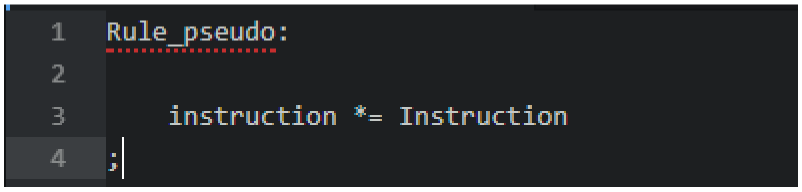
Error in model file.

Expected 'equal' or 'is greater than' or 'plus' or 'minus' or

at input.txt:(3, 3) => 'b plus c *'.

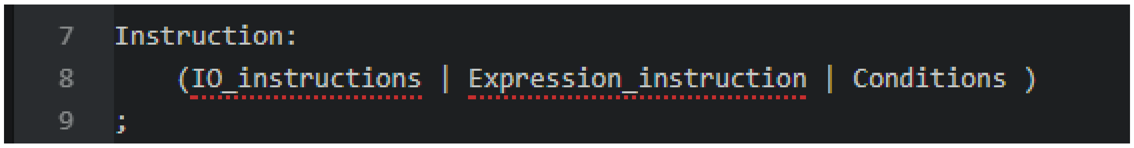
Grammar:

We start the grammar with an entity name ‘Rule_pseudo’ which has a rule as ‘Instruction’. ‘*= ‘ is used to represent zero or more repetitions along with the assignment of the rule to reference the main rule.



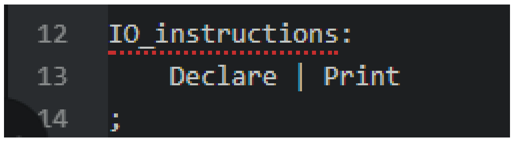
```
Rule_pseudo:

    instruction *= Instruction
;
```

Three main instruction are derived from the meta class Instruction. We separate them using ‘|’, which represents an ordered choice. This is used to match instructions from left to right. Any one of the below instructions can be chosen.



```
Instruction:
    (IO_instructions | Expression_instruction | Conditions )
;
```

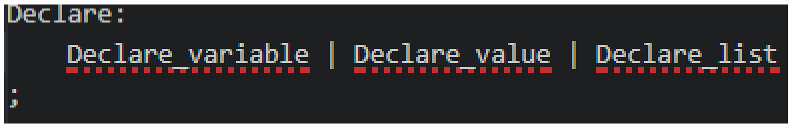
IO_instruction rule is derived from the Instruction rule. The IO_instruction can be taught of a child of the Instruction rule. IO_instruction contain Declare and Print Rules.



```
IO_instructions:
    Declare | Print
;
```

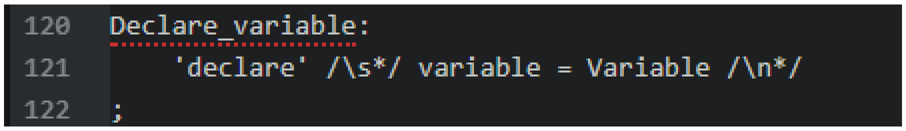
Declare rule:

Declare rule in turn gives rise to four different rules.



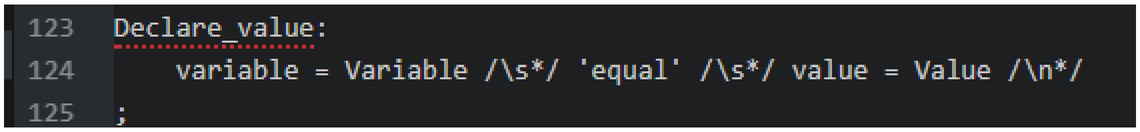
```
Declare:
    Declare_variable | Declare_value | Declare_list
;
```

The declaration of variable is followed by a keyword ‘declare’ which indicates the parser to go the declare variable rule. The keyword is the identity to the rule.



```
Declare_variable:
    'declare' /\s*/ variable = Variable /\n*/
;
```

To give a value to the declared variable, ‘equal’ keyword is used to assign value to the variable.

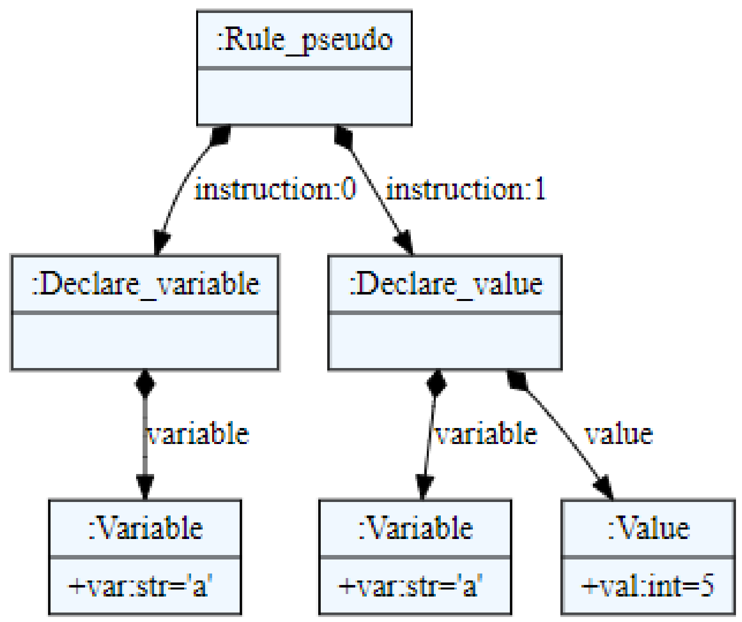


```
Declare_value:
    variable = Variable /\s*/ 'equal' /\s*/ value = Value /\n*/
;
```

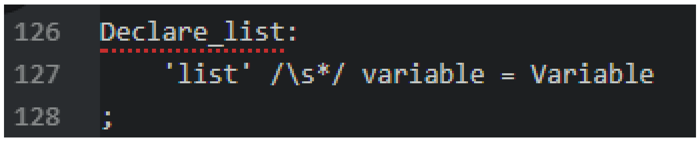
For example:

input =declare a

a equal 5

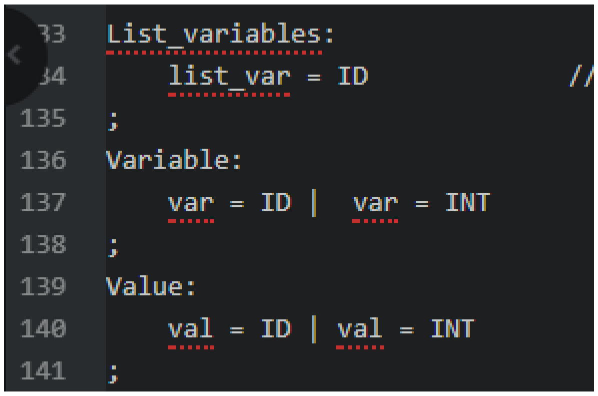


Declare list is designed to work with for loop, because for-loop works with lists and ranges.



```
Declare_list:
    'list' /\s*/ variable = Variable
;
```

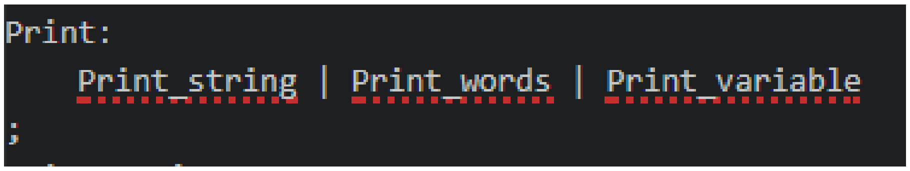
List variables is given to contain a list of variables. Variable and value define the type of input in which it will store its content ID OR INT, where ID can store alphanumeric values when we declare or print variables.



```
List_variables:
    list_var = ID           //
;
Variable:
    var = ID |  var = INT
;
Value:
    val = ID | val = INT
;
```

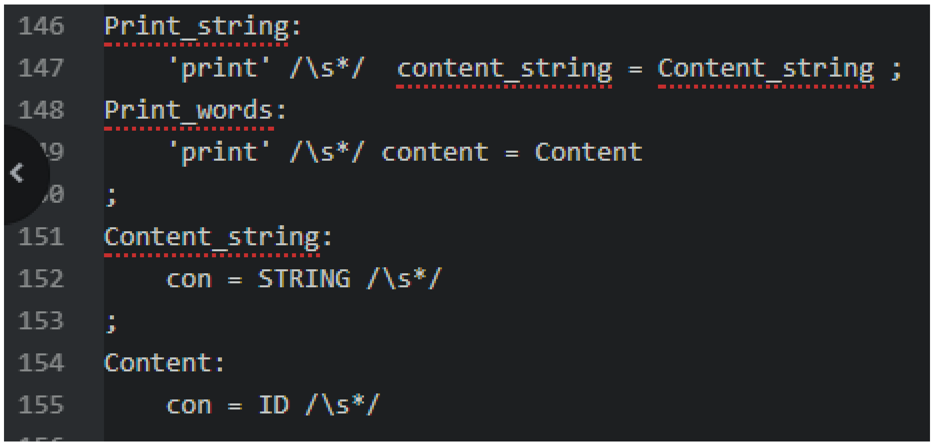
Print rule:

Print is sub divided into three independent rules namely Print_string, Print_words, Print_variable.



```
Print:
    Print_string | Print_words | Print_variable
;
```

Print string is given with a ‘print’ keyword which will take an input in string format and print words is also the same, but it takes input in ID format.

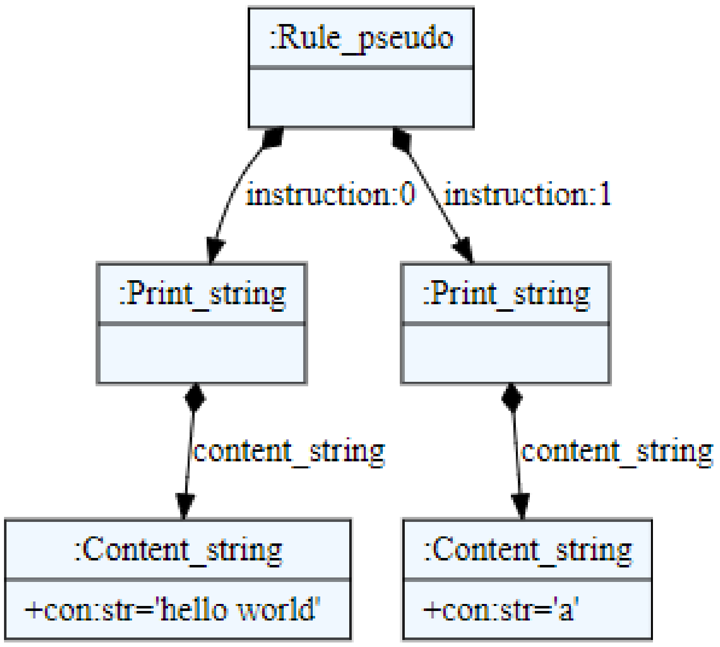


```
Print_string:
    'print' /\s*/  content_string = Content_string ;
Print_words:
    'print' /\s*/ content = Content
;
Content_string:
    con = STRING /\s*/
;
Content:
    con = ID /\s*/
```

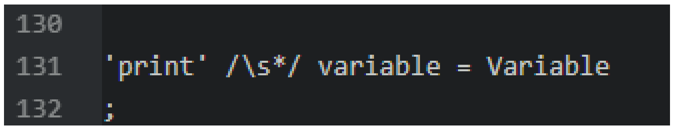
For example:

Input: Print ‘Hello world’

Print ‘a’



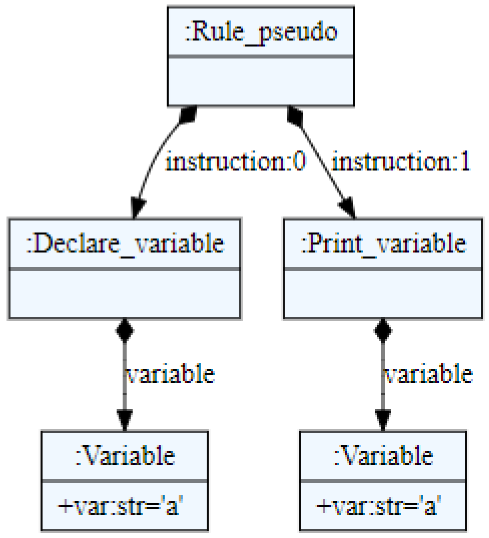
To print a variable 'print' keyword is used to define it.



```

'print' /\s*/ variable = Variable
;
```

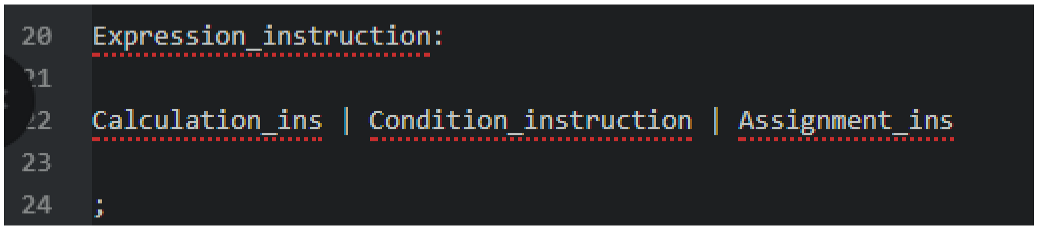
Input: declare a

Print a



Expression Instruction:

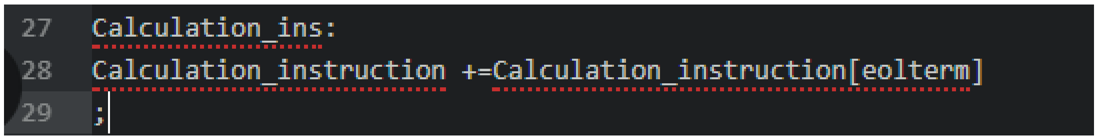
Expression Instruction is the second rule given in instruction class which has three rules like Calculation, Conditions and Assignment.



```
Expression_instruction:

Calculation_ins | Condition_instruction | Assignment_ins

;
```

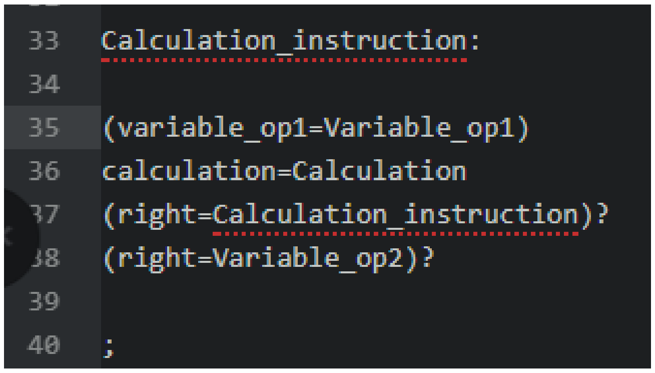
Calculation Instruction:

Calculation_ins, calculation_instruction is defined with '+' in between so that can occur one or more times and [eolterm] is used to mark the end of the line for the parser so that the next line makes a new command.



```
Calculation_ins:
Calculation_instruction +=Calculation_instruction[eolterm]
;
```

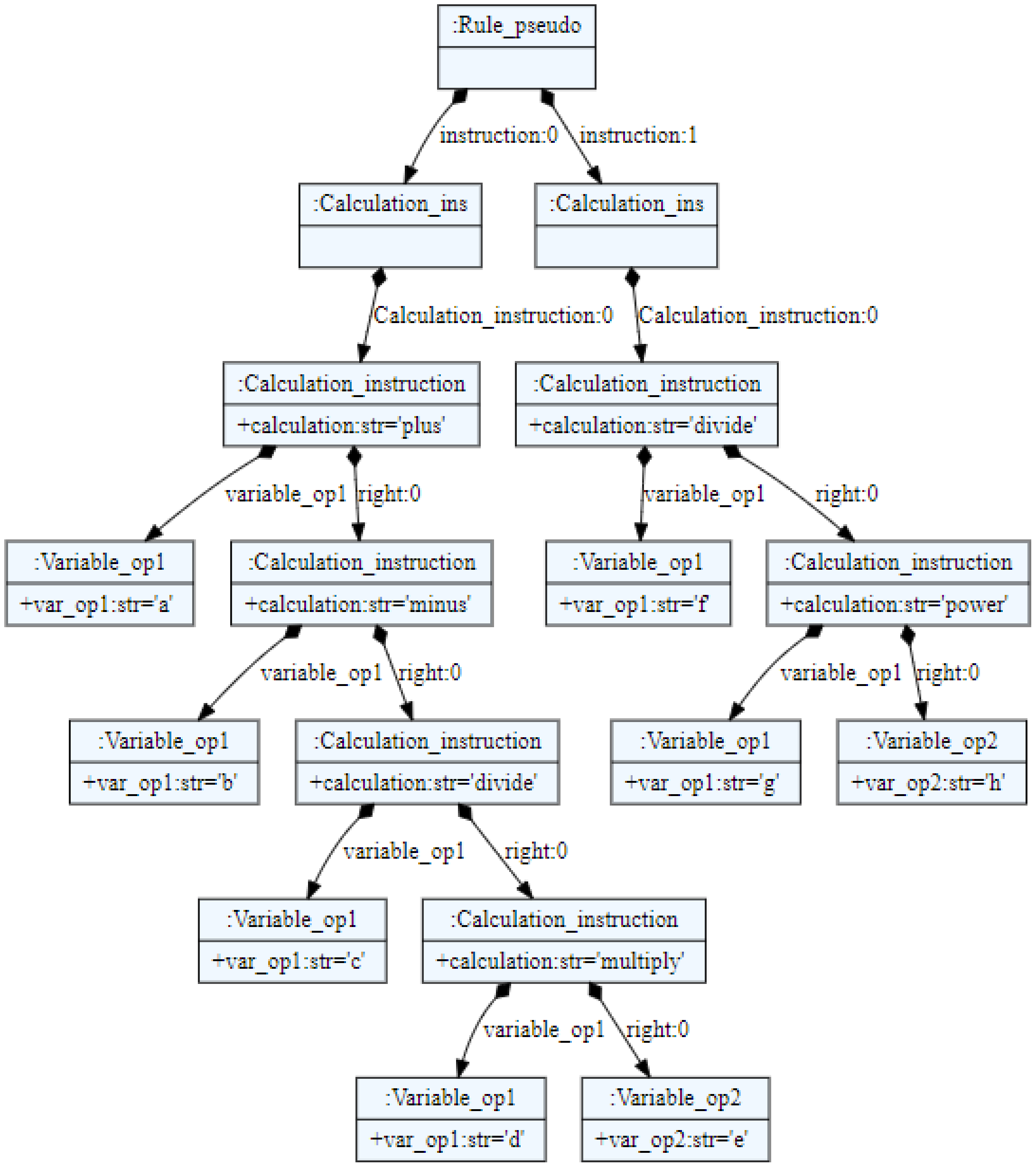
In this class we define the rules for the calculations to work for example: a plus b … Variable_op1 is a variable defined in the grammar which stores the variable . Then we have calculation operator which contains the operators like plus , minus , divide and so on. Then the right has calculation_instruction is again defined so as to form a recursive operation with multiple variables and conditions, it is ended with '?' to indicate the grammar that it is optional and can occur zero or one time. It is then followed by a right variable_op2 which is also defined in the grammar for a second variable input.



```
Calculation_instruction:

(variable_op1=Variable_op1)
calculation=Calculation
(right=Calculation_instruction)?
(right=Variable_op2)?

;
```

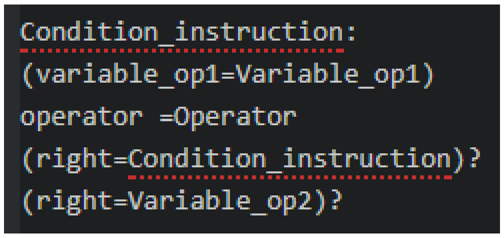
Input:- a plus b minus c divide d multiply e

f divide g power h



Conditional Instruction:

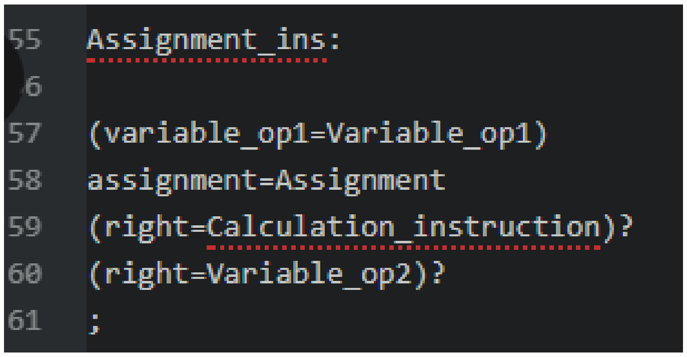
Condition_instruction is same as calculation instruction but it works on different operators which are comparison and logic operators. These are the rules which is used by if statements, while and for loops.



```
Condition_instruction:
(variable_op1=Variable_op1)
operator =Operator
(right=Condition_instruction)?
(right=Variable_op2)?
```

Assignment Instruction:

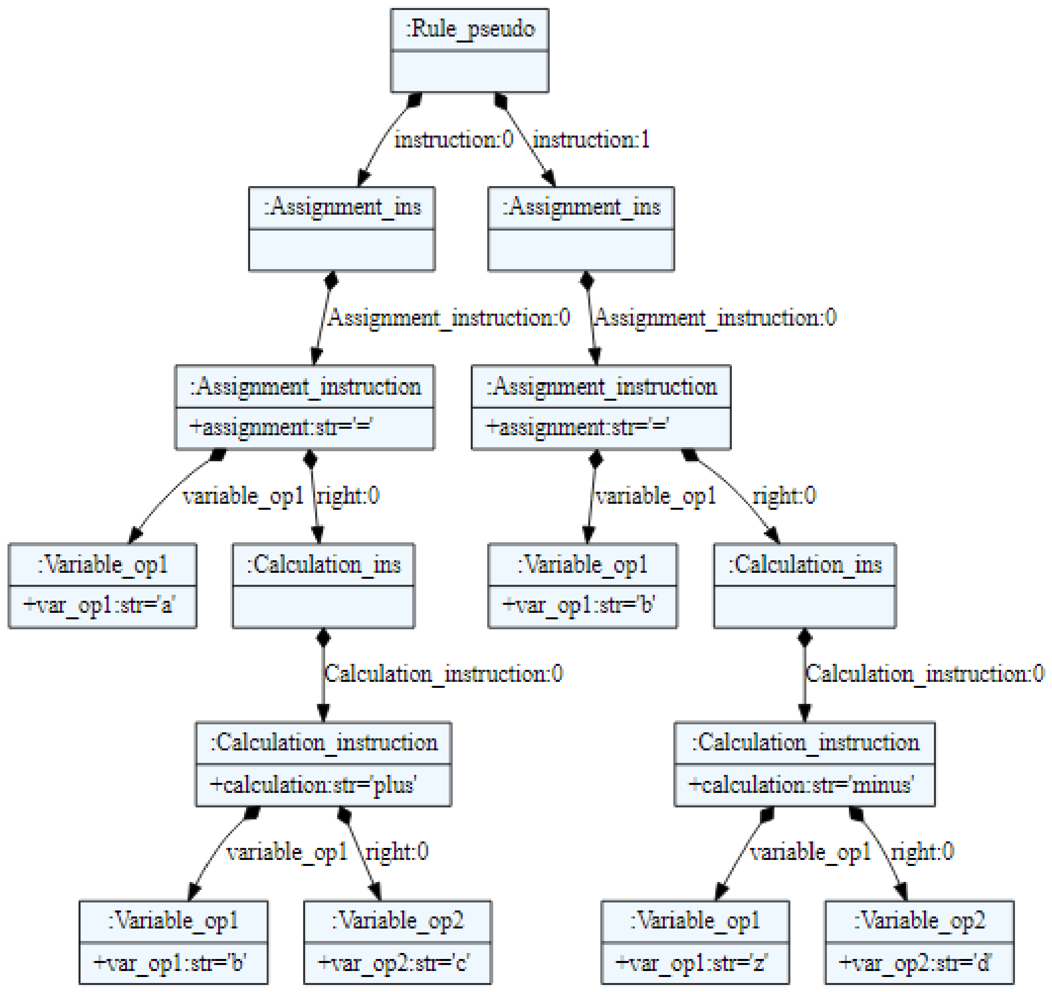
Assignment_ins is used to equate the operation like calculation or conditional to another variable.



```
Assignment_ins:

(variable_op1=Variable_op1)
assignment=Assignment
(right=Calculation_instruction)?
(right=Variable_op2)?
;
```

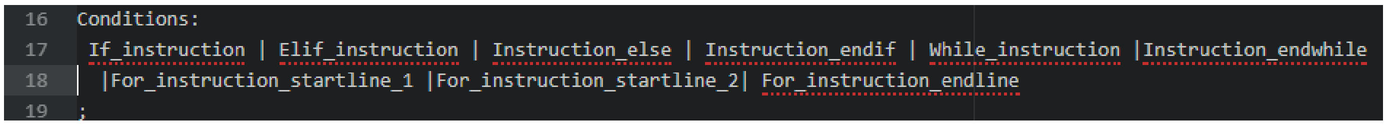
Input: a =b plus c

b= z minus d


:Rule_pseudo
instruction:0
instruction:1
:Assignment_ins
:Assignment_ins
Assignment_instruction:0
Assignment_instruction:0
:Assignment_instruction
+assignment:str='='
:Assignment_instruction
+assignment:str='='
variable_op1
right:0
variable_op1
right:0
:Variable_op1
+var_op1:str='a'
:Calculation_ins
:Variable_op1
+var_op1:str='b'
:Calculation_ins
Calculation_instruction:0
Calculation_instruction:0
:Calculation_instruction
+calculation:str='plus'
:Calculation_instruction
+calculation:str='minus'
variable_op1
right:0
variable_op1
right:0
:Variable_op1
+var_op1:str='b'
:Variable_op2
+var_op2:str='c'
:Variable_op1
+var_op1:str='z'
:Variable_op2
+var_op2:str='d'

Conditions:

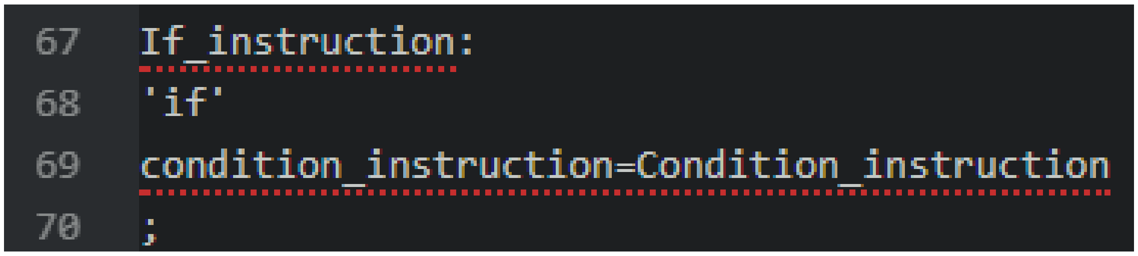
Conditions is the direct sub rule of Instructions which has subclasses for if , while and for conditions.



```
Conditions:
 If_instruction | Elif_instruction | Instruction_else | Instruction_endif | While_instruction |Instruction_endwhile
  |For_instruction_startline_1 |For_instruction_startline_2| For_instruction_endline
;
```

If Instruction:

If_instruction is defined by an ‘if’ keyword and followed by condition_instruction which has rules in it as defined above in condition_instruction.

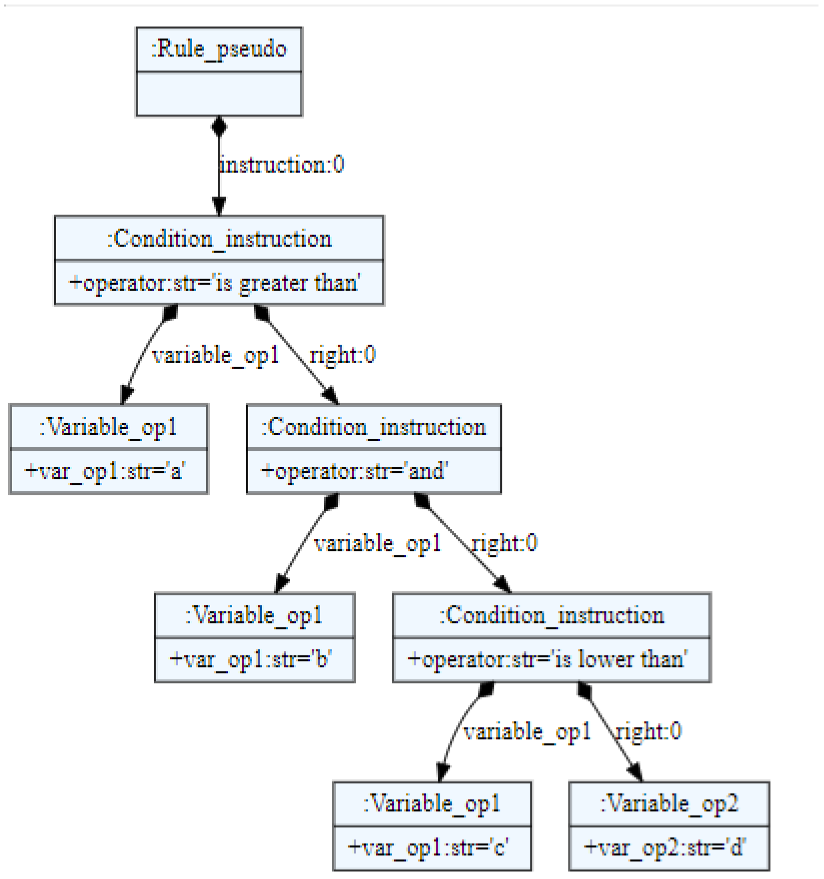


```
If_instruction:
'if'
condition_instruction=Condition_instruction
;
```

For example:-

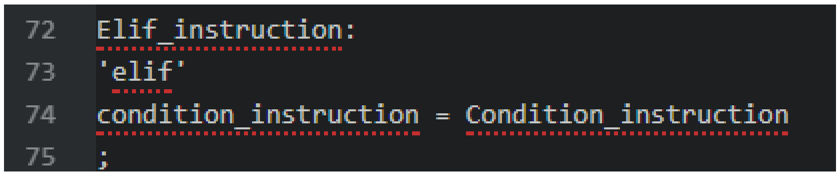
Input:- if a is greater than b and f is equal to

if c is lower than b



Elif Instruction:

Elif_instruction is same as if instruction but has ‘elif’ as the keyword.



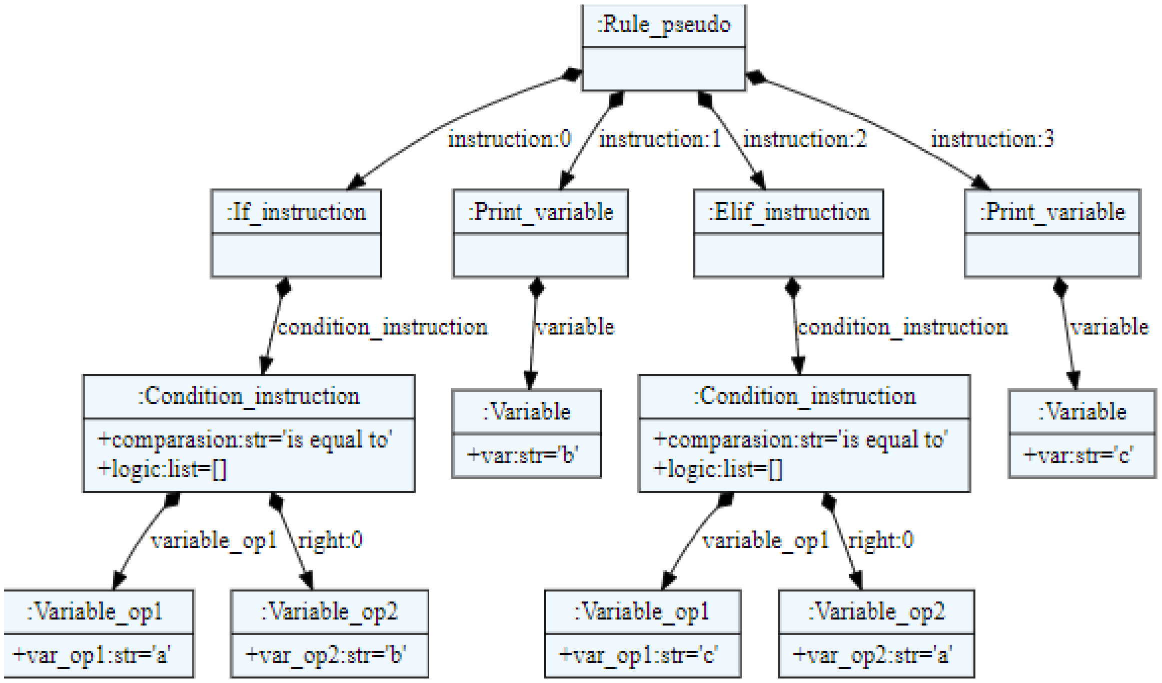
```
Elif_instruction:
'elif'
condition_instruction = Condition_instruction
;
```

For example:- Input:- if a is equal to b

Print b

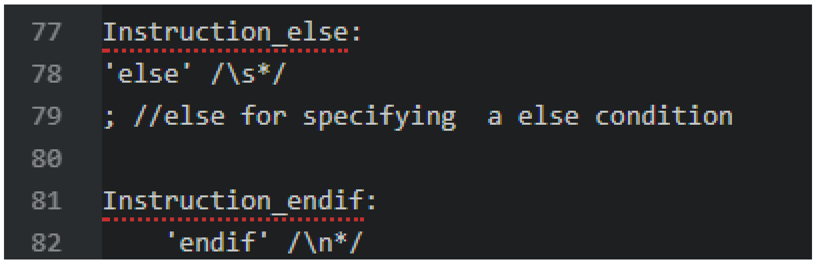
Elif c is equal to a

Print c



Else Instruction:

This has ‘else’ as the keyword to specify a else conditon. Instruction endif is to end the if condition.



```
Instruction_else:
'else' /\s*/
; //else for specifying  a else condition

Instruction_endif:
    'endif' /\n*/
```

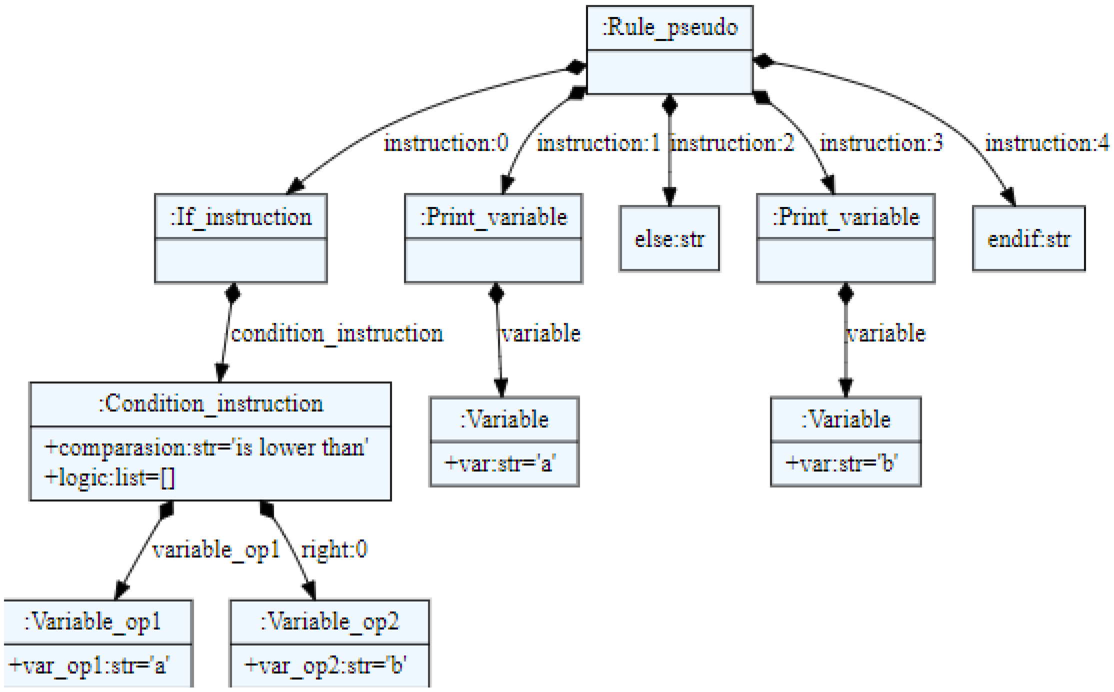
For example:- Input:- if a is lower than b

Print a

Else

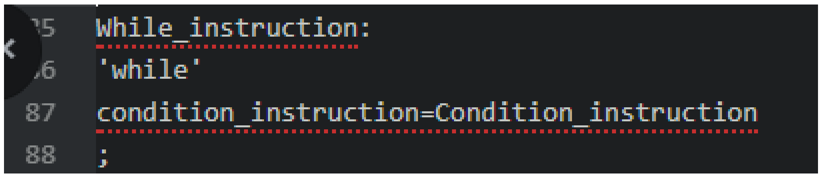
Print b

Endif

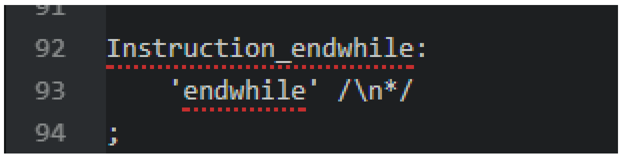


While Instruction:

While instruction is given by a ‘while’ keyword and then followed by a condition instruction to perform operations . Instruction endwhile is given with ‘endwhile’ keyword to end while condition.



```
While_instruction:
'while'
condition_instruction=Condition_instruction
;
```

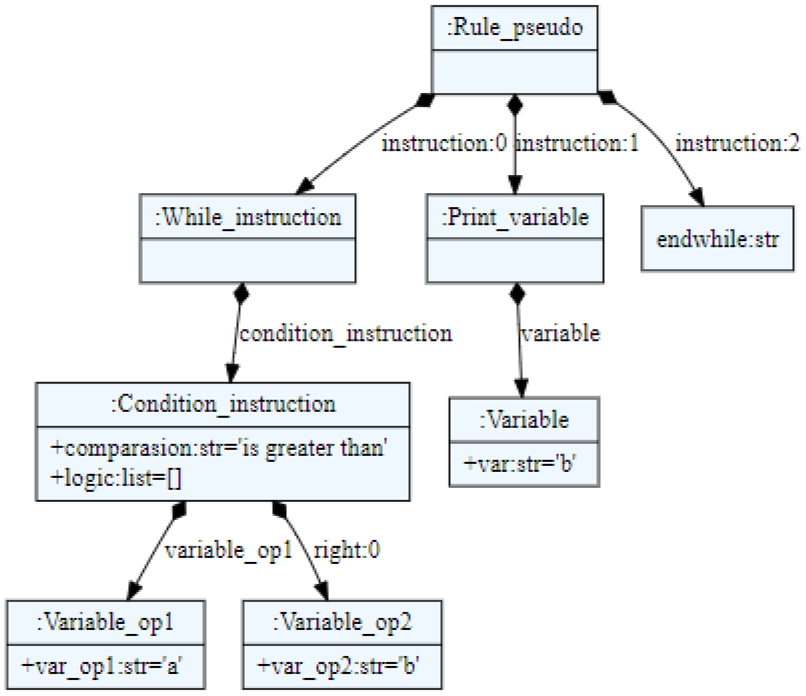


```
Instruction_endwhile:
    'endwhile' /\n*/
;
```

For example: Input: while a is greater than b

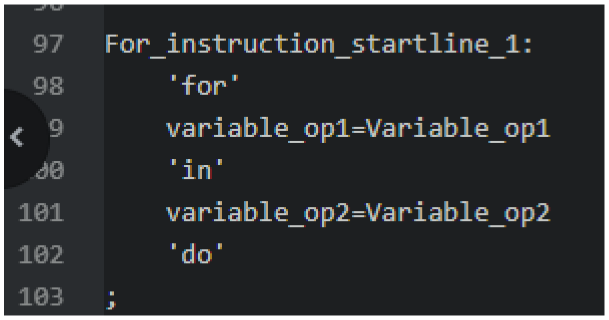
Print b

Endwhile



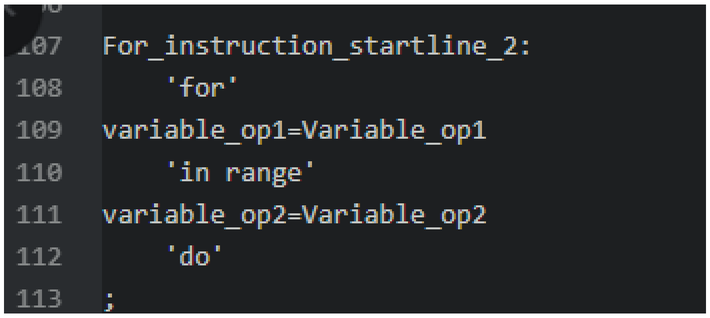
For Instruction:

For instruction is defined by 'for' keyword followed by a vairable_op1 defined in the grammar and then 'in' keyword followed by a variable_op2 to store the variables in then the 'do' keyword.



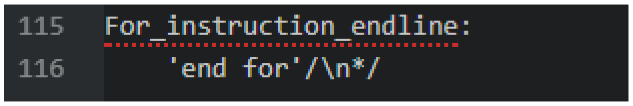
```
For_instruction_startline_1:
    'for'
    variable_op1=Variable_op1
    'in'
    variable_op2=Variable_op2
    'do'
;
```

For instruction 2 is also the same as for instruction 1, but with 'in range' keyword for this type of condition.



```
For_instruction_startline_2:
    'for'
variable_op1=Variable_op1
    'in range'
variable_op2=Variable_op2
    'do'
;
```

This is used to end for loop which is by the keyword 'end for'.



```
For_instruction_endline:
    'end for'/\n*/
```

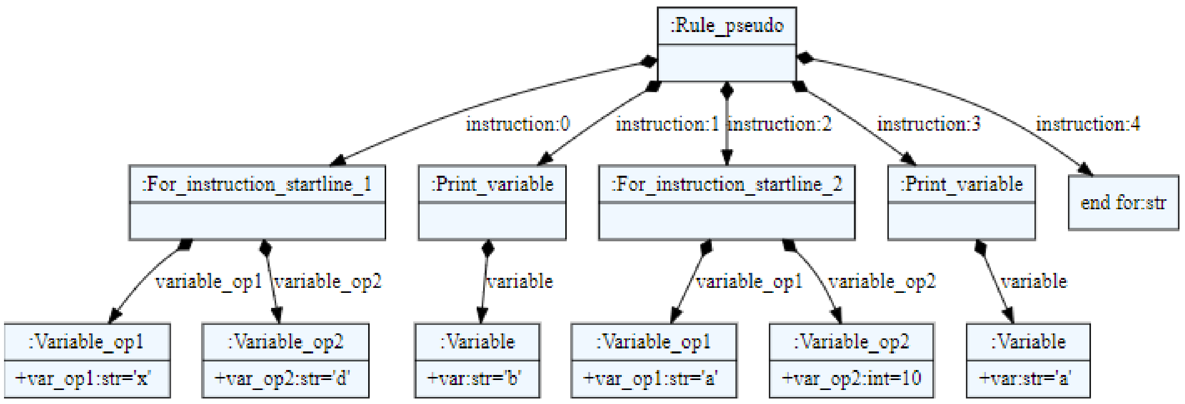
For example: input: for x in b do

Print b

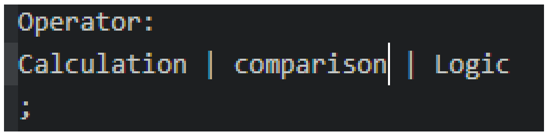
For a in range 10 do

Print a

End for



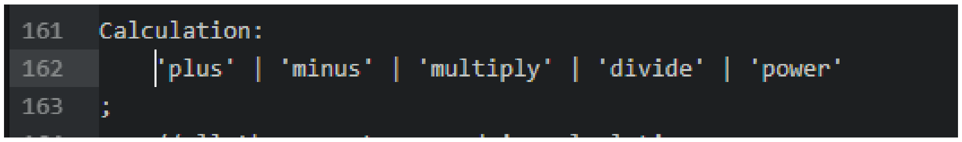
Operators:



```
Operator:
Calculation | comparison | Logic
;
```

Calculation Operator:

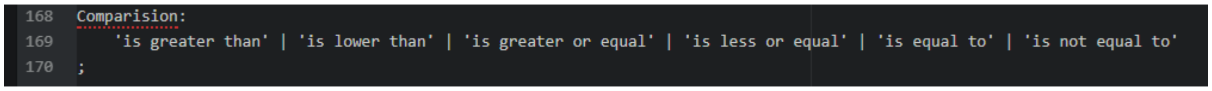
Calculation contains the operators used by calculation instructions for ex:- a plus b minus d. Here plus and minus are the calculation operators.



```
Calculation:
    'plus' | 'minus' | 'multiply' | 'divide' | 'power'
;
```

Comparison Operator:

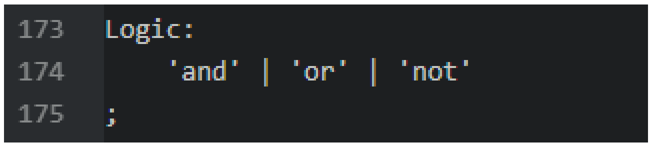
Comparison contains the operators used by condition instructions for ex:- a is greater than b is lower than d. Here is greater than and is lower than are the comparision operators.



```
Comparision:
    'is greater than' | 'is lower than' | 'is greater or equal' | 'is less or equal' | 'is equal to' | 'is not equal to'
;
```

Logic Operator:

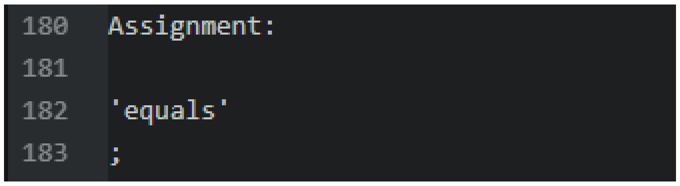
Logic contains the operators used by condition instructions for ex:- a is greater than b and f is lower than d. Here and is the logic operator.



```
Logic:
    'and' | 'or' | 'not'
;
```

Assignment Operator:

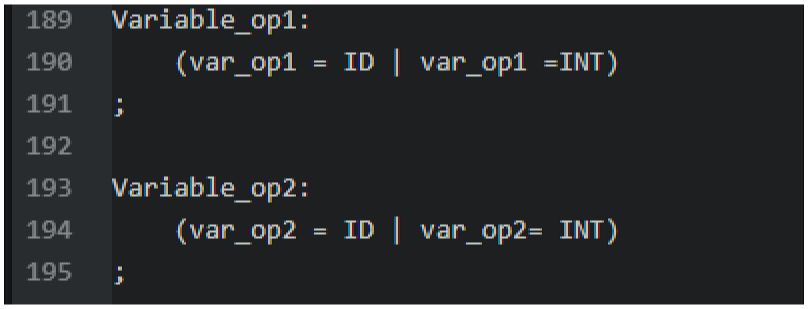
Assignment contains the operators used by assignment instructions for ex:- a equals b plus d. Here equals is the assignment operator.



```
Assignment:

'equals'
;
```

Variables:

These are the only 2 variables defined in the grammar which is used by expression instruction and conditons to store it’s variables in ID or INT format.



```
Variable_op1:
    (var_op1 = ID | var_op1 =INT)
;

Variable_op2:
    (var_op2 = ID | var_op2= INT)
;
```

References:

- http://textx.github.io/textX/stable/
- https://www.researchgate.net/publication/309705513_textX_A_Python_tool_for_Domain-Specific_Languages_Implementation
- https://www.eclipse.org/Xtext/documentation/
- http://flask.pocoo.org/docs/1.0/
- https://tomassetti.me/domain-specific-languages/
- https://ppci.readthedocs.io/en/latest/howto/toy.html
- https://esprima.readthedocs.io/en/latest/
- https://en.wikibooks.org/wiki/Introduction_to_Programming_Languages

GitHub:

`https://github.com/bharathreddy1997/End-of-course-Project.git`

Improvements:

- Implement operator precedence.
- Add python translation code to created grammar and link it with the website created.