Technical University of Košice
Faculty of Electrical Engineering and Informatics

Metamodel-based Parser Generator for Python

Master thesis

2022 **Bc. Sharoon Ilyas**

Technical University of Košice
Faculty of Electrical Engineering and Informatics

Metamodel-based Parser Generator for Python

Master thesis

Study Programme:	Informatika
Field of Study:	9.2.1. Informatika
Department:	Department of Computers and Informatics (DCI)
Supervisor:	Sergej Chodarev

Košice 2022 **Bc. Sharoon Ilyas**

Abstract

There are various tools available which can be used to develop parsers. The fact is, most of these tools are only concentrated on contrete syntax and does not focus on abstract syntax. In this thesis a metamodel based paraser generator in Python (PyCo) is created. The tool uses object-oriented approach to define a model-centered Domain Specific Language (DSL) in Python. Inspired by another similar tool in Java known as YAJCo, abstract syntax of the DSL can be expressed using object-oriented model and concrete syntax for the DSL can be expressed using decorators and field function of dataclasses Python library. Along with the construction of PyCo, this thesis fully describes the metamodel analysis, grammar generation and the creation of class instances using PyCo. The thesis also provides various DSL examples to learn about the use of PyCo.

Keywords in English

metamodel, programming, parser, Lark, Python, dataclasses

Abstract in Slovak

Existujú rôzne nástroje pre vývoj syntaktických analyzátorov. Väčšina z nich sa však sústreďuje iba na konkrétnu syntax a nezameriava sa na abstraktnú syntax. V tejto práci je vytvorený generator syntaktických analyzátorov pre jazyk Python založený na metamodeli (PyCo). Nástroj využíva objektovo-orientovaný prístup v Pythone k definícii doménovo-špecifických jazykov (DSL) pomocou modelu. Na základe inšpirácie podobným nástrojom pre jazyk Java, známym ako YAJCo, abstraktnú syntax DSL je možné vyjadriť pomocou objektovo-orientovaného modelu a konkrétnú syntax pomocou dekoratorov a funkcie field z knižnice dataclasses. Spolu s implementáciou nástroja PyCo, táto práca plne popisuje analýzu metamodelu, generovanie gramatiky a vytváranie inštancií tried pomocou PyCo. Práca tiež poskytuje príklady rôznych DSL, demonštrujúce použitie PyCo.

Keywords in Slovak

metamodel, programovanie, syntaktický analyzátor, Lark, Python, dataclasses

Bibliographic Citation

ILYAS, Sharoon. *Metamodel-based Parser Generator for Python*. Košice: Technical University of Košice, Faculty of Electrical Engineering and Informatics, 2022. 70s. Supervisor: Sergej Chodarev

65102

TECHNICAL UNIVERSITY OF KOŠICE

FACULTY OF ELECTRICAL ENGINEERING AND INFORMATICS

Department of Computers and Informatics

DIPLOMA THESIS ASSIGNMENT

Field of study: **Computer Science**

Study programme: **Informatics**

Thesis title:

Metamodel-based Parser Generator for Python

Generátor syntaktických analyzátorov založený na metamodeli pre Python

Student: **Bc. Sharoon Ilyas**

Supervisor: **Ing. Sergej Chodarev, PhD.**

Supervising department: **Department of Computers and Informatics**

Consultant:

Consultant's affiliation:

Thesis preparation instructions:

1. Analyze the topic of formal languages and parser generators.
2. Analyze YAJCo as an example of a parser generator based on a metamodel.
3. Design and implement a parser generator that uses Python classes as a definition of language metamodel.
4. Evaluate the generator by implementing several languages.
5. Prepare the documentation for language developers and developers of the generator.

Language of the thesis: English

Thesis submission deadline:

Assigned on:

TECHNICKÁ UNIVERZITA V KOŠICIACH FAKULTA ELEKTROTECHNIKY A INFORMATIKY -10-

.....

prof. Ing. Liberios Vokorokos, PhD.

Dean of the Faculty

Declaration

I hereby declare that this thesis is my own work and effort. Where other sources of information have been used, they have been acknowledged.

Košice, 22.4.2022 . . . . . .

Signature

Acknowledgement

At this point, I would like to thank my supervisor for his time and professional guidance during the solution of my final work.

I would also like to thank my parents and friends for their support and encouragement throughout my studies.

Last but not least, I would like to thank the gentlemen *Donaldovi E. Knuthovi* and *Leslie Lamportovi*for the typographic system LaTeX, with which I spent a number of unforgettable evenings.

Contents

List of Figures

Listings

Introduction

Developing a computer language and parser is an extremely important and extensively researched topic. Already there are many tools available to help us create domain specific languages and parsers, still this has been a rather difficult task and takes a lot of experience and effort [1]. This is one of the biggest reasons that some DSLs(domain specific language) are written based on existing languages and tools. [2]

There are many ways to define a domain specific languages, two of them are BNF (Backus Naur Formalism) and EBNF (Extended Backus Naur Formalism). Using BNF, we can define a set of rules for a language grammar. These rules can be used to find out if a language sentence is valid for that language grammar or not [3]. EBNF is just an extension of BNF for much simpler representation of the BNF. All context-free languages definition can be defined using BNF and EBNF.

Understanding the concept of metamodel is another crucial concept for this thesis. Although we can create a DSL using BNF and EBNF, creating a language using metamodel is yet another way of doing it [4]. By defining the DSL using metamodel, it is easier to specify the language syntax and semantics and is easier to understand.

To address the problem of creating complex domain specific languages, the metamodel approach [5] was used by a well developed tool name YAJCo(Yet Another Java Compiler compiler)[2].YAJCo tool handled the problem of language generation by using the object oriented programming approach of Java language. Instead of writing a parser from scratch and getting into deep complexity, language definitions could be defined in the form of classes in Java using YAJCo. This greatly helped to focus mainly on the language domain model.

In YAJCo, to specify abstract syntax of the language model, object oriented class implementation is enough, but to define the concrete syntax and semantics for the language, annotations of Java are used.[2]

Inspired by the fact that YAJCo tool is only limited to Java, the main goal of this thesis originated to build a library in Python which is similar to YAJCo and

also use object oriented approach. Similar to YAJCo tool's approach of using annotations, in Python, decorators will be used to help define the concrete syntax and semantics for the language.

Along with the implementation of decorators in Python, we can define the metadata in the fields of the Python classes. Thus, similar to YAJCo, the Python library will also be based upon metamodel structure to define the language.

The Python library will have the feature to generate class instances by processing the provided language sentence, same as YAJCo tool.

Implementation of such a tool in Python is going to be very beneficial, as a lot of complex applications are written in Python. This tool can help the developers to write domain specific language easily and perform complex tasks more efficiently.

To build and analyze this library, various already present parsing tools[6], books and research papers will be researched and evaluated to find what approaches the existing parsing tools are using and what improvement we can add to our Python library by examining them.

As YAJCo is also based upon an existing parser, the research into existing parsing tools is very important for the selection of the most appropriate parser for the Python library development.

The main objectives on which this thesis will be based up are:

1. Giving user the ability to define metamodel for DSL using Python classes.
2. Creating internal language model structure, which will store the metadata of classes and its fields.
3. Generating grammar from the internal language model.
4. Generating parse tree using provided language sentence and the existing parser LARK [7](parsing tool used for this thesis)
5. Creating class instances from generated parse tree with the help of internal language model.

1 YAJCo Tool

This section only focuses on the analysis of YAJCo tool and its specifications. The primary objective of this section is to explore the most common functionalities of JAJCo and find out how it is used and work.

1.1 YAJCo specifications

The core concept of YAJCo is very simple, YAJCo generate a language processor directly from the language metamodel. The language model of YAYCo is written in annotated Java classes. The general metamodel of DSL can be defined using just object oriented classes, but the concrete syntax and semantics are defined using annotations of Java language.[8]

As mentioned by the authors of [9], YAJCo uses annotations during the compilation to generate the parser related to the defined language. The parser would then analyze the language sentence according to the set of specified rules and results in instantiating abstract syntax classes which are used to produce a complete object model of the sentence which is being processed.

Specified by the authors Chodarev and Porubän in the research paper [2], YAJCo tool contains a Java annotation processor which is responsible for collecting annotations, that are attached to classes and their elements. YAJCo uses this language processor to create an internal model of the defined language and from it generate a parser. YAJCo has the tendency to find relationship between classes of the language model and can deduce the language syntax based upon this information. YAJCo deduce the missing information from the class annotations

Figure 1.1 taken from [2] to show a generic overview of YAJCo architecture.

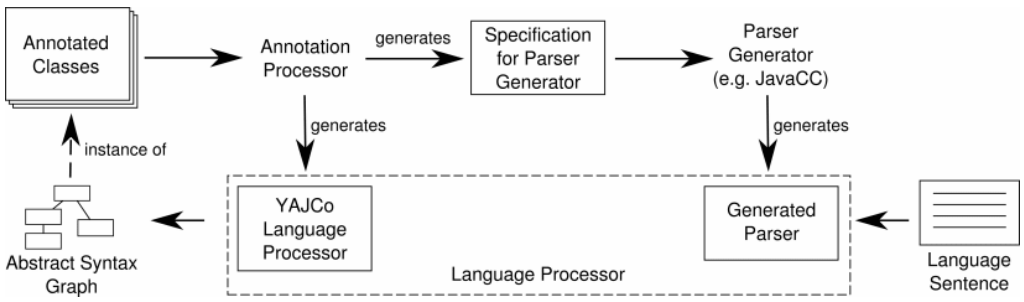

Figure 1.1: YAJCo architecture [2]

Figure 1.1 shows the complete architecture of YAJCo tool. It can be seen from the image, same as described earlier in this thesis, that YAJCo annotation processor takes annotated classes and creates specifications for Parser generator and YAJCo language processor. These then generate the parser which can be used to parse the language sentence.

YAJCo also supports other tools[2] beside just parsing. It has a pretty-printer feature which generates textual representation of the model, converting it back to the language sentence. This helps the tool to be capable of both serialization and de-serialization of the objects, if the objects have been provided in textual form.

Along with this another tool of YAJCo is the visitor class, which helps to simplify the traversing of the object graph.

The ability of YAJCo to incorporate the concepts from other languages makes it an extremely powerful tool. The language can be extended by incorporating concepts and ideas from other languages and connecting them through some relations.

1.2 How Language is defined in YAJCo tool

Defining a language model in YAJCo is very easy, this sub-section will specify briefly, how a simple YAJCo language metamodel can be created using object oriented classes along with class annotations in Java.

The main function of the language in listing 1.1 is to start with the id keyword which is then followed by small latin characters. The example below is taken from [10]

Suppose the language consist of concept known as SimpleIdentifier. A Java class is created with the same name. For specifying the root class, we will mark this class with annotation of @Parser. There are also other annotations in YAJCo tool, e.g @Before , @After. @Before is used to specify something that need to be before the identifier and language element. @After is used to specify something that needs to be after the identifier and language element. @TokenDef annotation is defined inside the @Parser annotation. In YAJCo, the name of the parameter is automatically mapped on the the relevant TokenDef

The method getIdentifier() is defined to act as an accessor of the identifier field.

Assuming that the language sentence that we wish to parse is 'id game'. The language model that will be used to parse this language sentence is described in listing 1.1.

Example 1 (taken from [10])

```
1
2  package mylang;
3
4  import yajco.annotation.*;
5  import yajco.annotation.config.*;
6
7  @Parser(tokens = @TokenDef(name = "ident", regexp = "[a-z]+"))
8  public class SimpleIdentifier {
9
10     private String identifier;
11
12     @Before("id")
13     public SimpleIdentifier(String ident) {
14         identifier = ident;
15     }
16
```

```
17      public String getIdentifier() {
18          return identifier;
19      }
20  }
```

Listing 1.1: YAJCo SimpleIdentifier example

If we now run mvn package, parser for our specified language will be created. However to run the parser, we need to create the instance of LALRSimpleIdentifierParser and import the defined class SimpleIdentifier and pass on the input to the parse method.

```
1  import mylang.SimpleIdentifier;
2  import mylang.parser.*;
3
4  public class MainClass {
5
6  public static void main(String[] args) throws ParseException {
7      String input = "id superman";
8      System.out.println("Going to parse: '"+input+"'");
9
10     LALRSimpleIdentifierParser parser =
11             new LALRSimpleIdentifierParser();
12     SimpleIdentifier simpleIdentifier = parser.parse(input);
13
14     System.out.println(
15         "identifier: "+simpleIdentifier.getIdentifier());
16     }
17 }
```

Listing 1.2: YAJCo runnning SimpleIdentifier example

When we parse our language sentence 'id game', the output of the program would be:

```
1  Going to parse: 'id game'
2  Parsed identifier: game
```

For example 2 in listing 1.3, we wish to create a language for controlling a robot, which takes a list commands of move or turn-left. To implement such kind of language we would have to design a metamodel in which, both Move and Turn-Left classes inherit from the Command interface. In the interface, we can define a method, which the inheriting classes can override.

According to example 2 in the listing 1.3, the language sentence should start with language element 'begin' and take a list of commands and then ends with language element 'end'

Listing 1.3 shows the definition of the metamodel to this Robot language specifications.

Example 2 (taken from [11])

```
1  import java.util.List;
2  import yajco.annotation.After;
3  import yajco.annotation.Before;
4
5  public interface Command {
6      public void execute();
7
8  }
9
10 public class Move implements Command{
11     @Before("move")
12     public Move() {
13     }
14
15     @Override
16     public void execute() {
17         System.out.println("going straight");
18     }
19 }
20
21 public class TurnLeft implements Command{
22     @Before("turn-left")
23     public TurnLeft() {
24     }
25
26     @Override
27     public void execute() {
28         System.out.println("turning left");
29     }
30 }
31
32 public class Robot {
33     List<Command> commands;
34
35     @Before("begin")
36     @After("end")
37     public Robot(List<Command> commands) {
38         this.commands = commands;
39     }
```

```
40
41      public List<Command> getCommands() {
42          return commands;
43      }
44
45      public void run() {
46          for (Command command : commands) {
47              command.execute();
48          }
49      }
50
51  }
52
53  public class Main {
54      public static void main(String[] args) throws Exception {
55          Parser parser = new Parser();
56          Robot robot =
57              parser.parse(begin move turn-left move end);
58          robot.run();
59      }
60
61  }
```

Listing 1.3: YAJCo Robot example

The example 2 in listing 1.3 is a bit more complicated as compared to example 1 in listing 1.1,

The thing to understand in the example 2 is that there is inheritance being used in the defined class metamodel, both Move and TurnLeft classes are inheriting from the Command interface, thus it creates a relation between Command interface, Move class and TurnLeft class. In both the Move class and the TurnLeft class, the execute method of the command interface is being overridden. Also both the Move and TurnLeft classes are defining @Before annotation specifying what these class expect before the language element.

In the Robot class, we are specifying a type List for commands variable. This variable will be used to store the commands list. The important thing to notice here is that the Robot class is implementing @Before and @After annotations to specify, what they actually expect before and after the list of commands.

By reviewing this much information of the class definitions and annotations, it can be inferred that the language sentence which is going to be valid for such a DSL have to start with "begin" and have to end with "end". In between there are only two types of inputs that can be entered, either turn-left or move.

Language sentence to be parsed

```
1    begin move turn-left move turn-left end
```

Listing 1.4: YAJCo robot control language sentence

Finally we see in our Main class, that we are calling the method called run of Robot instance. By examining the method we can see that it is calling the execute method of each command which can be either move or turn-left. When we go to the overridden execute method in the Move and TurnLeft classes, we see that in the Move class it will print "going straight" and in the TurnLeft class, it will print "turning left".

To represent the above relation, an overview diagram of the above description has been taken from [11].

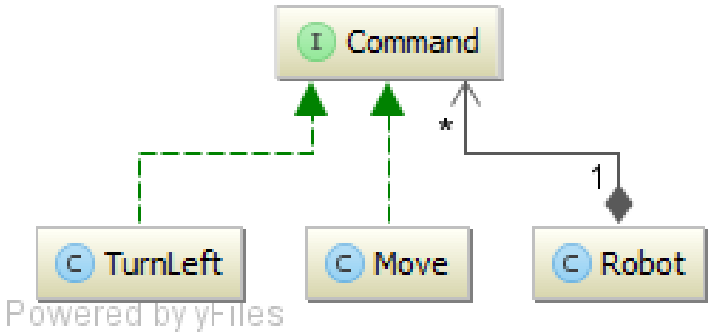

Figure 1.2: Simple Robot [11]

When the language sentence mentioned in listing 1.4 is parsed and the run method is executed.

The system output is:

```
1 goings straight
2 turning left
3 goings straight
4 turning left
```

1.3 Why JAYCo taken as inspiration

As specified above, YAJCo uses class based metamodel specification to generate a domain specific language, this is very similar to what we wish to achieve in Python. The way YAJCo create an internal language model to generate a parser

is another very interesting thing to implement in the Python library. This internal language model can help save a lot of metadata of the classes, relationships between classes and general information of the defined language metamodel. YAJCo is mainly taken as an inspiration because most of the desired steps are very similar to the main goal of this thesis.
Steps that will be similar to YAJCo.

1. define classes in Python.
2. create internal language model to generate grammar and class instances.
3. process generated grammar with pre-existing parsing tool (e.g LARK in our case).
4. generate parsing tree from parsing tool (e.g LARK in our case).
5. generate class instances using the parsing tree and internal language model generated in the second step.

Other features of YAJCo like pretty printer, inheritance and class relationship inference are also very interesting to research and experiment with. There are many different examples and research papers available related to this tool which makes it a primary choice to be taken as an inspiration.

2 Overview of existing parsing tools for the Python language

This section focuses mainly on the analysis of existing parsing libraries and tools for Python. Selecting an existing parser library for Python is one of the most crucial requirements of this thesis, as writing the parser for Python is not the main objective. By the analysis of various parsing tools for Python, a conclusive judgement can be driven in choosing the most appropriate parser for our needs.

There are various parsing tools already available in the market, that can be used to create a domain specific language. Studying and analyzing various libraries will not only help us to form a better judgement in choosing parser library, but it will also help to understand and have more new ideas that can be implemented in the Python parsing tool this theses is focusing upon.

The main goal of this process is to find a pre-existing parser in which:

1. Defining the grammar rules and syntax should be easy to read and understand(e.g EBNF).
2. There should option to view the parsed tree in graphical form.
3. Generating the parse tree is fast and consumes less memory.
4. There are multiple options to traverse the nodes of the tree.
5. There are filters available in the parser to filter out certain nodes of the tree, based on criteria.
6. There is good error reporting and handling available in the parser.

These requirements are considered in the selection of existing parser because it is much easier and efficient when a defined grammar is easy to read and understand(e.g EBNF).

With the availability of parse tree in a readable form, the analyzing and developing algorithms to create class instances takes less time.

Another major thing, which is considered, is the processing speed and memory consumption of the library. With the modern day technologies and trends, everything is related to speed and efficiency. If the parsing library will be fast and efficient, the metamodel based parser tool will also be fast and efficient.

Having multiple options to traverse the tree, opens up much more possibilities to improve or extend the features of the library in later stages.

Same as having multiple options to parse the tree, having filters available to find the desired results opens up more possibilities to improve and extend library features.

Having good error reporting and handling will help to debug and find solutions to reported errors more efficiently.

2.1 ANTLR

As mentioned by Parr & Quong in [12], ANTLR is a widely used parser and has been ported to the popular systems of UNIX, Macintosh and PC. ANTLR is developed with many industrial level collaborations.

ANTLR is strong and popular parsing tool which is good for reading, processing and executing the structured text. It also has a feature of translating structured text or binary files [13].

With ANTLR we can build many useful tools, such as configuration file readers, some kind of wiki markup reader, some JSON parser etc. Almost all forms of grammars can be processed with ANTLR except the indirect left recursion which indicate that the defined rules which reference to themselves must be direct [13].

With just the grammar ANTLR can generate the program, which is capable of confirming if the language sentence is valid for the defined language. ANTLR has a very sophisticated error reporting and recovery system. ANTLR also provides the compiler with the support for lexer, tree parser stages and a parser [14].

ANTLR has the capability to always recognize the valid input regardless of the complications of the defined grammar. ANTLR generates the parse trees automatically along with the tree walkers.

The Syntax text file in the figure 2.1 is used to write grammar syntax analysis rules and our custom lexical analysis. The Lexer generated by ANTLR is used for lexical analysis, which means the language sentence is decomposed into single words and processed by ANTLR. The Parser class is responsible to process various form of defined statements in EBNF format. The TreeParser class generates the parse tree.

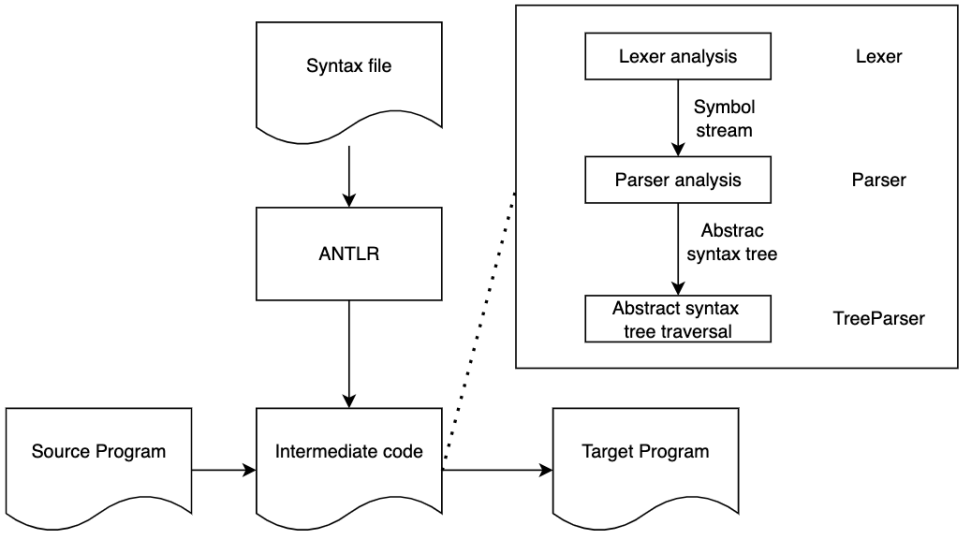

Figure 2.1: ANTLR architecture [14]

ANTLR has the capability to generate parsing code in Java, C++, Python Ruby etc [1]

Example below describe How ANTLR grammar is defined [15]

```
1
2  #ANTLR GRAMMAR DEFINITION
3
4  /*
5   * Lexer Rules
6   */
7
8  fragment A          : ('A'|'a') ;
9  fragment S          : ('S'|'s') ;
10 fragment Y          : ('Y'|'y') ;
11 fragment H          : ('H'|'h') ;
12 fragment O          : ('O'|'o') ;
13 fragment U          : ('U'|'u') ;
14 fragment T          : ('T'|'t') ;
15
16 fragment LOWERCASE  : [a-z] ;
17 fragment UPPERCASE  : [A-Z] ;
18
19 SAYS                : S A Y S ;
20 SHOUTS              : S H O U T S;
21 WORD                : (LOWERCASE | UPPERCASE | '_')+ ;
22 WHITESPACE          : (' ' | '\t') ;
23 NEWLINE             : ('\r'? '\n' | '\r')+ ;
```

[1] https://github.com/antlr/antlr4/blob/master/doc/getting-started.md

```
WHITESPACE          : ' ' -> skip ;

chat        : line+ EOF ;
line        : name command message NEWLINE;
message     : (emoticon | link | color | mention | WORD
              | WHITESPACE)+;
name        : WORD WHITESPACE;
command     : (SAYS | SHOUTS) ':' WHITESPACE ;
emoticon    : ':' '-'? ')'| ':' '-'? '(';
link        : TEXT TEXT ;
TEXT        : ('['|'(') ~[\])]+ (']'|')');
color       : '/' WORD '/' message '/';
mention     : '@' WORD ;
```

Listing 2.1: ANTLR Grammar

Command to run ANTLR

```
antlr4 -Dlanguage=Python3 Chat.g4
```

When this command is run in the project directory where Chat.g4 file is located, additional files are generated i.e newly generated parser and a lexer files. These files can be used to parse the expressions.

How the Grammar is interpreted

The grammar create rules that accepts a line. If we check the line representation, we can clearly see that the line should contain a name, a command with either SAYS literal or SHOUT literal, a ":" symbol, a message and end with new line.

The message, which is used, can be of anything , an emoticon, a link , color, and mention, The structure of all these rules are defined in the grammar.

Parsing language sentence in ANTLR

```
john SAYS: hello @michael this will not work
```

If we try to parse the above line with gui option of ANTLR [15], ANTLR tree will be produced (figure 2.2).

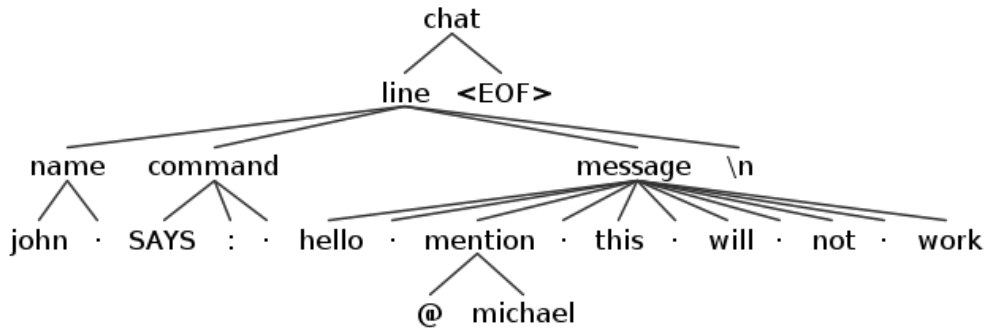

Figure 2.2: Graphical representation of ANTLR grammar [15]

2.2 pyParsing

pyParsing is a pure Python module that can be added to Python applications very easily. This library provides a set of classes to build up a parser from individual expression elements or complex variable syntax based expressions. These expressions can be combined with various operators such as + for combining one expression with another. To represent replication of expressions some pre-build classes of pyParsing library are used, such as OneOrMore, ZeroOrMore and Optional [16].

The basic form of steps that can be used for pyParsing are:

1. We can import names from pyParsing module.
2. Grammar can be defined using pyParsing helper method and pyParsing classes.
3. We can use the grammar to parse the input text.
4. We can process the results by parsing the input text.

Example of pyParsing [17]

```
1  import pyparsing as pp
2
3  greet = pp.Word(pp.alphas) + "," + pp.Word(pp.alphas) + "!"
4  for greeting_str in [
5              "Hello, World!",
6              "Bonjour, Monde!",
7              "Hola, Mundo!",
8              "Hallo, Welt!",
9          ]:
```

```
10      greeting = greet.parse_string(greeting_str)
11      print(greeting)
```

Listing 2.2: PyParsing example

In the above example, we are separating token by "," and in the end of the input the token "!" must be present, in order to make it a valid language sentence. When the above list is parsed using the for loop, the result returned will be

```
['Hello', ',', 'World', '!']
['Bonjour', ',', 'Monde', '!']
['Hola', ',', 'Mundo', '!']
['Hallo', ',', 'Welt', '!']
```

The class OneOrMore of pyParsing is used to define the multiplicity for the expressions. The class ZeroOrMore of pyParsing is used to define the multiplicity with a combination of the expression being optional. The class Optional of pyParsing is simple used to specify that the expression can be optional.

```
a= OneOrMore(expression)
b= ZeroOrMore(expression)
c= Optional(expression)
```

2.3 Parsimonious

Another parser written in Python is Parsimonious. Based upon the parsing expression grammar(PEGs), it takes simplified form of EBNF notation. Distinction between parsing and lexing is not drawn in PEG parsers and all the things are processed at once. This results in no lookahead limit, as there is in Yacc. Due to this, It is easier to write PEG grammars. With the help of caching, PEG take O(grammar size * text length) [18]. It has good error reporting and handling mechanism and use a very minimal and understandable Python code.

Example of Parsimonious [19]

```
from parsimonious.grammar import Grammar
from parsimonious.nodes import NodeVisitor

grammar = """\
entry = name sep gender? (sep age)?
sep = ws "," ws
ws = " "*
name = ~"[A-z]*"
```

```
  gender = "male" / "female"
  age = ~"[0-9]*"
  """


  class EntryParser(NodeVisitor):
      def __init__(self, grammar, text):
          self.entry = {}
          ast = Grammar(grammar).parse(text)
          self.visit(ast)
      def visit_name(self, n, vc):
          self.entry['name'] = n.text
      def visit_gender(self, n, vc):
          self.entry['gender'] = n.text
      def visit_age(self, n, vc):
          self.entry['age'] = n.text
      def generic_visit(self, n, vc):
          pass



  text = """\
  Bob, male, 26
  Kim,female,30
  Joe,male
  """

for line in text.splitlines():
    print EntryParser(grammar, line).entry
```

Listing 2.3: Parsimonious example

The example of Parsimonious mentioned above is very straight forward.

In the definition of the grammar string, we define entry which must contain a name and separator, gender which is optional or separator with age, which is also optional. WS represent space and can be as many spaces as possible. Multiple spaces are defined with the sign "*".

"name" is of the form of a regular expression which can be capital or small letters combination. "gender" only has two possible options , either male or female. "age" can be composed of any combination of numbers between 0 and 9.

From Parsimonious, NodeVisitor and Grammar is imported because we need to inherit from NodeVisitor class to implement our own custom node visitor. The EntryParser class which is inheriting from the NodeVisitor class takes two parameters to initialize, grammar and the text to be parsed. In the __init__ method

we call the Grammar class object from parsimonious and pass the grammar to it. This results in the internal parser generation for the specified grammar. Furthermore parse method is called with the text that needs to be parsed as a parameter. A parse tree in generated internally, of which, the nodes are visited by calling the visit method of the class. We also create the other methods called, visit_name , visit_gender , visit_age and generic_visit in the class. These methods are used to match the text entry at each node and store it in the dictionary.

We then use a for loop to call the EntryParser on each line in the language sentence text input. Finally when the text is parsed, the parsed dictionary object is returned with proper assigning of the relevant text.

The result of the above mentioned code would be.

```
{'gender': 'male', 'age': '26', 'name': 'Bob'}
{'gender': 'female', 'age': '30', 'name': 'Kim'}
{'gender': 'male', 'name': 'Joe'}
```

2.4 LARK

Lark is a modern Python parsing library, which can parse any form of text defined by the grammar provided to the Lark. It supports EBNF grammar syntax which makes it extremely easy to use. Lark provide a very flexible error handling and error messaging features which are extremely useful to debug the code. It has automated column and line tracking which can be used on both matched rules and tokens. Lark supports Python 2 and 3 completely and has its own standard library of terminals (numbers, strings, names, etc)

Lark has mainly tree types of parsing algorithms which we can choose, LALR, EARley, and CYK. LR syntax analysis method is a very useful technique for parsing a deterministic context free grammar [20]. With the use of LALR parser, number of states in the LR parser are tried to be reduced by merging of similar states [21]. EARley parser is able to parse any context-free grammar and is a general algorithm [22]. For CYK parsing algorithm we need the grammar to be in Chomsky Normal Form to proceed. CYK use algorithms with dynamic programming to find out if a string is valid for the language of the grammar.

Example of LARK

A simple example taken from LARK official documentation [23]

```
#Json Language Parsing
from lark import Lark
```

```
json_parser = Lark(r"""
    value: dict
           | list
           | ESCAPED_STRING
           | SIGNED_NUMBER
           | "true" | "false" | "null"

    list : "[" [value ("," value)*] "]"

    dict : "{" [pair ("," pair)*] "}"
    pair : ESCAPED_STRING ":" value

    %import common.ESCAPED_STRING
    %import common.SIGNED_NUMBER
    %import common.WS
    %ignore WS

    """, start='value')

```

Listing 2.4: Lark grammar definition and example

The example above, which is defining the grammar for json language is very interesting to look at. We are providing Lark library, a grammar in the form of a string. The string contains the grammar in EBNF form. The value can either be a dict, list, some ESCAPED_STRING , some SIGNED_NUMBER, "true" , "false" or "null".

Both SIGNED_NUMBER and ESCAPED_STRING are predefined regex to be used in LARK. It can be seen that at the end of the string SIGNED_NUMBER and ESCAPED_STRING are being imported.

We also imported WS (white space), which is also defined in the Lark parser. The %ignore WS indicates that the white space between or at the end of the tokens of the language sentence will be ignored.

For the definition of the the list, we are defining in the grammar, that the list should start with "[" and end with "]" and the list can have multiple value types in it, separated by ",".

"*" indicates, that multiple values can be inserted.

For the definition of the dictionary, a dictionary should start with "{" and end with "}". Inside the dictionary we are referencing pair, which is also defined in the grammar. A pair can have a some string and value (which can be of any type defined above) separated by ":". It can be seen that dictionary can have at least a single key-value pair or a list of key-value pairs.

Once this grammar string is passed as a parameter to Lark, Lark will create a language model from it, and store syntax and semantics related to grammar in it. If there will be wrong referencing in the grammar, or the structure of grammar would be bad, Lark has a very strong error handling approach available, which can let us know, what is wrong or missing in the grammar or the language sentence.

Parsing the language sentence

Continuing from the listing 2.4

```
'{"key": ["item0", "item1", 3.14]}'
```

```
text = '{"key": ["item0", "item1", 3.14]}'
tree=json_parser.parse(text)
print(tree.pretty() )
value
  dict
    pair
      "key"
      value
        list
          value "item0"
          value "item1"
          value 3.14
```

Listing 2.5: Lark language sentence processing

It can be seen in the output of the tree.pretty() that the parse tree of dictionary is generated, which has a key named as "key" and value as a list of three values("item0", "item1", 3.14).

2.5 Comparison of existing parsing tools

This section focuses on the comparison of the parser tools and libraries that we analyzed and studied. As the final goal of writing the metamodel based parser library is going to be written in Python, so more focus and preference will be given to that library which is easier to implement in Python and works as a Python module by default.

Many different aspects of the parser libraries will be taken into consideration.

1. What type of grammar syntax the parser library process
2. What type of parsing algorithms are available in the parser library

3. How fast is the parser library in processing the grammar and generating the parse tree.
4. What are the methods available in the library to traverse the parse tree.
5. Does the parser handle ambiguity in the defined grammar rules.
6. Does the parser process Context Free Grammar (CFG)
7. Does the parser keep record of lines and columns in the generated parse tree.
8. What types of outputs are available for parse tree (e.g pretty print).

2.5.1 Grammar definition & syntax of the parser library

If the language processing tools mentioned above in (ANTLR, Lark , PyParsing and Parsimonious) are compared together, it can be seen that defining the grammar rules are easier in ANTLR, LARK and Parsimonious. ANTLR, Lark and Parsimonious use EBNF, to specify grammar and syntax of the language. PyParsing on the other hand use combinator to define the language grammar.

2.5.2 Speed and Memory Comparison

If we compare the speed of ANTLR , Parsimonious and LARK, it can easily be concluded that LARK is the fastest of all. This can be demonstrated by the graph in figure 2.3 taken from the Lark's github repository. As demonstrated by the figure 2.4 taken from the Lark's github repository, Lark shows the least memory consumption as compared to other parser tools.

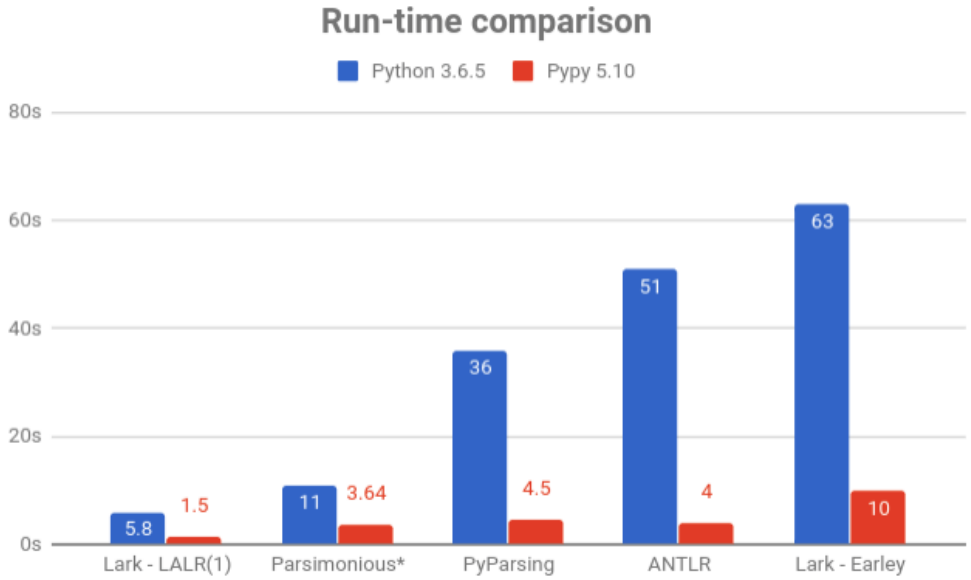

Figure 2.3: Runtime Comparison [7]

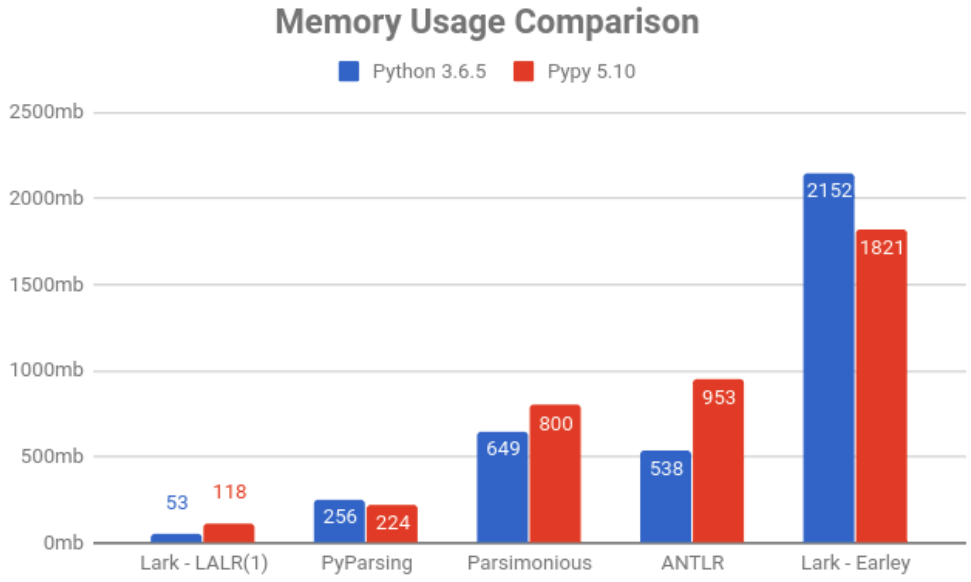

Figure 2.4: Memory Comparison [7]

2.5.3 Feature Comparison

A table taken from the git repository of Lark shows the features of Lark in comparison with the other Python parsing libraries.

Feature Comparison[7]

Library	Algorithm	Grammar	Ambiguity	CFG	line/col(ref)
Lark	EARley/LALR(1)	EBNF	YES	YES	YES
PyParsing	PEG	Combinators	NO	NO	NO
Parsimonious	PEG	EBNF	YES	NO	NO
ANTLR	LL(*)	EBNF	YES	NO	YES

2.6 Selected Parsing Tool

If we see the comparison of the tools mentioned above, it can be clearly concluded the choosing Lark is the most appropriate choice, as it is not only fast but it has so many other features, which opens up the possibilities of creating more feature in the metamodel based Python parsing library, which this thesis is focusing to build.

Lark provide automatic line and column tracking in the generated parse tree, that can be of use if needed for future extension of features.

It has flexible error handling and have good error responses with the indication of line where the error occurred to help debug the code faster.

One of the major features, for which LARK is selected for this thesis is that it is a pure Python module. This makes the integration and installation of Lark in existing Python code very easy.

Lark has many filter function to filter our the nodes of the parsed tree. It has many options by which we can traverse the nodes of the tree(i.e from top to bottom or from bottom to top).

The pretty print feature of Lark is also very beneficial, as the generated parse tree can be visualized in much more readable and understandable structure.

3 Design of Parser tool

3.1 Conceptual Overview and Design decisions

This sections explains the overall concepts that are going to be implemented in the creation of metamodel based parser named PyCo. Similar to YAJCo, the main idea is to generate a model of the language using Python classes with decorators to specify the the concrete syntax and specifications for the language. In PyCo additional metadata related to fields inside the class can be defined with dataclasses library function field. This provide a very cleaner way to specify the language model with concrete syntax and semantics.

Another major aspect which has been taken into consideration for PyCo is the declaration of classes, there is a lot of boilerplate code needed in Python to write classes. In order to deal with the boilerplate code, dataclasses library from Python is considered as a solution. With the implementation of dataclasses library, classes can be written with a lot of ease, in less lines of code and with simple syntax.

The use of dataclasses library enabled, an easier way for defining the metadata for fields defined in the classes. To assign metadata to the class field, all we have to do is use field function of dataclasses library and provide dictionary of metadata, to the metadata parameter.

Type checking in PyCo is another feature, which going to be in the library. When fields are defined inside a class, we can assign a field type to every defined field. To implement the type checking, typings library of Python is used.

User created classes can also become the types of the fields. This will make up the relationship between the classes much deeper. Concept of inheritance is another major feature which is going to be in the library. If abstract classes are used, all the classes which inherit from that abstract class forms a relationship. PyCo can find this relationship between classes and subclasses and process the structure accordingly.

When a language is defined using object-oriented and metamodel in PyCo,

PyCo first find the relationship between the abstract classes and subclasses that inherit from the abstract classes. During this process, PyCo go through each subclass and iterate over the fields of that class. While iterating, PyCo checks if the type of field is another user defined class or its one of the common types (list, dict, str, int, float).

PyCo process classes differently in these two scenarios. If the type is one of the common types, then it saves the metadata defined for each field accordingly.

On the other hand, if the type of a field is user defined class, PyCo will check the user defined class and follow the same process in recursive way to collect metadata. This metadata is converted into internal language metadata class instances and stored in internal language model. The internal language model is then used to create the grammar and class instances from the parse tree in the later stages of the PyCo.

PyCo will have the capability of creating class instance from the internal language metamodel and the generated parse tree. As mentioned earlier, after processing the classes, relationships between the classes and creating internal language model, PyCo is going to create a relevant grammar for the Lark parser.

Lark will process the grammar and Lark instance will be created to process the language sentence related to that grammar. When this Lark instance will parse a valid language sentence which satisfies the syntax and semantics of the defined grammar, Lark will produce a parse tree.

PyCo library deeply rely on Lark as a parser in order to process the grammar and generate parse tree from it.

With the help of internal language model and the generated parse tree by Lark, PyCo will be able to create the class instances.

The processes of internal language model generation and class instance generation are done in recursive way, which make PyCo very strong and powerful tool in processing deep level of inheritance and relationships between classes.

Easy to understand decorators have been defined in PyCo, which will help the user write complex language syntax in relatively easier and comprehensive way.

To write the code for this library, object-oriented approach is used. The main PyCo class when initializes with the parameters of starting base class and the user defined tokens dictionary, PyCo instance is created with pre-initialized language parser. Various methods can be called from this PyCo instance which will be discussed in detail in the coming sections.

3.2 Python dataclasses and type annotations

This sections explains about the dataclasses library and typing library of Python. Both of these libraries are extremely important in the usage of metamodel based parsing tool as metadata for the classes has to be defined by the use of these libraries.

3.2.1 dataclasses[24]

Python dataclasses library helps you to write less amount of code and gives additional features to define the metadata for individual class fields. Simply put, it helps to define classes with more functionality available out of the box [25].

If we wish to create a class, we would have to create a __init__ method and assign the constructor parameters to attributes.

```
class Game:
    def __init__(self, name, rating):
        self.name = name
        self.rating = rating
```

With the use of dataclasses, we don't need to create __init__ method. The @dataclass decorator will take care of everything.

The above example of class declaration can be written using dataclasses library as:

```
from dataclasses import dataclass

@dataclass
class Game:
    name: str
    rating: int
```

We can also assign the default values directly using dataclasses:

```
  from dataclasses import dataclass

  @dataclass
  class Game:
    name: str
    rating: int =78

```

Use of dataclasses helps you skip all the repeating methods declarations like `__str__ , __eq__ , __repr__`

Instances of dataclass can be represented as dictionaries and tuples

```
from dataclasses import dataclass, astuple, asdict
@dataclass
class Game:
    name: str
    rating: int

g = Person('Tomb Raider', 9)

print(astuple(g))
print(asdict(g)
```

```
('Tomb Raider', 9)
{'name': 'Tomb Raider', 'rating': 9}
```

Assign metadata to fields

To assign metadata to fields we can use the field function of dataclasses library.

```
from dataclasses import dataclass, field
@dataclass
class Game:
    name: str
    rating: int = field(metadata={"key":"value"})
```

3.2.2 Python type annotations [26]

The typing library for Python is used to give type hints and declare datatypes of variables, input functions and methods as well as output functions and methods.

Although declaring types makes the code understandable and precise, but in the case of PyCo type annotations are needed to assign relevant metadata to fields and create the corresponding class instances.

There are various types of type annotations that we are using from the typings library.

1. List
2. Dict
3. Optional

There are also general types available, like int, str , bool and float. These types are also used to define the datatype of the field of a class in PyCo.

Example of type annotation

suppose we need to create a class with the following attributes:

- name (store string value)
- age: (store integer value)
- height: (store float value)
- slovak: (store boolean value)
- subjects: (list of subject names)
- subject_scores: (dictionary of subject name in string and score in integer)
- over_all_grade: (optional field which stores a float value)

```
from dataclasses import dataclass
from typing import List, Dict, Optional

@dataclass
class Student:
    name: str
    age: int
    height:float
    slovak:bool
    subjects: List[str]
    subject_scores: Dict(str, int)
    over_all_grade: Optional[float]
```

Listing 3.1: Typing annotation example

The listing 3.1 shows, how the types are assigned to fields of a class.

3.3 How to use PyCo

This sections explains all the features and usage of PyCo library in depth. With the help of code snippets the usage and features of PyCo will be demonstrated. By going through the content of this section, anyone who wishes to use PyCo will easily be able to use the library.

3.3.1 PyCo Class Decorators

At the current development phase of the PyCo library, there has been only one decorator created called syntax In the @syntax decorator two parameters can be passed, before and after.

Parameter before is used to specify the literal that must be present before the defined grammar for the class onto which the decorator is applied.

Parameter After is used to specify the literal that must be present after the defined grammar for the class onto which the decorator is applied.

```
@dataclass
@syntax(before="[", after="]")
class Function():
  number: float=field(metadata={'token':'NUMBER_REGULAR_EXP)'})
```

Listing 3.2: PyCO decorator example

In the code snippet shown, the language sentence that can be parsed into parse tree will look like [x] , where x is some float number.

3.3.2 PyCo field metadata

For the metadata to be defined for the fields, a dictionary of key value pair is provided that can be applied. keys defined in the metadata dictionary are applicable according to the type annotation of the field.

Example

For the type of str, separator dictionary key is not going to be applicable and thus ignored.

```
@dataclass
class MetaExample():
  number: float=field(metadata={'token':'REGULAR_EXP_FOR_NUMBER)'})
class String:
  string: str=field(metadata={'token':'REGULAR_EXP_FOR_STRING)'})

@dataclass
class DictionaryExample():
  elements_dict: Dict[Sting, MetaExample] = field(
    metadata={'separator': ':',
         'element_separator': ',',
         'before': '{', 'after': '}'})

```

Listing 3.3: PyCO field metadata example

In the listing 3.3, the implementation of metadata related to each field can be seen. A parameter of metadata is passed to the field function of dataclasses library. The metadata parameter takes a dictionary key value pair.

Below is the list of dictionary keys that can be applied.

1. token (to define the regular expression of the input value)
2. separator (to define the list and dictionary separator)
3. element_separator(for dictionary to separate two key:value pairs)
4. before (to specify what should come before the language element)
5. after (to specify what should come after the language element)

There are numerous types that can be given to the fields in a class. Based on these types we can pass the relevant dictionary to the metadata parameter of the field function.

- For types *int* , *float* , *str*, *bool* **token** dictionary key in metadata of the field must be present. The keys **before** , **after** are also available, but they are optional.
- For types *List* **separator** dictionary key in metadata of the field must be present. **before** , **after** keys are optional.
- For types *Dict* **separator**, **element_separator** dictionary keys in metadata of the field must be present. **before** , **after** keys are optional.

If the type of some field is another user defined class, then the keys **before** , **after** are also available, but they are optional.

3.3.3 PyCo token definition

Tokens can be defined in a dictionary in Python, with key representing the name of the token and value representing the regular expression of the token.

```
from pyco.pyco_meta_parser import  NUMBER_REG, STRING_REG
token = {
    "STRING_TOKEN": STRING_REG,
    "NUMBER_TOKEN": NUMBER_REG,
    "BOOLEAN_TOKEN": '"true" -> true|"false" -> false'
  }
```

Both STRING_REG and NUMBER_REG are tokens which are defined in PyCo library. STRING_REG is used to represent all strings and NUMBER_REG is used to represent all signed numbers.

If we wish, we can also define our own regular expressions.

For the token BOOLEAN_TOKEN the value seems to be defined in a different way. It is important to know that if we wish to represent the boolean expression "true", we need to map it to true and respectively "false to false.

Example

If we wish to represent True as T and False as F

```
token = {"BOOLEAN_TOKEN": '"T" -> true | "F" -> false'}
```

In the language sentence then we must pass T to represent True in PyCo.

3.3.4 Types applicable to fields of a class in PyCo

PyCo parser expects the types to be defined for all the fields in a class. Types are very important as PyCo use these either to make relations with other related classes or to generate internal language model to later generate grammar and class instances.

There are various types available from the typing module in Python.

1. int
2. float
3. str
4. bool
5. Dict
6. List
7. User defined Classes

We will discus these types with code snippets example.

```
#Starting base class
@dataclass()
class JsonValue(ABC):
  pass


```

```
@dataclass
class JsonNumber(JsonValue):
  number: float = field(metadata={'token': 'REGULAR_EXP_FOR_NUM'})


@dataclass
class JsonString(JsonValue):
  string_value: str = field(metadata={'token': "
   REGULAR_EXP_FOR_STRINGS"})

  def __hash__(self):
    return hash(self.string_value)



@dataclass
class FieldExample(JsonValue):
  dict: Dict[JsonString, JsonValue] = field(
         metadata={'separator': ':',
         'element_separator': ',',
         'before': '{', 'after': '}'})


```

Listing 3.4: PyCO field types example

In the dict field of the class FieldExample, the type is Dict. As we know dictionary has a key value pair structure, so inside we define the types of the dictionary key and value by giving reference to other user defined classes JsonValue and JsonString. The class JsonString has a field which has a type *str*, which is one of types (str, int, float) and processed accordingly. On the other hand the class JsonValue is an abstract class, from which all other classes are inheriting, so the class JsonNumber and JsonString can be represented as a value for the dictionary.This means that the parsed tree and the class instances of this FieldExample can go to n^{th} depth.

3.3.5 Methods available in PyCO

There are three methods available in PyCo class instance.

1. parse_to_tree
2. create_instances

3. parse

The **parse_to_tree** method takes the language sentence as a parameter, first it passes the grammar string(generated during PyCo initialization) to Lark class instance. If Lark does not produce any error, then it passes the language sentence to the Lark parser's parse method. If the language sentence is valid, parse method returns a parse tree.

The **create_instances** method of PyCo just take the parse tree generated by **parse_to_tree** method and create instances of the class by using parsed tree and the internal language model(generated in the initialization of PyCo).

The last method parse takes language sentence as an input and returns the instance of the class by automatically performing the **parse_to_tree** method and **create_instances** method.

4 Implementation of PyCo

This chapter explains how the solution of PyCo has been implemented explaining all the structural code and algorithms that had been used. PyCo internal structure can be divided into tree main stages.

1. The internal language model(section 4.1)
2. The grammar generator(section 4.2)
3. The class instance generator(section 4.3)

Object oriented approach has been used in the structure of PyCo's internal algorithms and configurations. With object-oriented approach, PyCo has well organized and concise code. PyCo instance is created by passing the root class of the DSL and custom token dictionary.

4.1 The internal language model

The first and the foremost important stage of PyCo tool is the internal language model. When the metamodel of the DSL is defined and PyCo instance is initialized(details in appendix B.2), PyCo first start analyzing the classes and gather all their relationships and metadata. This metadata is converted into class instances and stored in a list based upon the types (details in appendix B.4).

Creation of internal language model is the most important task of PyCo because based upon this, the grammar and class instances are going to be created in the later stages.

This model stores all the information of classes and subclasses that are linked together through inheritance. When different types are defined, appropriate information is deduced by the helper functions to form the correct structure with correct data to be stored in the internal language model.

Figure 4.1, shows the class structure of internal language model and the class that is associated with it.

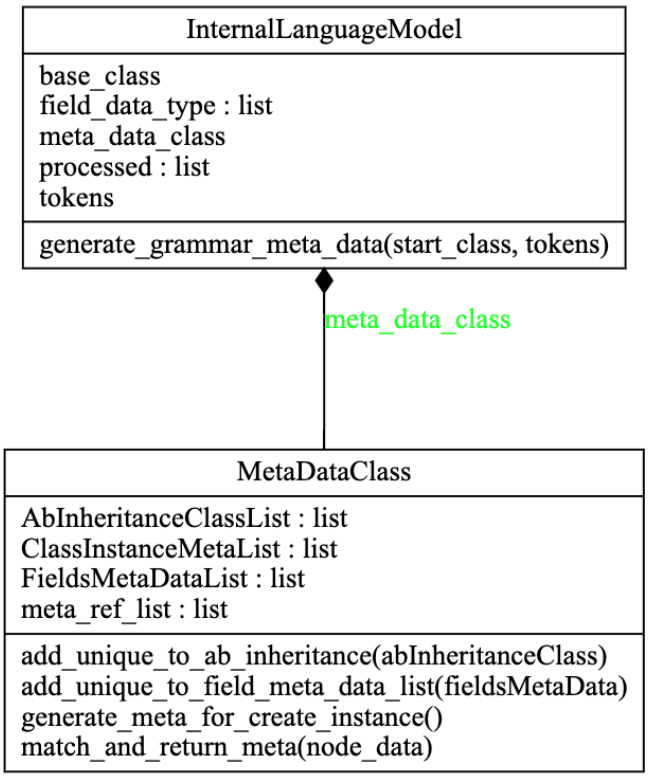

Figure 4.1: InternalLanguageModel & MetaDataClass Diagram

The figure 4.1 shows that the internal language model is dependent on the MetaDataClass. This is the main class which holds all the metadata information about the processed classes and fields.

Besides having the information about classes and fields, MetaDataClass is also responsible for holding the information about the relationships between the classes.

The attribute of MetaDataClass called AbInheritanceClassList, is one of the most important attribute as it stores the list of AbInheritanceClass(figure 4.2) instances.

The AbInheritanceClass hold the information about each class relationships(inheritance relationship).

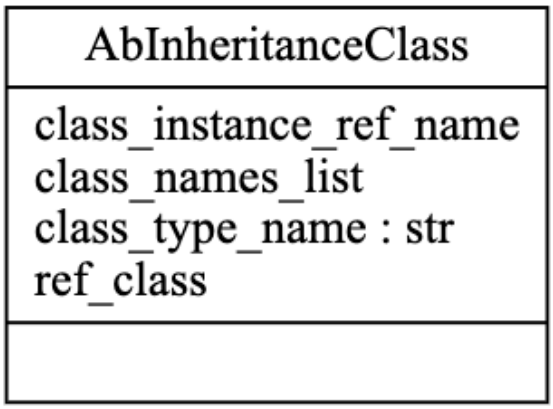

Figure 4.2: AbInheritanceClass(class)

AbInheritanceClass has an attribute called class_names_list, which stores a list of ClassMetaData instances.

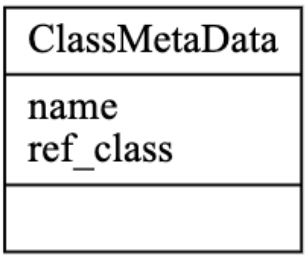

Figure 4.3: ClassMetaData(class)

The ClassMetaData class only stores the name of the class and an instance of the class.

The second most important attribute of MetaDataClass is FieldsMetaDataList, this attribute is responsible for storing a list of FieldsMetaData class instances. Each FieldsMetaData class instance contains the information about the fields defined in the classes, their metadata, the default value(if any) etc.

Figure 4.4 shows the structure of FieldsMetaData class.

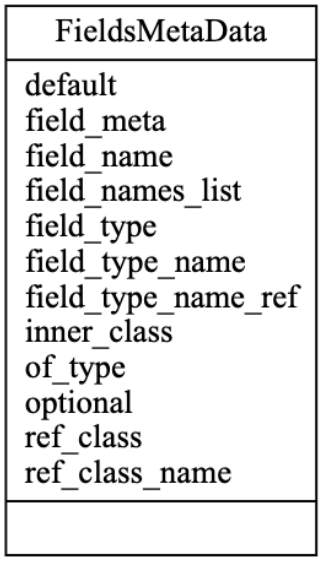

Figure 4.4: FieldsMetaData(class)

Last but the most important class is ClassInstanceMeta(figure 4.5). This class holds the combined information from the AbInheritanceClass and FieldsMetaData class. The MetaDataClass has an attribute called ClassInstanceMetaList. This attribute stores a list of ClassInstanceMeta instances. The ClassInstanceMetaList is used in the process of creating the instances of classes defined in the DSL metamodel.

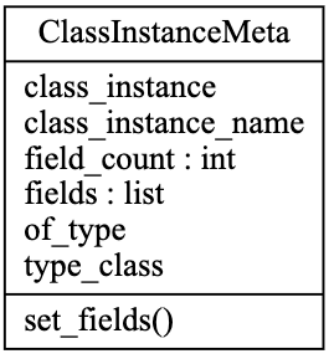

Figure 4.5: ClassInstanceMeta (class)

Each ClassInstanceMeta class has an attribute called fields, this attribute stores a list of Field(figure 4.6) class instances.

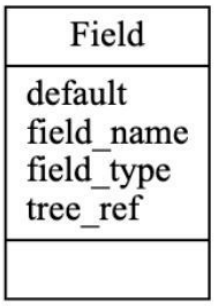

Figure 4.6: Field(class)

The Field Class instance stores the the useful information for the generation of class instances from the parse tree.

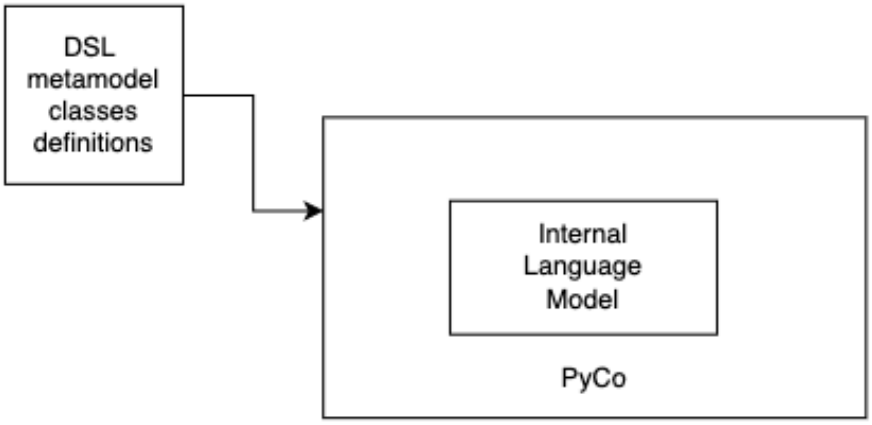

Figure 4.7: PyCo(Stage One)-Processing(internal language model)

The figure 4.7 shows the first stage of PyCo library. Creating internal language model is the first stage of the whole process and this internal language model is generated with the help of all the classes discussed in this section.

4.2 The grammar generator

The second important stage of the PyCo library is the grammar generator. This section is responsible of forming grammar strings by iterating through the list of metadata classes generated by internal language model 4.1

The generate_grammar method of the grammar generator takes the metadata list from the internal language model and generates the grammar in EBNF form.

FieldsMetaDataList is filtered in this method to find the metadata related to the fields for which the grammar string is being generated for.

This metadata is used to create the appropiate grammar string and is concatenated to the attribute of generated_grammmar of the Grammar class.

AbInheritanceClassList is also iterated in this method to generate grammar string, which represent inheritance relation between the classes.

When the grammar is fully generated, then we call two functions to finalize the grammar. Using first function, we concatenate the tokens that we passed during PyCo initialization to the generated grammar string. Using second function, we concatenate some Lark specific imports to ignore white space and add Lark's predefined regular expressions.

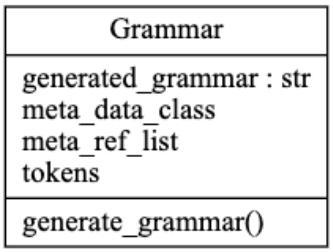

Figure 4.8: PyCo Grammar class

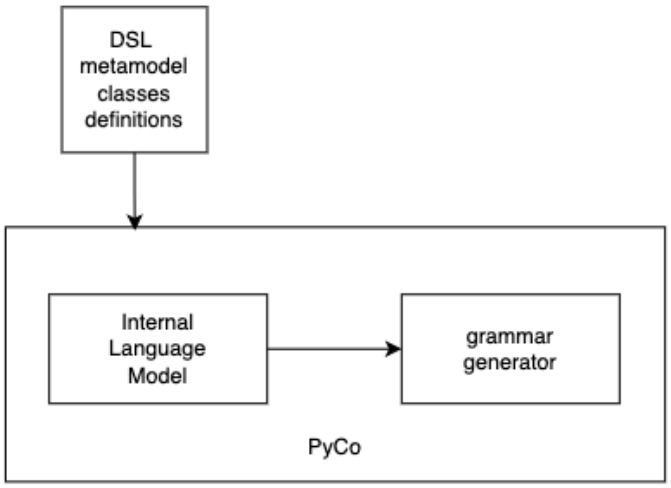

Figure 4.9: PyCo(Stage Two)-Processing(grammar generator)

Figure 4.9 shows the stage 2 of PyCo library, showing how internal language model is used to generate the grammar for the Lark to process.

4.3 The class instance generator

The class instance generator is the final main section in PyCo tool. The class instance generator is responsible of creating the class instance of the starting base class(the class which was provided to PyCo for processing while initialization).

The generated class instance can also have the other class instances inside, if they were specified in the language sentence and in the language metamodel.

To create the class instance, PyCo tool need two things, the internal language model (explained in section 4.1) and the parse tree.

Internal language model is already available from the first step of PyCo initialization. To get the parse tree, the method parse_to_tree of PyCo instance is called. In this method, PyCo uses the generated grammar from step two(explained in section 4.2) and pass it to Lark, which creates a Lark instance.

The initialized Lark instance have the definition and syntax of the grammar defined by the DSL metamodel. The function parse_to_tree then call the parse method of Lark instance to generate the tree.

To create the instances of the class, another method of PyCo instance named create_instance is called which takes the tree as an input use the internal language model stored in the attribute of PyCo instance. PyCo instance use both of these to create class instances recursively.

There are couple of things which we check while parsing the tree. If the tree has no attribute named 'data' then we know that the tree is already at the value of the node. In this case we take the value and return it to create the related class instance from it.

In case if the value of tree node has the attribute 'data' in it. Then this attribute contains the name of one of the classes that we defined in our metamodel. We use the name of this class and filter the matching class instance from the internal language model.

In case, the filter could not find the related metadata class in the internal language model, the next course of action would be to call the create_instance function recursively on the first child of the tree.

Assuming that at this point we have the metadata class instance related to the name of class in the data attribute. We check the type of the class, whether it belong to an abstract class or not, if it does, we perform the same operation and recursively call the create_instance method on the first child of the tree.

There is another attribute of this internal language model class instance called 'fields'. This attribute contains the names of all the fields related to a class.

We check if it contains at least one element in it. If the 'fields' attribute does not contain at least one element, we just simply create the instance of the class. On the other hand if 'fields' list contains at least one element, we iterate through all these elements and based upon certain criteria and find values from the parse tree, which corresponds to these fields. After find the value, we create class instance.

In the processing of these individual fields, we check if the field was optional or not. If the field was optional, we try to find the value for the field. If no value if found we assign a default value to it, which had been stored in the internal language model class.

After iterating through all the values in 'fields' attribute, we create the instance of the class, to which all these fields belonged.

Figure 4.10 shows the full structure of PyCo Class.

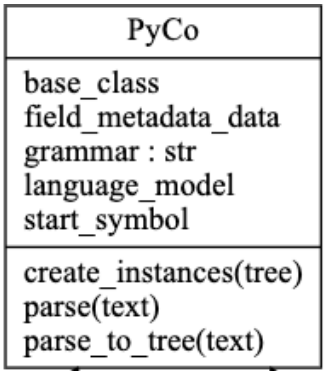

Figure 4.10: PyCo class diagram

There are also two very important types that we check during this process. If the type of field is List or Dict, we find all the children of the tree, either by using Lark find_data filter, or using tree.children and call the function create_instances recursively on each child.

To simplify the parsing and class instance generation, there is another method available in PyCo known as 'parse'. This method combines the functionalities of both parse_to_tree and create_instance and produce the final result(class instances).

Full PyCo library class structure would look like in figure 4.11.

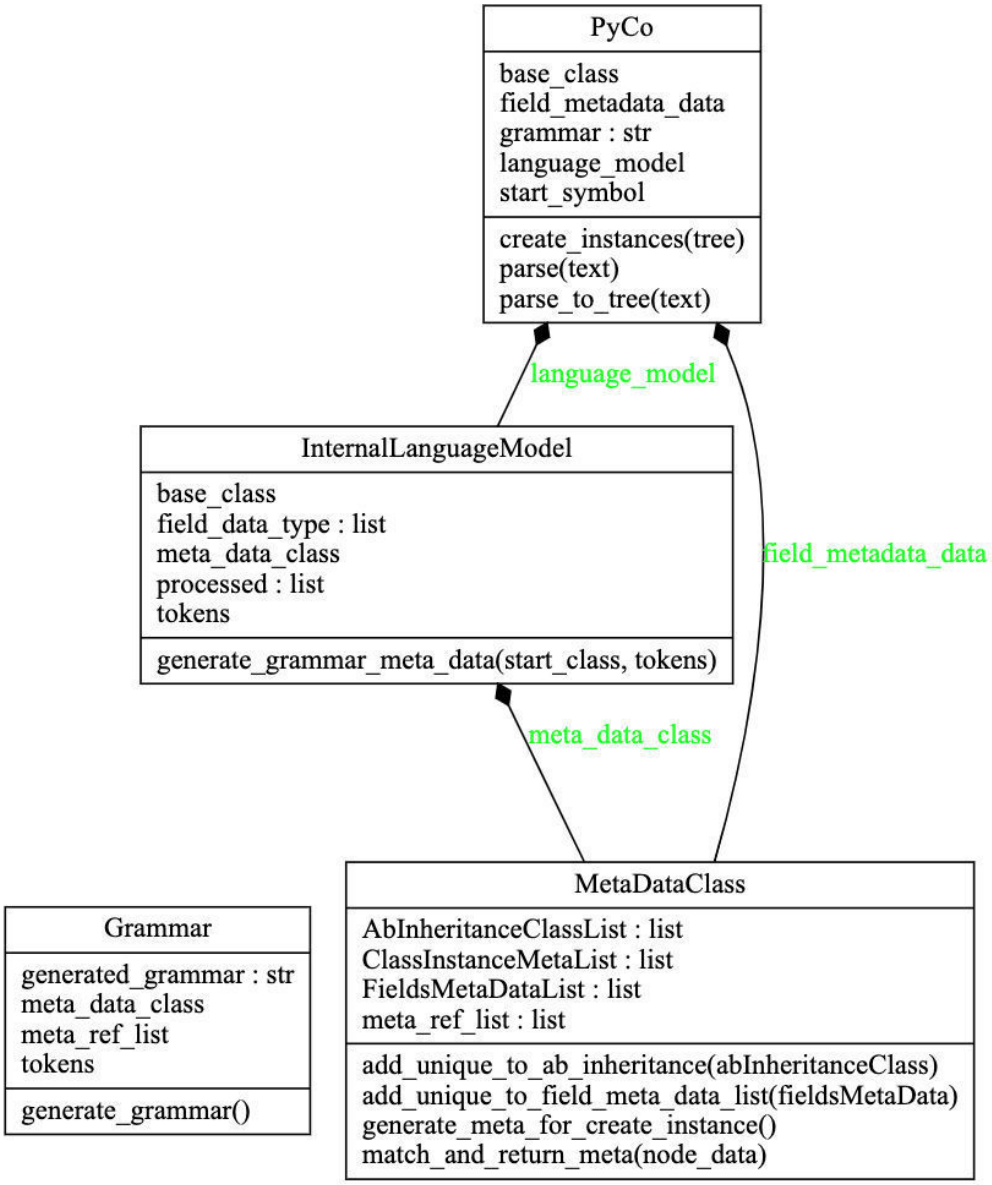

Figure 4.11: PyCo architecture(classes)

By examining the three stages, we can come to the conclusion of final structural diagram of PyCo showing all the processes and interactions of all the stages, metamodel and language sentences. The final diagram would look like the structure in the figure 4.12.

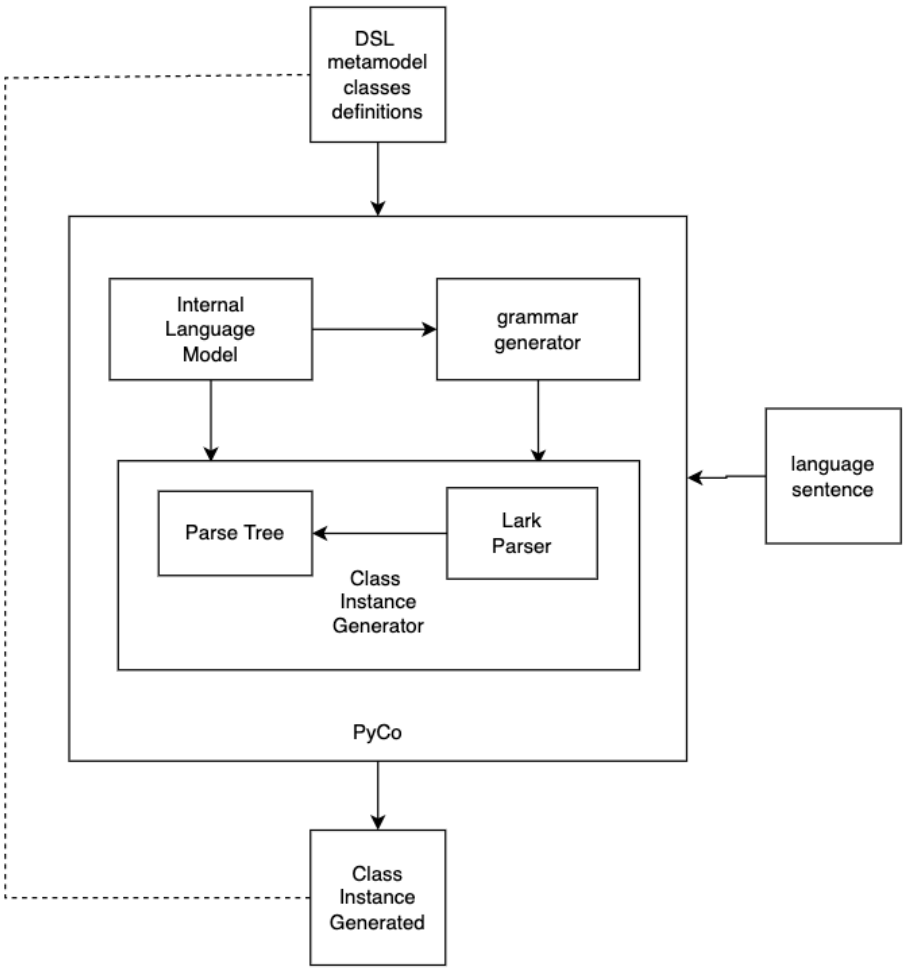

Figure 4.12: PyCo(Stage Three)-Processing(class instance generator)

4.4 Python package structure of PyCo

After going through each internal processing stage of PyCo and learning about its structure and relationships. An overall image can be concluded about its implementation. However there are also other modules and structural relationships that cannot be concluded without examining the flow of the code.

Figure 4.13 shows the whole package structure of PyCo.

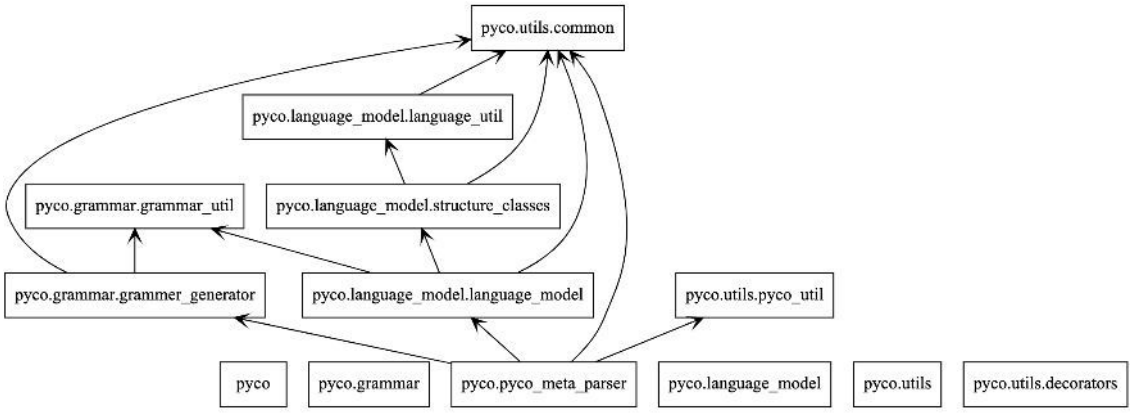

Figure 4.13: PyCo package structure

5 Evaluation

This chapter focuses on the implementation and evaluation of different DSL languages. Requirements for the DSL will be described, based on which the metamodels using object-oriented programming will be created.

We will test these metamodels by running them through PyCo tool and passing the desired language sentence.

Throughout whole this process, PyCo will be analyzed and evaluated for creating these DSLs and parsing the language sentences.

5.1 JSON language

JSON language is one of the most important structure of language in the modern era. The language is capable of handling string, integer, float , boolean and dictionary. Any of the string, integer, float , boolean and dictionary is a valid input.

For the purpose of evaluation, we are going to use different types of language sentences defined below.

- a simple string input: "a string"
- a number input: 10
- a null input : null
- a list input: ["a",10]
- a dictionary input {"a":10}
- a boolean input 'false' (false will represent False and true will represent True)
- a complex list within a dictionary input: {"a":["b",1,2,false]}

5.1.1 JSON metamodel classes

Metamodel of JSON language using PyCo is described in listing 5.1

```
from abc import ABC
from dataclasses import dataclass, field
from typing import List, Dict
from pyco.utils.decorators import syntax


@dataclass()
class JsonValue(ABC):
    pass


@dataclass
class JsonNumber(JsonValue):
    number: float = field(metadata={'token': 'NUMBER_TOKEN'})


@dataclass
class JsonString(JsonValue):
    string_value: str = field(metadata={'token': "STRING_TOKEN"})

    def __hash__(self):
        return hash(self.string_value)


@dataclass
class JsonBoolean(JsonValue):
    boolean: bool = field(metadata={'token': "BOOLEAN_TOKEN"})


@dataclass
class JsonArray(JsonValue):
    elements: List[JsonValue] = field(metadata={'separator': ',', '
   before': '[', 'after': ']'})


@dataclass
@syntax(before='null')
class JsonNull(JsonValue):
    pass


@dataclass
```

```
class JsonDictionary(JsonValue):
    elements_dict: Dict[JsonString, JsonValue] = field(
        metadata={'separator': ':', 'element_separator': ',', 'before'
    : '{', 'after': '}'})
```

Listing 5.1: JSON Metamodel Classes

Token used in JSON language

```
from pyco.pyco_meta_parser import STRING_REG, NUMBER_REG
token = {
    "STRING_TOKEN": STRING_REG,
    "NUMBER_TOKEN": NUMBER_REG,
    "BOOLEAN_TOKEN": '"true" -> true| "false" -> false'
}
```

5.1.2 JSON language grammar

Grammar generated by PyCo is shown in the listing 5.2 below.

```
jsonnumber     : number_token
jsonstring     : string_token
jsonboolean    : boolean_token
jsonarray      : "[" [jsonvalue ( "," jsonvalue)*] "]"
jsondictionary : "{" [jsonstring":"jsonvalue ( ","  jsonstring
                 ":" jsonvalue)*] "}"
jsonvalue      : jsonnumber | jsonstring | jsonboolean | jsonarray
                 | jsonnull | jsondictionary
jsonnull       : "null"

string_token   : ESCAPED_STRING
number_token   : SIGNED_NUMBER
boolean_token  : "true" -> true| "false" -> false

%import common.ESCAPED_STRING
%import common.SIGNED_NUMBER
%import common.WS
%ignore WS
```

Listing 5.2: JSON language grammar

5.1.3 JSON lang input tests

language sentence input "a"

Class instance result is:

```
JsonString(string_value='"a"')
```

language sentence input: 10

Class instance result is:

```
JsonNumber(number=10.0)
```

language sentence input: null

Class instance result is:

```
JsonNull()
```

language sentence input ["a", 10]

Class instance result is:

```
JsonArray(elements=[JsonString(string_value='"a"'), JsonNumber(number
    =10.0)])
```

language sentence input: {"a": 10}

Class instance result is:

```
JsonDictionary(elements_dict={JsonString(string_value='"a"'):
    JsonNumber(number=10.0)})
```

language sentence input: false

Class instance result is:

```
JsonBoolean(boolean=False)
```

language sentence input: {"a":["b",1,2,false,null]}

Parse tree is:

```
jsonvalue
    jsondictionary
      jsonstring
        token_1 "a"
      jsonvalue
        jsonarray
          jsonvalue
            jsonstring
              token_1 "b"
          jsonvalue
            jsonnumber
              token_2 1
          jsonvalue
            jsonnumber
              token_2 2
          jsonvalue
            jsonboolean
              false
          jsonvalue
            jsonnull
```

Class instance result is:

```
JsonDictionary(elements_dict={JsonString(string_value='"a"'):
    JsonArray(elements=[JsonString(string_value='"b"'), JsonNumber(
    number=1.0), JsonNumber(number=2.0), JsonBoolean(boolean=False),
    JsonNull()])})
```

By evaluating some of the possible input language sentences, it can be concluded the described language metamodel and PyCo are creating class instances the way it is supposed to be created. Thus JSON language implementation was fully successful.

5.2 Function language

If there is a requirement to process a language sentence in a way that it depicts the syntax of a function then PyCo is more than capable enough to handle such kind of task too. In this example of function definition language, we are not only going to design a metamodel of the function language but will also do some testing of Optional field type implementation.

The language sentence that we wish to parse and process will have the structure like:

```
fun: "function_name" ("y"=2 , "x"=6 ){"y"= 8, "z"=30}
```

For the purpose of evaluation, we are going to use different types of language sentences defined below.

- fun: "function_name" ("y"=2 , "x"=6){"y"= 8, "z"=30}
- fun: ("y"=2 , "x"=6){"y"= 8}' (to test Optional type of class field)

To test the Optional type of the field in a class implementation we are going to be make the function_name field in the metamodel Optional.

5.2.1 Function metamodel classes

Metamodel of Function language using PyCo is described in listing 5.3

```
from abc import ABC
from dataclasses import dataclass, field
from typing import Optional, List
from pyco.utils.decorators import syntax


@dataclass()
class FunctionAbstractClass(ABC):
    pass


@dataclass
class NumberClass(FunctionAbstractClass):
    number: float = field(metadata={'token': 'NUMBER_TOKEN'})


@dataclass
class StringClass(FunctionAbstractClass):
    string_value: str = field(metadata={'token': "STRING_TOKEN"})

    def __hash__(self):
        return hash(self.string_value)


@dataclass
class Statement:
    name_statement: StringClass = field(metadata={'after': "="})
    value_statement: NumberClass

@dataclass
class parameters:
    name_parameter: StringClass = field(metadata={'after': "="})
```

```
    value_parameter: NumberClass



@dataclass
@syntax(before="fun:")
class function(FunctionAbstractClass):
    function_name: Optional[str] = field(metadata={'token': "
    STRING_TOKEN", 'default': 'DEFAULT NAME'})
    parameter_list: List[parameters] = field(metadata={'separator': ',
    ', 'before': '(', 'after': ')'})
    body: List[Statement] = field(metadata={'before': '{', 'after': '}
    ', 'separator': ','})

    def get_parameters(self):
        return self.parameter_list
```

Listing 5.3: Funtion Metamodel Classes

Token used in Function language

```
token = {
    "STRING_TOKEN": STRING_REG,
    "NUMBER_TOKEN": NUMBER_REG,
}
```

5.2.2 Function language grammar

Grammar generated by PyCo is shown in the listing 5.4 below.

```
numberclass           : number_token
stringclass           : string_token
function              : "fun:"  [string_token]
                        "(" [parameters ( "," parameters)*] ")"
                        "{" [statement ( "," statement)*] "}"
parameters            : stringclass "="  numberclass
statement             : stringclass "="  numberclass
functionabstractclass : numberclass | stringclass | function

string_token : ESCAPED_STRING
number_token : SIGNED_NUMBER

%import common.ESCAPED_STRING
%import common.SIGNED_NUMBER
%import common.WS
```

```
%ignore WS
```

Listing 5.4: Function language grammar

5.2.3 Function language input tests

language sentence input fun: "function_name" ("y"=2,"x"=6){"y"=8,"Z"=30}'

Parse tree is:

```
functionabstractclass
  function
    string_token  "function_name"
    parameters
      stringclass
        string_token  "y"
      numberclass
        number_token  2
    parameters
      stringclass
        string_token  "x"
      numberclass
        number_token  6
    statement
      stringclass
        string_token  "y"
      numberclass
        number_token  8
    statement
      stringclass
        string_token  "z"
      numberclass
        number_token  30
```

Class instance result is:

```
function(function_name='"function_name"', parameter_list=[parameters(
   name_parameter=StringClass(string_value='"y"'), value_parameter=
   NumberClass(number=2.0)), parameters(name_parameter=StringClass(
   string_value='"x"'), value_parameter=NumberClass(number=6.0))],
   body=[Statement(name_statement=StringClass(string_value='"y"'),
   value_statement=NumberClass(number=8.0)), Statement(name_statement=
   StringClass(string_value='"z"'), value_statement=NumberClass(number
   =30.0))])
```

language sentence input 'fun: ("y"=2 , "x"=6){"y"= 8}'

Parse tree is:

```
functionabstractclass
  function
    None
    parameters
      stringclass
        string_token  "y"
      numberclass
        number_token  2
    parameters
      stringclass
        string_token  "x"
      numberclass
        number_token  6
    statement
      stringclass
        string_token  "y"
      numberclass
        number_token  8
```

Class instance result is:

```
function(function_name='DEFAULT NAME',
parameter_list=[parameters(name_parameter=StringClass(string_value='"y
    "'), value_parameter=NumberClass(number=2.0)), parameters(
    name_parameter=StringClass(string_value='"x"'), value_parameter=
    NumberClass(number=6.0))], body=[Statement(name_statement=
    StringClass(string_value='"y"'), value_statement=NumberClass(number
    =8.0))])
```

By evaluating some possible input language sentences, it can be concluded the described language metamodel and PyCo are creating class instances the way it is supposed to be created. Optional type for class field, has also been tested as in the second language sentence where we did not mention the function_name. The default value was automatically used. Thus Function language implementation was fully successful.

5.3 Robot language

Robot language is yet another DSL language that can be created using PyCo. This language has also been defined in YAJCo tool(listing 1.3). The language implemented in this section is similar to YAJCo's example but with one additional field

called speed. In PyCo example of the Robot language, the language sentence also contains the turning speed of the robot.

The language sentence that we wish to parse and process will have the structure like:

```
begin move , turn right 5, turn left 2, move end
```

For the purpose of evaluation, we are going to use language sentence defined below.

```
begin move , turn right 5, turn left 2, move end
```

5.3.1 Robot metamodel classes

Metamodel of Robot language using PyCo is described in listing 5.5

```
from abc import ABC
from dataclasses import dataclass, field
from typing import List
from pyco.utils.decorators import syntax


class Commands(ABC):
    def move(self):
        pass


@dataclass
@syntax(before="move")
class Move(Commands):
    def move(self):
        print('move forward')


@dataclass
@syntax(before="turn")
class Turn(Commands):
    directions: str = field(metadata={'token': 'DIRECTION_TOKEN'})
    speed: float = field(metadata={'token': 'SPEED_TOKEN'})

    def move(self):
        print('turn', self.directions)


@dataclass
@syntax(before="begin", after='end')
```

```
class Robot:
    body: List[Commands] = field(metadata={'separator': ','})

    def start_moving(self):
        for step in self.body:
            step.move()
```

Listing 5.5: Robot Metamodel Classes

Token used in Robot language

```
token = {
    "DIRECTION_TOKEN": '"back" ->back|"right" -> right|"left" ->left',
    "SPEED_TOKEN": NUMBER_REG,
}
```

5.3.2 Robot language grammar

Grammar generated by PyCo is shown in the listing 5.6 below.

```
robot     :  "begin"  [commands ( "," commands)*]   "end"
turn      :  "turn"  direction_token speed_token
commands  :   move | turn
move      :  "move"

direction_token : "back" -> back|"right" -> right|"left" ->left
speed_token : SIGNED_NUMBER

%import common.ESCAPED_STRING
%import common.SIGNED_NUMBER
%import common.WS
%ignore WS
```

Listing 5.6: Robotlanguage grammar

5.3.3 Robot language input tests

language sentence input 'begin move , turn right 5, turn left 2, move end'

Parse tree is:

```
robot
  commands
    move
  commands
```

```
    turn
      right
      speed_token 5
  commands
    turn
      left
      speed_token 2
  commands
    move
```

Class instance result is:

```
Robot(body=[Move(), Turn(directions='right', speed=5.0), Turn(
  directions='left', speed=2.0), Move()])
```

By evaluating some of the possible input language sentences, it can be concluded the described language metamodel and PyCo are creating class instances the way it is supposed to be created. Thus Function language implementation was fully successful.

5.4 Robot Complex language

Robot Complex language is an extension of Robot language. We are referencing different classes with multiple fields to link together so that instead of only providing movement, direction and speed, we are able to provide a range of speed and the information about the movement and its impact on the robot motion.

The language sentence that we wish to parse and process will have the structure like:

```
begin move , turn right [1 2 , 3 4] ("Damage" 6 ["OnTheEdge" 10,"
    Rudder" 11])  move  end
```

For the purpose of evaluation, we are going to use language sentence defined below.

```
begin move , turn right [1 2 , 3 4] ("Damage" 6 ["OnTheEdge" 10,"
  Rudder" 11])  move  end
```

5.4.1 Robot Complex metamodel classes

Metamodel of Robot language using PyCo is described in listing 5.7

```
from abc import ABC
from dataclasses import dataclass, field
from typing import List
```

```
from pyco.utils.decorators import syntax


class Commands(ABC):
    def move(self):
        pass


@dataclass
@syntax(before="move")
class Move(Commands):
    def move(self):
        print('move forward')


@dataclass
class RobotSpeed:
    min_speed: float = field(metadata={'token': 'MIN_SPEED_TOKEN'})
    max_speed: float = field(metadata={'token': 'MAX_SPEED_TOKEN'})


@dataclass
class InfoClass:
    description: str = field(metadata={'token': 'MIN_STR_TOKEN'})
    length: int = field(metadata={'token': 'LENGTH_TOKEN'})


@dataclass
class ImpactType:
    incident: str = field(metadata={'token': 'MIN_STR_TOKEN'})
    max_impact: float = field(metadata={'token': 'MAX_IMPACT_TOKEN'})
    info: List[InfoClass] = field(metadata={'separator': ',', 'before'
    : '[', 'after': ']'})


@dataclass
@syntax(before="turn")
class Turn(Commands):
    direction: str = field(metadata={'token': 'DIRECTION_TOKEN'})
    speed: List[RobotSpeed] = field(metadata={'separator': ',', '
    before': '[', 'after': ']'})
    impact: ImpactType = field(metadata={'before': '(', 'after': ')'})

    def move(self):
        print('turn', self.directions)
```

```
47
48
49 @dataclass
50 @syntax(before="begin", after='end')
51 class RobotComplex:
52     body: List[Commands] = field(metadata={'separator': ','})
53
54     def start_moving(self):
55         for step in self.body:
56             step.move()
```

Listing 5.7: Robot Complex Metamodel Classes

Token used in Robot Complex language

```
1 from pyco.pyco_meta_parser import NUMBER_REG, STRING_REG
2
3 token = {
4       "DIRECTION_TOKEN" : '"back" -> back|"right" -> right|"left" ->
    left',
5       "MIN_SPEED_TOKEN" : NUMBER_REG,
6       "MAX_SPEED_TOKEN" : NUMBER_REG,
7       "WEIGHT_TOKEN"    : NUMBER_REG,
8       "MIN_STR_TOKEN"   : STRING_REG,
9       "MAX_IMPACT_TOKEN": NUMBER_REG,
10      "LENGTH_TOKEN"    : NUMBER_REG
11   }
```

5.4.2 Robot Complex language grammar

Grammar generated by PyCo is shown in the listing 5.8 below.

```
1  robotcomplex   :  "begin"  [commands ( "," commands)*]   "end"
2  turn           :  "turn"  direction_token
3                    "[" [robotspeed ( "," robotspeed)*] "]"
4                    "(" impacttype ")"
5  robotspeed     :  min_speed_token max_speed_token
6  impacttype     :  min_str_token max_impact_token
7                    "[" [infoclass ( "," infoclass)*] "]"
8  infoclass      :  min_str_token length_token
9  commands       :  move | turn
10 move           :  "move"
11
12 direction_token : "back" -> back|"right" -> right|"left" ->left
13 min_speed_token : SIGNED_NUMBER
```

```
max_speed_token : SIGNED_NUMBER
weight_token : SIGNED_NUMBER
min_str_token : ESCAPED_STRING
max_impact_token : SIGNED_NUMBER
length_token : SIGNED_NUMBER

%import common.ESCAPED_STRING
%import common.SIGNED_NUMBER
%import common.WS
%ignore WS
```

Listing 5.8: Robot Complex language grammar

5.4.3 Robot Complex language input tests

language sentence input 'begin move , turn right 5, turn left 2, move end'

Parse tree is:

```
robotcomplex
  commands
    move
  commands
    turn
      right
      robotspeed
        min_speed_token 1
        max_speed_token 2
      robotspeed
        min_speed_token 3
        max_speed_token 4
      impacttype
        min_str_token "Damage"
        max_impact_token  6
        infoclass
          min_str_token "OnTheEdge"
          length_token  10
        infoclass
          min_str_token "Rudder"
          length_token  11
  commands
    move
```

Class instance result is:

```
RobotComplex(body=[Move(), Turn(direction='right', speed=[RobotSpeed(
   min_speed=1.0, max_speed=2.0), RobotSpeed(min_speed=3.0, max_speed
```

```
=4.0)], impact=ImpactType(incident='"Damage"', max_impact=6.0, info
=[InfoClass(description='"OnTheEdge"', length=10), InfoClass(
description='"Rudder"', length=11)])), Move()])
```

By evaluating some of the possible input language sentences, it can be concluded the described robot complex language metamodel and PyCo are creating class instances the way it is supposed to be created. Thus Robot Complex language implementation was fully successful.

5.5 Conclusion of tested DSL metamodels

This section list down the main evaluation results of PyCo library by examining each example and specifying the functionalities available in PyCo. Through these examples all the available class decorators along with their available parameters have been used. The examples also make use of all the possible combination of field metadata keys available.

Along with the metadata and decorators another important aspect that could be concluded is that PyCo also has the capability to use user defined classes as types and metadata can also be attached to these fields too. Thus PyCo was correctly generating all the class instances from language metamodel and language sentences.

5.6 Limitations of PyCo library

Like every other library, PyCo also have some limitations and set of rules that must be kept in mind and followed accordingly.

1. Defined class names should be different from the defined token names as this will generate ambiguity in the grammar and Lark won't be able to process the grammar.

2. If tokens are referenced in the metadata, those tokens must be passed in the PyCo class instance initialization.

3. To define boolean True and false, the expression must map the desired value onto tree or false (If you wish to represent T as true and F as false, then you would have to define the token like this "BooleanToken": '"T" -> true|"F" -> false').

4. When using dictionary, it is necessary to include a hash method in the class of field which is going to be used to represent keys for the dictionary.

5. PyCo has no implementation of type Set for field types in a class.

5.7 PyCo and YAJCo Comparison

This section in the evaluation chapter specifies the similarities and differences between YAJCo and PyCo. As PyCo is build with the example of YAJCo in mind, there are many common features between both these tools.

Both PyCo and YAJCo are supposed to perform similar task of creating a DSL based up the metamodel based class definitions. However, even when the features result is same, the approach in which these features are represented in the language metamodel is quite different.

5.7.1 Abstract Syntax Definition

In YAJCo the implementation of abstract syntax of DSL is implemented by the use of annotations of a class and fields [10]. However, PyCo uses decorators only on the classes and not on the fields. For fields PyCo uses field function of dataclasses library to assign the relevant metadata to the field of a class.

Similar to YAJCo, PyCo also uses inheritance to link class and subclasses together to generate the EBNF grammar.

5.7.2 Composition multiplicity

YAJCo implements multiplicity with the use of List and Set in Java. This feature is implemented in PyCo using List and dictionaries. YAJCo also has and additional feature to limit the number of elements in the list with @Range annotation.

5.7.3 Referencing(aggregation)

Another concept in YAJCo is to have the ability to have a reference to another language model which has been described somewhere else with the use of @References annotation. This relation is an example of aggregation in object model[10].

This feature is not implemented in PyCo yet.

5.7.4 Keywords and symbols

As described earlier in subsection 5.7.1, YAJCo uses annotation to define abstract syntax of a language.

In PyCo only classes have the ability to have a decorator names @syntax which can decorate the whole class.

@syntax in PyCo takes two parameters, before and after. This give us the ability to define the syntax of the overall class, describing what should come before or after the language elements related to that class.

Similar to this YAJCo's annotation are @Before and @After, which perform the same functionality. Along with this YAJCo annotation can also be applied to fields of a class.

The defining of metadata on the fields of a class in PyCo however is done using fields function of dataclasses library.

For the implementation of composition multiplicity @Separator annotation is used in YAJCo. However to separate list elements separator key is used in PyCo field metadata.

In PyCo to define dictionary we also have another metadata key for a class field known as element_separator. In the case of dictionary, separator key is used to, separate the key value pair and element_separator is used to separate elements(one element is key:value pair).

Implementation of dictionary is not available in YAJCo.

5.7.5 Operator definition

In YAJCo @Operator annotation is used to define priority and associativity. This is very useful for implementation of DSL based upon mathematical rules.

@Operator is often used with @Parentheses annotation to declare priority explicitly.

This feature is not available in PyCo.

5.7.6 Tokens with value

In YAJCo to define a token, @Token annotation is used and regular expressions are passed [10]. In YAJCo there are no predefined tokens. We can also define tokens globally when declaring the main class using @Parser annotation. This declaration of tokens is know as named tokens.

Tokens are passed when creating the instance of PyCo. PyCo takes a token dictionary, and these tokens can be referenced anywhere in the grammar.

Similar to YAJCo, we can also explicitly define token for a class field using a valid regular expression.

There are two predefined tokens(STRING_REG & NUMBER_REG) in PyCo that can be imported from PyCo library. Unlike PyCo, YAJCo does not have and predefined tokens available.

5.7.7 Additional annotations in YAJCo

There are other annotation available in YAJCo that helps to perform some additional functionalities in YAJCo. These functionalities are not yet available in PyCo.

The @Identifier annotation is used to reference a class by the identifier. This annotation helps to set up a unique identifier [10].

@FactoryMethod is an annotation available in YAJCo to specify static methods, which are used in the creation of parser.

@Exclude annotation is used to mark the constructor to be excluded from the abstract syntax specification.

@Parser annotation is known to be the main configuration YAJCo tool's element. This annotation is a must requirement to start YAJCo tool. It takes parameters to define the root concepts of the DSL. This is not needed in PyCo because when we create the instance, we specify the root class of the DSL along with tokens parameter.

@TokenDef is used to defined named tokens in @Parser annotation. PyCo takes token dictionary when initializing.

@Skip' annotation is used to skip the characters defined in the set or regular expression.

@Option' annotation is used to define the options for the YAJCo language.

@Newline' annotation is used to specify the place where there should be a line gap in the printer output.

@Indent' annotation is used to define indentation level for printed output.

5.8 Future improvements of PyCo

Based upon the examples evaluations, restrictions and the comparison of PyCo with YAJCo, this section specifies the possible improvements and additions which are possible to be implemented in PyCo.

- We can create error handling, which indicate if the token names and class names are same. This is needed because PyCo grammar becomes ambigu-

ous, if the token name and the class names are same. This will greatly help in handling ambiguity in the grammar.

- Similar to YAJCo, priority and parenthesis decorators can be added to PyCo. Internally this has change the structure of the grammar to make the appropriate parse tree. This will make the library to handle mathematical based equations.
- Functions can be defined in the library which will generate regular expressions, based upon some parameters.
- Similar to YAJCo @exclude decorator can be created to exclude the class from abstract syntax
- Further implementations of type Set and type Any can be included in the library.
- Limit on the elements in the list and dictionary can be implemented using something similar to @Range annotation of YAJCo.
- A list of more regular expressions can be created to quickly import and use from the library.

6 Conclusion

This thesis presents a research and implementation of a Metamodel-based Parser Generator for Python. Taking inspiration from YAJCo tool and generating something similar in Python was the main goal of this thesis. The reason such a tool in Python is needed because the YAJCo tool is only specific to Java. With the in-depth study of various parsing tools and analyzing the features of YAJCo, this thesis specified and laid the foundation of the construction of PyCo(metamodel based parser tool in Python).

The main objectives on which thesis successfully achieved are:

1. Giving user the ability to define metamodel for DSL using Python classes.
2. Creating internal language model structure, which will store the metadata of classes and their fields.
3. Generating grammar from the internal language model.
4. Generating parse tree using provided language sentence and the existing parser LARK [7](parsing tool used for this thesis)
5. Creating class instances from generated parse tree with the help of internal language model.

With Various examples, the thesis successfully demonstrated all the features of DSL generation using Metamodel-based Parser Generator for Python.

With the implementation of PyCo, many restrictions and limitations of the library surfaced, which has been mentioned in this thesis.

This thesis specified all the possible solutions to the restriction of PyCo along with further improvements suggestion by comparing the PyCo tool with YAJCo.

With the implementation of PyCo, a base foundation structure of a fully dynamic tool has been generated which can be easily worked with or improved with additional features in future.

Bibliography

1. PORUBÄN, Jaroslav; FORGÁC, Michal; SABO, Miroslav; BĚHÁLEK, Marek. Annotation based parser generator. *Computer Science and Information Systems*. 2010, vol. 7, no. 2, pp. 291–307. Available from DOI: 10.2298/csis1002291p.

2. PORUBÄN, Jaroslav; CHODAREV, Sergej. Model-aware language specification with Java. In: *2015 13th International Conference on Engineering of Modern Electric Systems (EMES)*. 2015, pp. 1–4. Available from DOI: 10.1109/EMES.2015.7158424.

3. MCCRACKEN, Daniel D; REILLY, Edwin D. Backus-naur form (bnf). In: *Encyclopedia of Computer Science*. 2003, pp. 129–131.

4. KLEPPE, Anneke. Towards the Generation of a Text-Based IDE from a Language Metamodel. In: AKEHURST, David H.; VOGEL, Régis; PAIGE, Richard F. (eds.). *Model Driven Architecture- Foundations and Applications*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 114–129. ISBN 978-3-540-72901-3.

5. CHODAREV, Sergej; HALAMA, Ján. Interconnecting YAJCo with Xtext: Experience Report. In: *2019 IEEE 15th International Scientific Conference on Informatics*. 2019, pp. 000195–000200. Available from DOI: 10.1109/Informatics47936.2019.9119305.

6. *Parsing in python: All the tools and libraries you can use*. 2022. Available also from: https://tomassetti.me/parsing-in-python/.

7. LARK-PARSER. *Lark-parser/lark: Lark is a parsing toolkit for python, built with a focus on ergonomics, performance and modularity.* [N.d.]. Available also from: https://github.com/lark-parser/lark.

8. CHODAREV, Sergej; BAČÍKOVÁ, Michaela. Abstract-syntax-driven development of oberon-0 using Yajco. *Journal of information and organizational sciences*. 2019, vol. 43, no. 2, pp. 145–162. Available from DOI: 10.31341/jios.43.2.2.

9. CHODAREV, Sergej; LAKATOŠ, Dominik; PORUBÄN, Jaroslav; KOLLÁR, Ján. Abstract syntax driven approach for language composition. *Open Computer Science*. 2014, vol. 4, no. 3, pp. 107–117. Available from DOI: doi:10.2478/s13537-014-0211-8.
10. KPI-TUKE. *KPI-Tuke/Yajco: YAJCo (yet another Java compiler compiler) is a language parser generator based on annotated model.* [N.d.]. Available also from: https://github.com/kpi-tuke/yajco.
11. KPI-TUKE. *Yajco-examples/robot.java at master · KPI-Tuke/yajco-examples*. 2017. Available also from: https://github.com/kpi-tuke/yajco-examples/blob/master/yajco-example-simpleRobot/src/main/java/yajco/robot/model/Robot.java.
12. PARR, T. J.; QUONG, R. W. Antlr: A predicated-ll(k) parser generator. *Software: Practice and Experience*. 1995, vol. 25, no. 7, pp. 789–810. Available from DOI: 10.1002/spe.4380250705.
13. PARR, Terence. *The definitive antlr 4 reference*. Pragmatic Bookshelf, 2014.
14. CAO, Danyang; BAI, Donghui. Design and implementation for SQL parser based on ANTLR. In: *2010 2nd international Conference on Computer engineering and technology*. 2010, vol. 4, pp. V4–276.
15. *The Antlr Mega Tutorial*. 2022. Available also from: https://tomassetti.me/antlr-mega-tutorial/.
16. MCGUIRE, Paul. *Getting started with pyparsing*. O'Reilly Media, Inc, 2007.
17. *Using the pyparsing module*. [N.d.]. Available also from: https://pyparsing-docs.readthedocs.io/en/latest/HowToUsePyparsing.html.
18. ERIKROSE. *Erikrose/Parsimonious: The fastest pure-python peg parser I can muster*. [N.d.]. Available also from: https://github.com/erikrose/parsimonious.
19. RIMKO, Jeff. *Parsing with parsimonious*. [N.d.]. Available also from: https://www.jeffcomput.es/posts/2013/05/parsing-with-parsimonious/.
20. AHO, Alfred V; JOHNSON, Stephen C. LR parsing. *ACM Computing Surveys (CSUR)*. 1974, vol. 6, no. 2, pp. 99–124.
21. HANDOUT, CS143. LALR Parsing. [N.d.].
22. AYCOCK, John; HORSPOOL, R. Nigel. Practical Earley Parsing. *The Computer Journal*. 2002, vol. 45, no. 6, pp. 620–630. Available from DOI: 10.1093/comjnl/45.6.620.

23. *Welcome to Lark's documentation!* [N.d.]. Available also from: `https://lark-parser.readthedocs.io/en/latest/`.

24. *Dataclasses - data classes*. [N.d.]. Available also from: `https://docs.python.org/3/library/dataclasses.html`.

25. *Python dataclass*. 2021. Available also from: `https://www.pythontutorial.net/python-oop/python-dataclass/`.

26. *Typing - support for type hints*. [N.d.]. Available also from: `https://docs.python.org/3/library/typing.html`.

List of Appendixes

Appendix A User Manual

Appendix B System Manual

Appendix C CD médium – záverečná práca v elektronickej podobe,

A User Manual

A.1 Requirements of PyCo library

Metamodel based parser in python (PyCo) library requirements are.

1. Minimum version of Python required is 3.7
2. The library is based on Lark version 1.1.1
3. Pytest 7.1.1 (if require to run defined test in the tests folder)
4. pylint 2.13.5 (to generate class diagrams and relationship model)

Minimum version of Python is 3.7 because, the use of dataclasses library was introduced in this version of Python library.

The current version of the PyCo is fully functional with Lark 1.1.1

A.2 Installing dependencies

There is a requirement.txt file in the root folder of the code.

use pip to install the dependencies

```
    pip install -r requirements.txt
```

A.3 How to use the library

A.3.1 Define metamodel

```
from dataclasses import dataclass, field

@dataclass()
class JsonValue(ABC):
```

```
    pass

@dataclass
class JsonNumber(JsonValue):
    number: float = field(metadata={'token': 'NUMBER_TOKEN'})
```

A.3.2 Initialize pyCo and create class instances

```
from pyco.pyco_meta_parser import PyCo, NUMBER_REG
from classes import JsonValue

token = {
    "NUMBER_TOKEN": NUMBER_REG,
}

if __name__ == '__main__':
    pyCo = PyCo(JsonValue, token)
    tree = pyCo.parse_to_tree('1')
    class_instance=pyCo.create_instances(tree)
```

A.4 Class decorators in PyCo

There is only one class decorator available in PyCO, with name syntax. @syntax can take two parameters before and after. These parameters are used to define the concrete syntax of the DSL.

Below is an example to define the class which represent None in python.

```
@syntax(before='null')
class Null()):
    pass
```

For the case where before is defined the language sentence will process null as None.

```
@syntax(before='null', after='!')
class Null()):
    pass
```

For the case where before and after both are defined the language sentence should look like null!

A.5 Metadata field keys in PyCo

1. For Strings: before, after, token

2. For Numbers: before, after, token
3. For List: before, after, separator
4. For Dict: before, after, separator, element_separator
5. For other Class Types: before ,after
6. For Optional Fields: default + all other key fields with respect to type

A.6 Methods available in PyCo

1. parse_to_tree (takes language sentence and return a tree)
2. create_instances (take tree as input and creates the instances of the class)
3. parse (combines both methods parse_to_tree and create_instances to return class instances)

Other features of PyCo.

- After PyCo initialization, PyCo has an attribute called grammar, that can be printed to visualize generated grammar.
- tree.pretty() can be called to return a tree in pretty form(you can print to visualize the tree)

A.7 Testing PyCo

To run test PyCo library, run the command **pytest** in the terminal of the project directory.

A.8 Generating Diagram of PyCo Structure

The library pylint is used to generate the class relationship and PyCo structure diagrams.

To generate the diagrams run the command: **pyreverse -o png pyco** in the terminal of the project directory.

A.9 Limitations of PyCo Library

1. An empty dictionary must be supplied to PyCO class instance generation, even if no tokens are defined.
2. Optional field only work for string, numbers and booleans at this stage.
3. Token names should be different from the class names to avoid grammar ambiguity.
4. For Dict: separator, element_separator keys must be present in the metadata parameter of field function of dataclasses.
5. default value must be defined if the field is set to be Optional
6. separator key must be present in the metadata parameter of field function of dataclasses.

B Systems Manual

It is much easier to understand the structure and workflow of PyCo by breaking down each category into further little code snippets. We will go in a sequence from the start to end, along with the internal processing flow of PyCO.

B.1 Conditions to be fulfilled for PyCo initialization

For the purpose of understanding, a simple DSL example will be used. To use the PyCo, we first create the metamodel structure of the DSL language.

Defining the metamodel: Suppose we wish to create a language which can either take a string, a number or a list of both.

```
from abc import ABC
from dataclasses import dataclass, field
from typing import List,

@dataclass()
class ExampleLang(ABC):
  pass

@dataclass
class LangNumber(ExampleLang):
  number: float = field(metadata={'token': 'NUMBER_TOKEN'})

@dataclass
class LangString(ExampleLang):
  string_value: str = field(metadata={'token': "STRING_TOKEN"})

@dataclass
class LangArray(ExampleLang):
    elements: List[ExampleLang] = field(
    metadata={'separator': ',','before': '[', 'after': ']'})
```

Listing B.1: PyCO DSL(ExampleLang) metamodel

When the metamodel of class is defined, we can continue with the initialization of PyCo. It can be noticed that there is token mentioned in the above example, which is referencing to "STRING_TOKEN" and "NUMBER_TOKEN". This means that with the initialization of the library, we also need to provide these tokens.

Initialization if PyCo class instance: To initialize the PyCo we are first going to define those tokens which had been referenced in the example and use Lark's predefined regex variable reference. These regular expressions(STRING_REG,NUMBER_REG) can be imported fro PyCo library.

```
from pyco.pyco_meta_parser import PyCo, STRING_REG, NUMBER_REG

token = {
    "STRING_TOKEN": STRING_REG,
    "NUMBER_TOKEN": NUMBER_REG,
}

#initialization of PyCO
pyCo = PyCo(ExampleLang, token)
```

Listing B.2: PyCo defining token

B.2 PyCo instance initialization

Once instance of PyCo is created. The class PyCO has some internal initialization attributes which are very important,

```
....
class PyCo:
class PyCo:
    """Main library class PyCo"""

    def __init__(self, a_class, tokens):
        self.start_symbol = a_class.__name__.lower()
        self.base_class = a_class
        self.language_model = InternalLanguageModel(base_class=a_class
    , tokens=token)
        self.field_metadata_data = self.language_model.meta_data_class
        self.field_metadata_data.generate_meta_for_create_instance()
        self.grammar = Grammar(tokens=tokens, meta_data_class=self.
    field_metadata_data).generate_grammar()


```

```
.....
```

Listing B.3: PyCo class initialization structure

In the listing B.3, it can be seen that when the class initializes, first it sets the start_symbol attribute to the root class name. The next most important step is to set the base_class attribute as root class.

Inside the language_model attribute of PyCo we are referencing to internal language model and initializing it by passing the token dictionary and the root class.

The language_model attribute which equals to InternalLanguageModel, has an attribute called meta_data_class.

From meta_data_class attribute, generate_meta_for_create_instance method is called. This method analyzes the classes metadata and their relationships to generate the metadata class instances which are needed to further proceed in the creating of grammar and class instances.

B.2.1 Implementation of PyCo methods

There are three methods available in PyCo class instance.

```
class PyCo:
...
    def parse_to_tree(self, language_sentence):
        parser = Lark(self.grammar,
                      start=self.start_symbol,
                      debug=True,
                      propagate_positions=True)
        return parser.parse(language_sentence)
...

```

Listing B.4: PyCo parse_to_tree method

PyCo's method parse_to_tree purpose is to process the EBNF grammar generated by the grammar generator by passing it to instance of Lark. Lark sets the rules and syntax with respect to the grammar, so only valid language sentences can be processed and converted into parse tree.

There are couple of parameters which we provide to Lark class instance initialization as shown in the listing B.4. First we provide the grammar, which is stored in the attribute grammar of PyCo class instance. Then we provide the other parameters like start, which is used to tell Lark instance to take this as a starting symbol from the defined grammar. Parameter debug is used to enable

the error messages with details in Lark, this help PyCo to let Lark handle the error reporting(related to grammar definition and language sentence) in more verbose manner.

The Parameter propagate_positions, is set to True enables Lark to keep track of rows and columns [23] of the parse tree..

The method Parse_to_tree takes language_sentence as a parameter. When Lark successfully initializes without any error while processing the grammar. We can pass this language sentence to Lark. If the language sentence is valid, Lark instance will generate a parse tree accordingly.

Another important method of PyCo is create_instances. This method use parse tree and internal language model to generate the class instance according to the metamodel of DSL. The method use recursion technique and create instance of each subclass to n^{th} depth.

```
class PyCo:

    def create_instances(self, tree):
    """this method uses language metadata model classes and parse tree
    to generate class instances"""
    params_list = []

    if not hasattr(tree, 'data'):
        return tree

    find_in_meta = self.field_metadata_data.match_and_return_meta(tree
    .data)

    if not find_in_meta:
        if len(tree.children) > 0:
            return self.create_instances(tree.children[0])
        return tree.data

    if find_in_meta.type_class == 'abc':
        return self.create_instances(tree.children[0])

    if len(find_in_meta.fields) == 0:
        return find_in_meta.class_instance()

    value = None

    for index, field in enumerate(find_in_meta.fields):
        if field.field_type == float.__name__:
            value = find_value_in_tree(tree, index)
```

```
28              if value != 'OPTIONAL':
29                  value = float(value)
30                  if len(tree.children[index].children) > 0:
31                      tree.children[index].children.pop()
32          if field.field_type == int.__name__:
33              value = find_value_in_tree(tree, index)
34              if value != 'OPTIONAL':
35                  value = int(value)
36                  if len(tree.children[index].children) > 0:
37                      tree.children[index].children.pop()
38
39          if field.field_type == str.__name__:
40              value = find_value_in_tree(tree, index)
41              if value != 'OPTIONAL':
42                  value = str(value)
43                  if len(tree.children[index].children) > 0:
44                      tree.children[index].children.pop()
45
46          if field.field_type == bool.__name__:
47              value = find_value_in_tree(tree, index)
48              if value != 'OPTIONAL':
49                  value = str_to_bool(value)
50                  if len(tree.children[index].children) > 0:
51                      tree.children[index].children.pop()
52
53          if field.field_type == dict.__name__:
54              value = create_dict([self.create_instances(child) for
    child in tree.children])
55
56          if field.field_type == list.__name__:
57              children_x = [x for x in tree.find_data(field.tree_ref)]
58              if len(children_x) < len(tree.children):
59                  value = [self.create_instances(child) for child in
    children_x]
60              else:
61                  value = [self.create_instances(child) for child in
    tree.children]
62
63          class_type = contains(self.field_metadata_data.
    FieldsMetaDataList,
64                                lambda x: x.ref_class_name == field.
    field_type)
65
66          if class_type:
67              children_x = [x for x in tree.find_data(field.field_type)]
```

```
68              for child in children_x:
69                  value = self.create_instances(child)
70
71          if value is None:
72              return self.create_instances(tree.children[index])
73          else:
74              if value == 'OPTIONAL':
75                  if field.field_type == 'float':
76                      params_list.append(float(field.default))
77                  elif field.field_type == 'int':
78                      params_list.append(int(field.default))
79                  elif field.field_type == 'str':
80                      params_list.append(str(field.default))
81                  elif field.field_type == 'bool':
82                      params_list.append(str_to_bool(field.default))
83              else:
84                  params_list.append(value)
85              if len(params_list) == len(find_in_meta.fields):
86                  return find_in_meta.class_instance(*params_list)
87
88  ...
```

Listing B.5: PyCo create_instances method

From the listing B.5, it can be seen that the create_instances method takes parsed tree as a parameter.

We initialize an empty list called params_list. This list is responsible to store all the params needed for the respective class instance that is currently being worked upon, along with the current node of the tree.

We check if the tree has a node that has an attribute data, this is important because sometimes the value comes at the node of the tree. In the case, when it doesn't have attribute data, we return the tree itself(because there is value stored in tree).

From PyCo instance, we call a method field_metadata_data, that is referring to instance of InternalLanguageModel's attribute meta_data_class.

The meta_data_class is of the type MetaDataClass. This has a method called match_and_return_meta. When this method is called using a value of the node on the tree, the method checks the ClassInstanceMetaList to filter out and return the ClassInstanceMeta instance. The instance of ClassInstanceMeta is then going to be processed in the rest of the code of create_instance method of PyCo.

If match_and_return_meta method returns False, and is unable to find the instance of matched ClassInstanceMeta class, and the tree still has more branches

below the current node which we are processing, we further call the create_instance method recursively on the first child of the tree.

If match_and_return_meta returns metadata instance, then we check if the type of the instance is of abstract class. To skip the abstract classes and go in the further depth of tree, we just recursively call create_instances method again but now pass the first child of the tree.

If proceeded further than the previous steps, We count the fields of the fields attribute of ClassInstanceMeta instance. If the number of fields are zero, we just simply return the class instance.

Furthermore we define a variable called value and set it to None. This variable is supposed to store the value at the node of tree.

We then iterate over the fields using enumerate in Python to also enable us to have index of the iteration. We check if the field_type matches one of the type str, float, int, dict, list. If the match is found we process it accordingly.

At this point we check if the metadata of the field has Optional attribute set to true or not, If Optional is not set to tree then call the find_value_in_tree function.

This function takes tree and index of the field as a parameter and return the value at the node.

```
def find_value_in_tree(tree, index=None):
    if hasattr(tree, 'children'):
        if hasattr(tree.children[index], 'value'):
            return tree.children[index].value
        if tree.children[index] is None:
            return 'OPTIONAL'

        if len(tree.children[index].children) != 0:
            if isinstance(tree.children[index].children, list):
                return str(tree.children[index].children[0])
            return find_value_in_tree(tree.children[index].children,
    index)
        elif hasattr(tree.children[index], "data"):
            return tree.children[index].data
    else:
        return tree
```

Listing B.6: find_values_in_tree helper function

The helper function find_values_in_tree takes tree and index of the field as a parameter and try to find the value recursively in the depth of that tree node. This function checks all the possible cases in which the node can have value.

When find_values_in_tree returns the value at the tree node, we convert this

value into its proper respective type. After storing the value we check if the length of children in the tree is more than 0, and if so we remove that child node from the tree.

Removing of the child is required because in the cases of n^{th} depth of classes relationships and inheritance, sometimes a node has extra previous values which have already been processed and the instance related to them, had already been created.

There is list and dictionary implementation in the create_instances method, which generate a list by generating elements recursively. In the list and dictionary implementation, we also make use of Lark find_data function, which returns a tree by filtering the tree according to node of tree value.

In the case if that field metadata had optional set to true, we take the default value defined in the field metadata and assign it to variable value.

Lastly we append this value to params_list and create the instance of the class, if all the fields have been successfully added to params_list and return it.

B.2.2 PyCo class helper functions

The str_to_bool function

This function is used to represent boolean expressions in Python according to the parameter passed to it.

```
def str_to_bool(param):
    if param == 'true':
        return True
    elif param == 'false':
        return False
    else:
        return False
```

Listing B.7: str_to_bool function

The create_dict function

This function converts a list into key value pair to form a dictionary and return it.

```
def create_dict(a_list):
    a_dict = {}
    if len(a_list) < 2:
        raise ValueError("Dictionary List Not Valid")
    for i in range(0, len(a_list), 2):
        a_dict[a_list[i]] = a_list[i + 1]
```

```
    return a_dict
```

Listing B.8: create_dict function

The find_value_in_tree function

This function is used to parse the tree and go to depth of the tree branch to find the value at the node of the tree. The function use recursion to achieve its goal.

```
def find_value_in_tree(tree, index=None):
    if hasattr(tree, 'children'):
        if hasattr(tree.children[index], 'value'):
            return tree.children[index].value
        if tree.children[index] is None:
            return 'OPTIONAL'

        if len(tree.children[index].children) != 0:
            if isinstance(tree.children[index].children, list):
                return str(tree.children[index].children[0])
            return find_value_in_tree(tree.children[index].children,
    index)
        elif hasattr(tree.children[index], "data"):
            return tree.children[index].data
    else:
        return tree
```

Listing B.9: find_value_in_tree function

B.3 Grammar generator

The Grammar generator(Grammar class) in PyCo is responsible for generating grammar that is passed onto Lark to set up language syntax and semantics rules for language sentence processing.

The generate_grammar method of the Grammar class takes the metadata list from the internal language model and generates the grammar in EBNF form. The method filters out the language model objects for fields metadata of every class defined in the metamodel and based upon the metadata defined in them generate a string and concatenate the string appropriately to generated_grammar attribute.

```
...
class Grammar:
    """This class is responsible to generate language metadata model
    classes and generate grammar
```

```
        for Lark parser"""

    def __init__(self, tokens, meta_data_class):
        self.tokens = tokens
        self.meta_data_class = meta_data_class
        self.meta_ref_list = meta_data_class.meta_ref_list
        self.generated_grammar = """"""
...
```

Listing B.10: Grammar class structure

The Grammar class also takes token dictionary as a parameter, if tokens are to be globally defined for the DSL.

Initial attributes of tokens is set to the parameter tokens and an attribute called meta_data_class is set to parameter meta_data_class.

An important attribute known as meta_ref_list is also created, which refers to the meta_ref_list inside the parameter meta_data_class.

Attribute meta_data_class have the structural classes, of the internal language model of the defined DSL in PyCo.

These classes hold the metadata which will be processed to generate the resultant grammar.

B.3.1 Grammar class methods

There is one method in Grammar class which PyCo uses internally to generate grammar, when PyCo class is initialized.

1. generate_grammar

In the listing B.3, PyCo instance in its attributes, calls the generate_grammar method of the Grammar class.

```
...
class Grammar:
    """This class is responsible to generate language metadata model
    classes and generate grammar for Lark parser"""

    def __init__(self, tokens, meta_data_class):
        self.tokens = tokens
        self.meta_data_class = meta_data_class
        self.meta_ref_list = meta_data_class.meta_ref_list
        self.generated_grammar = """"""

    def generate_grammar(self):
```

```
12      """generate_grammar method generates the final grammar from
    the generated metadata model"""
13      same_parent_list = []
14      processed_classes = []
15      self.meta_ref_list = unique_set_list(self.meta_ref_list)
16      for ref in self.meta_ref_list:
17          ref_class = None
18          compound_result = ""
19
20          generate_list = list(filter(lambda obj: obj.ref_class_name
     == ref, self.meta_data_class.FieldsMetaDataList))
21          for fieldGen1 in generate_list:
22
23              compound_result += get_result_string(fieldGen1.
    field_meta, fieldGen1.field_type_name_ref, fieldGen1.of_type,
24                                                   fieldGen1.
    field_type, fieldGen1.optional).lower() + ' '
25              ref_class = fieldGen1.ref_class
26              if ref_class not in same_parent_list:
27                  same_parent_list.append(ref_class)
28
29          if ref not in processed_classes:
30              processed_classes.append(ref)
31              self.generated_grammar += ref + ' : ' +
    apply_decorators(ref_class, compound_result) + '\n'
32
33      for fieldGen in self.meta_data_class.AbInheritanceClassList:
34          class_names_list = fieldGen.class_names_list
35          or_list = get_or_name_list(class_names_list) + '\n'
36          if fieldGen.class_instance_ref_name + ' : ' in self.
    generated_grammar:
37              self.generated_grammar = self.generated_grammar.
    replace(fieldGen.class_instance_ref_name + ' : ', '')
38          self.generated_grammar += fieldGen.class_instance_ref_name
     + ' : ' + or_list
39
40          temp_result = ""
41          for element in class_names_list:
42              if element.ref_class not in same_parent_list:
43                  self.generated_grammar += (element.name.lower() +
    ' : '
44                                             + apply_decorators(
    element.ref_class, temp_result) + '\n')
45
46      self.generated_grammar = add_tokens_to_grammar(self.
```

```
    generated_grammar, tokens=self.tokens)
47          self.generated_grammar = add_imports_to_grammar(self.
    generated_grammar)
48          return self.generated_grammar
```

Listing B.11: generate_grammar method of Grammar class

The generate_grammar method declares two empty lists. One of the list is called same_parent_list and the other is called processed_classes.

Attribute meta_ref_list is converted into in a list of unique values by passing the attribute value to a helper function called unique_set_list. This meta_ref_list hold the names of the all the processed classes by the internal language model.

Initially we set the variable ref_calss to None and variable compound_result to an empty string.

In the variable generate_list we are generating a list of instances which match the name in the meta_ref_list for the current iteration of the loop. The list is generated using the FieldsMetaDataList and filter function of Python.

We further iterate the generate_list to process each field of a class.

The function get_result_string is used to generate the string by the use of metadata to correctly form the grammar. This function takes the metadata of the field, type of the field, the name of the field, optional and the class type of the class to which the field belong to in order to process and generate grammar string.

After processing each field and generating the grammar string, we append name of the class to same_parent_list, if they are not already present there. We are only adding unique to prevent duplication and avoid ambiguity in the final generation of grammar. Furthermore we apply the main class decorators metadata, if the class have any and generate the final grammar strings.

Once all the iteration are finished, we start a new for loop for the grammar string generation of class inheritance relationships.

The AbInheritanceClassList contains the information about all the classes and their related subclasses. We iterate over each instance ot AbInheritanceClass in the AbInheritanceClassList and generate a string which corresponds to inheritance between the classes.

The grammar is generated very carefully to avoid any type of ambiguity in the grammar, and the grammar string generated in the first for loop and grammar string generated in the second for loop are properly combined.

Two functions, add_token_to_grammar, to add the globally defined tokens to the grammar and add_imports_to_grammar to add the imports of Lark library to the grammar are called in the end.

The finalized version of the EBNF grammar string is stored and available in the generated_grammar attribute of Grammar class instance.

B.3.2 Grammar class helper function

The get_result_string function

The get_result_string method is used to generate the grammar strings, by adding grammar rules to it based upon the metadata defined in the fields. In case of int, str, float and bool, only the the token, before and after dictionary keys are applied on, from the metadata defined in the field of the class.

In case of list and dict, there are additional parameters, that can be provided in order for the grammar to be successfully processed. Both list and dict have same metadata keys(separator, before and after) but dict have one additional key(element_separator).

If some class has a field which if referencing to another defined class as a type, then only the before and after keys are applicable for metadata dictionary.

```
def get_result_string(field_data_meta, type_of, class_name, field_type
    , optional):
    """this function applies the field metadata of each field in a
    class to generate relevant
    grammar string"""
    keys = field_data_meta.keys()
    result = ''

    if type_of == 'str':
        if 'token' in keys:
            result = field_data_meta['token']
        if 'before' in keys:
            result = wrap_in_doubleQuote(field_data_meta['before']) +
    result

        if 'after' in keys:
            result = result + wrap_in_doubleQuote(field_data_meta['
    after'])

    if type_of == 'bool':
        if 'token' in keys:
            result = field_data_meta['token']
        if 'before' in keys:
            result = wrap_in_doubleQuote(field_data_meta['before']) +
    result
```

```
21
22          if 'after' in keys:
23              result = result + wrap_in_doubleQuote(field_data_meta['
    after'])
24
25      if type_of == 'float':
26          if 'token' in keys:
27              result = field_data_meta['token']
28
29          if 'before' in keys:
30              result = wrap_in_doubleQuote(field_data_meta['before']) +
    result
31
32          if 'after' in keys:
33              result = result + wrap_in_doubleQuote(field_data_meta['
    after'])
34
35      if type_of == 'int':
36          if 'token' in keys:
37              result = field_data_meta['token']
38
39          if 'before' in keys:
40              result = wrap_in_doubleQuote(field_data_meta['before']) +
    result
41
42          if 'after' in keys:
43              result = result + wrap_in_doubleQuote(field_data_meta['
    after'])
44
45      if type_of == 'list':
46          if 'separator' not in keys:
47              result = class_name + '*'
48
49          if 'separator' in keys:
50              result = ('[' + class_name + ' (' + wrap_in_doubleQuote(
51                  field_data_meta['separator']) + class_name + ')*]')
52
53          if 'before' in keys:
54              result = wrap_in_doubleQuote(field_data_meta['before']) +
    result
55
56          if 'after' in keys:
57              result = result + wrap_in_doubleQuote(field_data_meta['
    after'])
58
```

```
    if type_of == 'dict':
        key = field_type.__dict__['__args__'][0].__name__
        value = field_type.__dict__['__args__'][1]
        result = ('[' + key.lower() + '":"' + value.__name__.lower() +
                  ' (' + wrap_in_doubleQuote(field_data_meta['
element_separator']) + ' '
                  + key.lower() + ' ":" ' + value.__name__.lower() + '
)*]')
        if 'before' in keys:
            result = wrap_in_doubleQuote(field_data_meta['before']) +
result

        if 'after' in keys:
            result = result + wrap_in_doubleQuote(field_data_meta['
after'])

    if type_of not in ['int', 'list', 'str', 'float', 'dict', 'bool']:
        result = class_name.lower()
        if 'before' in keys:
            result = wrap_in_doubleQuote(field_data_meta['before']) +
result

        if 'after' in keys:
            result = result + wrap_in_doubleQuote(field_data_meta['
after'])
    if optional:
        result = "[" + result + "]"
    return result


```

Listing B.12: get_result_string helper function

The wrap_in_double_quote function

This function is used to make code cleaner and easy to refactor. It wraps a given parameter value with double quotes and return it.

```
def wrap_in_double_quote(value):
    """this function wraps the string in quotes"""
    return ' "' + str(value) + '" '
```

Listing B.13: wrap_in_double_quote function

The space function

```
def space():
    """this function returns a space string"""
    return ' '
```

Listing B.14: space function

The apply_decorators function

This functions checks if the class has some predefined decorators. If so, the function finds the value of the decorator and applies onto the grammar string. The keys this function looks for is before and after.

```
def apply_decorators(ref_class, compound_result: str) -> str:
    """this function applies the decorators to the final grammar
   string of a class"""
    if hasattr(ref_class, 'decorators'):
        result_decorated = ""
        if 'before' in ref_class.decorators:
            result_decorated = wrap_in_double_quote(ref_class.
   __parser_syntax['before']) + space() + compound_result
        if 'after' in ref_class.decorators:
            result_decorated = result_decorated + space() +
   wrap_in_double_quote(ref_class.__parser_syntax['after'])

        return result_decorated
    return compound_result
```

Listing B.15: apply_decorators function

The get_or_name_list function

The classes and subclasses which are related through inheritance from the abstract classes, have to be represented in EBNF form in the final grammar. To make this happen the final grammar string is appended with the string containing the logic for inheritance. This helps to fully validate and implement correct syntax and semantics of the resultant grammar.

```
def get_or_name_list(or_list) -> str:
    """this functions generates a grammar string for all inherited
   classes and the class
        from which the subclasses are inheriting from.
        if b_class and c_class are inheriting from a_class then:
        result = a_class : b_class | c_class
    """
    result_list = ""
```

```
    for index, element in enumerate(or_list):
        if index < len(or_list) - 1:
            result_list += element.name.lower() + ' | '
        if index == len(or_list) - 1:
            result_list += element.name.lower()
    return result_list
```

Listing B.16: get_or_name_list function

In the listing B.1 example, When the metamodel of the language is processed, and the get_or_name_list is called. The result of this function is

```
"examplelang : langnumber | langstring | langarray"
```

The add_imports_to_grammar function

This function is used to add Lark specific imports and declarations to the grammar string. There imports include

- import common.ESCAPED_STRING (importing predefined regular expression for string)
- import common.SIGNED_NUMBER (importing predefined regular expression for numbers)
- import common.WS (importing white space from Lark)
- ignore WS (to ignore white space in the language sentence)

```
def add_imports_to_grammar(grammar):
    """imports can be added to the grammar string so we can use
    builtin Lark functions"""
    imp_str = """\n%import common.ESCAPED_STRING\n%import common.
    SIGNED_NUMBER\n%import common.WS\n%ignore WS\n"""
    grammar = grammar + imp_str
    return grammar
```

Listing B.17: add_imports_to_grammar function

The add_tokens_to_grammar function

This function is used to append the tokens dictionary key values as a string to grammar string, This token dictionary was passed to PyCo class instance during initialization.

```
def add_tokens_to_grammar(grammar, tokens):
    """this function is used to append defined tokens to generated
    grammar"""
    token_str = "\n"
    for i in tokens:
        token_str += i.lower() + ' : ' + tokens[i] + '\n'
    grammar = grammar + token_str
    return grammar
```

Listing B.18: add_tokens_to_grammar function

B.4 Internal language model

Creation of internal language model is the most important task of PyCo because based upon this, the grammar and class instances are going to be created in the later stages.

This model stores all the information of classes and subclasses that are linked together through inheritance.

This InternalLanguageModel class only takes two parameter to initialize. The first parameter base_class is used to find the classes and all the subclasses which are in relationship and generate class internal class instances structures structures(FieldsMetaData, AbInheritanceClass) to store metadata.

There is only one method(generate_grammar_meta_data) in this class and this method is always called during the initialization process. This method is used to start the process of finding, generating and storing the metadata of class fields.

The method first finds the list of subclasses, if the base class is being inherited by some other class.

If there are no subclasses, than we try to find if the class has an attribute named __dataclass_fields__.

If the class does have the attribute __dataclass_fields__, we find the fields of the class using dataclasses library fields function. We iterate on each of these fields to find the defined metadata. This metadata is converted into the instances of appropriate class structures(FieldsMetaData, AbInheritanceClass).

These class instances are further appended to the appropriate list of similar class structures(AbInheritanceClassList, FieldsMetaDataList).

```
class InternalLanguageModel:
    """This class is responsible to generate language metadata model
    classes and generate grammar for Lark parser"""
```

```
3
4    def __init__(self, base_class, tokens):
5        self.base_class = base_class
6        self.tokens = tokens
7        self.meta_data_class = MetaDataClass()
8        self.processed = []
9        self.field_data_type = ['int', 'float', 'str', 'dict', 'list',
     'bool', 'custom']
10       self.generate_grammar_meta_data(start_class=base_class, tokens
    =tokens)
11
12   def generate_grammar_meta_data(self, start_class, tokens):
13       """this method generates metadata classes for grammar generate
     and class instance generation process"""
14       subclasses_instances = get_all_subclasses_instances(
    start_class)
15       if len(subclasses_instances) > 0:
16           for x in subclasses_instances:
17               class_obj = AbInheritanceClass(
18                   class_instance=start_class,
19                   class_names_list=[ClassMetaData(class_instance=
    class_name) for class_name in subclasses_instances]
20               )
21               self.meta_data_class.add_unique_to_ab_inheritance(
    abInheritanceClass=class_obj)
22
23               if x not in self.processed:
24                   self.processed.append(x)
25                   self.generate_grammar_meta_data(x, tokens)
26
27       if hasattr(start_class, '__dataclass_fields__'):
28           fields_data = dataclasses.fields(start_class)
29
30           for field_data in fields_data:
31               if is_optional(field_data.type):
32                   if field_data.type.__args__[0].__class__.__name__
    == '_GenericAlias':
33                       class_type_name = get_class_type(field_data).
    __name__.lower()
34                       class_field = inspect.getmembers(field_data.
    type)[0][1][0]
35                       class_field_name = class_field.__name__.lower
    ()
36                       self.meta_data_class.
    add_unique_to_field_meta_data_list(
```

```
37                              FieldsMetaData(
38                                  ref_class_name=start_class.__name__.
    lower(),
39                                  field_name=field_data.name,
40                                  field_type=field_data.type.__args__
    [0],
41                                  field_meta=field_data.metadata,
42                                  field_names_list=list(start_class.
    __annotations__.keys()),
43                                  ref_class=start_class,
44                                  inner_class=class_field,
45                                  field_type_name_ref=class_type_name,
46                                  optional=True,
47                                  default=field_data.metadata['default'
    ],
48                                  of_type=class_field_name))
49                          if start_class.__name__.lower() not in self.
    meta_data_class.meta_ref_list:
50                              self.meta_data_class.meta_ref_list.append(
    start_class.__name__.lower())
51
52                          if class_field not in self.processed:
53                              self.processed.append(class_field)
54                              self.generate_grammar_meta_data(
    class_field, tokens)
55                      elif field_data.type.__args__[0].__name__.lower()
    in ['int', 'float', 'str', 'dict', 'list', 'bool',
56
        'optional']:
57                          self.meta_data_class.
    add_unique_to_field_meta_data_list(
58                              FieldsMetaData(
59                                  ref_class_name=start_class.__name__.
    lower(),
60                                  field_name=start_class.__name__.lower
    (),
61                                  field_type=type(start_class),
62                                  field_meta=field_data.metadata,
63                                  field_names_list=list(start_class.
    __annotations__.keys()),
64                                  ref_class=start_class,
65                                  inner_class=None,
66                                  optional=True,
67                                  default=field_data.metadata['default'
    ],
```

```
68                                  field_type_name_ref=field_data.type.
__args__[0].__name__,
69                                  of_type=field_data.type.__args__[0].
__name__))
70
71                          if start_class.__name__.lower() not in self.
meta_data_class.meta_ref_list:
72                              self.meta_data_class.meta_ref_list.append(
start_class.__name__.lower())
73                      elif field_data.type.__args__[0].__name__.lower()
not in ['int', 'float', 'str', 'dict', 'list',
74
            'bool']:
75                          self.meta_data_class.
add_unique_to_field_meta_data_list(
76                              FieldsMetaData(
77                                  ref_class_name=start_class.__name__.
lower(),
78                                  field_name=field_data.name,
79                                  field_type=field_data.type,
80                                  field_meta=field_data.metadata,
81                                  field_names_list=list(start_class.
__annotations__.keys()),
82                                  ref_class=start_class,
83                                  inner_class=None,
84                                  optional=True,
85                                  default=field_data.metadata['default'
],
86                                  field_type_name_ref=field_data.type.
__name__,
87                                  of_type=field_data.type.__name__))
88                          if field_data.type.__args__[0].__name__.lower
() not in ['int', 'float', 'str', 'dict', 'list',
89
              'bool']:
90                              self.meta_data_class.meta_ref_list.append(
field_data.type.__args__[0].__name__.lower())
91                          self.generate_grammar_meta_data(field_data.
type, tokens)
92
93                  elif field_data.type.__class__.__name__ == '
_GenericAlias':
94                      class_type_name = get_class_type(field_data).
__name__.lower()
95                      class_field = inspect.getmembers(field_data.type)
```

```
[0][1][0]
                class_field_name = class_field.__name__.lower()
                self.meta_data_class.
add_unique_to_field_meta_data_list(
                    FieldsMetaData(
                        ref_class_name=start_class.__name__.lower
(),
                        field_name=field_data.name,
                        field_type=field_data.type,
                        field_meta=field_data.metadata,
                        field_names_list=list(start_class.
__annotations__.keys()),
                        ref_class=start_class,
                        inner_class=class_field,
                        field_type_name_ref=class_type_name,
                        optional=False,
                        default='OPTIONAL_DEFAULT',
                        of_type=class_field_name))
                if start_class.__name__.lower() not in self.
meta_data_class.meta_ref_list:
                    self.meta_data_class.meta_ref_list.append(
start_class.__name__.lower())

                if class_field not in self.processed:
                    self.processed.append(class_field)
                    self.generate_grammar_meta_data(class_field,
tokens)
            elif field_data.type.__name__.lower() in self.
field_data_type:
                self.meta_data_class.
add_unique_to_field_meta_data_list(
                    FieldsMetaData(
                        ref_class_name=start_class.__name__.lower
(),
                        field_name=start_class.__name__.lower(),
                        field_type=type(start_class),
                        field_meta=field_data.metadata,
                        field_names_list=list(start_class.
__annotations__.keys()),
                        ref_class=start_class,
                        inner_class=None,
                        optional=False,
                        default='OPTIONAL_DEFAULT',
                        field_type_name_ref=field_data.type.
__name__,
```

```
                            of_type=field_data.type.__name__))

                    self.field_data_type.append(start_class.__name__.
    lower())
                    if start_class.__name__.lower() not in self.
    meta_data_class.meta_ref_list:
                        self.meta_data_class.meta_ref_list.append(
    start_class.__name__.lower())
                elif field_data.type.__name__.lower() not in self.
    field_data_type:
                    self.meta_data_class.
    add_unique_to_field_meta_data_list(
                        FieldsMetaData(
                            ref_class_name=start_class.__name__.lower
    (),
                            field_name=field_data.name,
                            field_type=field_data.type,
                            field_meta=field_data.metadata,
                            field_names_list=list(start_class.
    __annotations__.keys()),
                            ref_class=start_class,
                            inner_class=None,
                            optional=False,
                            default='OPTIONAL_DEFAULT',
                            field_type_name_ref=field_data.type.
    __name__,
                            of_type=field_data.type.__name__)
                    )

                    if field_data.type.__name__.lower() not in self.
    field_data_type:
                        self.meta_data_class.meta_ref_list.append(
    field_data.type.__name__.lower())
                    self.generate_grammar_meta_data(field_data.type,
    tokens)
```

Listing B.19: internal language model

```
...
@dataclasses.dataclass
class MetaDataClass:
    AbInheritanceClassList: List[AbInheritanceClass]
    FieldsMetaDataList: List[FieldsMetaData]
    ClassInstanceMetaList: List[ClassInstanceMeta]

    def __init__(self):
```

```
        self.AbInheritanceClassList: List[AbInheritanceClass] = []
        self.FieldsMetaDataList: List[FieldsMetaData] = []
        self.ClassInstanceMetaList: List[ClassInstanceMeta] = []
...
```

Listing B.20: MetaDataClass for internal language model

B.4.1 Class structures for InternalLanguageModel

The AbInheritanceClass has the structure to store the metadata of classes which forms an inheritance relationship with other classes. It store the structural metadata of the class in its attributes. It takes class_instance and class_names_list as a parameter.

The class_names_list is an attribute of AbInheritanceClass which has a type List[ClassMetaData]. The ClassMetaData stores in its attribute, the name of class and its class instance.

The next important class is FieldsMetaData class. This class is used to store the information about the class individual fields and relationships. When fields are being processed by the InternalLanguageModel class, all of the metadata related to individual fields gets stored using this class instances. The class takes a numbers of parameters to get initialized. All the parameters provided to this class are attributes of this class instance.

There is another class named ClassInstanceMeta. This class has an attribute fields which is a list of type Field class. The Field class has a structure to store the name of the class, the type of the class provided and the tree_ref attributes. This class keeps the count of the fields in an attribute called field_count.

There is a very important method in ClassInstanceMeta class called set_fields. If the class has an attribute known as __annotations__, then it is possible to get information about the fields of that class. This method checks if that attribute is present and based on its presence can filter out the fields in the class. The function find_type is used to find the exact type name of the class. Similar to this function the find_inner checks the field element, to find the type of the class in the case of both generic class and non generic class. The find_inner function use inspect library to find the correct type.

The MetaDataClass uses both the **AbInheritanceClass** and **FieldsMetaData** classes to combine them into one common structure to be used in the creating of main DSL class instances.

There are four methods available in MetaDataClass.

1. add_unique_to_ab_inheritance

2. add_unique_to_field_meta_data_list
3. generate_meta_for_create_instance
4. match_and_return_meta

The method add_unique_to_ab_inheritance is simply a method that checks if the AbInheritanceClassList attribute already contains the class with the type of AbInheritanceClass. It also checks if the reference class type (attribute of AbInheritanceClass) is not in any of the basic types like, list, str, int, dict and float. If both conditions returns true, an instance of AbInheritanceClass is added to AbInheritanceClassList.

The method add_unique_to_field_meta_data_list simply add the provided FieldsMetaData class instance to the FieldsMetaDataList attribute.

The method generate_meta_for_create_instance is the method which makes use of both AbInheritanceClassList and FieldsMetaDataList. The method iterate over both of these and lists and create a common structure composed of class instance of ClassInstanceMeta. This ClassInstanceMeta is then appended to ClassInstanceMetaList attribute, in case if ClassInstanceMetaList attribute does not contain the instance of ClassInstanceMeta.

```
...
@dataclasses.dataclass
class AbInheritanceClass:
    def __init__(self, class_instance, class_names_list):
        self.class_instance_ref_name = class_instance.__name__.lower()
        self.ref_class = class_instance
        self.class_names_list: List[ClassMetaData] = class_names_list

        if type(class_instance).__name__ != '_GenericAlias':
            self.class_type_name = 'abc'


@dataclasses.dataclass
class ClassMetaData:
    def __init__(self, class_instance):
        self.name = class_instance.__name__
        self.ref_class = class_instance


@dataclasses.dataclass
class FieldsMetaData:
    def __init__(self, ref_class_name, field_name, field_type,
                 field_meta, field_names_list, ref_class,
```

```
                inner_class, field_type_name_ref, of_type, optional,
    default):
        self.ref_class_name = ref_class_name
        self.field_name = field_name
        self.field_type = field_type
        self.field_meta = field_meta
        self.field_names_list = field_names_list
        self.ref_class = ref_class
        self.inner_class = inner_class
        self.field_type_name_ref = field_type_name_ref
        self.of_type = of_type
        self.optional = optional
        self.default = default

        try:
            self.field_type_name = field_type.__name__
        except Exception as e:
            self.field_type_name = field_type.__class__.__name__

    def __hash__(self):
        return hash(self.field_name)


@dataclasses.dataclass
class Field:
    def __init__(self, field_name: str, field_type: str, tree_ref: str
    , default: Any):
        self.field_name = field_name
        self.field_type = field_type
        self.tree_ref = tree_ref
        self.default = default


@dataclasses.dataclass
class ClassInstanceMeta:
    def __init__(self, class_instance, type_class, of_type):
        self.class_instance = class_instance
        self.class_instance_name = class_instance.__name__.lower()
        self.fields: List[Field] = []
        self.field_count = len(self.fields)
        self.set_fields()
        self.type_class = type_class
        self.of_type = of_type

    def set_fields(self):
```

```
67          if hasattr(self.class_instance, '__annotations__'):
68              for index, field in enumerate(self.class_instance.
    __annotations__):
69                  element = self.class_instance.__annotations__[field]
70                  default_value = dataclasses.fields(self.class_instance
    )[index].metadata.get('default')
71                  find = find_inner(element)
72                  type_element = find_type_name(element)
73
74                  field_element = Field(field, type_element, find,
    default_value)
75                  self.fields.append(field_element)
76
77
78  @dataclasses.dataclass
79  class MetaDataClass:
80      def __init__(self):
81          self.AbInheritanceClassList: List[AbInheritanceClass] = []
82          self.FieldsMetaDataList: List[FieldsMetaData] = []
83          self.ClassInstanceMetaList: List[ClassInstanceMeta] = []
84          self.meta_ref_list: List[str] = []
85
86      def add_unique_to_ab_inheritance(self, abInheritanceClass:
    AbInheritanceClass):
87          type_list = ['str', 'float', 'int', 'dict', 'list']
88          if not self.AbInheritanceClassList.__contains__(
    abInheritanceClass) \
89                  and abInheritanceClass.class_instance_ref_name not in
    type_list:
90              self.AbInheritanceClassList.append(abInheritanceClass)
91
92      def add_unique_to_field_meta_data_list(self, fieldsMetaData:
    FieldsMetaData):
93          contain = contains(self.ClassInstanceMetaList, lambda x: x.
    ref_class == fieldsMetaData.field_type)
94          if not contain:
95              self.FieldsMetaDataList.append(fieldsMetaData)
96
97      def generate_meta_for_create_instance(self):
98          field_metadata_set_list = unique_set_list(self.
    FieldsMetaDataList)
99          for element in field_metadata_set_list:
100             self.ClassInstanceMetaList.append(
101                 ClassInstanceMeta(element.ref_class, element.
    field_type_name_ref, element.of_type))
```

```
102
103         for element in self.AbInheritanceClassList:
104             contain = contains(self.ClassInstanceMetaList,
105                                lambda x: x.class_instance_name ==
    element.ref_class.__name__.lower())
106             if not contain:
107                 self.ClassInstanceMetaList.append(
108                     ClassInstanceMeta(element.ref_class, element.
    class_type_name, None))
109
110             for class_or_instance in element.class_names_list:
111                 contain = contains(self.ClassInstanceMetaList,
112                                    lambda x: x.class_instance_name ==
    class_or_instance.name.lower())
113                 if not contain:
114                     self.ClassInstanceMetaList.append(
115                         ClassInstanceMeta(class_or_instance.ref_class,
116                                           type(class_or_instance.
    ref_class), None))
117
118     def match_and_return_meta(self, node_data):
119         return contains(self.ClassInstanceMetaList, lambda x: x.
    class_instance_name == node_data)
```

Listing B.21: Type of classes for MetaDataClass attributes

B.4.2 Internal language model helper functions

The get_class_type function

This function returns the type of a class

```
1 def get_class_type(field_data):
2     """this function returns the type of a class"""
3     if typing.get_origin(field_data.type) is not None:
4         return typing.get_origin(field_data.type)
5     else:
6         return field_data.type
```

Listing B.22: get_class_type function

The get_all_subclasses_instances function

If we pass a class as a parameter to this function, this function will find all the subclasses which are inheriting from that class and return the list of it.

```
def get_all_subclasses_instances(class_to_check):
    """this function finds all the class which inherits from the
   class_to_check class"""
    all_subclasses = []
    for subclass in class_to_check.__subclasses__():
        if subclass not in all_subclasses:
            all_subclasses.append(subclass)
        get_all_subclasses_instances(subclass)

    return all_subclasses
```

Listing B.23: get_all_subclasses_instance function

The find_type function

The find_type function can find the type of class instances passed as a parameter to it. This function is different from function get_class_type as it also processes generic classes and return the name of the type in lower case letters.

```
def find_type(field_data):
if type(field_data).__name__ == '_GenericAlias':
  class_type_name = field_data._name.lower()
  return class_type_name
else:
  return field_data.__name__.lower()
```

Listing B.24: find_type function

The find_inner function

This function returns the inner type of the class if the class has a type _GernericAlias e.g List, Dict etc. It also finds the type if the class type doesn't belong to _GernericAlias type.

```
def find_inner(field_data):
    """this function returns the inner type of the class if
   _GernericAlias
        type from typing is used e.g List, Dict etc.
        It also finds the type if the class type doesn't belong to
   _GernericAlias
    """
    if type(field_data).__name__ == '_GenericAlias':
        return inspect.getmembers(field_data)[0][1][0].__name__.lower
   ()
    else:
```

```
9         return type(field_data)
```

Listing B.25: find_inner function

The find_type_name function

This function returns the name of the field in all possible types of fields, generic or non generic. The function uses inspect library of Python to find its results.

```
1  def find_type_name(field_data):
2      """this function returns the class name of the type given to a
       field"""
3      if type(field_data).__name__ == '_GenericAlias':
4          try:
5              class_type_name = field_data._name.lower()
6          except:
7              class_type_name = inspect.getmembers(field_data)[0][1][0].
       __name__.lower()
8          return class_type_name
9      else:
10         if is_optional(field_data):
11             return field_data.__args__[0].__name__.lower()
12         return field_data.__name__.lower()
```

Listing B.26: find_type_name function

B.4.3 Common helper functions

The contains function

This function takes a list of any type of values and a lambda function and return the filtered value if matched. If the value is not found then it return False.

```
1  def contains(list_values, filter_contain):
2  for x in list_values:
3      if filter_contain(x):
4          return x
5  return False
```

Listing B.27: contains function

The is_optional function

This function checks if the field is of optional type of not and return a boolean value.

```
def is_optional(field):
    """this functions return True or False by checking if the field is
    Optional or not"""
    return typing.get_origin(field) is typing.Union and type(None) in
   typing.get_args(field)
```

Listing B.28: is_optional function

The unique_set_list function

This function takes a list and remove the duplicates by using Python set, but it does it in such a way that the order of the list is persevered and returns back a list.

```
def unique_set_list(sequence):
    """this function creates a unique list of elements and remove the
   duplicates"""
    seen = set()
    return [x for x in sequence if not (x in seen or seen.add(x))]
```

Listing B.29: unique_set_list function

The syntax decorator function

This function defines a decorator to add before and after token to the class definition.

```
def syntax(before=None, after=None):
    """this decorator function takes two input before and after
        either of the inputs can be provided of none of them.
        The decorates are appended to the class metadata.
    """
    def syntax_decorator(class_to_apply_decorator_on):
        class_to_apply_decorator_on.__parser_syntax = {'before':
   before, 'after': after}
        class_to_apply_decorator_on.decorators = []
        if before is not None:
            class_to_apply_decorator_on.decorators.append('before')
        if after is not None:
            class_to_apply_decorator_on.decorators.append('after')
        return class_to_apply_decorator_on
    return syntax_decorator
```

Listing B.30: syntax decorator function