

FLEX
A PROGRAMMING LANGUAGE WITH MINIMAL SYNTAX
THAT CAN UNDERSTAND NATURAL LANGUAGE

A PROJECT REPORT

Submitted in the partial fulfillment of the requirements

for the award of the degree of

BACHELOR OF TECHNOLOGY

IN

COMPUTER ENGINEERING



Submitted by

Gourav Suri (14BCS0040)
Sanjay Kumar (14BCS0046)
Syed Faheel Ahmad (14BCS0067)

Under the supervision of

Danish Raza Rizvi
(Assistant Professor)

DEPARTMENT OF COMPUTER ENGINEERING
FACULTY OF ENGINEERING AND TECHNOLOGY
JAMIA MILLIA ISLAMIA, NEW DELHI - 110025
(YEAR – 2018)

CERTIFICATE

This is to certify that the dissertation/project report titled “**Flex: a programming language with minimal syntax that can understand natural language**” submitted by **Gourav Suri** (14BCS0040), **Sanjay Kumar** (14BCS0046) and **Syed Faheel Ahmad** (14BCS0067) for the partial fulfillment of the requirements for the award of the degree of Bachelor of Technology in Computer Engineering, is a record of bona fide work carried out by them under my guidance and supervision at the Department of Computer Engineering, Faculty of Engineering & Technology, Jamia Millia Islamia, New Delhi.

The matter embodied in this project work has not been submitted earlier for the award of any degree or diploma to the best of my knowledge.

Place: New Delhi

Date: 28th May, 2018

Prof. M.N. Doja

Professor and Head,

Dept. of Computer Engg.

Jamia Millia Islamia

Danish Raza Rizvi

Assistant Professor,

Dept. of Computer Engg.

Jamia Millia Islamia

ACKNOWLEDGEMENT

We would like to thank our mentor Danish Raza Rizvi (Assistant Professor, Dept. of Computer Engineering) for giving us the opportunity to take up the project. We thank him for his immense guidance, and appreciate his timely engagement.

We would like to extend special gratitude to the assistants and lab coordinators of the Department for providing us the infrastructural facilities necessary to complete the project.

Gourav Suri

14BCS0040

Student

Sanjay Kumar

14BCS0046

Student

Syed Faheel Ahmad

14BCS0067

Student

TABLE OF CONTENTS

1.	Abstract	6
2.	Introduction	7
3.	Technical Background	9
3.1.	Natural Language Understanding (NLU)	9
4.	Rasa, the NLU framework	11
4.1.	Working of Rasa	11
4.2.	Using Rasa locally from Python	15
4.3.	Advantages of Rasa over other NLU frameworks	18
5.	Language features	19
6.	Example code snippets	22
7.	Proposed method	23
8.	Statements currently supported by Flex	25
8.1.	Declaration	25
8.2.	Assignment	27
8.3.	Control	28
8.4.	Input / output	30
8.5.	Data Types	31
8.5.1.	Built-in	31
8.5.2.	User-defined	31
8.6.	Containers	32
8.7.	Functions	32
8.8.	Comments	32

9.	Snippets of code dictionaries for various target languages	33
9.1.	Python Dictionary	33
9.2.	C++ Dictionary	33
9.3.	Java Dictionary	34
10.	Result & Output	36
11.	Conclusion	41
12.	Future prospective	42
13.	Programming environment & Tools used	43
14.	References	44

TABLE OF FIGURES

Fig 1.1 Major Programming Languages	8
Fig 2.1 An example of NLU in action	10
Fig 3.1 Rasa's NLU processing pipeline for a chat bot	11
Fig 3.2 How Rasa build conversational software with ML	12
Fig 4.1 Graphical breakdown of the language features of Flex	21
Fig 5.1 Training the model	36
Fig 5.2 Transpiling Flex code to Python	37
Fig 5.3 Transpiling Flex code to CPP	38

ABSTRACT

The bird's eye view of our project is to create a programming language, that we call *Flex*, which has a minimal syntax, but more importantly it can be written in natural language, i.e. statements have no fixed syntax - they are written in English in the imperative mood.

This would allow both beginners and experts to prototype large and/or complex algorithms without worrying about the platform they're programming on, the syntax, and other less important details that hinder a programmer's productivity.

INTRODUCTION

When computers were invented, programmers used to program in Assembly language. When programming in Assembly, programmers had (and still have) to deal with registers, memory addresses and call stacks. This was quite complex, and a small error could completely change the working of the program. To simplify programming, high-level languages such as C, C++, Java, Python, Ruby, Kotlin etc. were developed over the years, which provided programmers a higher level of abstraction from machine language. Rather than dealing with registers, memory addresses and call stacks, high-level languages deal with variables, arrays, objects, complex arithmetic or boolean expressions, subroutines and functions, loops, threads, locks, and other abstract computer science concepts, with a focus on usability over optimal program efficiency. Unlike low-level assembly languages, high-level languages have few, if any, language elements that translate directly into a machine's native opcodes. Other features, such as string handling routines, object-oriented language features, and file input/output, may also be present.

One thing to note about high-level programming languages is that these languages allows the programmer to be detached and separated from the machine. That is, unlike low-level languages like assembly or machine language, high-level programming can amplify the programmer's instructions and trigger a lot of data movements in the background without their knowledge. The responsibility and power of executing instructions have been handed over to the machine from the programmer.

We intend to take this a step further, by designing a programming language that allows programmers to express their thoughts more clearly, in natural language, and in a form that is not bound to a fixed syntax, but is rather flexible. This would allow programmers to quickly code the logic

for a task they want the computer to perform, or to develop complex algorithms with a high level of abstraction, without worrying specifics of the language's syntax or about the platform they're programming on. This would greatly help improve their productivity, and will also help beginners to code easily. Also, we believe that such a programming language - with a minimalistic and flexible syntax - will also allow disabled programmers to code, as they would be able to enter code using speech-to-text converters and other means much more easily as compared to currently available high-level languages.



Fig. 1.1 Major programming languages

TECHNICAL BACKGROUND

In this section, we'll discuss a little about the different concepts we'll be using in our project.

NATURAL LANGUAGE UNDERSTANDING (NLU)

Natural language understanding (NLU) or natural language interpretation (NLI) is a subtopic of natural language processing in artificial intelligence that deals with machine reading comprehension.

The umbrella term "natural language understanding" can be applied to a diverse set of computer applications, ranging from small, relatively simple tasks such as short commands issued to robots, to highly complex endeavors such as the full comprehension of newspaper articles or poetry passages. Many real world applications fall between the two extremes, for instance text classification for the automatic analysis of emails and their routing to a suitable department in a corporation does not require in depth understanding of the text, but needs to deal with a much larger vocabulary and more diverse syntax than the management of simple queries to database tables with fixed schemata.

Throughout the years various attempts at processing natural language or English-like sentences presented to computers have taken place at varying degrees of complexity. Some attempts have not resulted in systems with deep understanding, but have helped overall system usability.

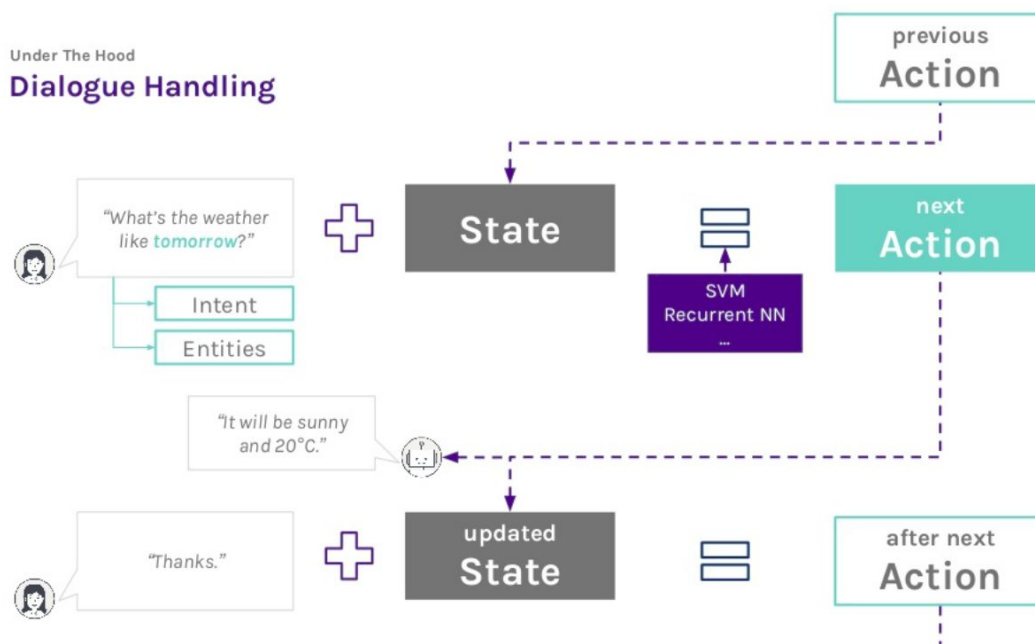


Fig. 2.1 An example of NLU in action, where a user asks a chatbot for weather information

For our project, we've used the Rasa NLU framework, which uses multiple helper libraries to carry out the various tasks that NLU comprises of, two most important ones being entity extraction and intent classification, which we now discuss in a little more detail, along with the processing pipeline of Rasa.

RASA, THE NLU FRAMEWORK

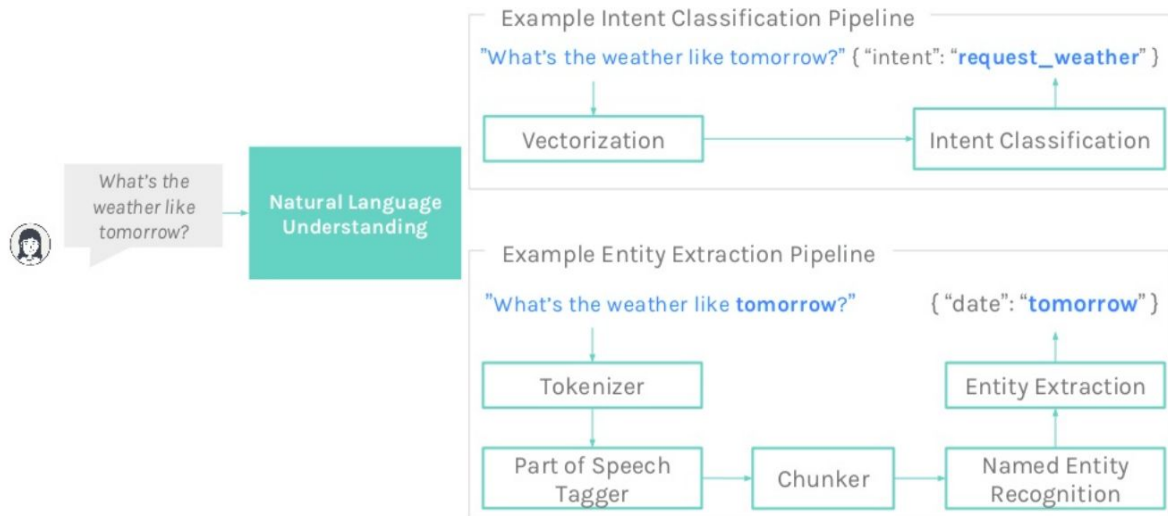


Fig. 3.1 A schematic diagram showing Rasa's NLU processing pipeline for a chat bot

WORKING OF RASA

Rasa NLU (Natural Language Understanding) is a tool for understanding what is being said in short pieces of text. Rasa NLU is primarily used to build chatbots and voice apps, where this is called intent classification and entity extraction. To use Rasa, *you have to provide some training data*. That is, a set of messages which you've already labelled with their intents and entities. Rasa then uses machine learning to pick up patterns and generalise to unseen sentences.

Rasa NLU is the natural language understanding module. It comprises loosely coupled modules combining a number of natural language processing and machine learning libraries in a consistent API. We aim for a balance between customisability and ease of use. To this end, there are pre-defined pipelines with sensible defaults which work well for most use cases. For example, the recommended pipeline, `spacy_sklearn`, processes text with the following components. First, the text is tokenised and parts of speech (POS) annotated using the spaCy NLP library. Then the spaCy featuriser looks up a GloVe vector for each token and pools these to create a representation of the whole sentence. Then the scikit-learn classifier trains an estimator for the dataset, by default a mutli

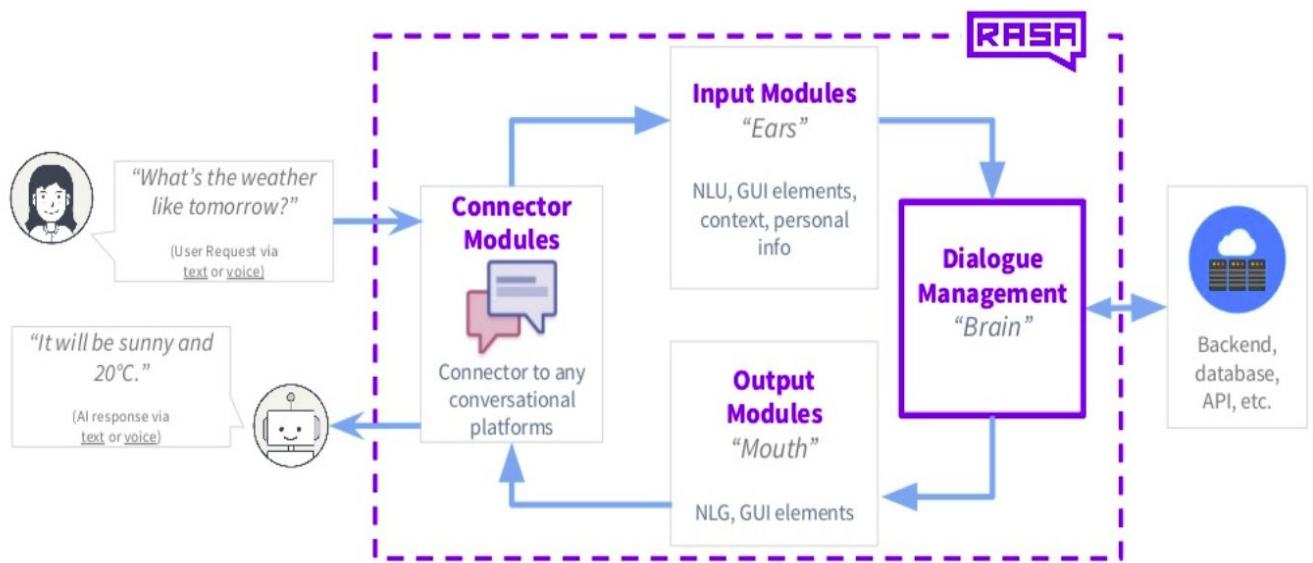


Fig. 3.2 How Rasa build conversational software with ML

class support vector classifier trained with five-fold cross-validation. The `ner_crf` component then trains a conditional random field to recognise the entities in the training data, using the tokens and POS tags as base features. Since each of these components implements the same API, it is easy to swap (say) the GloVe vectors for custom, domain-specific word embeddings, or to use a different machine learning library to train the classifier. There are further components for handling out-of-vocabulary words and 3 many customisation options for more advanced users. Three components: text, intent, and entities. The first two are strings while the last one is an array.

- The *text* is the search query; An example of what would be submitted for parsing.
- The *intent* is the intent that should be associated with the text.
- The *entities* are specific parts of the text which need to be identified.

Entities are specified with a start and end value, which together make a python style range to apply to the string. Entities can span multiple words,

and in fact the value field does not have to correspond exactly to the substring.

Rasa's processing pipeline

The process of incoming messages is split into different components. These components are executed one after another in a so called processing pipeline. There are components for entity extraction, for intent classification, pre-processing and there will be many more in the future. Each component processes the input and creates an output. The output can be used by any component that comes after this component in the pipeline. There are components which only produce information that is used by other components in the pipeline and there are other components that produce Output attributes which will be returned after the processing has finished.

For example, for the sentence "I am looking for Chinese food" the output:

```
{
  "text": "I am looking for Chinese food",
  "entities": [
    {"start": 8, "end": 15, "value": "chinese", "entity": "cuisine",
    "extractor": "ner_crf", "confidence": 0.864}
  ],
  "intent": {"confidence": 0.6485910906220309, "name":
  "restaurant_search"},
  "intent_ranking": [
    {"confidence": 0.6485910906220309, "name": "restaurant_search"},
    {"confidence": 0.1416153159565678, "name": "affirm"}
  ]
}
```

is created as a combination of the results of the different components in the pre-configured pipeline `spacy_sklearn`. For example, the `entities` attribute is created by the `ner_crf` component.

As discussed before, the two most important parts of NLU are entity extraction and intent classification, which we'll discuss here.

1. Entity extraction

For each entity extractor, the evaluation script logs its performance per entity type in your training data. So if you use `ner_crf` and `ner_duckling` in your pipeline, it will log two evaluation tables containing recall, precision, and f1 measure for each entity type. In the case of `ner_duckling` we actually run the evaluation for each defined duckling dimension. If you use the time and ordinal dimensions, you would get two evaluation tables: one for `ner_duckling (Time)` and one for `ner_duckling (Ordinal)`. `ner_synonyms` does not create an evaluation table, because it only changes the value of the found entities and does not find entity boundaries itself. Finally, keep in mind that entity types in your testing data have to match the output of the extraction components. This is particularly important for `ner_duckling`, because it is not fitted to your training data.

2. Intent classification

The evaluation script will log precision and f1 measure for each intent and once summarized for all. Furthermore, it creates a confusion matrix for you to see which intents are mistaken for which others.

USING RASA LOCALLY FROM PYTHON

We are using Rasa directly in our Python program. Rasa NLU currently supports both Python 2 and 3.

Training

For creating the models, we are training directly in Python with a script like the following (using spacy):

```
from rasa_nlu.training_data import load_data
from rasa_nlu.config import RasaNLUModelConfig
from rasa_nlu.model import Trainer
from rasa_nlu import config

training_data = load_data('data/examples/rasa/demo-rasa.json')
trainer = Trainer(config.load("sample_configs/config_spacy.yml"))
trainer.train(training_data)
model_directory = trainer.persist('./projects/default/') #
Returns the directory the model is stored in
```

Prediction

We can call Rasa NLU directly from your python script. To do so, we are loading the metadata of our model and instantiate an interpreter. The `metadata.json` in our model dir contains the necessary info to recover our model:

```
from rasa_nlu.model import Metadata, Interpreter

# where `model_directory` points to the folder the model is
```


persisted in

```
interpreter = Interpreter.load(model_directory)
```

We are then using the loaded interpreter to parse text:

```
interpreter.parse(u"The text I want to understand")
```

which returns the same **dict** as the HTTP api would (without emulation).

If multiple models are created, it is reasonable to share components between the different models. E.g. the **'nlp_spacy'** component, which is used by every pipeline that wants to have access to the spacy word vectors, can be cached to avoid storing the large word vectors more than once in main memory. To use the caching, a **ComponentBuilder** should be passed when loading and training models.

Here is a short example on how we are creating a component builder, that can be reused to train and run multiple models, to train a model:

```
from rasa_nlu.training_data import load_data
from rasa_nlu import config
from rasa_nlu.components import ComponentBuilder
from rasa_nlu.model import Trainer

builder = ComponentBuilder(use_cache=True)           # will cache
components between pipelines (where possible)

training_data = load_data('data/examples/rasa/demo-rasa.json')
trainer = Trainer(config.load("sample_configs/config_spacy.yml"),
```

```
builder)
trainer.train(training_data)
model_directory = trainer.persist('./projects/default/') #
Returns the directory the model is stored in
```

The same builder can be used to load a model (can be a totally different one). The builder only caches components that are safe to be shared between models. Here is a short example on how we are using the builder when loading models:

```
from rasa_nlu.model import Metadata, Interpreter
from rasa_nlu import config

# For simplicity we will load the same model twice, usually we
would want to use the metadata of
# different models

interpreter = Interpreter.load(model_directory, builder) # to
use the builder, pass it as an arg when loading the model
# the clone will share resources with the first model, as long as
the same builder is passed!
interpreter_clone = Interpreter.load(model_directory, builder)
```

ADVANTAGES OF RASA OVER OTHER NLU FRAMEWORKS

Runs Locally

- No network overhead
- Control QoS
- Can be deployed anywhere

Own Your Data

- Users keep ownership of data
- They don't need to hand data over to big tech companies
- Avoids vendor lock-in

Hackable

- Model can be tuned according to the use case

LANGUAGE FEATURES

Some of the important features of the language are listed below. Please note that not all of the following features are final, and are subject to change.

- **Execution:** A file having a `Main()` function will be treated as a source files, and files without a `Main()` will be treated as header files. Execution of source files will begin from their `Main()`.
- **Importing code:** Code from a source file can be imported into another the same way it's done in Python, using `import path/to/file` or `from path/to/file import method1, method2`. If the file has a `Main()` function, the `Main()` function will not be imported.
- **Code blocks:** Like in Python, indentation will define code blocks, and tabs and spaces will not be allowed to be mixed.
- **Data types:**
 - **Built-in:** `void`, `character`, `integer`, `real`, `complex`, `string`, and `pointers` to these types and to other pointers
 - **User-defined:** Will be implemented in a similar way as classes are in Python.

For example:

```
type Node
    data = Any
    left = pointer to Node
    right = pointer to Node
```

- **Containers:** `list`, `set`, `dictionary` and more.
- **Control statements:** It will have most of the control statements that are available in Python and Ruby, but with minimal and flexible syntax.

For example, all of the following will be valid versions of **for each** loop:

```
for each v in graph.vertices
```

or

```
for each vertex v in graph.vertices
```

or

```
for each vertex in graph.vertices as v
```

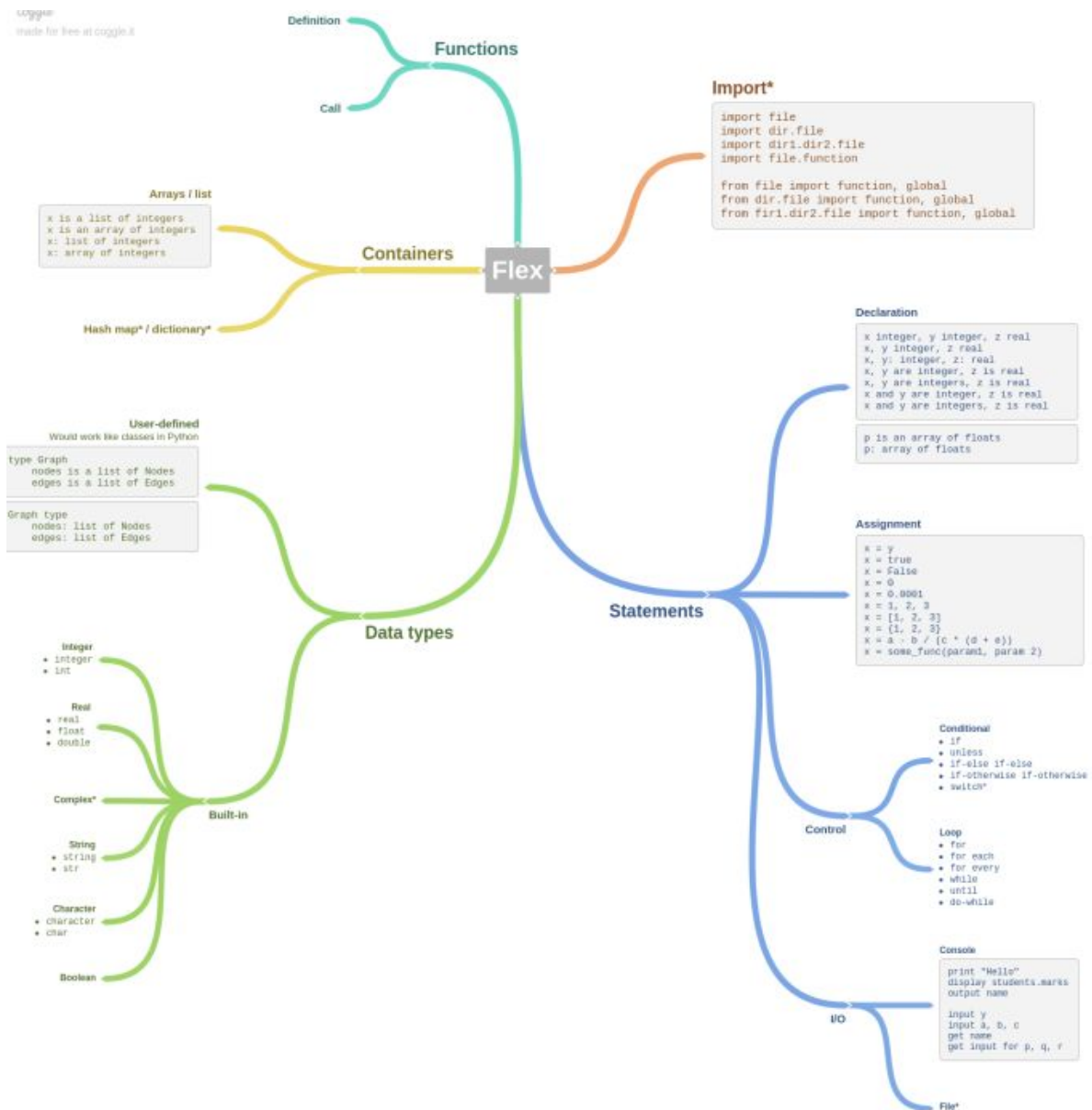


Fig. 4.1 Graphical breakdown of the language features of Flex

EXAMPLE CODE SNIPPETS

EXAMPLE 1

```
Dijkstra(graph, w, source)
  InitialiseSingleSource(graph, source)
  S = empty set
  Q = min priority queue of graph.vertices
  until Q is empty
    u = ExtractMin(Q)
    add u to S
    for each vertex v in graph.adjacency of u
      Relax(u, v, w)
```

EXAMPLE 2

```
Mean()
  arr is [1, 2, 5, 6, 6, 99]
  set total to 0
  set counter to 0
  for every element e in arr
    add e to total
    add 1 to counter
  mean = total / counter
  display mean
```

PROPOSED METHOD

1. Read a line
 - a. It could be composed of multiple statements separated by a semicolon, so split it at the semicolons to get individual statements. Add them all to the *statement queue*.
2. Process a statement
 - a. If the statement conforms to any one of the language's regular expressions, parse it and extract entities from it, storing them as a dictionary in the *parsed statement list*.
For example, the statement,

```
SomeFunction(arg1, arg2, arg3)
```

Will match the regular expression for the beginning of a function definition (since it's not indented, it cannot be a function call), and will be stored in the form:

```
{
  'intent': 'function_def',
  'entities': {
    'name': 'SomeFunction',
    'args': [
      {
        'name': 'arg1'
      },
      {
        'name': 'arg2'
      },
      {
        'name': 'arg3'
      }
    ]
  }
}
```

- b. Otherwise, pass it as a query to the Rasa server running locally.

- i. The server will parse the statement and will return a response in JSON format. As an example, the response returned after parsing the statement,

```
for each vertex v in graph.vertices
```

would look like:

```
{
  'intent': 'begin_loop',
  'entities' : {
    'loop_type': [{
      'value': 'for_each',
      'confidence': 0.993
    }],
    'loop_over': [{
      'value': 'graph.vertices',
      'confidence': 0.984
    }],
    'loop_as': [{
      'value': 'v',
      'confidence': 0.975
    }]
  }
}
```

- ii. If all entities have a confidence above a minimum threshold (say 90%), reduce the above response and store it in a dictionary in the *parsed statement list*, as in step 2.a. Otherwise, report an error at that line.

3. Parse each entry (line) from the *parsed statement list*, keeping track of the block and the scope of the variables, and convert it to the target language. If a required value is missing, report an error at the line `l+1`, where `l` is the index of the line in the *parsed statement list*.

STATEMENTS CURRENTLY SUPPORTED BY FLEX

1. Declaration

In the table below, following data types are supported: **Integer, int, real, float, double, complex, string, str, char, character, boolean, bool.**

Statement	Intent	Entity
real x	declare_var	<ul style="list-style-type: none">• name(x)• type(real)
x real	declare_var	<ul style="list-style-type: none">• name(x)• type(real)
x is real	declare_var	<ul style="list-style-type: none">• name(x)• type(real)
int x	declare_var	<ul style="list-style-type: none">• name(x)• type(int)
x integer	declare_var	<ul style="list-style-type: none">• name(x)• type(int)
x is int	declare_var	<ul style="list-style-type: none">• name(x)• type(int)
float x	declare_var	<ul style="list-style-type: none">• name(x)• type(float)
x float	declare_var	<ul style="list-style-type: none">• name(x)• type(float)
x is float	declare_var	<ul style="list-style-type: none">• name(x)• type(float)
double x	declare_var	<ul style="list-style-type: none">• name(x)• type(double)
x double	declare_var	<ul style="list-style-type: none">• name(x)• type(double)
x is double	declare_var	<ul style="list-style-type: none">• name(x)• type(double)
complex x	declare_var	<ul style="list-style-type: none">• name(x)• type(complex)

x complex	declare_var	<ul style="list-style-type: none"> • name(x) • type(complex)
x is complex	declare_var	<ul style="list-style-type: none"> • name(x) • type(complex)
char x	declare_var	<ul style="list-style-type: none"> • name(x) • type(char)
x character	declare_var	<ul style="list-style-type: none"> • name(x) • type(char)
x is char	declare_var	<ul style="list-style-type: none"> • name(x) • type(char)
str x	declare_var	<ul style="list-style-type: none"> • name(x) • type(string)
x str	declare_var	<ul style="list-style-type: none"> • name(x) • type(string)
x is string	declare_var	<ul style="list-style-type: none"> • name(x) • type(string)
bool x	declare_var	<ul style="list-style-type: none"> • name(x) • type(boolean)
x bool	declare_var	<ul style="list-style-type: none"> • name(x) • type(boolean)
x is boolean	declare_var	<ul style="list-style-type: none"> • name(x) • type(boolean)
Multi variable declaration		
x, y, z real	declare_multi_var	<ul style="list-style-type: none"> • name(x) • name(y) • name(z) • type(real)
real x, y, z	declare_multi_var	<ul style="list-style-type: none"> • name(x) • name(y) • name(z) • type(real)
x, y, z are real	declare_multi_var	<ul style="list-style-type: none"> • name(x) • name(y) • name(z) • type(real)
x, y, z integer	declare_multi_var	<ul style="list-style-type: none"> • name(x)

		<ul style="list-style-type: none"> • name(y) • name(z) • type(integer)
int x, y, z	declare_multi_var	<ul style="list-style-type: none"> • name(x) • name(y) • name(z) • type(int)
x, y, z are integers	declare_multi_var	<ul style="list-style-type: none"> • name(x) • name(y) • name(z) • type(integer)
Containers		
P: array of floats	declare_array	<ul style="list-style-type: none"> • name(P) • type(float)
P is a list of integers	declare_array	<ul style="list-style-type: none"> • name(P) • type(integer)
P is an array of integers	declare_array	<ul style="list-style-type: none"> • name(P) • type(integer)
x: list of integers	declare_array	<ul style="list-style-type: none"> • name(x) • type(integer)
x: array of integers	declare_array	<ul style="list-style-type: none"> • name(x) • type(integer)

2. Assignment

Statements	Intent	Entities
A is 12	assign	<ul style="list-style-type: none"> • name(A) • value(12)
A = 12	initialize_assign	<ul style="list-style-type: none"> • name(A) • value(12)
A = [1,2,3]	initialize_assign	<ul style="list-style-type: none"> • name(A) • value(1,2,3)
A = {1,2,3}	initialize_assign	<ul style="list-style-type: none"> • name(A) • value(1,2,3)
A = 12.05	initialize_assign	<ul style="list-style-type: none"> • name(A) • value(12.05)

A = "hello"	initialize_assign	<ul style="list-style-type: none"> • name(A) • value("hello")
A = B	initialize_assign	<ul style="list-style-type: none"> • name(A) • value(B)
Arr is [1,2,3]	initialize_assign	<ul style="list-style-type: none"> • name(A) • value(1,2,3)
x = y	initialize_assign	<ul style="list-style-type: none"> • name(x) • value(y)
x = true	initialize_assign	<ul style="list-style-type: none"> • name(x) • value(true)
x = false	initialize_assign	<ul style="list-style-type: none"> • name(x) • value(false)
x = 0	initialize_assign	<ul style="list-style-type: none"> • name(x) • value(0)
x = 0.0001	initialize_assign	<ul style="list-style-type: none"> • name(x) • value(0.0001)
x = 1,2,3	initialize_assign	<ul style="list-style-type: none"> • name(x) • value(1,2,3)
x = 1, 2, 3	initialize_assign	<ul style="list-style-type: none"> • name(x) • value(1, 2, 3)
x = {1, 2, 3}	initialize_assign	<ul style="list-style-type: none"> • name(x) • value(1, 2, 3)
x = [1, 2, 3]	initialize_assign	<ul style="list-style-type: none"> • name(x) • value(1, 2, 3)

3. Control

3.1. Conditional

Statement	Intent	Entity
If a>1	begin_if	condition(a>1)
Otherwise	begin_else	none
else	begin_else	none
Else if a==1	begin_else_if	condition(a==1)
Elif a==1	begin_else_if	condition(a==1)

Switch a	begin_switch	switch_var(a)
Case '1' =	begin_case	case_value('1')
Case 1 :	begin_case	case_value(1)
switch(a)	begin_switch	switch_var(a)
Case 1 ->	begin_case	case_value(1)
Otherwise if (a==1)	begin_else_if	condition(a==1)
Unless a<1	begin_unless	condition(a<1)
If a is greater than 1	begin_if	condition(a is greater than 1)
If a is equal to 1	begin_if	condition(a is equal to 1)
If a is not 1	begin_if	condition(a is not 1)
If a is lesser than 1	begin_if	condition(a is lesser than 1)

3.2. Loop

Statement	Intent	Entity
for every element e in container	begin_for_each	loop_over(container) loop_as(e)
for every e in c	begin_for_each	loop_over(c) loop_as(e)
while a is not 0	begin_while	condition(a is not 0)
while a != 0	begin_while	condition(a != 0)
until a is not 0	begin_until	condition(a is not 0)
for each item in items	begin_for_each	loop_over(items) loop_as(item)
for every item in items	begin_for_each	loop_over(items) loop_as(item)
unless a is 0	begin_unless	condition(a is 0)
do	begin_do	none

4. I/O

4.1. Console

4.1.1 Input

Statements	Intent	Entities
Input a	input	var_name(a)
Get value of a from user	input	var_name(a)
Input a, b	input	var_name(a) var_name(b)
Prompt user to enter a	input	var_name(a)
Get arr	input	var_name(arr)
Take arr as input	input	var_name(arr)
Get value of a, b from user	input	var_name(a) var_name(b)
Get a,b	input	var_name(a) var_name(b)
Take a, b as input	input	var_name(a) var_name(b)

4.1.2 Output

Statements	Intent	Entities
Print "hello"	print	to_print("hello")
print "some string of text"	print	to_print("some string of text")
Print hello	print	to_print(hello)
Print value of hello	print	to_print(hello)
Display total_marks	print	to_print(total_marks)
Print a	print	to_print(a)
Output student.marks	print	to_print(student.marks)
Display arr	print_elements	to_print(a)

Print all values of arr	print_elements	to_print(a)
Output arr	print_elements	to_print(arr)
Print all elements of arr	print_elements	to_print(arr)

5. Data types

5.1. Built-in

5.1.1. Integer

int, integer

5.1.2. Real

real, float, double

5.1.3. Complex

complex

5.1.4. String

string, str

5.1.5. Character

char, character

5.1.6. Boolean

bool, boolean

5.2. User-defined

Would work like classes in Python.

```
type Graph
    nodes is a list of Nodes
    edges is a list of Edges
```

```
Graph type
    nodes: list of Nodes
    edges: list of Edges
```


6. Containers

6.1. Array/list

Statement
x is a list of integers
x is an array of integers
x: list of integers
x: array of integers

7. Functions

Function signatures are parsed using regular expressions.

Statement
add(a int, b int) int
add(int a, int b) int
add(int a, int b) int

8. Comments

Comments are also parsed using regular expressions.

Statements
this is a comment
// this is a comment
/* multi line comment */

SNIPPETS OF CODE DICTIONARIES FOR VARIOUS TARGET LANGUAGES

1. Python

```
'end_block': '\n',

'default_code': '',

'begin_main': '''if __name__ == '__main__':\n''',

# Output
'print': {
    'entities': ['to_print'],
    'code': 'print({to_print})'
},
'print_elements': {
    'entities': ['to_print'],
    'code': '''
for element in {to_print}:
    print(element, end=', ')
print('\b\b ')
'''
},

# Input
'input': {
    'entities': ['var_name'],
    'code': 'input({var_name})'
},
```

2. C++

```
# Conditional
# if
'begin_if': {
    'entities': ['condition'],
    'code': 'if({condition}) {'
},
# else
'begin_else': {
    'entities': [],
```

```

        'code': 'else {{'
    },
    # else if
    'begin_else_if': {
        'entities': ['condition'],
        'code': 'else if ({condition}) {{'
    },
    # switch
    'begin_switch': {
        'entities': ['switch_var'],
        'code': 'switch ({switch_var}) {{'
    },
    # case
    'begin_case': {
        'entities': ['case_value'],
        'code': 'case {case_value} :'
    },
    # unless
    'begin_unless': {
        'entities': ['condition'],
        'code': 'if (!({condition})) {{'
    },

```

3. Java

```

# Loops
# for each
'begin_for_each': {
    'entities': ['loop_over', 'loop_as'],
    'code': 'for (Object {loop_as} : {loop_over}) {{'
},
# while
'begin_while': {
    'entities': ['condition'],
    'code': 'while ({condition}) {{'
},

# Declare variables
'declare_var': {
    'entities': ['name', 'type'],
    'code': '{type} {name} = new {type}();'
},
'declare_array': {
    'entities': ['name', 'type'],

```

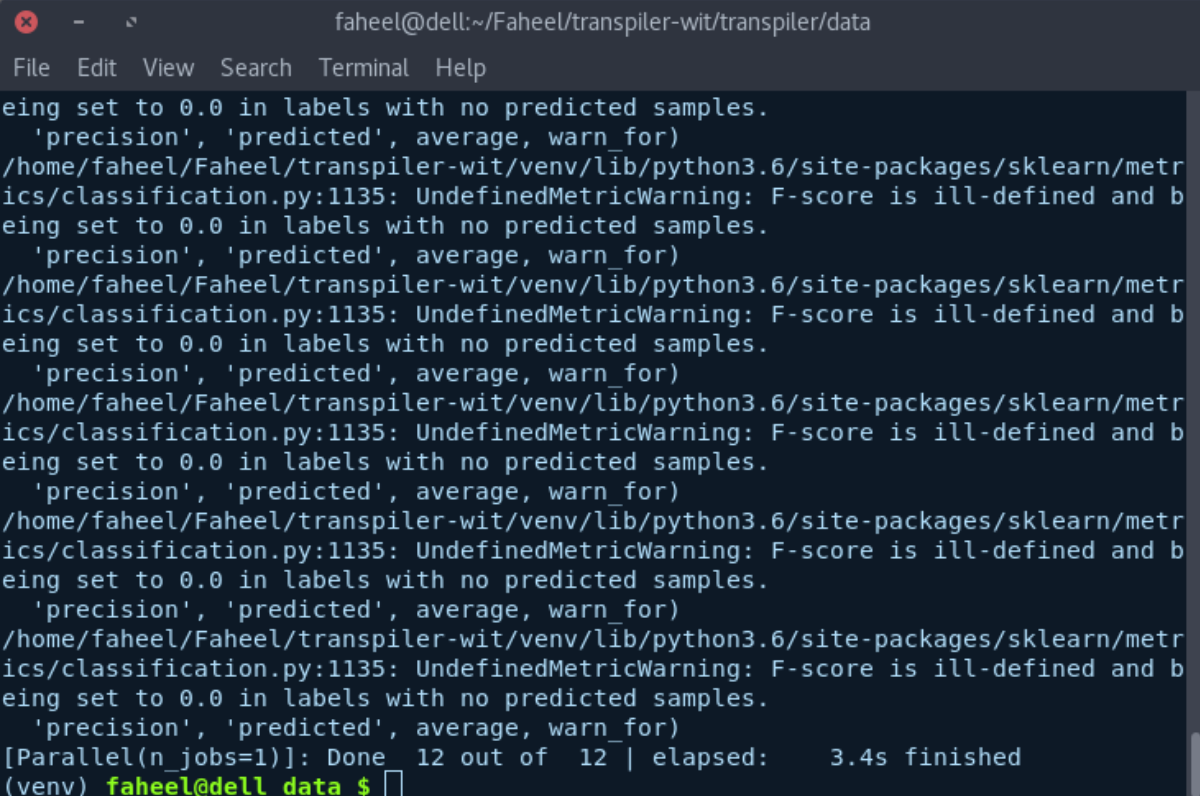
```
        'code': 'ArrayList<{type}> {name} = new ArrayList<{type}>();'
    },

    # Assignment / initialisation
    'initialize_assign': {
        'entities': ['name', 'value'],
        'code': '{name} = {value};'
    },
}
```

RESULT & OUTPUT

1. Training the model

```
python -m rasa_nlu.train --config model_config.yml --data intents
--path model
```



```
faheel@dell:~/Faheel/transpiler-wit/transpiler/data
File Edit View Search Terminal Help
eing set to 0.0 in labels with no predicted samples.
'precision', 'predicted', average, warn_for)
/home/faheel/Faheel/transpiler-wit/venv/lib/python3.6/site-packages/sklearn/metr
ics/classification.py:1135: UndefinedMetricWarning: F-score is ill-defined and b
eing set to 0.0 in labels with no predicted samples.
'precision', 'predicted', average, warn_for)
/home/faheel/Faheel/transpiler-wit/venv/lib/python3.6/site-packages/sklearn/metr
ics/classification.py:1135: UndefinedMetricWarning: F-score is ill-defined and b
eing set to 0.0 in labels with no predicted samples.
'precision', 'predicted', average, warn_for)
/home/faheel/Faheel/transpiler-wit/venv/lib/python3.6/site-packages/sklearn/metr
ics/classification.py:1135: UndefinedMetricWarning: F-score is ill-defined and b
eing set to 0.0 in labels with no predicted samples.
'precision', 'predicted', average, warn_for)
/home/faheel/Faheel/transpiler-wit/venv/lib/python3.6/site-packages/sklearn/metr
ics/classification.py:1135: UndefinedMetricWarning: F-score is ill-defined and b
eing set to 0.0 in labels with no predicted samples.
'precision', 'predicted', average, warn_for)
/home/faheel/Faheel/transpiler-wit/venv/lib/python3.6/site-packages/sklearn/metr
ics/classification.py:1135: UndefinedMetricWarning: F-score is ill-defined and b
eing set to 0.0 in labels with no predicted samples.
'precision', 'predicted', average, warn_for)
[Parallel(n_jobs=1)]: Done 12 out of 12 | elapsed: 3.4s finished
(venv) faheel@dell data $
```

Fig. 5.1 Training the model

2. Input source

a.

```
Main()
x is an integer
input x
value_list is an array of integers
value_list = [1, 5, 8, 6, 0, 4, 5]
print x
for every element in value_list
    element = x * element
    print element
```

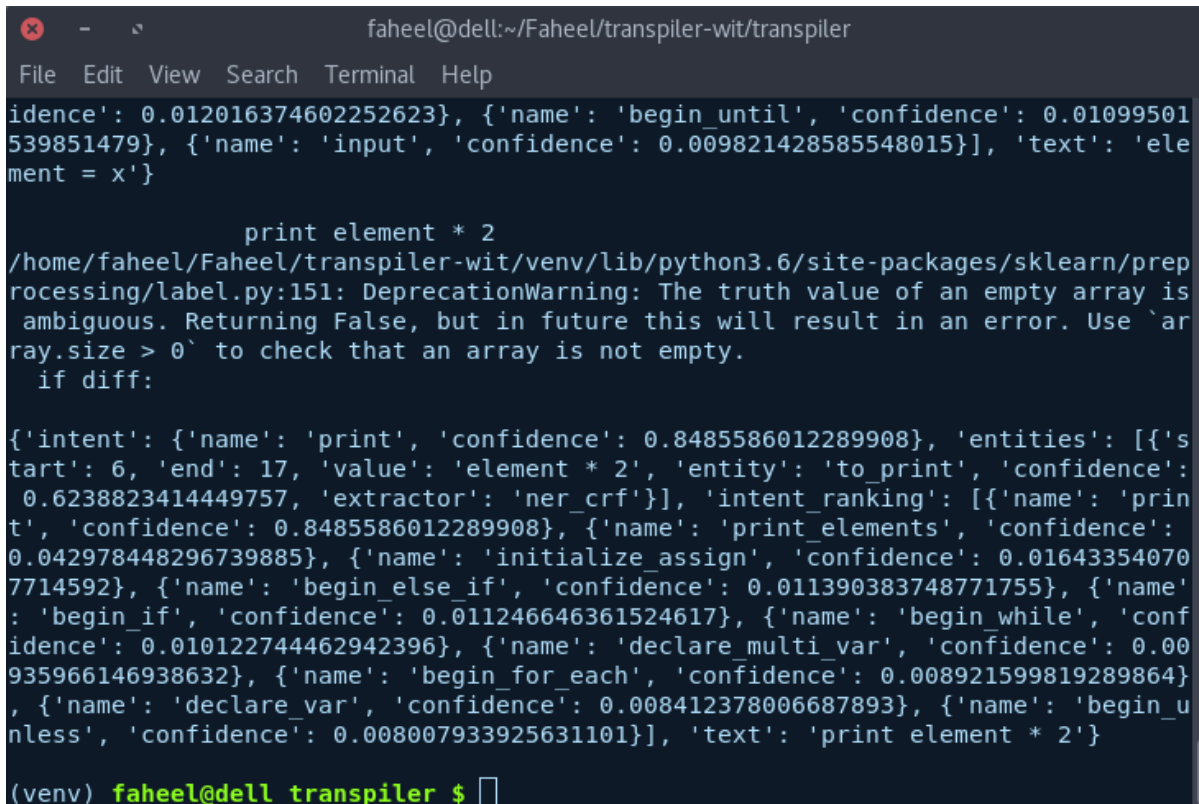
b.

```
Main()
  arr is an array of integers
  x is integer
  input 10 elements in arr
  input x
  flag: boolean
  flag is false
  for every element in arr
    if element == x
      display "FOUND!"
      flag is true
  if flag == false
    display "NOT FOUND!"
```

3. Transpile to targets

a. To Python

```
./flex_transpiler.py ../test/sample1.flex -l python -o
../test/sample1.py
```



```
faheel@dell:~/Faheel/transpiler-wit/transpiler
File Edit View Search Terminal Help
idence': 0.012016374602252623}, {'name': 'begin_until', 'confidence': 0.01099501
539851479}, {'name': 'input', 'confidence': 0.009821428585548015}], 'text': 'ele
ment = x'}

      print element * 2
/home/faheel/Faheel/transpiler-wit/venv/lib/python3.6/site-packages/sklearn/prep
rocessing/label.py:151: DeprecationWarning: The truth value of an empty array is
ambiguous. Returning False, but in future this will result in an error. Use `ar
ray.size > 0` to check that an array is not empty.
  if diff:

{'intent': {'name': 'print', 'confidence': 0.8485586012289908}, 'entities': [{ 's
tart': 6, 'end': 17, 'value': 'element * 2', 'entity': 'to_print', 'confidence':
0.6238823414449757, 'extractor': 'ner_crf'}], 'intent_ranking': [{ 'name': 'prin
t', 'confidence': 0.8485586012289908}, { 'name': 'print_elements', 'confidence':
0.042978448296739885}, { 'name': 'initialize_assign', 'confidence': 0.01643354070
7714592}, { 'name': 'begin_else_if', 'confidence': 0.011390383748771755}, { 'name'
: 'begin_if', 'confidence': 0.011246646361524617}, { 'name': 'begin_while', 'conf
idence': 0.010122744462942396}, { 'name': 'declare_multi_var', 'confidence': 0.00
935966146938632}, { 'name': 'begin_for_each', 'confidence': 0.008921599819289864}
, { 'name': 'declare_var', 'confidence': 0.008412378006687893}, { 'name': 'begin_u
nless', 'confidence': 0.008007933925631101}], 'text': 'print element * 2'}

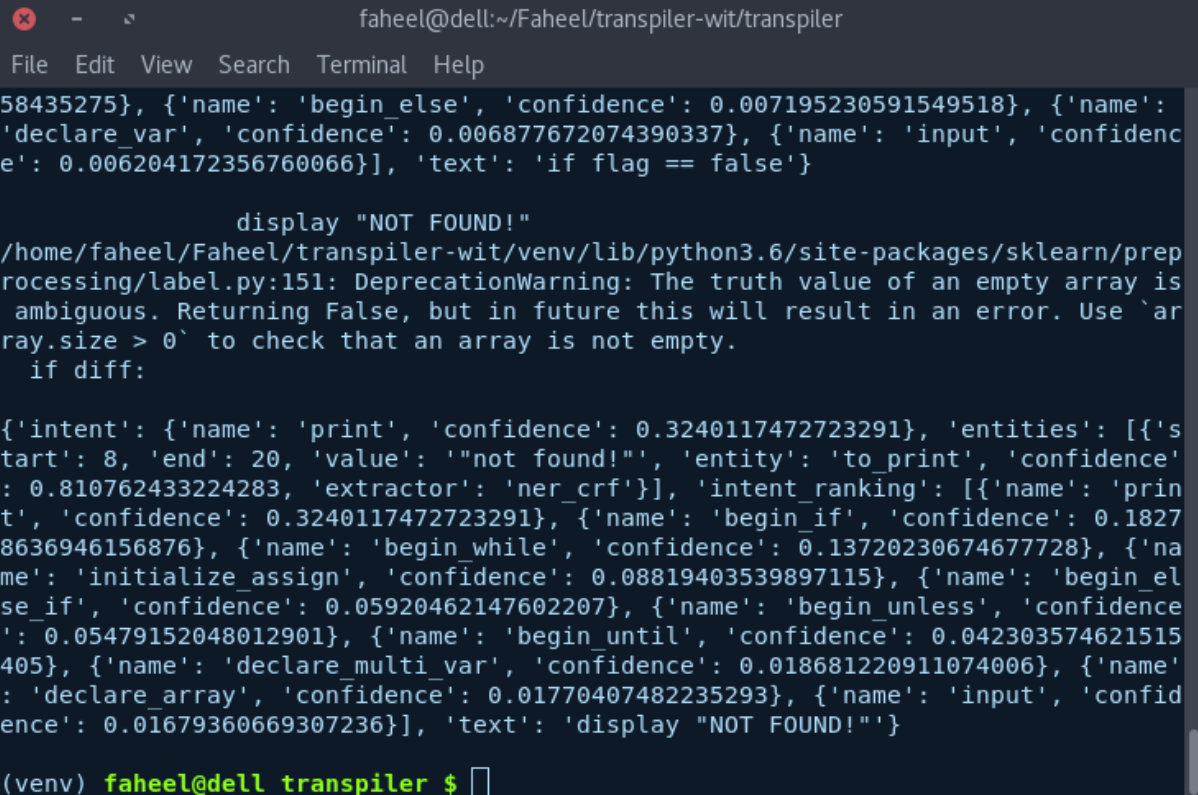
(venv) faheel@dell transpiler $
```

Fig. 5.2 Transpiling Flex code to Python

b. To C++ and Java

```
./flex_transpiler.py ../test/sample2.flex -l c++ -o  
../test/sample2.cpp
```

```
./flex_transpiler.py ../test/sample2.flex -l java -o  
../test/sample2.java
```



```
faheel@dell:~/Faheel/transpiler-wit/transpiler  
File Edit View Search Terminal Help  
58435275}, {'name': 'begin_else', 'confidence': 0.007195230591549518}, {'name':  
'declare_var', 'confidence': 0.006877672074390337}, {'name': 'input', 'confidenc  
e': 0.006204172356760066}], 'text': 'if flag == false'}  
  
display "NOT FOUND!"  
/home/faheel/Faheel/transpiler-wit/venv/lib/python3.6/site-packages/sklearn/prep  
rocessing/label.py:151: DeprecationWarning: The truth value of an empty array is  
ambiguous. Returning False, but in future this will result in an error. Use `ar  
ray.size > 0` to check that an array is not empty.  
if diff:  
  
{'intent': {'name': 'print', 'confidence': 0.3240117472723291}, 'entities': [{ 's  
tart': 8, 'end': 20, 'value': '"not found!"', 'entity': 'to_print', 'confidence'  
: 0.810762433224283, 'extractor': 'ner_crf'}], 'intent_ranking': [{ 'name': 'prin  
t', 'confidence': 0.3240117472723291}, { 'name': 'begin_if', 'confidence': 0.1827  
8636946156876}, { 'name': 'begin_while', 'confidence': 0.13720230674677728}, { 'na  
me': 'initialize_assign', 'confidence': 0.08819403539897115}, { 'name': 'begin_el  
se_if', 'confidence': 0.05920462147602207}, { 'name': 'begin_unless', 'confidence  
' : 0.05479152048012901}, { 'name': 'begin_until', 'confidence': 0.042303574621515  
405}, { 'name': 'declare_multi_var', 'confidence': 0.018681220911074006}, { 'name'  
: 'declare_array', 'confidence': 0.01770407482235293}, { 'name': 'input', 'confid  
ence': 0.01679360669307236}], 'text': 'display "NOT FOUND!"'}  
  
(venv) faheel@dell transpiler $
```

Fig. 5.3 Transpiling Flex code to CPP

4. Output sources

a.

```
if __name__ == '__main__':  
    input(x)  
    value_list = [1, 5, 8, 6, 0, 4, 5]  
    print(x)  
    for element in value_list:  
        element = x * element  
        print(element)
```

b. C++

```
#include <algorithm>
#include <array>
#include <cstdlib>
#include <functional>
#include <iostream>
#include <random>
#include <string>
#include <vector>

typedef int integer;
typedef float real;
typedef char character;
typedef bool boolean;

int main() {
    std::vector<integer> arr;
    integer x;
    for (size_t i = 0; i < 10; i++) {
        integer temp;
        std::cin >> temp;
        arr.push_back(temp);
    }
    std::cin >> x;
    boolean flag;
    flag = false;
    for (const auto& element : arr) {
        If (element == x) {
            std::cout << "found!";
            flag = true;
        }
    }
    if (flag == false) {
        std::cout << "not found!";
    }

    return 0;
}
```


Java

```
import java.awt.*;
import java.awt.geom.*;
import java.io.*;
import java.util.*;

public class test {
    public static void main() {
        ArrayList<integer> arr = new ArrayList<integer>();
        integer x = new integer();
        String arr = System.console().readLine();
        String x = System.console().readLine();
        boolean flag = new boolean();
        flag = false;
        for (Object element : arr) {
            if(element == x) {
                System.out.println("found!");
                flag = true;
            }
        }
        If (flag == false) {
            System.out.println("not found!");
        }
    }
}
```

CONCLUSION

As demonstrated through this project, we can create a programming language with flexible syntax that also has the capability of understanding natural language. Though the implementation is quite basic at the moment, it can be improved easily as discussed in the following section. So far we have implemented features such as variable declaration, assignment, control statements like if-else blocks and loops, I/O, and functions. The intent classification of almost all of these features has a high confidence score more than 95%, some reaching as high as 99.9% even with a handful of training samples (2-4) per sentence structure type, per statement category.

By all counts and with proven results, it is no wonder that Flex can handle all different types of natural language statements & convert them to corresponding mentioned programming language.

FUTURE PROSPECTIVE

- Flex currently implements basic functionality as shown in the output section. Other features such as import statements, Object Oriented Programming concepts, HashMaps, File Handling are into future consideration.
- Flex currently supports three languages: Python, Java, C++. We are planning to add as many language as possible. The modular architecture allows easy addition of new languages.
- Flex is an open source project currently hosted on GitHub. Users can review the code, open issues regarding bugs & errors, and submit patches for new features or bug fixes.
- Also, the language can be made more flexible by adding more training examples for each statement category

PROGRAMMING ENVIRONMENT & TOOLS USED

Wit.ai

For training an early version of the NLU model that worked by sending each line of text from the input source to a web server which then returned a response over the internet.

Rasa NLU

For training the NLU model on our systems.

Python 3

For programming the transpiler, including code dictionaries for C++, Java and Python.

Git and GitHub

For version control and code sharing among all team members, and code reviews on changes made.

REFERENCES

- Source code - <https://github.com/Flex-lang/transpiler>
- Rasa NLU
 - Homepage - <https://rasa.com/>
 - Documentation - <https://nlu.rasa.com/index.html>
 - Source - https://github.com/RasaHQ/rasa_nlu
- Wit.ai
 - Homepage - <https://wit.ai/>
 - Documentation - <https://wit.ai/docs>
- Intent classification using spAcy (part of the Rasa's processing pipeline) - <https://spacy.io/>
- Named entity extraction using Duckling (part of the Rasa's processing pipeline) - <https://duckling.wit.ai/>