Gourav Suri	14BCS-0040
Sanjay Kumar	14BCS-0046
Syed Faheel Ahmad	14BCS-0067

Major Project Synopsis

TITLE

Flex: A programming language with minimal syntax that can understand natural language

ABSTRACT

Our goal is to create a compiler for a programming language, that we call *Flex*, which has a minimal syntax, but more importantly it can 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.

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
```

Example code snippets

```
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)
```

PROPOSED METHOD

During parsing, if there is a statement that does not adhere to our language's grammar, it will be parsed by the NL model trained using Rasa, which will help in interpreting the statement by extracting entities and classifying the intent, using which the intermediate form is constructed.

Then the overall intermediate form will be converted to machine code the traditional way, using LLVM's backend.

PROGRAMMING ENVIRONMENT & TOOLS USED

- C++ to program the compiler
- LLVM as the compiler framework
- Rasa for interpreting natural language

REFERENCES

- LLVM
  - Homepage - http://llvm.org/
  - Documentation - http://llvm.org/docs/index.html
  - Architecture - http://www.aosabook.org/en/llvm.html
  - Source - https://github.com/llvm-mirror
- Rasa
  - Homepage - https://rasa.com/
  - Documentation - https://core.rasa.ai/index.html
  - Source - https://github.com/RasaHQ