Parsel Tongue: Milestone 1

Bharambe Soham Amit
Roll Number: 210264
Department of Computer Science
Indian Institute of Technology Kanpur
sohamb21@iitk.ac.in

Chetan
Roll Number: 210281
Department of Computer Science
Indian Institute of Technology Kanpur
chetan21@iitk.ac.in

Divyansh Chhabria
Roll Number: 210356
Department of Computer Science
Indian Institute of Technology Kanpur
divyanshc21@iitk.ac.in

Introduction & Motivation

In the magical world of Harry Potter, snakes possess the ability to communicate using ParselTongue, a language characterized by its hissing sounds. This skill is considered rare and is typically passed down through generations, known as a hereditary trait. Harry Potter, the famous wizard, had the unique ability to understand and speak ParselTongue.

But only very few of us have this ability to understand a serpent's language. Do we all need to learn ParselTongue to talk to serpents? The answer to this question cannot be a yes. While some of us could learn ParselTongue, others could be helped by these ParselTongue learners, in translating their message. But who would make the first move to learn ParselTongue?

Here we come to help you, and provide you with the magical powers to talk to serpents, atleast Python, if not all. On gaining this magical power, you would feel that you are speaking your everyday language, and on your behalf, we would translate your daily language to the ParselTongue, the language of serpents.

The analogy becomes clearer if you consider your daily language to be Python and ParselTongue, the languages of serpents, to be the x86-64 assembly code. It is much easier for us to program in a high level language like Python, but it would be tremendous amount of work to program directly in assembly. Hence our objective and motivation for this project is to create a compiler to translate specifically Python 3.12 to x86-64 assembly code.

Required Packages

Parsel Tongue is developed by utilizing the following packages:

- `flex` : `sudo apt-get install flex`
- `bison` : `sudo apt-get install bison`
- `graphviz` : `sudo apt-get install graphviz`

Executing Scripts

The source files for milestone 1 are all inside the directory `milestone1/src`. A bash wrapper `ParselTongue.sh` is provided for compilation and execution. This wrapper uses the Makefile for compilation and also refactors command line arguments for the main compiler.

The options provided are as follows:

- `-h` or `-help` or `--help` : Shows a manual for usage of compiler
- `-i` or `-input` or `--input` : One can specify multiple input Python files for compiler
- `-o` or `-output` or `--output` : One can specify multiple output PDF files for compiler
- `-v` or `-verbose` or `--verbose` : Passing this option will generate a Parse Tree at the output file instead of classic AST

Execution Examples

Inside `milestone1/src` :

- `./ParselTongue.sh ../tests/test1.py`
- `./ParselTongue.sh test1.py`
- `./ParselTongue.sh -i ../tests/test1.py`
- `./ParselTongue.sh -i ../tests/test1.py -o ../output/test1.pdf`
- `./ParselTongue.sh -v -i ../tests/test1.py -o ../output/test1.pdf`
- `./ParselTongue.sh ../tests/test1.py ../tests/test2.py -o ../output/test1.pdf`
- `./ParselTongue.sh ../tests/test1.py -v -i ../tests/test2.py -o ../output/test1.pdf`

Parsel Tongue will make new output file with the help of input files if there are not enough output files.

All the files specified before -o are by default taken as input files.

If an input file is not found in the `specified-path`, then the script looks for `../tests/specified-path` and takes the input file from there if it exists.

File extensions are not checked in the script so one can even pass file not ending in `.py` and still get the desired output.

From PEP Grammar To Bison Grammar

The PEP Grammar when converted to Bison, did not give a lot of conflicts and they were simple to solve after removing the productions which were not required for Parsel Tongue. We also used the Grammophone Github which helped visualize the conflicts in the form of Parsing Table.

From Bison Grammar To Parse Tree

To extensively use Grammophone tool we had initially designed a grammar modifier, which was a lexer taking as input the bison `parser.y` file, which helped refactor the grammar. A similar idea was again used to make the parse tree from the grammar. We designed an `action_adder.l` lexer which then added proper semantic actions for all the productions. This eased the making of parse tree.

From Parse Tree To Abstract Syntax Tree

Here we intensely cleaned the parse tree to get as close as possible to our ideal AST. This involved traversing the parse tree multiple times in a top-down depth-first traversal manner, to selectively remove non-terminals that we didn't desire to see in the AST, and to bring the operators one level up in the tree, which makes the AST intuitive to visualize the actual framework of the input program.

Generating DOT File

A Breadth First Traversal on the AST was implemented to make the DOT file. Different Tokens were colored differently based on their category, to make the AST more easier to look upon.

Testing

We tried to create 15 testcases, making our best effort to cover all possible productions that were included in the grammar. At each stage, we tested the front-end of our upcoming compiler with these testcases.