ParselTongue: Milestone 2

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

ParselTongue 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 2 are all inside the directory `milestone2/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` : Shows a manual for usage of compiler
- `-i` or `--input` : One can specify multiple input Python $(.py)$ files for compiler
- `-o` or `--output` : One can specify multiple output TAC $(.txt)$ files for compiler
- `-v` or `--verbose` : This will return a .debug file which contains the infomation of parsing such as stack and tokens.
- `-p` or `--ptree` : One can specify this option to have a parse tree pdf file at the output folder
- `-a` or `--ast` : One can specify this option to have a abstract syntax tree pdf file at the output folder
- `-d` or `--dot` : This option is supposed to be used with $-a$ or $-p$. It helps retain the dot file which was used to create the pdfs.

- `-c` or `--csv` : This option need not be passed for this milestone to get the csv dump of symbol tables but will be needed for future milestone. Passing this currently wont change any behavior.
- `-m` or `--markdown` : Passing this option will output a markdown file for the symbol table dump which can be used along with an extension for better visualization of the dump. The default csv dump will also be given in this case.

Execution Examples

Inside `milestone2/src` :

- `./ParselTongue.sh {input_file_1}.py {input_file_2}.py`
- `./ParselTongue.sh -i {input_file}.py`
- `./ParselTongue.sh -i {input_file}.py -o {output_file}.txt`
- `./ParselTongue.sh -v -i {input_file}.py -o {output_file}.txt`
- `./ParselTongue.sh {input_file_1}.py {input_file_2} -o {output_file}.txt`
- `./ParselTongue.sh -a -m {input_file_1}.py {input_file_2}.py`
- `./ParselTongue.sh -a -p -v {input_file_1}.py {input_file_2}.py`
- `./ParselTongue.sh {input_file_1}.py -vampi {input_file_2}.py -o {output_file}.txt`

Options can be passed together if needed as in above, -vampi means verbose, ast, markdown, ptree, input options.

ParselTongue 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.

Input files passed should have *.py* as file extension, while the output files should likewise have $.txt$ as extension.

From Abstract Syntax Tree To Symbol Tables

We did a Depth First Search on our AST which was made in the previous milestone to generate the symbol tables. During this process, we streamlined our grammar by simplifying and removing several rules. As a result, in this milestone, we can now focus on the positive aspect of the generating the symbol tables, as errors are automatically generated if unsupported features are requested, making the overall system more efficient and user-friendly.

Error Statements

For the above two processes we also created a good amount of customized errors for every case we could think of, now when something is wrong we can raise an error and maybe print a note which helps user better understand the error given. Error is raised when:

1. There is a use of undeclared variable
2. There is redefinition of a variable
3. A class attribute is used without defining it
4. The type provided during declaration is not a valid type. (A class can be a valid type.)
5. The left and right of an operator does not have compatible types
6. The iterated object is not iterable, it is not a list
7. The type was not specified for a parameter in a function definition
8. One tries to use keyword as a variable name
9. Class declaration is not done globally
10. An overloaded function creates ambiguity
11. The return type does not match the promised return type during function definition
12. A return type was not specified for a function
13. The lvalue for an assignment cannot be assigned a value
14. An object attribute is defined in some function other than `__init__`
15. `__init__` function is defined somewhere other than inside the class

16. The parent for a class is not a valid class
17. There are two or more `self` in function formal parameters
18. there is a mixed list, i.e., a list with with more than one different datatype
19. Nested lists are defined. (Only 1D lists are allowed)
20. An empty list is passed to a variable as they are not supported
21. A function which is defined inside a class does not have self as a parameter
22. The break and continue statements are used outside of a loop
23. Return statement is used when not in function
24. A variable which is not a list is dereferenced
25. The list access or conditions in loop have an unsupported type of value in them

Detailed error messages are provided, including error lines with row, column, and file name information. Notes are also used to indicate previous declarations or provide variable types, making it easier for end users to understand and infer from the errors.

From Parse Tree To Three Address Code

To generate the three address code, we implemented a Depth First Search (DFS) over the parse tree. This approach reduces the dependency on the correctness of the formed three address code compared to the correctness of the Abstract Syntax Tree (AST). The code for the overloaded range and len functions has been added to the three address code, integrating these functionalities into the overall implementation.

Generating CSV/Markdown File

A Breadth First Traversal on the Symbol Table Structure was implemented to make the dump file. Markdown File is generated for better visualization which vastly eased our nerves as compared to the CSV file.

Support for additional features

1. Dynamic `range` and `len` functions: The list data structure is modified to internally store the list's length in the first 8 bytes. This design choice ensures that the length is always readily available during list creation and access, streamlining operations that rely on this information.
2. Function Overloading: In ParselTongue, we've implemented function overloading with appropriate checks, allowing multiple functions with the same name but different parameters to be defined.

Testing

We conducted elaborate testing on over 80+ unique test cases (excluding testcases used for milestone 1) to ensure that our implementation in ParselTongue is correct. These test cases have been categorized based on whether they should run successfully or an error has to be raised. We plan to reuse these test cases for milestone 3.