ParselTongue: Milestone 3

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. Therefore we have created 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 3 are all inside the directory `milestone3/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. Make sure to run this script only from `milestone3/src` directory.

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 $(.s)$ 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.
- `-t` or `--tac` : Passing this option will output 3AC code file.

Execution Examples

Inside `milestone3/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}.s`
- `./ParselTongue.sh -v -i {input_file}.py -o {output_file}.s`
- `./ParselTongue.sh {input_file_1}.py {input_file_2} -o {output_file}.s`
- `./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 -vampit {input_file_2}.py -o {output_file}.s`

Options can be passed together if needed as in above, -vampit means verbose, ast, markdown, ptree, input, threeAC code 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 *.s* as extension.

Required Language Features Supported

1. Primitive data types (int, str, and bool)
   (a) Float data type was implemented till 3AC phase (milestone 2)
   (b) Implementation of float was shelved for x86 creation.
2. Supported 1D lists of primitive data types as well as list of class objects.
3. Basic operators:
   (a) Arithmetic operators: +, -, *, /, //, %, **
   (b) Relational operators: ==, !=, >, <, >=, <=
   (c) Logical operators: and, or, not
   (d) Bitwise operators: &, |, ^, ∼, <<, >>
   (e) Assignment operators: =, +=, -=, *=, /=, //=, % =, ** =, & =, | =, ^ =, <<=, >>=
4. Control flow via `if-elif-else`, `for`, `while`, `break` and `continue`.
   (a) Supported iterating over ranges specified using the range() function.
5. Support for recursion in methods.
6. Support for library function `print()`.
7. Support for classes and objects, including multilevel inheritance and constructors.
8. Supported Methods and method calls.
9. Supported string comparison between strings of variable length.
10. Type casting
    (a) int ⟶ bool.
    (b) bool ⟶ int.
    (c) int ⟶ float.
    (d) float ⟶ int.

Optional Language Features Supported

1. Method Overloading.
   (a) Multiple methods with the same name but different parameters are allowed.
   (b) Implemented name mangling for this feature.
2. Dynamic `range` and `len` functions are supported, variables that are evaluated at runtime can be passed as their arguments.
3. Iterating over 1D lists is supported.
4. Print is supported for primitive data types as well as their 1D lists.
5. Implemented run time error for list index out of bounds.
6. Implemented run time error for negative exponents which may otherwise result in float values.

Register Allocation

The implementation does not spill on every other instruction but uses a strategy similar to bottom up allocation based on variable usage i.e. we spill the variable whose use is the farthest in the future. The registers `RAX`, `RBP`, `RSP`, and `RBX` have been used as special registers, while the rest are allocated and spilled across basic blocks. In a function call, arguments numbered from 1-6 are filled in the registers `RDI`, `RSI`, `RDX`, `RCX`, `R8`, and `R9` and remaining arguments (if any) are pushed onto the stack.

Type Checking

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.

Test Cases

Inside `milestone3/src:`
To get `x86` file as well as the executable for `test1`
Run: `./ParselTongue.sh -i ../milestone3/demo/test1.py -o ../output/test1.s`

After the executable is made, to run it from this directory, give command:
`./../output/test1`

You can also *cd* into the output folder where x86 assembly and executable were created previously and run command:
`./test1`

We conducted elaborate testing on over 60+ unique test cases (excluding testcases used for milestone 1) to ensure that our implementation in ParselTongue is correct.

Note

1. Bool Handling
   Bools in ParselTongue are set to 1 and 0 for $True$ and $False$ respectively. When there is an evaluation over booleans, python converts the variable to integer. But in our implementation bools will be type-casted again after the expression evaluation to a bool.
2. Divisions
   When division operation is performed in python, the variables are converted to float. But in our implementation, the left hand side is assigned the quotient. If the left hand side is a bool then the quotient is type-casted to bool and then assigned.
3. Floats
   Floats are neglected in x86 implementation, so one won't receive any error upon usage of floats and the behavior will be undefined.
4. `if __name__ == "__main__": block`
   Don't define or declare any variable in this block. One can still call multiple functions inside this block but not declare a variable as the behavior is undefined.
5. Modifications To 3AC from milestone 2
   Introduced special commands to implement string comparison, string copy.

Contributions

1. All the team members sat together and discussed while writing the code.
2. Soham Bharambe, 210264: 33%
3. Chetan, 210281: 33%
4. Divyansh Chhabria, 210356: 33%