Google Summer of Code ‘22

LPython: Implementing a Very Fast Parser

4th April 2022

Akshansh Bhatt

About Me

The Student

Name: Akshansh Bhatt

E-Mail: qaz.akshansh@gmail.com

GitHub Profile: akshanshbhatt

Timezone: IST (UTC+05:30)

The Institute

University: Birla Institute of Technology and Science (BITS), Pilani

Major: Physics, Electronics and Instrumentation Eng. (Dual Major)

Duration of Degree: Five Years

Current Year: Third

Degree: Master of Science (M.Sc.) and Bachelor of Engineering (B.E.)

Programming Background

I use macOS as my operating system (Macbook Air with Apple Silicon M1), Visual Studio Code as my primary editor and debugger, and iTerm2 with zsh as my primary terminal emulator. Although for considerably large projects, I

often switch to Neovim for speed. I am familiar with the git and Github workflow.

I have contributed to numerous big projects in the past couple of years, not to mention that most have been in the Open Source domain. Last year, I was part of SymPy's GSoC developer team. I am currently the 43rd highest contributor, to SymPy's codebase, of all time and also an active member of the SymPy dev team, helping out new contributors make their first contribution.

You can look at my final report from last year over here. I also encourage you to check the weekly progress blog from last year, in which I have provided every small detail regarding my advancement in the project I chose.
Python is my favorite language due to its simple syntax, usage in many domains, and a vast community of awesome developers. It's simple yet so powerful. Apart from Python, I also code in C/C++ and JavaScript (but Python's the OG).

My Contributions

These are some of my contributions to LPython-

Merged Pull Requests

- **#245**: Add numpy to virtual env package list
- **#353**: Create environment.yml file
- **#363**: Fix the dedent token generation in new tokenizer
- **#400**: [Parser] Implement parsing rules for functions

Issues Closed

- **#350**: Setup environment.yml for creating a development Conda environment
- **#358**: New tokenizer fails when there are multiple indentations

Issues Raised

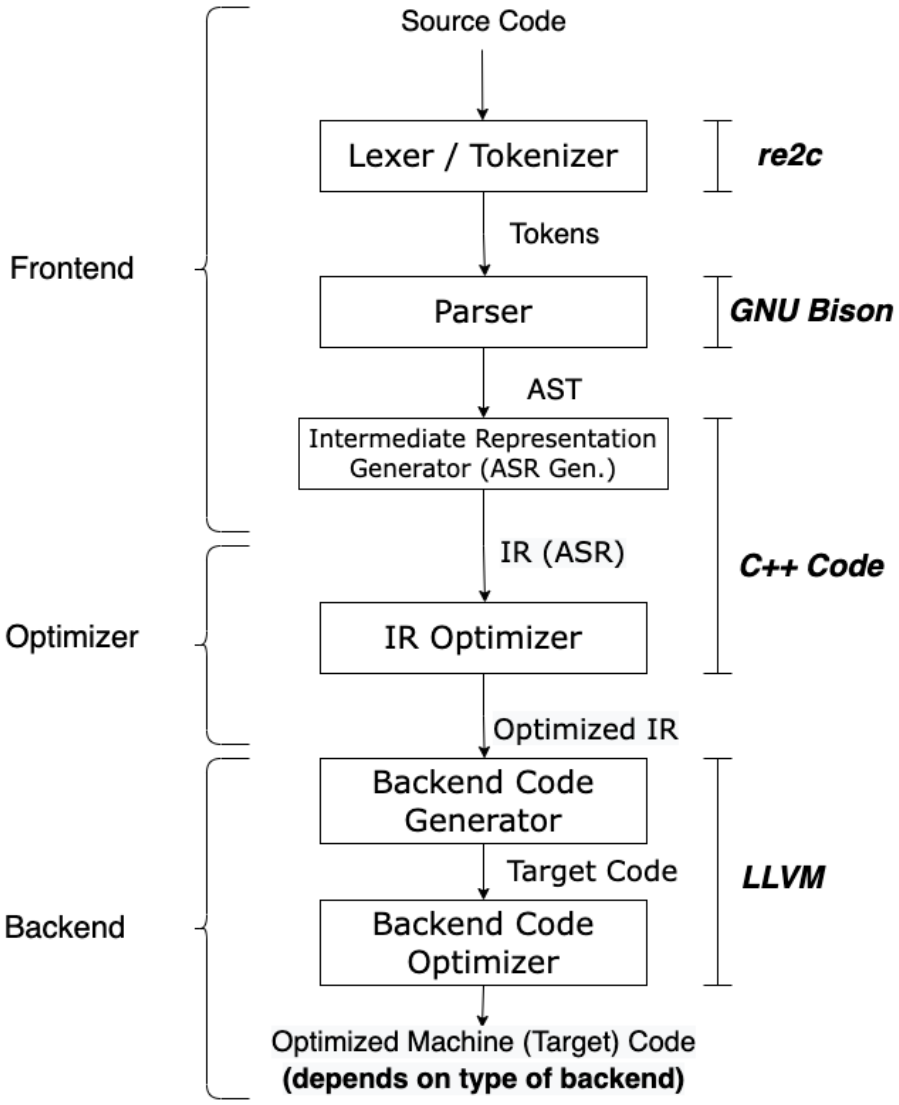
- **#243:** Issue with creating autogenerated files
- **#244:** Integration tests fail
- **#266:** Anomalous behavior of static types
- **#358**: New tokenizer fails when there are multiple indentations
- **#392:** The new parser generates unwanted AST nodes

Note: *There must be more PRs by the time my mentors will review my application, so I am providing the* ***GitHub link*** *to all my PRs.*

The Project

In this section, I will explain my project details. I expect this structure to be considerably improved under the guidance of my mentors.

Understanding the LPython Compiler



The diagram above depicts the proposed implementation of the LPython compiler. LPython compiler works in three different phases, which we can also reduce to two significant phases if we don't require code optimizations - the **Frontend** and the **Backend**.

The frontend is responsible for optimizing the high-level code into a tree structure before it is fed to the backend to generate machine-level code by traversing the tree. We can regenerate the source code from any intermediate step in the frontend pipeline. The frontend mainly deals with the syntactic and semantic analysis of the source code. We will discuss a more detailed analysis of the frontend and its components in the next section.

On the other hand, the backend's job is to traverse the nodes of the (presumably optimized) IR code (Abstract Semantic Representation in LPython's case) to the Target code (often machine level). The frontend deals with syntax, whereas the backend mainly handles low-level code generation and hardware-specific optimizations. LPython's backend can generate a wide variety of target code depending upon the need, thanks to the power of the LLVM. If the source code is generated till the LLVM IR level, LLVM can convert it to x86/ARM assembly or C++ equivalent very efficiently.

Frontend

Since my project mainly involves improving the frontend, I will discuss the essential components of the frontend in detail.

Tokenizer

Tokenization is the first step in the compilation process. The tokenizer converts the stream of characters of the source file to a stream of tokens.

Tokens are the most fundamental unit of the programming language. A token can be an identifier, separator, keyword, operator, literal, delimiter, or comment. The tokenizer further identifies the token type and passes tokens one at a time to the next stage of the compiler, that is, the parser. The errors caught in this part of the compilation process are due to incomplete syntax in the source file. For example, if a user forgets to open a parenthesis and forgets to close it later or starts a string with a single quote but closes it with a double quote. The tokenizer has no idea about the context of the code, so if someone mistakenly made some typo in a keyword, it will not be logged as an error to the user during this phase of compilation.

At present, everything up to AST generation is abstracted by the CPython's AST module. This includes Tokenization, Parsing, and finally, generation of AST. It is convenient but very slow. The key reason for such a slow frontend is writing and reading operations on a file (files are stored in the disk and not in the immediate memory, requiring more machine cycles to access). Not to mention that CPython's interpreter is already quite slow. Fast compilation speed is one of the key aims of LPython; therefore, we cannot compromise on it.

For this very reason, Ondrej decided to implement our tokenizer and parser from scratch. This would allow us to control every aspect of the compilation process and not compromise on its swiftness. Another advantage of this approach is that we can also expand the frontend to include features specific to LPython in the future.

The new tokenizer is not very mature at this point. It has been directly taken from the LFortran's codebase with some modifications. Re2c has been used to find patterns in the source code (using regular expressions) and convert them to its equivalent C++ code. This C++ code finally generates the tokens.

#260, #310, #312, #313, #314, #341 and most importantly #298 have relevant discussions/code changes related to the new tokenizer. I have formulated these points by taking information from these PRs, issues, Zulip chat discussions, and the Python documentation page. Here is my proposed plan to improve the tokenizer -

1. As tokenization is crucial in the compilation process, testing it thoroughly is an essential task. Ondrej mentioned in **this comment** that initially, our tokenizer does not have to be perfect. However, as we progress further, it is necessary to lessen the difference in the tokens generated by ours and the CPython's tokenizer. We can start testing smaller files first and then move to large files of renowned codebases and check for the difference in the tokens generated by both the tokenizers. Even if the tokens generated are not exactly the same, the difference should be such that later stages of compilation are not affected.
2. The new tokenizer does not log syntax errors when it should. For instance, I opened a square bracket but forgot to close it later in the program (example below). It is the tokenizer's job to throw a syntax error here. The CPython tokenizer caught this error, but the new re2c tokenizer didn't. The compiler should log errors where it is supposed to. Catching and logging errors in the later stage of compilation can cause issues. Testing these errors is also part of my proposed plan.

```
❯ cat examples/test_1.py #Sample Code
def main():
    a = [1, 2, 3, 4, 5
    print(a)
❯ python -m tokenize examples/test_1.py #CPython logs
```

```
errors
examples/test_1.py:4:0: error: EOF in multi-line
statement
❯ src/bin/lpython --show-tokens examples/test_1.py #No
error raised
(KEYWORD "def") 0:2
(TOKEN "identifier" main) 4:7
(TOKEN "(") 8:8
(TOKEN ")") 9:9
(TOKEN ":") 10:10
(NEWLINE) 11:11
(TOKEN "indent") 12:15
(TOKEN "identifier" a) 16:16
(TOKEN "=") 18:18
(TOKEN "[") 20:20
(TOKEN "integer" 1) 21:21
(TOKEN ",") 22:22
(TOKEN "integer" 2) 24:24
(TOKEN ",") 25:25
(TOKEN "integer" 3) 27:27
(TOKEN ",") 28:28
(TOKEN "integer" 4) 30:30
(TOKEN ",") 31:31
(TOKEN "integer" 5) 33:33
(NEWLINE) 34:34
(TOKEN "identifier" print) 39:43
(TOKEN "(") 44:44
(TOKEN "identifier" a) 45:45
(TOKEN ")") 46:46
(EOF) 47:47
```

3. Another major task is to ensure that there is no compromise in the tokenization speed by benchmarking it on large text files. Re2c has been

known for its insanely fast run time, so there should be a major speed boost compared to the prior tokenization method, but will it be faster than the regular CPython tokenizer? We can know the quantitative results only when we benchmark them.

Parser

After successful tokenization of the source code, the stream of tokens is passed to the next compilation step, parsing, carried out by a parser. In compiler architecture, a parser is a program that takes a stream of tokens as input and builds a data structure. In the case of LPython, this data structure is an AST (Abstract Syntax Tree). As discussed earlier, everything up to the AST generation is abstracted by the CPython AST module, which we plan to change by incorporating our new parser into the codebase.

CPython and LPython have different approaches to the parsing process and have different parser types. Generated using Bison, our parser will be an LR parser (employing LALR(1) parser tables). These parsers fall under a different class of parser (bottom-up parsers) compared to the current CPython parser, which is a PEG parser (these parsers are classified as Top-Down recursive-descent parsers, which is an entirely different class of parsers). PEG-based parsers work using backtracking algorithms. They try rules of parsing and backtrack when they don't match. Due to this reason, they are generally slower than the LALR-based parsers. Though PEG-based parsers might be slow, they are more robust than alternatives.

Before Python 3.9, CPython's parser was a LL(1)- based parser (also a Top-Down class of parser). I will not go into the details behind this transition, but you can read **this** if you are interested in learning more. All this information might seem too much, but it is essential to get a glimpse of all the

different approaches before finalizing to implement our own. For instance, we are using Bison for our parser program generation, but CPython's core development team stays skeptical of its usage and criticizes it (in the same PEP). If our current approach goes wrong or is way too complex to keep track of, a backup plan could be to use the current CPython Parser. **Pegen** is the official PEG parser generator used in CPython. It is maintained by the members of CPython's steering team. We can build CPython's parser directly using the peg_generator tool in the official CPython repository, which uses pegen to generate the parser. We just have to run a simple build command to make it work. Maybe, in the future, we can add both of these parsers and give a choice to the user for the parser he wants to use for compilation as a CLI argument.

LPython's parser is non-functional at this moment. PR #337 added the initial parser files to the codebase, but there has been no significant progress from this point onwards. Here is my proposed plan for implementing the Bison-based parser -

1. **Reviewing and modifying the grammar rules:** The grammar rules specified in the LPython's parser are not complete yet. I have been taking the CPython's syntax grammar as the reference for the grammar rules, and my first goal will be to match the grammar rules and apply them to the LPython's parser.

2. **Testing and Documenting the Parser:** The final AST produced by the parser should be equivalent to the one produced by CPythons's AST module. We can automate the testing for AST in the test suite. Proper testing for syntax errors during the parsing phase is also essential. Most people find parsers very perplexing, making them hesitant to contribute to them. I will be adequately documenting the source files of the parser

generator so that even beginners can easily contribute to it. Apart from the regular documentation, I will also maintain a wiki page for essential parser-related information that I'll learn from contributing.

3. **Logging Proper Error Messages during Parsing:** The objective of an error handler in a parser is to recover a syntax error. It plays a pivotal role in modern resilient parsers (like that of CPython), especially to be able to produce a valid AST even with syntax errors and provide proper and meaningful error messages. Bison has a very rich error handling support, and error messages can be specified with the grammar rules in the yy file. With the power of Bison, we can log meaningful error messages which will be at par, if not better, than the CPython parser.

4. **Benchmarking the Parser:** There have been some initial efforts to benchmark the parser. PR #359 aims to implement a parser benchmark for the new parser based on the description provided in #344. There can be better approaches to benchmarking the parser than this, but we can see this as a good initial step.

5. **Generate AST nodes directly:** As pointed out by Ondrej in this comment, we can explore the documentation of Bison to generate the actions of the parser automatically. It will generate AST nodes for most of the cases, and we can do it manually for the left out cases. It has been implemented in LFortran for macros and seems to work fine.

Timeline

Application Review Phase

- Since most of the code of my project will be written in C++, the first thing I plan to do during this phase is to brush up on my C/C++ skills. I plan to read about the best practices used in the industry so that my code is not vulnerable. Naive C/C++ code is infamous for memory leaks, and I don't want any stability issues later in the codebase.
- Open more PRs, report any issues I find, and continue contributing to the project in general. The flow should not break.
- Read articles/blogs related to the development of the CPython's Compiler, especially the frontend. This will help me get used to the design principles in general.

Community Bonding Period

- Set up my blog for posting weekly progress.
- Video Conference with my mentors and discuss any new idea that we should incorporate. I will also take suggestions on the implementation plan.
- Complete any pending PR from the previous phase.
- Open an Issue with a list of tasks to implement before the first evaluation phase deadline.

Coding Phase

Week 1-2

- Add new tests for the tokenizer module. These tests will include comparison of tokens generated by our re2c based tokenizer and the tokenizer module in Python. Check for the points of conflict between both the tokenizers and work on making the new tokenizer generate tokens identical to those of Python's.
- Add benchmark tests for the tokenizer. This will show the time taken by the new tokenizer as compared to Python's to the users.

Week 3-4

- Work on the meaningful error message generation in the tokenizer module. Making sure that tokenization error is shown under expected circumstances by the tokenizer.
- Start the initial work for the parser. Review the definitions and the grammar rules described in the parser file.

Week 5-6

- Work on the testing and documentation of the new parser.
- Write the report for the first evaluation phase and also complete any pending PRs / progress blogs.

Week 7-8

- Work on the error handling part of the parser. Make sure meaningful error messages are logged covering all the different circumstances.
- Add tests related to these error messages.

Week 9-10

- Work on benchmark suite for the new parser. This benchmark suite should have options to select multiple parsers to compare the results (eg. the old LL(1) parser and the new PEG-based parser).
- Work on automatic generation of the AST nodes.

Week 11-12

- Complete the pending PRs.
- Write the final report

Post-GSoC Plans

After GSoC, I will continue to contribute to LPython in any form I can. LPython's codebase will surely continue to grow once it achieves some of the fundamental functionalities of CPython. The conversion of code to a wide range of backends along with faster compile time will be game-changing and, in my opinion, would be the unique selling point of this project in the future. I will also help beginners get started with contributions and might become a mentor if LPython gets selected for GSoC next year.

References

1. Write a re2c+Bison parser for Python · Issue #298 · lcompilers/lpython (github.com)
2. Add an initial Bison parser by certik · Pull Request #337 · lcompilers/lpython (github.com)
3. Add initial tokenizer for Python by certik · Pull Request #310 · lcompilers/lpython (github.com)
4. Remove all the Fortran related items from tokenizer by Thirumalai-Shaktivel · Pull Request #314 · lcompilers/lpython (github.com)
5. Recognize Indent and Dedent as a Token by Thirumalai-Shaktivel · Pull Request #313 · lcompilers/lpython (github.com)
6. Do not recognize docstrings in the tokenizer by certik · Pull Request #312 · lcompilers/lpython (github.com)
7. Add a parser benchmark · Issue #344 · lcompilers/lpython (github.com)
8. Remove the Fortran parser completely by certik · Pull Request #260 · lcompilers/lpython (github.com)
9. https://github.com/python/cpython/blob/755be9b1505af591b9f2ee424a6525b6c2b65ce9/Grammar/Tokens
10. https://docs.python.org/3/reference/compound_stmts.html#grammar-token-suite
11. https://github.com/python/cpython/blob/755be9b1505af591b9f2ee424a6525b6c2b65ce9/Grammar/python.gram
12. cpython/token.h at main · python/cpython (github.com)
13. 2. Lexical analysis — Python 3.10.4 documentation
14. Guide to CPython's Parser - Python Developer's Guide

15. A. Skrobov - How CPython parser works, and how to make it work better - YouTube
16. LFortran Design - LFortran Documentation