Language Design and Implementation

Lab 01: A Simple Math Expression Lexer

V. Markos
Mediterranean College

Spring, 2024 – 2025, Week 01

Abstract

This lab series aims to build a simple Python interpreter on top of Python, much in the spirit of PyPy, just not utilising JIT. In contrast to PyPy, our interpreter will not be a fast and competitive one, but one that can help us explore the core concepts behind crafting tree–walking interpreters. In this first lab we will explore some key concepts around lexing and craft a simple maths scanner class that we will use and modify in subsequent labs.

1 Getting Started

You can find some useful resources in the `./source` directory for this lab. Namely:

- `ppython.py`, which serves as the main source of execution for our project. You can run it using two options:

  ```
  python[3] ppython.py
  ```

  which gets you in an interactive mode, and:

  ```
  python[3] ppython.py [script.py]
  ```

  which interprets a ppython source file.

- `Token`, which implements the `Token` class and any required support classes.

- `Scanner`, which implements the actual scanner (lexer) for this lab.

None of the above files contains a full implementation of the requested functionalities for this lab, however they all provide some (minor or major) hints towards that. The rest is up to you to fill–in and improvise. You can always feel free to ignore those files and start from scratch on your own.

As a side note valid for all labs in this series, we will be using type hints wherever we can. While this changes virtually nothing in terms of runtime speed

or compilation / interpretation for Python, it will help us and our IDE (VSCode, most probably) avoid common pitfalls of dynamically typed languages, such as mixing around types in variables. In case you choose a dynamically typed language for your project implementation, go for type hints, if possible, since, as the project grows bigger, type errors get really annoying and time–consuming at the debugging stage.

2 Tokenisation

The first step in scanning / lexing is to properly tokenise the provided source code. To do so, we have to, as we formally say, split the source code into *lexemes*, i.e., the quanta of meaning in a source file. As you might already suspects, all lexemes are not born equal. There are some characters that have a very specific meaning in a language, e.g., parentheses, brackets, reserved keywords, while some others can have any meaning from a particular list of choices, e.g., variable names, numerical constants, etc. One way to distinguish between all those is to use different token types for tokens that fall in some of the above categories. With that in mind, here comes your first exercise:

Exercise 2.1. Think of what token types you might need for your language. You can brainstorm, look around the web, open up the source code of PyPy to draw some inspiration or just write some Python code to see what sort of entities appear in a piece of source code. Having done so, edit `Token.py` (or your own corresponding source file) and add any token types you see fit in the `TokenType` enumeration class. Some classical token types have been already written for you:

```
class TokenType(Enum):
    # single character tokens
    LEFT_PAREN = 0
    RIGHT_PAREN = 1
    # add more here

    # < 3 character tokens
    BANG = 10 # you might want to change this value
    EQUAL = 11 # this too
    BANG_EQUAL = 12 # this too
    # add more here

    # EOF
    EOF = 100 # you might want to change this too
```

Hint: If all this feels overwhelming, focus on math–related tokens, as this is what you have to implement for today. You can postpone worrying about more complex expressions for the next lab.

Given a few token types, we now have to actually parse them. So, here comes your next exercise:

Exercise 2.2. Implement the `Scanner.scan_tokens()` method, as found in `Scanner.py` with the aim of properly scanning for simple mathematical expressions — we

will extend this functionality in subsequent labs. This function should get a piece of ppython source code and return a list of the corresponding tokens or raise appropriate exceptions — which you will have to handle in our next exercise. Some things you might want to take into account for your implementation might include:

- You better implement a few helper functions to help you recognise operators, digits, floats, integers, parenthesised expressions and anything else you see fit.
- Reserve parentheses and their balancing for last, since this might be a bit tricky.
- You will need to keep track of where you are standing in the provided source, in terms of line and column (i.e., character within that line), so, feel free to add them as class attributes or local variables in your functions. You will also need this piece of information for error logging, next.
- You might want to utilise regular expressions to facilitate parsing, however, note that in subsequent labs you will most probably need to drop this approach, as it makes several computations and pattern matching quite heavy in the case of more complex expressions. So, to save yourselves from some future work, avoid depending that much on regex.

The provided `Scanner` class is quite minimal, waiting for you to actually implement everything there:

```
# Scanner.py

from Token import Token

class Scanner:
    def __init__(self, source: str) -> None:
        self.source: str = source

    def scan_tokens(self) -> list[Token]:
        return []
```

As a last exercise for this lab, you now have to implement a simple error handling code, that you will improve in subsequent labs. Since we are working towards a parser, the only errors we can catch are syntax ones. Postponing error handling for the future would be a choice, however this would make things way more complex for us when we would have to implement it. So, let us start from this simple math lexer and introduce some error logging:

Exercise 2.3. In `ppython.py` you can find, among others, the following two functions:

```
def error(line: int, message: str) -> None:
    report(line, message)

def report(line: int, message: str) -> None:
    print(f"{line} | Error: {message}")
```

Those are intended to handle error logging and report then on the various places one might need to report an error, e.g., stderr, logfiles, etc. Evidently, they are quite naive in their implementation, awaiting from your to improve them as needed. After doing so, utilise them in order to properly throw errors on terminal. Bear in mind that your error handling should ensure that the user is properly informed about the root cause of the error, while, when in interactive mode, you should not exit the interactive mode loop, just report the error and wait for the next instruction.