Joonas Pelttari

What is lexical analysis and how is it related to other parts in compilation?

Lexical analysis is the first step in compilation where we take the source code as plain text and map everything to tokens that the language has. This is done mostly with regex and using a maximum length approach where the lexer tries to match the longest possible regex. For example, a word “int” will be matched to keyword *int* even thou it could also be an identifier *in* and identifier *t*.

If no match is found for some word, it is a syntax error. The main purpose of lexical analysis is to allow the next compilation steps to be able to handle tokens and not plain text. And also to check that the stuff we wrote is “grammarly” correct.

How is the lexical structure of the language expressed in the PLY tool? I.e., what parts are needed in the code and how are they related to lexical rules of the language?

In the code we have to tell the lexer all token types that exist, e.g. LCURLY, STRING. Then we have to define the regex for every token that to match it and sometimes a function to allow more adjusting like if integer is too large. Or in my code I checked for reserved words inside identifier function. For better error messages we also need to keep count of the newlines to be able to pinpoint the line where we have a syntax error.

Explain how the following are recognized and handled in your code:

a. Keywords are saved in a dict and handled inside the *t_IDENT* function. Basically the lexer finds a word like *unless* and then inside the function it checks the dict and realizes that, actually, this is a keyword (reserved word) and returns that token with type: UNLESS, value: unless
b. Comments start with (% and end with %) and these are checked in regex. And also to make sure line number counting is correct, the code checks how many newlines there are inside the comment and increments lexer’s line number by that amount. This is because the comment is allowed be in many lines.
c. The ignores whitespaces with t_ignore = “ “
d. These are all handled with simple regex like r'\+'
e. Regex finds a possible date and then the function checks that the date is actually valid with datetime.date() function. i.e. the month is not 13 or more etc.
f. Regex finds something that starts and ends with char “ and then there can be anything inside except char “. Also before returning we remove the quotes so the value is hello not “hello”
g. Regex checks for something that starts with two A-Z and then continues with at least 1 of A-Z0-9_. This probably would not need to be a function in my code because it has regex only, but it is.
h. I implemented the thousand separator so that the int_literal starts with optional minus, then it has 1-3 digits and then it can have many times a combination of one thousand separator + 3 digits. The thousand separators are removed from the token value and the number is checked to not be too large.

How can the lexer distinguish between the following lexical elements:

a. Regex checks that procedure/function names start with uppercase character, identifiers start with lowercase character
b. Regex checks that procedure starts with two uppercase characters and function starts with one uppercase character and does not have more of them ever.
c. These both trigger the t_IDENT function and inside the function the code checks a dictionary of keywords and realizes that the identifier is actually a keyword.
d. Date must have minus signs between digits but can't start with minus sign. Integer literal can start with minus sign but can't have them between digits.
e. Comments are inside (% and %). Everything inside those is comment, not code.

What did you think of this assignment? What was difficult? What was easy? Did you learn anything useful?

I think the assignment was very fun. The ply.lex is super simple and works like magic almost, so I can just focus on the tokens and regexes. I created a testing script (test.py file in the repository) that automatically runs all the example/test files provided by the course and compares the output to the expected output, and received a huge help from that because this lexical analysis seemed to be a textbook example of fixing something = breaking something. Manually testing again and again when changing some regex would probably have been very tedious.

I had some difficulties in the thousand separator and also in the IDENT when trying to make it work with keywords but the final solution using the dictionary for keywords worked well. Also it did take some time before I realized that I can just count the newlines inside a comment to make so that multiline comments don't break the lexer line number. Most small problems were basically just because I haven't written regex myself much.

One letter tokens were easy. Func and Proc were easy (because I did them last after learning/struggling with identifiers). Date was surprisingly easy. The actual program was pretty easy to setup with the use of the simple example to get going. I did the arguments handling myself without libraries but I hope it works.

I did learn to write my own regex stuff and again the importance of automatic testing. Without the ability to run and check all the course example files automatically I probably would have a lot more bugs and fixing small problems would have been much harder. I almost never managed to fix a problem without breaking something else the first try.