PLY Tutorial

By

Prof. Preet Kanwal
Associate Professor
Department of CSE,
PES University, Bengaluru

Python Lex-Yacc(PLY) Tutorial

This document explains how to construct a compiler using lex and yacc.
Let's first start by installing PLY.

1. Installation

PLY requires the use of Python 2.6 or greater. However, you should use the latest Python release if possible. It should work on just about any platform.

On **Linux**, run this first (not required for other OS)

```
apt-get install python3-setuptools
```

Next, use this link to download the .tar.gz file, unzip it, and run this command in the file path

```
C:\...\ply-x.y> python setup.py install
```

NOTE: You may need administrator privileges to run the command.
python - for Windows and Mac
python3 for Linux-based OS

!! PLY is no longer maintained as a pip-installable package. pip installs a broken package !!.

2. What is lex and yacc?

Lex and yacc are popular compiler construction tools used to generate lexical analyzers and parsers. PLY is a pure-Python implementation of lex and yacc.

What does that mean?

[1] Given a program, how does your compiler try to understand what it says? After all, we know the machine only understands 0s and 1s.To understand the code in programs with structured input, two tasks that occur over and over are

- **dividing the input** into meaningful units
- discovering the **relationship** among the units

Eg: For a C program, the units are variable names, constants, strings, operators, punctuation, and so forth.

This division into units (which are usually called **tokens**) is known as lexical analysis, or lexing for short. lex helps you by taking a set of descriptions of possible tokens and producing a routine, which we call a lexical analyzer/lexer/scanner, that can identify those tokens.

For example, a C program may contain something like:

```
      {
            int int; //initialising a variable named "int" of type int
            int = 33;
            printf("int: %d\n",int);
      }
```

(Example 2.1)

In this case, the lexical analyser will have broken the input stream into a series of "tokens" like this:

- {
- int
- int
- ;
- int

- =
- 33
- ;
- printf
- (
- "int: %d\n"
- ,
- int
-)
- ;
- }

The token descriptions that lex uses are known as **regular expressions**. Lex turns these regular expressions into a form that the lexer can use to scan the input text extremely fast, independent of the number of expressions that it is trying to match.

OK, but what do we do with these tokens?

As the input is divided into tokens, a program often needs to establish the **relationship** among the tokens. To run a program, a compiler needs to find the expressions, statements, declarations, blocks, and procedures in the program. This task is known as **parsing,** and the list of rules that define the relationships that the program understands is called **grammar**.

yacc(yet another compiler compiler) takes a concise description of a grammar and produces a routine that can parse that grammar, a **parser**. The yacc parser automatically detects whenever a sequence of input tokens matches one of the rules in the grammar and also detects a syntax error whenever its input doesn't match any of the rules.

Note that in example 2.1, the lexical analyser has already determined that where the keyword `int` appears within quotes, it is really just part of a literal string. It is up to the parser to decide if the token `int` is being used as a keyword or variable. Or it may choose to reject the use of the name `int` as a variable name. The parser also ensures that each statement ends with a `;` and that the brackets balance.

To summarise, when a task involves **dividing the input** into units and establishing some **relationship** among those units, you should think of lex and yacc!

3. PLY Overview

PLY consists of two separate modules; `lex.py` and `yacc.py`, both of which are found in a Python package called ply. The `lex.py` module is used to break input text into a collection of tokens specified by a collection of regular expression rules. `yacc.py` is used to recognize language syntax that has been specified in the form of a context-free grammar.

The two tools are meant to work together. Specifically, `lex.py` provides an external interface in the form of a `token()` function that returns the next valid token on the input stream. `yacc.py` calls this repeatedly to retrieve tokens and invoke grammar rules.

4. The structure of a PLY program

4.1 A familiar language

L = {$a^n b^n$, n ≥ 1}

This grammar checks if the string is of the form ab, aabb, aaabbb, etc.
Strings that would be rejected by the language: aab, aaa, bb, etc.

Here is the grammar for the language:

```
S → aSb | ab
```

Before we jump into the code to understand our input, we have to define our tokens and write regular expressions to recognize them.

```
tokens = ('A', 'B')

t_A = r'a'
t_B = r'b'
```

4.2 The lexer module

Create a file named as `lexer.py`

```
# Importing lex
import ply.lex as lex

# List of token names. This is always required
tokens = ('A', 'B')

# Regular expression rules for simple tokens
t_A = r'a'
t_B = r'b'

# Error handling rule
def t_error(t):
    print(f"Illegal character '{t.value[0]}'")
    t.lexer.skip(1)

# Building the lexer
lexer = lex.lex()
```

Regular expression rules and regular expression functions should always be prefixed with `t_`

Now, we can write a function to recognize the grammar of our language:

```
def p_sequence(p):
    '''S : A S B
         | A B '''
```

4.3 The parser module

Create a file named as `parser.py`

```
# Importing lex
import ply.yacc as yacc

# Get the token map from the lexer.
from lexer import tokens

def p_sequence(p):
    '''S : A S B
         | A B '''

# Error rule for syntax errors
def p_error(p):
    print("Syntax error")
    exit(1)

# Build the parser
parser = yacc.yacc()
```

Grammar functions should always be prefixed with a `p_`
Single-line grammar rules are to be enclosed in `'` on both sides
Multiline grammar rules are to be enclosed in `'''` on both sides

4.4 The main module

Create a file named `main.py`

```
from lexer import lexer
from parser import parser

while True:
      data = input("Enter a^n b^n sequence (e.g., 'aaabbb'): ")
      lexer.input(data)
      parsed = parser.parse(data)
      if parsed == None:
            print("Accepted")
```

Save all 3 files in the same directory!

On Windows/Mac, run the file as

```
python main.py
```

On Linux-based systems, use

```
python3 main.py
```

5. How to declare tokens?

Suppose you wrote a simple expression: `y = a + 2 * b`

When this is passed through ply.py, the following tokens are generated

```
'y','=', 'a', '+', '2', '*', 'b'
```

These generated tokens are usually used with token names, which are **always** required.

```
# The token list of the above tokens will be
tokens = ('ID','EQUAL','ID', 'PLUS', 'NUMBER', 'TIMES','ID' )

#The regular expression rules for the above example
 t_PLUS    = r'\+'
 t_MINUS   = r'-'
 t_TIMES   = r'\*'
 t_DIVIDE  = r'/'
```

More specifically, these can be represented as tuples of token type and token

```
('ID', 'y'), ('EQUALS', '='), ('ID', 'a'), ('PLUS', '+'),
('NUMBER', '2'), ('TIMES', '*'), ('ID', 'b')
```

This module provides an external interface in the form of token(), which returns the valid tokens from the input.

6. Examples of Commonly Used Grammar

6.1 Declaring Variables in C

Here's a simple example highlighting how we declare variables in C.

Before we write a grammar to validate our inputs, let's list out our tokens and their corresponding regular expressions.

The regular expression for the token 'ID' is not so simple. It also has to make sure to check if the variable name is a reserved keyword.

```
# List of token names
tokens = (
    'ID',         # Identifier (variable name)
    'INT',        # Integer constant
    'SEMICOLON', # Semicolon
    'COMMA',      # Comma
    'TYPE',       # Data types (int, char, etc.)
)

# Regular expressions for tokens
t_SEMICOLON = r';'
t_COMMA = r','
t_TYPE = r'int|char|float|double'

def t_ID(t):
    r'[a-zA-Z_][a-zA-Z_0-9]*'
    t.type = reserved.get(t.value, 'ID')  # Check for reserved words
    return t

def t_INT(t):
    r'\d+'
    t.value = int(t.value)
    return t

# Ignored characters (whitespace and newline)
t_ignore = ' \t\n'

# Reserved words
reserved = {
    'int': 'TYPE',
    'char': 'TYPE',
    'float': 'TYPE',
    'double': 'TYPE',
}
```

Now we can write the grammar for our problem:

```
Declaration → Type VariableList Semicolon
VariableList → ID | ID Comma VariableList
```

Note that
{ID, Type, Semicolon, Comma} are tokens and not non-terminals like Declaration and VariableList.

```
def p_declaration(p):
    '''declaration : TYPE var_list SEMICOLON'''
```

```
def p_var_list(p):
    '''var_list : ID
                  | ID COMMA var_list'''
```

TODO:
This grammar concerns itself with just declaring variables. C also allows us to assign these variables with a value while declaring them. Write the grammar for the same in PLY.

6.2 Expression Validation

lexer.py

```
import ply.lex as lex

tokens = (
   'NUMBER',
   'PLUS',
   'MINUS',
   'TIMES',
   'DIVIDE',
   'LPAREN',
   'RPAREN',
)

# Regular expression rules for simple tokens
t_PLUS    = r'\+'
t_MINUS   = r'-'
t_TIMES   = r'\*'
t_DIVIDE  = r'/'
t_LPAREN  = r'\('
t_RPAREN  = r'\)'

# A regular expression rule with some action code
def t_NUMBER(t):
    r'\d+'
    t.value = int(t.value)
    return t

# Define a rule so we can track line numbers
def t_newline(t):
    r'\n+'
    t.lexer.lineno += len(t.value)

# A string containing ignored characters (spaces and tabs)
t_ignore  = ' \t'

# Error handling rule
def t_error(t):
    print("Illegal character '%s'" % t.value[0])
    t.lexer.skip(1)

# Build the lexer
lexer = lex.lex()
```

parser.py

```
import ply.yacc as yacc

# Get the token map from the lexer.  This is required.
from lexer import tokens

def p_expression_plus(p):
    'expression : expression PLUS term'

def p_expression_minus(p):
    'expression : expression MINUS term'

def p_expression_term(p):
    'expression : term'

def p_term_times(p):
    'term : term TIMES factor'

def p_term_div(p):
    'term : term DIVIDE factor'

def p_term_factor(p):
    'term : factor'

def p_factor_num(p):
    'factor : NUMBER'

def p_factor_expr(p):
    'factor : LPAREN expression RPAREN'

# Error rule for syntax errors
def p_error(p):
    print("Syntax error in input!")
    exit(1)

# Build the parser
parser = yacc.yacc()
```

main.py

```
from lexer import lexer
from parser import parser

while True:
      data = input("Enter a valid numerical expression: ")
      lexer.input(data)
      parsed = parser.parse(data)
      if parsed == None:
            print("Accepted")
```

TODO:
Evaluating arithmetic expressions can give rise to ambiguity. Modify the grammar to take care of this. (Hint: BODMAS)

Once you are done with these exercises, you should test them out! Run these programs and test your answer by coming up with all kinds of input cases.

Not sure how to run them? The next section contains the basic structure of simple programs in PLY.

!!! MAKE SURE TO REPLACE THE EXAMPLES WITH THEIR APPROPRIATE COUNTERPARTS. OTHERWISE, THE CODE WILL NOT RUN. READ THE COMMENTS CAREFULLY AND FILL IN THE APPROPRIATE BLANKS !!!

7. Skeleton Codes for your reference:

7.1 lexer.py

```
# Importing lex
import ply.lex as lex

# List of token names. This is always required
tokens = ('EXAMPLE1', 'EXAMPLE2', 'EXAMPLE3') # Example

# Regular expression rules for simple tokens
t_EXAMPLE1 = r'<example1>' # ADD YOUR OWN EXAMPLES!!
t_EXAMPLE2 = r'<example2>' # This is just an example for your reference.

# A regular expression rule with some action code
def t_EXAMPLE3(t):
    <do something with t>
# A string containing ignored characters (spaces and tabs)
t_ignore  = ' \t'

# Error handling rule
def t_error(t):
    print(f"Illegal character '{t.value[0]}'")
    t.lexer.skip(1)

# Building the lexer
lexer = lex.lex()
```

7.2 parser.py

```
# Importing lex
import ply.yacc as yacc

# Get the token map from the lexer.
from lexer import tokens

def p_expression(p): # An example to highlight the syntax of a grammar
    'expression : expression PLUS term'

def p_expression(p): # An example for combining grammar rules
    '''expression : expression PLUS term
                  | expression MINUS term'''

# Error rule for syntax errors
def p_error(p):
    print("Syntax error")
    exit(1)

# Build the parser
parser = yacc.yacc()
```

7.3 main.py

```
from lexer import lexer
from parser import parser

while True:
      data = input("Enter your input: ")
      lexer.input(data)
      parsed = parser.parse(data)
      if parsed == None:
            print("Accepted")
```

Acknowledgements:

[1] https://www.oreilly.com/library/view/lex-yacc/9781565920002/ch01.html

[2] https://www.dabeaz.com/ply/ply.html

References:

I highly recommend you to read the official documentation [2] and constantly refer to it for any clarifications. Here is a shorter version of it:
https://devtut.github.io/python/python-lex-yacc.html#python-lex-yacc
Here is a resource to help understand BNF grammars, which is the grammar used by lex and yacc.
https://cs61.seas.harvard.edu/site/2021/BNFGrammars/

Feel free to contact me for any clarifications/feedback.