Compiler Design

Python Lex-Yacc Project

Teaching Assistant Junhui Kim (jhkim.cau@gmail.com)

School of Computer Science and Engineering, Chung-Ang University, Seoul, Republic of Korea

- Python Lex-Yacc (PLY) is a Python version of the Lex-Yacc.

- PLY consists of two Python modules.
  - ply.lex
  - ply.yacc

- To use PLY, simply import the modules.

- PLY provides extensive error reporting and diagnostic information to assist in parser construction.

- PLY provides full support for empty productions, error recovery, precedence specifiers, and moderately ambiguous grammars.

- Parsing is based on LR-parsing which is fast, memory efficient, better suited to large grammars, and which has a number of nice properties when dealing with syntax errors and other parsing problems.

- Currently, PLY builds its parsing tables using the LALR(1) algorithm used in yacc.

- Only two pure-Python modules. And it is not part of "parser framework".
- Doesn't rely upon C extension modules or third party tools.
- Underlying parser is table driven.
- Parsing tables are saved and only regenerated if the grammar changes.

- A module for writing lexer.
- Tokens are specified using regular expressions.
- Provides functions for reading input text.

- t_[token name] : defines the token as variable or function.
- input() : feeds a string into the lexer.
- token() : returns the next token or None.
  - type
  - value
  - line
  - lexpos

- You don’t need to the tokenizer directly. This is used by the parser module.

$$x = 3 + 42 * (s - t)$$

- A tokenizer splits the string into individual tokens.
  - 'x', '=', '3', '+', '42', '*', '(', 's', '-', 't', ')'

- Tokens are usually given name to indicate what they are.
  - 'ID', 'EQUALS', 'NUMBER', 'PLUS', 'NUMBER', 'TIMES', 'LPAREN', 'ID', 'MINUS', 'ID', 'RPAREN'

- Finally, the input is broken into pairs of token types and values.
  - ('ID','x'), ('EQUALS','='), ('NUMBER','3'), ('PLUS','+'), ('NUMBER','42), ('TIMES','*'),
  - ('LPAREN','('), ('ID','s'), ('MINUS','-'), ('ID','t'), ('RPAREN',')'

Example

3 + 4 * 10 + -20 * 2

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

- Define a list of tokens that can be produced by the lexer.
- This is also used by the yacc module to identify terminals.

Example

3 + 4 * 10 + -20 * 2

```
1  t_PLUS    = r'\+'
2  t_MINUS   = r'-'
3  t_TIMES   = r'\*'
4  t_DIVIDE  = r'/'
5  t_LPAREN  = r'\('
6  t_RPAREN  = r'\)'
7
8  def t_NUMBER(t):
9      r'\d+'
10     t.value = int(t.value)
11     return t
12
13 def t_newline(t):
14     r'\n+'
15     t.lexer.lineno += len(t.value)
16
17 t_ignore  = ' \t'
18
19 def t_error(t):
20     print("Illegal character '%s'" % t.value[0])
21     t.lexer.skip(1)
22
23 lexer = lex.lex()
```

- Define regular expression rules for tokens.
  - Line 1~6 are the regular expression rules for simple tokens.
  - Line 8~11 is a rule with action code.
- Line 13~15 defines a rule which tracks line numbers.
- Line 17 ignores spaces and tabs.
- Line 19~21 handles the error.
- Line 23 builds the lexer.

Example

$$3 + 4 * 10 + -20 * 2$$

```
1  data = '''
2  3 + 4 * 10
3    + -20 *2
4  '''
5
6  lexer.input(data)
7
8  while True:
9      tok = lexer.token()
10     if not tok:
11         break
12     print(tok)
```

- Line 1~4 is a data to test.
- Line 6 feeds the input using input() method.
- Line 8~12 tokenizes repeating token() method.

```
Grammar                                   Action
-------------------------------           -------------------------------------------
expression0 : expression1 + term          expression0.val = expression1.val + term.val
            | expression1 - term          expression0.val = expression1.val - term.val
            | term                        expression0.val = term.val

term0       : term1 * factor              term0.val = term1.val * factor.val
            | term1 / factor              term0.val = term1.val / factor.val
            | factor                      term0.val = factor.val

factor      : NUMBER                      factor.val = int(NUMBER.lexval)
            | ( expression )              factor.val = expression.val
```

- An unambiguous grammar specification is needed.
  - First, write the expression grammar.
  - Then, write the specification for that grammar.

```
Step Symbol Stack              Input Tokens          Action
---- --------------------      --------------------  ------------------------------
1                              3 + 5 * ( 10 - 20 )$  Shift 3
2     3                          + 5 * ( 10 - 20 )$  Reduce factor : NUMBER
3     factor                     + 5 * ( 10 - 20 )$  Reduce term   : factor
4     term                       + 5 * ( 10 - 20 )$  Reduce expr : term
5     expr                       + 5 * ( 10 - 20 )$  Shift +
6     expr +                       5 * ( 10 - 20 )$  Shift 5
7     expr + 5                       * ( 10 - 20 )$  Reduce factor : NUMBER
8     expr + factor                  * ( 10 - 20 )$  Reduce term   : factor
9     expr + term                    * ( 10 - 20 )$  Shift *
10    expr + term *                    ( 10 - 20 )$  Shift (
11    expr + term * (                    10 - 20 )$  Shift 10
12    expr + term * ( 10                    - 20 )$  Reduce factor : NUMBER
13    expr + term * ( factor                - 20 )$  Reduce term : factor
14    expr + term * ( term                  - 20 )$  Reduce expr : term
15    expr + term * ( expr                  - 20 )$  Shift -
16    expr + term * ( expr -                  20 )$  Shift 20
17    expr + term * ( expr - 20                  )$  Reduce factor : NUMBER
18    expr + term * ( expr - factor              )$  Reduce term : factor
19    expr + term * ( expr - term                )$  Reduce expr : expr - term
20    expr + term * ( expr                       )$  Shift )
21    expr + term * ( expr )                      $  Reduce factor : (expr)
22    expr + term * factor                        $  Reduce term : term * factor
23    expr + term                                 $  Reduce expr : expr + term
24    expr                                        $  Reduce expr
25                                                $  Success!
```

- A module for creating a parser.
- You should define a BNF grammar.

- p_[rule name]() : encodes grammar rules as functions.
    - p_assign(p)
    - p_expr(p)
    - p_term(p)
    - p_factor(p)
    - …
- yacc() : builds the parser using introspection.
- parse() : parses the text and invokes grammar rules.

Example

```
1  from calclex import tokens
2
3  def p_expression_plus(p):
4      'expression : expression PLUS term'
5      p[0] = p[1] + p[3]
6
7  def p_expression_minus(p):
8      'expression : expression MINUS term'
9      p[0] = p[1] - p[3]
10
11 def p_expression_term(p):
12     'expression : term'
13     p[0] = p[1]
```

- Line 1 imports tokens of the lexer.
- Line 3~13 define the grammar rules for expression.

Example

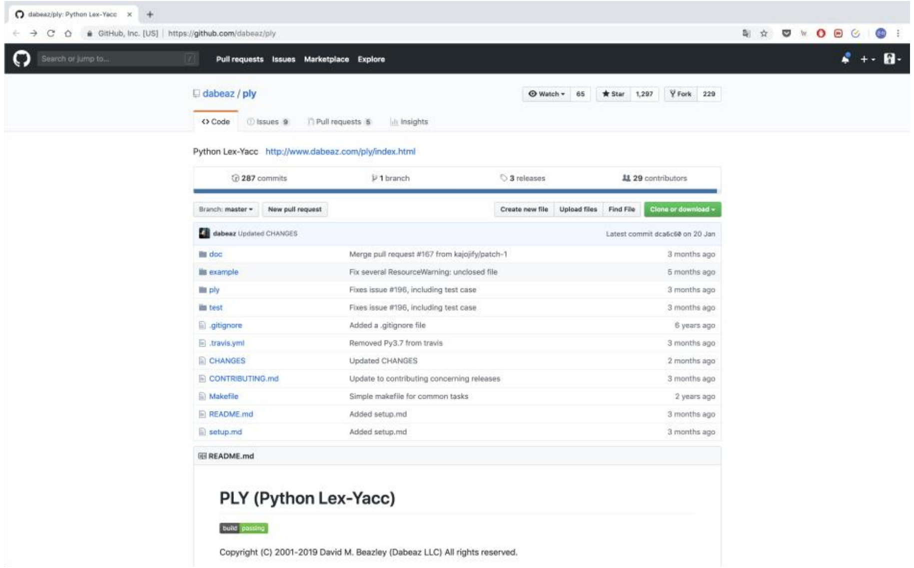
```
1  def p_term_times(p):
2      'term : term TIMES factor'
3      p[0] = p[1] * p[3]
4
5  def p_term_div(p):
6      'term : term DIVIDE factor'
7      p[0] = p[1] / p[3]
8
9  def p_term_factor(p):
10     'term : factor'
11     p[0] = p[1]
12
13 def p_factor_num(p):
14     'factor : NUMBER'
15     p[0] = p[1]
16
17 def p_factor_expr(p):
18     'factor : LPAREN expression RPAREN'
19     p[0] = p[2]
```

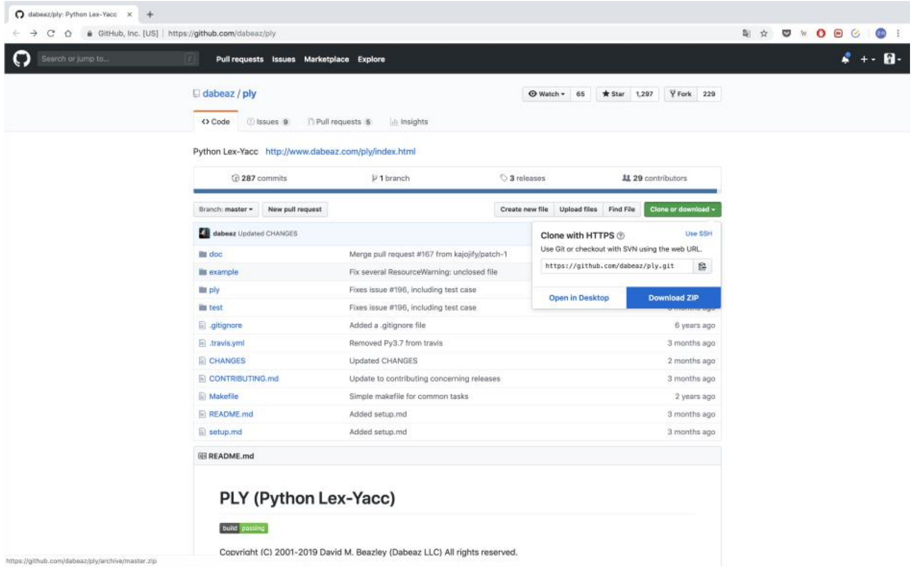
- Line 1~11 define the grammar rule for term.
- Lien 13~19 define the grammar rule for factor.

Example

```python
1  def p_error(p):
2      print("Syntax error in input!")
3
4  parser = yacc.yacc()
5
6  while True:
7     try:
8         s = raw_input('calc > ')
9     except EOFError:
10        break
11    if not s: continue
12    result = parser.parse(s)
13    print(result)
```

- Line 1~2 define error rule for syntax errors.
- Line 4 builds the parser.
- Line 6~13 parses the input.





- Move to https://github.com/dabeaz/ply
- Download this.

- 'lex.py' and 'yacc.py' are contained within the 'ply' directory which may also be used as a Python package.

- To use PLY, simply copy the 'ply' directory to your project and import lex and yacc from the associated 'ply' package.

- Alternatively, you can copy just the files 'lex.py' and 'yacc.py' individually and use them as modules.

```
from .ply import lex
from .ply import yacc
```

```
import lex
import yacc
```

Example

```
1 tokens = (
2     'NAME','NUMBER',
3     'PLUS','MINUS','TIMES','DIVIDE','EQUALS',
4     'LPAREN','RPAREN',
5     )
6
7 t_PLUS    = r'\+'
8 t_MINUS   = r'-'
9 t_TIMES   = r'\*'
10 t_DIVIDE  = r'/'
11 t_EQUALS  = r'='
12 t_LPAREN  = r'\('
13 t_RPAREN  = r'\)'
14 t_NAME    = r'[a-zA-Z_][a-zA-Z0-9_]*'
15
16 def t_NUMBER(t):
17     r'\d+'
18     t.value = int(t.value)
19     return t
20
21 t_ignore = " \t"
22
23 def t_newline(t):
24     r'\n+'
25     t.lexer.lineno += t.value.count("\n")
26
27 def t_error(t):
28     print("Illegal character '%s'" % t.value[0])
29     t.lexer.skip(1)
```

- Define the token types
  - Line 1~5
- Define a list of tokens.
  - Line 7~19
  - r'' means a regular expression.
  - Token could also be a function.
    - t_NUMBER(t)
- t_ignore, t_newline(t), t_error(t) are ignored characters.
  - Line 21~29

Example

```
 1 import ply.lex as lex
 2 lex.lex()
 3
 4 precedence = (
 5     ('left','PLUS','MINUS'),
 6     ('left','TIMES','DIVIDE'),
 7     ('right','UMINUS'),
 8     )
 9
10 names = { }
```

- Line 1: import the lex module for building the lexer.
- Line 2: build the lexer using lex() method.
- Line 4~8: define a set of precedence rules for the arithmetic operators.
- Line 10: for storing variables, make an object named 'names' which is a dictionary of names.

Example

```python
1 def p_statement_assign(p):
2     'statement : NAME EQUALS expression'
3     names[p[1]] = p[3]
4
5 def p_statement_expr(p):
6     'statement : expression'
7     print(p[1])
```

- Line 1~3: define the assign statement rule.
- Line 5~7: define the expression statement rule.
- The index of parameter p means the element of statement.
  - For example, in assign statement, the index of 'statement' is 0, 'NAME' is 1, 'EQUALS' is 2, and 'expression' is 3.

Example

```
1 def p_expression_binop(p):
2     '''expression : expression PLUS expression
3                   | expression MINUS expression
4                   | expression TIMES expression
5                   | expression DIVIDE expression'''
6     if p[2] == '+'  : p[0] = p[1] + p[3]
7     elif p[2] == '-': p[0] = p[1] - p[3]
8     elif p[2] == '*': p[0] = p[1] * p[3]
9     elif p[2] == '/': p[0] = p[1] / p[3]
10
11 def p_expression_uminus(p):
12     'expression : MINUS expression %prec UMINUS'
13     p[0] = -p[2]
14
15 def p_expression_group(p):
16     'expression : LPAREN expression RPAREN'
17     p[0] = p[2]
18
19 def p_expression_number(p):
20     'expression : NUMBER'
21     p[0] = p[1]
22
23 def p_expression_name(p):
24     'expression : NAME'
25     try:
26         p[0] = names[p[1]]
27     except LookupError:
28         print("Undefined name '%s'" % p[1])
29         p[0] = 0
30
31 def p_error(p):
32     print("Syntax error at '%s'" % p.value)
```

- Line 1~9: define the expression rule for binary operation.
- Line 11~13: define the unary minus expression rule.
- Line 15~17: define the group expression rule.
- Line 19~21: define the number expression rule.
- Line 23~29: define the expression rule for variables.
- Line 31~32: define the syntax error and print the error log.

Example

```python
1 import ply.yacc as yacc
2 yacc.yacc()
3
4 while True:
5     try:
6         s = input('calc > ')
7     except EOFError:
8         break
9     yacc.parse(s)
```

- Line 1~2: import yacc and build the parser.
- Line 4~9: receive the input from user and parse it.

```
(venv) C:\Users\DPS\PycharmProjects\ply>python calc.py
calc > a=60
calc > b=70
calc > a*2-30+b/2
125.0
calc > a
60
calc > b
70
calc > c = a+b
calc > c
130
calc > exit
Traceback (most recent call last):
  File "calc.py", line 119, in <module>
    s = input('calc > ')  # Use raw_input on Python 2
KeyboardInterrupt
```

- Run 'python calc.py' to execute our lex-yacc program.
- Enter the equations to calculate.
- After the program executes, 'parser.out' and 'parsetab.py' are generated.
- If you want to exit the program, enter 'control + c'.

```
calc.py     parser.out  parsetab.py ply
~/Desktop/my-compiler  cat parsetab.py

# parsetab.py
# This file is automatically generated. Do not edit.
# pylint: disable=W,C,R
_tabversion = '3.10'

_lr_method = 'LALR'

_lr_signature = 'leftPLUSMINUSleftTIMESDIVIDErightUMINUSDIVIDE EQUALS LPAREN MINUS NAME NUMBER PLUS RPAREN TIMESstatement : NAME EQUALS expressionstatement : expressio
nexpression : expression PLUS expression\n                    | expression MINUS expression\n                    | expression TIMES expression\n                    | express
ion DIVIDE expressionexpression : MINUS expression %prec UMINUSexpression : LPAREN expression RPARENexpression : NUMBERexpression : NAME'

_lr_action_items = {'$end':([1,2,3,5,9,14,15,16,17,18,19,20,],[-10,-9,0,-2,-10,-7,-1,-8,-3,-6,-4,-5,]),'RPAREN':([2,8,9,14,16,17,18,19,20,],[-9,16,-10,-7,-8,-3,-6,-4,-
5,]),'DIVIDE':([1,2,5,8,9,14,15,16,17,18,19,20,],[-10,-9,11,11,-10,-7,11,-8,11,-6,11,-5,]),'EQUALS':([1,],[7,]),'NUMBER':([0,4,6,7,10,11,12,13,],[2,2,2,2,2,2,2,2,]),'P
LUS':([1,2,5,8,9,14,15,16,17,18,19,20,],[-10,-9,10,10,-10,-7,10,-8,-3,-6,-4,-5,]),'LPAREN':([0,4,6,7,10,11,12,13,],[4,4,4,4,4,4,4,4,]),'TIMES':([1,2,5,8,9,14,15,16,17,
18,19,20,],[-10,-9,13,13,-10,-7,13,-8,13,-6,13,-5,]),'MINUS':([0,1,2,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,],[6,-10,-9,6,12,6,6,12,-10,6,6,6,6,-7,12,-8,-3,-6,-4
,-5,]),'NAME':([0,4,6,7,10,11,12,13,],[1,9,9,9,9,9,9,9,]),}

_lr_action = {}
for _k, _v in _lr_action_items.items():
   for _x,_y in zip(_v[0],_v[1]):
      if not _x in _lr_action:  _lr_action[_x] = {}
      _lr_action[_x][_k] = _y
del _lr_action_items

_lr_goto_items = {'expression':([0,4,6,7,10,11,12,13,],[5,8,14,15,17,18,19,20,]),'statement':([0,],[3,]),}

_lr_goto = {}
for _k, _v in _lr_goto_items.items():
   for _x, _y in zip(_v[0], _v[1]):
       if not _x in _lr_goto: _lr_goto[_x] = {}
       _lr_goto[_x][_k] = _y
del _lr_goto_items
_lr_productions = [
  ("S' -> statement","S'",1,None,None,None),
  ('statement -> NAME EQUALS expression','statement',3,'p_statement_assign','calc.py',55),
  ('statement -> expression','statement',1,'p_statement_expr','calc.py',59),
  ('expression -> expression PLUS expression','expression',3,'p_expression_binop','calc.py',63),
  ('expression -> expression MINUS expression','expression',3,'p_expression_binop','calc.py',64),
  ('expression -> expression TIMES expression','expression',3,'p_expression_binop','calc.py',65),
  ('expression -> expression DIVIDE expression','expression',3,'p_expression_binop','calc.py',66),
  ('expression -> MINUS expression','expression',2,'p_expression_uminus','calc.py',73),
  ('expression -> LPAREN expression RPAREN','expression',3,'p_expression_group','calc.py',77),
  ('expression -> NUMBER','expression',1,'p_expression_number','calc.py',81),
  ('expression -> NAME','expression',1,'p_expression_name','calc.py',85),
]
~/Desktop/my-compiler
```

- 'parsertab.py' is a table for parsing.

Example

```
~/Desktop/my-compiler  cat parser.out
Created by PLY version 3.11 (http://www.dabeaz.com/ply)

Grammar

Rule 0     S' -> statement
Rule 1     statement -> NAME EQUALS expression
Rule 2     statement -> expression
Rule 3     expression -> expression PLUS expression
Rule 4     expression -> expression MINUS expression
Rule 5     expression -> expression TIMES expression
Rule 6     expression -> expression DIVIDE expression
Rule 7     expression -> MINUS expression
Rule 8     expression -> LPAREN expression RPAREN
Rule 9     expression -> NUMBER
Rule 10    expression -> NAME

Terminals, with rules where they appear

DIVIDE               : 6
EQUALS               : 1
LPAREN               : 8
MINUS                : 4 7
NAME                 : 1 10
NUMBER               : 9
PLUS                 : 3
RPAREN               : 8
TIMES                : 5
error                :

Nonterminals, with rules where they appear

expression           : 1 2 3 3 4 4 5 5 6 6 7 8
statement            : 0

Parsing method: LALR

state 0

    (0) S' -> . statement
    (1) statement -> . NAME EQUALS expression
    (2) statement -> . expression
    (3) expression -> . expression PLUS expression
    (4) expression -> . expression MINUS expression
    (5) expression -> . expression TIMES expression
    (6) expression -> . expression DIVIDE expression
    (7) expression -> . MINUS expression
    (8) expression -> . LPAREN expression RPAREN
    (9) expression -> . NUMBER
```

- 'parser.out' is a debugging file.

- Parse the c source files and count how many items are used.

- You should count and print the result of the following.
  - #include
  - Declared Functions
  - Declared Variables
  - Conditional Statements
  - Loop
  - Called Functions

Project


```c
#include <stdio.h>

int main() {
    int num;
    int i;


    printf("Enter the number : ");
    scanf("%d", &num);


    for (i = 2; i < num; i++) {
        printf("%d %% %d = %d\n", num, i, num % i);
        if (num % i == 0) {
            break;
        }
    }


    if (i == num) {
        printf("Prime");
    } else {
        printf("Not Prime");
    }


    return 0;
}
```

Input source file

```
#include: 1
#define: 0

Declared Functions: 1
Declared Variables: 2

Conditional Statements: 2
Loop: 1

Called Functions: 5
```

Output

- #include : <stdio.h>
- Declared Functions : int main()
- Declared Variables : int num, int i
- Conditional Statements : if(i==num){…}else{…}, if(num%i==0){…}
- Loop : for(i=2;i<num;i++){…}
- Called Functions : printf(), scanf(), printf(), printf(), printf()

- You should parse
  - #include
  - int, void, array
  - return, break
  - if(…){…}, if(…){…}else{…}
  - for(…;…;…){…}, while(…){…}
  - +, -, *, /, %, ++, --
  - <, >, ==, >=, <=
  - &&, ||, !, &
  - =
  - printf(), scanf()
  - Etc.