dabeaz

PLY (Python Lex-Yacc)

David M. Beazley
dave@dabeaz.com
PLY Version: 3.11

- Preface and Requirements
- Introduction
- PLY Overview
- Lex
  - Lex Example
  - The tokens list
  - Specification of tokens
  - Token values
  - Discarded tokens
  - Line numbers and positional information
  - Ignored characters
  - Literal characters
  - Error handling
  - EOF Handling
  - Building and using the lexer
  - The @TOKEN decorator
  - Optimized mode
  - Debugging
  - Alternative specification of lexers
  - Maintaining state
  - Lexer cloning
  - Internal lexer state
  - Conditional lexing and start conditions
  - Miscellaneous Issues
- Parsing basics
- Yacc
  - An example
  - Combining Grammar Rule Functions
  - Character Literals
  - Empty Productions
  - Changing the starting symbol
  - Dealing With Ambiguous Grammars
  - The parser.out file
  - Syntax Error Handling
    - Recovery and resynchronization with error rules

        - Panic mode recovery
        - Signalling an error from a production
        - When Do Syntax Errors Get Reported
        - General comments on error handling
    - Line Number and Position Tracking
    - AST Construction
    - Embedded Actions
    - Miscellaneous Yacc Notes
- Multiple Parsers and Lexers
- Using Python's Optimized Mode
- Advanced Debugging
    - Debugging the lex() and yacc() commands
    - Run-time Debugging
- Packaging Advice
- Where to go from here?

1. Preface and Requirements

This document provides an overview of lexing and parsing with PLY. Given the intrinsic complexity of parsing, I would strongly advise that you read (or at least skim) this entire document before jumping into a big development project with PLY.

PLY-3.5 is compatible with both Python 2 and Python 3. If you are using Python 2, you have to use Python 2.6 or newer.

2. Introduction

PLY is a pure-Python implementation of the popular compiler construction tools lex and yacc. The main goal of PLY is to stay fairly faithful to the way in which traditional lex/yacc tools work. This includes supporting LALR(1) parsing as well as providing extensive input validation, error reporting, and diagnostics. Thus, if you've used yacc in another programming language, it should be relatively straightforward to use PLY.

Early versions of PLY were developed to support an Introduction to Compilers Course I taught in 2001 at the University of Chicago. Since PLY was primarily developed as an instructional tool, you will find it to be fairly picky about token and grammar rule specification. In part, this added formality is meant to catch common programming mistakes made by novice users. However, advanced users will also find such features to be useful when building complicated grammars for real programming languages. It should also be noted that PLY does not provide much in the way of bells and whistles (e.g., automatic construction of abstract syntax trees, tree traversal, etc.). Nor would I consider it to be a parsing framework. Instead, you will find a bare-bones, yet fully capable lex/yacc implementation written entirely in Python.

The rest of this document assumes that you are somewhat familiar with parsing theory, syntax directed translation, and the use of compiler construction tools such as lex and yacc in other programming languages. If you are unfamiliar with these topics, you will probably want to consult an introductory text such as "Compilers: Principles, Techniques, and Tools", by Aho, Sethi, and Ullman. O'Reilly's "Lex and Yacc" by John Levine may also be handy. In fact, the O'Reilly book can be used as a reference for PLY as the concepts are virtually identical.

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. The output of `yacc.py` is often an Abstract Syntax Tree (AST). However, this is entirely up to the user. If desired, `yacc.py` can also be used to implement simple one-pass compilers.

Like its Unix counterpart, `yacc.py` provides most of the features you expect including extensive error checking, grammar validation, support for empty productions, error tokens, and ambiguity resolution via precedence rules. In fact, almost everything that is possible in traditional yacc should be supported in PLY.

The primary difference between `yacc.py` and Unix `yacc` is that `yacc.py` doesn't involve a separate code-generation process. Instead, PLY relies on reflection (introspection) to build its lexers and parsers. Unlike traditional lex/yacc which require a special input file that is converted into a separate source file, the specifications given to PLY *are* valid Python programs. This means that there are no extra source files nor is there a special compiler construction step (e.g., running yacc to generate Python code for the compiler). Since the generation of the parsing tables is relatively expensive, PLY caches the results and saves them to a file. If no changes are detected in the input source, the tables are read from the cache. Otherwise, they are regenerated.

4. Lex

`lex.py` is used to tokenize an input string. For example, suppose you're writing a programming language and a user supplied the following input string:

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

A tokenizer splits the string into individual tokens

```
'x','=', '3', '+', '42', '*', '(', 's', '-', 't', ')'
```

Tokens are usually given names to indicate what they are. For example:

```
'ID','EQUALS','NUMBER','PLUS','NUMBER','TIMES',
'LPAREN','ID','MINUS','ID','RPAREN'
```

More specifically, the input is broken into pairs of token types and values. For example:

```
('ID','x'), ('EQUALS','='), ('NUMBER','3'),
('PLUS','+'), ('NUMBER','42), ('TIMES','*'),
('LPAREN','('), ('ID','s'), ('MINUS','-'),
('ID','t'), ('RPAREN',')'
```

The identification of tokens is typically done by writing a series of regular expression rules. The next section shows how this is done using `lex.py`.

4.1 Lex Example

The following example shows how `lex.py` is used to write a simple tokenizer.

```
# ------------------------------------------------------------
# calclex.py
#
# tokenizer for a simple expression evaluator for
# numbers and +,-,*,/
# ------------------------------------------------------------
import ply.lex as lex

# List of token names.   This is always required
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()
```

To use the lexer, you first need to feed it some input text using its `input()` method. After that, repeated calls to `token()` produce tokens. The following code shows how this works:

```
# Test it out
data = '''
3 + 4 * 10
  + -20 *2
'''

# Give the lexer some input
lexer.input(data)

# Tokenize
while True:
    tok = lexer.token()
    if not tok:
        break      # No more input
    print(tok)
```

When executed, the example will produce the following output:

```
$ python example.py
LexToken(NUMBER,3,2,1)
LexToken(PLUS,'+',2,3)
LexToken(NUMBER,4,2,5)
LexToken(TIMES,'*',2,7)
LexToken(NUMBER,10,2,10)
LexToken(PLUS,'+',3,14)
LexToken(MINUS,'-',3,16)
LexToken(NUMBER,20,3,18)
LexToken(TIMES,'*',3,20)
LexToken(NUMBER,2,3,21)
```

Lexers also support the iteration protocol. So, you can write the above loop as follows:

```
for tok in lexer:
    print(tok)
```

The tokens returned by `lexer.token()` are instances of `LexToken`. This object has attributes `tok.type`, `tok.value`, `tok.lineno`, and `tok.lexpos`. The following code shows an example of accessing these attributes:

```
# Tokenize
while True:
    tok = lexer.token()
    if not tok:
        break      # No more input
    print(tok.type, tok.value, tok.lineno, tok.lexpos)
```

The `tok.type` and `tok.value` attributes contain the type and value of the token itself. `tok.line` and `tok.lexpos` contain information about the location of the token. `tok.lexpos` is the index of the token relative to the start of the input text.

4.2 The tokens list

All lexers must provide a list `tokens` that defines all of the possible token names that can be produced by the lexer. This list is always required and is used to perform a variety of validation checks. The tokens list is also used by the `yacc.py` module to identify terminals.

In the example, the following code specified the token names:

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

4.3 Specification of tokens

Each token is specified by writing a regular expression rule compatible with Python's `re` module. Each of these rules are defined by making declarations with a special prefix `t_` to indicate that it defines a token. For simple tokens, the regular expression can be specified as strings such as this (note: Python raw strings are used since they are the most convenient way to write regular expression strings):

```
t_PLUS = r'\+'
```

In this case, the name following the `t_` must exactly match one of the names supplied in `tokens`. If some kind of action needs to be performed, a token rule can be specified as a function. For example, this rule matches numbers and converts the string into a Python integer.

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

When a function is used, the regular expression rule is specified in the function documentation string. The function always takes a single argument which is an instance of `LexToken`. This object has attributes of `t.type` which is the token type (as a string), `t.value` which is the lexeme (the actual text matched), `t.lineno` which is the current line number, and `t.lexpos` which is the position of the token relative to the beginning of the input text. By default, `t.type` is set to the name following the `t_` prefix. The action function can modify the contents of the `LexToken` object as appropriate. However, when it is done, the resulting token should be returned. If no value is returned by the action function, the token is simply discarded and the next token read.

Internally, `lex.py` uses the `re` module to do its pattern matching. Patterns are compiled using the `re.VERBOSE` flag which can be used to help readability. However, be aware that unescaped whitespace is ignored and comments are allowed in this mode. If your pattern involves whitespace, make sure you use `\s`. If you need to match the `#` character, use `[#]`.

When building the master regular expression, rules are added in the following order:

1. All tokens defined by functions are added in the same order as they appear in the lexer file.
2. Tokens defined by strings are added next by sorting them in order of decreasing regular expression length (longer expressions are added first).

Without this ordering, it can be difficult to correctly match certain types of tokens. For example, if you wanted to have separate tokens for "=" and "==", you need to make sure that "==" is checked first. By sorting regular expressions in order of decreasing length, this problem is solved for rules defined as strings. For functions, the order can be explicitly controlled since rules appearing first are checked first.

To handle reserved words, you should write a single rule to match an identifier and do a special name lookup in a function like this:

```
reserved = {
   'if' : 'IF',
   'then' : 'THEN',
   'else' : 'ELSE',
   'while' : 'WHILE',
   ...
}

tokens = ['LPAREN','RPAREN',...,'ID'] + list(reserved.values())

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
```

This approach greatly reduces the number of regular expression rules and is likely to make things a little faster.

Note: You should avoid writing individual rules for reserved words. For example, if you write rules like this,

```
t_FOR   = r'for'
t_PRINT = r'print'
```

those rules will be triggered for identifiers that include those words as a prefix such as "forget" or "printed". This is probably not what you want.

4.4 Token values

When tokens are returned by lex, they have a value that is stored in the `value` attribute. Normally, the value is the text that was matched. However, the value can be assigned to any Python object. For instance, when lexing identifiers, you may want to return both the identifier name and information from some sort of symbol table. To do this, you might write a rule like this:

```
def t_ID(t):
    ...
    # Look up symbol table information and return a tuple
    t.value = (t.value, symbol_lookup(t.value))
    ...
    return t
```

It is important to note that storing data in other attribute names is *not* recommended. The `yacc.py` module only exposes the contents of the `value` attribute. Thus, accessing other attributes may be unnecessarily awkward. If you need to store multiple values on a token, assign a tuple, dictionary, or instance to `value`.

4.5 Discarded tokens

To discard a token, such as a comment, simply define a token rule that returns no value. For example:

```
def t_COMMENT(t):
    r'\#.*'
    pass
    # No return value. Token discarded
```

Alternatively, you can include the prefix "ignore_" in the token declaration to force a token to be ignored. For example:

```
t_ignore_COMMENT = r'\#.*'
```

Be advised that if you are ignoring many different kinds of text, you may still want to use functions since these provide more precise control over the order in which regular expressions are matched (i.e., functions are matched in order of specification whereas strings are sorted by regular expression length).

4.6 Line numbers and positional information

By default, `lex.py` knows nothing about line numbers. This is because `lex.py` doesn't know anything about what constitutes a "line" of input (e.g., the newline character or even if the input is textual data). To update this information, you need to write a special rule. In the example, the `t_newline()` rule shows how to do this.

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

Within the rule, the `lineno` attribute of the underlying lexer `t.lexer` is updated. After the line number is updated, the token is simply discarded since nothing is returned.

`lex.py` does not perform any kind of automatic column tracking. However, it does record positional information related to each token in the `lexpos` attribute. Using this, it is usually possible to compute column information as a separate step. For instance, just count backwards until you reach a newline.

```
# Compute column.
#     input is the input text string
#     token is a token instance
def find_column(input, token):
    line_start = input.rfind('\n', 0, token.lexpos) + 1
    return (token.lexpos - line_start) + 1
```

Since column information is often only useful in the context of error handling, calculating the column position can be performed when needed as opposed to doing it for each token.

4.7 Ignored characters

The special `t_ignore` rule is reserved by `lex.py` for characters that should be completely ignored in the input stream. Usually this is used to skip over whitespace and other non-essential characters. Although it is possible to define a regular expression rule for whitespace in a manner similar to `t_newline()`, the use of `t_ignore` provides substantially better lexing performance because it is handled as a special case and is checked in a much more efficient manner than the normal regular expression rules.

The characters given in `t_ignore` are not ignored when such characters are part of other regular expression patterns. For example, if you had a rule to capture quoted text, that pattern can include the ignored characters (which will be captured in the normal way). The main purpose of `t_ignore` is to ignore whitespace and other padding between the tokens that you actually want to parse.

4.8 Literal characters

Literal characters can be specified by defining a variable `literals` in your lexing module. For example:

```
literals = [ '+','-','*','/' ]
```

or alternatively

```
literals = "+-*/"
```

A literal character is simply a single character that is returned "as is" when encountered by the lexer. Literals are checked after all of the defined regular expression rules. Thus, if a rule starts with one of the literal characters, it will always take precedence.

When a literal token is returned, both its `type` and `value` attributes are set to the character itself. For example, `'+'`.

It's possible to write token functions that perform additional actions when literals are matched. However, you'll need to set the token type appropriately. For example:

```
literals = [ '{', '}' ]

def t_lbrace(t):
    r'\{'
    t.type = '{'      # Set token type to the expected literal
    return t

def t_rbrace(t):
    r'\}'
    t.type = '}'      # Set token type to the expected literal
    return t
```

4.9 Error handling

The `t_error()` function is used to handle lexing errors that occur when illegal characters are detected. In this case, the `t.value` attribute contains the rest of the input string that has not been tokenized. In the example, the error function was defined as follows:

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

In this case, we simply print the offending character and skip ahead one character by calling `t.lexer.skip(1)`.

4.10 EOF Handling

The `t_eof()` function is used to handle an end-of-file (EOF) condition in the input. As input, it receives a token type `'eof'` with the `lineno` and `lexpos` attributes set appropriately. The main use of this function is provide more input to the lexer so that it can continue to parse. Here is an example of how this works:

```
# EOF handling rule
def t_eof(t):
    # Get more input (Example)
    more = raw_input('... ')
    if more:
        self.lexer.input(more)
        return self.lexer.token()
    return None
```

The EOF function should return the next available token (by calling `self.lexer.token()`) or `None` to indicate no more data. Be aware that setting more input with the `self.lexer.input()` method does NOT reset the lexer state or the `lineno` attribute used for position tracking. The `lexpos` attribute is reset so be aware of that if you're using it in error reporting.

4.11 Building and using the lexer

To build the lexer, the function `lex.lex()` is used. For example:

```
lexer = lex.lex()
```

This function uses Python reflection (or introspection) to read the regular expression rules out of the calling context and build the lexer. Once the lexer has been built, two methods can be used to control the lexer.

- `lexer.input(data)`. Reset the lexer and store a new input string.
- `lexer.token()`. Return the next token. Returns a special `LexToken` instance on success or None if the end of the input text has been reached.

4.12 The @TOKEN decorator

In some applications, you may want to define build tokens from as a series of more complex regular expression rules. For example:

```
digit            = r'([0-9])'
nondigit         = r'([_A-Za-z])'
identifier       = r'(' + nondigit + r'(' + digit + r'|' + nondigit + r')*)'

def t_ID(t):
    # want docstring to be identifier above. ?????
    ...
```

In this case, we want the regular expression rule for `ID` to be one of the variables above. However, there is no way to directly specify this using a normal documentation string. To solve this problem, you can use the `@TOKEN` decorator. For example:

```
from ply.lex import TOKEN

@TOKEN(identifier)
def t_ID(t):
    ...
```

This will attach `identifier` to the docstring for `t_ID()` allowing `lex.py` to work normally.

4.13 Optimized mode

For improved performance, it may be desirable to use Python's optimized mode (e.g., running Python with the `-O` option). However, doing so causes Python to ignore documentation strings. This presents special problems for `lex.py`. To handle this case, you can create your lexer using the `optimize` option as follows:

```
lexer = lex.lex(optimize=1)
```

Next, run Python in its normal operating mode. When you do this, `lex.py` will write a file called `lextab.py` in the same directory as the module containing the lexer specification. This file contains all of the regular expression rules and tables used during lexing. On subsequent executions, `lextab.py` will simply be imported to build the lexer. This approach substantially improves the startup time of the lexer and it works in Python's optimized mode.

To change the name of the lexer-generated module, use the `lextab` keyword argument. For example:

```
lexer = lex.lex(optimize=1,lextab="footab")
```

When running in optimized mode, it is important to note that lex disables most error checking. Thus, this is really only recommended if you're sure everything is working correctly and you're ready to start releasing production code.

4.14 Debugging

For the purpose of debugging, you can run `lex()` in a debugging mode as follows:

```
lexer = lex.lex(debug=1)
```

This will produce various sorts of debugging information including all of the added rules, the master regular expressions used by the lexer, and tokens generating during lexing.

In addition, `lex.py` comes with a simple main function which will either tokenize input read from standard input or from a file specified on the command line. To use it, simply put this in your lexer:

```
if __name__ == '__main__':
     lex.runmain()
```

Please refer to the "Debugging" section near the end for some more advanced details of debugging.

4.15 Alternative specification of lexers

As shown in the example, lexers are specified all within one Python module. If you want to put token rules in a different module from the one in which you invoke `lex()`, use the `module` keyword argument.

For example, you might have a dedicated module that just contains the token rules:

```
# module: tokrules.py
# This module just contains the lexing rules

# List of token names.   This is always required
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)
```

Now, if you wanted to build a tokenizer from these rules from within a different module, you would do the following (shown for Python interactive mode):

```
>>> import tokrules
>>> lexer = lex.lex(module=tokrules)
>>> lexer.input("3 + 4")
>>> lexer.token()
LexToken(NUMBER,3,1,1,0)
>>> lexer.token()
LexToken(PLUS,'+',1,2)
>>> lexer.token()
LexToken(NUMBER,4,1,4)
>>> lexer.token()
None
>>>
```

The `module` option can also be used to define lexers from instances of a class. For example:

```
import ply.lex as lex

class MyLexer(object):
    # List of token names.   This is always required
    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
    # Note addition of self parameter since we're in a class
    def t_NUMBER(self,t):
        r'\d+'
        t.value = int(t.value)
        return t

    # Define a rule so we can track line numbers
    def t_newline(self,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(self,t):
        print("Illegal character '%s'" % t.value[0])
        t.lexer.skip(1)

    # Build the lexer
    def build(self,**kwargs):
        self.lexer = lex.lex(module=self, **kwargs)

    # Test it output
    def test(self,data):
        self.lexer.input(data)
        while True:
             tok = self.lexer.token()
             if not tok:
                 break
             print(tok)

# Build the lexer and try it out
m = MyLexer()
m.build()           # Build the lexer
m.test("3 + 4")     # Test it
```

When building a lexer from class, *you should construct the lexer from an instance of the class*, not the class object itself. This is because PLY only works properly if the lexer actions are defined by bound-methods.

When using the `module` option to `lex()`, PLY collects symbols from the underlying object using the `dir()` function. There is no direct access to the `__dict__` attribute of the object supplied as a module value.

Finally, if you want to keep things nicely encapsulated, but don't want to use a full-fledged class definition, lexers can be defined using closures. For example:

```
import ply.lex as lex

# List of token names.   This is always required
tokens = (
  'NUMBER',
  'PLUS',
  'MINUS',
  'TIMES',
  'DIVIDE',
  'LPAREN',
  'RPAREN',
)

def MyLexer():
    # 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 from my environment and return it
    return lex.lex()
```

Important note: If you are defining a lexer using a class or closure, be aware that PLY still requires you to only define a single lexer per module (source file). There are extensive validation/error checking parts of the PLY that may falsely report error messages if you don't

follow this rule.

4.16 Maintaining state

In your lexer, you may want to maintain a variety of state information. This might include mode settings, symbol tables, and other details. As an example, suppose that you wanted to keep track of how many NUMBER tokens had been encountered.

One way to do this is to keep a set of global variables in the module where you created the lexer. For example:

```
num_count = 0
def t_NUMBER(t):
    r'\d+'
    global num_count
    num_count += 1
    t.value = int(t.value)
    return t
```

If you don't like the use of a global variable, another place to store information is inside the Lexer object created by `lex()`. To this, you can use the `lexer` attribute of tokens passed to the various rules. For example:

```
def t_NUMBER(t):
    r'\d+'
    t.lexer.num_count += 1     # Note use of lexer attribute
    t.value = int(t.value)
    return t

lexer = lex.lex()
lexer.num_count = 0            # Set the initial count
```

This latter approach has the advantage of being simple and working correctly in applications where multiple instantiations of a given lexer exist in the same application. However, this might also feel like a gross violation of encapsulation to OO purists. Just to put your mind at some ease, all internal attributes of the lexer (with the exception of `lineno`) have names that are prefixed by `lex` (e.g., `lexdata`,`lexpos`, etc.). Thus, it is perfectly safe to store attributes in the lexer that don't have names starting with that prefix or a name that conflicts with one of the predefined methods (e.g., `input()`, `token()`, etc.).

If you don't like assigning values on the lexer object, you can define your lexer as a class as shown in the previous section:

```
class MyLexer:
    ...
    def t_NUMBER(self,t):
        r'\d+'
        self.num_count += 1
        t.value = int(t.value)
        return t

    def build(self, **kwargs):
        self.lexer = lex.lex(object=self,**kwargs)

    def __init__(self):
        self.num_count = 0
```

The class approach may be the easiest to manage if your application is going to be creating multiple instances of the same lexer and you need to manage a lot of state.

State can also be managed through closures. For example, in Python 3:

```
def MyLexer():
    num_count = 0
    ...
    def t_NUMBER(t):
        r'\d+'
        nonlocal num_count
        num_count += 1
        t.value = int(t.value)
        return t
    ...
```

4.17 Lexer cloning

If necessary, a lexer object can be duplicated by invoking its `clone()` method. For example:

```
lexer = lex.lex()
...
newlexer = lexer.clone()
```

When a lexer is cloned, the copy is exactly identical to the original lexer including any input text and internal state. However, the clone allows a different set of input text to be supplied which may be processed separately. This may be useful in situations when you are writing a parser/compiler that involves recursive or reentrant processing. For instance, if you needed to

scan ahead in the input for some reason, you could create a clone and use it to look ahead. Or, if you were implementing some kind of preprocessor, cloned lexers could be used to handle different input files.

Creating a clone is different than calling `lex.lex()` in that PLY doesn't regenerate any of the internal tables or regular expressions.

Special considerations need to be made when cloning lexers that also maintain their own internal state using classes or closures. Namely, you need to be aware that the newly created lexers will share all of this state with the original lexer. For example, if you defined a lexer as a class and did this:

```
m = MyLexer()
a = lex.lex(object=m)      # Create a lexer

b = a.clone()              # Clone the lexer
```

Then both `a` and `b` are going to be bound to the same object `m` and any changes to `m` will be reflected in both lexers. It's important to emphasize that `clone()` is only meant to create a new lexer that reuses the regular expressions and environment of another lexer. If you need to make a totally new copy of a lexer, then call `lex()` again.

4.18 Internal lexer state

A Lexer object `lexer` has a number of internal attributes that may be useful in certain situations.

`lexer.lexpos`

This attribute is an integer that contains the current position within the input text. If you modify the value, it will change the result of the next call to `token()`. Within token rule functions, this points to the first character *after* the matched text. If the value is modified within a rule, the next returned token will be matched at the new position.

`lexer.lineno`

The current value of the line number attribute stored in the lexer. PLY only specifies that the attribute exists---it never sets, updates, or performs any processing with it. If you want to track line numbers, you will need to add code yourself (see the section on line numbers and positional information).

`lexer.lexdata`

The current input text stored in the lexer. This is the string passed with the `input()` method. It would probably be a bad idea to modify this unless you really know what you're doing.

`lexer.lexmatch`

This is the raw `Match` object returned by the Python `re.match()` function (used internally by PLY) for the current token. If you have written a regular expression that contains named groups, you can use this to retrieve those values. Note: This attribute is only updated when tokens are defined and processed by functions.

4.19 Conditional lexing and start conditions

In advanced parsing applications, it may be useful to have different lexing states. For instance, you may want the occurrence of a certain token or syntactic construct to trigger a different kind of lexing. PLY supports a feature that allows the underlying lexer to be put into a series of different states. Each state can have its own tokens, lexing rules, and so forth. The implementation is based largely on the "start condition" feature of GNU flex. Details of this can be found at http://flex.sourceforge.net/manual/Start-Conditions.html.

To define a new lexing state, it must first be declared. This is done by including a "states" declaration in your lex file. For example:

```
states = (
   ('foo','exclusive'),
   ('bar','inclusive'),
)
```

This declaration declares two states, `'foo'` and `'bar'`. States may be of two types; `'exclusive'` and `'inclusive'`. An exclusive state completely overrides the default behavior of the lexer. That is, lex will only return tokens and apply rules defined specifically for that state. An inclusive state adds additional tokens and rules to the default set of rules. Thus, lex will return both the tokens defined by default in addition to those defined for the inclusive state.

Once a state has been declared, tokens and rules are declared by including the state name in token/rule declaration. For example:

```
t_foo_NUMBER = r'\d+'                       # Token 'NUMBER' in state 'foo'
t_bar_ID     = r'[a-zA-Z_][a-zA-Z0-9_]*'    # Token 'ID' in state 'bar'

def t_foo_newline(t):
    r'\n'
    t.lexer.lineno += 1
```

A token can be declared in multiple states by including multiple state names in the declaration. For example:

```
t_foo_bar_NUMBER = r'\d+'         # Defines token 'NUMBER' in both state 'foo' and
'bar'
```

Alternative, a token can be declared in all states using the 'ANY' in the name.

```
t_ANY_NUMBER = r'\d+'         # Defines a token 'NUMBER' in all states
```

If no state name is supplied, as is normally the case, the token is associated with a special state `'INITIAL'`. For example, these two declarations are identical:

```
t_NUMBER = r'\d+'
t_INITIAL_NUMBER = r'\d+'
```

States are also associated with the special `t_ignore`, `t_error()`, and `t_eof()` declarations. For example, if a state treats these differently, you can declare:

```
t_foo_ignore = " \t\n"       # Ignored characters for state 'foo'

def t_bar_error(t):          # Special error handler for state 'bar'
    pass
```

By default, lexing operates in the `'INITIAL'` state. This state includes all of the normally defined tokens. For users who aren't using different states, this fact is completely transparent. If, during lexing or parsing, you want to change the lexing state, use the `begin()` method. For example:

```
def t_begin_foo(t):
    r'start_foo'
    t.lexer.begin('foo')             # Starts 'foo' state
```

To get out of a state, you use `begin()` to switch back to the initial state. For example:

```
def t_foo_end(t):
    r'end_foo'
    t.lexer.begin('INITIAL')        # Back to the initial state
```

The management of states can also be done with a stack. For example:

```
def t_begin_foo(t):
    r'start_foo'
    t.lexer.push_state('foo')             # Starts 'foo' state

def t_foo_end(t):
    r'end_foo'
    t.lexer.pop_state()                   # Back to the previous state
```

The use of a stack would be useful in situations where there are many ways of entering a new lexing state and you merely want to go back to the previous state afterwards.

An example might help clarify. Suppose you were writing a parser and you wanted to grab sections of arbitrary C code enclosed by curly braces. That is, whenever you encounter a starting brace '{', you want to read all of the enclosed code up to the ending brace '}' and return it as a string. Doing this with a normal regular expression rule is nearly (if not actually) impossible. This is because braces can be nested and can be included in comments and strings. Thus, simply matching up to the first matching '}' character isn't good enough. Here is how you might use lexer states to do this:

```
# Declare the state
states = (
  ('ccode','exclusive'),
)

# Match the first {. Enter ccode state.
def t_ccode(t):
    r'\{'
    t.lexer.code_start = t.lexer.lexpos        # Record the starting position
    t.lexer.level = 1                          # Initial brace level
    t.lexer.begin('ccode')                     # Enter 'ccode' state

# Rules for the ccode state
def t_ccode_lbrace(t):
    r'\{'
    t.lexer.level +=1

def t_ccode_rbrace(t):
    r'\}'
    t.lexer.level -=1

    # If closing brace, return the code fragment
    if t.lexer.level == 0:
         t.value = t.lexer.lexdata[t.lexer.code_start:t.lexer.lexpos+1]
         t.type = "CCODE"
         t.lexer.lineno += t.value.count('\n')
         t.lexer.begin('INITIAL')
         return t
```

```
# C or C++ comment (ignore)
def t_ccode_comment(t):
    r'(/\*(.|\n)*?\*/)|(//.*)'
    pass

# C string
def t_ccode_string(t):
   r'\"([^\\\n]|(\\.))*?\"'

# C character literal
def t_ccode_char(t):
   r'\'([^\\\n]|(\\.))*?\''

# Any sequence of non-whitespace characters (not braces, strings)
def t_ccode_nonspace(t):
   r'[^\s\{\}\'\"]+'

# Ignored characters (whitespace)
t_ccode_ignore = " \t\n"

# For bad characters, we just skip over it
def t_ccode_error(t):
    t.lexer.skip(1)
```

In this example, the occurrence of the first '{' causes the lexer to record the starting position and enter a new state `'ccode'`. A collection of rules then match various parts of the input that follow (comments, strings, etc.). All of these rules merely discard the token (by not returning a value). However, if the closing right brace is encountered, the rule `t_ccode_rbrace` collects all of the code (using the earlier recorded starting position), stores it, and returns a token 'CCODE' containing all of that text. When returning the token, the lexing state is restored back to its initial state.

4.20 Miscellaneous Issues

- The lexer requires input to be supplied as a single input string. Since most machines have more than enough memory, this rarely presents a performance concern. However, it means that the lexer currently can't be used with streaming data such as open files or sockets. This limitation is primarily a side-effect of using the `re` module. You might be able to work around this by implementing an appropriate `def t_eof()` end-of-file handling rule. The main complication here is that you'll probably need to ensure that data is fed to the lexer in a way so that it doesn't split in in the middle of a token.

- The lexer should work properly with both Unicode strings given as token and pattern matching rules as well as for input text.

- If you need to supply optional flags to the re.compile() function, use the reflags option to lex. For example:

```
lex.lex(reflags=re.UNICODE | re.VERBOSE)
```

Note: by default, `reflags` is set to `re.VERBOSE`. If you provide your own flags, you may need to include this for PLY to preserve its normal behavior.

- Since the lexer is written entirely in Python, its performance is largely determined by that of the Python `re` module. Although the lexer has been written to be as efficient as possible, it's not blazingly fast when used on very large input files. If performance is concern, you might consider upgrading to the most recent version of Python, creating a hand-written lexer, or offloading the lexer into a C extension module.

If you are going to create a hand-written lexer and you plan to use it with `yacc.py`, it only needs to conform to the following requirements:

- It must provide a `token()` method that returns the next token or `None` if no more tokens are available.
- The `token()` method must return an object `tok` that has `type` and `value` attributes. If line number tracking is being used, then the token should also define a `lineno` attribute.

5. Parsing basics

`yacc.py` is used to parse language syntax. Before showing an example, there are a few important bits of background that must be mentioned. First, *syntax* is usually specified in terms of a BNF grammar. For example, if you wanted to parse simple arithmetic expressions, you might first write an unambiguous grammar specification like this:

```
expression : expression + term
           | expression - term
           | term

term       : term * factor
           | term / factor
           | factor

factor     : NUMBER
           | ( expression )
```

In the grammar, symbols such as `NUMBER`, `+`, `-`, `*`, and `/` are known as *terminals* and correspond to raw input tokens. Identifiers such as `term` and `factor` refer to grammar rules comprised of a collection of terminals and other rules. These identifiers are known as *non-terminals*.

The semantic behavior of a language is often specified using a technique known as syntax directed translation. In syntax directed translation, attributes are attached to each symbol in a given grammar rule along with an action. Whenever a particular grammar rule is recognized, the action describes what to do. For example, given the expression grammar above, you might write the specification for a simple calculator like this:

```
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
```

A good way to think about syntax directed translation is to view each symbol in the grammar as a kind of object. Associated with each symbol is a value representing its "state" (for example, the `val` attribute above). Semantic actions are then expressed as a collection of functions or methods that operate on the symbols and associated values.

Yacc uses a parsing technique known as LR-parsing or shift-reduce parsing. LR parsing is a bottom up technique that tries to recognize the right-hand-side of various grammar rules. Whenever a valid right-hand-side is found in the input, the appropriate action code is triggered and the grammar symbols are replaced by the grammar symbol on the left-hand-side.

LR parsing is commonly implemented by shifting grammar symbols onto a stack and looking at the stack and the next input token for patterns that match one of the grammar rules. The details of the algorithm can be found in a compiler textbook, but the following example illustrates the steps that are performed if you wanted to parse the expression `3 + 5 * (10 - 20)` using the grammar defined above. In the example, the special symbol `$` represents the end of input.

```
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!
```

When parsing the expression, an underlying state machine and the current input token determine what happens next. If the next token looks like part of a valid grammar rule (based on other items on the stack), it is generally shifted onto the stack. If the top of the stack contains a valid right-hand-side of a grammar rule, it is usually "reduced" and the symbols replaced with the symbol on the left-hand-side. When this reduction occurs, the appropriate action is triggered (if defined). If the input token can't be shifted and the top of stack doesn't match any grammar rules, a syntax error has occurred and the parser must take some kind of recovery step (or bail out). A parse is only successful if the parser reaches a state where the symbol stack is empty and there are no more input tokens.

It is important to note that the underlying implementation is built around a large finite-state machine that is encoded in a collection of tables. The construction of these tables is non-trivial and beyond the scope of this discussion. However, subtle details of this process explain why, in the example above, the parser chooses to shift a token onto the stack in step 9 rather than reducing the rule `expr : expr + term`.

6. Yacc

The `ply.yacc` module implements the parsing component of PLY. The name "yacc" stands for "Yet Another Compiler Compiler" and is borrowed from the Unix tool of the same name.

6.1 An example

Suppose you wanted to make a grammar for simple arithmetic expressions as previously described. Here is how you would do it with `yacc.py`:

```
# Yacc example

import ply.yacc as yacc

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

def p_expression_plus(p):
    'expression : expression PLUS term'
    p[0] = p[1] + p[3]

def p_expression_minus(p):
    'expression : expression MINUS term'
    p[0] = p[1] - p[3]

def p_expression_term(p):
    'expression : term'
    p[0] = p[1]

def p_term_times(p):
    'term : term TIMES factor'
    p[0] = p[1] * p[3]

def p_term_div(p):
    'term : term DIVIDE factor'
    p[0] = p[1] / p[3]

def p_term_factor(p):
    'term : factor'
    p[0] = p[1]

def p_factor_num(p):
    'factor : NUMBER'
    p[0] = p[1]

def p_factor_expr(p):
    'factor : LPAREN expression RPAREN'
    p[0] = p[2]

# Error rule for syntax errors
def p_error(p):
```

```
    print("Syntax error in input!")

# Build the parser
parser = yacc.yacc()

while True:
   try:
       s = raw_input('calc > ')
   except EOFError:
       break
   if not s: continue
   result = parser.parse(s)
   print(result)
```

In this example, each grammar rule is defined by a Python function where the docstring to that function contains the appropriate context-free grammar specification. The statements that make up the function body implement the semantic actions of the rule. Each function accepts a single argument `p` that is a sequence containing the values of each grammar symbol in the corresponding rule. The values of `p[i]` are mapped to grammar symbols as shown here:

```
def p_expression_plus(p):
    'expression : expression PLUS term'
    #   ^            ^        ^    ^
    #  p[0]         p[1]     p[2] p[3]

    p[0] = p[1] + p[3]
```

For tokens, the "value" of the corresponding `p[i]` is the *same* as the `p.value` attribute assigned in the lexer module. For non-terminals, the value is determined by whatever is placed in `p[0]` when rules are reduced. This value can be anything at all. However, it probably most common for the value to be a simple Python type, a tuple, or an instance. In this example, we are relying on the fact that the `NUMBER` token stores an integer value in its value field. All of the other rules simply perform various types of integer operations and propagate the result.

Note: The use of negative indices have a special meaning in yacc---specially `p[-1]` does not have the same value as `p[3]` in this example. Please see the section on "Embedded Actions" for further details.

The first rule defined in the yacc specification determines the starting grammar symbol (in this case, a rule for `expression` appears first). Whenever the starting rule is reduced by the parser and no more input is available, parsing stops and the final value is returned (this value will be whatever the top-most rule placed in `p[0]`). Note: an alternative starting symbol can be specified using the `start` keyword argument to `yacc()`.

The `p_error(p)` rule is defined to catch syntax errors. See the error handling section below for more detail.

To build the parser, call the `yacc.yacc()` function. This function looks at the module and attempts to construct all of the LR parsing tables for the grammar you have specified. The first time `yacc.yacc()` is invoked, you will get a message such as this:

```
$ python calcparse.py
Generating LALR tables
calc >
```

Since table construction is relatively expensive (especially for large grammars), the resulting parsing table is written to a file called `parsetab.py`. In addition, a debugging file called `parser.out` is created. On subsequent executions, `yacc` will reload the table from `parsetab.py` unless it has detected a change in the underlying grammar (in which case the tables and `parsetab.py` file are regenerated). Both of these files are written to the same directory as the module in which the parser is specified. The name of the `parsetab` module can be changed using the `tabmodule` keyword argument to `yacc()`. For example:

```
parser = yacc.yacc(tabmodule='fooparsetab')
```

If any errors are detected in your grammar specification, `yacc.py` will produce diagnostic messages and possibly raise an exception. Some of the errors that can be detected include:

- Duplicated function names (if more than one rule function have the same name in the grammar file).
- Shift/reduce and reduce/reduce conflicts generated by ambiguous grammars.
- Badly specified grammar rules.
- Infinite recursion (rules that can never terminate).
- Unused rules and tokens
- Undefined rules and tokens

The next few sections discuss grammar specification in more detail.

The final part of the example shows how to actually run the parser created by `yacc()`. To run the parser, you simply have to call the `parse()` with a string of input text. This will run all of the grammar rules and return the result of the entire parse. This result return is the value assigned to `p[0]` in the starting grammar rule.

6.2 Combining Grammar Rule Functions

When grammar rules are similar, they can be combined into a single function. For example, consider the two rules in our earlier example:

```
def p_expression_plus(p):
    'expression : expression PLUS term'
    p[0] = p[1] + p[3]

def p_expression_minus(t):
    'expression : expression MINUS term'
    p[0] = p[1] - p[3]
```

Instead of writing two functions, you might write a single function like this:

```
def p_expression(p):
    '''expression : expression PLUS term
                  | expression MINUS term'''
    if p[2] == '+':
        p[0] = p[1] + p[3]
    elif p[2] == '-':
        p[0] = p[1] - p[3]
```

In general, the doc string for any given function can contain multiple grammar rules. So, it would have also been legal (although possibly confusing) to write this:

```
def p_binary_operators(p):
    '''expression : expression PLUS term
                  | expression MINUS term
       term       : term TIMES factor
                  | term DIVIDE factor'''
    if p[2] == '+':
        p[0] = p[1] + p[3]
    elif p[2] == '-':
        p[0] = p[1] - p[3]
    elif p[2] == '*':
        p[0] = p[1] * p[3]
    elif p[2] == '/':
        p[0] = p[1] / p[3]
```

When combining grammar rules into a single function, it is usually a good idea for all of the rules to have a similar structure (e.g., the same number of terms). Otherwise, the corresponding action code may be more complicated than necessary. However, it is possible to handle simple cases using len(). For example:

```
def p_expressions(p):
    '''expression : expression MINUS expression
                  | MINUS expression'''
    if (len(p) == 4):
        p[0] = p[1] - p[3]
    elif (len(p) == 3):
        p[0] = -p[2]
```

If parsing performance is a concern, you should resist the urge to put too much conditional processing into a single grammar rule as shown in these examples. When you add checks to see which grammar rule is being handled, you are actually duplicating the work that the parser has already performed (i.e., the parser already knows exactly what rule it matched). You can eliminate this overhead by using a separate `p_rule()` function for each grammar rule.

6.3 Character Literals

If desired, a grammar may contain tokens defined as single character literals. For example:

```
def p_binary_operators(p):
    '''expression : expression '+' term
                  | expression '-' term
       term       : term '*' factor
                  | term '/' factor'''
    if p[2] == '+':
        p[0] = p[1] + p[3]
    elif p[2] == '-':
        p[0] = p[1] - p[3]
    elif p[2] == '*':
        p[0] = p[1] * p[3]
    elif p[2] == '/':
        p[0] = p[1] / p[3]
```

A character literal must be enclosed in quotes such as `'+'`. In addition, if literals are used, they must be declared in the corresponding `lex` file through the use of a special `literals` declaration.

```
# Literals.  Should be placed in module given to lex()
literals = ['+','-','*','/' ]
```

Character literals are limited to a single character. Thus, it is not legal to specify literals such as `'<='` or `'=='`. For this, use the normal lexing rules (e.g., define a rule such as `t_EQ = r'=='`).

6.4 Empty Productions

`yacc.py` can handle empty productions by defining a rule like this:

```
def p_empty(p):
    'empty :'
    pass
```

Now to use the empty production, simply use 'empty' as a symbol. For example:

```
def p_optitem(p):
    'optitem : item'
    '        | empty'
    ...
```

Note: You can write empty rules anywhere by simply specifying an empty right hand side. However, I personally find that writing an "empty" rule and using "empty" to denote an empty production is easier to read and more clearly states your intentions.

6.5 Changing the starting symbol

Normally, the first rule found in a yacc specification defines the starting grammar rule (top level rule). To change this, simply supply a `start` specifier in your file. For example:

```
start = 'foo'

def p_bar(p):
    'bar : A B'

# This is the starting rule due to the start specifier above
def p_foo(p):
    'foo : bar X'
...
```

The use of a `start` specifier may be useful during debugging since you can use it to have yacc build a subset of a larger grammar. For this purpose, it is also possible to specify a starting symbol as an argument to `yacc()`. For example:

```
parser = yacc.yacc(start='foo')
```

6.6 Dealing With Ambiguous Grammars

The expression grammar given in the earlier example has been written in a special format to eliminate ambiguity. However, in many situations, it is extremely difficult or awkward to write grammars in this format. A much more natural way to express the grammar is in a more compact form like this:

```
expression : expression PLUS expression
           | expression MINUS expression
           | expression TIMES expression
           | expression DIVIDE expression
           | LPAREN expression RPAREN
           | NUMBER
```

Unfortunately, this grammar specification is ambiguous. For example, if you are parsing the string "3 * 4 + 5", there is no way to tell how the operators are supposed to be grouped. For example, does the expression mean "(3 * 4) + 5" or is it "3 * (4+5)"?

When an ambiguous grammar is given to `yacc.py` it will print messages about "shift/reduce conflicts" or "reduce/reduce conflicts". A shift/reduce conflict is caused when the parser generator can't decide whether or not to reduce a rule or shift a symbol on the parsing stack. For example, consider the string "3 * 4 + 5" and the internal parsing stack:

```
Step Symbol Stack           Input Tokens            Action
---- --------------------  ---------------------   ------------------------------
1    $                             3 * 4 + 5$    Shift 3
2    $ 3                             * 4 + 5$    Reduce : expression : NUMBER
3    $ expr                          * 4 + 5$    Shift *
4    $ expr *                          4 + 5$    Shift 4
5    $ expr * 4                          + 5$    Reduce: expression : NUMBER
6    $ expr * expr                       + 5$    SHIFT/REDUCE CONFLICT ????
```

In this case, when the parser reaches step 6, it has two options. One is to reduce the rule `expr : expr * expr` on the stack. The other option is to shift the token + on the stack. Both options are perfectly legal from the rules of the context-free-grammar.

By default, all shift/reduce conflicts are resolved in favor of shifting. Therefore, in the above example, the parser will always shift the + instead of reducing. Although this strategy works in many cases (for example, the case of "if-then" versus "if-then-else"), it is not enough for arithmetic expressions. In fact, in the above example, the decision to shift + is completely wrong---we should have reduced `expr * expr` since multiplication has higher mathematical precedence than addition.

To resolve ambiguity, especially in expression grammars, `yacc.py` allows individual tokens to be assigned a precedence level and associativity. This is done by adding a variable `precedence` to the grammar file like this:

```
precedence = (
    ('left', 'PLUS', 'MINUS'),
    ('left', 'TIMES', 'DIVIDE'),
)
```

This declaration specifies that PLUS/MINUS have the same precedence level and are left-associative and that TIMES/DIVIDE have the same precedence and are left-associative. Within the precedence declaration, tokens are ordered from lowest to highest precedence. Thus, this declaration specifies that TIMES/DIVIDE have higher precedence than PLUS/MINUS (since they appear later in the precedence specification).

The precedence specification works by associating a numerical precedence level value and associativity direction to the listed tokens. For example, in the above example you get:

```
PLUS      : level = 1,  assoc = 'left'
MINUS     : level = 1,  assoc = 'left'
TIMES     : level = 2,  assoc = 'left'
DIVIDE    : level = 2,  assoc = 'left'
```

These values are then used to attach a numerical precedence value and associativity direction to each grammar rule. *This is always determined by looking at the precedence of the right-most terminal symbol.* For example:

```
expression : expression PLUS expression                 # level = 1, left
           | expression MINUS expression                # level = 1, left
           | expression TIMES expression                # level = 2, left
           | expression DIVIDE expression               # level = 2, left
           | LPAREN expression RPAREN                   # level = None (not
specified)
           | NUMBER                                     # level = None (not
specified)
```

When shift/reduce conflicts are encountered, the parser generator resolves the conflict by looking at the precedence rules and associativity specifiers.

1. If the current token has higher precedence than the rule on the stack, it is shifted.
2. If the grammar rule on the stack has higher precedence, the rule is reduced.
3. If the current token and the grammar rule have the same precedence, the rule is reduced for left associativity, whereas the token is shifted for right associativity.
4. If nothing is known about the precedence, shift/reduce conflicts are resolved in favor of shifting (the default).

For example, if "expression PLUS expression" has been parsed and the next token is "TIMES", the action is going to be a shift because "TIMES" has a higher precedence level than "PLUS". On the other hand, if "expression TIMES expression" has been parsed and the next token is "PLUS", the action is going to be reduce because "PLUS" has a lower precedence than "TIMES."

When shift/reduce conflicts are resolved using the first three techniques (with the help of precedence rules), `yacc.py` will report no errors or conflicts in the grammar (although it will print some information in the `parser.out` debugging file).

One problem with the precedence specifier technique is that it is sometimes necessary to change the precedence of an operator in certain contexts. For example, consider a unary-minus operator in "3 + 4 * -5". Mathematically, the unary minus is normally given a very high precedence--being evaluated before the multiply. However, in our precedence specifier, MINUS has a lower precedence than TIMES. To deal with this, precedence rules can be given for so-called "fictitious tokens" like this:

```
precedence = (
    ('left', 'PLUS', 'MINUS'),
    ('left', 'TIMES', 'DIVIDE'),
    ('right', 'UMINUS'),            # Unary minus operator
)
```

Now, in the grammar file, we can write our unary minus rule like this:

```
def p_expr_uminus(p):
    'expression : MINUS expression %prec UMINUS'
    p[0] = -p[2]
```

In this case, `%prec UMINUS` overrides the default rule precedence--setting it to that of UMINUS in the precedence specifier.

At first, the use of UMINUS in this example may appear very confusing. UMINUS is not an input token or a grammar rule. Instead, you should think of it as the name of a special marker in the precedence table. When you use the `%prec` qualifier, you're simply telling yacc that you want the precedence of the expression to be the same as for this special marker instead of the usual precedence.

It is also possible to specify non-associativity in the `precedence` table. This would be used when you *don't* want operations to chain together. For example, suppose you wanted to support comparison operators like < and > but you didn't want to allow combinations like `a < b < c`. To do this, simply specify a rule like this:

```
precedence = (
    ('nonassoc', 'LESSTHAN', 'GREATERTHAN'),  # Nonassociative operators
    ('left', 'PLUS', 'MINUS'),
    ('left', 'TIMES', 'DIVIDE'),
    ('right', 'UMINUS'),            # Unary minus operator
)
```

If you do this, the occurrence of input text such as `a < b < c` will result in a syntax error. However, simple expressions such as `a < b` will still be fine.

Reduce/reduce conflicts are caused when there are multiple grammar rules that can be applied to a given set of symbols. This kind of conflict is almost always bad and is always resolved by picking the rule that appears first in the grammar file. Reduce/reduce conflicts are almost always caused when different sets of grammar rules somehow generate the same set of symbols. For example:

```
assignment :  ID EQUALS NUMBER
           |  ID EQUALS expression

expression : expression PLUS expression
           | expression MINUS expression
           | expression TIMES expression
           | expression DIVIDE expression
           | LPAREN expression RPAREN
           | NUMBER
```

In this case, a reduce/reduce conflict exists between these two rules:

```
assignment  : ID EQUALS NUMBER
expression  : NUMBER
```

For example, if you wrote "a = 5", the parser can't figure out if this is supposed to be reduced as `assignment : ID EQUALS NUMBER` or whether it's supposed to reduce the 5 as an expression and then reduce the rule `assignment : ID EQUALS expression`.

It should be noted that reduce/reduce conflicts are notoriously difficult to spot simply looking at the input grammar. When a reduce/reduce conflict occurs, `yacc()` will try to help by printing a warning message such as this:

```
 WARNING: 1 reduce/reduce conflict
 WARNING: reduce/reduce conflict in state 15 resolved using rule (assignment -> ID
EQUALS NUMBER)
 WARNING: rejected rule (expression -> NUMBER)
```

This message identifies the two rules that are in conflict. However, it may not tell you how the parser arrived at such a state. To try and figure it out, you'll probably have to look at your grammar and the contents of the `parser.out` debugging file with an appropriately high level of caffeination.

6.7 The parser.out file

Tracking down shift/reduce and reduce/reduce conflicts is one of the finer pleasures of using an LR parsing algorithm. To assist in debugging, `yacc.py` creates a debugging file called 'parser.out' when it generates the parsing table. The contents of this file look like the following:

```
Unused terminals:


Grammar

Rule 1     expression -> expression PLUS expression
Rule 2     expression -> expression MINUS expression
Rule 3     expression -> expression TIMES expression
Rule 4     expression -> expression DIVIDE expression
Rule 5     expression -> NUMBER
Rule 6     expression -> LPAREN expression RPAREN

Terminals, with rules where they appear

TIMES                : 3
error                :
MINUS                : 2
RPAREN               : 6
LPAREN               : 6
DIVIDE               : 4
PLUS                 : 1
NUMBER               : 5

Nonterminals, with rules where they appear

expression           : 1 1 2 2 3 3 4 4 6 0


Parsing method: LALR
```

```
state 0

    S' -> . expression
    expression -> . expression PLUS expression
    expression -> . expression MINUS expression
    expression -> . expression TIMES expression
    expression -> . expression DIVIDE expression
    expression -> . NUMBER
    expression -> . LPAREN expression RPAREN

    NUMBER          shift and go to state 3
    LPAREN          shift and go to state 2


state 1

    S' -> expression .
    expression -> expression . PLUS expression
    expression -> expression . MINUS expression
    expression -> expression . TIMES expression
    expression -> expression . DIVIDE expression

    PLUS            shift and go to state 6
    MINUS           shift and go to state 5
    TIMES           shift and go to state 4
    DIVIDE          shift and go to state 7


state 2

    expression -> LPAREN . expression RPAREN
    expression -> . expression PLUS expression
    expression -> . expression MINUS expression
    expression -> . expression TIMES expression
    expression -> . expression DIVIDE expression
    expression -> . NUMBER
    expression -> . LPAREN expression RPAREN

    NUMBER          shift and go to state 3
    LPAREN          shift and go to state 2


state 3

    expression -> NUMBER .

    $               reduce using rule 5
    PLUS            reduce using rule 5
    MINUS           reduce using rule 5
```

```
    TIMES           reduce using rule 5
    DIVIDE          reduce using rule 5
    RPAREN          reduce using rule 5


state 4

    expression -> expression TIMES . expression
    expression -> . expression PLUS expression
    expression -> . expression MINUS expression
    expression -> . expression TIMES expression
    expression -> . expression DIVIDE expression
    expression -> . NUMBER
    expression -> . LPAREN expression RPAREN

    NUMBER          shift and go to state 3
    LPAREN          shift and go to state 2


state 5

    expression -> expression MINUS . expression
    expression -> . expression PLUS expression
    expression -> . expression MINUS expression
    expression -> . expression TIMES expression
    expression -> . expression DIVIDE expression
    expression -> . NUMBER
    expression -> . LPAREN expression RPAREN

    NUMBER          shift and go to state 3
    LPAREN          shift and go to state 2


state 6

    expression -> expression PLUS . expression
    expression -> . expression PLUS expression
    expression -> . expression MINUS expression
    expression -> . expression TIMES expression
    expression -> . expression DIVIDE expression
    expression -> . NUMBER
    expression -> . LPAREN expression RPAREN

    NUMBER          shift and go to state 3
    LPAREN          shift and go to state 2


state 7
```

```
    expression -> expression DIVIDE . expression
    expression -> . expression PLUS expression
    expression -> . expression MINUS expression
    expression -> . expression TIMES expression
    expression -> . expression DIVIDE expression
    expression -> . NUMBER
    expression -> . LPAREN expression RPAREN

    NUMBER          shift and go to state 3
    LPAREN          shift and go to state 2


state 8

    expression -> LPAREN expression . RPAREN
    expression -> expression . PLUS expression
    expression -> expression . MINUS expression
    expression -> expression . TIMES expression
    expression -> expression . DIVIDE expression

    RPAREN          shift and go to state 13
    PLUS            shift and go to state 6
    MINUS           shift and go to state 5
    TIMES           shift and go to state 4
    DIVIDE          shift and go to state 7


state 9

    expression -> expression TIMES expression .
    expression -> expression . PLUS expression
    expression -> expression . MINUS expression
    expression -> expression . TIMES expression
    expression -> expression . DIVIDE expression

    $               reduce using rule 3
    PLUS            reduce using rule 3
    MINUS           reduce using rule 3
    TIMES           reduce using rule 3
    DIVIDE          reduce using rule 3
    RPAREN          reduce using rule 3

  ! PLUS            [ shift and go to state 6 ]
  ! MINUS           [ shift and go to state 5 ]
  ! TIMES           [ shift and go to state 4 ]
  ! DIVIDE          [ shift and go to state 7 ]

state 10
```

```
    expression -> expression MINUS expression .
    expression -> expression . PLUS expression
    expression -> expression . MINUS expression
    expression -> expression . TIMES expression
    expression -> expression . DIVIDE expression

    $               reduce using rule 2
    PLUS            reduce using rule 2
    MINUS           reduce using rule 2
    RPAREN          reduce using rule 2
    TIMES           shift and go to state 4
    DIVIDE          shift and go to state 7

  ! TIMES           [ reduce using rule 2 ]
  ! DIVIDE          [ reduce using rule 2 ]
  ! PLUS            [ shift and go to state 6 ]
  ! MINUS           [ shift and go to state 5 ]

state 11

    expression -> expression PLUS expression .
    expression -> expression . PLUS expression
    expression -> expression . MINUS expression
    expression -> expression . TIMES expression
    expression -> expression . DIVIDE expression

    $               reduce using rule 1
    PLUS            reduce using rule 1
    MINUS           reduce using rule 1
    RPAREN          reduce using rule 1
    TIMES           shift and go to state 4
    DIVIDE          shift and go to state 7

  ! TIMES           [ reduce using rule 1 ]
  ! DIVIDE          [ reduce using rule 1 ]
  ! PLUS            [ shift and go to state 6 ]
  ! MINUS           [ shift and go to state 5 ]

state 12

    expression -> expression DIVIDE expression .
    expression -> expression . PLUS expression
    expression -> expression . MINUS expression
    expression -> expression . TIMES expression
    expression -> expression . DIVIDE expression

    $               reduce using rule 4
    PLUS            reduce using rule 4
    MINUS           reduce using rule 4
```

```
    TIMES           reduce using rule 4
    DIVIDE          reduce using rule 4
    RPAREN          reduce using rule 4

  ! PLUS            [ shift and go to state 6 ]
  ! MINUS           [ shift and go to state 5 ]
  ! TIMES           [ shift and go to state 4 ]
  ! DIVIDE          [ shift and go to state 7 ]

state 13

    expression -> LPAREN expression RPAREN .

    $               reduce using rule 6
    PLUS            reduce using rule 6
    MINUS           reduce using rule 6
    TIMES           reduce using rule 6
    DIVIDE          reduce using rule 6
    RPAREN          reduce using rule 6
```

The different states that appear in this file are a representation of every possible sequence of valid input tokens allowed by the grammar. When receiving input tokens, the parser is building up a stack and looking for matching rules. Each state keeps track of the grammar rules that might be in the process of being matched at that point. Within each rule, the "." character indicates the current location of the parse within that rule. In addition, the actions for each valid input token are listed. When a shift/reduce or reduce/reduce conflict arises, rules *not* selected are prefixed with an !. For example:

```
  ! TIMES           [ reduce using rule 2 ]
  ! DIVIDE          [ reduce using rule 2 ]
  ! PLUS            [ shift and go to state 6 ]
  ! MINUS           [ shift and go to state 5 ]
```

By looking at these rules (and with a little practice), you can usually track down the source of most parsing conflicts. It should also be stressed that not all shift-reduce conflicts are bad. However, the only way to be sure that they are resolved correctly is to look at `parser.out`.

6.8 Syntax Error Handling

If you are creating a parser for production use, the handling of syntax errors is important. As a general rule, you don't want a parser to simply throw up its hands and stop at the first sign of trouble. Instead, you want it to report the error, recover if possible, and continue parsing so that all of the errors in the input get reported to the user at once. This is the standard behavior found in compilers for languages such as C, C++, and Java.

In PLY, when a syntax error occurs during parsing, the error is immediately detected (i.e., the parser does not read any more tokens beyond the source of the error). However, at this point, the parser enters a recovery mode that can be used to try and continue further parsing. As a general rule, error recovery in LR parsers is a delicate topic that involves ancient rituals and black-magic. The recovery mechanism provided by `yacc.py` is comparable to Unix yacc so you may want consult a book like O'Reilly's "Lex and Yacc" for some of the finer details.

When a syntax error occurs, `yacc.py` performs the following steps:

1. On the first occurrence of an error, the user-defined `p_error()` function is called with the offending token as an argument. However, if the syntax error is due to reaching the end-of-file, `p_error()` is called with an argument of `None`. Afterwards, the parser enters an "error-recovery" mode in which it will not make future calls to `p_error()` until it has successfully shifted at least 3 tokens onto the parsing stack.

2. If no recovery action is taken in `p_error()`, the offending lookahead token is replaced with a special `error` token.

3. If the offending lookahead token is already set to `error`, the top item of the parsing stack is deleted.

4. If the entire parsing stack is unwound, the parser enters a restart state and attempts to start parsing from its initial state.

5. If a grammar rule accepts `error` as a token, it will be shifted onto the parsing stack.

6. If the top item of the parsing stack is `error`, lookahead tokens will be discarded until the parser can successfully shift a new symbol or reduce a rule involving `error`.

6.8.1 Recovery and resynchronization with error rules

The most well-behaved approach for handling syntax errors is to write grammar rules that include the `error` token. For example, suppose your language had a grammar rule for a print statement like this:

```
def p_statement_print(p):
     'statement : PRINT expr SEMI'
     ...
```

To account for the possibility of a bad expression, you might write an additional grammar rule like this:

```
def p_statement_print_error(p):
     'statement : PRINT error SEMI'
     print("Syntax error in print statement. Bad expression")
```

In this case, the `error` token will match any sequence of tokens that might appear up to the first semicolon that is encountered. Once the semicolon is reached, the rule will be invoked and the `error` token will go away.

This type of recovery is sometimes known as parser resynchronization. The `error` token acts as a wildcard for any bad input text and the token immediately following `error` acts as a synchronization token.

It is important to note that the `error` token usually does not appear as the last token on the right in an error rule. For example:

```
def p_statement_print_error(p):
    'statement : PRINT error'
    print("Syntax error in print statement. Bad expression")
```

This is because the first bad token encountered will cause the rule to be reduced--which may make it difficult to recover if more bad tokens immediately follow.

6.8.2 Panic mode recovery

An alternative error recovery scheme is to enter a panic mode recovery in which tokens are discarded to a point where the parser might be able to recover in some sensible manner.

Panic mode recovery is implemented entirely in the `p_error()` function. For example, this function starts discarding tokens until it reaches a closing '}'. Then, it restarts the parser in its initial state.

```
def p_error(p):
    print("Whoa. You are seriously hosed.")
    if not p:
        print("End of File!")
        return

    # Read ahead looking for a closing '}'
    while True:
        tok = parser.token()             # Get the next token
        if not tok or tok.type == 'RBRACE':
            break
    parser.restart()
```

This function simply discards the bad token and tells the parser that the error was ok.

```
def p_error(p):
    if p:
         print("Syntax error at token", p.type)
         # Just discard the token and tell the parser it's okay.
         parser.errok()
    else:
         print("Syntax error at EOF")
```

More information on these methods is as follows:

- `parser.errok()`. This resets the parser state so it doesn't think it's in error-recovery mode. This will prevent an `error` token from being generated and will reset the internal error counters so that the next syntax error will call `p_error()` again.
- `parser.token()`. This returns the next token on the input stream.
- `parser.restart()`. This discards the entire parsing stack and resets the parser to its initial state.

To supply the next lookahead token to the parser, `p_error()` can return a token. This might be useful if trying to synchronize on special characters. For example:

```
def p_error(p):
    # Read ahead looking for a terminating ";"
    while True:
        tok = parser.token()             # Get the next token
        if not tok or tok.type == 'SEMI': break
    parser.errok()

    # Return SEMI to the parser as the next lookahead token
    return tok
```

Keep in mind in that the above error handling functions, `parser` is an instance of the parser created by `yacc()`. You'll need to save this instance someplace in your code so that you can refer to it during error handling.

6.8.3 Signalling an error from a production

If necessary, a production rule can manually force the parser to enter error recovery. This is done by raising the `SyntaxError` exception like this:

```
def p_production(p):
    'production : some production ...'
    raise SyntaxError
```

The effect of raising `SyntaxError` is the same as if the last symbol shifted onto the parsing stack was actually a syntax error. Thus, when you do this, the last symbol shifted is popped off of the parsing stack and the current lookahead token is set to an `error` token. The parser then enters error-recovery mode where it tries to reduce rules that can accept `error` tokens. The steps that follow from this point are exactly the same as if a syntax error were detected and `p_error()` were called.

One important aspect of manually setting an error is that the `p_error()` function will **NOT** be called in this case. If you need to issue an error message, make sure you do it in the production that raises `SyntaxError`.

Note: This feature of PLY is meant to mimic the behavior of the YYERROR macro in yacc.

6.8.4 When Do Syntax Errors Get Reported

In most cases, yacc will handle errors as soon as a bad input token is detected on the input. However, be aware that yacc may choose to delay error handling until after it has reduced one or more grammar rules first. This behavior might be unexpected, but it's related to special states in the underlying parsing table known as "defaulted states." A defaulted state is parsing condition where the same grammar rule will be reduced regardless of what *valid* token comes next on the input. For such states, yacc chooses to go ahead and reduce the grammar rule *without reading the next input token*. If the next token is bad, yacc will eventually get around to reading it and report a syntax error. It's just a little unusual in that you might see some of your grammar rules firing immediately prior to the syntax error.

Usually, the delayed error reporting with defaulted states is harmless (and there are other reasons for wanting PLY to behave in this way). However, if you need to turn this behavior off for some reason. You can clear the defaulted states table like this:

```
parser = yacc.yacc()
parser.defaulted_states = {}
```

Disabling defaulted states is not recommended if your grammar makes use of embedded actions as described in Section 6.11.

6.8.5 General comments on error handling

For normal types of languages, error recovery with error rules and resynchronization characters is probably the most reliable technique. This is because you can instrument the grammar to catch errors at selected places where it is relatively easy to recover and continue parsing. Panic mode

recovery is really only useful in certain specialized applications where you might want to discard huge portions of the input text to find a valid restart point.

6.9 Line Number and Position Tracking

Position tracking is often a tricky problem when writing compilers. By default, PLY tracks the line number and position of all tokens. This information is available using the following functions:

- `p.lineno(num)`. Return the line number for symbol *num*
- `p.lexpos(num)`. Return the lexing position for symbol *num*

For example:

```
def p_expression(p):
    'expression : expression PLUS expression'
    line   = p.lineno(2)        # line number of the PLUS token
    index  = p.lexpos(2)        # Position of the PLUS token
```

As an optional feature, `yacc.py` can automatically track line numbers and positions for all of the grammar symbols as well. However, this extra tracking requires extra processing and can significantly slow down parsing. Therefore, it must be enabled by passing the `tracking=True` option to `yacc.parse()`. For example:

```
yacc.parse(data,tracking=True)
```

Once enabled, the `lineno()` and `lexpos()` methods work for all grammar symbols. In addition, two additional methods can be used:

- `p.linespan(num)`. Return a tuple (startline,endline) with the starting and ending line number for symbol *num*.
- `p.lexspan(num)`. Return a tuple (start,end) with the starting and ending positions for symbol *num*.

For example:

```
def p_expression(p):
    'expression : expression PLUS expression'
    p.lineno(1)        # Line number of the left expression
    p.lineno(2)        # line number of the PLUS operator
    p.lineno(3)        # line number of the right expression
    ...
    start,end = p.linespan(3)    # Start,end lines of the right expression
    starti,endi = p.lexspan(3)   # Start,end positions of right expression
```

Note: The `lexspan()` function only returns the range of values up to the start of the last grammar symbol.

Although it may be convenient for PLY to track position information on all grammar symbols, this is often unnecessary. For example, if you are merely using line number information in an error message, you can often just key off of a specific token in the grammar rule. For example:

```
def p_bad_func(p):
    'funccall : fname LPAREN error RPAREN'
    # Line number reported from LPAREN token
    print("Bad function call at line", p.lineno(2))
```

Similarly, you may get better parsing performance if you only selectively propagate line number information where it's needed using the `p.set_lineno()` method. For example:

```
def p_fname(p):
    'fname : ID'
    p[0] = p[1]
    p.set_lineno(0,p.lineno(1))
```

PLY doesn't retain line number information from rules that have already been parsed. If you are building an abstract syntax tree and need to have line numbers, you should make sure that the line numbers appear in the tree itself.

6.10 AST Construction

`yacc.py` provides no special functions for constructing an abstract syntax tree. However, such construction is easy enough to do on your own.

A minimal way to construct a tree is to simply create and propagate a tuple or list in each grammar rule function. There are many possible ways to do this, but one example would be something like this:

```
def p_expression_binop(p):
    '''expression : expression PLUS expression
                  | expression MINUS expression
                  | expression TIMES expression
                  | expression DIVIDE expression'''

    p[0] = ('binary-expression',p[2],p[1],p[3])

def p_expression_group(p):
    'expression : LPAREN expression RPAREN'
    p[0] = ('group-expression',p[2])

def p_expression_number(p):
    'expression : NUMBER'
    p[0] = ('number-expression',p[1])
```

Another approach is to create a set of data structure for different kinds of abstract syntax tree nodes and assign nodes to `p[0]` in each rule. For example:

```
class Expr: pass

class BinOp(Expr):
    def __init__(self,left,op,right):
        self.type = "binop"
        self.left = left
        self.right = right
        self.op = op

class Number(Expr):
    def __init__(self,value):
        self.type = "number"
        self.value = value

def p_expression_binop(p):
    '''expression : expression PLUS expression
                  | expression MINUS expression
                  | expression TIMES expression
                  | expression DIVIDE expression'''

    p[0] = BinOp(p[1],p[2],p[3])

def p_expression_group(p):
    'expression : LPAREN expression RPAREN'
    p[0] = p[2]

def p_expression_number(p):
    'expression : NUMBER'
    p[0] = Number(p[1])
```

The advantage to this approach is that it may make it easier to attach more complicated semantics, type checking, code generation, and other features to the node classes.

To simplify tree traversal, it may make sense to pick a very generic tree structure for your parse tree nodes. For example:

```
class Node:
    def __init__(self,type,children=None,leaf=None):
         self.type = type
         if children:
              self.children = children
         else:
              self.children = [ ]
         self.leaf = leaf

def p_expression_binop(p):
    '''expression : expression PLUS expression
                  | expression MINUS expression
                  | expression TIMES expression
                  | expression DIVIDE expression'''

    p[0] = Node("binop", [p[1],p[3]], p[2])
```

6.11 Embedded Actions

The parsing technique used by yacc only allows actions to be executed at the end of a rule. For example, suppose you have a rule like this:

```
def p_foo(p):
    "foo : A B C D"
    print("Parsed a foo", p[1],p[2],p[3],p[4])
```

In this case, the supplied action code only executes after all of the symbols `A`, `B`, `C`, and `D` have been parsed. Sometimes, however, it is useful to execute small code fragments during intermediate stages of parsing. For example, suppose you wanted to perform some action immediately after `A` has been parsed. To do this, write an empty rule like this:

```
def p_foo(p):
    "foo : A seen_A B C D"
    print("Parsed a foo", p[1],p[3],p[4],p[5])
    print("seen_A returned", p[2])

def p_seen_A(p):
    "seen_A :"
    print("Saw an A = ", p[-1])   # Access grammar symbol to left
    p[0] = some_value            # Assign value to seen_A
```

In this example, the empty `seen_A` rule executes immediately after `A` is shifted onto the parsing stack. Within this rule, `p[-1]` refers to the symbol on the stack that appears immediately to the left of the `seen_A` symbol. In this case, it would be the value of `A` in the `foo` rule immediately above. Like other rules, a value can be returned from an embedded action by simply assigning it to `p[0]`

The use of embedded actions can sometimes introduce extra shift/reduce conflicts. For example, this grammar has no conflicts:

```
def p_foo(p):
    """foo : abcd
           | abcx"""

def p_abcd(p):
    "abcd : A B C D"

def p_abcx(p):
    "abcx : A B C X"
```

However, if you insert an embedded action into one of the rules like this,

```
def p_foo(p):
    """foo : abcd
           | abcx"""

def p_abcd(p):
    "abcd : A B C D"

def p_abcx(p):
    "abcx : A B seen_AB C X"

def p_seen_AB(p):
    "seen_AB :"
```

an extra shift-reduce conflict will be introduced. This conflict is caused by the fact that the same symbol `C` appears next in both the `abcd` and `abcx` rules. The parser can either shift the symbol (`abcd` rule) or reduce the empty rule `seen_AB` (`abcx` rule).

A common use of embedded rules is to control other aspects of parsing such as scoping of local variables. For example, if you were parsing C code, you might write code like this:

```
def p_statements_block(p):
    "statements: LBRACE new_scope statements RBRACE"""
    # Action code
    ...
    pop_scope()        # Return to previous scope

def p_new_scope(p):
    "new_scope :"
    # Create a new scope for local variables
    s = new_scope()
    push_scope(s)
    ...
```

In this case, the embedded action `new_scope` executes immediately after a `LBRACE` ({) symbol is parsed. This might adjust internal symbol tables and other aspects of the parser. Upon completion of the rule `statements_block`, code might undo the operations performed in the embedded action (e.g., `pop_scope()`).

6.12 Miscellaneous Yacc Notes

- By default, `yacc.py` relies on `lex.py` for tokenizing. However, an alternative tokenizer can be supplied as follows:

  ```
  parser = yacc.parse(lexer=x)
  ```

  in this case, `x` must be a Lexer object that minimally has a `x.token()` method for retrieving the next token. If an input string is given to `yacc.parse()`, the lexer must also have an `x.input()` method.

- By default, the yacc generates tables in debugging mode (which produces the parser.out file and other output). To disable this, use

  ```
  parser = yacc.yacc(debug=False)
  ```

- To change the name of the `parsetab.py` file, use:

  ```
  parser = yacc.yacc(tabmodule="foo")
  ```

  Normally, the `parsetab.py` file is placed into the same directory as the module where the parser is defined. If you want it to go somewhere else, you can given an absolute package name for `tabmodule` instead. In that case, the tables will be written there.

- To change the directory in which the `parsetab.py` file (and other output files) are written, use:

```
parser = yacc.yacc(tabmodule="foo",outputdir="somedirectory")
```

Note: Be aware that unless the directory specified is also on Python's path (`sys.path`), subsequent imports of the table file will fail. As a general rule, it's better to specify a destination using the `tabmodule` argument instead of directly specifying a directory using the `outputdir` argument.

- To prevent yacc from generating any kind of parser table file, use:

```
parser = yacc.yacc(write_tables=False)
```

Note: If you disable table generation, yacc() will regenerate the parsing tables each time it runs (which may take awhile depending on how large your grammar is).

- To print copious amounts of debugging during parsing, use:

```
parser.parse(input_text, debug=True)
```

- Since the generation of the LALR tables is relatively expensive, previously generated tables are cached and reused if possible. The decision to regenerate the tables is determined by taking an MD5 checksum of all grammar rules and precedence rules. Only in the event of a mismatch are the tables regenerated.

  It should be noted that table generation is reasonably efficient, even for grammars that involve around a 100 rules and several hundred states.

- Since LR parsing is driven by tables, the performance of the parser is largely independent of the size of the grammar. The biggest bottlenecks will be the lexer and the complexity of the code in your grammar rules.

- `yacc()` also allows parsers to be defined as classes and as closures (see the section on alternative specification of lexers). However, be aware that only one parser may be defined in a single module (source file). There are various error checks and validation steps that may issue confusing error messages if you try to define multiple parsers in the same source file.

- Decorators of production rules have to update the wrapped function's line number. `wrapper.co_firstlineno = func.__code__.co_firstlineno`:

```
from functools import wraps
from nodes import Collection


def strict(*types):
    def decorate(func):
        @wraps(func)
        def wrapper(p):
            func(p)
            if not isinstance(p[0], types):
                raise TypeError

        wrapper.co_firstlineno = func.__code__.co_firstlineno
        return wrapper

    return decorate

@strict(Collection)
def p_collection(p):
    """
    collection  : sequence
                | map
    """
    p[0] = p[1]
```

7. Multiple Parsers and Lexers

In advanced parsing applications, you may want to have multiple parsers and lexers.

As a general rules this isn't a problem. However, to make it work, you need to carefully make sure everything gets hooked up correctly. First, make sure you save the objects returned by `lex()` and `yacc()`. For example:

```
lexer  = lex.lex()       # Return lexer object
parser = yacc.yacc()     # Return parser object
```

Next, when parsing, make sure you give the `parse()` function a reference to the lexer it should be using. For example:

```
parser.parse(text,lexer=lexer)
```

If you forget to do this, the parser will use the last lexer created--which is not always what you want.

Within lexer and parser rule functions, these objects are also available. In the lexer, the "lexer" attribute of a token refers to the lexer object that triggered the rule. For example:

```
def t_NUMBER(t):
   r'\d+'
   ...
   print(t.lexer)           # Show lexer object
```

In the parser, the "lexer" and "parser" attributes refer to the lexer and parser objects respectively.

```
def p_expr_plus(p):
   'expr : expr PLUS expr'
   ...
   print(p.parser)          # Show parser object
   print(p.lexer)           # Show lexer object
```

If necessary, arbitrary attributes can be attached to the lexer or parser object. For example, if you wanted to have different parsing modes, you could attach a mode attribute to the parser object and look at it later.

8. Using Python's Optimized Mode

Because PLY uses information from doc-strings, parsing and lexing information must be gathered while running the Python interpreter in normal mode (i.e., not with the -O or -OO options). However, if you specify optimized mode like this:

```
lex.lex(optimize=1)
yacc.yacc(optimize=1)
```

then PLY can later be used when Python runs in optimized mode. To make this work, make sure you first run Python in normal mode. Once the lexing and parsing tables have been generated the first time, run Python in optimized mode. PLY will use the tables without the need for doc strings.

Beware: running PLY in optimized mode disables a lot of error checking. You should only do this when your project has stabilized and you don't need to do any debugging. One of the purposes of optimized mode is to substantially decrease the startup time of your compiler (by assuming that everything is already properly specified and works).

9. Advanced Debugging

Debugging a compiler is typically not an easy task. PLY provides some advanced diagostic capabilities through the use of Python's `logging` module. The next two sections describe this:

9.1 Debugging the lex() and yacc() commands

Both the `lex()` and `yacc()` commands have a debugging mode that can be enabled using the `debug` flag. For example:

```
lex.lex(debug=True)
yacc.yacc(debug=True)
```

Normally, the output produced by debugging is routed to either standard error or, in the case of `yacc()`, to a file `parser.out`. This output can be more carefully controlled by supplying a logging object. Here is an example that adds information about where different debugging messages are coming from:

```
# Set up a logging object
import logging
logging.basicConfig(
    level = logging.DEBUG,
    filename = "parselog.txt",
    filemode = "w",
    format = "%(filename)10s:%(lineno)4d:%(message)s"
)
log = logging.getLogger()

lex.lex(debug=True,debuglog=log)
yacc.yacc(debug=True,debuglog=log)
```

If you supply a custom logger, the amount of debugging information produced can be controlled by setting the logging level. Typically, debugging messages are either issued at the `DEBUG`, `INFO`, or `WARNING` levels.

PLY's error messages and warnings are also produced using the logging interface. This can be controlled by passing a logging object using the `errorlog` parameter.

```
lex.lex(errorlog=log)
yacc.yacc(errorlog=log)
```

If you want to completely silence warnings, you can either pass in a logging object with an appropriate filter level or use the `NullLogger` object defined in either `lex` or `yacc`. For example:

```
yacc.yacc(errorlog=yacc.NullLogger())
```

9.2 Run-time Debugging

To enable run-time debugging of a parser, use the `debug` option to parse. This option can either be an integer (which simply turns debugging on or off) or an instance of a logger object. For example:

```
log = logging.getLogger()
parser.parse(input,debug=log)
```

If a logging object is passed, you can use its filtering level to control how much output gets generated. The `INFO` level is used to produce information about rule reductions. The `DEBUG` level will show information about the parsing stack, token shifts, and other details. The `ERROR` level shows information related to parsing errors.

For very complicated problems, you should pass in a logging object that redirects to a file where you can more easily inspect the output after execution.

10. Packaging Advice

If you are distributing a package that makes use of PLY, you should spend a few moments thinking about how you want to handle the files that are automatically generated. For example, the `parsetab.py` file generated by the `yacc()` function.

Starting in PLY-3.6, the table files are created in the same directory as the file where a parser is defined. This means that the `parsetab.py` file will live side-by-side with your parser specification. In terms of packaging, this is probably the easiest and most sane approach to manage. You don't need to give `yacc()` any extra arguments and it should just "work."

One concern is the management of the `parsetab.py` file itself. For example, should you have this file checked into version control (e.g., GitHub), should it be included in a package distribution as a normal file, or should you just let PLY generate it automatically for the user when they install your package?

As of PLY-3.6, the `parsetab.py` file should be compatible across all versions of Python including Python 2 and 3. Thus, a table file generated in Python 2 should work fine if it's used on Python 3. Because of this, it should be relatively harmless to distribute the `parsetab.py` file yourself if you need to. However, be aware that older/newer versions of PLY may try to regenerate the file if there are future enhancements or changes to its format.

To make the generation of table files easier for the purposes of installation, you might way to make your parser files executable using the `-m` option or similar. For example:

```
# calc.py
...
...
def make_parser():
    parser = yacc.yacc()
    return parser

if __name__ == '__main__':
    make_parser()
```

You can then use a command such as `python -m calc.py` to generate the tables. Alternatively, a `setup.py` script, can import the module and use `make_parser()` to create the parsing tables.

If you're willing to sacrifice a little startup time, you can also instruct PLY to never write the tables using `yacc.yacc(write_tables=False, debug=False)`. In this mode, PLY will regenerate the parsing tables from scratch each time. For a small grammar, you probably won't notice. For a large grammar, you should probably reconsider--the parsing tables are meant to dramatically speed up this process.

During operation, is is normal for PLY to produce diagnostic error messages (usually printed to standard error). These are generated entirely using the `logging` module. If you want to redirect these messages or silence them, you can provide your own logging object to `yacc()`. For example:

```
import logging
log = logging.getLogger('ply')
...
parser = yacc.yacc(errorlog=log)
```

11. Where to go from here?

The `examples` directory of the PLY distribution contains several simple examples. Please consult a compilers textbook for the theory and underlying implementation details or LR parsing.