PLY (Python Lex-Yacc)

David M. Beazley
Department of Computer Science
University of Chicago
Chicago, IL 60637
beazley@cs.uchicago.edu

Documentation version: $Header: /cvs/projects/PLY/doc/ply.html,v 1.3 2004/05/27 17:30:56 beazley Exp $

PLY Version: 1.5

Introduction

PLY is a Python-only implementation of the popular compiler construction tools lex and yacc. The implementation borrows ideas from a number of previous efforts; most notably John Aycock's SPARK toolkit. However, the overall flavor of the implementation is more closely modeled after the C version of lex and yacc. The other significant feature of PLY is that it provides extensive input validation and error reporting--much more so than other Python parsing tools.

Early versions of PLY were developed to support the Introduction to Compilers Course at the University of Chicago. In this course, students built a fully functional compiler for a simple Pascal-like language. Their compiler, implemented entirely in Python, had to include lexical analysis, parsing, type checking, type inference, nested scoping, and code generation for the SPARC processor. Approximately 30 different compiler implementations were completed in this course. Most of PLY's interface and operation has been motivated by common usability problems encountered by students.

Because PLY was primarily developed as an instructional tool, you will find it to be *MUCH* more picky about token and grammar rule specification than most other Python parsing tools. 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.). 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 familar with parsing theory, syntax directed translation, and automatic tools such as lex and yacc. If you are unfamilar with these topics, you will probably want to consult an introductory text such as "Compilers: Principles, Techniques, and Tools", by Aho, Sethi, and Ullman. "Lex and Yacc" by John Levine may also be handy.

PLY Overview

PLY consists of two separate tools; `lex.py` and `yacc.py`. `lex.py` 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. Currently, `yacc.py` uses LR parsing and generates its parsing tables using either the SLR (the default) or LALR(1) table generation algorithms.

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. The primary difference between `yacc.py` and Unix `yacc` is that `yacc.py` uses the SLR

parsing algorithm instead of LALR(1) as the default. However, LALR(1) support can also be used with PLY if you enable it as an option.

Finally, it is important to note that 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).

Lex Example

`lex.py` is used to write tokenizers. To do this, each token must be defined by a regular expression rule. The following file implements a very simple lexer for tokenizing simple integer expressions:

```
# ------------------------------------------------------------
# calclex.py
#
# tokenizer for a simple expression evaluator for
# numbers and +,-,*,/
# ------------------------------------------------------------
import 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+'
    try:
         t.value = int(t.value)
    except ValueError:
         print "Line %d: Number %s is too large!" % (t.lineno,t.value)
         t.value = 0
    return t

# Define a rule so we can track line numbers
def t_newline(t):
    r'\n+'
    t.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.skip(1)

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

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

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

# Tokenize
while 1:
    tok = lex.token()
    if not tok: break      # No more input
    print tok
```

In the example, the `tokens` list 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. Following the `tokens` list, regular expressions are written for each token. 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:

```
def t_NUMBER(t):
    r'\d+'
    try:
         t.value = int(t.value)
    except ValueError:
         print "Number %s is too large!" % t.value
         t.value = 0
    return t
```

In this case, 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, `t.value` which is the lexeme, and `t.lineno` which is the current line number. 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.

The rule `t_newline()` illustrates a regular expression rule for a discarded token. In this case, a rule is written to match newlines so that proper line number tracking can be performed. By returning no value, the function causes the newline character to be discarded.

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.

Finally, 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, we simply print the offending character and skip ahead one character by calling `t.skip(1)`.

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

- `lex.input(data)`. Reset the lexer and store a new input string.

- `lex.token()`. Return the next token. Returns a special `LexToken` instance on success or None if the end of the input text has been reached.

The code at the bottom of the example shows how the lexer is actually used. When executed, the following output will be produced:

```
$ python example.py
LexToken(NUMBER,3,2)
LexToken(PLUS,'+',2)
LexToken(NUMBER,4,2)
LexToken(TIMES,'*',2)
LexToken(NUMBER,10,2)
LexToken(PLUS,'+',3)
LexToken(MINUS,'-',3)
LexToken(NUMBER,20,3)
LexToken(TIMES,'*',3)
LexToken(NUMBER,2,3)
```

Lex Implementation Notes

- `lex.py` uses the `re` module to do its patten matching. 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 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.

- 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.

- To handle reserved words, it is usually easier to just match an identifier and do a special name lookup in a function like this:

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

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
```

- The lexer requires tokens to be defined as class instances with `t.type`, `t.value`, and `t.lineno` attributes. By default, tokens are created as instances of the `LexToken` class defined internally to `lex.py`. If desired, you can create new kinds of tokens provided that they have the three required attributes. However, in practice, it is probably safer to stick with the default.

- The only safe attribute for assigning token properties is `t.value`. In some cases, you may want to attach a number of different properties to a token (e.g., symbol table entries for identifiers). To do this, replace `t.value` with a tuple or class instance. For example:

```
def t_ID(t):
    ...
    # For identifiers, create a (lexeme, symtab) tuple
    t.value = (t.value, symbol_lookup(t.value))
    ...
    return t
```

Although allowed, do NOT assign additional attributes to the token object. For example,

```
def t_ID(t):
    ...
    # Bad implementation of above
    t.symtab = symbol_lookup(t.value)
    ...
```

The reason you don't want to do this is that the `yacc.py` module only provides public access to the `t.value` attribute of each token. Therefore, any other attributes you assign are inaccessible (if you are familiar with the internals of C lex/yacc, `t.value` is the same as `yylval.tok`).

- To track line numbers, the lexer internally maintains a line number variable. Each token automatically gets the value of the current line number in the `t.lineno` attribute. To modify the current line number, simply change the `t.lineno` attribute in a function rule (as previously shown for `t_newline()`). Even if the resulting token is discarded, changes to the line number remain in effect for subsequent tokens.

- To support multiple scanners in the same application, the `lex.lex()` function actually returns a special `Lexer` object. This object has two methods `input()` and `token()` that can be used to supply input and get tokens. For example:

```
lexer = lex.lex()
lexer.input(sometext)
while 1:
    tok = lexer.token()
    if not tok: break
    print tok
```

The functions `lex.input()` and `lex.token()` are bound to the `input()` and `token()` methods of the last lexer created by the lex module.

- To reduce compiler startup time and improve performance, the lexer can be built in optimized mode as follows:

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

When used, most error checking and validation is disabled. This provides a slight performance gain while tokenizing and tends to chop a few tenths of a second off startup time. Since it disables error checking, this mode is not the default and is not recommended during development. However, once you have your compiler fully working, it is usually safe to disable the error checks.

- You can enable some additional debugging by building the lexer like this:

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

- To help you debug your lexer, `lex.py` comes with a simple main program which will either tokenize input read from standard input or from a file. To use it, simply put this in your lexer:

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

Then, run you lexer as a main program such as `python mylex.py`

- 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. Sorry. 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. In defense of `lex.py`, it's performance is not *that* bad when used on reasonably sized input files. For instance, lexing a 4700 line C program with 32000 input tokens takes about 20 seconds on a 200 Mhz PC. Obviously, it will run much faster on a more speedy machine.

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 context free grammar (CFG). 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 )
```

Next, 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
```

Finally, 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. The details of the algorithm can be found in a compiler text, 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:

```
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 to do 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).

It is important to note that the underlying implementation is actually built around a large finite-state machine and some tables. The construction of these tables is quite complicated 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`.

Yacc 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 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
    yacc.yacc()

    # Use this if you want to build the parser using LALR(1) instead of SLR
    # yacc.yacc(method="LALR")

    while 1:
       try:
           s = raw_input('calc > ')
       except EOFError:
           break
       if not s: continue
       result = yacc.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 (an idea borrowed from John Aycock's SPARK toolkit). 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" in the corresponding `p[i]` is the *same* as the value of 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 store the result.

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]`).

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
yacc: Generating SLR parsing table...
calc >
```

Since table construction is relatively expensive (especially for large grammars), the resulting parsing table is written to the current directory in 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).

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 now discuss a few finer points of grammar construction.

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]
```

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'
    ...
```

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 this 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 a "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 the 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 (including the ambiguous 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 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. Furthermore, the declaration specifies that `TIMES`/`DIVIDE` have higher precedence than `PLUS`/`MINUS` (since they appear later in the precedence specification).

The precedence specification is used to attach a numerical precedence value and associativity direction to each grammar rule. This is always determined by the precedence of the right-most terminal symbol. Therefore, if PLUS/MINUS had a precedence of 1 and TIMES/DIVIDE had a precedence of 2, the grammar rules would have precedence values as follows:

```
expression : expression PLUS expression                # prec = 1, left
           | expression MINUS expression               # prec = 1, left
           | expression TIMES expression               # prec = 2, left
           | expression DIVIDE expression              # prec = 2, left
           | LPAREN expression RPAREN                  # prec = unknown
           | NUMBER                                    # prec = unknown
```

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, 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).

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.

One problem with the precedence specifier technique is that it is sometimes necessary to change the precedence of an operator in certain contents. For example, consider a unary-minus operator in "3 + 4 * -5". Normally, unary minus has 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 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.

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 a 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
)
```

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 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`.

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: SLR


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
```

In the file, each state of the grammar is described. Within each state the "." indicates the current location of the parse within any applicable grammar rules. 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`.

Syntax Error Handling

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). 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. 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`.

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.

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."
    # Read ahead looking for a closing '}'
    while 1:
        tok = yacc.token()             # Get the next token
        if not tok or tok.type == 'RBRACE': break
    yacc.restart()
```

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

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

Within the `p_error()` function, three functions are available to control the behavior of the parser:

- `yacc.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.
- `yacc.token()`. This returns the next token on the input stream.
- `yacc.restart()`. This discards the entire parsing stack and resets the parser to its initial state.

Note: these functions are only available when invoking `p_error()` and are not available at any other time.

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 1:
```

```
            tok = yacc.token()             # Get the next token
            if not tok or tok.type == 'SEMI': break
        yacc.errok()

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

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.

Line Number Tracking

`yacc.py` automatically tracks line numbers for all of the grammar symbols and tokens it processes. To retrieve the line numbers, two functions are used in grammar rules:

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

Since line numbers are managed internally by the parser, there is usually no need to modify the line numbers. However, if you want to save the line numbers in a parse-tree node, you will need to make your own private copy.

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. Simply create a data structure for 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])
```

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])
```

Yacc implementation notes

- The default parsing method is SLR. To use LALR(1) instead, run yacc() as follows:

  ```
  yacc.yacc(method="LALR")
  ```

  Note: LALR table generation takes approximately twice as long as SLR table generation. There is no difference in actual parsing performance---the same code is used in both cases.

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

  ```
  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

  ```
  yacc.yacc(debug=0)
  ```

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

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

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

  ```
  yacc.yacc(write_tables=0)
  ```

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:

```
yacc.parse(debug=1)
```

- To redirect the debugging output to a filename of your choosing, use:

```
yacc.parse(debug=1, debugfile="debugging.out")
```

- The `yacc.yacc()` function really returns a parser object. If you want to support multiple parsers in the same application, do this:

```
p = yacc.yacc()
...
p.parse()
```

Note: The function `yacc.parse()` is bound to the last parser that was generated.

- Since the generation of the SLR 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. For more complex languages such as C, table generation may take 30-60 seconds on a slow machine. Please be patient.

- Since LR parsing is mostly 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 your grammar rules.

Parser and Lexer State Management

In advanced parsing applications, you may want to have multiple parsers and lexers. Furthermore, the parser may want to control the behavior of the lexer in some way.

To do this, it is important to note that both the lexer and parser are actually implemented as objects. These objects are returned by the `lex()` and `yacc()` functions respectively. For example:

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

Within lexer and parser rules, these objects are also available. In the lexer, the "lexer" attribute of a token refers to the lexer object in use. 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.

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.

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.