# 【译】Python Lex Yacc手册

看云文档小组

# 目　录

# 0 一些翻译约定

本文是PLY (Python Lex-Yacc)的中文翻译版。转载请注明出处。

如果你从事编译器或解析器的开发工作，你可能对lex和yacc不会陌生，PLY是David Beazley实现的基于Python的lex和yacc。作者最著名的成就可能是其撰写的Python Cookbook, 3rd Edition。我因为偶然的原因接触了PLY，觉得是个好东西，但是似乎国内没有相关的资料。于是萌生了翻译的想法，虽然内容不算多，但是由于能力有限，很多概念不了解，还专门补习了编译原理，这对我有很大帮助。为了完成翻译，经过初译，复审，排版等，花费我很多时间，最终还是坚持下来了，希望对需要的人有所帮助。另外，第一次大规模翻译英文，由于水平有限，如果错误或者不妥的地方还请指正，非常感谢。

## 0 一些翻译约定

| 英文词汇 | 翻译约定 |
| --- | --- |
| token | 标记 |
| context free grammar | 上下文无关文法 |
| syntax directed translation | 语法制导的翻译 |
| ambiguity | 二义 |
| terminals | 终结符 |
| non-terminals | 非终结符 |
| documentation string | 文档字符串（python中的 `_docstring_` ） |
| shift-reduce | 移进-归约 |
| Empty Productions | 空产生式 |
| Panic mode recovery | 悲观恢复模式 |

# 1 前言和预备

本文指导你使用PLY进行词法分析和语法解析的，鉴于解析本身是个复杂性的事情，在你使用PLY投入大规模的开发前，我强烈建议你完整地阅读或者浏览本文档。

PLY-3.0能同时兼容 `Python2` 和 `Python3` 。需要注意的是，对于Python3的支持是新加入的，还没有广泛的测试（尽管所有的例子和单元测试都能够在Pythone3下通过）。如果你使用的是Python2，应该使用Python2.4以上版本，虽然，PLY最低能够支持到Python2.2，不过一些可选的功能需要新版本模块的支持。

# 2 介绍

PLY是纯粹由Python实现的Lex和yacc（流行的编译器构建工具）。PLY的设计目标是尽可能的沿袭传统lex和yacc工具的工作方式，包括支持LALR(1)分析法、提供丰富的输入验证、错误报告和诊断。因此，如果你曾经在其他编程语言下使用过yacc，你应该能够很容易的迁移到PLY上。

2001年，我在芝加哥大学教授“编译器简介”课程时开发了的早期的PLY。学生们使用Python和PLY构建了一个类似Pascal的语言的完整编译器，其中的语言特性包括：词法分析、语法分析、类型检查、类型推断、嵌套作用域，并针对SPARC处理器生成目标代码等。最终他们大约实现了30种不同的编译器！PLY在接口设计上影响使用的问题也被学生们所提出。从2001年以来，PLY继续从用户的反馈中不断改进。为了适应对未来的改进需求，PLY3.0在原来基础上进行了重大的重构。

由于PLY是作为教学工具来开发的，你会发现它对于标记和语法规则是相当严谨的，这一定程度上是为了帮助新手用户找出常见的编程错误。不过，高级用户也会发现这有助于处理真实编程语言的复杂语法。还需要注意的是，PLY没有提供太多花哨的东西（例如，自动构建抽象语法树和遍历树），我也不认为它是个分析框架。相反，你会发现它是一个用Python实现的，基本的，但能够完全胜任的lex/yacc。

本文的假设你多少熟悉分析理论、语法制导的翻译、基于其他编程语言使用过类似lex和yacc的编译器构建工具。如果你对这些东西不熟悉，你可能需要先去一些书籍中学习一些基础，比如：Aho, Sethi和Ullman的《Compilers: Principles, Techniques, and Tools》（《编译原理》），和O’ Reilly’ 出版的John Levine的《lex and yacc》。事实上，《lex and yacc》和PLY使用的概念几乎相同。

# 3 PLY概要

PLY包含两个独立的模块： `lex.py` 和 `yacc.py` ，都定义在ply包下。lex.py模块用来将输入字符通过一系列的正则表达式分解成标记序列，yacc.py通过一些上下文无关的文法来识别编程语言语法。yacc.py使用LR解析法，并使用LALR(1)算法（默认）或者SLR算法生成分析表。

这两个工具是为了一起工作的。lex.py通过向外部提供 `token()` 方法作为接口，方法每次会从输入中返回下一个有效的标记。yacc.py将会不断的调用这个方法来获取标记并匹配语法规则。yacc.py的的功能通常是生成抽象语法树( `AST` )，不过，这完全取决于用户，如果需要，yacc.py可以直接用来完成简单的翻译。

就像相应的unix工具，yacc.py提供了大多数你期望的特性，其中包括：丰富的错误检查、语法验证、支持空产生式、错误的标记、通过优先级规则解决二义性。事实上，传统yacc能够做到的PLY都应该支持。

yacc.py与Unix下的yacc的主要不同之处在于，yacc.py没有包含一个独立的代码生成器，而是在PLY中依赖反射来构建词法分析器和语法解析器。不像传统的lex/yacc工具需要一个独立的输入文件，并将之转化成一个源文件，Python程序必须是一个可直接可用的程序，这意味着不能有额外的源文件和特殊的创建步骤（像是那种执行yacc命令来生成Python代码）。又由于生成分析表开销较大，PLY会缓存生成的分析表，并将它们保存在独立的文件中，除非源文件有变化，会重新生成分析表，否则将从缓存中直接读取。

# 4 Lex

`lex.py` 是用来将输入字符串标记化。例如，假设你正在设计一个编程语言，用户的输入字符串如下：

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

标记器将字符串分割成独立的标记：

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

标记通常用一组名字来命名和表示：

```
'ID','EQUALS','NUMBER','PLUS','NUMBER','TIMES','LPAREN','ID','MINUS','ID','RPAR
EN'
```

将标记名和标记值本身组合起来：

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

正则表达式是描述标记规则的典型方法，下一节展示如何用lex.py实现。

# 4.1 Lex的例子

下面的例子展示了如何使用lex.py对输入进行标记

```
# ------------------------------------------------------------
# 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()
```

为了使lexer工作，你需要给定一个输入，并传递给 `input()` 方法。然后，重复调用 `token()` 方法来获取标记序列，下面的代码展示了这种用法：

```
# 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
```

程序执行，将给出如下输出：

```
$ 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也同时支持迭代，你可以把上面的循环写成这样：

```
for tok in lexer:
    print tok
```

由lexer.token()方法返回的标记是LexToken类型的实例，拥有 `tok.type` ， `tok.value` ， `tok.lineno` 和 `tok.lexpos` 属性，下面的代码展示了如何访问这些属性：

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

`tok.type` 和 `tok.value` 属性表示标记本身的类型和值。 `tok.line` 和 `tok.lexpos` 属性包含了标记的位置信息， `tok.lexpos` 表示标记相对于输入串起始位置的偏移。

# 4.2 标记列表

词法分析器必须提供一个标记的列表，这个列表将所有可能的标记告诉分析器，用来执行各种验证，同时也提供给yacc.py作为终结符。

在上面的例子中，是这样给定标记列表的：

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

# 4.3 标记的规则

每种标记用一个正则表达式规则来表示，每个规则需要以”t_“开头声明，表示该声明是对标记的规则定义。对于简单的标记，可以定义成这样（在Python中使用raw string能比较方便的书写正则表达式）：

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

这里，紧跟在t_后面的单词，必须跟标记列表中的某个标记名称对应。如果需要执行动作的话，规则可以写成一个方法。例如，下面的规则匹配数字字串，并且将匹配的字符串转化成Python的整型：

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

如果使用方法的话，正则表达式写成方法的文档字符串。方法总是需要接受一个LexToken实例的参数，该实例有一个t.type的属性（字符串表示）来表示标记的类型名称，t.value是标记值（匹配的实际的字符串），t.lineno表示当前在源输入串中的作业行，t.lexpos表示标记相对于输入串起始位置的偏移。默认情况下，t.type是以t_开头的变量或方法的后面部分。方法可以在方法体里面修改这些属性。但是，如果这样做，应该返回结果token，否则，标记将被丢弃。

在lex内部，lex.py用 `re` 模块处理模式匹配，在构造最终的完整的正则式的时候，用户提供的规则按照下面的顺序加入：

1. 所有由方法定义的标记规则，按照他们的出现顺序依次加入
2. 由字符串变量定义的标记规则按照其正则式长度倒序后，依次加入（长的先入）
3. 顺序的约定对于精确匹配是必要的。比如，如果你想区分‘=’和‘==’，你需要确保‘==’优先检查。如果用字符串来定义这样的表达式的话，通过将较长的正则式先加入，可以帮助解决这个问题。用方法定义标记，可以显示地控制哪个规则优先检查。

为了处理保留字，你应该写一个单一的规则来匹配这些标识，并在方法里面作特殊的查询：

```
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
```

这样做可以大大减少正则式的个数，并稍稍加快处理速度。注意：你应该避免为保留字编写单独的规则，例如，如果你像下面这样写：

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

但是，这些规则照样也能够匹配以这些字符开头的单词，比如’forget’或者’printed’，这通常不是你想要的。

# 4.4 标记的值

标记被lex返回后，它们的值被保存在 `value` 属性中。正常情况下，value是匹配的实际文本。事实上，value可以被赋为任何Python支持的类型。例如，当扫描到标识符的时候，你可能不仅需要返回标识符的名字，还需要返回其在符号表中的位置，可以像下面这样写：

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

需要注意的是，不推荐用其他属性来保存值，因为yacc.py模块只会暴露出标记的value属性，访问其他属性会变得不自然。如果想保存多种属性，可以将元组、字典、或者对象实例赋给value。

# 4.5 丢弃标记

想丢弃像注释之类的标记，只要不返回value就行了，像这样：

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

为标记声明添加”ignore_“前缀同样可以达到目的：

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

如果有多种文本需要丢弃，建议使用方法来定义规则，因为方法能够提供更精确的匹配优先级控制（方法根据出现的顺序，而字符串的正则表达式依据正则表达式的长度）

# 4.6 行号和位置信息

默认情况下，lex.py对行号一无所知。因为lex.py根本不知道何为”行”的概念（换行符本身也作为文本的一部分）。不过，可以通过写一个特殊的规则来记录行号：

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

在这个规则中，当前lexer对象t.lexer的lineno属性被修改了，而且空行被简单的丢弃了，因为没有任何的返回。lex.py也不自动做列跟踪。但是，位置信息被记录在了每个标记对象的 `lexpos` 属性中，这样，就有可能来计算列信息了。例如：每当遇到新行的时候就重置列值：

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

通常，计算列的信息是为了指示上下文的错误位置，所以只在必要时有用。

# 4.7 忽略字符

`t_ignore` 规则比较特殊，是lex.py所保留用来忽略字符的，通常用来跳过空白或者不需要的字符。虽然可以通过定义像 `t_newline()` 这样的规则来完成相同的事情，不过使用t_ignore能够提供较好的词法分析性能，因为相比普通的正则式，它被特殊化处理了。

# 4.8 字面字符

字面字符可以通过在词法模块中定义一个 `literals` 变量做到，例如：

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

或者

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

字面字符是指单个字符，表示把字符本身作为标记，标记的 `type` 和 `value` 都是字符本身。不过，字面字符是在其他正则式之后被检查的，因此如果有规则是以这些字符开头的，那么这些规则的优先级较高。

# 4.9 错误处理

最后，在词法分析中遇到非法字符时， `t_error()` 用来处理这类错误。这种情况下， `t.value` 包含了余下还未被处理的输入字串，在之前的例子中，错误处理方法是这样的：

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

这个例子中，我们只是简单的输出不合法的字符，并且通过调用 `t.lexer.skip(1)` 跳过一个字符。

# 4.10 构建和使用lexer

函数 `lex.lex()` 使用Python的反射机制读取调用上下文中的正则表达式，来创建lexer。lexer一旦创建好，有两个方法可以用来控制lexer对象：

- `lexer.input(data)` 重置lexer和输入字串
- `lexer.token()` 返回下一个LexToken类型的标记实例，如果进行到输入字串的尾部时将返回 `None`

推荐直接在lex()函数返回的lexer对象上调用上述接口，尽管也可以向下面这样用模块级别的lex.input()和lex.token()：

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

在这个例子中，lex.input()和lex.token()是模块级别的方法，在lex模块中，input()和token()方法绑定到最新创建的lexer对象的对应方法上。最好不要这样用，因为这种接口可能不知道在什么时候就失效（译者注：垃圾回收？）

# 4.11 @TOKEN装饰器

在一些应用中，你可能需要定义一系列辅助的记号来构建复杂的正则表达式，例如：

```
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. ?????
    ...
```

在这个例子中，我们希望ID的规则引用上面的已有的变量。然而，使用文档字符串无法做到，为了解决这个问题，你可以使用 `@TOKEN` 装饰器：

```
from ply.lex import TOKEN

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

装饰器可以将identifier关联到t_ID()的文档字符串上以使lex.py正常工作，一种等价的做法是直接给文档字符串赋值：

```
def t_ID(t):
    ...

t_ID.__doc__ = identifier
```

注意：@TOKEN装饰器需要Python-2.4以上的版本。如果你在意老版本Python的兼容性问题，使用上面的等价办法。

# 4.12 优化模式

为了提高性能，你可能希望使用Python的优化模式（比如，使用-o选项执行Python）。然而，这样的话，Python会忽略文档字串，这是lex.py的特殊问题，可以通过在创建lexer的时候使用optimize选项：

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

接着，用Python常规的模式运行，这样，lex.py会在当前目录下创建一个lextab.py文件，这个文件会包含所有的正则表达式规则和词法分析阶段的分析表。然后，lextab.py可以被导入用来构建lexer。这种方法大大改善了词法分析程序的启动时间，而且可以在Python的优化模式下工作。
想要更改生成的文件名，使用如下参数：

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

在优化模式下执行，需要注意的是lex会被禁用大多数的错误检查。因此，建议只在确保万事俱备准备发布最终代码时使用。

# 4.13 调试

如果想要调试，可以使lex()运行在调试模式：

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

这将打出一些调试信息，包括添加的规则、最终的正则表达式和词法分析过程中得到的标记。
除此之外，lex.py有一个简单的主函数，不但支持对命令行参数输入的字串进行扫描，还支持命令行参数指定的文件名：

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

想要了解高级调试的详情，请移步至最后的高级调试部分。

# 4.14 其他方式定义词法规则

上面的例子，词法分析器都是在单个的Python模块中指定的。如果你想将标记的规则放到不同的模块，使用module关键字参数。例如，你可能有一个专有的模块，包含了标记的规则：

```
# 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)
```

现在，如果你想要从不同的模块中构建分析器，应该这样（在交互模式下）：

```
>>> 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
```

`module` 选项也可以指定类型的实例，例如：

```
import ply.lex as lex

class MyLexer:
    # 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 = 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
```

当从类中定义lexer，你需要创建类的实例，而不是类本身。这是因为，lexer的方法只有被绑定（bound-methods）对象后才能使PLY正常工作。

当给lex()方法使用module选项时，PLY使用 `dir()` 方法，从对象中获取符号信息，因为不能直接访问对象的 `__dict__` 属性。（译者注：可能是因为兼容性原因，__dict__这个方法可能不存在）

最后，如果你希望保持较好的封装性，但不希望什么东西都写在类里面，lexers可以在闭包中定义，例如：

```
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()
```

# 4.15 额外状态维护

在你的词法分析器中，你可能想要维护一些状态。这可能包括模式设置，符号表和其他细节。例如，假设你想要跟踪 `NUMBER` 标记的出现个数。
一种方法是维护一个全局变量：

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

如果你不喜欢全局变量，另一个记录信息的地方是lexer对象内部。可以通过当前标记的lexer属性访问：

```
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
```

上面这样做的优点是当同时存在多个lexer实例的情况下，简单易行。不过这看上去似乎是严重违反了面向对象的封装原则。lexer的内部属性（除了lineno）都是以lex开头命名的（lexdata、lexpos）。因此，只要不以lex开头来命名属性就很安全的。
如果你不喜欢给lexer对象赋值，你可以自定义你的lexer类型，就像前面看到的那样：

```
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
```

如果你的应用会创建很多lexer的实例，并且需要维护很多状态，上面的类可能是最容易管理的。

状态也可以用闭包来管理，比如，在Python3中：

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

# 4.16 Lexer克隆

如果有必要的话，lexer对象可以通过 `clone()` 方法来复制：

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

当lexer被克隆后，复制品能够精确的保留输入串和内部状态，不过，新的lexer可以接受一个不同的输出字串，并独立运作起来。这在几种情况下也许有用：当你在编写的解析器或编译器涉及到递归或者回退处理时，你需要扫描先前的部分，你可以clone并使用复制品，或者你在实现某种预编译处理，可以clone一些lexer来处理不同的输入文件。
创建克隆跟重新调用lex.lex()的不同点在于，PLY不会重新构建任何的内部分析表或者正则式。当lexer是用类或者闭包创建的，需要注意类或闭包本身的的状态。换句话说你要注意新创建的lexer会共享原始lexer的这些状态，比如：

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

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

# 4.17 Lexer的内部状态

lexer有一些内部属性在特定情况下有用：

- `lexer.lexpos` 。这是一个表示当前分析点的位置的整型值。如果你修改这个值的话，这会改变下一个token()的调用行为。在标记的规则方法里面，这个值表示紧跟匹配字串后面的第一个字符的位置，如果这个值在规则中修改，下一个返回的标记将从新的位置开始匹配
- `lexer.lineno` 。表示当前行号。PLY只是声明这个属性的存在，却永远不更新这个值。如果你想要跟踪行号的话，你需要自己添加代码（ 4.6 行号和位置信息）
- `lexer.lexdata` 。当前lexer的输入字串，这个字符串就是input()方法的输入字串，更改它可能是个糟糕的做法，除非你知道自己在干什么。
- `lexer.lexmatch` 。PLY内部调用Python的re.match()方法得到的当前标记的原始的Match对象，该对象被保存在这个属性中。如果你的正则式中包含分组的话，你可以通过这个对象获得这些分组的值。注意：这个属性只在有标记规则定义的方法中才有效。

# 4.18 基于条件的扫描和启动条件

在高级的分析器应用程序中，使用状态化的词法扫描是很有用的。比如，你想在出现特定标记或句子结构的时候触发开始一个不同的词法分析逻辑。PLY允许lexer在不同的状态之间转换。每个状态可以包含一些自己独特的标记和规则等。这是基于GNU flex的“启动条件”来实现的，关于flex详见
http://flex.sourceforge.net/manual/Start-Conditions.html#Start-Conditions
要使用lex的状态，你必须首先声明。通过在lex模块中声明”states”来做到：

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

这个声明中包含有两个状态：’ foo’ 和’ bar’ 。状态可以有两种类型：’ 排他型’ 和’ 包容型’ 。排他型的状态会使得lexer的行为发生完全的改变：只有能够匹配在这个状态下定义的规则的标记才会返回；包容型状态会将定义在这个状态下的规则添加到默认的规则集中，进而，只要能匹配这个规则集的标记都会返回。
一旦声明好之后，标记规则的命名需要包含状态名：

```
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
```

一个标记可以用在多个状态中，只要将多个状态名包含在声明中：

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

同样的，在任何状态下都生效的声明可以在命名中使用 `ANY` ：

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

不包含状态名的情况下，标记被关联到一个特殊的状态 `INITIAL` ，比如，下面两个声明是等价的：

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

特殊的 `t_ignore()` 和 `t_error()` 也可以用状态关联：

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

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

词法分析默认在 `INITIAL` 状态下工作，这个状态下包含了所有默认的标记规则定义。对于不希望使用“状态”的用户来说，这是完全透明的。在分析过程中，如果你想要改变词法分析器的这种的状态，使用 `begin()` 方法：

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

使用begin()切换回初始状态：

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

状态的切换可以使用栈：

```
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
```

当你在面临很多状态可以选择进入，而又仅仅想要回到之前的状态时，状态栈比较有用。

举个例子会更清晰。假设你在写一个分析器想要从一堆C代码中获取任意匹配的闭合的大括号里面的部分：这意味着，当遇到起始括号‘{ ‘，你需要读取与之匹配的‘}’以上的所有部分。并返回字符串。使用通常的正则表达式几乎不可能，这是因为大括号可以嵌套，而且可以有注释，字符串等干扰。因此，试图简单的匹配第一个出现的‘}’是不行的。这里你可以用lex的状态来做到：

```
# 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)
```

这个例子中，第一个’{‘使得lexer记录了起始位置，并且进入新的状态’ccode’。一系列规则用来匹配接下来的输入，这些规则只是丢弃掉标记（不返回值），如果遇到闭合右括号，t_ccode_rbrace规则收集其中所有的代码（利用先前记录的开始位置），并保存，返回的标记类型为’CCODE’，与此同时，词法分析的状态退回到初始状态。

# 4.19 其他问题

- lexer需要输入的是一个字符串。好在大多数机器都有足够的内存，这很少导致性能的问题。这意味着，lexer现在还不能用来处理文件流或者socket流。这主要是受到re模块的限制。
- lexer支持用Unicode字符描述标记的匹配规则，也支持输入字串包含Unicode
- 如果你想要向 `re.compile()` 方法提供flag，使用reflags选项：lex.lex(reflags=re.UNICODE)
- 由于lexer是全部用Python写的，性能很大程度上取决于Python的re模块，即使已经尽可能的高效了。当接收极其大量的输入文件时表现并不尽人意。如果担忧性能，你可以升级到最新的Python，或者手工创建分析器，或者用C语言写lexer并做成扩展模块。

如果你要创建一个手写的词法分析器并计划用在yacc.py中，只需要满足下面的要求：

- 需要提供一个token()方法来返回下一个标记，如果没有可用的标记了，则返回None。
- token()方法必须返回一个tok对象，具有type和value属性。如果行号需要跟踪的话，标记还需要定义lineno属性。

# 5 语法分析基础

yacc.py用来对语言进行语法分析。在给出例子之前，必须提一些重要的背景知识。首先，‘语法’通常用BNF范式来表达。例如，如果想要分析简单的算术表达式，你应该首先写下无二义的文法：

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

term       : term * factor
           | term / factor
           | factor

factor     : NUMBER
           | ( expression )
```

在这个文法中，像 `NUMBER` ，`+` ，`-` ，`*` ，`/` 的符号被称为终结符，对应原始的输入。类似 `term` ，`factor` 等称为非终结符，它们由一系列终结符或其他规则的符号组成，用来指代语法规则。

通常使用一种叫语法制导翻译的技术来指定某种语言的语义。在语法制导翻译中，符号及其属性出现在每个语法规则后面的动作中。每当一个语法被识别，动作就能够描述需要做什么。比如，对于上面给定的文法，想要实现一个简单的计算器，应该写成下面这样：

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

一种理解语法指导翻译的好方法是将符号看成对象。与符号相关的值代表了符号的“状态”（比如上面的val属性），语义行为用一组操作符号及符号值的函数或者方法来表达。

Yacc用的分析技术是著名的LR分析法或者叫移进-归约分析法。LR分析法是一种自下而上的技术：首先尝试识别右部的语法规则，每当右部得到满足，相应的行为代码将被触发执行，当前右边的语法符号将被替换为左边的语

法符号。（归约）

LR分析法一般这样实现：将下一个符号进栈，然后结合栈顶的符号和后继符号（译者注：下一个将要输入符号），与文法中的某种规则相比较。具体的算法可以在编译器的手册中查到，下面的例子展现了如果通过上面定义的文法，来分析3 + 5 * ( 10 - 20 )这个表达式，$用来表示输入结束

```
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!
```

（译者注：action里面的Shift就是进栈动作，简称移进；Reduce是归约）

在分析表达式的过程中，一个相关的自动状态机和后继符号决定了下一步应该做什么。如果下一个标记看起来是一个有效语法（产生式）的一部分（通过栈上的其他项判断这一点），那么这个标记应该进栈。如果栈顶的项可以组成一个完整的右部语法规则，一般就可以进行“归约”，用产生式左边的符号代替这一组符号。当归约发生时，相应的行为动作就会执行。如果输入标记既不能移进也不能归约的话，就会发生语法错误，分析器必须进行相应的错误恢复。分析器直到栈空并且没有另外的输入标记时，才算成功。 需要注意的是，这是基于一个有限自动机实现的，有限自动器被转化成分析表。分析表的构建比较复杂，超出了本文的讨论范围。不过，这构建过程的微妙细节能够解释为什么在上面的例子中，解析器选择在步骤9将标记转移到堆栈中，而不是按照规则expr : expr + term做归约。

# 6 Yacc

ply.yacc模块实现了PLY的分析功能， ‘yacc’ 是 ‘Yet Another Compiler Compiler’ 的缩写并保留了其作为Unix工具的名字。

# 6.1 一个例子

假设你希望实现上面的简单算术表达式的语法分析，代码如下：

```
# 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
```

在这个例子中，每个语法规则被定义成一个Python的方法，方法的文档字符串描述了相应的上下文无关文法，方法的语句实现了对应规则的语义行为。每个方法接受一个单独的p参数，p是一个包含有当前匹配语法的符号的序列，p[i]与语法符号的对应关系如下：

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

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

其中，p[i]的值相当于词法分析模块中对p.value属性赋的值，对于非终结符的值，将在归约时由p[0]的赋值决定，这里的值可以是任何类型，当然，大多数情况下只是Python的简单类型、元组或者类的实例。在这个例子中，我们依赖这样一个事实：NUMBER标记的值保存的是整型值，所有规则的行为都是得到这些整型值的算术运算结果，并传递结果。

> 注意：在这里负数的下标有特殊意义–这里的p[-1]不等同于p[3]。详见下面的嵌入式动作部分

在yacc中定义的第一个语法规则被默认为起始规则（这个例子中的第一个出现的expression规则）。一旦起始规则被分析器归约，而且再无其他输入，分析器终止，最后的值将返回（这个值将是起始规则的p[0]）。注意：也可以通过在yacc()中使用start关键字参数来指定起始规则
p_error(p)规则用于捕获语法错误。详见处理语法错误部分
为了构建分析器，需要调用yacc.yacc()方法。这个方法查看整个当前模块，然后试图根据你提供的文法构建LR分析表。第一次执行yacc.yacc()，你会得到如下输出：

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

由于分析表的得出相对开销较大（尤其包含大量的语法的情况下），分析表被写入当前目录的一个叫

parsetab.py的文件中。除此之外，会生成一个调试文件parser.out。在接下来的执行中，yacc直到发现文法发生变化，才会重新生成分析表和parsetab.py文件，否则yacc会从parsetab.py中加载分析表。注：如果有必要的话这里输出的文件名是可以改的。
如果在你的文法中有任何错误的话，yacc.py会产生调试信息，而且可能抛出异常。一些可以被检测到的错误如下：

- 方法重复定义（在语法文件中具有相同名字的方法）
- 二义文法产生的移进-归约和归约-归约冲突
- 指定了错误的文法
- 不可终止的递归（规则永远无法终结）
- 未使用的规则或标记
- 未定义的规则或标记

下面几个部分将更详细的讨论语法规则
这个例子的最后部分展示了如何执行由yacc()方法创建的分析器。你只需要简单的调用parse()，并将输入字符串作为参数就能运行分析器。它将运行所有的语法规则，并返回整个分析的结果，这个结果就是在起始规则中赋给p[0]的值。

# 6.2 将语法规则合并

如果语法规则类似的话，可以合并到一个方法中。例如，考虑前面例子中的两个规则：

```
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]
```

比起写两个方法，你可以像下面这样写在一个方法里面：

```
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]
```

总之，方法的文档字符串可以包含多个语法规则。所以，像这样写也是合法的（尽管可能会引起困惑）：

```
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]
```

如果所有的规则都有相似的结构，那么将语法规则合并才是个不错的注意（比如，产生式的项数相同）。不然，语义动作可能会变得复杂。不过，简单情况下，可以使用 `len()` 方法区分，比如：

```
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]
```

如果考虑解析的性能，你应该避免像这些例子一样在一个语法规则里面用很多条件来处理。因为，每次检查当前究竟匹配的是哪个语法规则的时候，实际上重复做了分析器已经做过的事（分析器已经准确的知道哪个规则被匹配了）。为每个规则定义单独的方法，可以消除这点开销。

# 6.3 字面字符

如果愿意，可以在语法规则里面使用单个的字面字符，例如：

```
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]
```

字符必须像’+’那样使用单引号。除此之外，需要将用到的字符定义单独定义在lex文件的 `literals` 列表里：

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

字面的字符只能是单个字符。因此，像’<=’或者’==’都是不合法的，只能使用一般的词法规则（例如t_EQ = r’==’）。

# 6.4 空产生式

## 6.6 处理二义文法

上面例子中，对表达式的文法描述用一种特别的形式规避了二义文法。然而，在很多情况下，这样的特殊文法很难写，或者很别扭。一个更为自然和舒服的语法表达应该是这样的：

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

不幸的是，这样的文法是存在二义性的。举个例子，如果你要解析字符串”3 * 4 + 5”，操作符如何分组并没有指明，究竟是表示”(3 * 4) + 5”还是”3 * (4 + 5)”呢？
如果在yacc.py中存在二义文法，会输出”移进归约冲突”或者”归约归约冲突”。在分析器无法确定是将下一个符号移进栈还是将当前栈中的符号归约时会产生移进归约冲突。例如，对于”3 * 4 + 5”，分析器内部栈是这样工作的：

```
Step Symbol Stack           Input Tokens            Action
---- ---------------------  ---------------------   ----------------------------
----
1    $                                3 * 4 + 5$    Shift 3
2    $ 3                                * 4 + 5$    Reduce : expression : NUMBE
R
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 ????
```

在这个例子中，当分析器来到第6步的时候，有两种选择：一是按照expr : expr * expr归约，一是将标记’ +’ 继续移进栈。两种选择对于上面的上下文无关文法而言都是合法的。
默认情况下，所有的移进归约冲突会倾向于使用移进来处理。因此，对于上面的例子，分析器总是会将’ +’ 进栈，而不是做归约。虽然在很多情况下，这个策略是合适的（像”if-then”和”if-then-else”），但这对于算术表达式是不够的。事实上，对于上面的例子，将’ +’ 进栈是完全错误的，应当先将expr * expr归约，因为乘法的优先级要高于加法。
为了解决二义文法，尤其是对表达式文法，yacc.py允许为标记单独指定优先级和结合性。需要像下面这样增加一个precedence变量：

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

这样的定义说明PLUS/MINUS标记具有相同的优先级和左结合性，TIMES/DIVIDE具有相同的优先级和左结合性。在precedence声明中，标记的优先级从低到高。因此，这个声明表明TIMES/DIVIDE（他们较晚加入precedence）的优先级高于PLUS/MINUS。
由于为标记添加了数字表示的优先级和结合性的属性，所以，对于上面的例子，将会得到：

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

随后这些值被附加到语法规则的优先级和结合性属性上，这些值由最右边的终结符的优先级和结合性决定：

```
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 spe
cified)
           | NUMBER                                     # level = None (not spe
cified)
```

当出现移进归约冲突时，分析器生成器根据下面的规则解决二义文法：

1. 如果当前的标记的优先级高于栈顶规则的优先级，移进当前标记
2. 如果栈顶规则的优先级更高，进行归约
3. 如果当前的标记与栈顶规则的优先级相同，如果标记是左结合的，则归约，否则，如果是右结合的则移进
4. 如果没有优先级可以参考，默认对于移进归约冲突执行移进

比如，当解析到”expression PLUS expression”这个语法时，下一个标记是TIMES，此时将执行移进，因为TIMES具有比PLUS更高的优先级；当解析到”expression TIMES expression”，下一个标记是PLUS，此时将执行归约，因为PLUS的优先级低于TIMES。
如果在使用前三种技术解决已经归约冲突后，yacc.py将不会报告语法中的冲突或者错误（不过，会在parser.out这个调试文件中输出一些信息）
使用precedence指定优先级的技术会带来一个问题，有时运算符的优先级需要基于上下文。例如，考虑”3 + 4 * -5”中的一元的’ - ’。数学上讲，一元运算符应当拥有较高的优先级。然而，在我们的precedence定义中，

MINUS的优先级却低于TIMES。为了解决这个问题，precedene规则中可以包含”虚拟标记”：

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

在语法文件中，我们可以这么表示一元算符：

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

在这个例子中，%prec UMINUS覆盖了默认的优先级（MINUS的优先级），将UMINUS指代的优先级应用在该语法规则上。
起初，UMINUS标记的例子会让人感到困惑。UMINUS既不是输入的标记也不是语法规则，你应当将其看成precedence表中的特殊的占位符。当你使用%prec宏时，你是在告诉yacc，你希望表达式使用这个占位符所表示的优先级，而不是正常的优先级。
还可以在precedence表中指定”非关联”。这表明你不希望链式运算符。比如，假如你希望支持比较运算符’’，但是你不希望支持 a < b < c，只要简单指定规则如下：

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

此时，当输入形如 a < b < c时，将产生语法错误，却不影响形如 a < b 的表达式。
对于给定的符号集，存在多种语法规则可以匹配时会产生归约/归约冲突。这样的冲突往往很严重，而且总是通过匹配最早出现的语法规则来解决。归约/归约冲突几乎总是相同的符号集合具有不同的规则可以匹配，而在这一点上无法抉择，比如：

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

expression : expression PLUS expression
           | expression MINUS expression
           | expression TIMES expression
           | expression DIVIDE expression
```

```
            | LPAREN expression RPAREN
            | NUMBER
```

这个例子中，对于下面这两条规则将产生归约/归约冲突：

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

比如，对于”a = 5”，分析器不知道应当按照assignment : ID EQUALS NUMBER归约，还是先将5归约成expression，再归约成assignment : ID EQUALS expression。
应当指出的是，只是简单的查看语法规则是很难减少归约/归约冲突。如果出现归约/归约冲突，yacc()会帮助打印出警告信息：

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

上面的信息标识出了冲突的两条规则，但是，并无法指出究竟在什么情况下会出现这样的状态。想要发现问题，你可能需要结合语法规则和 `parser.out` 调试文件的内容。

## 6.7 parser.out调试文件

使用LR分析算法跟踪移进/归约冲突和归约/归约冲突是件乐在其中的事。为了辅助调试，yacc.py在生成分析表时会创建出一个调试文件叫parser.out：

```
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
```

文件中出现的不同状态，代表了有效输入标记的所有可能的组合，这是依据文法规则得到的。当得到输入标记时，分析器将构造一个栈，并找到匹配的规则。每个状态跟踪了当前输入进行到语法规则中的哪个位置，在每个规则中，’.’表示当前分析到规则的哪个位置，而且，对于在当前状态下，输入的每个有效标记导致的动作也被罗列出来。当出现移进/归约或归约/归约冲突时，被忽略的规则前面会添加!，就像这样：

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

通过查看这些规则并结合一些实例，通常能够找到大部分冲突的根源。应该强调的是，不是所有的移进归约冲突都是不好的，想要确定解决方法是否正确，唯一的办法就是查看parser.out。

## 6.8 处理语法错误

如果你创建的分析器用于产品，处理语法错误是很重要的。一般而言，你不希望分析器在遇到错误的时候就抛出异常并终止，相反，你需要它报告错误，尽可能的恢复并继续分析，一次性的将输入中所有的错误报告给用户。这是一些已知语言编译器的标准行为，例如C,C++,Java。在PLY中，在语法分析过程中出现错误，错误会被立即检测到（分析器不会继续读取源文件中错误点后面的标记）。然而，这时，分析器会进入恢复模式，这个模式能够用来尝试继续向下分析。LR分析器的错误恢复是个理论与技巧兼备的问题，yacc.py提供的错误机制与Unix下的yacc类似，所以你可以从诸如O’Reilly出版的《Lex and yacc》的书中找到更多的细节。
当错误发生时，yacc.py按照如下步骤进行：

1. 第一次错误产生时，用户定义的p_error()方法会被调用，出错的标记会作为参数传入；如果错误是因为到达文件结尾造成的，传入的参数将为None。随后，分析器进入到“错误恢复”模式，该模式下不会在产生 `p_error()` 调用，直到它成功的移进3个标记，然后回归到正常模式。

2. 如果在p_error()中没有指定恢复动作的话，这个导致错误的标记会被替换成一个特殊的error标记。
3. 如果导致错误的标记已经是error的话，原先的栈顶的标记将被移除。
4. 如果整个分析栈被放弃，分析器会进入重置状态，并从他的初始状态开始分析。
5. 如果此时的语法规则接受error标记，error标记会移进栈。
6. 如果当前栈顶是error标记，之后的标记将被忽略，直到有标记能够导致error的归约。

### 6.8.1 根据error规则恢复和再同步

最佳的处理语法错误的做法是在语法规则中包含error标记。例如，假设你的语言有一个关于print的语句的语法规则：

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

为了处理可能的错误表达式，你可以添加一条额外的语法规则：

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

这样（expr错误时），error标记会匹配任意多个分号之前的标记（分号是 `SEMI` 指代的字符）。一旦找到分号，规则将被匹配，这样error标记就被归约了。
这种类型的恢复有时称为”分析器再同步”。error标记扮演了表示所有错误标记的通配符的角色，而紧随其后的标记扮演了同步标记的角色。
重要的一个说明是，通常error不会作为语法规则的最后一个标记，像这样：

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

这是因为，第一个导致错误的标记会使得该规则立刻归约，进而使得在后面还有错误标记的情况下，恢复变得困难。

### 6.8.2 悲观恢复模式

另一个错误恢复方法是采用“悲观模式”：该模式下，开始放弃剩余的标记，直到能够达到一个合适的恢复机会。
悲观恢复模式都是在p_error()方法中做到的。例如，这个方法在开始丢弃标记后，直到找到闭合的’}’，才重置分析器到初始化状态：

```
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()
```

下面这个方法简单的抛弃错误的标记，并告知分析器错误被接受了：

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

在 `p_error()` 方法中，有三个可用的方法来控制分析器的行为：

- `yacc.errok()` 这个方法将分析器从恢复模式切换回正常模式。这会使得不会产生error标记，并重置内部的error计数器，而且下一个语法错误会再次产生p_error()调用
- `yacc.token()` 这个方法用于得到下一个标记
- `yacc.restart()` 这个方法抛弃当前整个分析栈，并重置分析器为起始状态

注意：这三个方法只能在 `p_error()` 中使用，不能用在其他任何地方。

p_error()方法也可以返回标记，这样能够控制将哪个标记作为下一个标记返回给分析器。这对于需要同步一些特殊标记的时候有用，比如：

```
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
```

### 6.8.3 从产生式中抛出错误

如果有需要的话，产生式规则可以主动的使分析器进入恢复模式。这是通过抛出 `SyntacError` 异常做到的：

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

raise SyntaxError错误的效果就如同当前的标记是错误标记一样。因此，当你这么做的话，最后一个标记将被弹出栈，当前的下一个标记将是error标记，分析器进入恢复模式，试图归约满足error标记的规则。此后的步骤与检测到语法错误的情况是完全一样的，p_error()也会被调用。
手动设置错误有个重要的方面，就是p_error()方法在这种情况下不会调用。如果你希望记录错误，确保在抛出SyntaxError错误的产生式中实现。
注：这个功能是为了模仿yacc中的 `YYERROR` 宏的行为

### 6.8.4 错误恢复总结

对于通常的语言，使用error规则和再同步标记可能是最合理的手段。这是因为你可以将语法设计成在一个相对容易恢复和继续分析的点捕获错误。悲观恢复模式只在一些十分特殊的应用中有用，这些应用往往需要丢弃掉大量输入，再寻找合理的同步点。

## 6.9 行号和位置的跟踪

位置跟踪通常是个设计编译器时的技巧性玩意儿。默认情况下，PLY跟踪所有标记的行号和位置，这些信息可以这样得到：

- p.lineno(num)返回第num个符号的行号
- p.lexpos(num)返回第num个符号的词法位置偏移

例如：

```
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
```

注意：lexspan()方法只会返回的结束位置是最后一个符号的起始位置。
虽然，PLY对所有符号的行号和位置的跟踪很管用，但经常是不必要的。例如，你仅仅是在错误信息中使用行号，你通常可以仅仅使用关键标记的信息，比如：

```
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)
```

类似的，为了改善性能，你可以有选择性的将行号信息在必要的时候进行传递，这是通过p.set_lineno()实现的，

例如：

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

对于已经完成分析的规则，PLY不会保留行号信息，如果你是在构建抽象语法树而且需要行号，你应该确保行号保留在树上。

## 6.10 构造抽象语法树

yacc.py没有构造抽像语法树的特殊方法。不过，你可以自己很简单的构造出来。
一个最为简单的构造方法是为每个语法规则创建元组或者字典，并传递它们。有很多中可行的方案，下面是一个例子：

```
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])
```

另一种方法可以是为不同的抽象树节点创建一系列的数据结构，并赋值给p[0]：

```
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])
```

这种方式的好处是在处理复杂语义时比较简单：类型检查、代码生成、以及其他针对树节点的功能。

为了简化树的遍历，可以创建一个通用的树节点结构，例如：

```
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 嵌入式动作

yacc使用的分析技术只允许在规则规约后执行动作。假设有如下规则：

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

方法只会在符号A,B,C和D都完成后才能执行。可是有的时候，在中间阶段执行一小段代码是有用的。假如，你想在A完成后立即执行一些动作，像下面这样用空规则：

```
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
```

在这个例子中，空规则seen_A将在A移进分析栈后立即执行。p[-1]指代的是在分析栈上紧跟在seen_A左侧的符号。在这个例子中，是A符号。像其他普通的规则一样，在嵌入式行为中也可以通过为p[0]赋值来返回某些值。使用嵌入式动作可能会导致移进归约冲突，比如，下面的语法是没有冲突的：

```
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"
```

可是，如果像这样插入一个嵌入式动作：

```
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 :"
```

会产生移进归约冲，只是由于对于两个规则abcd和abcx中的C，分析器既可以根据abcd规则移进，也可以根据abcx规则先将空的seen_AB归约。

嵌入动作的一般用于分析以外的控制，比如为本地变量定义作用于。对于C语言：

```
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)
    ...
```

在这个例子中，new_scope作为嵌入式行为，在左大括号{之后立即执行。可以是调正内部符号表或者其他方面。statements_block一完成，代码可能会撤销在嵌入动作时的操作（比如，pop_scope()）

## 6.12 Yacc的其他

- 默认的分析方法是LALR，使用SLR请像这样运行 yacc()：yacc.yacc(method=”SLR”)注意：LRLR生成的分析表大约要比SLR的大两倍。解析的性能没有本质的区别，因为代码是一样的。由于LALR能力稍强，所以更多的用于复杂的语法。
- 默认情况下，yacc.py依赖lex.py产生的标记。不过，可以用一个等价的词法标记生成器代替：yacc.parse(lexer=x) 这个例子中，x必须是一个Lexer对象，至少拥有x.token()方法用来获取标记。如果将输入字串提供给yacc.parse()，lexer还必须具有x.input()方法。
- 默认情况下，yacc在调试模式下生成分析表（会生成parser.out文件和其他东西），使用yacc.yacc(debug=0)禁用调试模式。
- 改变parsetab.py的文件名：yacc.yacc(tabmodule=”foo”)
- 改变parsetab.py的生成目录：yacc.yacc(tabmodule=”foo”,outputdir=”somedirectory”)
- 不生成分析表：yacc.yacc(write_tables=0)。注意：如果禁用分析表生成，yacc()将在每次运行的时候重新构建分析表（这里耗费的时候取决于语法文件的规模）
- 想在分析过程中输出丰富的调试信息，使用：yacc.parse(debug=1)
- yacc.yacc()方法会返回分析器对象，如果你想在一个程序中支持多个分析器：

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

注意：yacc.parse()方法只绑定到最新创建的分析器对象上。

- 由于生成生成LALR分析表相对开销较大，先前生成的分析表会被缓存和重用。判断是否重新生成的依据是对所有的语法规则和优先级规则进行MD5校验，只有不匹配时才会重新生成。生成分析表是合理有效的办法，即使是面对上百个规则和状态的语法。对于复杂的编程语言，像C语言，在一些慢的机器上生成分析表可能要花费30-60秒，请耐心。
- 由于LR分析过程是基于分析表的，分析器的性能很大程度上取决于语法的规模。最大的瓶颈可能是词法分析器和语法规则的复杂度。

## 7 多个语法和词法分析器

在高级的分析器程序中，你可能同时需要多个语法和词法分析器。
依照规则行事不会有问题。不过，你需要小心确定所有东西都正确的绑定(hooked up)了。首先，保证将lex()和yacc()返回的对象保存起来：

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

接着，在解析时，确保给parse()方法一个正确的lexer引用：

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

如果遗漏这一步，分析器会使用最新创建的lexer对象，这可能不是你希望的。
词法器和语法器的方法中也可以访问这些对象。在词法器中，标记的lexer属性指代的是当前触发规则的词法器对象：

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

在语法器中，lexer和parser属性指代的是对应的词法器对象和语法器对象

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

如果有必要，lexer对象和parser对象都可以附加其他属性。例如，你想要有不同的解析器状态，可以为为parser对象附加更多的属性，并在后面用到它们。

## 8 使用Python的优化模式

由于PLY从文档字串中获取信息，语法解析和词法分析信息必须通过正常模式下的Python解释器得到（不带有-O或者-OO选项）。不过，如果你像这样指定optimize模式：

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

PLY可以在下次执行，在Python的优化模式下执行。但你必须确保第一次执行是在Python的正常模式下进行，一旦词法分析表和语法分析表生成一次后，在Python优化模式下执行，PLY会使用生成好的分析表而不再需要文档字串。
注意：在优化模式下执行PLY会禁用很多错误检查机制。你应该只在程序稳定后，不再需要调试的情况下这样做。使用优化模式的目的应该是大幅减少你的编译器的启动时间（万事俱备只欠东风时）

## 9 高级调试

调试一个编译器不是件容易的事情。PLY提供了一些高级的调试能力，这是通过Python的logging模块实现的，下面两节介绍这一主题：

### 9.1 调试lex()和yacc()命令

lex()和yacc()命令都有调试模式，可以通过debug标识实现：

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

正常情况下，调试不仅输出标准错误，对于yacc()，还会给出parser.out文件。这些输出可以通过提供logging对象来精细的控制。下面这个例子增加了对调试信息来源的输出：

```
# 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)
```

如果你提供一个自定义的logger，大量的调试信息可以通过分级来控制。典型的是将调试信息分为DEBUG,INFO,或者WARNING三个级别。
PLY的错误和警告信息通过日志接口提供，可以从errorlog参数中传入日志对象

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

如果想完全过滤掉警告信息，你除了可以使用带级别过滤功能的日志对象，也可以使用lex和yacc模块都内建的Nulllogger对象。例如：

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

### 9.2 运行时调试

为分析器指定debug选项，可以激活语法分析器的运行时调试功能。这个选项可以是整数（表示对调试功能是开还是关），也可以是logger对象。例如：

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

如果传入日志对象的话，你可以使用其级别过滤功能来控制内容的输出。INFO级别用来产生归约信息；DEBUG级别会显示分析栈的信息、移进的标记和其他详细信息。ERROR级别显示分析过程中的错误相关信息。
对于每个复杂的问题，你应该用日志对象，以便输出重定向到文件中，进而方便在执行结束后检查。

## 10 如何继续

PLY分发包中的example目录包含几个简单的示例。对于理论性的东西以及LR分析发的实现细节，应当从编译器相关的书籍中学习。

# 6.5 改变起始符号

默认情况下，在yacc中的第一条规则是起始语法规则（顶层规则）。可以用start标识来改变这种行为：

```
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'
...
```

用start标识有助于在调试的时候将大型的语法规则分成小部分来分析。也可把start符号作为yacc的参数：

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

# 6.6 处理二义文法

上面例子中，对表达式的文法描述用一种特别的形式规避了二义文法。然而，在很多情况下，这样的特殊文法很难写，或者很别扭。一个更为自然和舒服的语法表达应该是这样的：

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

不幸的是，这样的文法是存在二义性的。举个例子，如果你要解析字符串”3 * 4 + 5”，操作符如何分组并没有指明，究竟是表示”(3 * 4) + 5”还是”3 * (4 + 5)”呢？
如果在yacc.py中存在二义文法，会输出”移进归约冲突”或者”归约归约冲突”。在分析器无法确定是将下一个符号移进栈还是将当前栈中的符号归约时会产生移进归约冲突。例如，对于”3 * 4 + 5”，分析器内部栈是这样工作的：

```
Step Symbol Stack           Input Tokens            Action
---- ---------------------  ---------------------   ----------------------------
----
1    $                                3 * 4 + 5$    Shift 3
2    $ 3                                * 4 + 5$    Reduce : expression : NUMBE
R
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 ????
```

在这个例子中，当分析器来到第6步的时候，有两种选择：一是按照expr : expr * expr归约，一是将标记’ +’ 继续移进栈。两种选择对于上面的上下文无关文法而言都是合法的。
默认情况下，所有的移进归约冲突会倾向于使用移进来处理。因此，对于上面的例子，分析器总是会将’ +’ 进栈，而不是做归约。虽然在很多情况下，这个策略是合适的（像”if-then”和”if-then-else” ），但这对于算术表达式是不够的。事实上，对于上面的例子，将’ +’ 进栈是完全错误的，应当先将expr * expr归约，因为乘法的优先级要高于加法。
为了解决二义文法，尤其是对表达式文法，yacc.py允许为标记单独指定优先级和结合性。需要像下面这样增加一个precedence变量：

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

```
    ('left', 'TIMES', 'DIVIDE'),
)
```

这样的定义说明PLUS/MINUS标记具有相同的优先级和左结合性，TIMES/DIVIDE具有相同的优先级和左结合性。在precedence声明中，标记的优先级从低到高。因此，这个声明表明TIMES/DIVIDE（他们较晚加入precedence）的优先级高于PLUS/MINUS。
由于为标记添加了数字表示的优先级和结合性的属性，所以，对于上面的例子，将会得到：

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

随后这些值被附加到语法规则的优先级和结合性属性上，这些值由最右边的终结符的优先级和结合性决定：

```
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 spe
cified)
           | NUMBER                                     # level = None (not spe
cified)
```

当出现移进归约冲突时，分析器生成器根据下面的规则解决二义文法：

1. 如果当前的标记的优先级高于栈顶规则的优先级，移进当前标记
2. 如果栈顶规则的优先级更高，进行归约
3. 如果当前的标记与栈顶规则的优先级相同，如果标记是左结合的，则归约，否则，如果是右结合的则移进
4. 如果没有优先级可以参考，默认对于移进归约冲突执行移进

比如，当解析到”expression PLUS expression”这个语法时，下一个标记是TIMES，此时将执行移进，因为TIMES具有比PLUS更高的优先级；当解析到”expression TIMES expression”，下一个标记是PLUS，此时将执行归约，因为PLUS的优先级低于TIMES。
如果在使用前三种技术解决已经归约冲突后，yacc.py将不会报告语法中的冲突或者错误（不过，会在parser.out这个调试文件中输出一些信息）
使用precedence指定优先级的技术会带来一个问题，有时运算符的优先级需要基于上下文。例如，考虑”3 + 4 * -5”中的一元的’ - ’。数学上讲，一元运算符应当拥有较高的优先级。然而，在我们的precedence定义中，MINUS的优先级却低于TIMES。为了解决这个问题，precedene规则中可以包含”虚拟标记”：

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

在语法文件中，我们可以这么表示一元算符：

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

在这个例子中，%prec UMINUS覆盖了默认的优先级（MINUS的优先级），将UMINUS指代的优先级应用在该语法规则上。
起初，UMINUS标记的例子会让人感到困惑。UMINUS既不是输入的标记也不是语法规则，你应当将其看成precedence表中的特殊的占位符。当你使用%prec宏时，你是在告诉yacc，你希望表达式使用这个占位符所表示的优先级，而不是正常的优先级。
还可以在precedence表中指定”非关联”。这表明你不希望链式运算符。比如，假如你希望支持比较运算符’<’和’>’，但是你不希望支持 `a < b < c` ，只要简单指定规则如下：

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

此时，当输入形如 a < b < c时，将产生语法错误，却不影响形如 a < b 的表达式。
对于给定的符号集，存在多种语法规则可以匹配时会产生归约/归约冲突。这样的冲突往往很严重，而且总是通过匹配最早出现的语法规则来解决。归约/归约冲突几乎总是相同的符号集合具有不同的规则可以匹配，而在这一点上无法抉择，比如：

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

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

这个例子中，对于下面这两条规则将产生归约/归约冲突：

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

比如，对于”a = 5”，分析器不知道应当按照assignment : ID EQUALS NUMBER归约，还是先将5归约成expression，再归约成assignment : ID EQUALS expression。
应当指出的是，只是简单的查看语法规则是很难减少归约/归约冲突。如果出现归约/归约冲突，yacc()会帮助打印出警告信息：

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

上面的信息标识出了冲突的两条规则，但是，并无法指出究竟在什么情况下会出现这样的状态。想要发现问题，你可能需要结合语法规则和 `parser.out` 调试文件的内容。

# 6.7 parser.out调试文件

使用LR分析算法跟踪移进/归约冲突和归约/归约冲突是件乐在其中的事。为了辅助调试，yacc.py在生成分析表时会创建出一个调试文件叫parser.out：

```
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
```

文件中出现的不同状态，代表了有效输入标记的所有可能的组合，这是依据文法规则得到的。当得到输入标记时，分析器将构造一个栈，并找到匹配的规则。每个状态跟踪了当前输入进行到语法规则中的哪个位置，在每个规则中，‘.’表示当前分析到规则的哪个位置，而且，对于在当前状态下，输入的每个有效标记导致的动作也被罗列出来。当出现移进/归约或归约/归约冲突时，被忽略的规则前面会添加!，就像这样：

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

通过查看这些规则并结合一些实例，通常能够找到大部分冲突的根源。应该强调的是，不是所有的移进归约冲突都是不好的，想要确定解决方法是否正确，唯一的办法就是查看parser.out。

# 6.8 处理语法错误

如果你创建的分析器用于产品，处理语法错误是很重要的。一般而言，你不希望分析器在遇到错误的时候就抛出异常并终止，相反，你需要它报告错误，尽可能的恢复并继续分析，一次性的将输入中所有的错误报告给用户。这是一些已知语言编译器的标准行为，例如C,C++,Java。在PLY中，在语法分析过程中出现错误，错误会被立即检测到（分析器不会继续读取源文件中错误点后面的标记）。然而，这时，分析器会进入恢复模式，这个模式能够用来尝试继续向下分析。LR分析器的错误恢复是个理论与技巧兼备的问题，yacc.py提供的错误机制与Unix下的yacc类似，所以你可以从诸如O’ Reilly出版的《Lex and yacc》的书中找到更多的细节。
当错误发生时，yacc.py按照如下步骤进行：

1. 第一次错误产生时，用户定义的p_error()方法会被调用，出错的标记会作为参数传入；如果错误是因为到达文件结尾造成的，传入的参数将为None。随后，分析器进入到“错误恢复”模式，该模式下不会在产生 `p_error()` 调用，直到它成功的移进3个标记，然后回归到正常模式。
2. 如果在p_error()中没有指定恢复动作的话，这个导致错误的标记会被替换成一个特殊的error标记。
3. 如果导致错误的标记已经是error的话，原先的栈顶的标记将被移除。
4. 如果整个分析栈被放弃，分析器会进入重置状态，并从他的初始状态开始分析。
5. 如果此时的语法规则接受error标记，error标记会移进栈。
6. 如果当前栈顶是error标记，之后的标记将被忽略，直到有标记能够导致error的归约。

## 6.8.1 根据error规则恢复和再同步

最佳的处理语法错误的做法是在语法规则中包含error标记。例如，假设你的语言有一个关于print的语句的语法规则：

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

为了处理可能的错误表达式，你可以添加一条额外的语法规则：

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

这样（expr错误时），error标记会匹配任意多个分号之前的标记（分号是 `SEMI` 指代的字符）。一旦找到分号，规则将被匹配，这样error标记就被归约了。
这种类型的恢复有时称为”分析器再同步”。error标记扮演了表示所有错误标记的通配符的角色，而紧随其后的标记扮演了同步标记的角色。

重要的一个说明是，通常error不会作为语法规则的最后一个标记，像这样：

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

这是因为，第一个导致错误的标记会使得该规则立刻归约，进而使得在后面还有错误标记的情况下，恢复变得困难。

### 6.8.2 悲观恢复模式

另一个错误恢复方法是采用“悲观模式”：该模式下，开始放弃剩余的标记，直到能够达到一个合适的恢复机会。
悲观恢复模式都是在p_error()方法中做到的。例如，这个方法在开始丢弃标记后，直到找到闭合的’}’，才重置分析器到初始化状态：

```
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()
```

下面这个方法简单的抛弃错误的标记，并告知分析器错误被接受了：

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

在 `p_error()` 方法中，有三个可用的方法来控制分析器的行为：

- `yacc.errok()` 这个方法将分析器从恢复模式切换回正常模式。这会使得不会产生error标记，并重置内部的error计数器，而且下一个语法错误会再次产生p_error()调用
- `yacc.token()` 这个方法用于得到下一个标记
- `yacc.restart()` 这个方法抛弃当前整个分析栈，并重置分析器为起始状态

注意：这三个方法只能在 `p_error()` 中使用，不能用在其他任何地方。
p_error()方法也可以返回标记，这样能够控制将哪个标记作为下一个标记返回给分析器。这对于需要同步一些特殊标记的时候有用，比如：

```
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
```

### 6.8.3 从产生式中抛出错误

如果有需要的话，产生式规则可以主动的使分析器进入恢复模式。这是通过抛出 `SyntacError` 异常做到的：

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

raise SyntaxError错误的效果就如同当前的标记是错误标记一样。因此，当你这么做的话，最后一个标记将被弹出栈，当前的下一个标记将是error标记，分析器进入恢复模式，试图归约满足error标记的规则。此后的步骤与检测到语法错误的情况是完全一样的，p_error()也会被调用。

手动设置错误有个重要的方面，就是p_error()方法在这种情况下不会调用。如果你希望记录错误，确保在抛出SyntaxError错误的产生式中实现。

注：这个功能是为了模仿yacc中的 `YYERROR` 宏的行为

### 6.8.4 错误恢复总结

对于通常的语言，使用error规则和再同步标记可能是最合理的手段。这是因为你可以将语法设计成在一个相对容易恢复和继续分析的点捕获错误。悲观恢复模式只在一些十分特殊的应用中有用，这些应用往往需要丢弃掉大量输入，再寻找合理的同步点。

# 6.9 行号和位置的跟踪

位置跟踪通常是个设计编译器时的技巧性玩意儿。默认情况下，PLY跟踪所有标记的行号和位置，这些信息可以这样得到：

- p.lineno(num)返回第num个符号的行号
- p.lexpos(num)返回第num个符号的词法位置偏移

例如：

```
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
```

注意：lexspan()方法只会返回的结束位置是最后一个符号的起始位置。

虽然，PLY对所有符号的行号和位置的跟踪很管用，但经常是不必要的。例如，你仅仅是在错误信息中使用行号，你通常可以仅仅使用关键标记的信息，比如：

```
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)
```

类似的，为了改善性能，你可以有选择性的将行号信息在必要的时候进行传递，这是通过p.set_lineno()实现的，例如：

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

对于已经完成分析的规则，PLY不会保留行号信息，如果你是在构建抽象语法树而且需要行号，你应该确保行号保留在树上。

# 6.10 构造抽象语法树

yacc.py没有构造抽像语法树的特殊方法。不过，你可以自己很简单的构造出来。

一个最为简单的构造方法是为每个语法规则创建元组或者字典，并传递它们。有很多中可行的方案，下面是一个例子：

```
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])
```

另一种方法可以是为不同的抽象树节点创建一系列的数据结构，并赋值给p[0]：

```
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])
```

这种方式的好处是在处理复杂语义时比较简单：类型检查、代码生成、以及其他针对树节点的功能。

为了简化树的遍历，可以创建一个通用的树节点结构，例如：

```
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 嵌入式动作

yacc使用的分析技术只允许在规则规约后执行动作。假设有如下规则：

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

方法只会在符号A,B,C和D都完成后才能执行。可是有的时候，在中间阶段执行一小段代码是有用的。假如，你想在A完成后立即执行一些动作，像下面这样用空规则：

```
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
```

在这个例子中，空规则seen_A将在A移进分析栈后立即执行。p[-1]指代的是在分析栈上紧跟在seen_A左侧的符号。在这个例子中，是A符号。像其他普通的规则一样，在嵌入式行为中也可以通过为p[0]赋值来返回某些值。使用嵌入式动作可能会导致移进归约冲突，比如，下面的语法是没有冲突的：

```
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"
```

可是，如果像这样插入一个嵌入式动作：

```
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 :"
```

会产生移进归约冲，只是由于对于两个规则abcd和abcx中的C，分析器既可以根据abcd规则移进，也可以根据abcx规则先将空的seen_AB归约。

嵌入动作的一般用于分析以外的控制，比如为本地变量定义作用于。对于C语言：

```
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)
    ...
```

在这个例子中，new_scope作为嵌入式行为，在左大括号{之后立即执行。可以是调正内部符号表或者其他方面。statements_block一完成，代码可能会撤销在嵌入动作时的操作（比如，pop_scope()）

# 6.12 Yacc的其他

- 默认的分析方法是LALR，使用SLR请像这样运行 yacc()：yacc.yacc(method=”SLR”)注意：LRLR生成的分析表大约要比SLR的大两倍。解析的性能没有本质的区别，因为代码是一样的。由于LALR能力稍强，所以更多的用于复杂的语法。
- 默认情况下，yacc.py依赖lex.py产生的标记。不过，可以用一个等价的词法标记生成器代替：yacc.parse(lexer=x) 这个例子中，x必须是一个Lexer对象，至少拥有x.token()方法用来获取标记。如果将输入字串提供给yacc.parse()，lexer还必须具有x.input()方法。
- 默认情况下，yacc在调试模式下生成分析表（会生成parser.out文件和其他东西），使用yacc.yacc(debug=0)禁用调试模式。
- 改变parsetab.py的文件名：yacc.yacc(tabmodule=”foo”)
- 改变parsetab.py的生成目录：yacc.yacc(tabmodule=”foo”,outputdir=”somedirectory”)
- 不生成分析表：yacc.yacc(write_tables=0)。注意：如果禁用分析表生成，yacc()将在每次运行的时候重新构建分析表（这里耗费的时候取决于语法文件的规模）
- 想在分析过程中输出丰富的调试信息，使用：yacc.parse(debug=1)
- yacc.yacc()方法会返回分析器对象，如果你想在一个程序中支持多个分析器：

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

注意：yacc.parse()方法只绑定到最新创建的分析器对象上。

- 由于生成生成LALR分析表相对开销较大，先前生成的分析表会被缓存和重用。判断是否重新生成的依据是对所有的语法规则和优先级规则进行MD5校验，只有不匹配时才会重新生成。生成分析表是合理有效的办法，即使是面对上百个规则和状态的语法。对于复杂的编程语言，像C语言，在一些慢的机器上生成分析表可能要花费30-60秒，请耐心。
- 由于LR分析过程是基于分析表的，分析器的性能很大程度上取决于语法的规模。最大的瓶颈可能是词法分析器和语法规则的复杂度。

# 7 多个语法和词法分析器

在高级的分析器程序中，你可能同时需要多个语法和词法分析器。

依照规则行事不会有问题。不过，你需要小心确定所有东西都正确的绑定(hooked up)了。首先，保证将lex()和yacc()返回的对象保存起来：

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

接着，在解析时，确保给parse()方法一个正确的lexer引用：

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

如果遗漏这一步，分析器会使用最新创建的lexer对象，这可能不是你希望的。

词法器和语法器的方法中也可以访问这些对象。在词法器中，标记的lexer属性指代的是当前触发规则的词法器对象：

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

在语法器中，lexer和parser属性指代的是对应的词法器对象和语法器对象

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

如果有必要，lexer对象和parser对象都可以附加其他属性。例如，你想要有不同的解析器状态，可以为为parser对象附加更多的属性，并在后面用到它们。

# 8 使用Python的优化模式

由于PLY从文档字串中获取信息，语法解析和词法分析信息必须通过正常模式下的Python解释器得到（不带有-O或者-OO选项）。不过，如果你像这样指定optimize模式：

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

PLY可以在下次执行，在Python的优化模式下执行。但你必须确保第一次执行是在Python的正常模式下进行，一旦词法分析表和语法分析表生成一次后，在Python优化模式下执行，PLY会使用生成好的分析表而不再需要文档字串。

注意：在优化模式下执行PLY会禁用很多错误检查机制。你应该只在程序稳定后，不再需要调试的情况下这样做。使用优化模式的目的应该是大幅减少你的编译器的启动时间（万事俱备只欠东风时）

# 9 高级调试

调试一个编译器不是件容易的事情。PLY提供了一些高级的调试能力，这是通过Python的logging模块实现的，下面两节介绍这一主题：

# 9.1 调试lex()和yacc()命令

lex()和yacc()命令都有调试模式，可以通过debug标识实现：

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

正常情况下，调试不仅输出标准错误，对于yacc()，还会给出parser.out文件。这些输出可以通过提供logging对象来精细的控制。下面这个例子增加了对调试信息来源的输出：

```
# 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)
```

如果你提供一个自定义的logger，大量的调试信息可以通过分级来控制。典型的是将调试信息分为DEBUG,INFO,或者WARNING三个级别。

PLY的错误和警告信息通过日志接口提供，可以从errorlog参数中传入日志对象

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

如果想完全过滤掉警告信息，你除了可以使用带级别过滤功能的日志对象，也可以使用lex和yacc模块都内建的Nulllogger对象。例如：

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

# 9.2 运行时调试

为分析器指定debug选项，可以激活语法分析器的运行时调试功能。这个选项可以是整数（表示对调试功能是开还是关），也可以是logger对象。例如：

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

如果传入日志对象的话，你可以使用其级别过滤功能来控制内容的输出。INFO级别用来产生归约信息；DEBUG级别会显示分析栈的信息、移进的标记和其他详细信息。ERROR级别显示分析过程中的错误相关信息。
对于每个复杂的问题，你应该用日志对象，以便输出重定向到文件中，进而方便在执行结束后检查。

# 10 如何继续

PLY分发包中的example目录包含几个简单的示例。对于理论性的东西以及LR分析发的实现细节，应当从编译器相关的书籍中学习。