PLY Package

- PLY consists of two Python modules

```
ply.lex
ply.yacc
```

- You simply import the modules to use them
- However, PLY is not a code generator

ply.lex

- A module for writing lexers
- Tokens specified using regular expressions
- Provides functions for reading input text
- An annotated example follows...

ply.lex example

```
import ply.lex as lex
tokens = [ 'NAME','NUMBER','PLUS','MINUS','TIMES',
           'DIVIDE', EQUALS' ]
t_ignore = ' \t'
t_PLUS   = r'\+'
t_MINUS  = r'-'
t_TIMES  = r'\*'
t_DIVIDE = r'/'
t_EQUALS = r'='
t_NAME   = r'[a-zA-Z_][a-zA-Z0-9_]*'

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

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

ply.lex example

```
import ply.lex as lex
tokens = [ 'NAME','NUMBER','PLUS','MINUS','TIMES',
           'DIVIDE', EQUALS' ]
t_ignore = ' \t'
t_PLUS   = r'\+'
t_MINUS  = r'-'
t_TIMES  = r'\*'
t_DIVIDE = r'/'
t_EQUALS = r'='
t_NAME   = r'[a-zA-Z_][a-zA-Z0-9_]*'

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

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

tokens list specifies
all of the possible tokens

ply.lex example

```
import ply.lex as lex
tokens = [ 'NAME','NUMBER','PLUS','MINUS','TIMES',
           'DIVIDE', EQUALS' ]
t_ignore = ' \t'
t_PLUS   = r'\+'
t_MINUS  = r'-'
t_TIMES  = r'\*'
t_DIVIDE = r'/'
t_EQUALS = r'='
t_NAME   = r'[a-zA-Z_][a-zA-Z0-9_]*'

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

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

Each token has a matching declaration of the form t_*TOKNAME*

ply.lex example

```
import ply.lex as lex
tokens = [ 'NAME','NUMBER','PLUS','MINUS','TIMES',
           'DIVIDE', EQUALS'
t_ignore = ' \t'
t_PLUS   = r'\+'
t_MINUS  = r'-'
t_TIMES  = r'\*'
t_DIVIDE = r'/'
t_EQUALS = r'='
t_NAME   = r'[a-zA-Z_][a-zA-Z0-9_]*'

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

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

These names must match

ply.lex example

```
import ply.lex as lex
tokens = [ 'NAME','NUMBER','PLUS','MINUS','TIMES',
           'DIVIDE', EQUALS' ]
t_ignore = ' \t'
t_PLUS   = r'\+'
t_MINUS  = r'-'
t_TIMES  = r'\*'
t_DIVIDE = r'/'
t_EQUALS = r'='
t_NAME   = r'[a-zA-Z_][a-zA-Z0-9_]*'

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

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

Tokens are defined by regular expressions

ply.lex example

```
import ply.lex as lex
tokens = [ 'NAME','NUMBER','PLUS','MINUS','TIMES',
           'DIVIDE', EQUALS' ]
t_ignore = ' \t'
t_PLUS   = r'\+'
t_MINUS  = r'-'
t_TIMES  = r'\*'
t_DIVIDE = r'/'
t_EQUALS = r'='
t_NAME   = r'[a-zA-Z_][a-zA-Z0-9_]*'

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

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

For simple tokens, strings are used.

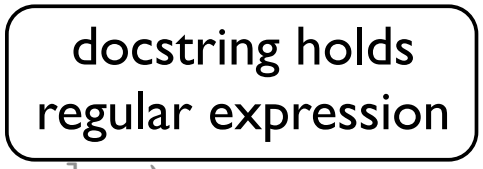
ply.lex example

```
import ply.lex as lex
tokens = [ 'NAME','NUMBER','PLUS','MINUS','TIMES',
           'DIVIDE', EQUALS' ]
t_ignore = ' \t'
t_PLUS   = r'\+'
t_MINUS  = r'-'
t_TIMES  = r'\*'
t_DIVIDE = r'/'
t_EQUALS = r'='
t_NAME   = r'[a-zA-Z_]

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

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

Functions are used when special action code must execute

ply.lex example

```
import ply.lex as lex
tokens = [ 'NAME','NUMBER','PLUS','MINUS','TIMES',
           'DIVIDE', EQUALS' ]
t_ignore = ' \t'
t_PLUS   = r'\+'
t_MINUS  = r'-'
t_TIMES  = r'\*'
t_DIVIDE = r'/'
t_EQUALS = r'='
t_NAME   = r'[a-zA-Z_][a-zA-Z0-9_]*'

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

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



ply.lex example

```
import ply.lex as lex
tokens = [ 'NAME','NUM
           'DIVIDE', E
t_ignore = ' \t'
t_PLUS   = r'\+'
t_MINUS  = r'-'
t_TIMES  = r'\*'
t_DIVIDE = r'/'
t_EQUALS = r'='
t_NAME   = r'[a-zA-Z_][a-zA-Z0-9_]*'

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

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

Specifies ignored characters between tokens (usually whitespace)

ply.lex example

```
import ply.lex as lex
tokens = [ 'NAME','NUMBER','PLUS','MINUS','TIMES',
           'DIVIDE', EQUALS' ]
t_ignore = ' \t'
t_PLUS   = r'\+'
t_MINUS  = r'-'
t_TIMES  = r'\*'
t_DIVIDE = r'/'
t_EQUALS = r'='
t_NAME   = r'[a-zA-Z_][a-zA-Z0-9_]*'

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

lex.lex()
```

Builds the lexer by creating a master regular expression

ply.lex example

```
import ply.lex as lex
tokens = [ 'NAME','NUMBER','PLUS','MINUS','TIMES',
           'DIVIDE', EQUALS' ]

t_ignore = ' \t'
t_PLUS   = r'\+'
t_MINUS  = r'-'
t_TIMES  = r'\*'
t_DIVIDE = r'/'
t_EQUALS = r'='
t_NAME   = r'[a-zA-Z_][a-zA-Z0-9_]*'

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

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

Introspection used to examine contents of calling module.

ply.lex example

```
import ply.lex as lex
tokens = [ 'NAME','NUMBER','PLUS','MINUS','TIMES',
           'DIVIDE', EQUALS' ]

t_ignore = ' \t'
t_PLUS   = r'\+'
t_MINUS  = r'-'
t_TIMES  = r'\*'
t_DIVIDE = r'/'
t_EQUALS = r'='
t_NAME   = r'[a-zA-Z_][a-zA-Z0-9_]*'

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

lex.lex()          # Build
```

Introspection used to examine contents of calling module.

```
__dict__ = {
  'tokens' : [ 'NAME' ...],
  't_ignore' : ' \t',
  't_PLUS' : '\\+',
  ...
  't_NUMBER' : <function ...
}
```

ply.lex use

- Two functions: input() and token()

```
...
lex.lex()          # Build the lexer
...
lex.input("x = 3 * 4 + 5 * 6")
while True:
     tok = lex.token()
     if not tok: break

     # Use token
     ...
```

ply.lex use

- Two functions: input() and token()

```
...
lex.lex()          # Build the lexer
...
lex.input("x = 3 * 4 + 5 * 6")
while True:
     tok = lex.token()
     if not tok: break

     # Use token
     ...
```

input() feeds a string into the lexer

ply.lex use

- Two functions: input() and token()

```
...
lex.lex()          # Build the lexer
...
lex.input("x = 3 * 4 + 5 * 6")
while True:
     tok = lex.token()
     if not tok: break

     # Use token
     ...
```

token() returns the next token or None

ply.lex use

- Two functions: input() and token()

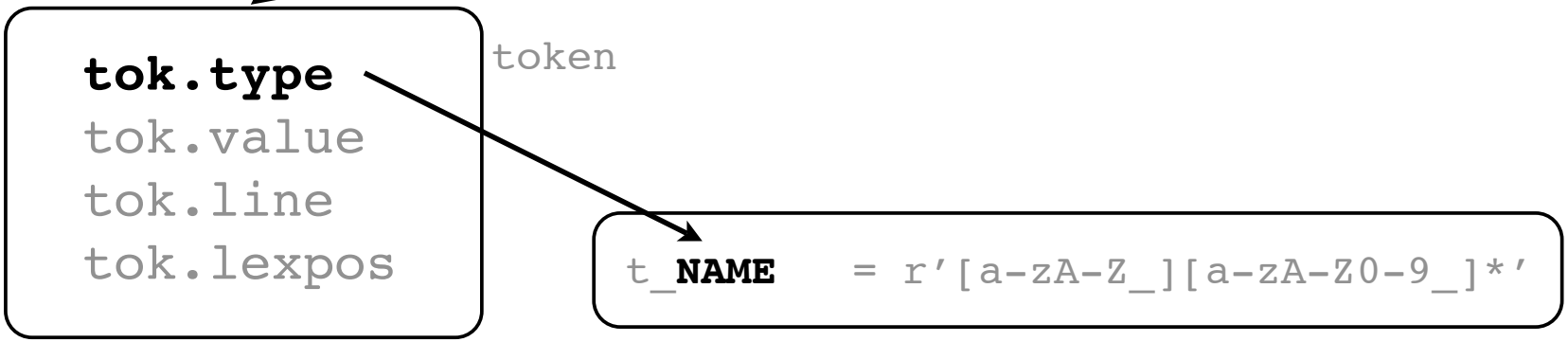
```
...
lex.lex()          # Build the lexer
...
lex.input("x = 3 * 4 + 5 * 6")
while True:
     tok = lex.token()
     if not tok: break

     token
```

```
tok.type
tok.value
tok.line
tok.lexpos
```

ply.lex use

- Two functions: input() and token()

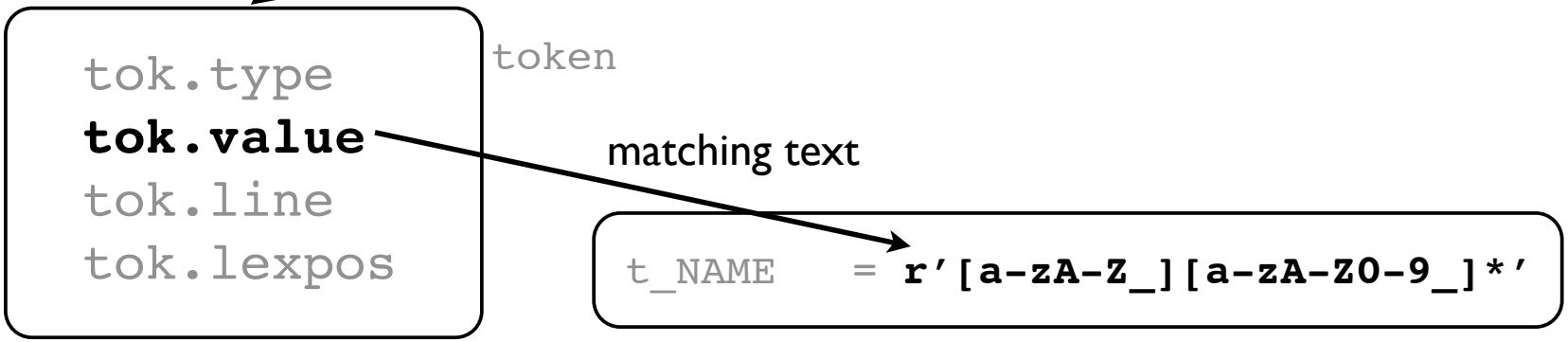
```
...
lex.lex()          # Build the lexer
...
lex.input("x = 3 * 4 + 5 * 6")
while True:
     tok = lex.token()
     if not tok: break
```



ply.lex use

- Two functions: input() and token()

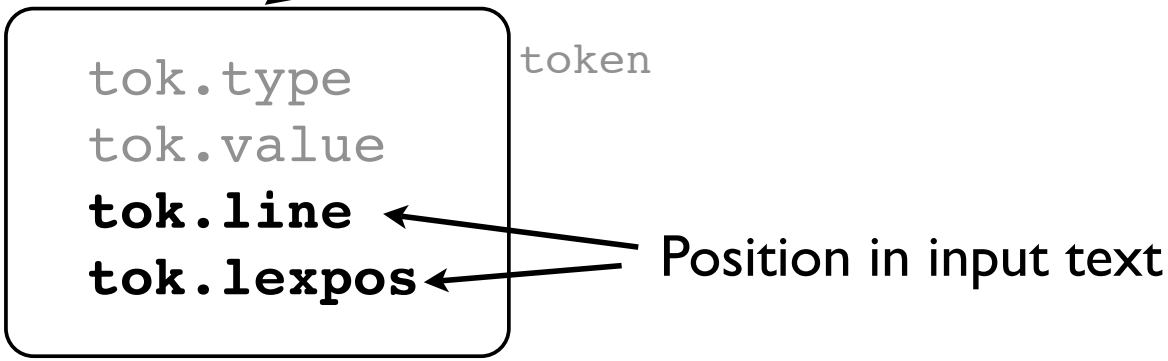
```
...
lex.lex()          # Build the lexer
...
lex.input("x = 3 * 4 + 5 * 6")
while True:
     tok = lex.token()
     if not tok: break
```



ply.lex use

- Two functions: input() and token()

```
...
lex.lex()          # Build the lexer
...
lex.input("x = 3 * 4 + 5 * 6")
while True:
     tok = lex.token()
     if not tok: break
```



ply.lex Commentary

- Normally you don't use the tokenizer directly
- Instead, it's used by the parser module

ply.yacc preliminaries

- ply.yacc is a module for creating a parser
- Assumes you have defined a BNF grammar

```
assign : NAME EQUALS expr
expr   : expr PLUS term
       | expr MINUS term
       | term
term   : term TIMES factor
       | term DIVIDE factor
       | factor
factor : NUMBER
```

ply.yacc example

```
import ply.yacc as yacc
import mylexer              # Import lexer information
tokens = mylexer.tokens     # Need token list

def p_assign(p):
    '''assign : NAME EQUALS expr'''

def p_expr(p):
    '''expr : expr PLUS term
            | expr MINUS term
            | term'''
def p_term(p):
    '''term : term TIMES factor
            | term DIVIDE factor
            | factor'''
def p_factor(p):
    '''factor : NUMBER'''

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

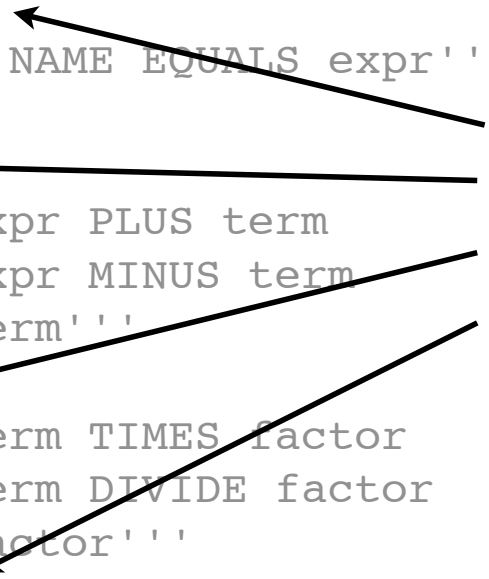
ply.yacc example

```
import ply.yacc as yacc
import mylexer
tokens = mylexer.tokens

def p_assign(p):
    '''assign : NAME EQUALS expr'''

def p_expr(p):
    '''expr : expr PLUS term
            | expr MINUS term
            | term'''
def p_term(p):
    '''term : term TIMES factor
            | term DIVIDE factor
            | factor'''
def p_factor(p):
    '''factor : NUMBER'''

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

token information imported from lexer

ply.yacc example

```
import ply.yacc as yacc
import mylexer              # Import lexer information
tokens = mylexer.tokens     # Need token list

def p_assign(p):
    '''assign : NAME EQUALS expr'''

def p_expr(p):
    '''expr : expr PLUS term
            | expr MINUS term
            | term'''
def p_term(p):
    '''term : term TIMES factor
            | term DIVIDE factor
            | factor'''
def p_factor(p):
    '''factor : NUMBER'''

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

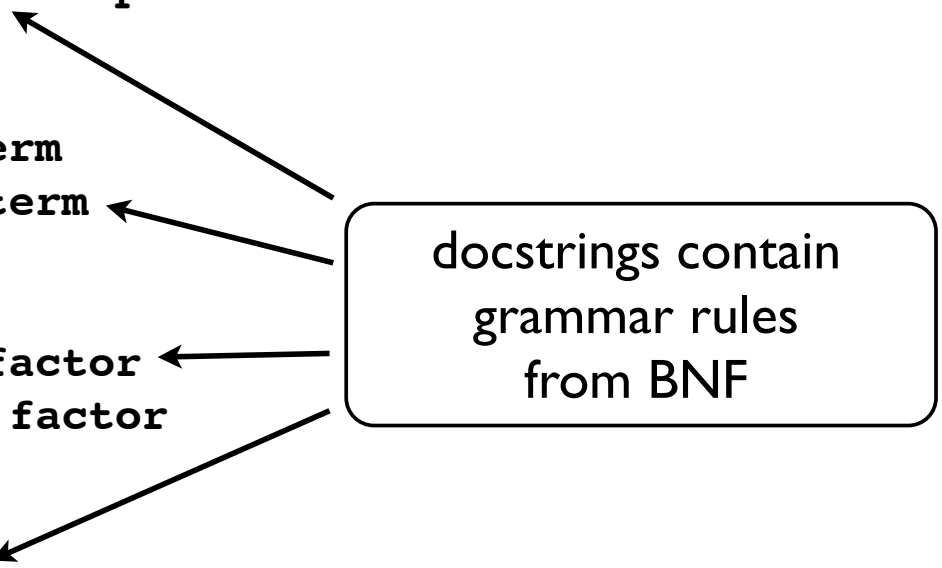


grammar rules encoded as functions with names p_*rulename*

Note: Name doesn't matter as long as it starts with p_

ply.yacc example

```
import ply.yacc as yacc
import mylexer              # Import lexer information
tokens = mylexer.tokens     # Need token list

def p_assign(p):
    '''assign : NAME EQUALS expr'''

def p_expr(p):
    '''expr : expr PLUS term
            | expr MINUS term
            | term'''
def p_term(p):
    '''term : term TIMES factor
            | term DIVIDE factor
            | factor'''
def p_factor(p):
    '''factor : NUMBER'''

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



ply.yacc example

```
import ply.yacc as yacc
import mylexer              # Import lexer information
tokens = mylexer.tokens     # Need token list

def p_assign(p):
    '''assign : NAME EQUALS expr'''

def p_expr(p):
    '''expr : expr PLUS term
            | expr MINUS term
            | term'''
def p_term(p):
    '''term : term TIMES factor
            | term DIVIDE factor
            | factor'''
def p_factor(p):
    '''factor : NUMBER'''

yacc.yacc()
```

Builds the parser using introspection

ply.yacc parsing

- yacc.parse() function

```
yacc.yacc()      # Build the parser
...
data = "x = 3*4+5*6"
yacc.parse(data)   # Parse some text
```

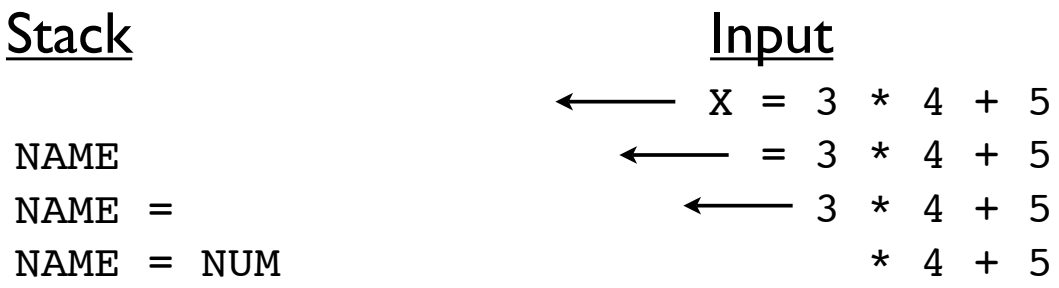
- This feeds data into lexer
- Parses the text and invokes grammar rules

A peek inside

- PLY uses LR-parsing. LALR(1)
- AKA: Shift-reduce parsing
- Widely used parsing technique
- Table driven

General Idea

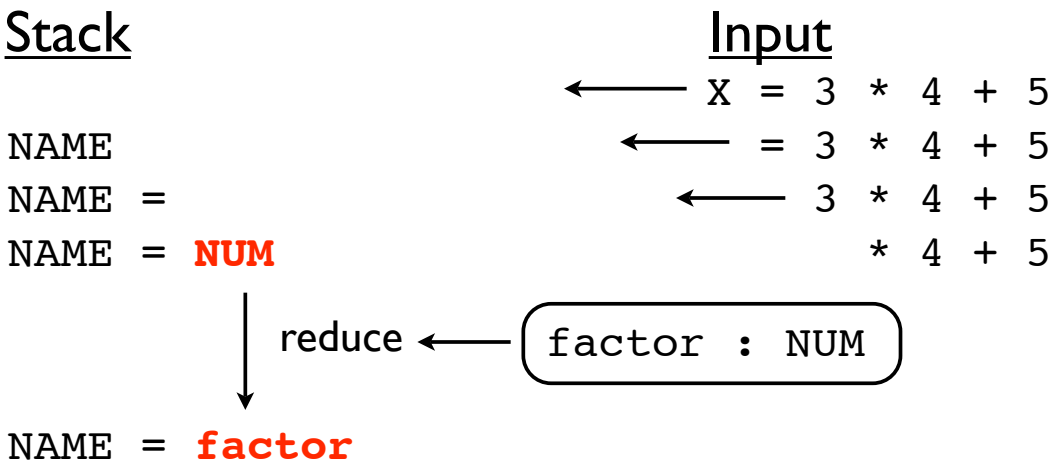
- Input tokens are shifted onto a parsing stack



- This continues until a complete grammar rule appears on the top of the stack

General Idea

- If rules are found, a "reduction" occurs



- RHS of grammar rule replaced with LHS

Rule Functions

- During reduction, rule functions are invoked

```
def p_factor(p):
    'factor : NUMBER'
```

- Parameter p contains grammar symbol values

```
def p_factor(p):
    'factor : NUMBER'
       ↑         ↑
     p[0]      p[1]
```

Using an LR Parser

- Rule functions generally process values on right hand side of grammar rule
- Result is then stored in left hand side
- Results propagate up through the grammar
- Bottom-up parsing

Example: Calculator

```
def p_assign(p):
    '''assign : NAME EQUALS expr'''
    vars[p[1]] = p[3]

def p_expr_plus(p):
    '''expr : expr PLUS term'''
    p[0] = p[1] + p[3]

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

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

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

Example: Parse Tree

```
def p_assign(p):
    '''assign : NAME EQUALS expr'''
    p[0] = ('ASSIGN',p[1],p[3])

def p_expr_plus(p):
    '''expr : expr PLUS term'''
    p[0] = ('+',p[1],p[3])

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

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

def p_factor(p):
    '''factor : NUMBER'''
    p[0] = ('NUM',p[1])
```

Example: Parse Tree

```
>>> t = yacc.parse("x = 3*4 + 5*6")
>>> t
('ASSIGN','x',('+',
                   ('*',('NUM',3),('NUM',4)),
                   ('*',('NUM',5),('NUM',6))
                )
)
>>>
```