CSE 304 Compiler Design
PLY – Python Lex/Yacc

PLY Specification Example

A bit of history:

- Yacc: ~1973. Stephen Johnson (AT&T)
- Lex : ~1974. Eric Schmidt and Mike Lesk(AT&T)
- PLY: 2001
- PLY: Python Lex-Yacc
  - Implementation of lex and yacc for Python by David Beazley:
  - http://www.dabeaz.com/ply/

PLY Specification Example

PLY is not a code generator

- PLY consists of two Python modules
  - ply.lex - A module for writing lexers
    - Tokens specified using regular expressions
    - Provides functions for reading input text
  - ply.yacc - A module for writing grammars
- You simply import the modules to use them
  - The grammar must be in a file

PLY Specification Example

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

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

Functions are used when special action code must execute

Builds the lexer by creating a master regular expression

PLY Specification Example

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

token() returns the next token or None

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

tok.type → t_NAME = r'[a-zA-Z_][a-zA-Z0-9_]*'

tok.value → matching text

tok.line, tok.lexpos: Position in input text

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

token information imported from lexer

grammar rules encoded as functions with names p_*rulename*

docstrings contain grammar rules from BNF

PLY Specification Example

PLY uses LR-parsing LALR(1)

- Shift-reduce parsing
- Table driven
- Input tokens are shifted onto a parsing stack

X = 3 * 4 + 5	->
= 3 * 4 + 5	-> NAME
3 * 4 + 5	-> NAME =
* 4 + 5	-> NAME = NUM

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

reduce	factor : NUM
* 4 + 5	-> NAME = factor

PLY Specification Example

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]

PLY Specification Example

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

PLY does Bottom-up parsing

PLY Calculator Example

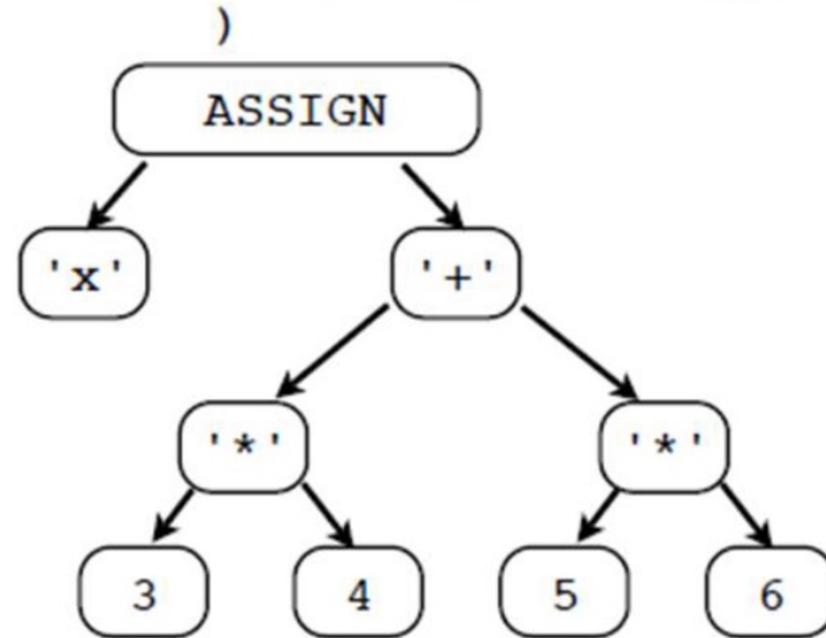
```
def p_assign(p):
    '''assign : NAME EQUALS expr'''
    vars[p[1]] = p[3]

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]
```

PLY Calculator Example

```
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])
```

PLY Calculator Example

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



PLY Precedence Specifiers

Precedence Specifiers

- highest precedence at bottom:

```
precedence = (
        ('left','PLUS','MINUS’),
        ('left','TIMES','DIVIDE’),
        ('nonassoc','UMINUS’),
)
def p_expr_uminus(p):
        'expr : MINUS expr %prec UMINUS’
        p[0] = -p[1]
. . .
```

PLY Best Documentation

Google Mailing list/group:

http://groups.google.com/group/ply-hack