Writing Parsers and Compilers with PLY

Slides by David Beazley
http://www.dabeaz.com

Compiler Design (CS335)

Ayush Tulsyan and Rahul Gupta

Overview

- An introduction to PLY
- Notable PLY features (why use it?)
- Experience writing a compiler/interpreter in Python

Example

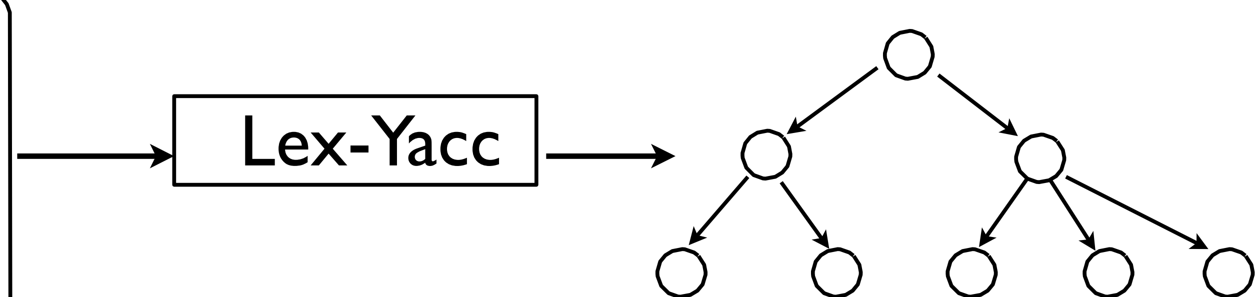
- Parse and generate assembly code

```
/* Compute GCD of two integers */
fun gcd(x:int, y:int)
    g: int;
    begin
       g := y;
       while x > 0 do
           begin
              g := x;
              x := y - (y/x)*x;
              y := g
           end;
       return g
    end
```

Compilers 101

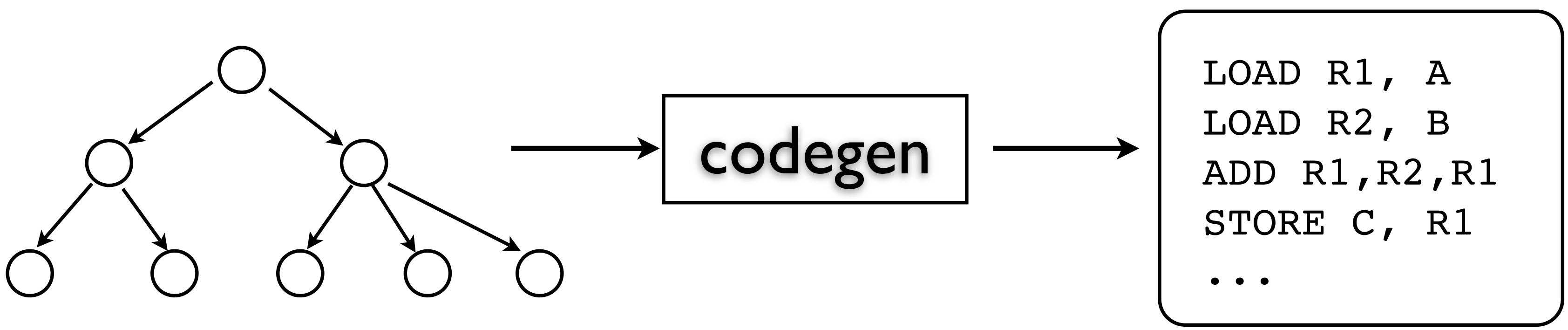
- Compilers have multiple phases
- First phase usually concerns "lexing" and "parsing"
- Read program and create abstract representation

```
/* Compute GCD of two integers */
fun gcd(x:int, y:int)
    g: int;
    begin
       g := y;
       while x > 0 do
            begin
              g := x;
              x := y - (y/x)*x;
              y := g
            end;
       return g
    end
```



Compilers 101

- Code generation phase
- Process the abstract representation
- Produce some kind of output



Commentary

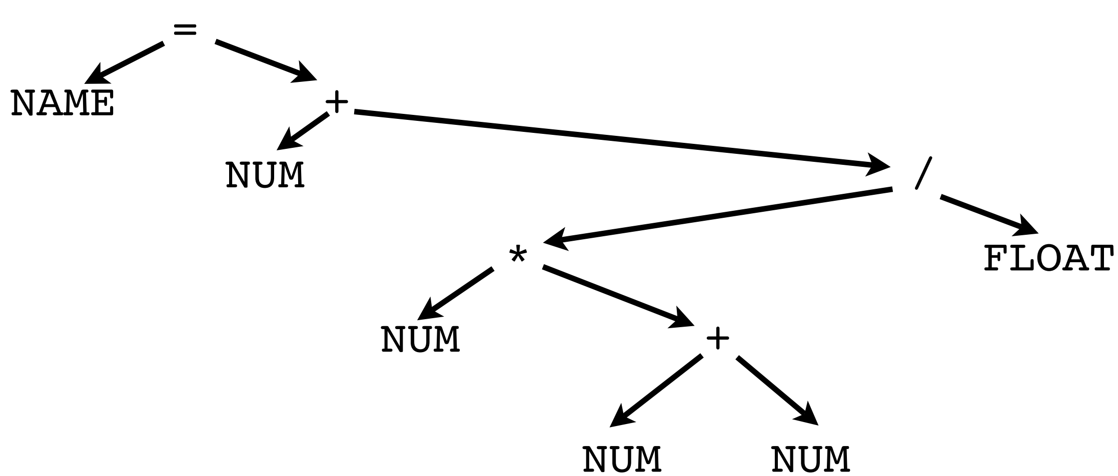
- There are many advanced details
- Most people care about code generation
- Yet, parsing is often the most annoying problem
- A major focus of tool building

Parsing in a Nutshell

- Lexing : Input is split into tokens

```
b = 40 + 20*(2+3)/37.5
NAME = NUM + NUM * ( NUM + NUM ) / FLOAT
```

- Parsing : Applying language grammar rules



Lex & Yacc

- Programming tools for writing parsers
- Lex - Lexical analysis (tokenizing)
- Yacc - Yet Another Compiler Compiler (parsing)

Lex/Yacc Big Picture

lexer.l

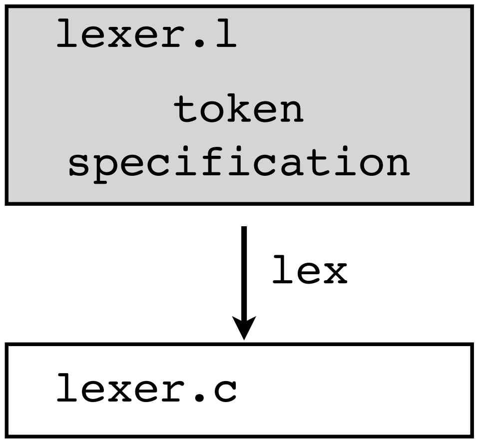
token specification

Lex/Yacc Big Picture

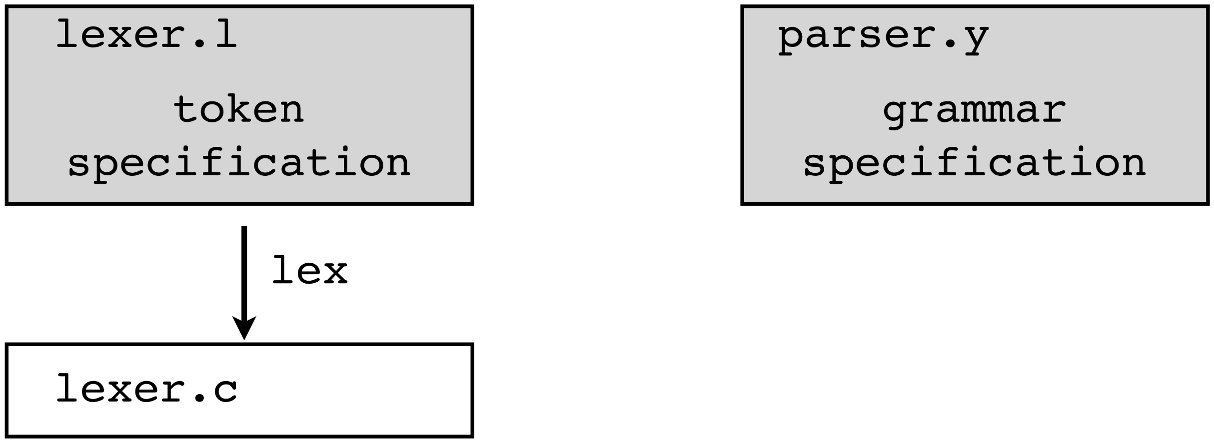
lexer.l

```
/* lexer.l */
%{
#include “header.h”
int lineno = 1;
%}
%%
[ \t]* ;     /* Ignore whitespace */
\n                          { lineno++; }
[0-9]+                      { yylval.val = atoi(yytext);
                              return NUMBER; }
[a-zA-Z_][a-zA-Z0-9_]* { yylval.name = strdup(yytext);
                              return ID; }
\+                          { return PLUS; }
-                           { return MINUS; }
\*                          { return TIMES; }
\/                          { return DIVIDE; }
=                           { return EQUALS; }
%%
```

Lex/Yacc Big Picture



Lex/Yacc Big Picture



Lex/Yacc Big Picture

lexer.l

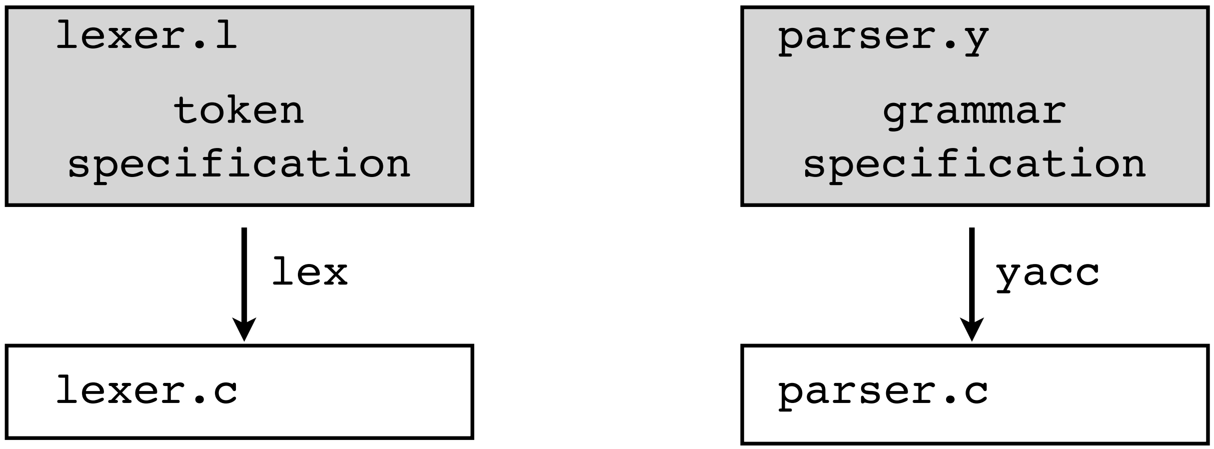
spe

parser.y

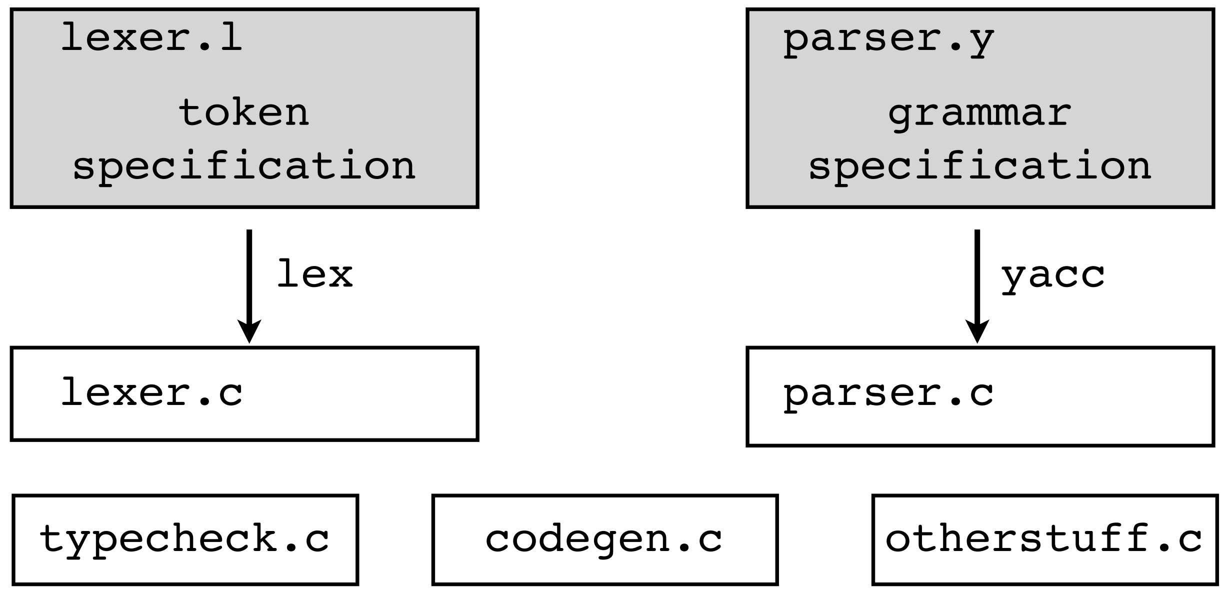
lexe

```
/* parser.y */
%{
#include “header.h”
%}
%union {
   char *name;
   int   val;
}
%token PLUS MINUS TIMES DIVIDE EQUALS
%token<name> ID;
%token<val> NUMBER;
%%
start : ID EQUALS expr;
expr : expr PLUS term
     | expr MINUS term
     | term
     ;
...
```

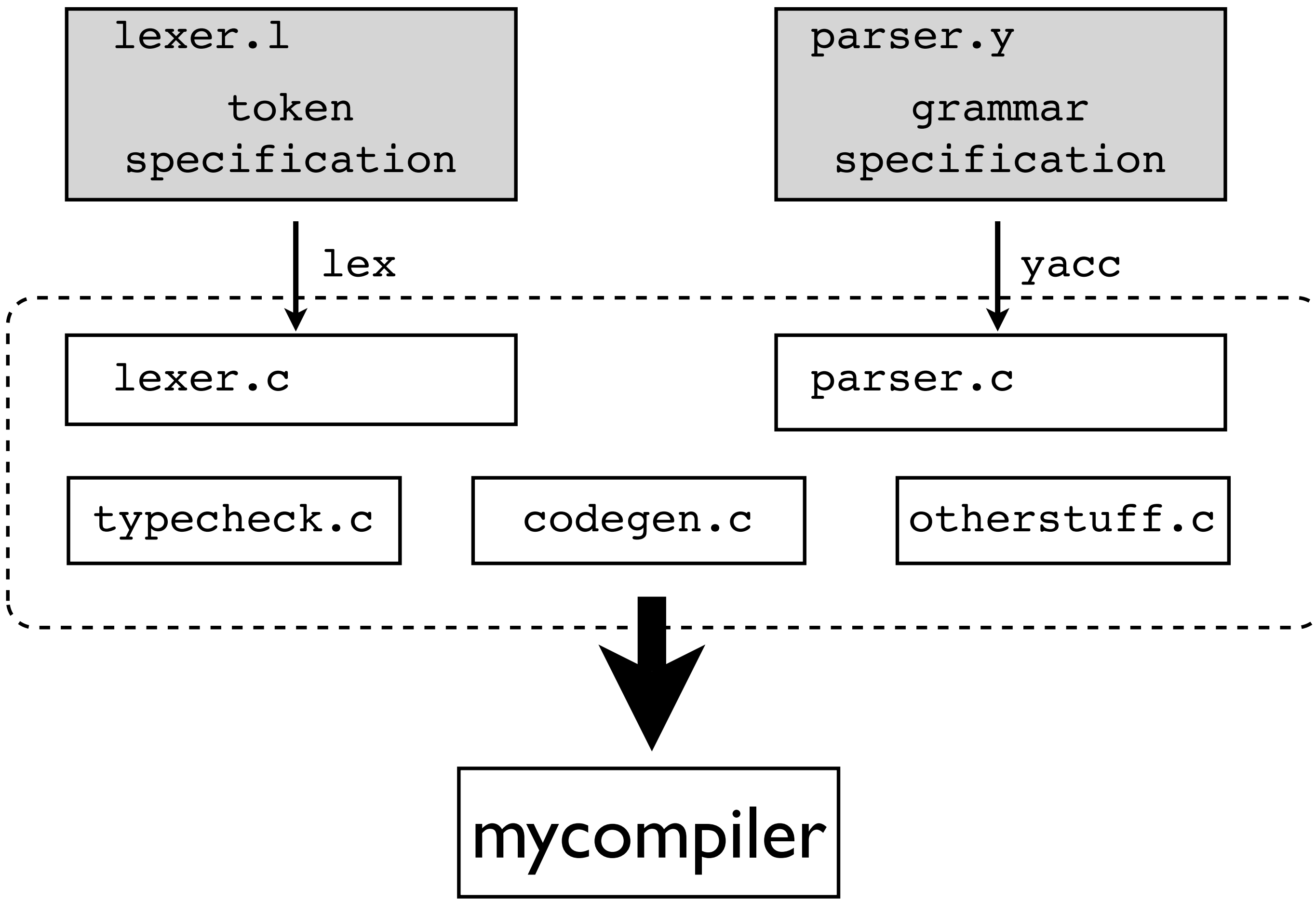
Lex/Yacc Big Picture



Lex/Yacc Big Picture



Lex/Yacc Big Picture



What is PLY?

- PLY = Python Lex-Yacc
- A Python version of the lex/yacc toolset
- Same functionality as lex/yacc
- But a different interface

PLY Package

- PLY consists of two Python modules

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

- You simply import the modules to use them
- However, PLY is <u>not</u> 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
```

docstring holds
regular expression

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
```

token

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

```
t_NAME   = r'[a-zA-Z_][a-zA-Z0-9_]*'
```

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
```

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

token

matching text

```
t_NAME   = r'[a-zA-Z_][a-zA-Z0-9_]*'
```

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
```

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

token

Position in input text

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
```

docstrings contain grammar rules from BNF

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

```
Stack                              Input
                          <---- X = 3 * 4 + 5
NAME                        <----   = 3 * 4 + 5
NAME =                        <----   3 * 4 + 5
NAME = NUM                              * 4 + 5
```

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

General Idea

- If rules are found, a "reduction" occurs

Stack	Input
	X = 3 * 4 + 5
NAME	= 3 * 4 + 5
NAME =	3 * 4 + 5
NAME = **NUM**	* 4 + 5

reduce ← `factor : NUM`

NAME = **factor**	

- RHS of grammar rule replaced with LHS

Precedence Specifiers

- Yacc

```
%left PLUS MINUS
%left TIMES DIVIDE
%nonassoc UMINUS
...
expr : MINUS expr %prec UMINUS {
    $$ = -$1;
}
```

- PLY

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

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

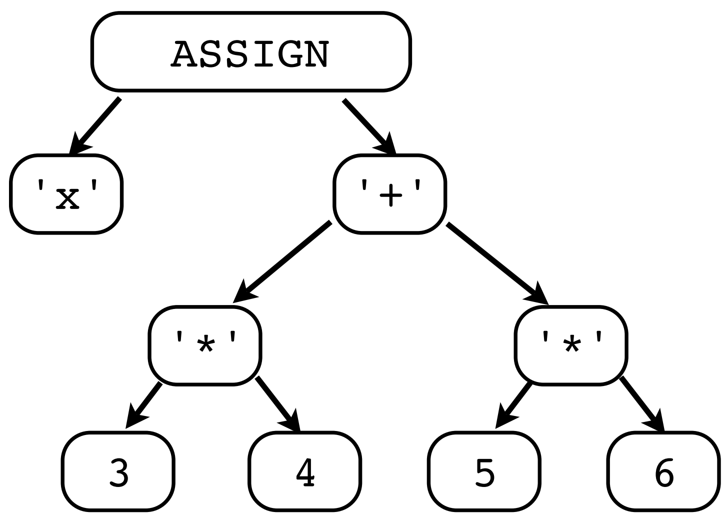
- Rule functions generally process values on right hand side of grammar rule
- Result is then stored in left hand side
- Results propagate up through the grammar
- Bottom-up parsing

Example: Abstract Syntax 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: Abstract Syntax Tree

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



PLY Validation

- PLY validates all token/grammar specs
- Duplicate rules
- Malformed regexs and grammars
- Missing rules and tokens
- Unused tokens and rules
- Improper function declarations
- Infinite recursion

Error Productions

- Yacc

```
funcall_err : ID LPAREN error RPAREN {
                 printf("Syntax error in arguments\n");
          }
          ;
```

- PLY

```
def p_funcall_err(p):
    '''ID LPAREN error RPAREN'''
    print "Syntax error in arguments\n"
```

Resources

- PLY homepage

  `http://www.dabeaz.com/ply`

- Mailing list/group

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

Thank You