Writing Compilers in Python

(with PLY)

Dave Beazley
http://www.dabeaz.com

October 12, 2006

Overview

- Crash course on compilers
- Lex/yacc
- An introduction to PLY
- Blood and guts (Rated R)
- Various PLY features (more gore)
- Examples

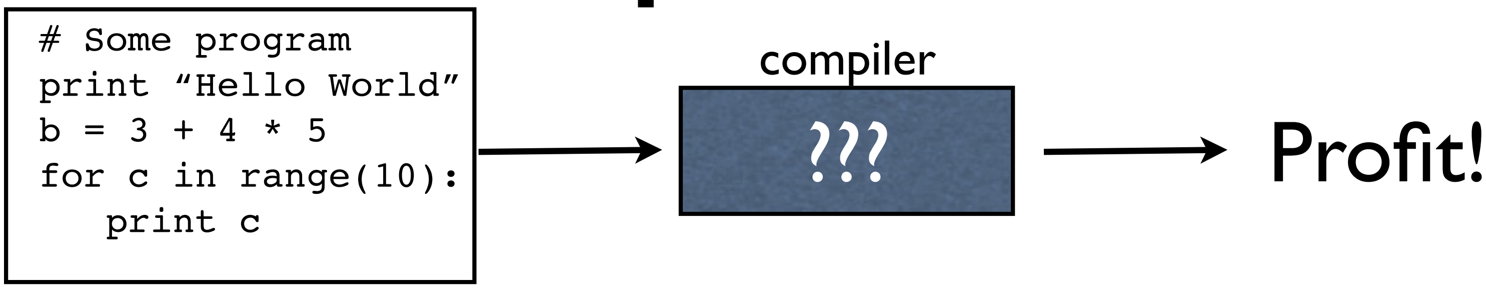
Disclaimer

- Compilers is an advanced topic
- Please stop me for questions!

Motivation

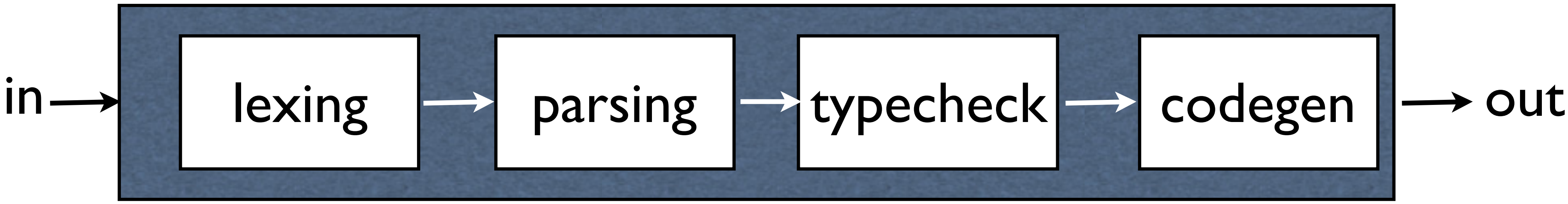
- Writing a compiler is hard
- Writing Python code seems to be easy
- So why not write a compiler in Python?

Compilers 101



- What is a compiler?
- A program that processes other programs
- Typically implements a programming lang.
- Examples:
    - gcc, javac, SWIG, Doxygen, Python

Compiler Design



- Compiler broken into stages
- Lexing/parsing related to reading input
- Type checking is error checking/validation
- Code generation does something

Example

- Parse and generate code for the following:

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

Lexing

- Splits input text into tokens
- Makes sure the input uses right alphabet

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

NAME = NUM + NUM * ( NUM + NUM ) / FLOAT
```

- Detects illegal symbols

```
b = 40 * $5

Illegal Character
```

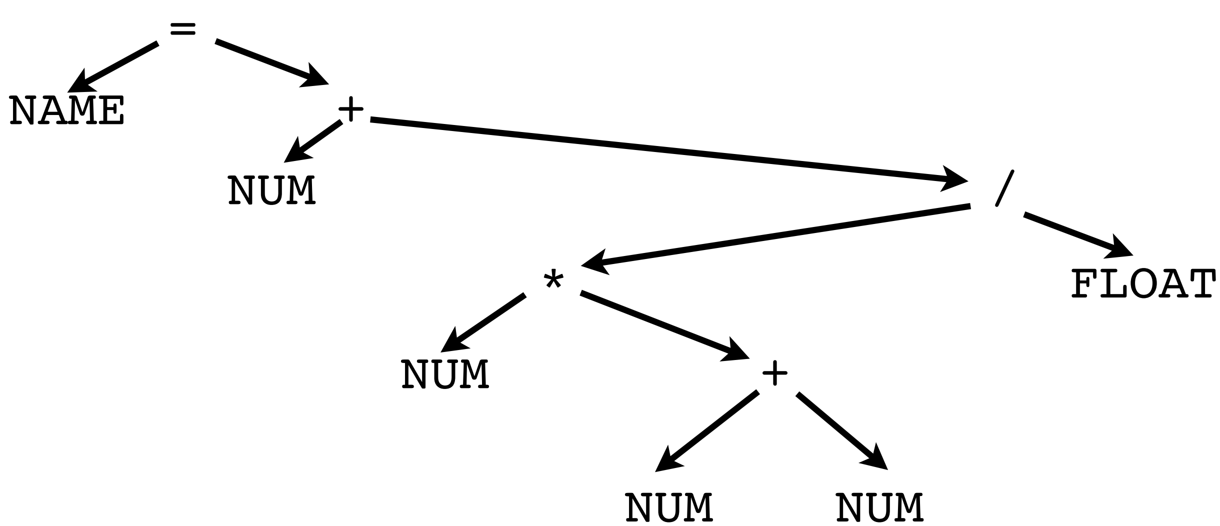
Parsing

- Makes sure input is structurally correct

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

- Builds program structure (e.g., parse tree)

```
NAME = NUM + NUM * ( NUM + NUM ) / FLOAT
```



Parsing

- Detects syntax errors

```
b = 40 + “hello”      (Syntax OK)
b = 3 * 4 7 /         (Syntax error)
```

- If a program parses, it is at least well-formed
- Still don’t know if program is correct

```
b = 40 + “hello”      (???)
```

Type checking

- Enforces underlying semantics

```
b = 40 + 20*(2+3)/37.5      (OK)
c = 3 + “hello”             (TYPE ERROR)
d[4.5] = 4                  (BAD INDEX)
```

- Example: + operator

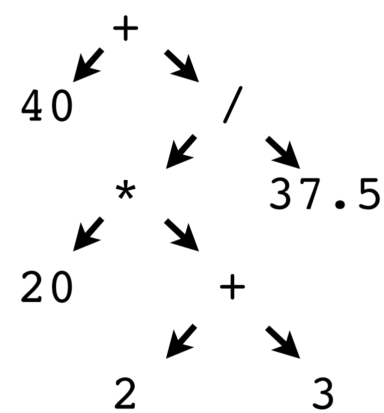
$$
\begin{tikzcd}
 & + \arrow[dl] \arrow[dr] & \\
\mathtt{LHS} & & \mathtt{RHS}
\end{tikzcd}
$$

```
1. LHS and RHS must be the same type
2. If different types, must be convertible to same type
```

Code Generation

- Processing the parse tree in some way
- Usually a traversal of the parse tree

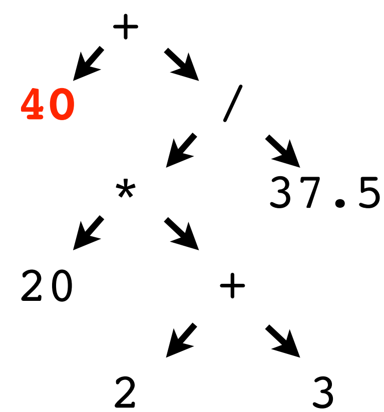
```
b = 40 + 20*(2+3)/37.5
```



Code Generation

- Processing the parse tree in some way
- Usually a traversal of the parse tree

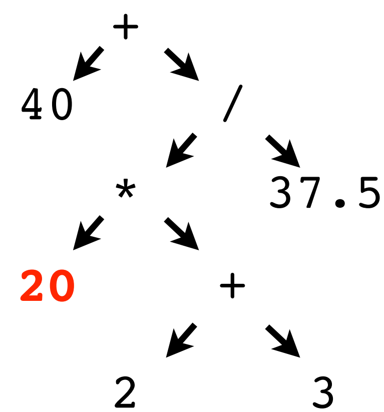
```
b = 40 + 20*(2+3)/37.5

LOAD R1, 40
```



Code Generation

- Processing the parse tree in some way
- Usually a traversal of the parse tree

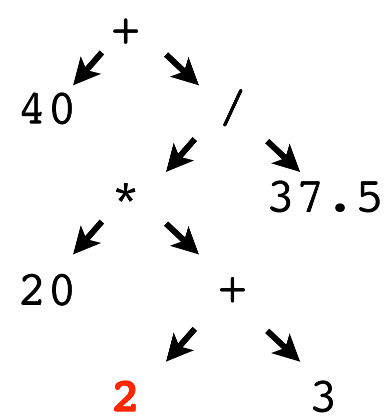
```
b = 40 + 20*(2+3)/37.5

LOAD R1, 40
LOAD R2, 20
```



Code Generation

- Processing the parse tree in some way
- Usually a traversal of the parse tree

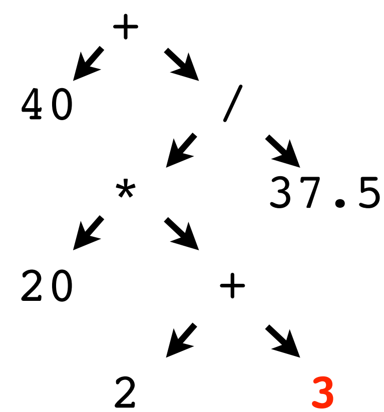
```
b = 40 + 20*(2+3)/37.5

LOAD R1, 40
LOAD R2, 20
LOAD R3, 2
```



Code Generation

- Processing the parse tree in some way
- Usually a traversal of the parse tree

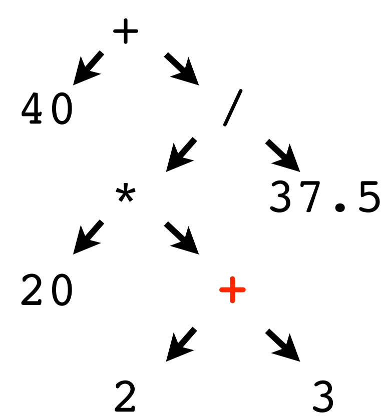
```
b = 40 + 20*(2+3)/37.5

LOAD R1, 40
LOAD R2, 20
LOAD R3, 2
LOAD R4, 3
```



Code Generation

- Processing the parse tree in some way
- Usually a traversal of the parse tree

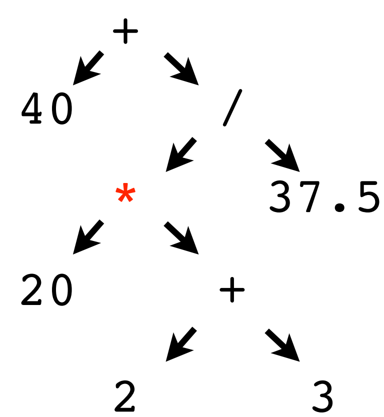
```
b = 40 + 20*(2+3)/37.5

LOAD R1, 40
LOAD R2, 20
LOAD R3, 2
LOAD R4, 3
ADD  R3, R4, R3  ; R3 = (2+3)
```



Code Generation

- Processing the parse tree in some way
- Usually a traversal of the parse tree

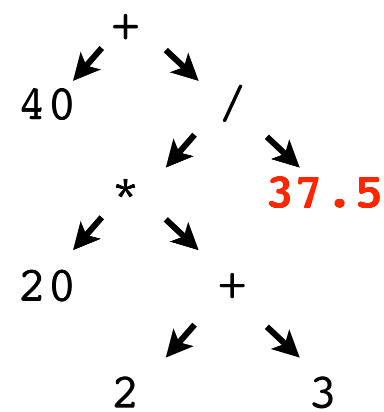
```
b = 40 + 20*(2+3)/37.5

LOAD R1, 40
LOAD R2, 20
LOAD R3, 2
LOAD R4, 3
ADD  R3, R4, R3   ; R3 = (2+3)
MUL  R2, R3, R2   ; R2 = 20*(2+3)
```



Code Generation

- Processing the parse tree in some way
- Usually a traversal of the parse tree

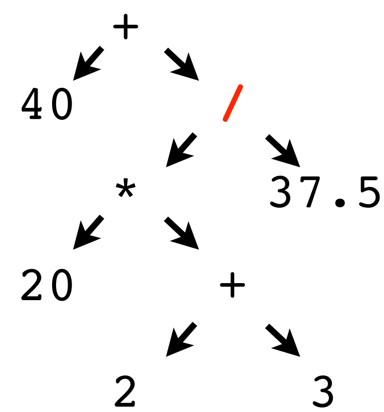
```
b = 40 + 20*(2+3)/37.5

LOAD R1, 40
LOAD R2, 20
LOAD R3, 2
LOAD R4, 3
ADD  R3, R4, R3   ; R3 = (2+3)
MUL  R2, R3, R2   ; R2 = 20*(2+3)
LOAD R3, 37.5
```



Code Generation

- Processing the parse tree in some way
- Usually a traversal of the parse tree

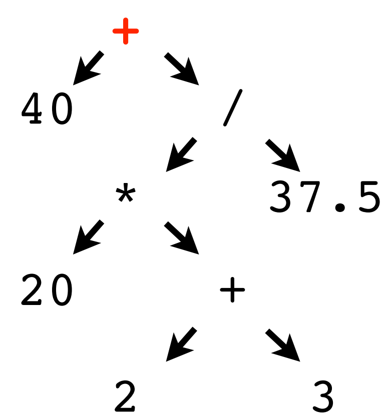
```
b = 40 + 20*(2+3)/37.5

LOAD R1, 40
LOAD R2, 20
LOAD R3, 2
LOAD R4, 3
ADD  R3, R4, R3   ; R3 = (2+3)
MUL  R2, R3, R2   ; R2 = 20*(2+3)
LOAD R3, 37.5
DIV  R2, R3, R2   ; R2 = 20*(2+3)/37.5
```



Code Generation

- Processing the parse tree in some way
- Usually a traversal of the parse tree

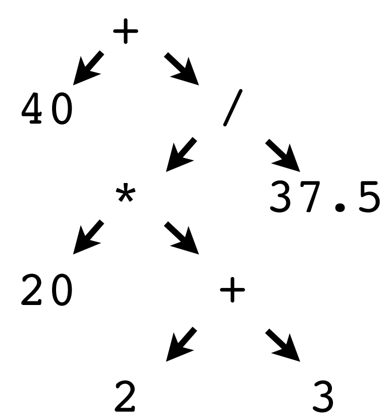
```
b = 40 + 20*(2+3)/37.5

LOAD R1, 40
LOAD R2, 20
LOAD R3, 2
LOAD R4, 3
ADD  R3, R4, R3   ; R3 = (2+3)
MUL  R2, R3, R2   ; R2 = 20*(2+3)
LOAD R3, 37.5
DIV  R2, R3, R2   ; R2 = 20*(2+3)/37.5
ADD  R1, R2, R1   ; R1 = 40+20*(2+3)/37.5
```



Code Generation

- Processing the parse tree in some way
- Usually a traversal of the parse tree

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

LOAD R1, 40
LOAD R2, 20
LOAD R3, 2
LOAD R4, 3
ADD  R3, R4, R3   ; R3 = (2+3)
MUL  R2, R3, R2   ; R2 = 20*(2+3)
LOAD R3, 37.5
DIV  R2, R3, R2   ; R2 = 20*(2+3)/37.5
ADD  R1, R2, R1   ; R1 = 40+20*(2+3)/37.5
STORE R1, "b"
```



Comments

- Concept is mostly straightforward
- Omitting many horrible details
- More covered in a compilers course.

Parsing (revisited)

- Parsing is probably most annoying problem
- Not a matter of simple text processing
- Not obvious
- Not fun

Lex & Yacc

- Programming tools for writing parsers
- Lex - Lexical analysis (tokenizing)
- Yacc - Yet Another Compiler Compiler (parsing)
- History:
  - Yacc : ~1973. Stephen Johnson (AT&T)
  - Lex : ~1974. Eric Schmidt and Mike Lesk (AT&T)
  - Both are standard Unix utilities
  - GNU equivalents: flex and bison
  - Part of IEEE POSIX 1003.2 standard
  - Implementations available for most programming languages

Lex/Yacc Big Picture

`scanner.l`

token specification

`parser.y`

grammar specification

Lex/Yacc Big Picture

scanner.l

parser.y

```
/* scanner.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

scanner.l

spe

parser.y

```
/* 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

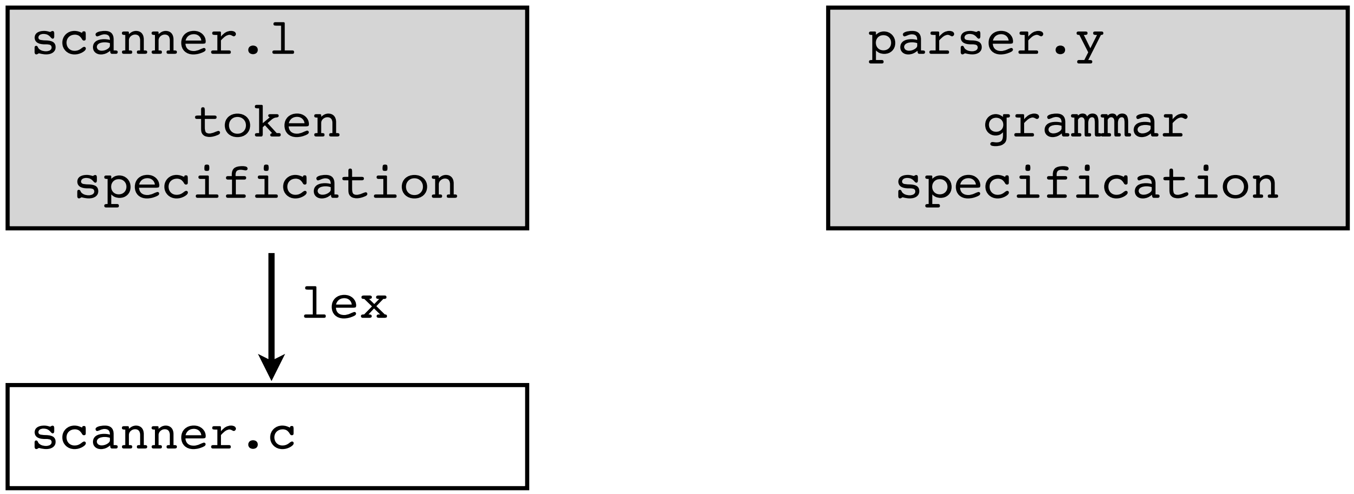
scanner.l

token specification

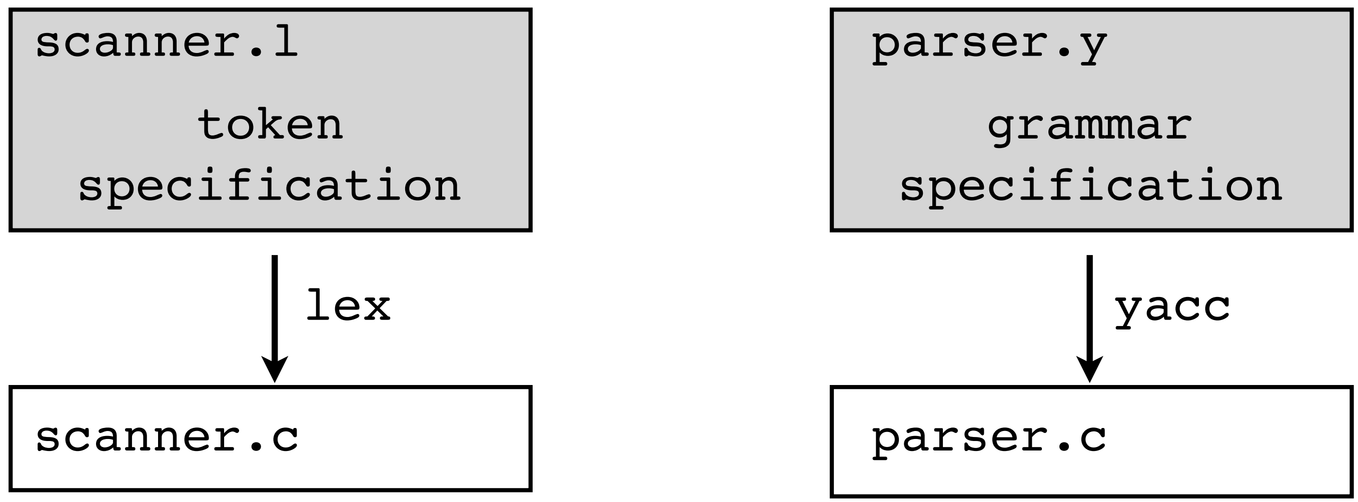
parser.y

grammar specification

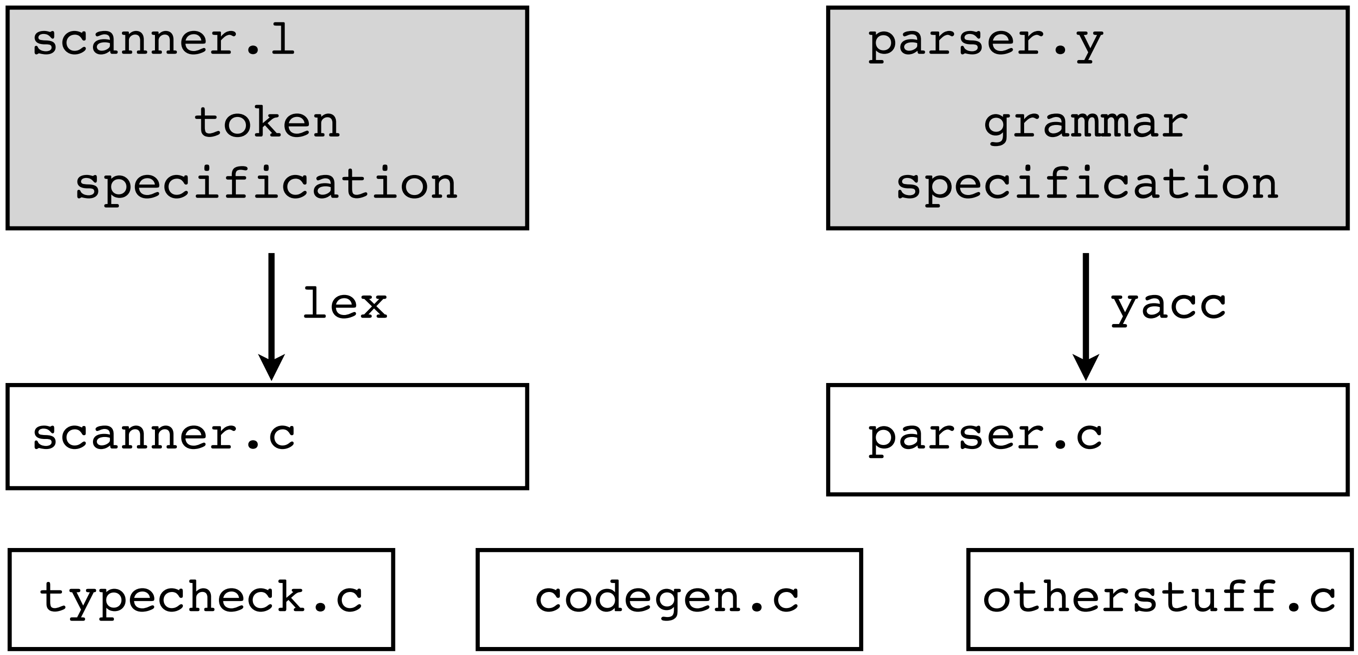
Lex/Yacc Big Picture



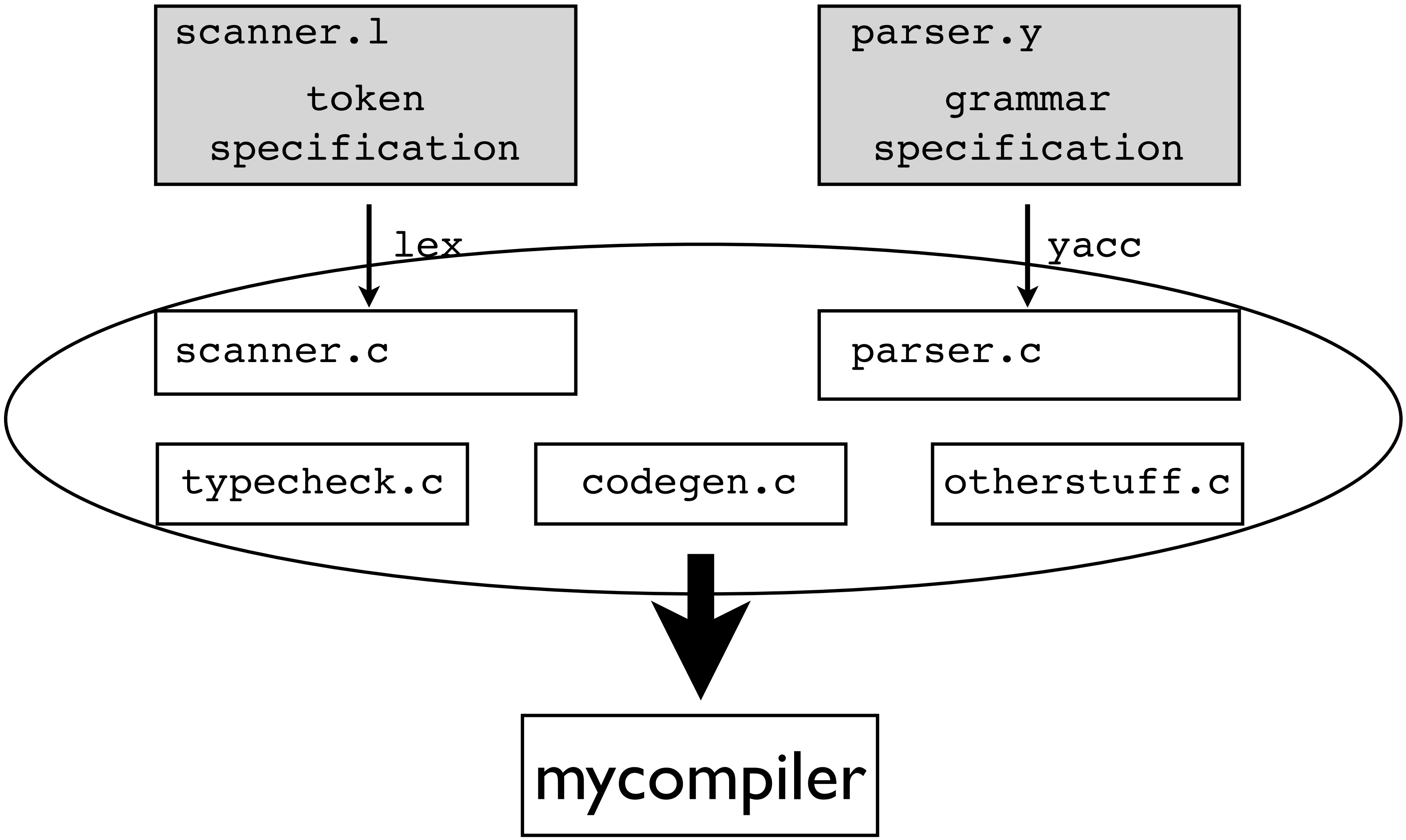
Lex/Yacc Big Picture



Lex/Yacc Big Picture



Lex/Yacc Big Picture



Lex/Yacc Comments

- Code generators
- Create a parser from a specification
- Classic versions create C code.
- Variants target other languages

PLY

- Python Lex-Yacc
- 100% Python version of lex/yacc toolset
- History:
  - Late 90's. "Write don't you rewrite SWIG in Python?"
  - 2000 : "No! Now stop bugging me about it!"
  - 2001 : Dave teaches a compilers course at UofC. An experiment.
    Students write a compiler in Python.
  - 2001 : PLY-1.0 developed and released.
  - 2002 - 2005 : Occasional maintenance and bug fixes.
  - 2006 : Major update to PLY-2.x (in progress).
- This is the first talk about it

PLY Overview

- Provide same functionality as lex/yacc
  - Identical parsing algorithm (LALR(1))
  - Extensive error checking.
  - Comparable debugging features (sic)
  - Keep it simple (ha!)
- Make use of Python features

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
- This is where it gets interesting

lex.py 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
```

lex.py specification

- Tokens denoted by t_*TOKEN* declarations
- Tokens are defined by regular expressions

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

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

- May be a simple variable or a function
- For functions, regex is in docstring.

lex.py construction

- lex() function is used to build the lexer

```
import ply.lex as lex
... token specifications ...

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

- Uses introspection to read spec
- Token information taken out of calling module
- Big difference between Unix lex and PLY

lex.py construction

- lex() function is used to build the lexer

```
import ply.lex as lex
... token specifications ...

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

- Uses intros
- Token info
- Big difference

Sick introspection hack

```
try: raise RuntimeError
except RuntimeError:
   e,b,t = sys.exc_info()
   f = t.tb_frame
   f = f.f_back
   mdict = f.f_globals
```

lex.py Validation

- lex.lex() performs extensive error checking
- Bad tokens, duplicate tokens, malformed functions, etc.

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

calc.py:20 Rule t_NAME redefined.
           Previously defined on line 14.
```

- Goal: informative debugging messages

lex.py use

- Two functions: input(), token()

```
import ply.lex as lex
...
lex.lex()                       # Build the lexer
...
data = "x = 3*4+5-6"
lex.input(data)                 # Feed some text
while 1:
    tok = lex.token()           # Get next token
    if not tok: break
    print tok
```

- Call token() repeatedly to fetch tokens

Example

yacc.py preliminaries

- yacc.py 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
```

yacc.py 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
```

yacc.py rules

- All rules defined by p_*funcname*(p) funcs
- Grammar specified in docstrings

```
def p_expr(p):
    '''expr : expr PLUS term
            | expr MINUS term
            | term'''
```

- Rules may be split apart or combined

```
def p_expr_plus(p):
    'expr : expr PLUS term'
def p_expr_minus(p):
    'expr : expr MINUS term'
def p_expr_term(p):
    'expr : term'
```

yacc.py construction

- yacc() function builds the parser

```
import ply.yacc as yacc
... rule specifications ...

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

- Uses introspection (as before)
- Generates parsing tables and diagnostics

```
% python myparser.py
yacc: Warning. no p_error() function is defined
yacc: Generating LALR parsing table...
```

yacc.py performance

- yacc.yacc() is expensive (several seconds)
- Parsing tables written to file parsetab.py
- Only regenerated when grammar changes
- Avoids performance hit on repeated use

yacc.py validation

- yacc.yacc() also performs validation
- Duplicate rules, malformed grammars, infinite recursion, undefined symbols, bad arguments, etc.
- Provides the same error messages provided by Unix yacc.

yacc.py parsing

- yacc.parse() function

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

- This implicitly feeds data into lexer

Example

A peek inside

- PLY is based on LR-parsing. LALR(1)
- AKA: Shift-reduce parsing
- Widely used.
- Table driven.
- Speed is independent of grammar size

LR Parsing

- Three basic components:
  - A stack of grammar symbols and values.
  - Two operators: shift, reduce
  - An underlying state machine.
- Example

LR Example: Step 1

stack

input

```
X = 3 + 4 * 5 $end
```

Action:

Grammar

PLY Rules

```
(1) assign : NAME EQUALS expr           -> p_assign(p)
(2) expr   : expr PLUS term             -> p_expr(p)
(3)        | expr MINUS term
(4)        | term
(5) term   : term TIMES factor          -> p_term(p)
(6)        | term DIVIDE factor
(7)        | factor
(8) factor : NUMBER                     -> p_factor(p)
```

LR Example: Step 1

stack

input

```
x = 3 + 4 * 5 $end
```

Action:

Grammar

PLY Rules

```
(1) assign : NAME EQUALS expr          -> p_assign(p)
(2) expr   : expr PLUS term            -> p_expr(p)
(3)        | expr MINUS term
(4)        | term
(5) term   : term TIMES factor         -> p_term(p)
(6)        | term DIVIDE factor
(7)        | factor
(8) factor : NUMBER                    -> p_factor(p)
```

LR Example: Step 1

stack

```
NAME
'X'
```

input

```
= 3 + 4 * 5 $end
```

Action: shift

Grammar

PLY Rules

```
(1) assign : NAME EQUALS expr            -> p_assign(p)
(2) expr   : expr PLUS term              -> p_expr(p)
(3)        | expr MINUS term
(4)        | term
(5) term   : term TIMES factor           -> p_term(p)
(6)        | term DIVIDE factor
(7)        | factor
(8) factor : NUMBER                      -> p_factor(p)
```

LR Example: Step 1

stack

Symbol type

```
NAME
'X'
```

Symbol value

Action: shift

input

```
= 3 + 4 * 5 $end
```

Grammar / PLY Rules

```
(1) assign : NAME EQUALS expr            -> p_assign(p)
(2) expr   : expr PLUS term              -> p_expr(p)
(3)        | expr MINUS term
(4)        | term
(5) term   : term TIMES factor           -> p_term(p)
(6)        | term DIVIDE factor
(7)        | factor
(8) factor : NUMBER                      -> p_factor(p)
```

LR Example: Step 2

stack

```
NAME
'X'
```

input

```
= 3 + 4 * 5 $end
```

Action:

Grammar

PLY Rules

```
(1) assign : NAME EQUALS expr          -> p_assign(p)
(2) expr   : expr PLUS term            -> p_expr(p)
(3)        | expr MINUS term
(4)        | term
(5) term   : term TIMES factor         -> p_term(p)
(6)        | term DIVIDE factor
(7)        | factor
(8) factor : NUMBER                    -> p_factor(p)
```

LR Example: Step 2

stack

```
NAME
'X'
```

input

```
= 3 + 4 * 5 $end
```

Action:

Grammar

PLY Rules

```
(1) assign : NAME EQUALS expr         -> p_assign(p)
(2) expr   : expr PLUS term           -> p_expr(p)
(3)        | expr MINUS term
(4)        | term
(5) term   : term TIMES factor        -> p_term(p)
(6)        | term DIVIDE factor
(7)        | factor
(8) factor : NUMBER                   -> p_factor(p)
```

LR Example: Step 2

stack

```
NAME EQUALS
'X'  '='
```

input

```
3 + 4 * 5 $end
```

Action: shift

Grammar

PLY Rules

```
(1) assign : NAME EQUALS expr          -> p_assign(p)
(2) expr   : expr PLUS term            -> p_expr(p)
(3)        | expr MINUS term
(4)        | term
(5) term   : term TIMES factor         -> p_term(p)
(6)        | term DIVIDE factor
(7)        | factor
(8) factor : NUMBER                    -> p_factor(p)
```

LR Example: Step 3

stack

```
NAME EQUALS
'X'  '='
```

input

```
3 + 4 * 5 $end
```

Action:

Grammar

PLY Rules

```
(1) assign : NAME EQUALS expr          -> p_assign(p)
(2) expr   : expr PLUS term            -> p_expr(p)
(3)        | expr MINUS term
(4)        | term
(5) term   : term TIMES factor         -> p_term(p)
(6)        | term DIVIDE factor
(7)        | factor
(8) factor : NUMBER                    -> p_factor(p)
```

LR Example: Step 3

stack

```
NAME EQUALS
'X'  '='
```

input

```
3 + 4 * 5 $end
```

Action:

Grammar

PLY Rules

```
(1) assign : NAME EQUALS expr          -> p_assign(p)
(2) expr   : expr PLUS term            -> p_expr(p)
(3)        | expr MINUS term
(4)        | term
(5) term   : term TIMES factor         -> p_term(p)
(6)        | term DIVIDE factor
(7)        | factor
(8) factor : NUMBER                    -> p_factor(p)
```

LR Example: Step 3

stack

```
NAME EQUALS NUMBER
'X'  '='    3
```

input

```
+ 4 * 5 $end
```

Action: shift

Grammar

PLY Rules

```
(1) assign : NAME EQUALS expr        -> p_assign(p)
(2) expr   : expr PLUS term          -> p_expr(p)
(3)        | expr MINUS term
(4)        | term
(5) term   : term TIMES factor       -> p_term(p)
(6)        | term DIVIDE factor
(7)        | factor
(8) factor : NUMBER                  -> p_factor(p)
```

LR Example: Step 3

stack

```
NAME EQUALS NUMBER
'X'  '='    3
```

input

```
+ 4 * 5 $end
```

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

Action: shift

Grammar

PLY Rules

```
(1) assign : NAME EQUALS expr       -> p_assign(p)
(2) expr   : expr PLUS term         -> p_expr(p)
(3)        | expr MINUS term
(4)        | term
(5) term   : term TIMES factor      -> p_term(p)
(6)        | term DIVIDE factor
(7)        | factor
(8) factor : NUMBER                 -> p_factor(p)
```

LR Example: Step 4

stack

```
NAME EQUALS NUMBER
'X'  '='    3
```

input

```
+ 4 * 5 $end
```

Action:

Grammar

```
(1) assign : NAME EQUALS expr
(2) expr   : expr PLUS term
(3)        | expr MINUS term
(4)        | term
(5) term   : term TIMES factor
(6)        | term DIVIDE factor
(7)        | factor
(8) factor : NUMBER
```

PLY Rules

```
-> p_assign(p)
-> p_expr(p)
-> p_term(p)
-> p_factor(p)
```

LR Example: Step 4

stack

```
NAME EQUALS NUMBER
'X'  '='    3
```

input

```
+ 4 * 5 $end
```

Action:

Grammar

PLY Rules

```
(1) assign : NAME EQUALS expr          -> p_assign(p)
(2) expr   : expr PLUS term            -> p_expr(p)
(3)        | expr MINUS term
(4)        | term
(5) term   : term TIMES factor         -> p_term(p)
(6)        | term DIVIDE factor
(7)        | factor
(8) factor : NUMBER                    -> p_factor(p)
```

LR Example: Step 4

stack

```
NAME EQUALS NUMBER
'X'  '='    3
```

input

```
+ 4 * 5 $end
```

Action: reduce using rule 8

Grammar

PLY Rules

```
(1) assign : NAME EQUALS expr       -> p_assign(p)
(2) expr   : expr PLUS term         -> p_expr(p)
(3)        | expr MINUS term
(4)        | term
(5) term   : term TIMES factor      -> p_term(p)
(6)        | term DIVIDE factor
(7)        | factor
(8) factor : NUMBER                 -> p_factor(p)
```

LR Example: Step 4

stack

```
NAME EQUALS factor
'X'  '='    None
```

input

```
+ 4 * 5 $end
```

Action: reduce using rule 8

Grammar

```
(1) assign : NAME EQUALS expr
(2) expr   : expr PLUS term
(3)        | expr MINUS term
(4)        | term
(5) term   : term TIMES factor
(6)        | term DIVIDE factor
(7)        | factor
(8) factor : NUMBER
```

PLY Rules

```
-> p_assign(p)
-> p_expr(p)
-> p_term(p)
-> p_factor(p)
```

LR Example: Step 4

stack

```
NAME EQUALS factor
'X'  '='    None
```

input

```
+ 4 * 5 $end
```

This is None because p_factor() didn't do anything. More later.

Action: reduce

Grammar / **PLY Rules**

```
(1) assign : NAME EQUALS expr        -> p_assign(p)
(2) expr   : expr PLUS term          -> p_expr(p)
(3)        | expr MINUS term
(4)
                                     -> p_term(p)
(7)
(8) factor : NUMBER                  -> p_factor(p)
```

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

LR Example: Step 5

stack

```
NAME EQUALS factor
'X'  '='    None
```

input

```
+ 4 * 5 $end
```

Action:

Grammar

```
(1) assign : NAME EQUALS expr
(2) expr   : expr PLUS term
(3)        | expr MINUS term
(4)        | term
(5) term   : term TIMES factor
(6)        | term DIVIDE factor
(7)        | factor
(8) factor : NUMBER
```

PLY Rules

```
-> p_assign(p)
-> p_expr(p)
-> p_term(p)
-> p_factor(p)
```

LR Example: Step 5

stack

```
NAME EQUALS factor
'X'  '='    None
```

input

```
+ 4 * 5 $end
```

Action:

Grammar

```
(1) assign : NAME EQUALS expr
(2) expr   : expr PLUS term
(3)        | expr MINUS term
(4)        | term
(5) term   : term TIMES factor
(6)        | term DIVIDE factor
(7)        | factor
(8) factor : NUMBER
```

PLY Rules

```
-> p_assign(p)
-> p_expr(p)
-> p_term(p)
-> p_factor(p)
```

LR Example: Step 5

stack

```
NAME EQUALS factor
'X'  '='    None
```

input

```
+ 4 * 5 $end
```

Action: reduce using rule 7

Grammar

```
(1) assign : NAME EQUALS expr
(2) expr   : expr PLUS term
(3)        | expr MINUS term
(4)        | term
(5) term   : term TIMES factor
(6)        | term DIVIDE factor
(7)        | factor
(8) factor : NUMBER
```

PLY Rules

```
-> p_assign(p)
-> p_expr(p)
-> p_term(p)
-> p_factor(p)
```

LR Example: Step 5

stack

```
NAME EQUALS term
'X'  '='    None
```

input

```
+ 4 * 5 $end
```

Action: reduce using rule 7

Grammar

```
(1) assign : NAME EQUALS expr
(2) expr   : expr PLUS term
(3)        | expr MINUS term
(4)        | term
(5) term   : term TIMES factor
(6)        | term DIVIDE factor
(7)        | factor
(8) factor : NUMBER
```

PLY Rules

```
-> p_assign(p)
-> p_expr(p)
-> p_term(p)
-> p_factor(p)
```

LR Example: Step 6

stack

```
NAME EQUALS term
'X'  '='    None
```

input

```
+ 4 * 5 $end
```

Action:

Grammar

PLY Rules

```
(1) assign : NAME EQUALS expr           -> p_assign(p)
(2) expr   : expr PLUS term             -> p_expr(p)
(3)        | expr MINUS term
(4)        | term
(5) term   : term TIMES factor          -> p_term(p)
(6)        | term DIVIDE factor
(7)        | factor
(8) factor : NUMBER                     -> p_factor(p)
```

LR Example: Step 6

stack

```
NAME EQUALS term
'X'  '='    None
```

input

```
+ 4 * 5 $end
```

Action:

Grammar

```
(1) assign : NAME EQUALS expr
(2) expr   : expr PLUS term
(3)        | expr MINUS term
(4)        | term
(5) term   : term TIMES factor
(6)        | term DIVIDE factor
(7)        | factor
(8) factor : NUMBER
```

PLY Rules

```
-> p_assign(p)
-> p_expr(p)

-> p_term(p)

-> p_factor(p)
```

LR Example: Step 6

stack

```
NAME EQUALS term
'X'   '='   None
```

input

```
+ 4 * 5 $end
```

Action: reduce using rule 4

Grammar

```
(1) assign : NAME EQUALS expr
(2) expr   : expr PLUS term
(3)        | expr MINUS term
(4)        | term
(5) term   : term TIMES factor
(6)        | term DIVIDE factor
(7)        | factor
(8) factor : NUMBER
```

PLY Rules

```
-> p_assign(p)
-> p_expr(p)
-> p_term(p)
-> p_factor(p)
```

LR Example: Step 6

stack

```
NAME EQUALS expr
'X'  '='    None
```

input

```
+ 4 * 5 $end
```

Action: reduce using rule 4

Grammar

```
(1) assign : NAME EQUALS expr
(2) expr   : expr PLUS term
(3)        | expr MINUS term
(4)        | term
(5) term   : term TIMES factor
(6)        | term DIVIDE factor
(7)        | factor
(8) factor : NUMBER
```

PLY Rules

```
-> p_assign(p)
-> p_expr(p)
-> p_term(p)
-> p_factor(p)
```

LR Example: Step 7

stack

```
NAME EQUALS expr
'X"  '='    None
```

input

```
+ 4 * 5 $end
```

Action:

Grammar

PLY Rules

```
(1) assign : NAME EQUALS expr          -> p_assign(p)
(2) expr   : expr PLUS term            -> p_expr(p)
(3)        | expr MINUS term
(4)        | term
(5) term   : term TIMES factor         -> p_term(p)
(6)        | term DIVIDE factor
(7)        | factor
(8) factor : NUMBER                    -> p_factor(p)
```

LR Example: Step 7

stack

```
NAME EQUALS expr
'X'  '='    None
```

input

```
+ 4 * 5 $end
```

Action: ????

Grammar / PLY Rules

```
(1) assign : NAME EQUALS expr        -> p_assign(p)
(2) expr   : expr PLUS term          -> p_expr(p)
(3)        | expr MINUS term
(4)        | term
(5) term   : term TIMES factor       -> p_term(p)
(6)        | term DIVIDE factor
(7)        | factor
(8) factor : NUMBER                  -> p_factor(p)
```

LR Example: Step 7

stack

```
NAME EQUALS expr
'X'  '='    None
```

input

```
+ 4 * 5 $end
```

Action: shift

Grammar

PLY Rules

```
(1) assign : NAME EQUALS expr        -> p_assign(p)
(2) expr   : expr PLUS term          -> p_expr(p)
(3)        | expr MINUS term
(4)        | term
(5) term   : term TIMES factor       -> p_term(p)
(6)        | term DIVIDE factor
(7)        | factor
(8) factor : NUMBER                  -> p_factor(p)
```

LR Example: Step 7

stack

```
NAME EQUALS expr PLUS
'X'  '='    None '+'
```

input

```
4 * 5 $end
```

Action: shift

Grammar

PLY Rules

```
(1) assign : NAME EQUALS expr         -> p_assign(p)
(2) expr   : expr PLUS term           -> p_expr(p)
(3)        | expr MINUS term
(4)        | term
(5) term   : term TIMES factor        -> p_term(p)
(6)        | term DIVIDE factor
(7)        | factor
(8) factor : NUMBER                   -> p_factor(p)
```

LR Example: Step 8

stack

```
NAME EQUALS expr PLUS
'X'  '='    None '+'
```

input

```
NUMBER
<-----------        4 * 5 $end
```

Action: shift

Grammar

PLY Rules

```
(1) assign : NAME EQUALS expr        -> p_assign(p)
(2) expr   : expr PLUS term          -> p_expr(p)
(3)        | expr MINUS term
(4)        | term
(5) term   : term TIMES factor       -> p_term(p)
(6)        | term DIVIDE factor
(7)        | factor
(8) factor : NUMBER                  -> p_factor(p)
```

LR Example: Step 9

stack

```
NAME EQUALS expr PLUS NUMBER
'X'  '='    None '+'  4
```

input

```
* 5 $end
```

Action: reduce using rule 8

Grammar

PLY Rules

```
(1) assign : NAME EQUALS expr        -> p_assign(p)
(2) expr   : expr PLUS term          -> p_expr(p)
(3)        | expr MINUS term
(4)        | term
(5) term   : term TIMES factor       -> p_term(p)
(6)        | term DIVIDE factor
(7)        | factor
(8) factor : NUMBER                  -> p_factor(p)
```

LR Example: Step 9

stack

```
NAME EQUALS expr PLUS factor
'X'  '='    None '+'  None
```

input

```
* 5 $end
```

Action: reduce using rule 7

Grammar

```
(1) assign : NAME EQUALS expr          -> p_assign(p)
(2) expr   : expr PLUS term            -> p_expr(p)
(3)        | expr MINUS term
(4)        | term
(5) term   : term TIMES factor         -> p_term(p)
(6)        | term DIVIDE factor
(7)        | factor
(8) factor : NUMBER                    -> p_factor(p)
```

PLY Rules

LR Example: Step 10

stack

```
NAME EQUALS expr PLUS term        TIMES          * 5 $end
'X'  '='    None '+'  None   <--------------
```

input

Action: shift

Grammar

PLY Rules

```
(1) assign : NAME EQUALS expr          -> p_assign(p)
(2) expr   : expr PLUS term            -> p_expr(p)
(3)        | expr MINUS term
(4)        | term
(5) term   : term TIMES factor         -> p_term(p)
(6)        | term DIVIDE factor
(7)        | factor
(8) factor : NUMBER                    -> p_factor(p)
```

LR Example: Step 11

stack

```
NAME EQUALS expr PLUS term TIMES
'X'  '='    None '+'  None '*'
```

input

```
<-- NUMBER   5 $end
```

Action: shift

Grammar

```
(1) assign : NAME EQUALS expr
(2) expr   : expr PLUS term
(3)        | expr MINUS term
(4)        | term
(5) term   : term TIMES factor
(6)        | term DIVIDE factor
(7)        | factor
(8) factor : NUMBER
```

PLY Rules

```
-> p_assign(p)
-> p_expr(p)
-> p_term(p)
-> p_factor(p)
```

LR Example: Step 12

stack

```
NAME EQUALS expr PLUS term TIMES NUMBER
'X'  '='    None '+'  None '*'   5
```

input

```
$end
```

Action: reduce using rule 8

Grammar

```
(1) assign : NAME EQUALS expr
(2) expr   : expr PLUS term
(3)        | expr MINUS term
(4)        | term
(5) term   : term TIMES factor
(6)        | term DIVIDE factor
(7)        | factor
(8) factor : NUMBER
```

PLY Rules

```
-> p_assign(p)
-> p_expr(p)
-> p_term(p)
-> p_factor(p)
```

LR Example: Step 13

stack

```
NAME EQUALS expr PLUS term TIMES factor
'X'  '='    None '+'  None '*'   None
```

input

```
$end
```

Action: reduce using rule 5

Grammar

```
(1) assign : NAME EQUALS expr
(2) expr   : expr PLUS term
(3)        | expr MINUS term
(4)        | term
(5) term   : term TIMES factor
(6)        | term DIVIDE factor
(7)        | factor
(8) factor : NUMBER
```

PLY Rules

```
-> p_assign(p)
-> p_expr(p)
-> p_term(p)
-> p_factor(p)
```

LR Example: Step 14

stack

```
NAME EQUALS expr PLUS term
'X'  '='    None '+'  None
```

input

```
$end
```

Action: reduce using rule 2

Grammar

```
(1) assign : NAME EQUALS expr
(2) expr   : expr PLUS term
(3)        | expr MINUS term
(4)        | term
(5) term   : term TIMES factor
(6)        | term DIVIDE factor
(7)        | factor
(8) factor : NUMBER
```

PLY Rules

```
-> p_assign(p)
-> p_expr(p)
-> p_term(p)
-> p_factor(p)
```

LR Example: Step 15

stack

```
NAME EQUALS expr
'X'  '='    None
```

input

```
$end
```

Action: reduce using rule 1

Grammar

```
(1) assign : NAME EQUALS expr
(2) expr   : expr PLUS term
(3)        | expr MINUS term
(4)        | term
(5) term   : term TIMES factor
(6)        | term DIVIDE factor
(7)        | factor
(8) factor : NUMBER
```

PLY Rules

```
-> p_assign(p)
-> p_expr(p)
-> p_term(p)
-> p_factor(p)
```

LR Example: Step 16

stack

```
assign
None
```

input

```
$end
```

Action: Done.

Grammar

PLY Rules

```
(1) assign : NAME EQUALS expr          -> p_assign(p)
(2) expr   : expr PLUS term            -> p_expr(p)
(3)        | expr MINUS term
(4)        | term
(5) term   : term TIMES factor         -> p_term(p)
(6)        | term DIVIDE factor
(7)        | factor
(8) factor : NUMBER                    -> p_factor(p)
```

Yacc Rule Execution

- Rules are executed during reduction

```
def p_term_mul(p):
    'term : term TIMES factor'
```

- Parameter p refers to values on stack

```
stack: NAME EQUALS expr PLUS term TIMES factor
                             p[1] p[2]   p[3]

                reduce term : term TIMES factor
                       p[0]

stack: NAME EQUALS expr PLUS term
```

Example: Calculator

```
def p_assign(p):
    '''assign : NAME EQUALS expr'''
    print "Assigning", p[1],"value",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_factor(p):
    '''factor : NUMBER'''
    p[0] = p[1]
```

LR Example: Step 4

stack

```
NAME EQUALS NUMBER
'X'  '='    3
```

input

```
+ 4 * 5 $end
```

Action:

Grammar

```
(1) assign : NAME EQUALS expr
(2) expr   : expr PLUS term
(3)        | expr MINUS term
(4)        | term
(5) term   : term TIMES factor
(6)        | term DIVIDE factor
(7)        | factor
(8) factor : NUMBER
```

PLY Rules

```
-> p_assign(p)
-> p_expr(p)
-> p_term(p)
-> p_factor(p)
```

LR Example: Step 4

stack

```
NAME EQUALS NUMBER
'X'  '='    3
```

input

```
+ 4 * 5 $end
```

Action:

Grammar

PLY Rules

```
(1) assign : NAME EQUALS expr         -> p_assign(p)
(2) expr   : expr PLUS term           -> p_expr(p)
(3)        | expr MINUS term
(4)        | term
(5) term   : term TIMES factor        -> p_term(p)
(6)        | term DIVIDE factor
(7)        | factor
(8) factor : NUMBER                   -> p_factor(p)
```

LR Example: Step 4

stack

```
NAME EQUALS NUMBER
'X'  '='    3
```

input

```
+ 4 * 5 $end
```

Action: reduce using rule 8

Grammar

PLY Rules

```
(1) assign : NAME EQUALS expr          -> p_assign(p)
(2) expr   : expr PLUS term            -> p_expr(p)
(3)        | expr MINUS term
(4)        | term
(5) term   : term TIMES factor         -> p_term(p)
(6)        | term DIVIDE factor
(7)        | factor
(8) factor : NUMBER                    -> p_factor(p)
```

LR Example: Step 4

stack

```
NAME EQUALS factor
'X'  '='    3
```

input

```
+ 4 * 5 $end
```

Action: reduce using rule 8

Grammar

```
(1) assign : NAME EQUALS expr
(2) expr   : expr PLUS term
(3)        | expr MINUS term
(4)        | term
(5) term   : term TIMES factor
(6)        | term DIVIDE factor
(7)        | factor
(8) factor : NUMBER
```

PLY Rules

```
-> p_assign(p)
-> p_expr(p)
-> p_term(p)
-> p_factor(p)
```

LR Example: Step 4

stack

```
NAME EQUALS factor
'X'  '='    3
```

input

```
+ 4 * 5 $end
```

Action: reduc

This retains its value because of assignment in p_factor().

Grammar

```
(1) assign : NAME EQUALS expr
(2) expr   : expr PLUS term
(3)        | expr MINUS term
(4
(7)
(8) factor : NUMBER
```

PLY Rules

```
-> p_assign(p)
-> p_expr(p)
-> p_term(p)
-> p_factor(p)
```

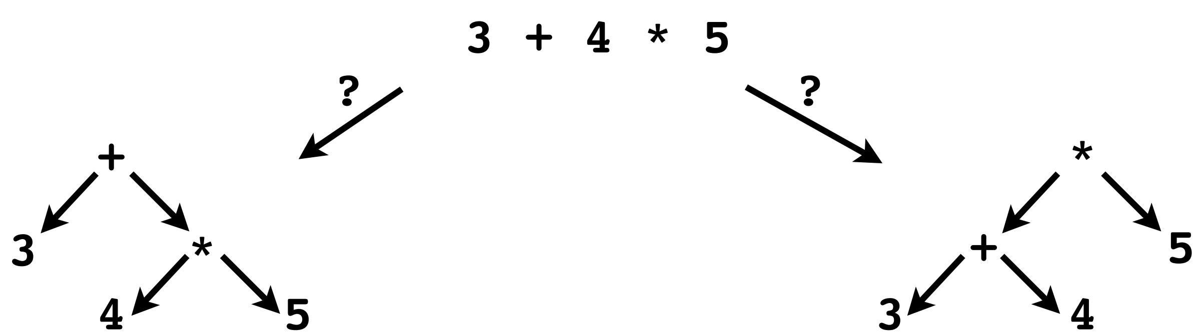
```
def p_factor(p):
    '''factor : NUMBER'''
    p[0] = p[1]
```

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

Ambiguous Grammars

```
def p_assign(p):
    ‘’’assign : NAME EQUALS expr’’’

def p_expr(p):
    ‘’’expr : expr PLUS expr
            | expr MINUS expr
            | expr TIMES expr
            | expr DIVIDE expr
            | NUMBER’’’
```



Ambiguous Grammars

- Multiple possible parse trees
- Is reported as a "shift/reduce conflict"

```
yacc: Generating LALR parsing table...
yacc: 16 shift/reduce conflicts
```

- May also get "reduce/reduce conflict"
- Probably most mysterious aspect of yacc

Shift/Reduce Conflict Explained

stack

```
NAME EQUALS expr PLUS expr
```

input

```
* 5 $end
```

Grammar

```
(1) assign : NAME EQUALS expr
(2) expr   : expr PLUS expr
(3)        | expr MINUS expr
(4)        | expr TIMES expr
(5)        | expr DIVIDE expr
(6)        | NUMBER
```

Possible Actions:

Shift/Reduce Conflict Explained

stack

```
NAME EQUALS expr PLUS expr
```

input

```
* 5 $end
```

Grammar

```
(1) assign : NAME EQUALS expr
(2) expr   : expr PLUS expr
(3)        | expr MINUS expr
(4)        | expr TIMES expr
(5)        | expr DIVIDE expr
(6)        | NUMBER
```

Possible Actions:

reduce using rule 2

Shift/Reduce Conflict Explained

stack

```
NAME EQUALS expr PLUS expr
```

input

```
* 5 $end
```

```
NAME EQUALS expr
NAME EQUALS expr TIMES
NAME EQUALS expr TIMES NUMBER
NAME EQUALS expr TIMES expr
NAME EQUALS expr
```

Grammar

```
(1) assign : NAME EQUALS expr
(2) expr   : expr PLUS expr
(3)        | expr MINUS expr
(4)        | expr TIMES expr
(5)        | expr DIVIDE expr
(6)        | NUMBER
```

Possible Actions:

reduce using rule 2

Shift/Reduce Conflict Explained

stack

```
NAME EQUALS expr PLUS expr
```

input

```
* 5 $end
```

```
NAME EQUALS expr
NAME EQUALS expr TIMES
NAME EQUALS expr TIMES NUMBER
NAME EQUALS expr TIMES expr
NAME EQUALS expr
```

Grammar

```
(1) assign : NAME EQUALS expr
(2) expr   : expr PLUS expr
(3)        | expr MINUS expr
(4)        | expr TIMES expr
(5)        | expr DIVIDE expr
(6)        | NUMBER
```

Possible Actions:

reduce using rule 2

shift TIMES

Shift/Reduce Conflict Explained

stack

```
NAME EQUALS expr PLUS expr
```

input

```
* 5 $end
```

```
NAME EQUALS expr
NAME EQUALS expr TIMES
NAME EQUALS expr TIMES NUMBER
NAME EQUALS expr TIMES expr
NAME EQUALS expr
```

```
NAME EQUALS expr PLUS expr TIMES
NAME EQUALS expr PLUS expr TIMES NUMBER
NAME EQUALS expr PLUS expr TIMES expr
NAME EQUALS expr PLUS expr
NAME EQUALS expr
```

Grammar

```
(1) assign : NAME EQUALS expr
(2) expr   : expr PLUS expr
(3)        | expr MINUS expr
(4)        | expr TIMES expr
(5)        | expr DIVIDE expr
(6)        | NUMBER
```

Possible Actions:

reduce using rule 2
shift TIMES

Shift/reduce resolution

- Default action is to always shift
- Can sometimes control with precedence

```
precedence = (
    ('left','PLUS','MINUS'),
    ('left','TIMES','DIVIDE'),
)
def p_assign(p):
    '''assign : NAME EQUALS expr'''

def p_expr(p):
    '''expr : expr PLUS expr
            | expr MINUS expr
            | expr TIMES expr
            | expr DIVIDE expr
            | NUMBER'''
```

Error handling/recovery

- Syntax errors first fed through p_error()

```
def p_error(p):
    print "Syntax error"
```

- Then an 'error' symbol is shifted onto stack
- Stack is unwound until error is consumed

Error recovery

stack

```
NAME EQUALS expr PLUS expr
'X'  '='    3    '+'  4
```

input

```
5 $end
```

Grammar

```
(1) assign : NAME EQUALS expr
(2)        | NAME EQUALS error
(3) expr   : expr PLUS expr
(4)        | expr MINUS expr
(5)        | expr TIMES expr
(6)        | expr DIVIDE expr
(7)        | NUMBER
```

Error recovery

stack

```
NAME EQUALS expr PLUS expr
'X'  '='     3   '+'   4
```

input

```
5 $end
```

syntax error

Grammar

```
(1) assign : NAME EQUALS expr
(2)        | NAME EQUALS error
(3) expr   : expr PLUS expr
(4)        | expr MINUS expr
(5)        | expr TIMES expr
(6)        | expr DIVIDE expr
(7)        | NUMBER
```

Error recovery

stack

```
NAME EQUALS expr PLUS expr
'X'  '='    3    '+'  4
```

input

```
5 $end
```

syntax error

```
def p_error(p):
    print "Syntax error"
```

Grammar

```
(1) assign : NAME EQUALS expr
(2)        | NAME EQUALS error
(3) expr   : expr PLUS expr
(4)        | expr MINUS expr
(5)        | expr TIMES expr
(6)        | expr DIVIDE expr
(7)        | NUMBER
```

Error recovery

stack

```
NAME EQUALS expr PLUS expr error
'X'  '='    3    '+'  4
```

input

```
$end
```

Grammar

```
(1) assign : NAME EQUALS expr
(2)        | NAME EQUALS error
(3) expr   : expr PLUS expr
(4)        | expr MINUS expr
(5)        | expr TIMES expr
(6)        | expr DIVIDE expr
(7)        | NUMBER
```

Error recovery

stack

```
NAME EQUALS expr PLUS error
'X'  '='    3    '+'
```

input

```
$end
```

Grammar

```
(1) assign : NAME EQUALS expr
(2)        | NAME EQUALS error
(3) expr   : expr PLUS expr
(4)        | expr MINUS expr
(5)        | expr TIMES expr
(6)        | expr DIVIDE expr
(7)        | NUMBER
```

Error recovery

stack

```
NAME EQUALS expr error
'X'  '='    3
```

input

```
$end
```

Grammar

```
(1) assign : NAME EQUALS expr
(2)        | NAME EQUALS error
(3) expr   : expr PLUS expr
(4)        | expr MINUS expr
(5)        | expr TIMES expr
(6)        | expr DIVIDE expr
(7)        | NUMBER
```

Error recovery

stack

```
NAME EQUALS error
'X'  '='
```

input

```
$end
```

Grammar

```
(1) assign : NAME EQUALS expr
(2)        | NAME EQUALS error
(3) expr   : expr PLUS expr
(4)        | expr MINUS expr
(5)        | expr TIMES expr
(6)        | expr DIVIDE expr
(7)        | NUMBER
```

Error recovery

stack

```
NAME EQUALS error
'X'  '='
```

input

```
$end
```

Grammar

```
(1) assign : NAME EQUALS expr
(2)        | NAME EQUALS error
(3) expr   : expr PLUS expr
(4)        | expr MINUS expr
(5)        | expr TIMES expr
(6)        | expr DIVIDE expr
(7)        | NUMBER
```

```
def p_assign_err(p):
    'assign : NAME EQUALS error'
    print "Bad assignment"
```

Error recovery

stack

```
assign
```

input

```
$end
```

Grammar

```
(1) assign : NAME EQUALS expr
(2)        | NAME EQUALS error
(3) expr   : expr PLUS expr
(4)        | expr MINUS expr
(5)        | expr TIMES expr
(6)        | expr DIVIDE expr
(7)        | NUMBER
```

```
def p_assign_err(p):
    'assign : NAME EQUALS error'
    print "Bad assignment"
```

Debugging Output

- PLY creates a file parser.out
- Contains detailed debugging information
- Reading it involves voodoo and magic
- Useful if trying to track down conflicts

Debugging Output

```
Grammar

Rule 1     statement -> NAME = expression
Rule 2     statement -> expression
Rule 3     expression -> expression + expression
Rule 4     expression -> expression - expression
Rule 5     expression -> expression * expression
Rule 6     expression -> expression / expression
Rule 7     expression -> NUMBER

Terminals, with rules where they appear

*                    : 5
+                    : 3
-                    : 4
/                    : 6
=                    : 1
NAME                 : 1
NUMBER               : 7
error                :

Nonterminals, with rules where they appear

expression           : 1 2 3 3 4 4 5 5 6 6
statement            : 0

Parsing method: LALR

state 0

    (0) S' -> . statement
    (1) statement -> . NAME = expression
    (2) statement -> . expression
    (3) expression -> . expression + expression
    (4) expression -> . expression - expression
    (5) expression -> . expression * expression
    (6) expression -> . expression / expression
    (7) expression -> . NUMBER

    NAME            shift and go to state 1
    NUMBER          shift and go to state 2


    expression                     shift and go to state 4
    statement                      shift and go to state 3

state 1

    (1) statement -> NAME . = expression

    =               shift and go to state 5
```

```
state 10

    (1) statement -> NAME = expression .
    (3) expression -> expression . + expression
    (4) expression -> expression . - expression
    (5) expression -> expression . * expression
    (6) expression -> expression . / expression

    $end            reduce using rule 1 (statement -> NAME = expression .)
    +               shift and go to state 7
    -               shift and go to state 6
    *               shift and go to state 8
    /               shift and go to state 9


state 11

    (4) expression -> expression - expression .
    (3) expression -> expression . + expression
    (4) expression -> expression . - expression
    (5) expression -> expression . * expression
    (6) expression -> expression . / expression

  ! shift/reduce conflict for + resolved as shift.
  ! shift/reduce conflict for - resolved as shift.
  ! shift/reduce conflict for * resolved as shift.
  ! shift/reduce conflict for / resolved as shift.
    $end            reduce using rule 4 (expression -> expression - expression .)
    +               shift and go to state 7
    -               shift and go to state 6
    *               shift and go to state 8
    /               shift and go to state 9

  ! +               [ reduce using rule 4 (expression -> expression - expression .) ]
  ! -               [ reduce using rule 4 (expression -> expression - expression .) ]
  ! *               [ reduce using rule 4 (expression -> expression - expression .) ]
  ! /               [ reduce using rule 4 (expression -> expression - expression .) ]
```

Debugging Output

```
...
state 11

    (4) expression -> expression - expression .
    (3) expression -> expression . + expression
    (4) expression -> expression . - expression
    (5) expression -> expression . * expression
    (6) expression -> expression . / expression

  ! shift/reduce conflict for + resolved as shift.
  ! shift/reduce conflict for - resolved as shift.
  ! shift/reduce conflict for * resolved as shift.
  ! shift/reduce conflict for / resolved as shift.
    $end            reduce using rule 4 (expression -> expression - expression .)
    +               shift and go to state 7
    -               shift and go to state 6
    *               shift and go to state 8
    /               shift and go to state 9

  ! +               [ reduce using rule 4 (expression -> expression - expression .) ]
  ! -               [ reduce using rule 4 (expression -> expression - expression .) ]
  ! *               [ reduce using rule 4 (expression -> expression - expression .) ]
  ! /               [ reduce using rule 4 (expression -> expression - expression .) ]
...
```

```
    =               shift and go to state 5
```

Advanced PLY

- PLY supports more advanced yacc features
  - Empty productions
  - Error handling/recovery
  - Inherited attributes
  - Embedded actions

Advanced PLY (cont)

- Some Python specific features
  - Lexers/parsers can be defined as classes
  - Support for multiple lexers and parsers
  - Support for optimized mode (-O)

Class Example

```
import ply.yacc as yacc

class MyParser:
    def p_assign(self,p):
        '''assign : NAME EQUALS expr'''
    def p_expr(self,p):
        '''expr : expr PLUS term
                | expr MINUS term
                | term'''
    def p_term(self,p):
        '''term : term TIMES factor
                | term DIVIDE factor
                | factor'''
    def p_factor(self,p):
        '''factor : NUMBER'''
    def build(self):
        self.parser = yacc.yacc(object=self)
```

Summary

- This has been a quick tour of PLY/yacc
- Have skipped a lot of subtle details.

Why use PLY?

- Standard lex/yacc well known and used
- Suitable for large grammars
- Decent performance
- Very extensive error checking/validation

PLY Usage

- Thousands of downloads over five years
- Some applications (that I know of)
    - Teaching compilers
    - numbler.com   (Carl Shimer)
    - Parsing Ada source code.
    - Parsing molecule descriptions
    - Reading configuration files

Resources

- PLY homepage

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

- Mailing list/group

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

Reduce/Reduce Conflict Explained

stack

input

```
NAME EQUALS NUMBER                                    $end
```

Grammar

```
(1) assign : NAME EQUALS expr
(2)        | NAME EQUALS NUMBER
(3) expr   : expr PLUS expr
(4)        | expr MINUS expr
(5)        | expr TIMES expr
(6)        | expr DIVIDE expr
(7)        | NUMBER
```

Possible Actions:

Reduce/Reduce Conflict Explained

stack

input

`NAME EQUALS NUMBER`

`$end`

Grammar

```
(1) assign : NAME EQUALS expr
(2)        | NAME EQUALS NUMBER
(3) expr   : expr PLUS expr
(4)        | expr MINUS expr
(5)        | expr TIMES expr
(6)        | expr DIVIDE expr
(7)        | NUMBER
```

Possible Actions:

reduce using rule 2

Reduce/Reduce Conflict Explained

stack	input
`assign`	`$end`

Grammar

```
(1) assign : NAME EQUALS expr
(2)        | NAME EQUALS NUMBER
(3) expr   : expr PLUS expr
(4)        | expr MINUS expr
(5)        | expr TIMES expr
(6)        | expr DIVIDE expr
(7)        | NUMBER
```

Possible Actions:

reduce using rule 2

Reduce/Reduce Conflict Explained

stack

```
assign
NAME EQUALS NUMBER
```

input

```
$end
```

Grammar

```
(1) assign : NAME EQUALS expr
(2)        | NAME EQUALS NUMBER
(3) expr   : expr PLUS expr
(4)        | expr MINUS expr
(5)        | expr TIMES expr
(6)        | expr DIVIDE expr
(7)        | NUMBER
```

Possible Actions:

reduce using rule 2

reduce using rule 7

Reduce/Reduce Conflict Explained

stack

```
assign
NAME EQUALS expr
```

input

```
$end
```

Grammar

```
(1) assign : NAME EQUALS expr
(2)        | NAME EQUALS NUMBER
(3) expr   : expr PLUS expr
(4)        | expr MINUS expr
(5)        | expr TIMES expr
(6)        | expr DIVIDE expr
(7)        | NUMBER
```

Possible Actions:

reduce using rule 2

reduce using rule 7