Writing Parsers and Compilers with PLY

David Beazley
http://www.dabeaz.com

February 23, 2007

Overview

- Crash course on compilers
- An introduction to PLY
- Notable PLY features (why use it?)
- Experience writing a compiler in Python

Background

- Programs that process other programs
- Compilers
- Interpreters
- Wrapper generators
- Domain-specific languages
- Code-checkers

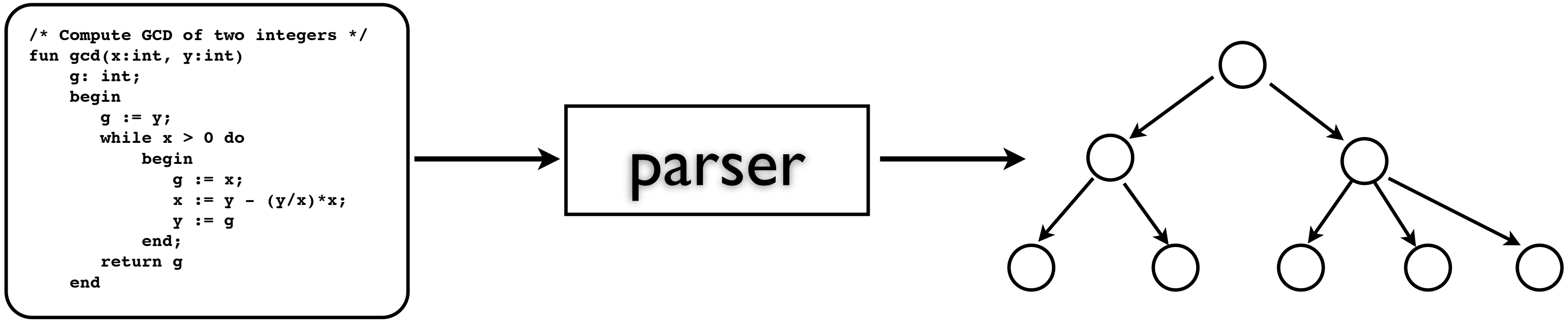
Example

- 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

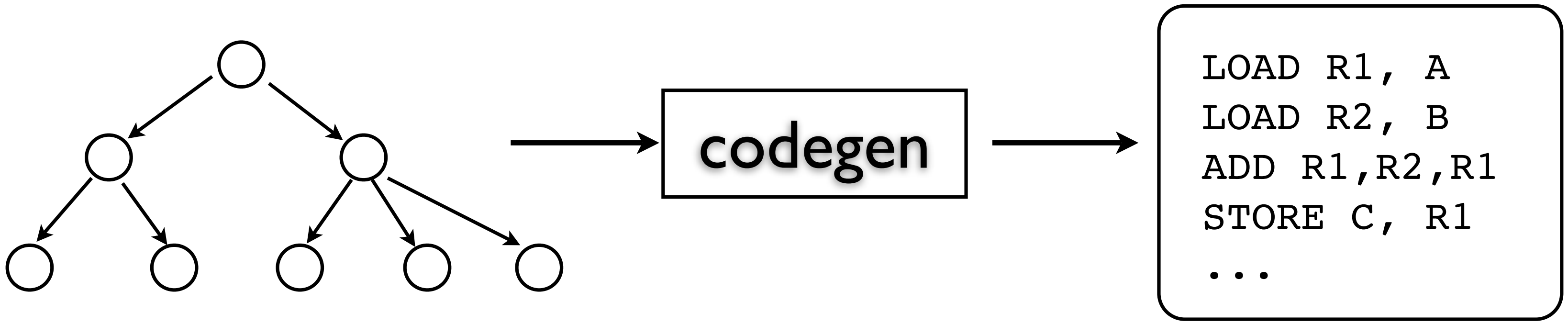
- Compilers have multiple phases
- First phase usually concerns "parsing"
- Read program and create abstract representation



```
/* 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



codegen

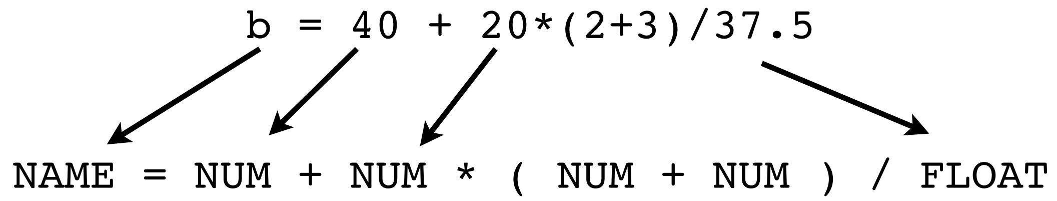
```
LOAD R1, A
LOAD R2, B
ADD R1,R2,R1
STORE C, R1
...
```

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

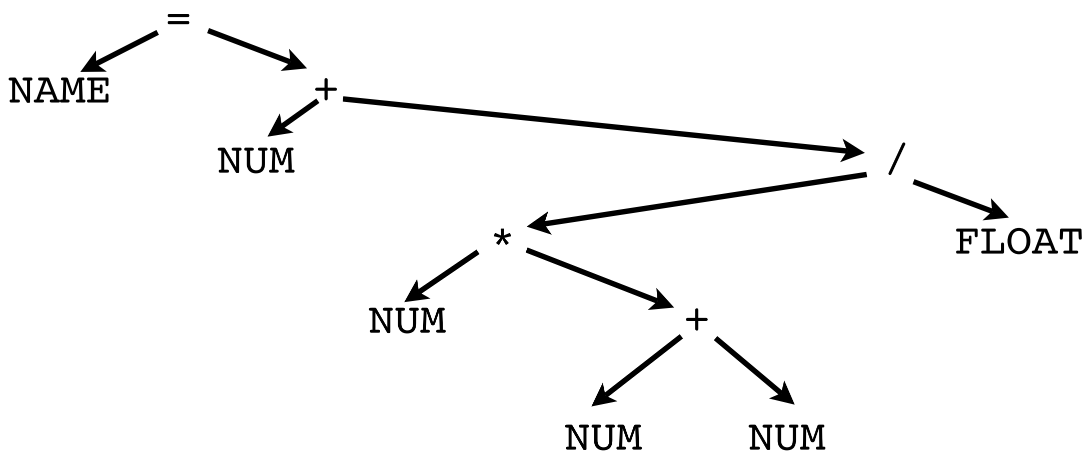
Parsing in a Nutshell

- Lexing : Input is split into tokens



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

- Parsing : Applying language grammar rules



=

NAME

+

NUM

/

*

FLOAT

NUM

+

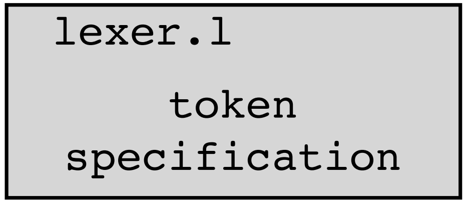
NUM

NUM

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)
- Variations of both tools are widely known
- Covered in compilers classes and textbooks

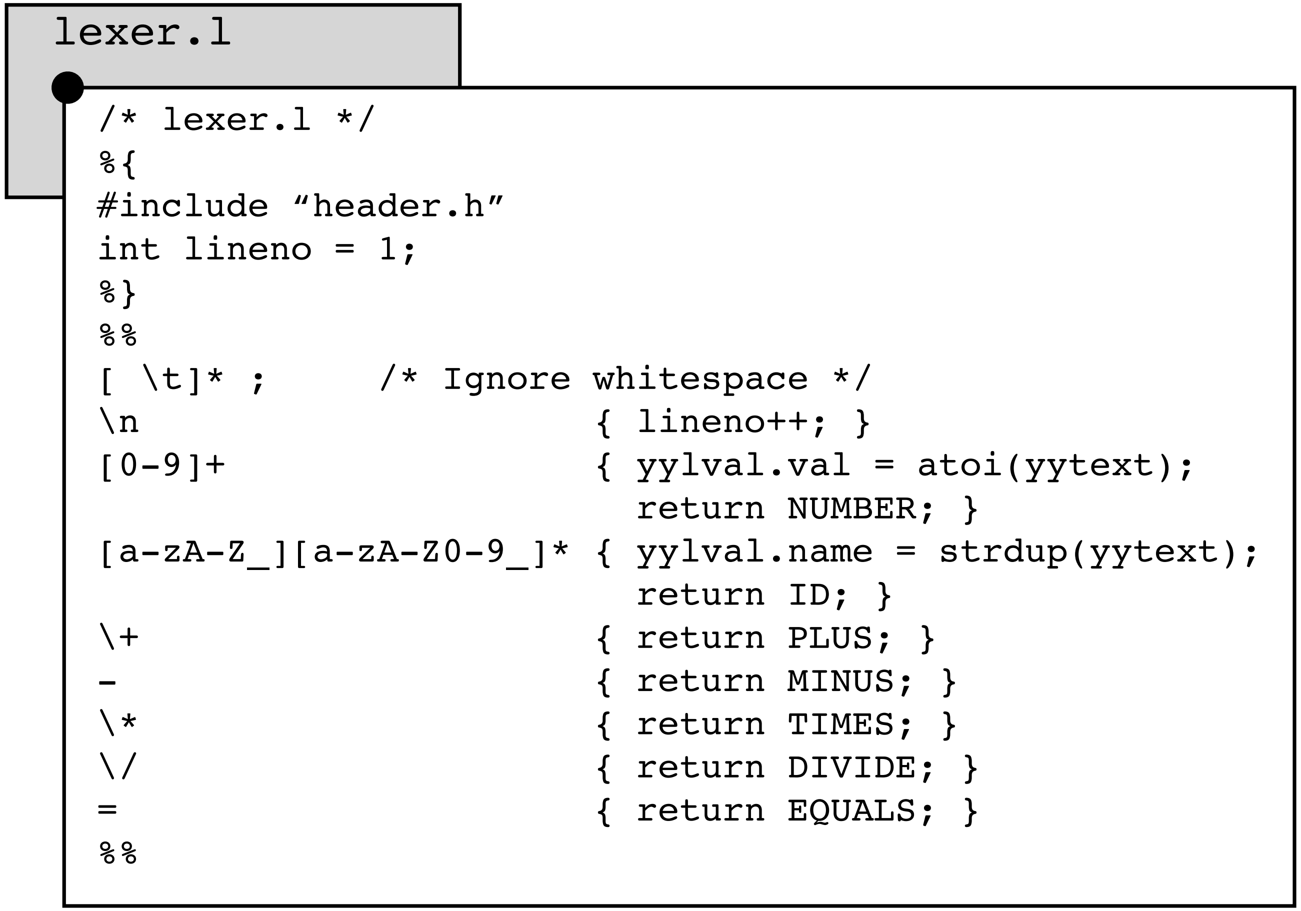
Lex/Yacc Big Picture



lexer.l

token specification

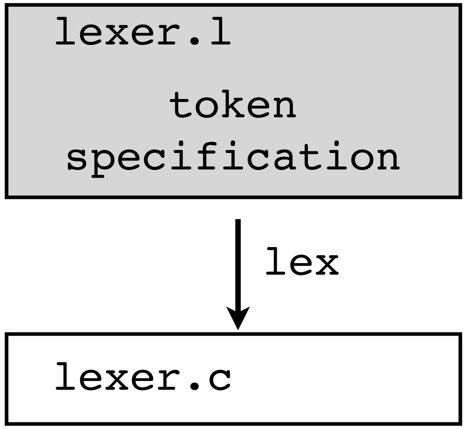
Lex/Yacc Big Picture



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



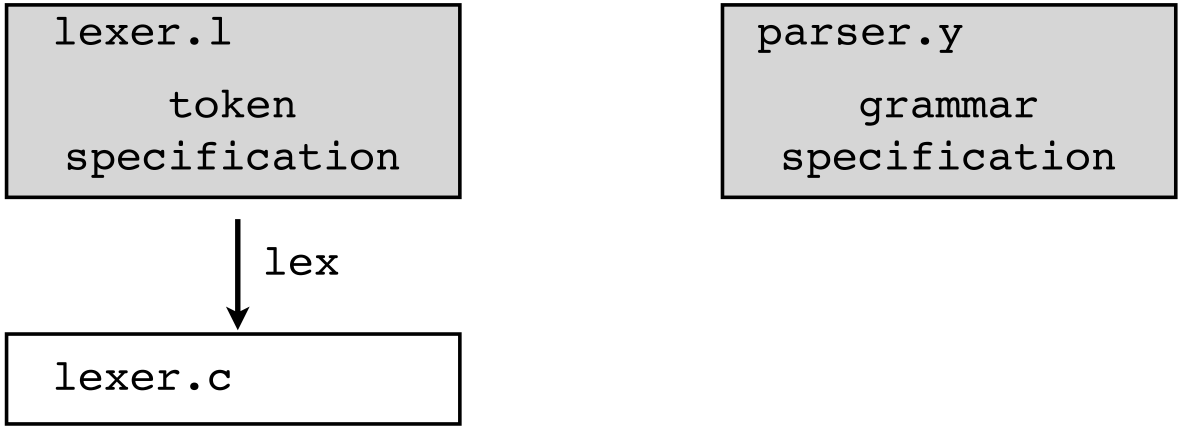
lexer.l

token specification

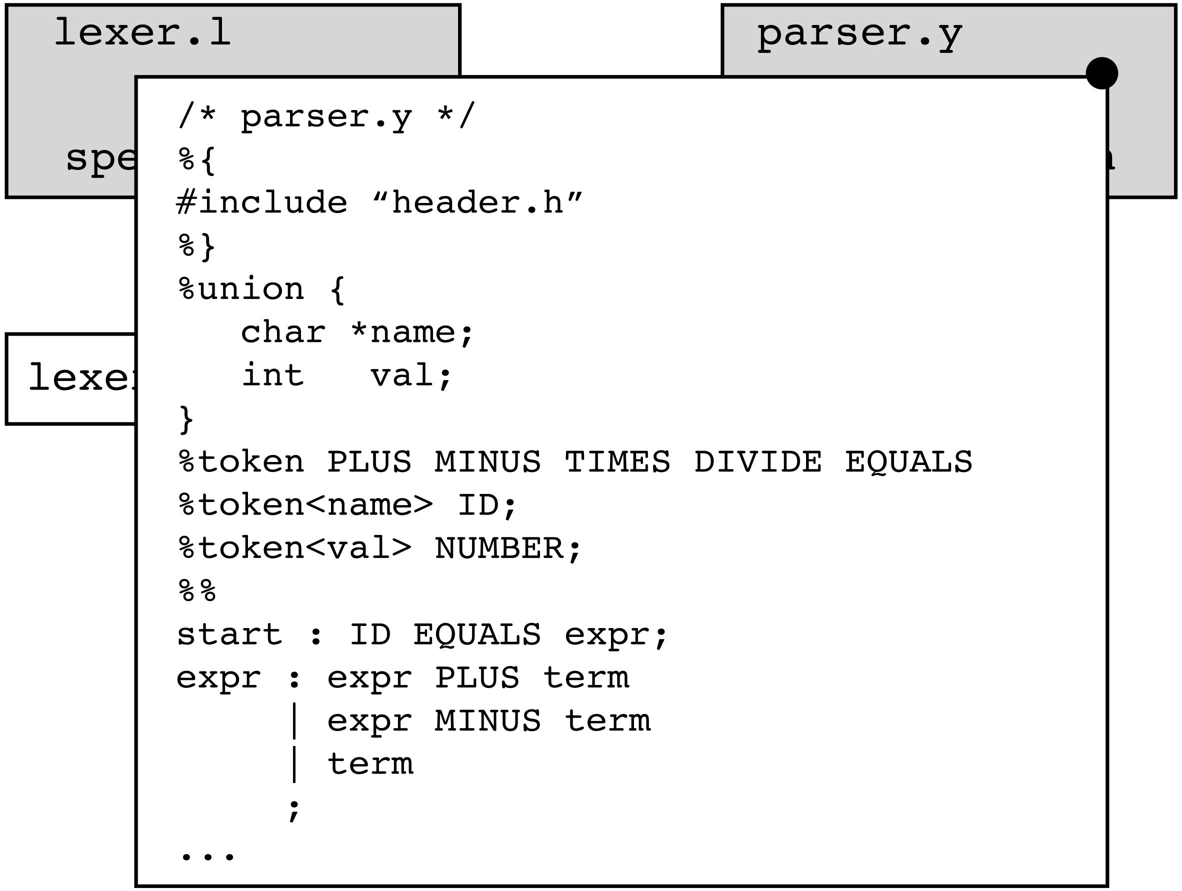
lex

lexer.c

Lex/Yacc Big Picture

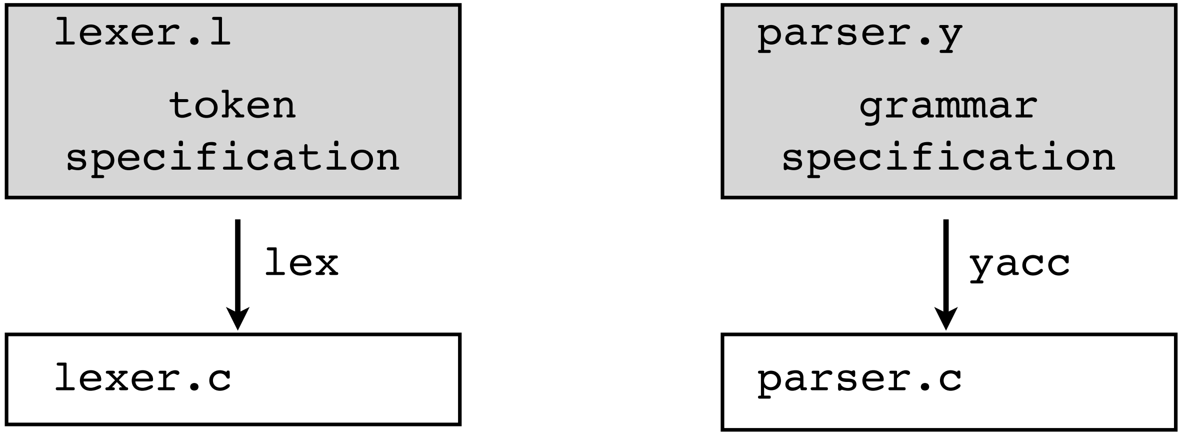


Lex/Yacc Big Picture

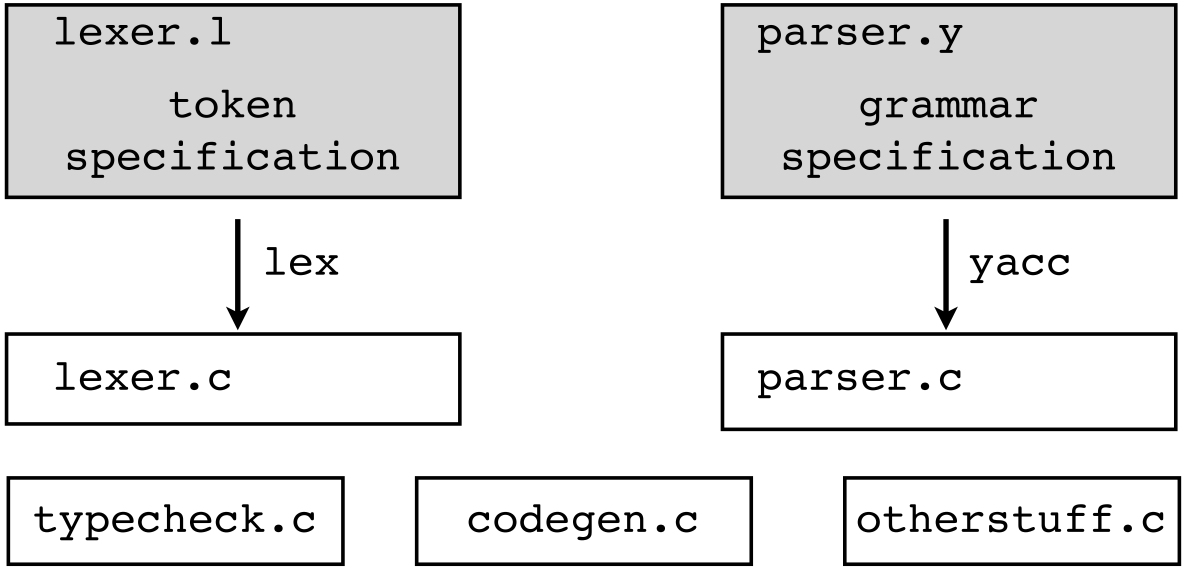


```
/* parser.y */
%{
#include "header.h"
%}
%union {
   char *name;
   int   val;
}
%token PLUS MINUS TIMES DIVIDE EQUALS
%token<name> ID;
%token<val> NUMBER;
%%
start : ID EQUALS expr;
expr : expr PLUS term
     | expr MINUS term
     | term
     ;
...
```

Lex/Yacc Big Picture



Lex/Yacc Big Picture



lexer.l

token specification

lex

lexer.c

parser.y

grammar specification

yacc

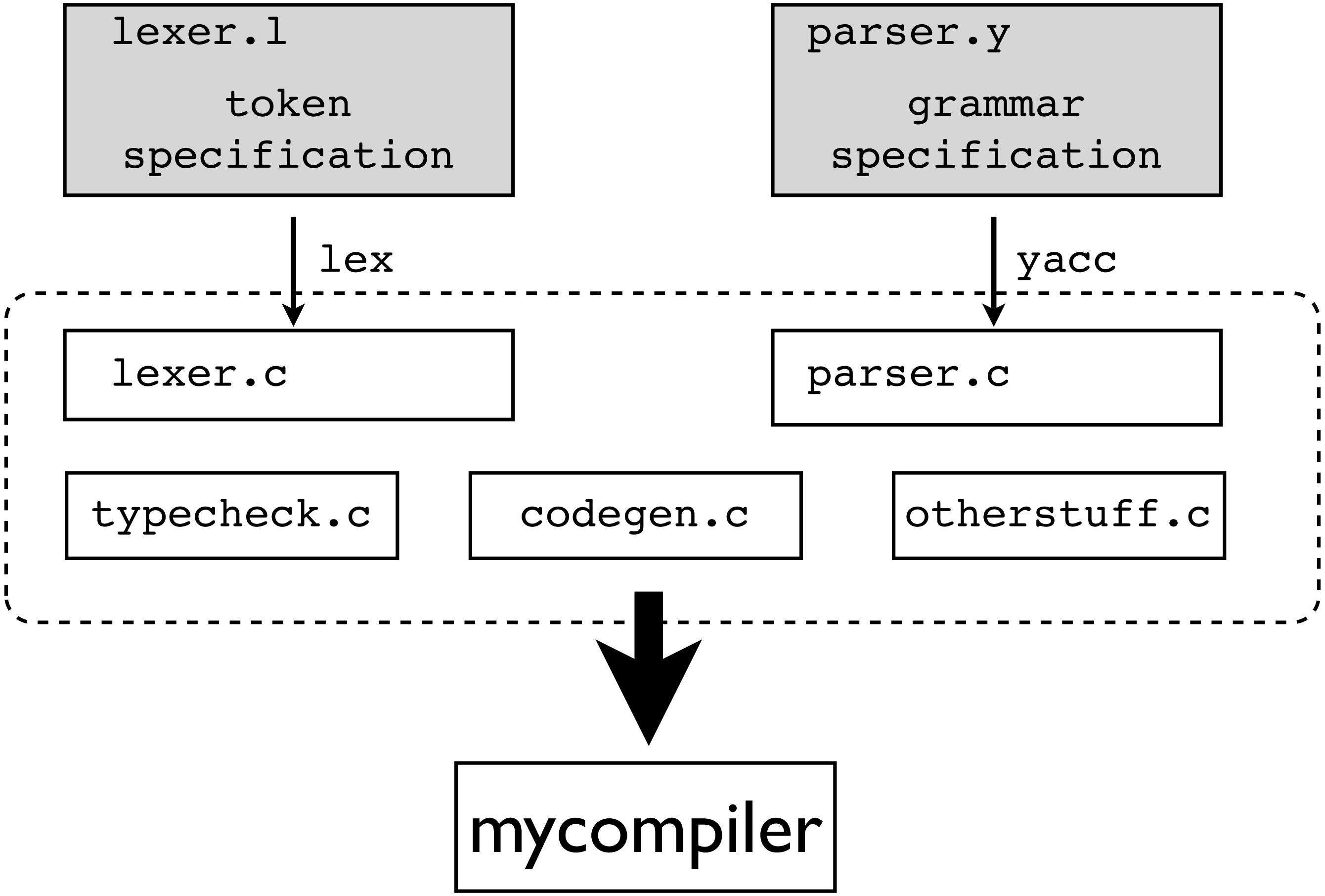
parser.c

typecheck.c

codegen.c

otherstuff.c

Lex/Yacc Big Picture



lexer.l

token specification

parser.y

grammar specification

lex

yacc

lexer.c

parser.c

typecheck.c

codegen.c

otherstuff.c

mycompiler

What is PLY?

- PLY = Python Lex-Yacc
- A Python version of the lex/yacc toolset
- Same functionality as lex/yacc
- But a different interface
- Influences : Unix yacc, SPARK (John Aycock)

Some History

- Late 90's : "Why isn't SWIG written in Python?"
- 2001 : Taught a compilers course. Students write a compiler in Python as an experiment.
- 2001 : PLY-1.0 developed and released
- 2001-2005: Occasional maintenance
- 2006 : Major update to PLY-2.x.

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
```

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()

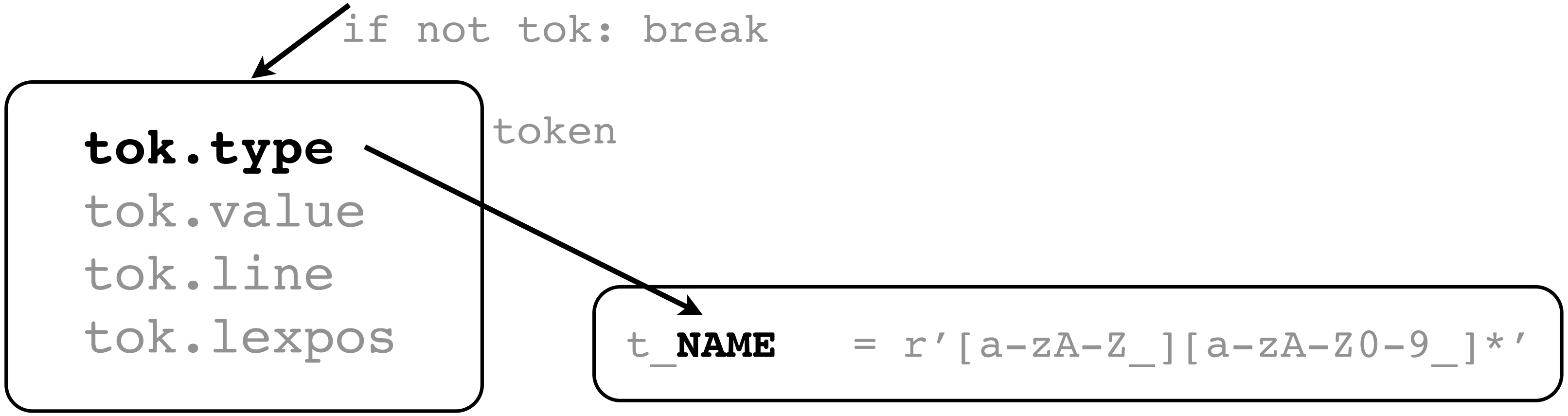
```
...
lex.lex()         # Build the lexer
...
lex.input("x = 3 * 4 + 5 * 6")
while True:
     tok = lex.token()
     if not tok: break

           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
```



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

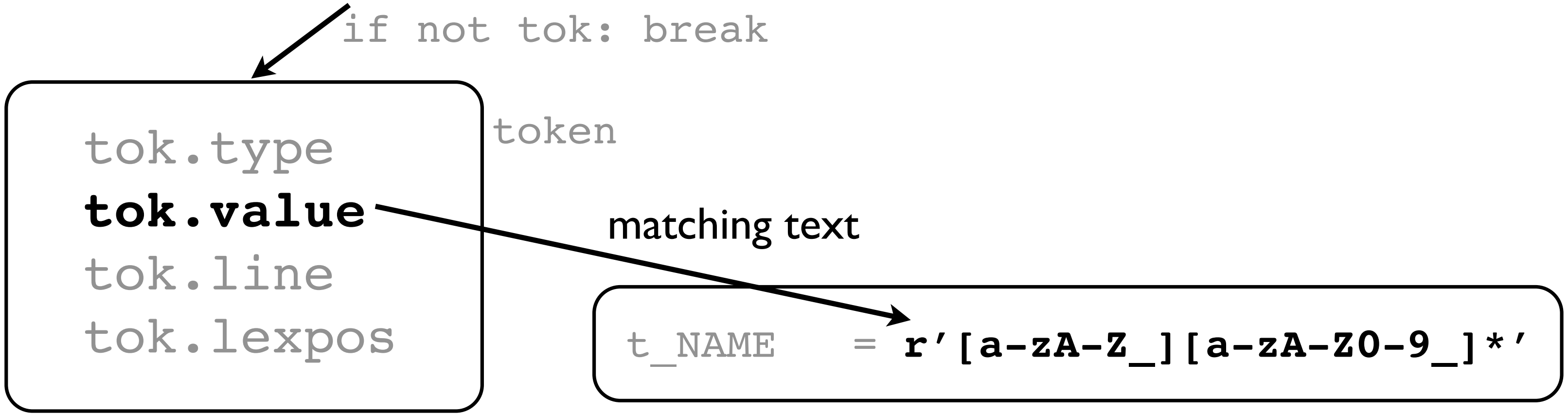
token

```
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

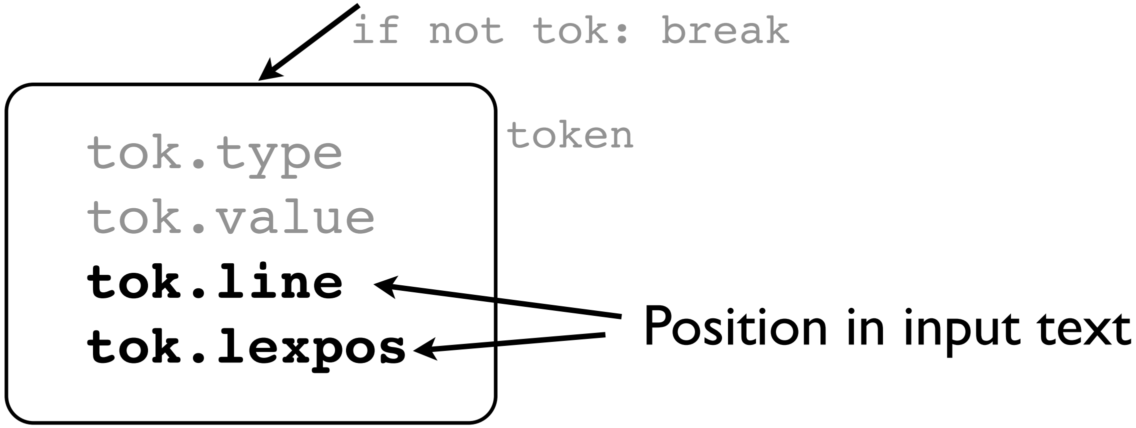
matching text

```
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