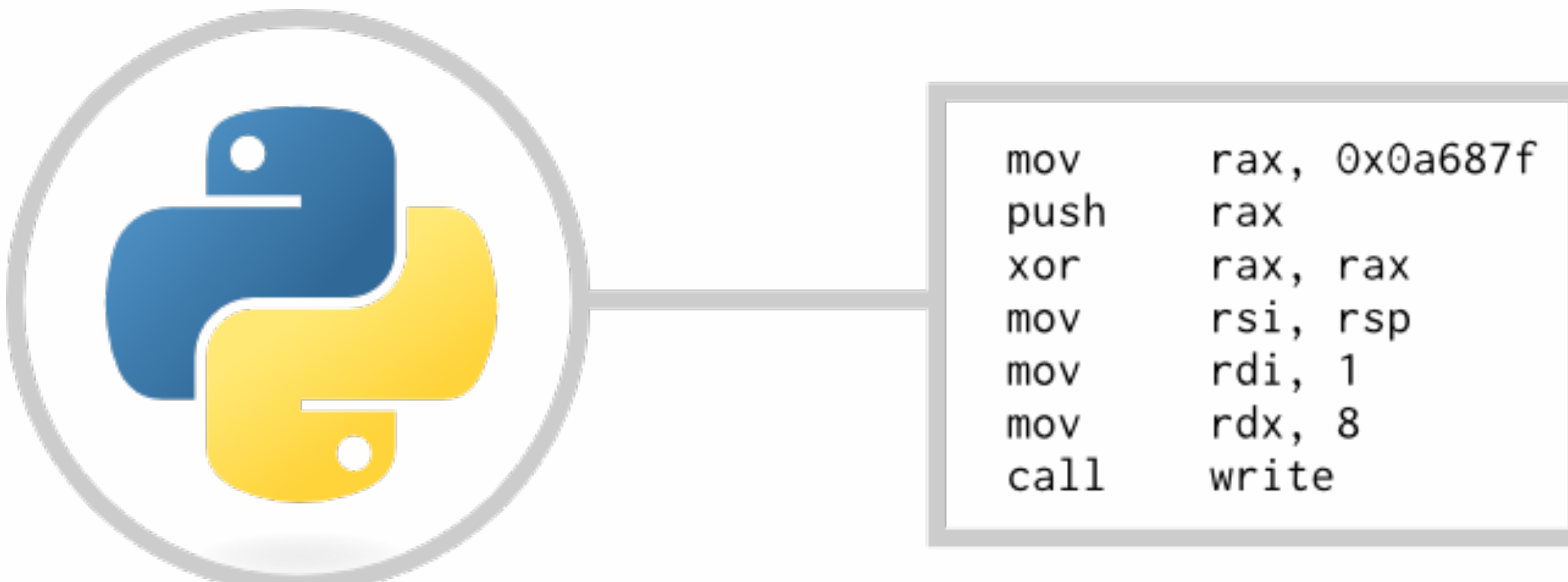
CS 480

Translators (Compilers)



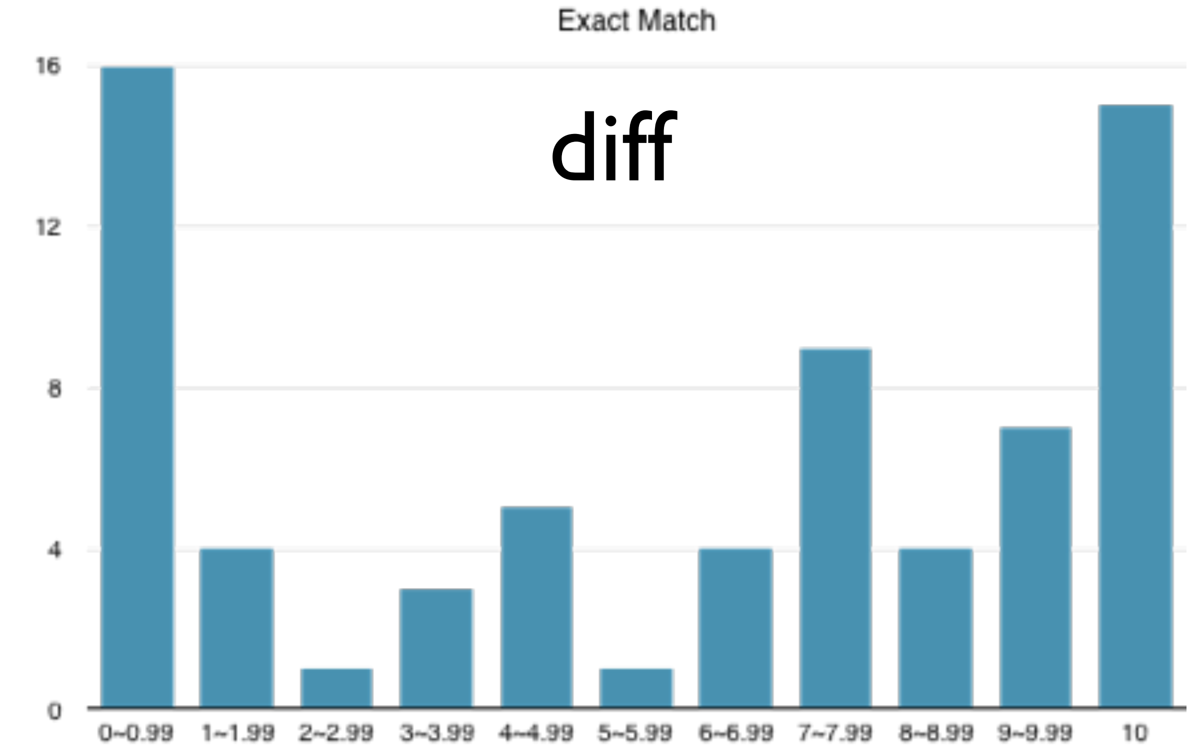
weeks 2-3: SDT (cont'd), cython, lexer and lex/yacc

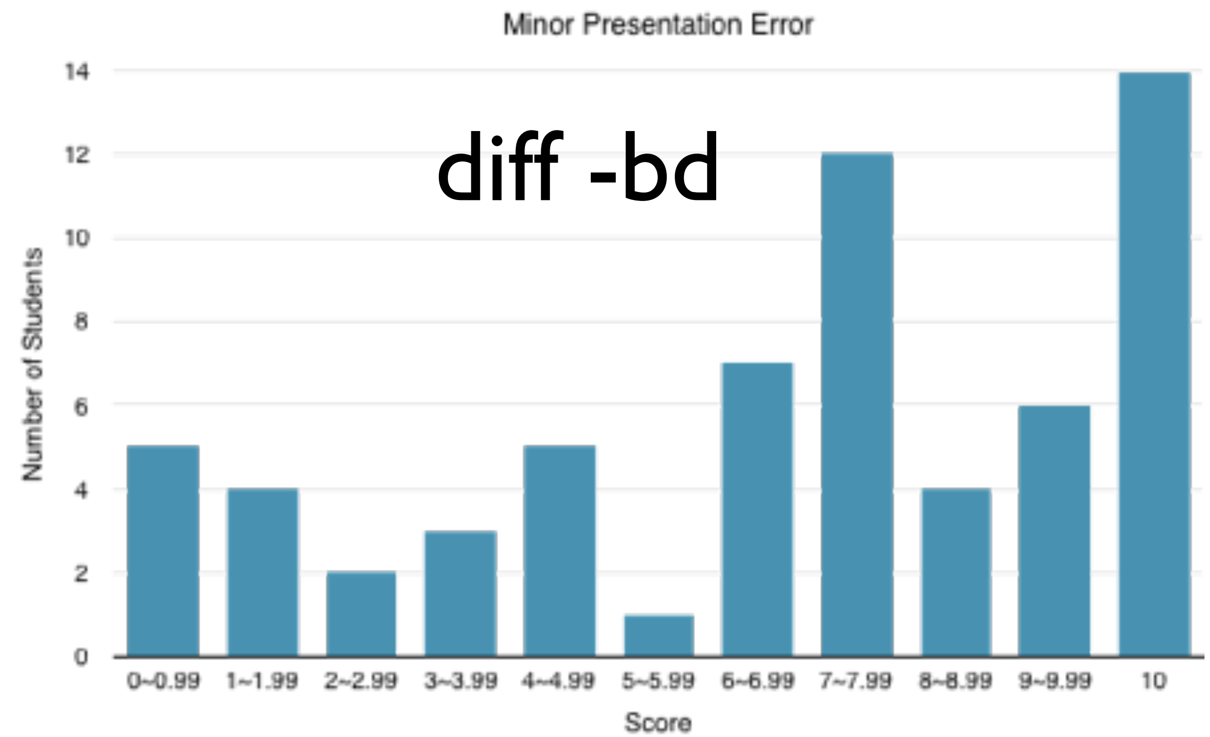
Instructor: Liang Huang

(some slides courtesy of David Beazley)

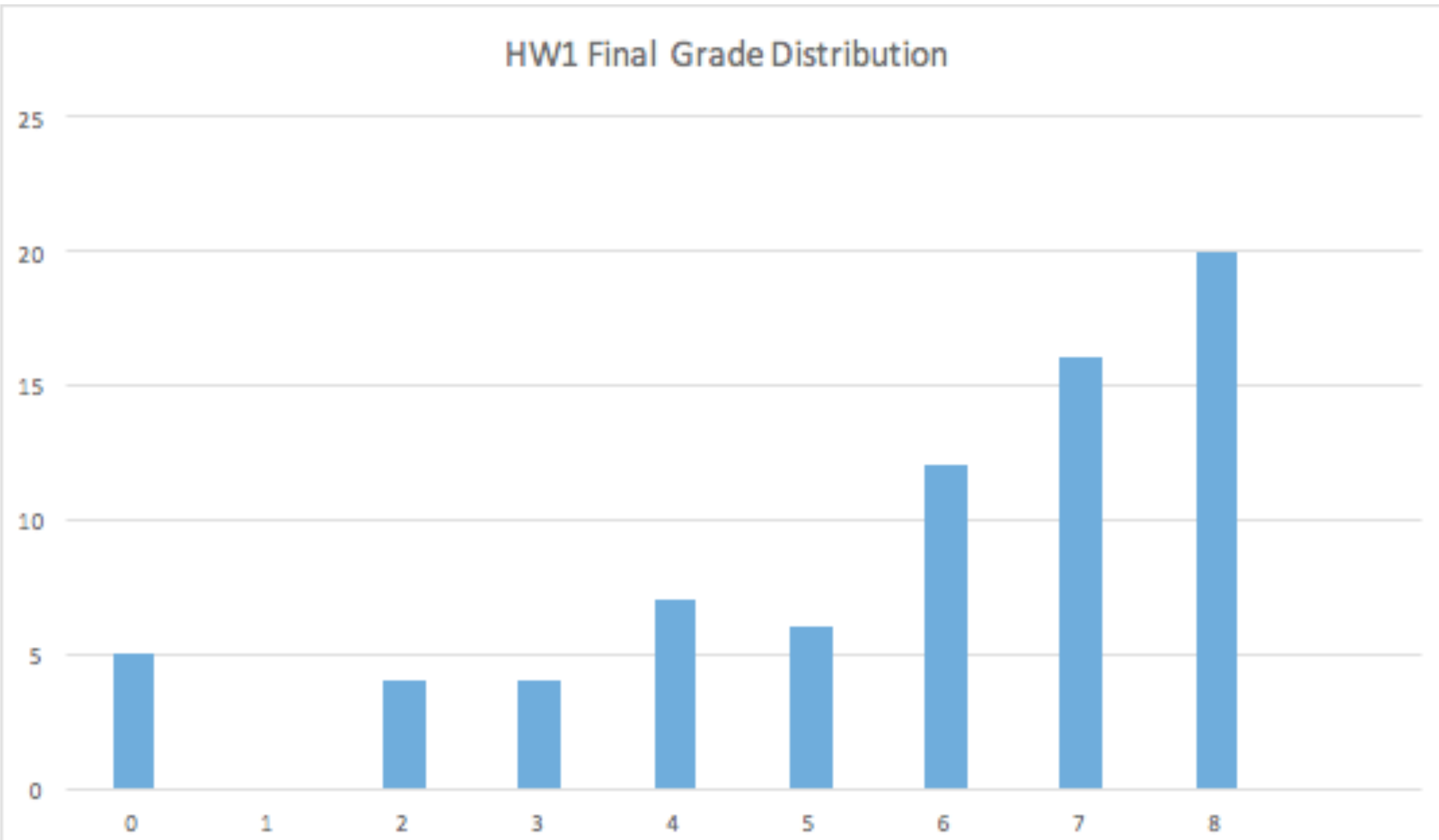
HW1 caveats & coding grades

- scope: either declare all vars upfront, or before for-loop
- auxiliary loop variable takes care of lots of corner cases
  - ok for changing i in the loop
  - no need for i-- after loop
  - ok with range(0)
- also need to cache range limit
- generous partial grades for mistakes in printing multi-items





HW1 overall grade



Switching to `ast` from `compiler`

- the `compiler` packages is deprecated; replaced by `ast`
- similar structure, but easier to use, and faster

```
>>> import compiler
>>> compiler.parse("if 3==4: print 5")
Module(None, Stmt([If([(Compare(Const(3), [('==', Const(4))]),
Stmt([Printnl([Const(5)], None)]))], None)]))

>>> compiler.parse("if 3==4: print 5").node.nodes[0].__dict__
{'tests': [(Compare(Const(3), [('==', Const(4))]),
Stmt([Printnl([Const(5)], None)]))], 'else_': None, 'lineno': 1}
```

```
>>> import ast
>>> ast.dump(ast.parse("if 3==4: print 5"))
'Module(body=[If(test=Compare(left=Num(n=3), ops=[Eq()],
comparators=[Num(n=4)]), body=[Print(dest=None, values=[Num(n=5)],
nl=True)], orelse=[])])’
```

http://greentreesnakes.readthedocs.org/en/latest/

Switching to `ast` from `compiler`

- the `compiler` packages is deprecated; replaced by `ast`
- similar structure, but easier to use, and faster

https://docs.python.org/2/library/ast.html

CFG for AST

```
mod = Module(stmt* body)

stmt = Assign(expr* targets, expr value)
     | Print(..., expr* values, ...)
     | For(expr target, expr iter, stmt* body, ...)
     | If(expr test, stmt* body, ...)
     ...

expr = BoolOp(boolop op, expr* values)
     | BinOp(expr left, operator op, expr right)
     | UnaryOp(unaryop op, expr operand)
     ...
```

nonterminals only
(*abstract* syntax tree)

CFG for Python

```
module: (NEWLINE | stmt)* ENDMARKER

stmt: simple_stmt | compound_stmt
simple_stmt: small_stmt (';' small_stmt)* [';']
small_stmt: (expr_stmt | print_stmt ...)
expr_stmt: testlist (augassign (yield_expr|testl
                    ('=' (yield_expr|testlist)
augassign: ('+=' | '-=' | '*=' | '/=' | '%=' |
            '<<=' | '>>=' | '**=' | '//=')
print_stmt: 'print' ( [ test (',' test)* [','] 

compound_stmt: if_stmt | while_stmt | for_stmt
if_stmt: 'if' test ':' suite ('elif' test ':' su
while_stmt: 'while' test ':' suite ['else' ':' s
for_stmt: 'for' exprlist 'in' testlist ':' suite
suite: simple_stmt | NEWLINE INDENT stmt+ DEDENT
...
```

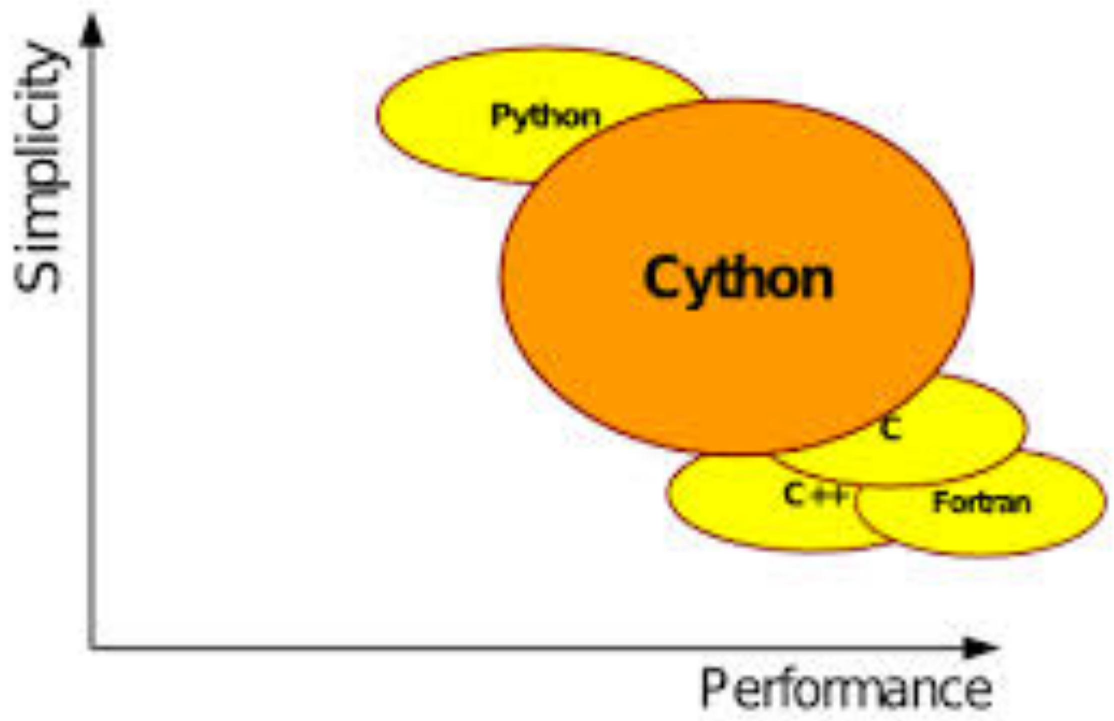
nonterminals and terminals

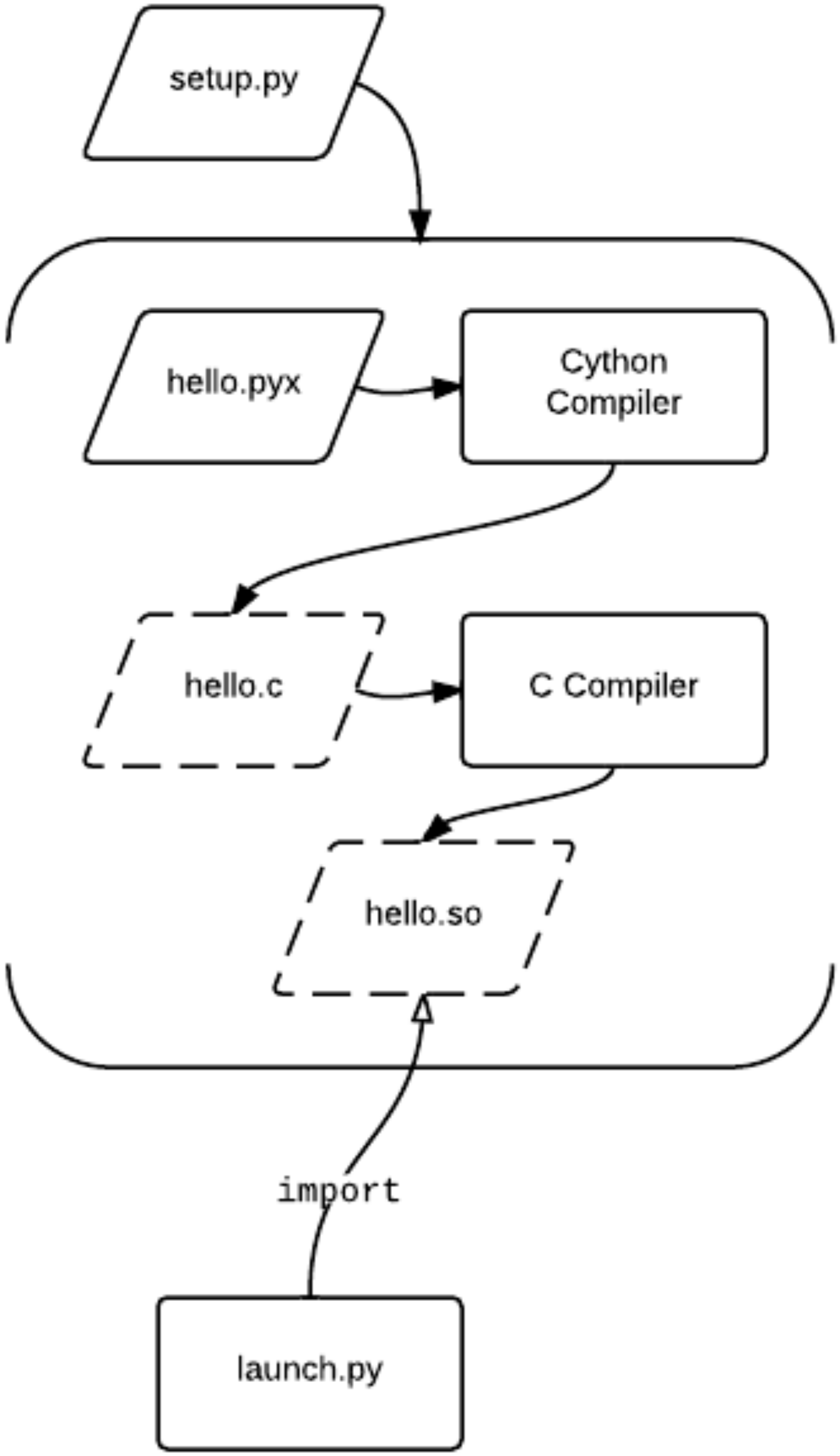
HW2 Grammar (P_2 subset)

```
program : module
module : stmt+
stmt : (simple_stmt | if_stmt | for_stmt) NEWLINE

simple_stmt : "print" expr ("," expr)*
            | int_name "=" int_expr
            | bool_name "=" bool_expr

expr : int_expr | bool_expr

if_stmt : "if" bool_expr ":" (simple_stmts | suite)
for_stmt : "for" name "in" "range" "(" int_expr ")" ":" (simple_stmts | suite)

simple_stmts : simple_stmt (";" simple_stmt)+
suite : NEWLINE INDENT stmt+ DEDENT

int_expr : int_name
         | decint
         | "-" int_expr
         | int_expr "+" int_expr
         | "(" int_expr ")"
         | int_expr "if" bool_expr "else" int_expr

bool_expr : bool_name
          | bool_expr "and" bool_expr
          | bool_expr "or" bool_expr
          | "not" bool_expr
          | "(" bool_expr ")"
          | int_expr (comp_op int_expr)+
          | "True"
          | "False"
          | "(" bool_expr "if" bool_expr "else" bool_expr ")"

comp_op : '<' | '>' | '==' | '>=' | '<=' | '<>' | '!='
```

- HW2: translate P_2 into C
  - expand hw1.py
  - no recursive type analysis
  - caution on nested loops
- HW3: use PLY (lex/yacc) to build a lexer/parser to replace compiler.parse()

HW1/2 Motivations: Cython





- Cython (2007~): compile a large Python subset into C
- gradually being replaced by PyPy (just-in-time compilation)

```
cython --embed $1.py
clang -Os -I /usr/include/python2.7 -lpython2.7 -lpthread -lm -lutil -ldl $1.c -o $1.out
```

Python int vs C int

C/C++	int	long	long long
32-bit	32		64
64-bit	32	64	

- Python has arbitrary precision integer arithmetic (<type ‘long’>)
  - when Python int exceeds ‘int’ range, it becomes ‘long’ and stays ‘long’
- you can turn it off in cython by “cdef int
  - much faster, but overflows

.py

```
import sys
n, m = map(int, sys.argv[1:3])

for _ in xrange(m):
    a, b = 0, 1
    for _ in xrange(n):
        a, b = b, a+b
print "fib[%d] = %d" % (n, b)
```

~15% faster than Python

.pyx

```
import sys
n, m = map(int, sys.argv[1:3])
cdef int a, b
for _ in xrange(m):
    a, b = 0, 1
    for _ in xrange(n):
        a, b = b, a+b
print "fib[%d] = %d" % (n, b)
```

~95% faster than Python,
but overflows!

(this is more like your HW1/2)

Compiled Cython Program (.cpp)

```
def fib(int n):
    l = []
    cdef int a=0, b=1
    for i in range(n):
        l.append(b)
        a, b = b, a+b
    return l
```

```
/* Generated by Cython 0.14 on Wed Jun  1 15:38:37 2011 */

#include "Python.h"

...

  /* "/Users/lhuang/install/svector/tutorial/fib_cy.pyx":2
 * def fib(int n):
 *     l = []             # <<<<<<<<<<<<<<
 *     cdef int a=0, b=1
 *     for i in range(n):
 */
  __pyx_t_1 = PyList_New(0); if (unlikely(!__pyx_t_1)) {__pyx_filename = __pyx_f[0];
__pyx_lineno = 2; __pyx_clineno = __LINE__; goto __pyx_L1_error;}
  __Pyx_GOTREF(((PyObject *)__pyx_t_1));
  __Pyx_DECREF(((PyObject *)__pyx_v_l));
  __pyx_v_l = __pyx_t_1;
  __pyx_t_1 = 0;

  /* "/Users/lhuang/install/svector/tutorial/fib_cy.pyx":3
 * def fib(int n):
 *     l = []
 *     cdef int a=0, b=1             # <<<<<<<<<<<<<<
 *     for i in xrange(n):
 *         l.append(b)
 */
  __pyx_v_a = 0;
  __pyx_v_b = 1;
```

Compiled Cython Program (.cpp)

```
/* "/Users/lhuang/install/svector/tutorial/fib_cy.pyx":4
 *     l = []
 *     cdef int a=0, b=1
 *     for i in range(n):             # <<<<<<<<<<<<<<
 *         l.append(b)
 *         a, b = b, a+b
 */
  __pyx_t_2 = __pyx_v_n;
  for (__pyx_t_3 = 0; __pyx_t_3 < __pyx_t_2; __pyx_t_3+=1) {
    __pyx_v_i = __pyx_t_3;

/* "/Users/lhuang/install/svector/tutorial/fib_cy.pyx":5
 *     cdef int a=0, b=1
 *     for i in xrange(n):
 *         l.append(b)             # <<<<<<<<<<<<<<
 *         a, b = b, a+b
 *     return l
 */
    if (unlikely(__pyx_v_l == Py_None)) {
      PyErr_SetString(PyExc_AttributeError, "'NoneType' object has no attribute 'append'");
{__pyx_filename = __pyx_f[0]; __pyx_lineno = 5; __pyx_clineno = __LINE__; goto
__pyx_L1_error;}      }
    __pyx_t_1 = PyInt_FromLong(__pyx_v_b); if (unlikely(!__pyx_t_1)) {__pyx_filename =
__pyx_f[0]; __pyx_lineno = 5; __pyx_clineno = __LINE__; goto __pyx_L1_error;}
    __Pyx_GOTREF(__pyx_t_1);
    __pyx_t_4 = PyList_Append(__pyx_v_l, __pyx_t_1); if (unlikely(__pyx_t_4 == -1))
{__pyx_filename = __pyx_f[0]; __pyx_lineno = 5; __pyx_clineno = __LINE__; goto __pyx_L1_error;}
    __Pyx_DECREF(__pyx_t_1); __pyx_t_1 = 0;
```

*very clever: for loop detected! but should always use **xrange** in your .pyx or .py!*

Compiled Cython Program (.cpp)

```
/* "/Users/lhuang/install/svector/tutorial/fib_cy.pyx":6
 *     for i in xrange(n):
 *         l.append(b)
 *         a, b = b, a+b              # <<<<<<<<<<<<<<
 *     return l
 */
    __pyx_t_4 = __pyx_v_b;
    __pyx_t_5 = (__pyx_v_a + __pyx_v_b);
    __pyx_v_a = __pyx_t_4;
    __pyx_v_b = __pyx_t_5;
  }

...
```

correctly handles simultaneous assignment

```
static PyObject *__pyx_pf_6fib_cy_0fib(PyObject *__pyx_self, PyObject *__pyx_arg_n) {

  __pyx_v_n = __Pyx_PyInt_AsInt(__pyx_arg_n);
  __pyx_t_1 = PyList_New(0);
  __pyx_v_a = 0;
  __pyx_v_b = 1;

  for (__pyx_t_3 = 0; __pyx_t_3 < __pyx_t_n; __pyx_t_3+=1) {

    __pyx_t_1 = PyInt_FromLong(__pyx_v_b);
    __pyx_t_4 = PyList_Append(__pyx_v_l, __pyx_t_1);
    __Pyx_DECREF(__pyx_t_1);

    __pyx_t_4 = __pyx_v_b;
    __pyx_t_5 = (__pyx_v_a + __pyx_v_b);
    __pyx_v_a = __pyx_t_4;
    __pyx_v_b = __pyx_t_5;
  }

  ...
}
```

By Comparison... using python int

```python
import sys
n, m = map(int, sys.argv[1:3])

for _ in xrange(m):
    a, b = 0, 1
    for _ in xrange(n):
        a, b = b, a+b
print "fib[%d] = %d" % (n, b)
```

```c
__pyx_t_2 = __Pyx_GetModuleGlobalName(__pyx_n_s_b); if (unlikely(!__pyx_t_2)) {__pyx_filename = __pyx_f[0];
__pyx_lineno = 12; __pyx_clineno = __LINE__; goto __pyx_L1_error;}
      __Pyx_GOTREF(__pyx_t_2);
      __pyx_t_4 = __Pyx_GetModuleGlobalName(__pyx_n_s_a); if (unlikely(!__pyx_t_4)) {__pyx_filename = __pyx_f[0]
__pyx_lineno = 12; __pyx_clineno = __LINE__; goto __pyx_L1_error;}
      __Pyx_GOTREF(__pyx_t_4);
      __pyx_t_10 = __Pyx_GetModuleGlobalName(__pyx_n_s_b); if (unlikely(!__pyx_t_10)) {__pyx_filename =
__pyx_f[0]; __pyx_lineno = 12; __pyx_clineno = __LINE__; goto __pyx_L1_error;}
      __Pyx_GOTREF(__pyx_t_10);
      __pyx_t_11 = PyNumber_Add(__pyx_t_4, __pyx_t_10); if (unlikely(!__pyx_t_11)) {__pyx_filename = __pyx_f[0];
__pyx_lineno = 12; __pyx_clineno = __LINE__; goto __pyx_L1_error;}
      __Pyx_GOTREF(__pyx_t_11);
      __Pyx_DECREF(__pyx_t_4); __pyx_t_4 = 0;
      __Pyx_DECREF(__pyx_t_10); __pyx_t_10 = 0;
      if (PyDict_SetItem(__pyx_d, __pyx_n_s_a, __pyx_t_2) < 0) {__pyx_filename = __pyx_f[0]; __pyx_lineno = 12;
__pyx_clineno = __LINE__; goto __pyx_L1_error;}
      __Pyx_DECREF(__pyx_t_2); __pyx_t_2 = 0;
      if (PyDict_SetItem(__pyx_d, __pyx_n_s_b, __pyx_t_11) < 0) {__pyx_filename = __pyx_f[0]; __pyx_lineno = 12;
__pyx_clineno = __LINE__; goto __pyx_L1_error;}
      __Pyx_DECREF(__pyx_t_11); __pyx_t_11 = 0;
```

Lexer within Compiler Pipeline

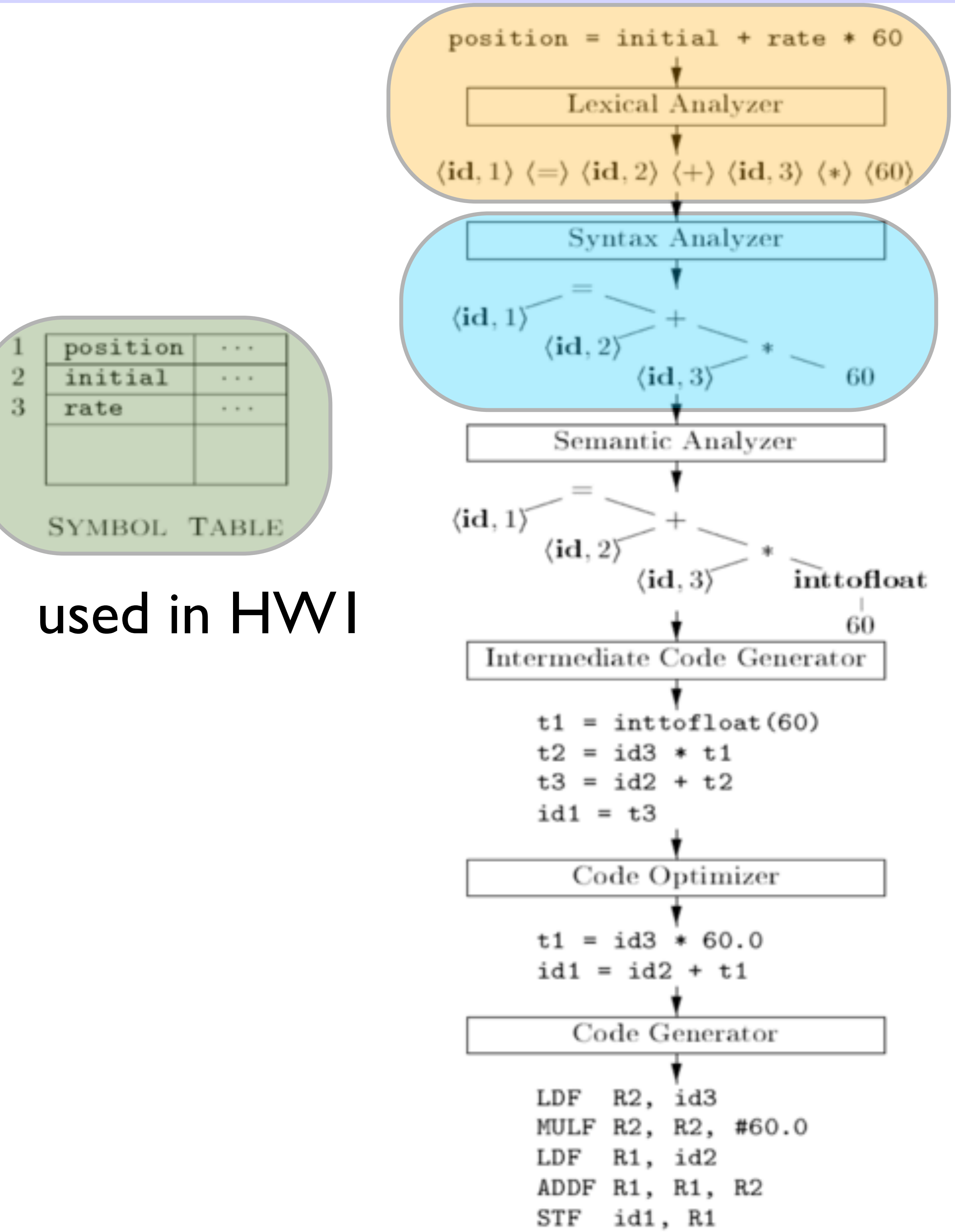
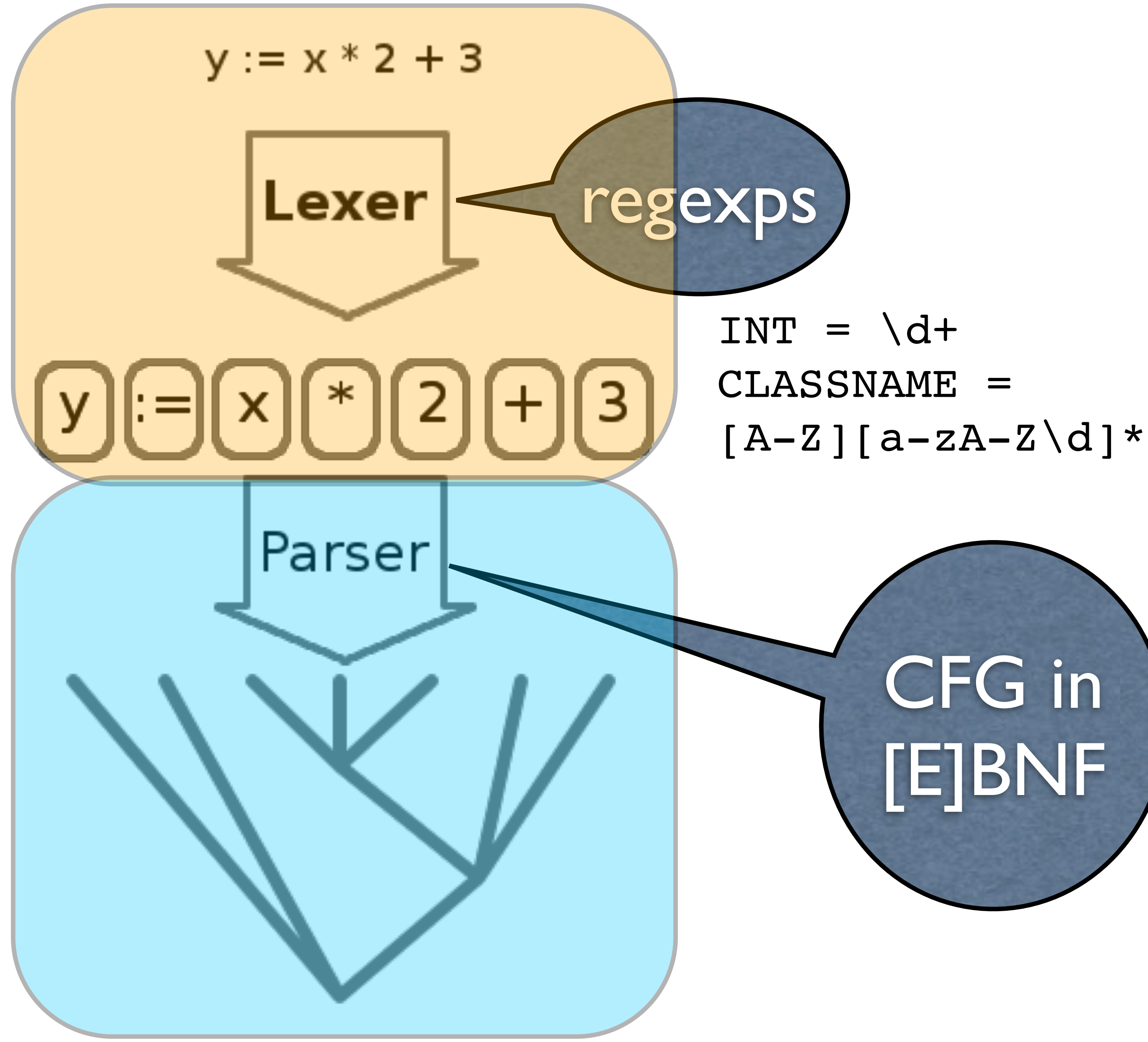


Figure 1.7: Translation of an assignment statement

used in HW1




```
INT = \d+
CLASSNAME =
[A-Z][a-zA-Z\d]*
```

```
simple_stmt : "print" expr ("," expr)*
            | name "=" int_expr
```

Regular Expression Examples

- integers and floats

  `(-?)\d+`   `(-?) ((\d+\.\d*) | (\d*\.\d+))`

- Python/Java/C variable names (starts with letter or _)

  `[a-zA-Z_] [a-zA-Z\d_]*`

- Python variable/function naming convention: joined_lower

  ~~`[a-z_]+`~~   `(_?)([a-z]+)(_[a-z]+)*(_?)`

- Java variable/function naming convention: camelCase

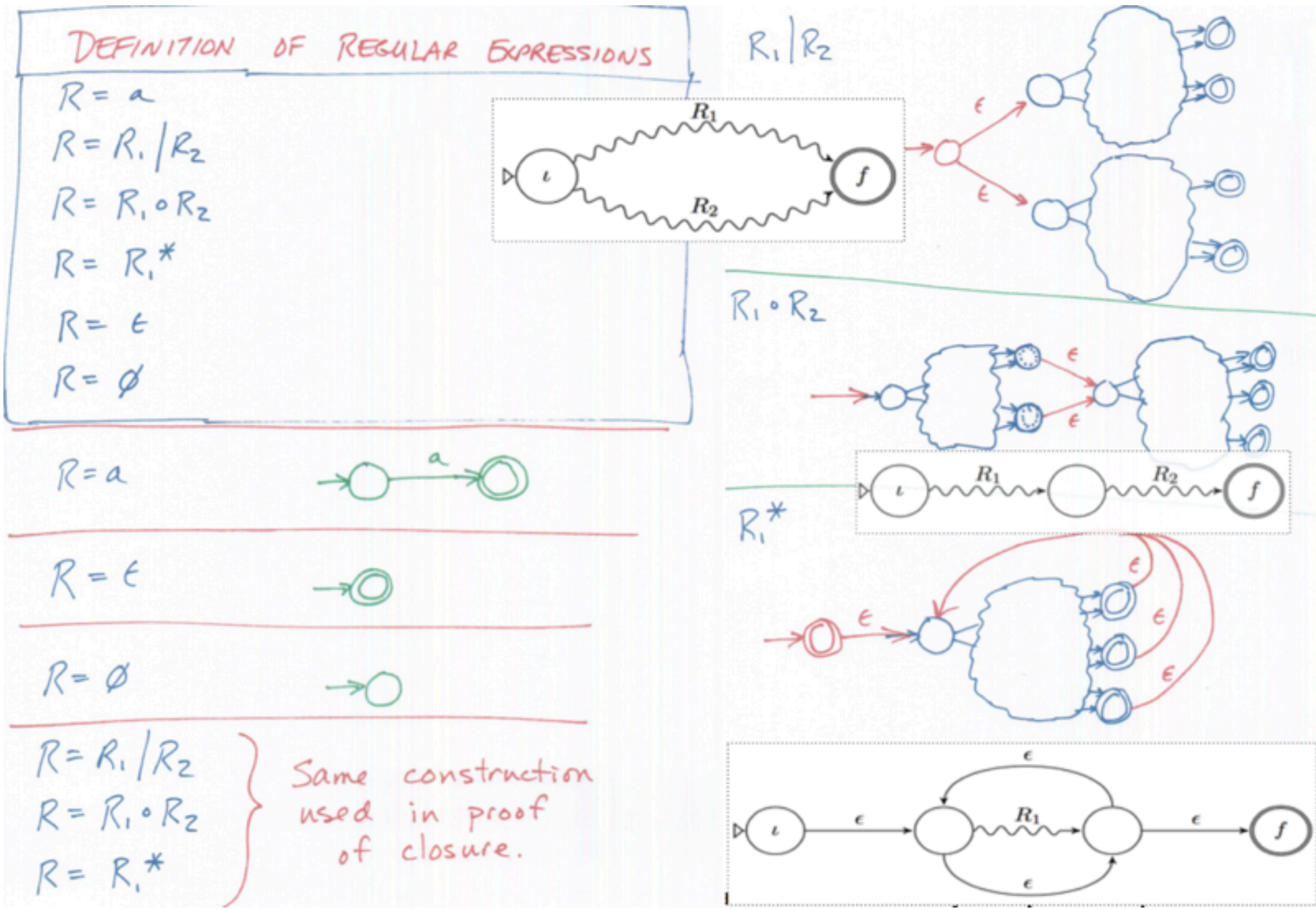
  `[a-z]+([A-Z][a-z]+)+`

- Python/Java class naming convention: StudlyCaps

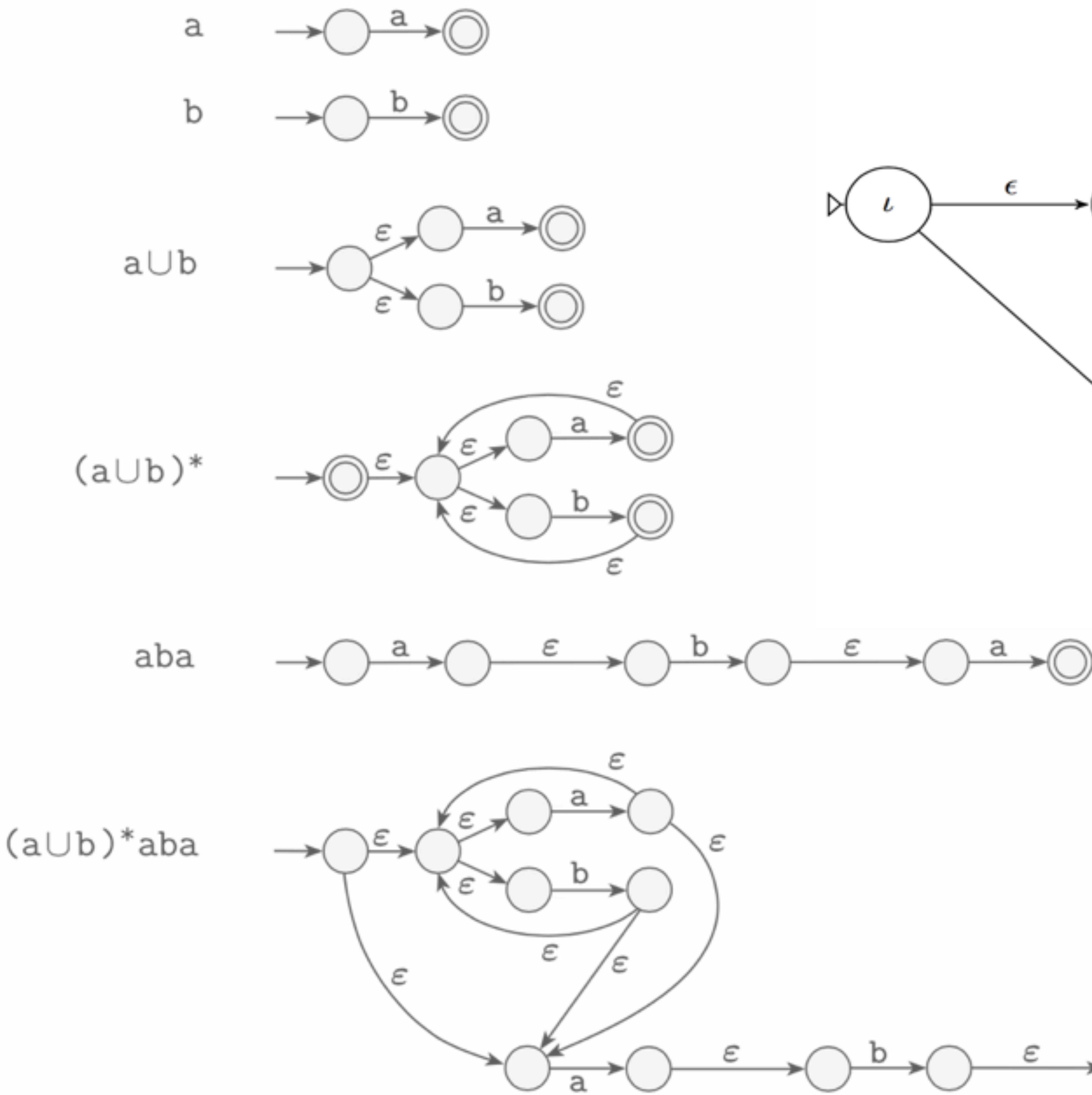
  `([A-Z][a-z]+)+`

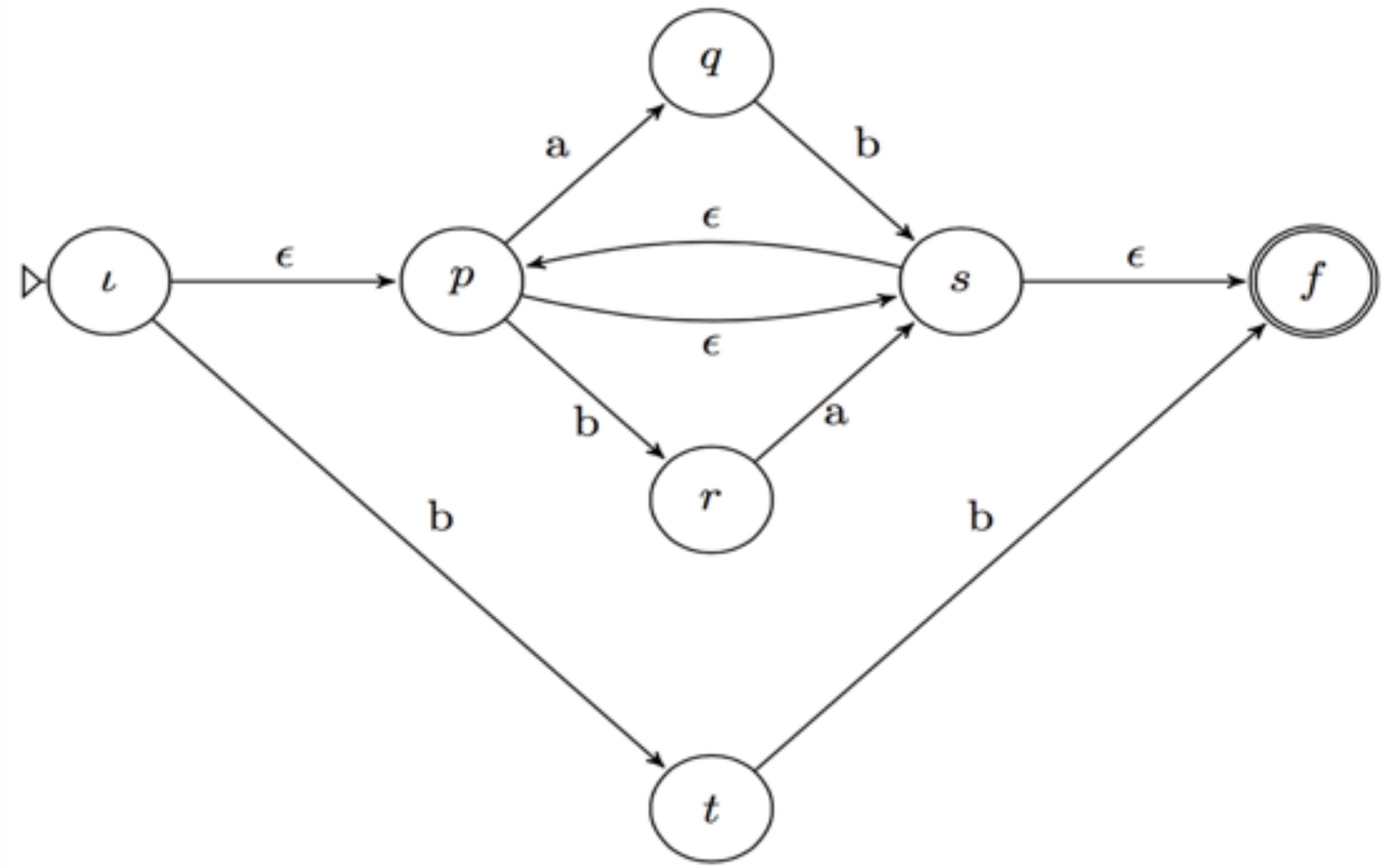
How RegExps are implemented

- regexp => NFA => DFA



Two Examples: $(a|b)^*aba$ $(ab|ba)^*|bb$





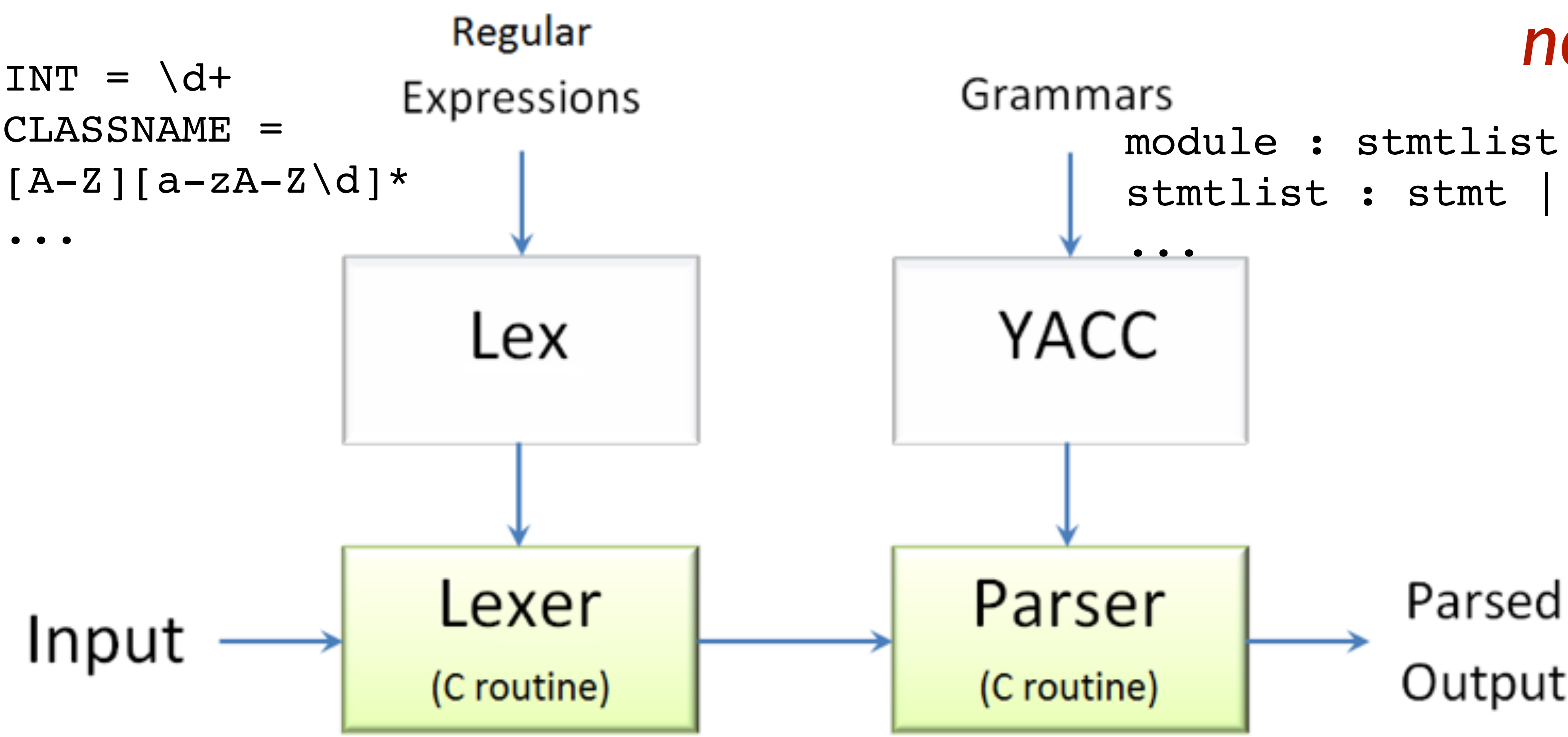
Writing a Lexer is Boring...

- we often use a lexer-generator

only allows BNF not EBNF!

```
INT = \d+
CLASSNAME =
[A-Z][a-zA-Z\d]*
...
```

```
module : stmtlist
stmtlist : stmt | stmt stmtlist
...
```



~1974 ~1973

Writing Parsers and Compilers with PLY (David Beazley)

Lex & Yacc

Google

CEO/chairman of

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

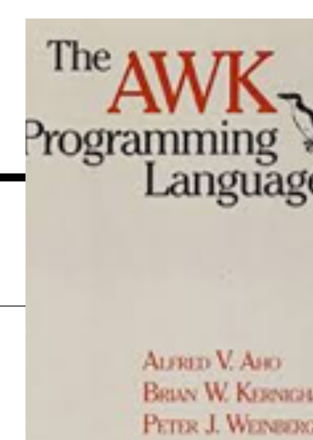
author of

 - 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

intern of

author of



Lex/Yacc Big Picture

lexer.l

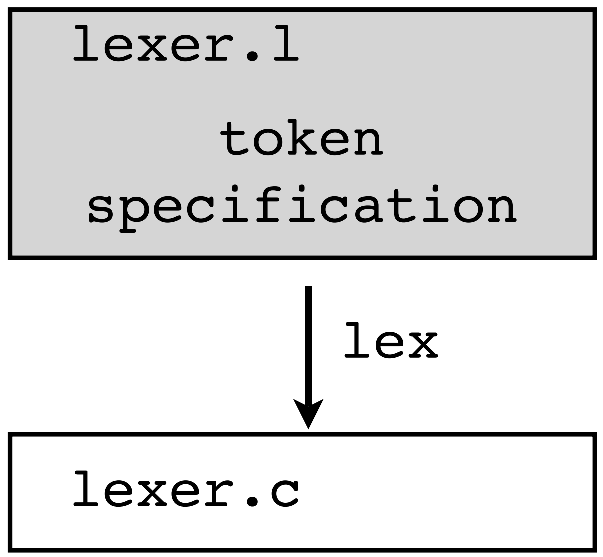
token
specification

Lex/Yacc Big Picture

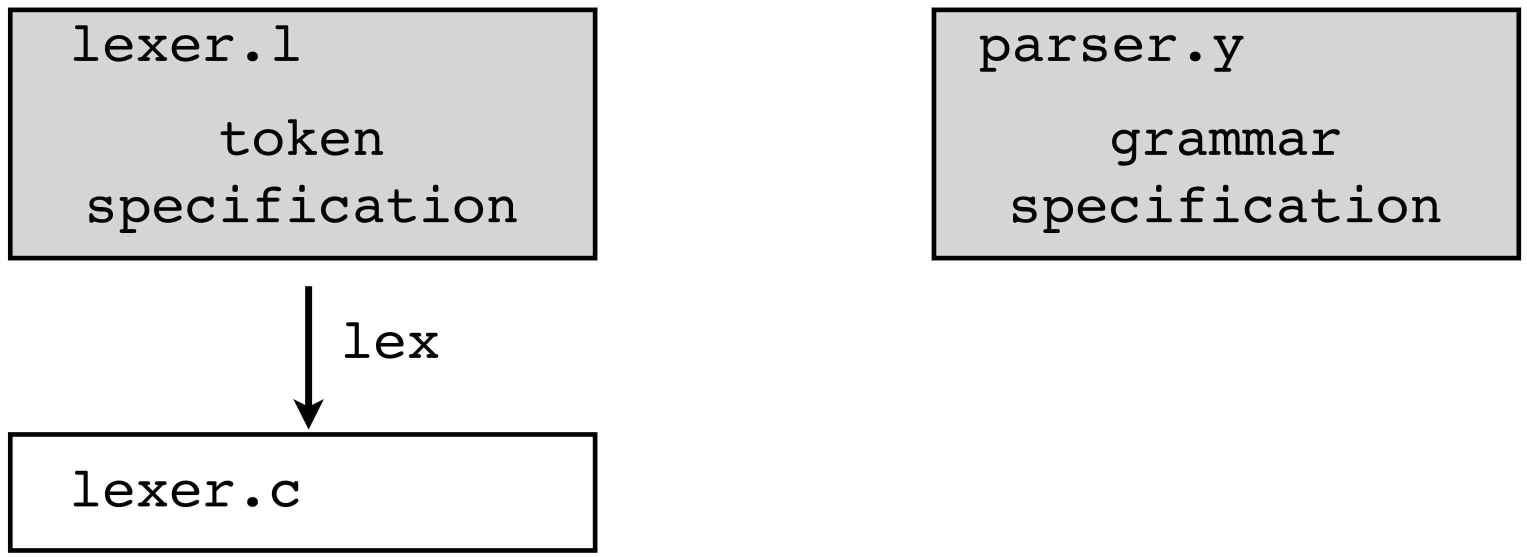
lexer.l

```
/* 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

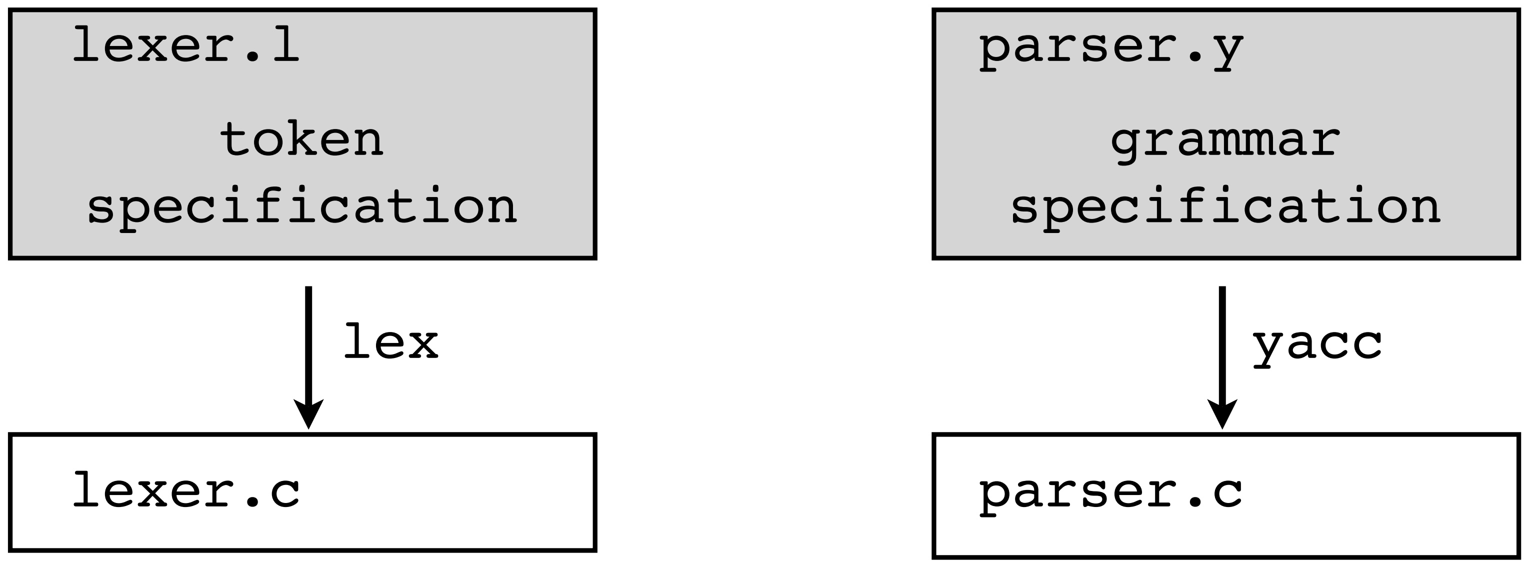
spe

parser.y

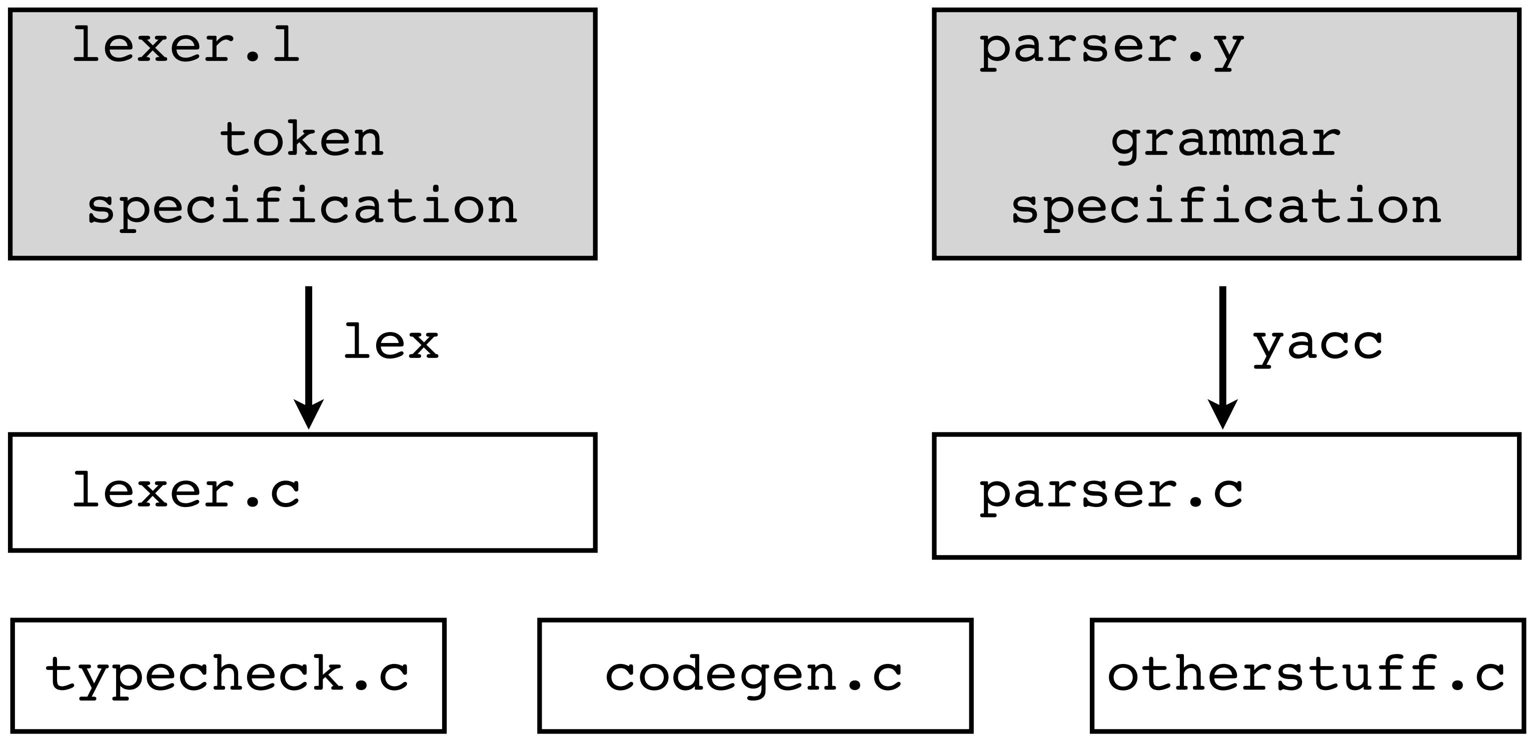
lexe

```
/* 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
     ;
...
```

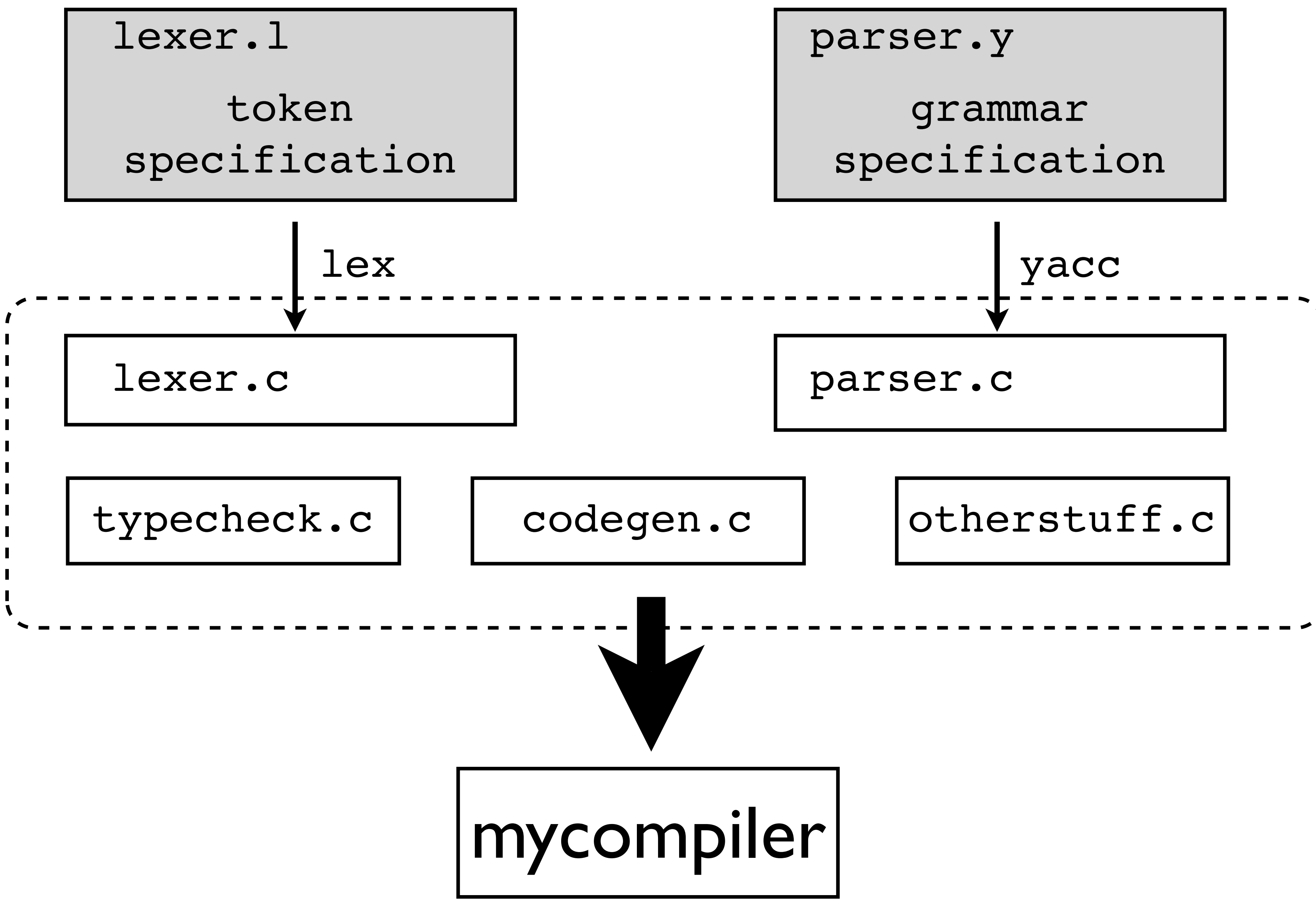
Lex/Yacc Big Picture



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
- Influences : Unix yacc, SPARK (John Aycock)

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
```

REALLY AWFUL INTERFACE!!

SHOULD BE MODERNIZED!!

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
```

token

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

Position in input text

Actually this doesn't work!

- key words will be mixed with identifiers (variable names...)

```
tokens = ('PRINT','INT','PLUS', 'NAME')
t_ignore = ' \t'
t_PRINT = r'print'
t_PLUS = r'\+'
t_NAME = r'[a-zA-Z_][a-zA-Z\d_]*'
```

```
$ echo -e "print a" | ./hw2a.py
(NAME, 'print') (NAME, 'a')
```

```
keywords = {'print' : 'PRINT', ...}

tokens = ['NAME','INT','PLUS'] + keywords.values()
t_ignore = ' \t'
t_PLUS = r'\+'
t_PRINT = r'print'
t_NAME = r'[a-zA-Z_][a-zA-Z\d_]*'

def t_NAME(t):
    r'[a-zA-Z_][a-zA-Z_0-9]*'
    t.type = keywords.get(t.value, 'NAME')  # Check for reserved words
    return t
```

```
$ echo -e "print a" | ./hw2b.py
(PRINT, 'print') (NAME, 'a')
```

Python's INDENT/DEDENT

- this is beyond context-free, and thus way beyond regular!
- use a stack of indentation levels; INDENT/DEDENT is produced whenever indentation level *changes*

```
for i in range(1):
__for j in range(2):
____print i,j
____6
print 5
```

(FOR, 'for')	[0]
(NAME, 'i')	[0]
(IN, 'in')	[0]
(RANGE, 'range')	[0]
(LPAREN, '(')	[0]
(INT, 1)	[0]
(RPAREN, ')')	[0]
(COLON, ':')	[0]
(INDENT, 1)	**[0, 2]**
(FOR, 'for')	[0, 2]
(NAME, 'j')	[0, 2]
(IN, 'in')	[0, 2]
(RANGE, 'range')	[0, 2]
(LPAREN, '(')	[0, 2]
(INT, 2)	[0, 2]
(RPAREN, ')')	[0, 2]
(COLON, ':')	[0, 2]
(INDENT, 1)	**[0, 2, 4]**
(PRINT, 'print')	[0, 2, 4]
(NAME, 'i')	[0, 2, 4]
(COMMA, ',')	[0, 2, 4]
(NAME, 'j')	[0, 2, 4]
(INT, 6)	[0, 2, 4]
(DEDENT, 2)	**[0]**
(DEDENT, 2)	[0]
(PRINT, 'print')	[0]
(INT, 5)	[0]

```
# t_ignore = ' \t' # can't ignore spaces
indents = [0]
def t_indent(t):
    r'\n[ ]*' # tab is not allowed here
    t.lexer.lineno += 1
    if t.lexer.lexpos >= len(t.lexer.lexdata) \
            or t.lexer.lexdata[t.lexer.lexpos] == "\n": # empty line
        return None             # ignore empty line
    width = len(t.value) - 1 # exclude \n
    last_pos = t.lexer.lexpos - width
    if width == indents[-1]:
        return None # same indentation level
    elif width > indents[-1]: # one more level
        t.type = 'INDENT'
        t.value = 1
        indents.append(width)
        return t
    else: # try one or more DEDENTS
        ded = 0
        while len(indents) > 1:
            indents.pop()
            ded += 1
            if width == indents[-1]:
                t.type = 'DEDENT'
                t.value = ded # remember how many dedents
                return t
        raise Exception("bad indent level at line %d: %s" % (t.lexer.lineno - 1,
                                                           t.lexer.lines[t.lexer.lineno-1]))

def t_space(t):
    r'[ ]+'
    return None # ignore white space
```

```
if_stmt : "if" bool_expr ":" suite
suite : NEWLINE INDENT stmt+ DEDENT
```

Python's INDENT/DEDENT

- this is beyond context-free, and thus way beyond regular!
- use a stack of indentation levels; INDENT/DEDENT is produced whenever indentation level *changes*

```
class MyLexer(object):
    def __init__(self):
        lex.lex() # build regexps
        self.lexer = lex.lexer
        self.dedent_balance = 0

    def input(self, stream):
        # the initial \n is to simplify indent
        # the final \n is to simplify dedent
        stream = "\n" + stream + "\n"
        self.lexer.input(stream)
        self.lexer.lines = stream.split("\n") # internal
        print >> logs, "now lexing..."

    def tokenstr(self, tok):
        return "(%s, %s)" % (tok.type,
                     ("'%s'" if type(tok.value) is str else '%s') % tok.value)

    def token(self):
        if self.dedent_balance != 0:
            self.dedent_balance -= 1
            tok = self.last_tok # (DEDENT, 1)
            print >> logs, self.tokenstr(tok),
        else:
            tok = self.lexer.token() # lexer.token
            if not tok:
                print >> logs
                return None
            print >> logs, self.tokenstr(tok),
            if tok.type == 'DEDENT':
                self.dedent_balance = tok.value - 1
                self.last_tok = tok
        return tok
```

```
for i in range(1):
__for j in range(2):
____print i,j
____6
print 5
```

```
if_stmt : "if" bool_expr ":" suite
suite : NEWLINE INDENT stmt+ DEDENT
```

(FOR, 'for')	[0]
(NAME, 'i')	[0]
(IN, 'in')	[0]
(RANGE, 'range')	[0]
(LPAREN, '(')	[0]
(INT, 1)	[0]
(RPAREN, ')')	[0]
(COLON, ':')	[0]
(INDENT, 1)	**[0, 2]**
(FOR, 'for')	[0, 2]
(NAME, 'j')	[0, 2]
(IN, 'in')	[0, 2]
(RANGE, 'range')	[0, 2]
(LPAREN, '(')	[0, 2]
(INT, 2)	[0, 2]
(RPAREN, ')')	[0, 2]
(COLON, ':')	[0, 2]
(INDENT, 1)	**[0, 2, 4]**
(PRINT, 'print')	[0, 2, 4]
(NAME, 'i')	[0, 2, 4]
(COMMA, ',')	[0, 2, 4]
(NAME, 'j')	[0, 2, 4]
(INT, 6)	[0, 2, 4]
(DEDENT, 2)	**[0]**
(DEDENT, 2)	[0]
(PRINT, 'print')	[0]
(INT, 5)	[0]

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
```

```
compare with (ambiguity):

expr : expr PLUS expr
     | expr TIMES expr
     | NUMBER

---- or ----

expr : expr PLUS expr
     | term
term : term TIMES term
     | 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
         v
NAME = factor
```

- 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

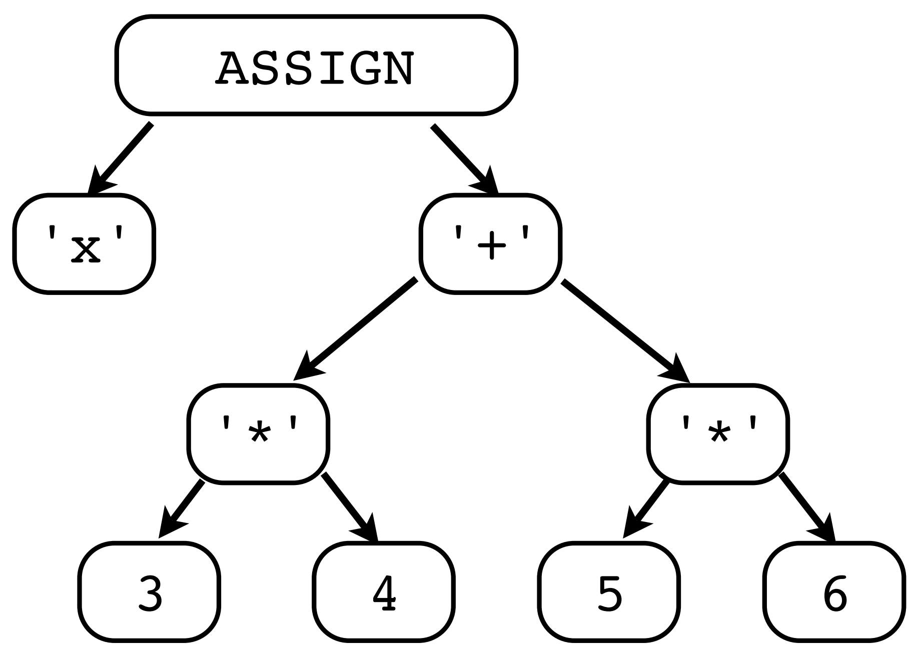
```
def p_assign(p):
    '''assign : NAME EQUALS expr'''
    vars[p[1]] = p[3]

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))
               )
)
>>>
```



Why use PLY?

- There are many Python parsing tools
- Some use more powerful parsing algorithms
- Isn't parsing a "solved" problem anyways?

PLY is Informative

- Compiler writing is hard
- Tools should not make it even harder
- PLY provides extensive diagnostics
- Major emphasis on error reporting
- Provides the same information as yacc

PLY Diagnostics

- PLY produces the same diagnostics as yacc
- Yacc

```
% yacc grammar.y
4 shift/reduce conflicts
2 reduce/reduce conflicts
```

- PLY

```
% python mycompiler.py
yacc: Generating LALR parsing table...
4 shift/reduce conflicts
2 reduce/reduce conflicts
```

- PLY also produces the same debugging output

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
```

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 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-
t_MINUS  = r'-'
t_POWER  = r'\^'

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

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

```
example.py:12: Rule t_MINUS redefined.
               Previously defined on line 6
```

Error 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_]*’
t_MINUS  = r'-'
t_POWER  = r'\^'

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

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

```
lex: Rule 't_POWER' defined for an
unspecified token POWER
```

Error 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_]*'
t_MINUS  = r'-'
t_POWER  = r'\^'

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

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

```
example.py:15: Rule 't_NUMBER' requires
an argument.
```

PLY is Yacc

- PLY supports all of the major features of Unix lex/yacc
- Syntax error handling and synchronization
- Precedence specifiers
- Character literals
- Start conditions
- Inherited attributes

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]
```

Character Literals

- Yacc

```
expr : expr '+' expr { $$ = $1 + $3; }
     | expr '-' expr { $$ = $1 - $3; }
     | expr '*' expr { $$ = $1 * $3; }
     | expr '/' expr { $$ = $1 / $3; }
     ;
```

- PLY

```
def p_expr(p):
    '''expr : expr '+' expr
            | expr '-' expr
            | expr '*' expr
            | expr '/' expr'''
    ...
```

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"
```

PLY is Simple

- Two pure-Python modules. That's it.
- Not part of a "parser framework"
- Use doesn't involve exotic design patterns
- Doesn't rely upon C extension modules
- Doesn't rely on third party tools

PLY is Fast

- For a parser written entirely in Python
- Underlying parser is table driven
- Parsing tables are saved and only regenerated if the grammar changes
- Considerable work went into optimization from the start (developed on 200Mhz PC)

PLY Performance

- Parse file with 1000 random expressions (805KB) and build an abstract syntax tree
    - PLY-2.3 : 2.95 sec, 10.2 MB (Python)
    - DParser : 0.71 sec, 72 MB (Python/C)
    - BisonGen : 0.25 sec, 13 MB (Python/C)
    - Bison : 0.063 sec, 7.9 MB (C)
- 12x slower than BisonGen (mostly C)
- 47x slower than pure C
- System: MacPro 2.66Ghz Xeon, Python-2.5

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)
```

Limitations

- LALR(1) parsing
- Not easy to work with very complex grammars (e.g., C++ parsing)
- Retains all of yacc's black magic
- Not as powerful as more general parsing algorithms (ANTLR, SPARK, etc.)
- Tradeoff : Speed vs. Generality