How to Create a Programming Language

LECTURE 13: PARSING TOOLS

1

LECTURE OUTLINE

- THEORETICAL BACKGROUND
  - REGULAR EXPRESSIONS
  - CHOMSKY HIERARCHY
  - CONTEXT FREE GRAMMARS
- SCANNING
- PARSING
  - TOP-DOWN
  - BOTTOM-UP
  - LL AND LR PARSING
- **CLASSIC PARSING TOOLS**
  - **BISON**
  - **LEX**
  - **PLY**
  - **TPG**

2

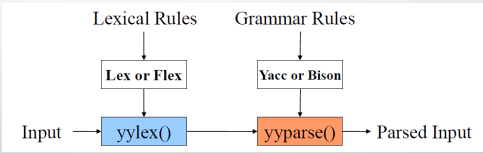
PROGRAMMING TOOLS

- *COMPILER-COMPILER (PARSER GENERATOR, COMPILER GENERATOR)*
  - PROGRAMMING TOOL
  - CREATES A PARSER, INTERPRETER, OR COMPILER
  - USES A FORMAL DESCRIPTION OF A LANGUAGE AND MACHINE
  - INPUT IS A GRAMMAR (USUALLY IN BNF) FOR A PROGRAMMING LANGUAGE
  - GENERATED OUTPUT IS THE SOURCE CODE OF A PARSER
- EXAMPLES OF PARSER GENERATORS:
  - CLASSICAL PARSING TOOLS: LEX, YACC, BISON AND FLEX, ANTLR
  - PLY: PYTHON IMPLEMENTATION OF LEX AND YACC
  - PYTHON TPG PARSER
  - ANTLR FOR PYTHON

3

CLASSIC PARSING TOOLS

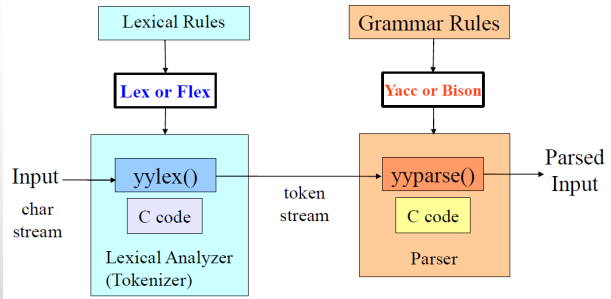
- **LEX:** ORIGINAL UNIX LEXICAL ANALYSIS GENERATOR
  - CREATE A C FUNCTION THAT WILL SCAN INPUT ACCORDING TO A SET OF REGULAR EXPRESSIONS
- **YACC:** YET ANOTHER COMPILER COMPILER
  - GENERATE A C PROGRAM FOR A PARSER FROM BNF RULES
- **BISON** AND **FLEX** ("FAST LEX")
  - MORE POWERFUL, FREE VERSIONS OF YACC AND LEX
  - FROM GNU SOFTWARE FOUNDATION



4

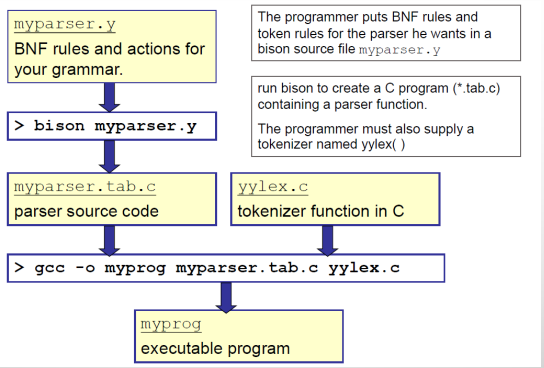
CLASSIC PARSING TOOLS

- LEX AND YACC GENERATE C CODE FOR YOUR ANALYZER & PARSER



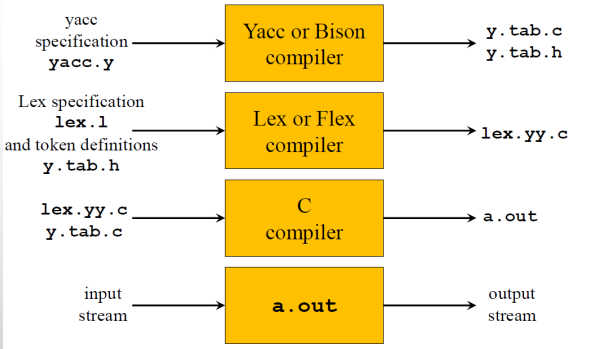
5

BISON OVERVIEW



6

COMBINING LEX/FLEX WITH YACC/BISON



7

LEX SPECIFICATION EXAMPLE

```
%option noyywrap
%{
#include "y.tab.h"
extern double yylval;
%}
number  [0-9]+\.?|[0-9]*\.[0-9]+
%%
[ ]             { /* skip blanks */ }
{number}        { sscanf(yytext, "%lf", &yylval);
                  return NUMBER;
                }
\n|.            { return yytext[0]; }
```

Generated by Yacc, contains `#define NUMBER ...`

Defined in `y.tab.c`

8

BISON SPECIFICATION EXAMPLE

- A GRAMMAR WRITTEN IN BNF.
  - BISON CREATES A C PROGRAM THAT PARSES ACCORDING TO THE RULES

```
term    : term '*' factor { $$ = $1 * $3; }
        | term '/' factor { $$ = $1 / $3; }
        | factor          { $$ = $1; }
        ;
factor  : ID              { $$ = valueof($1); }
        | NUMBER          { $$ = $1; }
        ;
```

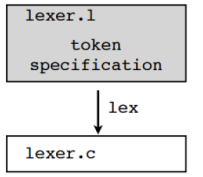
```
yacc -d example2.y
lex example2.l
gcc y.tab.c lex.yy.c
./a.out
```

```
bison -d -y example2.y
flex example2.l
gcc y.tab.c lex.yy.c
./a.out
```


9

LEX EXAMPLE

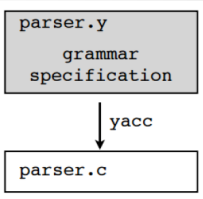
```
/* lexer.l */
%{
#include "header.h"
int lineno= 1;
%}
%%
[ \t]* ;    /* Ignore whitespace */
\n          { lineno++; }
[0-9]+      { yylval.val = atoi(yytext);
              return NUMBER; }
[a-zA-Z_][a-zA-Z0-9_]*   { yylval.name = strdup(yytext);
                           return ID; }
\+   { return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



10

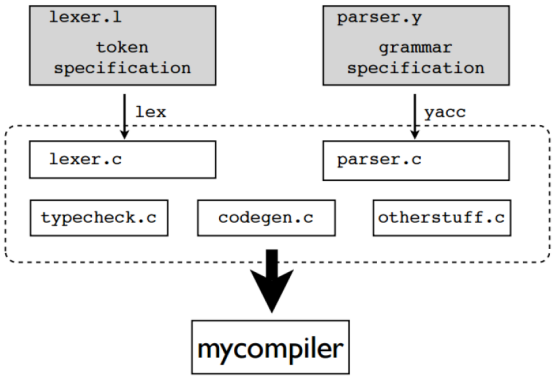
YACC EXAMPLE

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



11

LEX & YACC: THE BIG PICTURE



12

PLY SPECIFICATION EXAMPLE

- A BIT OF HISTORY:
  - YACC: ~1973. STEPHEN JOHNSON (AT&T)
  - LEX : ~1974. ERIC SCHMIDT AND MIKE LESK(AT&T)
  - PLY: 2001
  - PLY: PYTHON LEX-YACC
    - IMPLEMENTATION OF LEX AND YACC FOR PYTHON BY DAVID BEAZLEY:
    - HTTP://WWW.DABEAZ.COM/PLY/

13

PLY SPECIFICATION EXAMPLE

- PLY IS NOT A CODE GENERATOR
  - PLY CONSISTS OF TWO PYTHON MODULES
    - PLY.LEX - A MODULE FOR WRITING LEXERS
      - TOKENS SPECIFIED USING REGULAR EXPRESSIONS
      - PROVIDES FUNCTIONS FOR READING INPUT TEXT
    - PLY.YACC - A MODULE FOR WRITING GRAMMARS
  - YOU SIMPLY IMPORT THE MODULES TO USE THEM
    - THE GRAMMAR MUST BE IN A FILE

14

PLY SPECIFICATION EXAMPLE

ply.lex example:

```
import ply.lex as lex
tokens = [ 'NAME','NUMBER','PLUS','MINUS','TIMES',
 'DIVIDE', EQUALS' ]
t_ignore = ' \t'
t_PLUS = r'\+'
t_MINUS = r'-'
t_TIMES = r'\*'
t_DIVIDE = r'/'
t_EQUALS = r'='
t_NAME = r'[a-zA-Z_][a-zA-Z0-9_]*'

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

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

tokens list specifies all of the possible tokens

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

Functions are used when special action code must execute

Builds the lexer by creating a master regular expression

15

PLY SPECIFICATION EXAMPLE

Two functions: input() and token()

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

    # Use token
```

input() feeds a string into the lexer

token() returns the next token or None

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

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

matching text

Position in input text

16

PLY SPECIFICATION EXAMPLE

```
import ply.yacc as yacc
import mylexer                  # Import lexer information
tokens = mylexer.tokens         # Need token list

def p_assign(p):
     '''assign : NAME EQUALS expr'''
def p_expr(p):
     '''expr : expr PLUS term
             | expr MINUS term
             | term'''
def p_term(p):
     '''term : term TIMES factor
             | term DIVIDE factor
             | factor'''
def p_factor(p):
     '''factor : NUMBER'''

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

token information imported from lexer

grammar rules encoded as functions with names p_*rulename*

docstrings contain grammar rules from BNF

17

PLY SPECIFICATION EXAMPLE

- PLY USES LR-PARSING LALR(1)
  - SHIFT-REDUCE PARSING
  - TABLE DRIVEN
  - INPUT TOKENS ARE SHIFTED ONTO A PARSING STACK

    X = 3 * 4 + 5 ->

    = 3 * 4 + 5 -> NAME

    3 * 4 + 5 -> NAME =

    * 4 + 5 -> NAME = NUM
  - THIS CONTINUES UNTIL A COMPLETE GRAMMAR RULE APPEARS ON THE TOP OF THE STACK

    reduce factor : NUM

    * 4 + 5 -> NAME = factor

18

PLY SPECIFICATION EXAMPLE

- DURING REDUCTION, RULE FUNCTIONS ARE INVOKED

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

- PARAMETER P CONTAINS GRAMMAR SYMBOL VALUES

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


19

PLY SPECIFICATION EXAMPLE

- RULE FUNCTIONS GENERALLY PROCESS VALUES ON RIGHT HAND SIDE OF GRAMMAR RULE
- RESULT IS THEN STORED IN LEFT HAND SIDE
- RESULTS PROPAGATE UP THROUGH THE GRAMMAR
- PLY DOES BOTTOM-UP PARSING

20

PLY CALCULATOR EXAMPLE

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

def p_expr_plus(p):
    '''expr : expr PLUS term'''
    p[0] = p[1] + p[3]

def p_term_mul(p):
    '''term : term TIMES factor'''
    p[0] = p[1] * p[3]

def p_term_factor(p):
    '''term : factor'''
    p[0] = p[1]

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


21

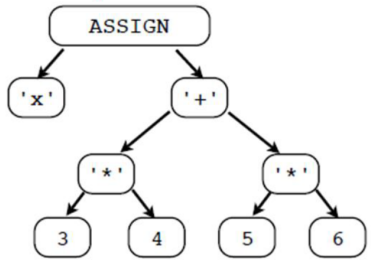
PLY CALCULATOR EXAMPLE

```
def p_assign(p):
    '''assign : NAME EQUALS expr'''
    p[0] = ('ASSIGN',p[1],p[3])

def p_expr_plus(p):
    '''expr : expr PLUS term'''
    p[0] = ('+',p[1],p[3])

def p_term_mul(p):
    '''term : term TIMES factor'''
    p[0] = ('*',p[1],p[3])

def p_term_factor(p):
    '''term : factor'''
    p[0] = p[1]

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


22

PLY CALCULATOR EXAMPLE

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



23

PLY PRECEDENCE SPECIFIERS

- PRECEDENCE SPECIFIERS
  - HIGHEST PRECEDENCE AT BOTTOM:

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


24

PLY BEST DOCUMENTATION

Google Mailing list/group:
http://groups.google.com/group/ply-hack

25

TPG EXAMPLE

- TPG IS A LEXICAL AND SYNTACTIC PARSER GENERATOR FOR PYTHON.
  - YACC TOO COMPLEX FOR SIMPLE CASES
    - CALCULATORS, CONFIGURATION FILES, SMALL PROGRAMMING LANGUAGES, …
  - CAN ADD PYTHON CODE DIRECTLY INTO GRAMMAR RULES AND BUILD ABSTRACT SYNTAX TREES WHILE PARSING

26

PYTHON TPG LEXER

Toy Parser Generator (TPG): http://cdsoft.fr/tpg
Syntax:
token <name> <regex> <function> ;
separator <name> <regex>;
Example:

```
token integer '\d+' int;
token float '\d+\.\d*|\.\d+' float;
token rbrace'{';
separator space '\s+';
```


27

PYTHON TPG LEXER

Embed TPG in Python:

```
import tpg
Class Calc:
  r"""
  separator spaces: '\s+' ;
  token number: '\d+' ;
  token add: '[+-]' ;
  token mul: '[*/]' ;
  """
```

Try it in Python:
download TPG from http://cdsoft.fr/tpg

28

TPG EXAMPLE

Defining the grammar:

Terminal symbols for expressions:

number	[0 -9]+ or \d+	One or more digits
add	[+-]	a + or a -
mul	[*/]	a * or a /
spaces	\s+	One or more spaces

Non-terminal productions:

```
START -> Expr;
Expr-> Term ( add Term )* ;
Term -> Fact ( mulFact )* ;
Fact -> number | '\(' Expr '\)' ;
```


29

TPG EXAMPLE

```
import tpg
    class Calc:
        r"""
        separator spaces: '\s+' ;
        token number: '\d+' ;
        token add: '[+-]' ;
        token mul: '[*/]' ;

        START -> Expr ;
        Expr -> Term ( add Term )* ;
        Term -> Fact ( mulFact )* ;
        Fact -> number | '\(' Expr '\)' ;
        """
```


30

TPG EXAMPLE

Reading the input and returning values:

```
separator spaces: '\s+' ;
token number: '\d+' int;
token add: '[+-]' make_op;
token mul: '[*/]' make_op;
```

Transform tokens into defined operations:

```
def make_op(s):
    return{
    '+': lambda x,y: x+y,
    '-': lambda x,y: x-y,
    '*': lambda x,y: x*y,
    '/': lambda x,y: x/y,
    }[s]
```


31

TPG EXAMPLE

After terminal is recognized, will store it in a Python variable

Example: save a number in a variable n: *number/n*.

Include Python code example:

```
Expr/t -> Term/t ( add/ opTerm/f  $t=op(t,f)$)* ;
Term/f -> Fact/f ( mul/ op Fact/a $f=op(f,a)$ )* ;
Fact/a -> number/a | '\(' Expr/a '\)' ;
```


32

TPG EXAMPLE

```
import math                     # Simple calculator calc.py
import operator
import string
import tpg
def make_op(s):
        return {
                '+': lambda x,y: x+y,
                '-': lambda x,y: x-y,
                '*': lambda x,y: x*y,
                '/': lambda x,y: x/y,
        }[s]
class Calc(tpg.Parser):
        r"""
        separator spaces: '\s+' ;
        token number: '\d+' int ;
        token add: '[+-]' make_op ;
        token mul: '[*/]' make_op ;
        START/e -> Term/e ;
```


33

TPG EXAMPLE

```
        Term/t -> Fact/t ( add/op Fact/f $t=op(t,f)$ )* ;
        Fact/f -> Atom/f ( mul/op Atom/a $f=op(f,a)$ )* ;
        Atom/a -> number/a | '\(' Term/a '\)' ;
        """
calc = Calc()

if tpg.__python__ == 3:
    operator.div = operator.truediv
    raw_input = input

expr = raw_input('Enter an expression: ')
print(expr, '=', calc(expr))
```


34

TPG EXAMPLE

```
#!/usr/bin/env python
# Larger example: scientific_calc.py
import math
import operator
import string
import tpg
if tpg.__python__ == 3:
    operator.div = operator.truediv
    raw_input = input
def make_op(op):
    return {
        '+'   : operator.add,
        '-'   : operator.sub,
        '*'   : operator.mul,
        '/'   : operator.div,
        '%'   : operator.mod,
        '^'   : lambda x,y:x**y,
        '**'  : lambda x,y:x**y,
        'cos' : math.cos,
        'sin' : math.sin,
        'tan' : math.tan,
        'acos': math.acos,
```


35

TPG EXAMPLE

```
        'asin': math.asin,
        'atan': math.atan,
        'sqr' : lambda x:x*x,
        'sqrt': math.sqrt,
        'abs' : abs,
        'norm': lambda x,y:math.sqrt(x*x+y*y),
    }[op]
class Calc(tpg.Parser, dict):
    r"""
        separator space '\s+' ;
        token pow_op    '\^|\*\*' $ make_op
        token add_op    '[+-]'    $ make_op
        token mul_op    '[*/%]'   $ make_op
        token funct1    '(cos|sin|tan|acos|asin|atan|sqr|sqrt|abs)\b' $ make_op
        token funct2    '(norm)\b' $ make_op
        token real      '(\d+\.\d*|\d*\.\d+)([eE][-+]?\d+)?|\d+[eE][-+]?\d+'
                $ float
        token integer   '\d+' $ int
        token VarId     '[a-zA-Z_]\w*'
        ;
```


36

TPG EXAMPLE

```
START/e ->
        'vars'                  $ e=self.mem()
    |   VarId/v '=' Expr/e      $ self[v]=e
    |   Expr/e
;
Var/$self.get(v,0)$ -> VarId/v ;
Expr/e -> Term/e ( add_op/op Term/t      $ e=op(e,t)
                 )*
;
Term/t -> Fact/t ( mul_op/op Fact/f      $ t=op(t,f)
                 )*
;
Fact/f ->
        add_op/op Fact/f                 $ f=op(0,f)
    |   Pow/f
;
Pow/f -> Atom/f ( pow_op/op Fact/e       $ f=op(f,e)
                )?
;
```


37

TPG EXAMPLE

```
    Atom/a ->
            real/a
        |   integer/a
        |   Function/a
        |   Var/a
        |   '\(' Expr/a '\)'
    ;
    Function/y ->
            funct1/f '\(' Expr/x '\)'                $ y = f(x)
        |   funct2/f '\(' Expr/x1 ',' Expr/x2 '\)'   $ y = f(x1,x2)
    ;
"""
def mem(self):
    vars = sorted(self.items())
    memory = [ "%s = %s"%(var, val) for (var, val) in vars ]
    return "\n\t" + "\n\t".join(memory)
```


38

TPG EXAMPLE

```
print("Calc (TPG example)")
calc = Calc()
while 1:
    l = raw_input("\n:")
    if l:
        try:
            print(calc(l))
        except Exception:
            print(tpg.exc())
    else:
        break
```


39

QUESTIONS

40