



# COMPILER CONSTRUCTION

## Yacc Yet Another Compiler-Compiler

Chia-Heng Tu

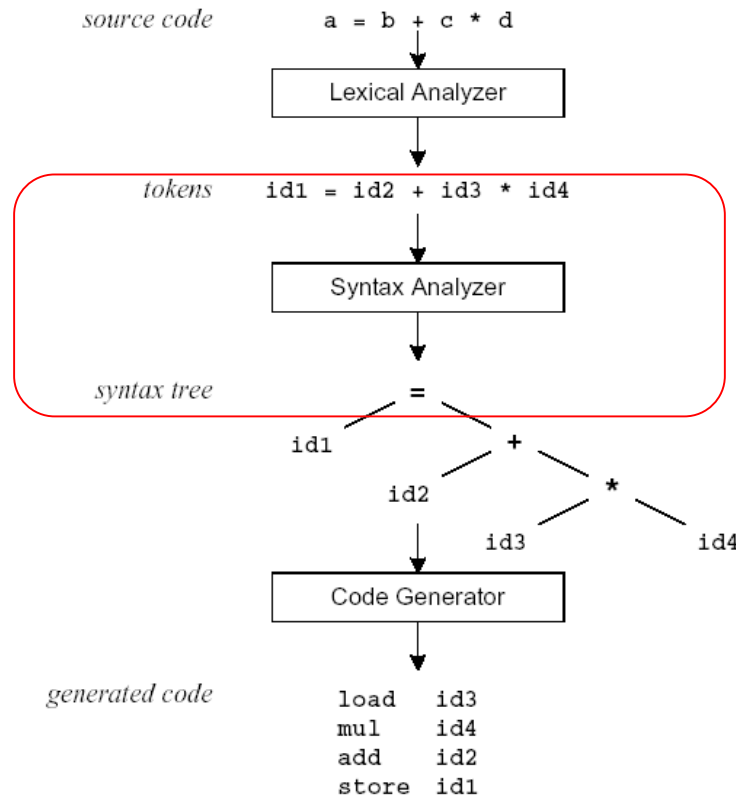
Dept. of Computer Science and Information  
Engineering

National Cheng Kung University  
Spring 2019



# Where are we?

- Lex and Yacc are able to do the following
- Now, our target is Yacc



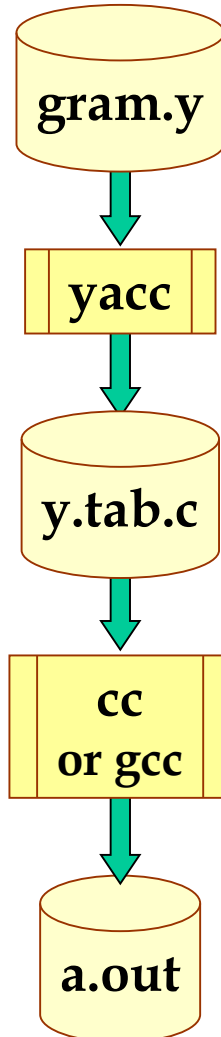


# Introduction

- What is **YACC** ?
  - Tool which will produce a parser for a given grammar
- YACC (Yet Another Compiler Compiler) is a program designed
  - to compile a LALR(1) grammar and
  - to produce the source code of the syntactic analyzer of the language produced by this grammar



# How Yacc Works?



File containing desired grammar in yacc format

*yacc program (executable)*

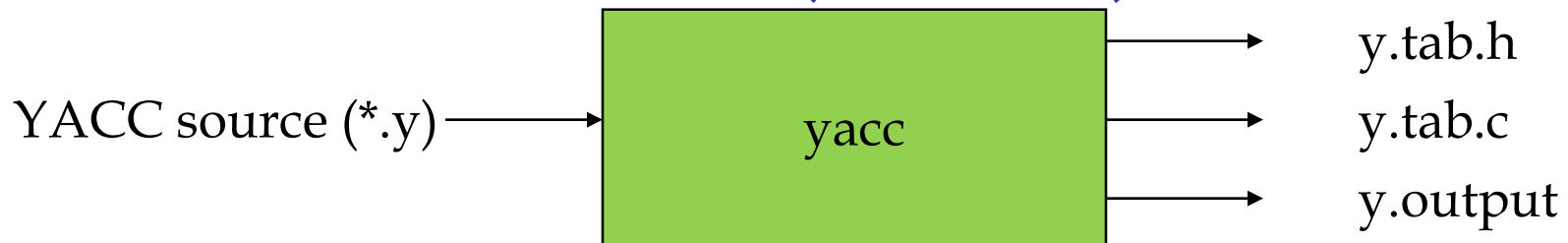
C source program created by yacc

*C compiler (executable)*

Executable program that will parse grammar given in gram.y



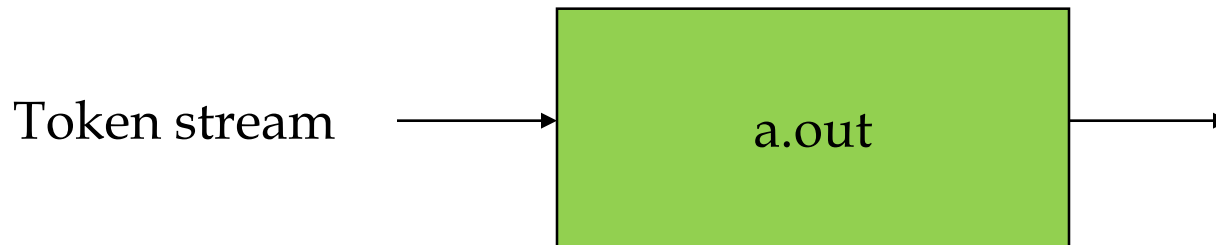
# How Yacc Works? (Cont'd)



(1) Parser generation time



(2) Compile time



(3) Run time

Abstract Syntax  
 Tree  
 (We dump  
 messages in the  
*actions* of the  
 matched rules)



# Yacc and Lex

**LEX**  
yylex()

**YACC**  
yyparse()

a.out



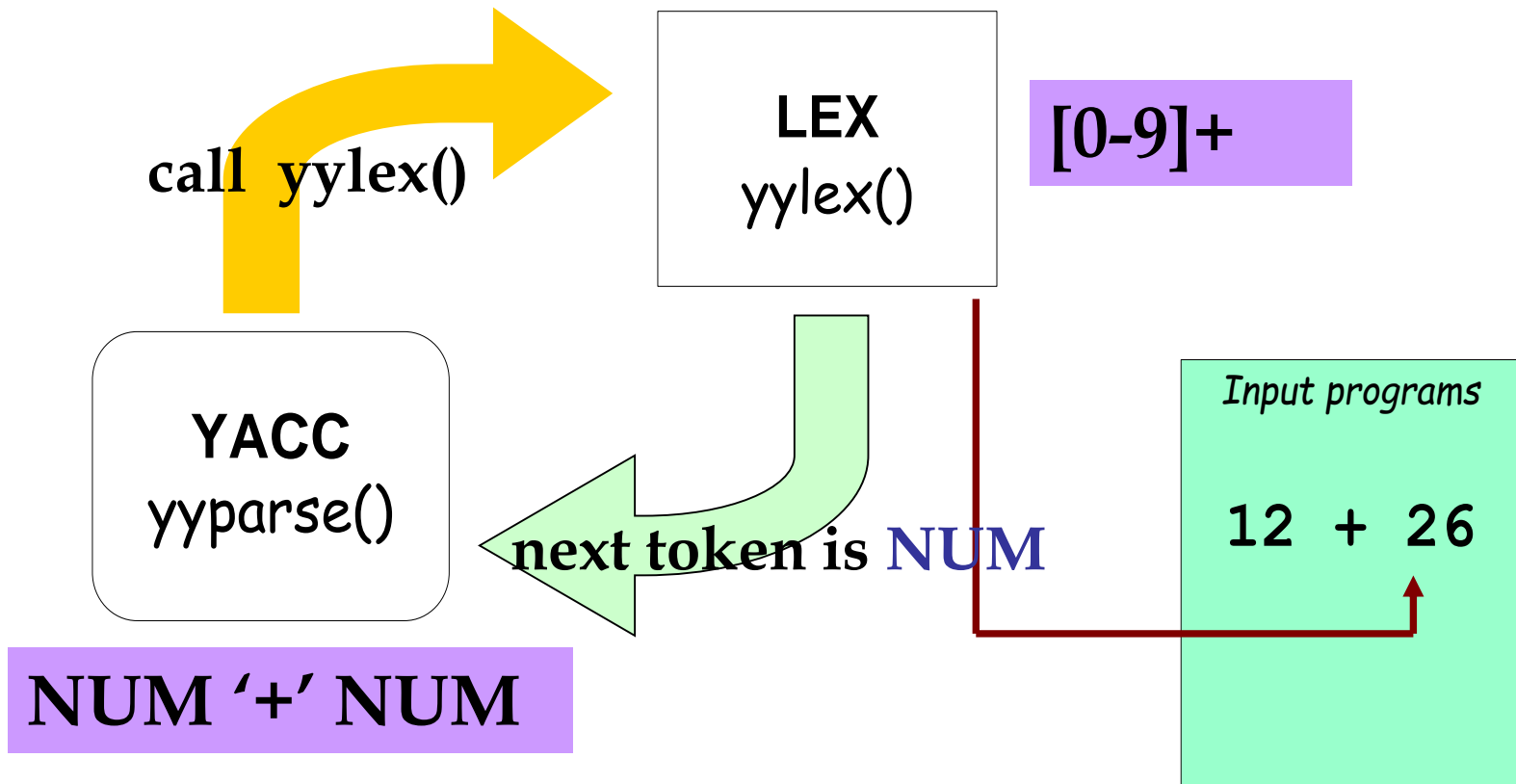
What's going on?

*Input programs*

**12 + 26**



# Yacc and Lex (Control Flow)





# An Yacc File Example

- Similar to Lex, Yacc program could be divided into three parts

```
%{
#include <stdio.h>
%}
```

C code

```
%token NAME NUMBER
%%

statement: NAME '=' expression
          | expression           { printf("= %d\n", $1); }
          ;

expression: expression '+' NUMBER { $$ = $1 + $3; }
           | expression '-' NUMBER { $$ = $1 - $3; }
           | NUMBER               { $$ = $1; }
           ;

%%
```

Grammar rules  
and actions

```
int yyerror(char *s)
{
    fprintf(stderr, "%s\n", s);
    return 0;
}

int main(void)
{
    vyparse();
    return 0;
}
```

C routines





# Yacc File Format

%{

*C declarations*

%}

*yacc declarations*

% %

*Grammar rules*

% %

*Additional C code*

**Comments enclosed in `/* ... */` may appear in any of the sections.**



# Declarations

```
% {
```

```
#include <stdio.h>
```

```
#include <stdlib.h>
```

```
% }
```

```
% token ID NUM
```

```
% start expr
```

It is a terminal

由 expr 開始parse



# Start Symbol

- The first non-terminal specified in the *grammar specification section*

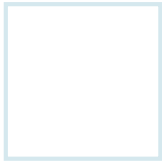
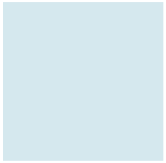
- To overwrite it with `%start` declaration

`%start non-terminal`



# Grammar Rules Section

- This section defines grammar
- Example
  - $\text{expr} : \text{expr} '+' \text{term} \mid \text{term};$
  - $\text{term} : \text{term} '*' \text{factor} \mid \text{factor};$
  - $\text{factor} : '(' \text{expr} ') ' \mid \text{ID} \mid \text{NUM};$



# Grammar Rules Section (Cont'd)

- Typically, the yacc's rules in the `.y` file look like below
- Example

```
expr    : expr '+' term
        | term
        ;

term     : term '*' factor
        | factor
        ;

factor  : '(' expr ')'
        | ID
        | NUM
        ;
```





# The Position of Grammar Rules (1/4)

```

expr  : expr '+' term      { $$ = $1 + $3; }
        | term                { $$ = $1; }
        ;

term   : term '*' factor     { $$ = $1 * $3; }
        | factor              { $$ = $1; }
        ;

factor : '(' expr ')'        { $$ = $2; }
        | ID
        | NUM
        ;
    
```



## The Position of Grammar Rules (2/4)

**expr** : **expr** '+' **term** { \$\$ = \$1 + \$3; }  
| **term** { \$\$ = \$1; }  
;  
**term** : **term** '\*' **factor** { \$\$ = \$1 \* \$3; }  
| **factor** { \$\$ = \$1; }  
;  
**factor** : '(' **expr** ')' { \$\$ = \$2; }  
| ID  
| NUM  
;



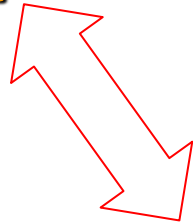
# The Position of Grammar Rules (3/4)

```

expr : expr '+' term      { $$ = $1 + $3; }
      | term              { $$ = $1; }
      ;

term : term '*' factor    { $$ = $1 * $3; }
      | factor            { $$ = $1; }
      ;

factor : '(' expr ')'     { $$ = $2; }
        | ID
        | NUM
        ;
    
```


  
\$2





# The Position of Grammar Rules (4/4)

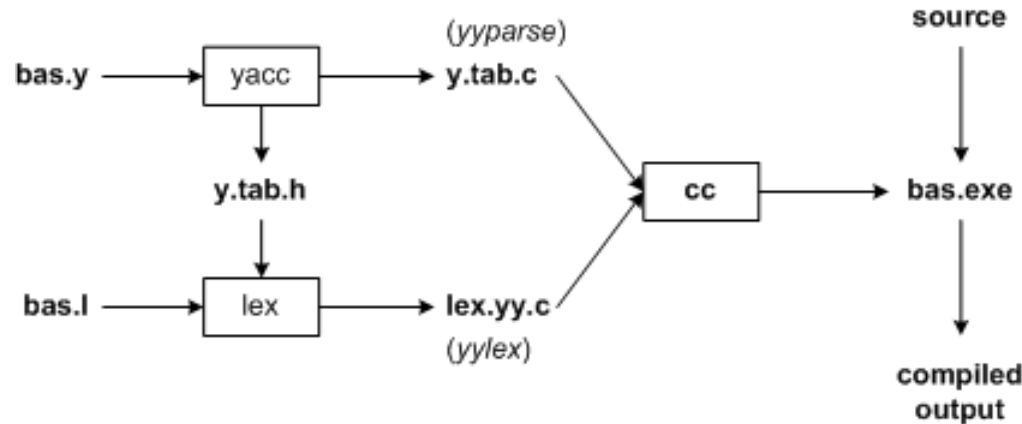
```
expr : expr '+' term      { $$ = $1 + $3; }  
      | term                { $$ = $1; }  
      ;  
  
term : term '*' factor    { $$ = $1 * $3; }  
      | factor              { $$ = $1; }  
      ;  
  
factor : '(' expr ')'     { $$ = $2; }  
        | ID  
        | NUM  
        ;
```

← Default:  $$$ = \$1$ ;

**\$3**



# More about the Lex & Yacc Files

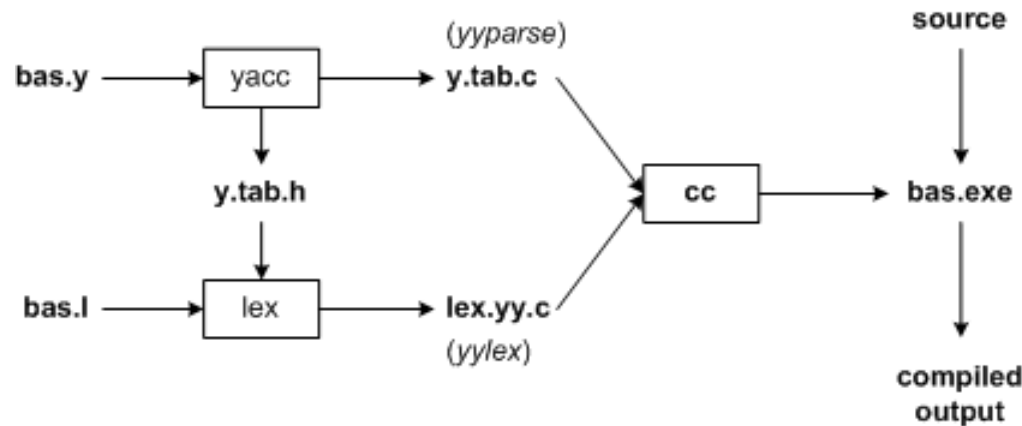


- The figure illustrates the file naming conventions used by lex and yacc
- We need to specify all pattern matching rules for Lex (**bas.l**) and grammar rules for Yacc (**bas.y**)
- Commands to create the compiler, **bas.exe**, are listed below:
 

<b>yacc -d bas.y</b>	# create y.tab.h, y.tab.c
<b>lex bas.l</b>	# create lex.yy.c
<b>cc lex.yy.c y.tab.c -o bas.exe</b>	# compile/link



# More about the Lex & Yacc Files (Cont'd)



- Yacc reads the grammar descriptions in **bas.y** and generates a syntax analyzer (parser)
  - that includes function `yyparse`, in file **y.tab.c**
  - Included in file **bas.y** are token declarations
  - The `-d` option asks yacc to generate definitions for tokens and place them in file **y.tab.h**
- Lex reads the pattern descriptions in **bas.l**, includes file **y.tab.h**, and
  - generates a lexical analyzer, function `yylex`, in file **lex.yy.c**
- Finally, the lexer and parser are compiled and linked together to create executable **bas.exe**
  - From **main**, we call `yyparse` to run the compiler
  - Function `yyparse` automatically calls `yylex` to obtain each token



# Data Sharing between Lex and Yacc

```
%{
#include <stdio.h>
#include "y.tab.h"
}%
id    [_a-zA-Z][_a-zA-Z0-9]*
%%
int    { return INT; }
char   { return CHAR; }
float  { return FLOAT; }
{id}   { return ID; }
```

scanner.l

yacc -d xxx.y  
Produced  
y.tab.h:

```
# define CHAR 258
# define FLOAT 259
# define ID 260
# define INT 261
```

```
%{
#include <stdio.h>
#include <stdlib.h>
}%
%token CHAR, FLOAT, ID, INT
%%
```

parser.y



# Internals of Yacc

- Rules may be recursive
  - Rules may be ambiguous
  - Uses bottom-up parsing
    - Also known as Shift/Reduce parsing
    - Get a token
    - Push onto stack
    - Can it reduced (How do we know?)
      - If yes: Reduce using a rule
      - If no: Get another token
  - Yacc cannot look ahead more than one token
- ← No problem
- ← You have learnt how to avoid ambiguous grammar.
- ← Use **printf** wisely



# Internals of Yacc (Cont'd)

- **shift/reduce conflict**
  - occurs when a grammar is written in such a way that a decision between shifting and reducing can not be made
  - E.g., IF-ELSE ambiguous ( $\Rightarrow$  keep a *short* rule.)
- To resolve this conflict, yacc will choose to shift
- In order to take control of the parsing procedure
  - You could rewrite the grammar to avoid the conflict



# Put It All Together

## Parser

```

expr : expr '+' term    { $$ = $1 + $3; }
      | term             { $$ = $1; }
      ;
term  : term '*' factor  { $$ = $1 * $3; }
      | factor           { $$ = $1; }
      ;
factor : '(' expr ')'    { $$ = $2; }
       | ID
       | NUM
       ;

```

← Default:  $$$ = \$1$ ;

An expression:

$a = 4 + 6$

//  $a=10$

## Scanner

```

%{
#include "y.tab.h"
#include "parser.h"
#include <math.h>
%}
%%
([0-9]+|([0-9]*\.[0-9]+)([eE][+-]?[0-9]+)?) {
    yylval.dval = atof(yytext);
    return NUM;
}

[ \t] ; /* ignore white space */

```



# Yacc Declarations

**`%start'**

Specify the grammar's start symbol

**`%union'**

Declare the collection of data types that semantic values may have

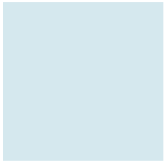
**`%token'**

Declare a terminal symbol (token type name) with no precedence or associativity specified

**`%type'**

Declare the type of semantic values for a nonterminal symbol  
Using the declared names from the %union





# Yacc Declarations (Cont'd)

**``%right'`**

Declare a terminal symbol (token type name) that is right-associative

**``%left'`**

Declare a terminal symbol (token type name) that is left-associative

**``%nonassoc'`**

Declare a terminal symbol (token type name) that is nonassociative

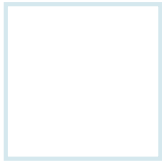
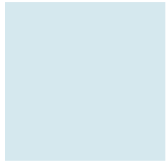
I.e., using it in a way that would be associative is a syntax error, e.g.,  $x \text{ op. } y \text{ op. } z$  is syntax error



# References

- Please refer to the [online manual for Yacc](#) on [The Lex & Yacc Page](#)
- lex & yacc, 2nd Edition
  - by John R. Levine, Tony Mason & Doug Brown
  - O'Reilly
  - ISBN: 1-56592-000-7





# QUESTIONS?