



中山大學  
SUN YAT-SEN UNIVERSITY



国家超级计算广州中心  
NATIONAL SUPERCOMPUTER CENTER IN GUANGZHOU

# Compiler Design 编译器构造实验

---

## Lab 4: YACC

张献伟

[xianweiz.github.io](https://xianweiz.github.io)

DCS292, 3/25/2021



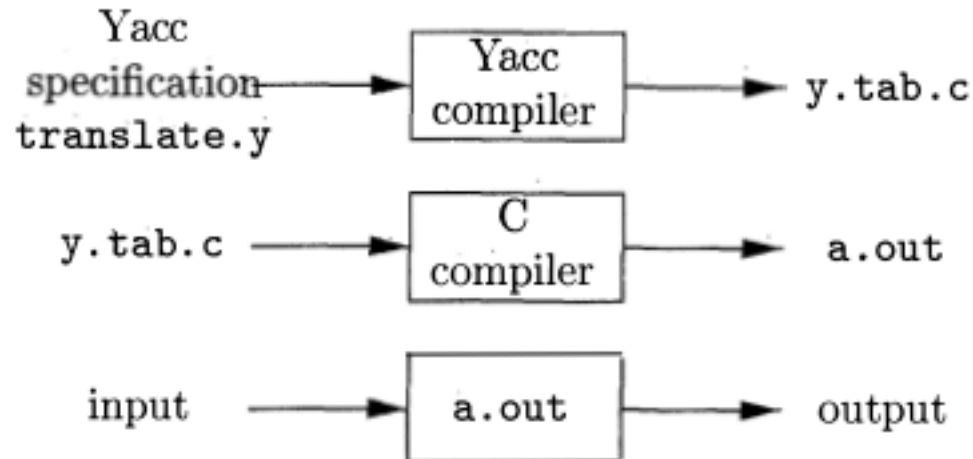
中山大學  
SUN YAT-SEN UNIVERSITY



# Yacc Overview

---

- Yacc is an LALR(1) parser generator
  - YACC: Yet Another Compiler-Compiler
  - Parse a language described by a context-free grammar (**CFG**)
  - Yacc constructs an **LALR(1)** table
- Available as a command on the UNIX system
  - Bison: free GNU project alternative to Yacc



# Yacc Specification

---

- **Definitions** section:
  - C declarations within %{ %}
  - Token declarations
- **Rules** section:
  - Each rule consists of a grammar production and the associated semantic action
- **Subroutines** section:
  - User-defined auxiliary functions

```
%{  
  #include ...  
%}  
%token NUM VAR  
%%  
production { semantic action }  
...  
%%  
...
```

# Write a Grammar in Yacc

---

- A set of productions  $\langle \text{head} \rangle \rightarrow \langle \text{body} \rangle_1 \mid \dots \mid \langle \text{body} \rangle_n$  would be written in YACC as:

```
 $\langle \text{head} \rangle : \langle \text{body} \rangle_1 \{ \langle \text{semantic action} \rangle_1 \}$   
...  
:  $\langle \text{body} \rangle_n \{ \langle \text{semantic action} \rangle_n \}$   
;
```

- Usages

- Tokens that are single characters can be used directly within productions, e.g. `'+'`
- Named tokens must be declared first in the declaration part using `%token TokenName`

# Write a Grammar in Yacc (cont.)

---

- Semantic actions may refer to values of the synthesized attributes of terminals and non-terminals in a production:

$X : Y_1 Y_2 Y_3 \dots Y_n \{ \text{action} \}$

- $\$ \$$  refers to the value of the attribute of  $X$  (non-terminal)
- $\$ i$  refers to the value of the attribute of  $Y_i$  (terminal or non-terminal)

- Example:  $E \rightarrow E + T \mid T$

$\text{expr} : \text{expr} '+' \text{term} \{ \$ \$ = \$1 + \$2 \}$

$\mid \text{term}$

;

Default action:  $\{ \$ \$ = \$1 \}$

# Example: $E \rightarrow E+E | E-E | E * E | E/E | (E) | \text{num}$

```
1 %{
2  #include <ctype.h>
3  #include <stdio.h>
4  #define YYSTYPE double /* double type for Yacc stack */
5  %}
6  %token NUMBER
7
8  %left '+' '-'
9  %left '*' '/'
10
11 %%
12
13 lines : lines expr '\n' { printf("= %g\n", $2); }
14       | lines '\n'
15       | /* empty */
16       ;
17 expr  : expr '+' expr { $$ = $1 + $3; }
18       | expr '-' expr { $$ = $1 - $3; }
19       | expr '*' expr { $$ = $1 * $3; }
20       | expr '/' expr { $$ = $1 / $3; }
21       | '(' expr ')' { $$ = $2; }
22       | NUMBER
23       ;
```

Can we remove those two lines?

Allow to evaluate a sequence of expressions, one to a line

# Example (cont.)

```
24
25 %%%
26
27 int yylex() {
28     int c;
29     while ((c = getchar()) == ' ');
30     if ((c == '.') || isdigit(c)) {
31         ungetc(c, stdin);
32         scanf("%lf", &yylval);
33         return NUMBER;
34     }
35     return c;
36 }
37
38 int main() {
39     if (yyparse() != 0)
40         fprintf(stderr, "Abnormal exit\n");
41     return 0;
42 }
43
44 int yyerror(char *s) {
45     fprintf(stderr, "Error: %s\n", s);
46 }
```

calls yylex() to get successive tokens

# Compile and Run ...

---

- Compile

- `$yacc -d parser.y`
- `$gcc -o test y.tab.c`

- Run

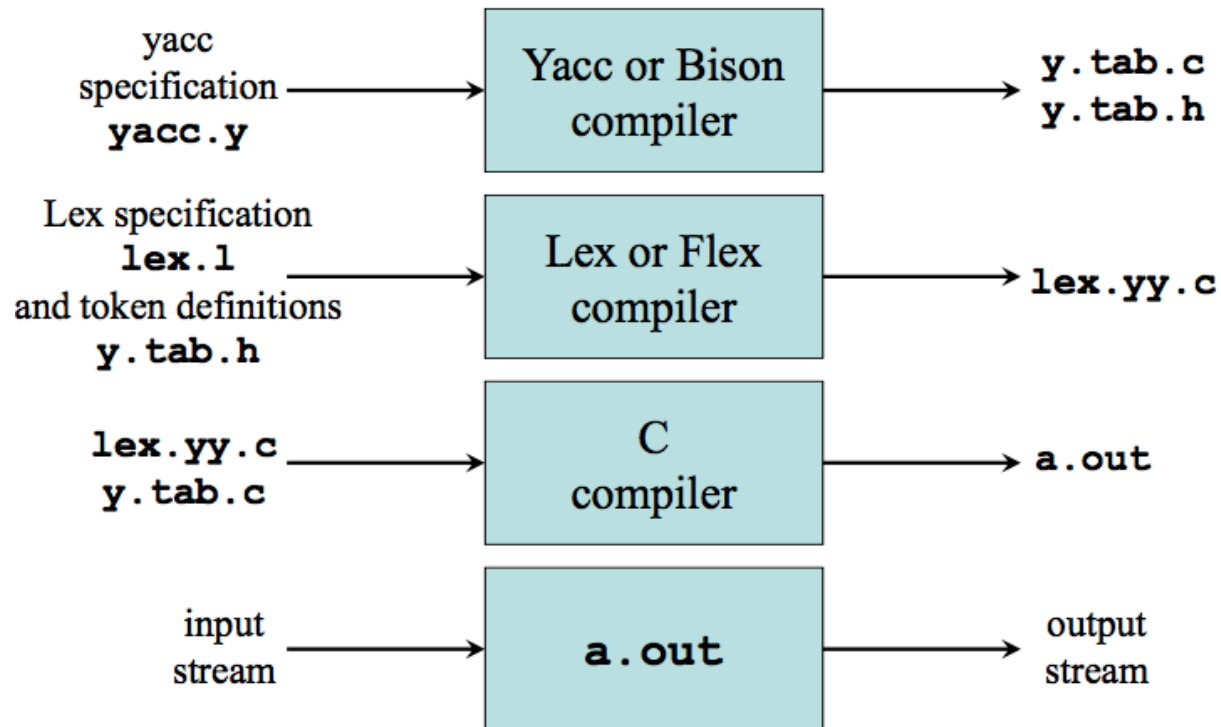
- `$. /test < exprs.txt`

```
1 1 + 5
2 1 * 2 + 10
3 10 - 2 -3
```



# Yacc + Lex

- Lex was designed to produce lexical analyzers that could be used with Yacc
- Yacc generates a parser in `y.tab.c` and a header `y.tab.h`
- Lex includes the header and utilizes token definitions
- Yacc calls `yylex()` to obtain tokens



# Example: Yacc + Lex

## parser.y

```
1 %{
2 #include <ctype.h>
3 #include <stdio.h>
4 #define YYSTYPE double /* double type for Yacc stack */
5 %}
6 %token NUMBER
7
8 %left '+' '-'
9 %left '*' '/'
10
11 %%
12
13 lines : lines expr '\n' { printf("= %g\n", $2); }
14       | lines '\n'
15       /* empty */
16       ;
17 expr : expr '+' expr { $$ = $1 + $3; }
18       | expr '-' expr { $$ = $1 - $3; }
19       | expr '*' expr { $$ = $1 * $3; }
20       | expr '/' expr { $$ = $1 / $3; }
21       | '(' expr ')' { $$ = $2; }
22       | NUMBER
23       ;
24
25 %%
26
27 /*
28 int yylex() {
29     int c;
30     while ((c = getchar()) == ' ');
31     if ((c == '.' || isdigit(c)) {
32         ungetc(c, stdin);
33         scanf("%lf", &yylval);
34         return NUMBER;
35     }
36     return c;
37 }
38 */
39
40 int main() {
41     if (yyparse() != 0)
42         fprintf(stderr, "Abnormal exit\n");
43     return 0;
44 }
45
46 int yyerror(char *s) {
47     fprintf(stderr, "Error: %s\n", s);
48 }
```

## lexer.l

```
1 %{
2 #define YYSTYPE double
3 #include "y.tab.h"
4 extern double yyval;
5 %}
6 number [0-9]+\.[0-9]*|[0-9]*\.[0-9]+
7
8 %%
9
10 [ ]      { /* skip blanks */ }
11 {number} { sscanf(yytext, "%lf", &yyval);
12             return NUMBER; }
13 \n|.     { return yytext[0]; }
14
15 %%
16
17 int yywrap(void) {
18     return 1;
19 }
```

Generated by Yacc

Defined in y.tab.c

# Compile and Run ...

---

- Compile

- `$yacc -d parser.y`
- `$lex lexer.l`
- `$gcc -o test y.tab.c lex.yy.c`

- Run

- `$. /test < exprs.txt`

```
1 1 + 5
2 1 * 2 + 10
3 10 - 2 -3
```

# References

---

- 编译原理（第2版）， 章节4.9
- Yacc/Bison - Parser Generators,  
<https://tldp.org/LDP/LG/issue87/ramankutty.html>
- Lex and Yacc – A Quick Tour,  
<https://courses.cs.washington.edu/courses/cse322/07au/slides/lec25.pdf>
- ANTLR, Yacc, and Bison,  
<https://www.cs.csustan.edu/~xliang/Courses/CS4300-20F/Notes/Ch4c.pdf>
- Yacc Practice, <https://epaperpress.com/lexandyacc/pry1.html>
- The Lex & Yacc Page, <http://dinosaur.compilertools.net/>