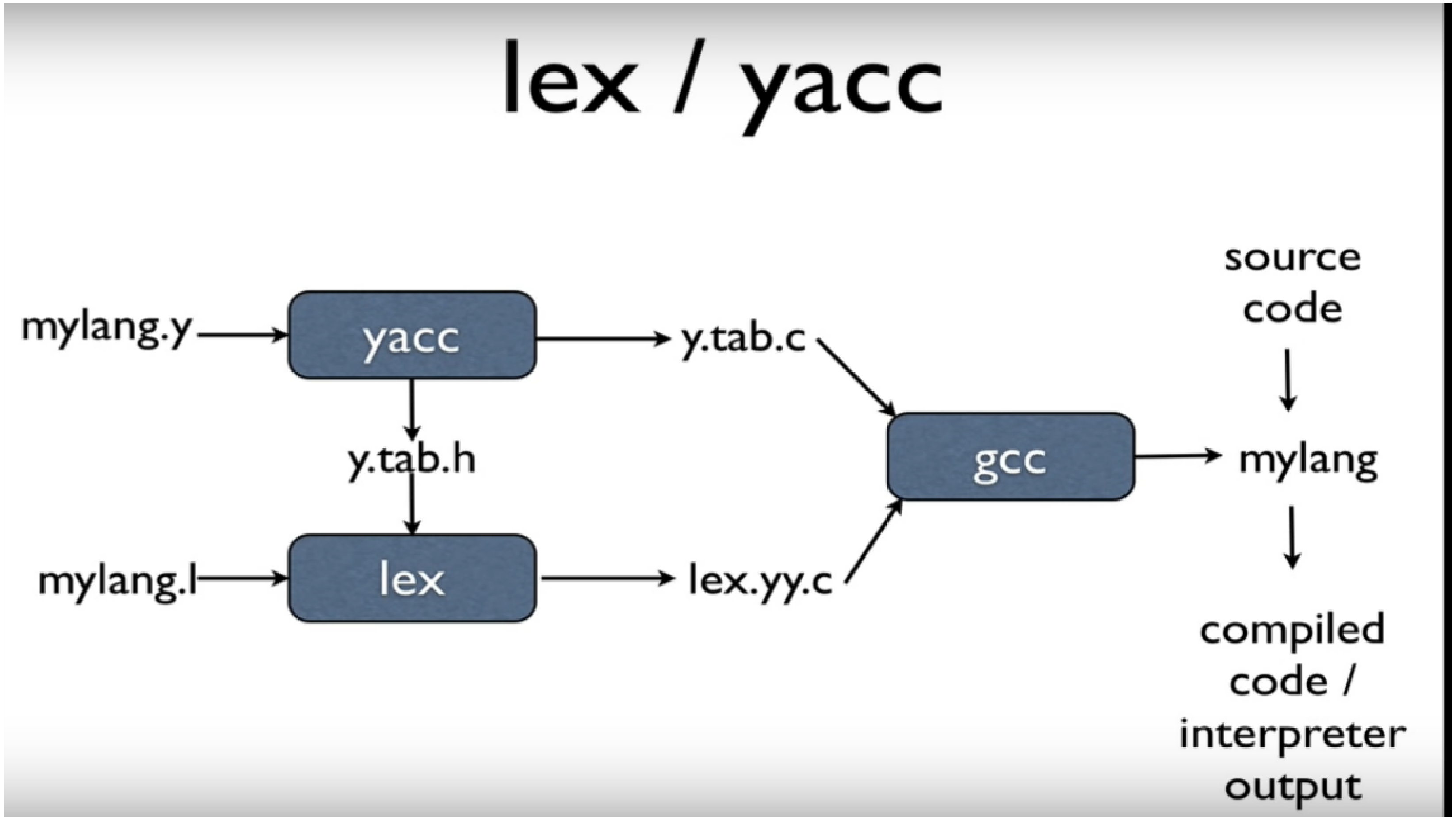
Yacc & lax used together

1. lex: semantic analysis
   > Split the input into tokens

2. yacc: **y**et **a**nother **c**ompiler **c**ompiler.
   > parser and does semantic processing on
   the stream of tokens produced by lex

3.bison: GNU parser parser, upward compatiablity with yacc



This figure shows the whole compilation process in yacc:

1. we have to make two seperate file yacc file (filename.y) and lex file (filename.l)
2. On compiling yacc file we will get two files in output y.tab.c and y.tab.h (y.tab.h will be inculded in the header of lex file) (command yacc -d filename.y)
3. lex filename.l
4. gcc lex.yy.c y.tab.c

Yacc input

- Similar to lex file it also have threee section

1.First part

2. %%

production (grammar rule -> action)

%%

3. Third part ()

Yacc first part

- First part of a yacc specification include:

  >C decleration enclosed in %{ %}

  > yacc defination :

  - %start (to designate the root or start of productions)
  - %token (different token identity or token types)
  - %union (in yacc we could have tokens of different types so in a order to return token of different types we use this)
  - %type (to represent the type of token)

Production part/second part

1. The middle section represent a grammar

- a set of production. The left-hand side (non terminal) of a production is followed by a colon, and a right-hand side.

2. Multiple right-hand side may follow seperated by a ‘|’

3. Actions associated with a rule are entered in braces.

Example of yacc Production/rules

statements: statement

{printf('statement');} (associated action is written in { })

| statement statements

{printf('statements\n');}

statement: identifier '+' identifier

{printf('plus\n');}

statement: identifier '-' identifier

{printf('minus\n');}

Yacc production

1. $1, $2, $3, $4can be refer to the value associated with symbol

2. $$ refer to the vaule of the left

3. Every symbol have value associated with it (incuding token and non-terminals)

4. Default action: $$= $I

Example: Statement: identifier ‘+’ identifier

{$$ = $1+$3 }

Statement: identifier ‘-’ identifier

{$$ = $1-$3}

Sample Production for Arithmetic expression

```
E:E'+'E {$$=$1+$3;}
 |E'-'E {$$=$1-$3;}
 |E'*'E {$$=$1*$3;}
 |E'/'E {$$=$1/$3;}
 |E'%'E {$$=$1%$3;}
 |'('E')' {$$=$2;}
 | NUMBER {$$=$1;}
 ;
```

Symbol notation for understanding:

E: -> statement: (in previous slide)

{} associated action

| alternative production

Yacc third part

> contains vaild C code that support the langauge processing

1. ***Symbol table implementation***
2. ***Function that might be called by action associated with the production in the second part***

Example arthmatic expression

Section 1

```
%{
void yyerror (char *s);
int yylex();
#include <stdio.h>
#include <stdlib.h>
%}
%start E (indicate root of production rules)
%token NUM ID
%left '+' '-'
%left '*' '/'
```

Section 2

```
%%
E : T {printf("Result = %d\n", $$);
       Return 0;}

T :
    T '+' T { $$ = $1 + $3; }
    | T '-' T { $$ = $1 - $3; }
    | T '*' T { $$ = $1 * $3; }
    | T '/' T { $$ = $1 / $3; }
    | '-' NUM { $$ = -$2; }
    | '-' ID { $$ = -$2; }
    | '(' T ')' { $$ = $2; }
    | NUM { $$ = $1; }
    | ID { $$ = $1; };
%%
```

Section 3 yacc

```
int main() {
    printf("Enter the expression\n");
    yyparse();
}

/* To show error messages */
int yyerror(char* s) {
    printf("\nExpression is invalid\n");
}
```

Lex file

```
%{
  #include "y.tab.h" (important)
  extern yylval;
}%
%%
[0-9]+    {
              yylval = atoi(yytext);
              return NUMBER;
          }
[a-zA-Z]+    { return ID; }
[ \t]+         ;  /*For skipping whitespaces*/

\n            { return 0; }
.            { return yytext[0]; }

%%
int yywrap(void) {return 1;}
```

atoi conversion to int

How to compile file

- yacc -d arth.y
  - In the output it will generate two files: y.tab.c y.tab.h
- lex cal.l
  - Will generate lex.yy.c
- gcc lex.yy.c y.tab.c -o output_file

Input: 3+4

Result = 7