

# Lex & Yacc

by  
H. Altay Güvenir

A compiler or an interpreter performs its task in 3 stages:

## 1) Lexical Analysis:

**Lexical analyzer**: scans the input stream and converts sequences of characters into tokens.

**Token**: a classification of groups of characters.

Examples:	<b>Lexeme</b>	<b>Token</b>
	Sum	ID
	for	FOR
	=	ASSIGN_OP
	==	EQUAL_OP
	57	INTEGER_CONST
	"Abcd"	STRING_CONST
	*	MULT_OP
	,	COMMA
	:	SEMICOLUMN
	(	LEFT_PAREN

**Lex** is a tool for writing lexical analyzers.

## 2) Syntactic Analysis (Parsing):

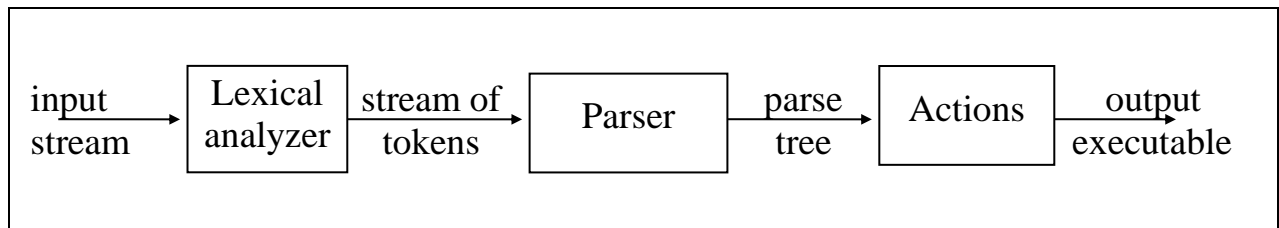
Parser: reads tokens and assembles them into language constructs using the grammar rules of the language.

**Yacc** (Yet Another Compiler Compiler) is a tool for constructing parsers.

## 3) Actions:

Acting upon input is done by code supplied by the compiler writer.

*Basic model of parsing for interpreters and compilers:*

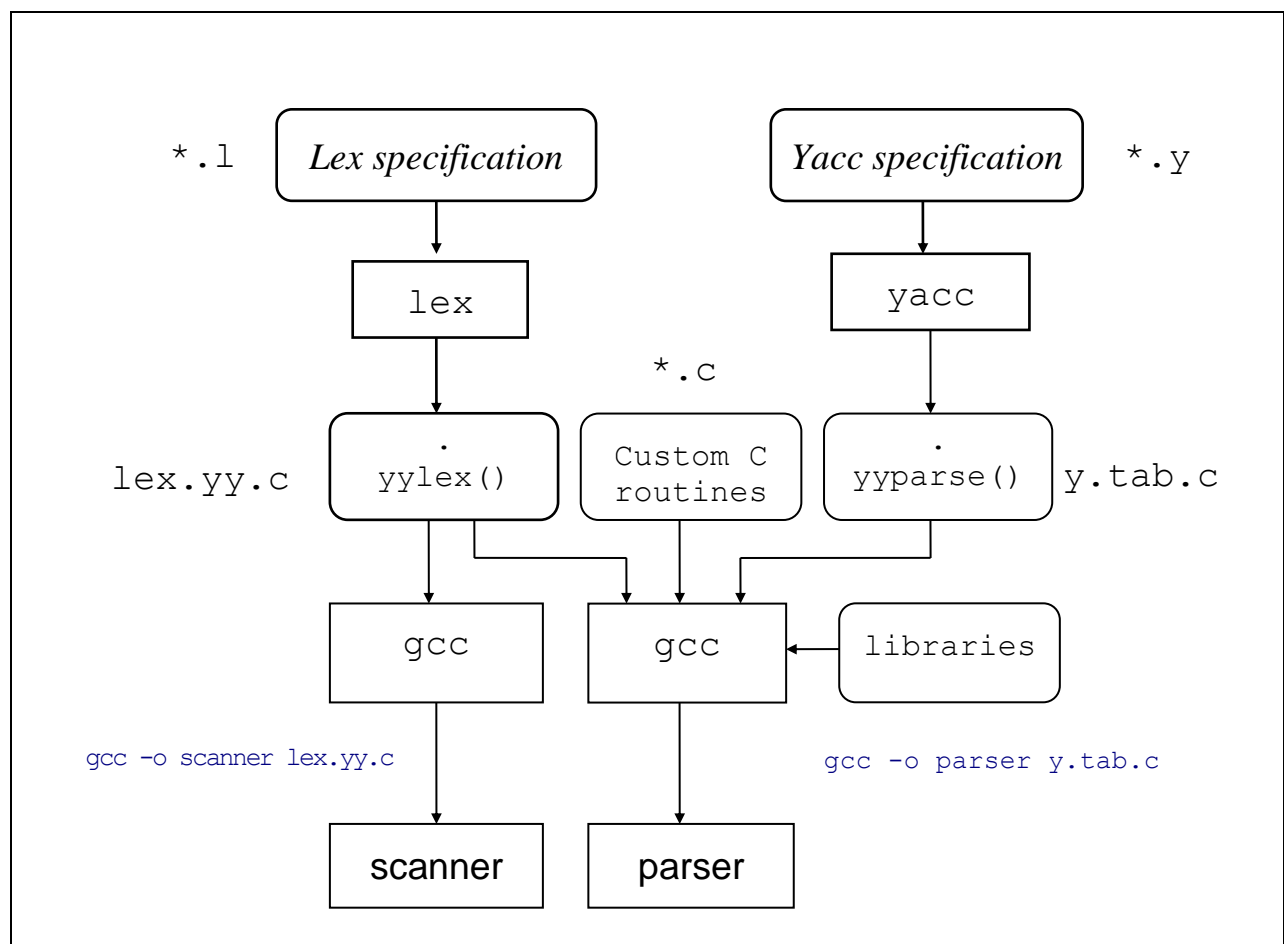


**Lex:** reads a specification file containing regular expressions and generates a C routine that performs lexical analysis.

Matches sequences that identify tokens.

**Yacc:** reads a specification file that codifies the grammar of a language and generates a parsing routine.

Using lex and yacc tools:



## Lex

### Regular Expressions in lex:

a	matches a
abc	matches abc
[abc]	matches a, b or c
[a-f]	matches a, b, c, d, e, or f
[0-9]	matches any digit
X+	matches one or more of X
X*	matches zero or more of X
[0-9]+	matches any integer
(...)	grouping an expression into a single unit
	alternation (or)
(a b c)*	is equivalent to [a-c]*
X?	X is optional (0 or 1 occurrence)
if(def)?	matches if or ifdef (equivalent to if ifdef)
[A-Za-z]	matches any alphabetical character
.	matches any character except newline character
\.	matches the . character
\n	matches the newline character
\t	matches the tab character
\\	matches the \ character
[ \t]	matches either a space or tab character
[^a-d]	matches any character other than a,b,c and d

### Examples:

Real numbers, e.g., 0, 27, 2.10, .17

```
[0-9]+| [0-9]+\.[0-9]+| \.[0-9]+  
[0-9]+(\.[0-9]+)?| \.[0-9]+  
[0-9]*(\.)?[0-9]+
```

To include an optional preceding sign: `[+-]?[0-9]*(\.)?[0-9]+`

Contents of a lex specification file:

```
definitions
%%
regular expressions and associated actions (rules)
%%
user routines
```

**Example** (\$ is the unix prompt):

```
$emacs ex1.1
$ls
ex1.1
$cat ex1.1
%option main
%%
zippy printf("I recognized ZIPPY");
$lex ex1.1
$ls
ex1.1 lex.yy.c
$gcc -o ex1 lex.yy.c
$ls
ex1 ex1.1 lex.yy.c
$emacs test1
$cat test1
tom
zippy
ali zip
and zippy here
$cat test1 | ./ex1                    or $./ex1 < test1
tom
I recognized ZIPPY
ali zip
and I recognized ZIPPY here
```

During pattern matching, lex searches the set of patterns for the **single longest possible match**.

```
$cat ex2.1
%option main
%%
zip      printf("ZIP");
zippy    printf("ZIPPY");
```



<u>Input</u>	<u>Output</u>
ali-7.8veli	ali>-7.800000<veli
ali--07.8veli	ali->-7.800000<veli
+3.7.5	>3.700000<>0.500000<

### Other examples

```
/* echo-upcase-wrods.1 */
%option main
%%
[A-Z]+[ \t\n\.\,] printf("%s",yytext);
. ; /* no action specified */
```

The scanner with the specification above echoes all strings of capital letters, followed by a space tab (\t) or newline (\n) dot (\.) or comma (\,) to stdout, and all other characters will be ignored.

<u>Input</u>		<u>Output</u>
Ali VELI	→	A7, X. 12
HAMI BEY a		VELI → X.
		HAMI BEY

Definitions can be used in definitions

```
/* def-in-def.1 */
%option main
alphanumeric [A-Za-z_$_]
digit [0-9]
alphanumeric ({alphanumeric}|{digit})
%%
{alphanumeric}{alphanumeric}* printf("Java identifier");
\, printf("Comma");
\{ printf("Left brace");
\= printf("Assignment op");
\=\= printf("Equality op");
```

Among rules that match the same number of characters, the rule given first will be chosen.

Example,

```
/* rule-order.1 */
%option main
%%
for printf("FOR");
[a-z]+ printf("IDENTIFIER");
```

for input

for count = 1 to 10

the output would be

FOR IDENTIFIER = 1 IDENTIFIER 10

However, if we swap the two lines in the specification file:

```
%option main
%%
[a-z]+ printf("IDENTIFIER");
for    printf("FOR");
```

for the same input

the output would be

IDENTIFIER IDENTIFIER = 1 IDENTIFIER 10

Important note:

Do not leave extra spaces and/or empty lines at the end of a lex specification file.

# Yacc

Yacc specification describes a CFG, that can be used to generate a parser.

## Elements of a CFG:

1. Terminals: tokens and literal characters,
2. Variables (nonterminals): syntactical elements,
3. Production rules, and
4. Start rule.

Format of a production rule:

```

symbol:    definition
          {action}
          ;

```

Example:

`<a> → <b>c` in BNF is written as `a: b 'c';` in yacc

## Format of a yacc specification file:

```

declarations
%%
grammar rules and associated actions
%%
C programs

```

**Declarations:** To define tokens and their characteristics

%token:	declare names of tokens
%left:	define left-associative operators
%right:	define right-associative operators
%nonassoc:	define operators that may not associate with themselves
%type:	declare the type of variables
%union:	declare multiple data types for semantic values
%start:	declare the start symbol (default is the first variable in rules)
%prec:	assign precedence to a rule
{	
C declarations	directly copied to the resulting C program
}	(E.g., variables, types, macros...)



**Example:** A yacc specification to accept  $L = \{a^n b^n \mid n > 0\}$ .

```
/* anbn0.l */
%%
a  return (A);
b  return (B);
.  return (yytext[0]);
\n return ('\n');
%%
int yywrap() { return 1; }
```

Function **yywrap()** is called by lex when input is exhausted.

Return 1 if you are done or 0 if more processing is required.

```
/*anbn0.y */
%token A B
%%
start:  anbn '\n' {return 0;}
anbn:   A B
        | A anbn B
        ;
%%
#include "lex.yy.c"
main() {
    return yyparse();
}
int yyerror( char *s ) { fprintf(stderr, "%s\n", s); }
```

If the input stream does not match `start`, the default message of "syntax error" is printed and program terminates.

However, customized error messages can be generated.

```
/*anbn1.y */
%token A B
%%
start:  anbn '\n' {printf("  is in anbn\n");
                    return 0;}
anbn:   A B
        | A anbn B
        ;
%%
#include "lex.yy.c"
yyerror(char *s) { printf("%s, it is not in anbn\n", s); }
main() {
    return yyparse();
}
```

```

$./anbn
aabb
    is in anbn
$./anbn
acadbefbg
Syntax error, it is not in anbn
$

```

A grammar to accept  $L = \{a^n b^n \mid n \geq 0\}$ .

```

/*anbn_0.y */
%token A B
%%
start:  anbn '\n' {printf("  is in anbn_0\n");
                  return 0;}
anbn:   empty
        |   A anbn B
        ;
empty:  ;
%%
#include "lex.yy.c"
yyerror(char *s) { printf("%s, it is not in anbn_0\n", s); }
main() {
    return yyparse();
}

```

Positional assignment of values for items.

**\$\$**: left-hand side

**\$1**: first item in the right-hand side

**\$n**: *n*th item in the right-hand side

**Example**: printing integers

```

/* print-int.l */
%%
[0-9]+  {sscanf(yytext, "%d", &yyval);
        return(INTEGER);
        }
\n      return(NEWLINE);
.       return(yytext[0]);
%%
int yywrap() { return 1; }

```

```

/* print-int.y */
%token INTEGER NEWLINE
%%
lines: /* empty */
      | lines NEWLINE
      | lines line NEWLINE {printf("=%d\n", $2);}
      | error NEWLINE {yyerror("Reenter:"); yerrok;}
      ;
line:  INTEGER {$$ = $1;}
      ;
%%
#include "lex.yy.c"
yyerror(char *s) { printf("%s, it is not in anbn_0\n", s); }
main() {
    return yyparse();
}

```

### Execution:

```

$./print-int
7
=7
007
=7
zippy
syntax error
Reenter:
—

```

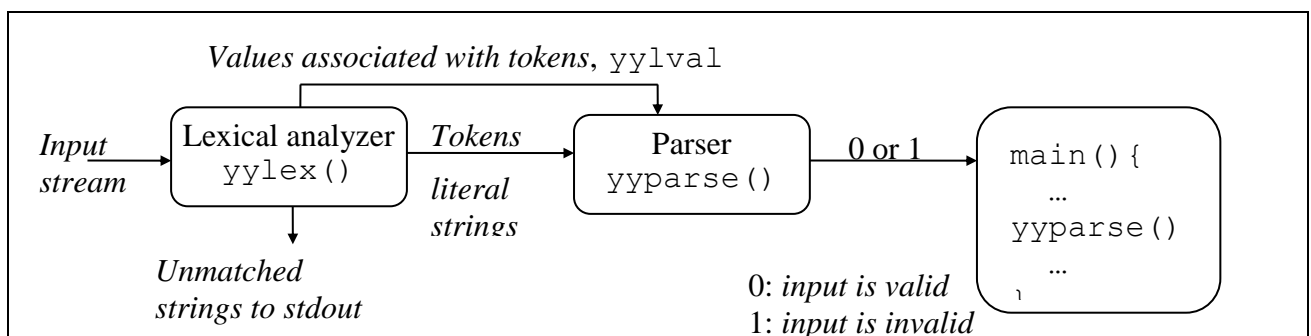
Although right-recursive rules can be used in yacc, **left-recursive rules are preferred**, and, in general, generate more efficient parsers.

The type of **yylval** is `int` by default. To change the type of `yylval` use macro `YYSTYPE` in the declarations section of a yacc specifications file.

```

%{
#define YYSTYPE double
%}

```



If there are more than one data types for token values,  
yylval is declared as a union.

**Example** with three possible types for yylval:

```
%union{
    double  real;    /* real value */
    int      integer; /* integer value */
    char     str[30]; /* string value */
}
```

**Example:**

```
yytext = "0012", type of yylval: int, value of yylval: 12
yytext = "+1.70", type of yylval: float, value of yylval: 1.7
```

The **type of** associated values of **tokens** can be specified by %token as

```
%token <real> REAL
%token <integer> INTEGER
%token <str> IDENTIFIER STRING
```

**Type of variables** can be defined by %type as

```
%type <real> real-expr
%type <integer> integer-expr
```

To return values for tokens from a lexical analyzer:

```
/* lexical-analyzer.l */
alphabetic  [A-Za-z]
digit       [0-9]
alphanumeric ({alphabetic}|{digit})

[+-]?{digit}*{(\.)?{digit}+    {sscanf(yytext, %lf", &yylval.real);
                                return REAL;
                                }
{alphabetic}{alphanumeric}*    {strcpy(yylval.str, yytext);
                                return IDENTIFIER;
                                }

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

**Example:** yacc specification of a calculator is given the web page of the course.

## Actions between rule elements:

```
/* lex specification */
%%
a return A;
b return B;
\n return NL;
. ;
%%
int yywrap() { return 1; }
```

```
/* yacc specification */
%{
#include <stdio.h>
%}
%token A B NL
%%
s: {printf("1");}
   a
   {printf("2");}
   b
   {printf("3");}
   NL
   {return 0;}
;
a: {printf("4");}
   A
   {printf("5");}
;
b: {printf("6");}
   B
   {printf("7");}
;
%%
#include "lex.yy.c"
int yyerror(char *s) {
    printf ("%s\n", s);
}
int main(void){ yyparse(); }
```

input: ab

output: 1452673

input: aa

output: 14 syntax error  
526

input: ba

output: 14 syntax error

## Conflicts

**Pointer model:** A pointer moves (right) on the RHS of a rule while input tokens and variables are processed.

```
%token A B C
%%
start: A B C      /* after reading A: start: A B C */
      ↑           ↑
```

When all elements on the right-hand side are processed (pointer reaches the end of a rule), the rule is **reduced**.

If a rule reduces, the pointer then returns to the rule it was called.

**Conflict:** There is a **conflict** if a rule is reduced when there is more than one pointer. **yacc looks one-token-ahead** to see if the number of tokens reduces to one before declaring a conflict.

**Example:**

```
%token A B C D E F
%%
start: x | y;
x: A B C D;
y: A B E F;
    ↑   ↑
```

After tokens A and B, either one of the tokens, or both will disappear. For example, if the next token is E, the first, if the next token is C the second token will disappear. If the next token is anything other than C or E both pointers will disappear. Therefore there is no conflict.

The other way for pointers to disappear is for them to merge in a common subrule.

**Example:**

```
%token A B C D E F
%%
start: x | y;
x: A B z E;
y: A B z F;
z: C D;
    ↑   ↑
```

Initially there are two pointers, one in  $x$ , the other in  $y$  rules. After reading A, and B, these two pointers shift. Then these two pointers merge in the  $z$  rule. The state after reading token C is shown below.

```
%token A B C D E F
%%
start: x | y ;
x: A B z D E ;
y: A B z D F ;
z: C↑;
```

However, after reading A B C, the  $z$  rule reduces. There is only one pointer when  $z$  reduces. Then this pointer splits again into two pointers in  $x$  and  $y$  rules.

```
%token A B C D E F
%%
start: x | y ;
x: A B z↑D E ;
y: A B z↑D F ;
z: C;
```

### Conflict example:

```
%token A B
%%
start: x B | y B ;
x: A↑;      reduce
y: A↑;      reduce      reduce/reduce conflict on B.
```

After A, there are two pointers. Both rules ( $x$  and  $y$ ) want to reduce at the same time. If the next token is B, there will be still two pointers. Such conflicts are called **reduce/reduce** conflict.

Note that yacc looks **one-token-ahead** before declaring any conflict.

```
%token A B C D E
%%
start: A x C D | A y C E ;
x: B↑;
y: B↑;
```

The pointers in  $x$  and  $y$  rules will reduce on C, resulting on reduce/reduce conflict on C, although the grammar is not ambiguous. If yacc has looked two tokens ahead, it would have realized that only one pointer would remain on tokens D or E, and no pointer otherwise, so it would not declare any conflict.

Another type of conflict occurs when one rule reduces while the other shifts.  
Such conflicts are called **shift/reduce** conflicts.

### Example:

```
%token A R
%%
start: x | y R;
x: A↑R ;      shift
y: A↑;        reduce      shift/reduce conflict on R
```

After A, y rule reduces, x rule shifts. The next token for both cases is R.

### Example:

```
%token A
%%
start: x | y;
x: A;↑      reduce
y: A;↑      reduce      reduce/reduce conflict on $end.
```

At the end of each string there is a \$end token. Therefore, yacc declares reduce/reduce conflict on \$end for the grammar above.

### Empty rules:

```
%token A B
%%
start: empty A A      equivalently   start: {...} A A
      | A B;
empty: ;
```

Without any tokens

```
%token A B
%%
start: empty A A
      | ↑A B;
empty: ↑;      shift/reduce conflict on A
```

If the next token is A, the empty rule will reduce and second rule (of start) will shift. Therefore yacc declares **shift/reduce conflict** on A for this grammar.

### Debugging:

```
$yacc -v filename.y
```

produces a file named y.output for debugging purposes.



## Example:

```
%token A P
%%
s: x | y P;
x: A P; /* shifts on P */
y: A;   /* reduces on P */
```

The `y.output` file for the grammar above is shown below:

```
0 $accept : s $end
1 s : x
2   | y P
3 x : A P
4 y : A

state 0
$accept : . s $end
A shift 1
. error
s goto 2
x goto 3
y goto 4

1: shift/reduce conflict (shift 5, reduce 4) on P
state 1
x : A . P (3)
y : A . (4)
P shift 5

state 2
$accept : s . $end (0)
$end accept

state 3
s : x . (1)
. reduce 1
```

**s : x is called rule number 1**

**If A is seen, shift the pointer, goto state 1**

**Otherwise call yyerror()**

**Shift and goto state 5**

**Reduce rule 4**

**Shift/reduce conflict on P**

**If the next token is P, pointer in rule 3 will shift**

**If the next token is P, this rule (4) will reduce.**

**Default action: The system will choose to shift and goto state 5.**

**State2: input matched the start variable s, if this is the end of string, accept it.**

**State 3: rule (1) s: x is to reduce on any text token**

**Any character or token**

State 4: pointer is in rule 2. After y rule is processed

```

state 4
    s : y . P (2)
    P shift 6
    . error
  
```

If the next token is P, shift the pointer, go to state 6

If the next token is anything else, call yyerror()

```

state 5
    x : A P . (3)
    . reduce 3
  
```

```

state 6
    s : y P . (2)
    . reduce 2
  
```

Rules never reduced:

```

y : A (4)
  
```

State 1 contains 1 shift/reduce conflict.

\$end, A, P, .

\$accept, s, x, y

4 terminals, 4 nonterminals

5 grammar rules, 7 states

Using the make utility on linux systems:

Contents of the file named Makefile:

```

parser: y.tab.c
    gcc -o parser y.tab.c
y.tab.c: parser.y lex.yy.c
    yacc parser.y
lex.yy.c: scanner.l
    lex scanner.l
  
```

Just type

make

It automatically determines which source files (in this example, y.tab.c, parser.y, lex.yy.c, scanner.l) of a program (parser in this example) need to be recompiled and/or linked