

Lex & Yacc

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:	Lexeme	Token
	Sum	ID
	for	FOR
	:=	ASSIGN_OP
	=	EQUAL_OP
	57	INTEGER_CONST
	*	MULT_OP
	,	COMMA
	(LEFT_PAREN

Lex is a tool for writing lexical analyzers.

2) Syntactic Analysis (Parsing):

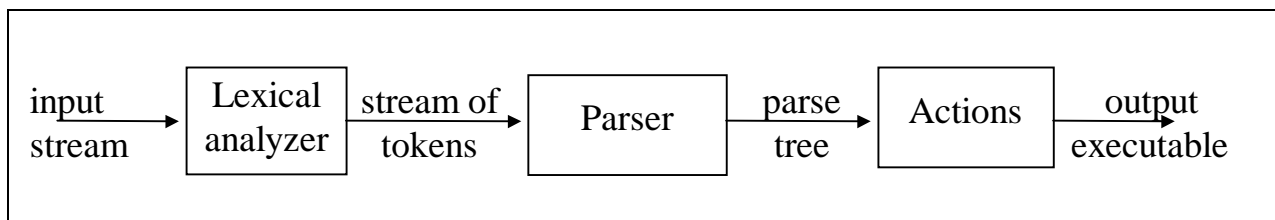
Parser: reads tokens and assembles them into language constructs using the grammar rules of the language.

Yacc 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:

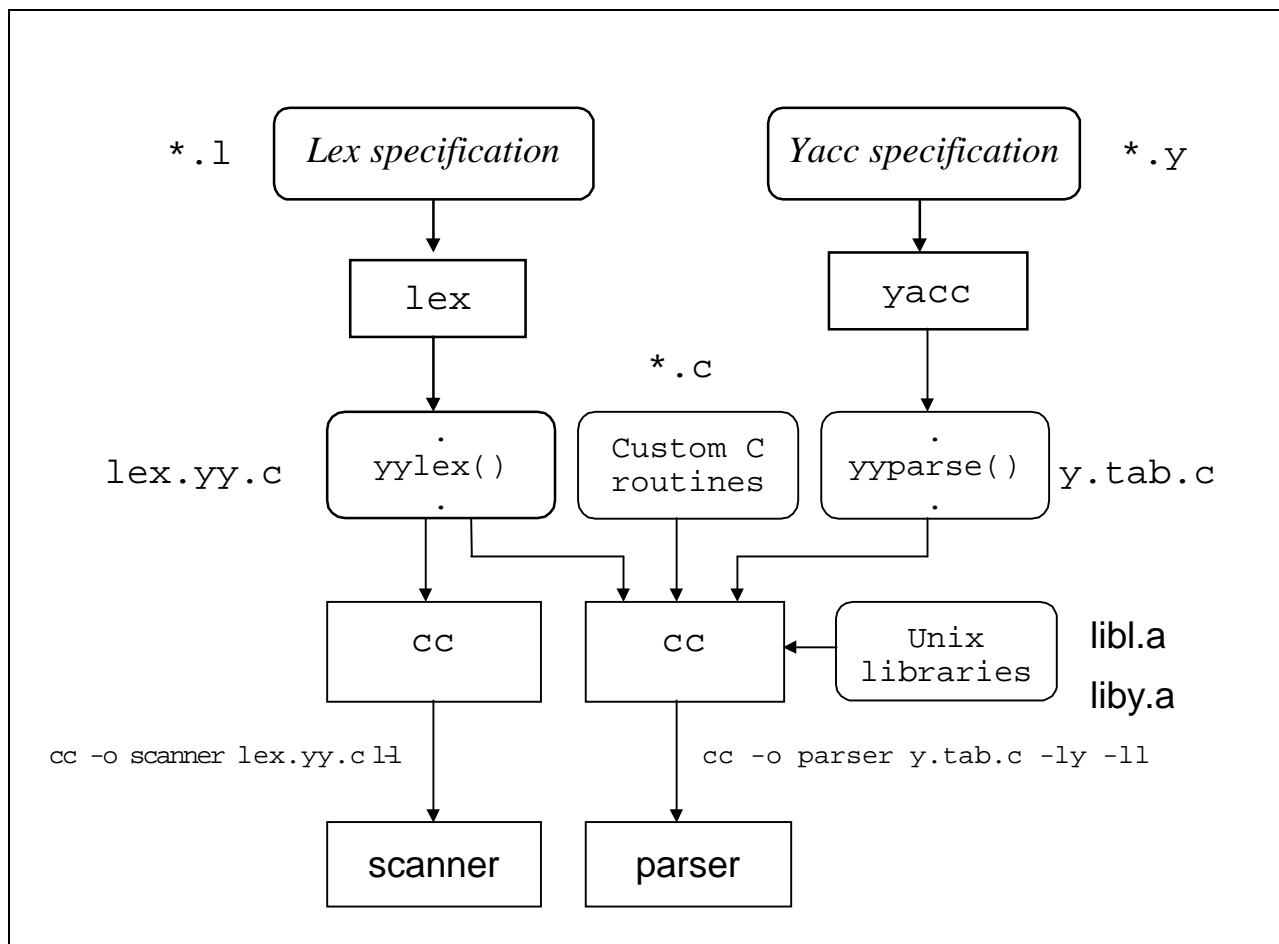


Lex: reads a specification file containing regular expressions and generates a C routine that performs lexical analysis.

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:



The order of libraries is important: `-ly` must precede `-ll`.

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
%%
zippy printf("I recognized ZIPPY");
$lex ex1.1
$ls
ex1.1 lex.yy.c
$cc -o exllex.yy.c tl
$ls
exl 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 Irecongized ZIPPY here

```

Lex matches the input string the longest regular expression possible

```

$cat ex2.1
%%
zip    printf("ZIP");
zippy  printf("ZIPPY");
$cat test2
Azip andzippyr zipzippy
$cat test2 | ex2
AZIP andZIPPYr ZIPZIPPY

```

Lex declares an external variable called `yytext` which conatins the matched string

```

$cat ex3.1
%%
tom|jerry printf(">%s<" yytext);
$cat test3
Did tom chasejerry?
$cat test3 | ex3
Did >tom< chasejerry<?

```

Definitions:

```

%%
[+-]?[0-9]*(\.)?[0-9]+ printf("FLOAT");

```

input: ab7.3c-5.4.3+d++5-

output: abFLOATc-FLOATFLOAT+d+FLOAT-

The same lex specification can be written as:

```

digit [0-9]
%%
[+-]?{digit}*(\.)?{digit}+ printf("FLOAT");

```

Local variables can be defined:

```

digit      [0-9]
sign       [+ -]
%%
float val;
{sign}?{digit}*(\.)?{digit}{sscanf(yytext, "%f", val);
                           printf(">%f<", val);}

```

Input

```

ali-7.8veli
ali--07.8veli
+3.7.5

```

Output

```

ali>-7.800000<veli
ali->-7.800000<veli
>3.700000<>0.500000<

```

Other examples

```

%%
[A-Z]+[ \t\n] printf("%s", yytext);
. ; /* no action specified */

```

The scanner for the specification above echo all strings of capital letters, space and tab to stdout, and all other characters will be ignored.

Input

```

Ali VELI → A7 X 12
HAMI BEY a

```

Output

```

VELI → X
HAMI BEY

```

Definitions can be used in definitions

```

alphanumeric [A-Za-z]
digit [0-9]
alphanumeric      ({alphanumeric}|{digit})
%%
{alphanumeric}{alphanumeric}*printf("Pascal variable");
\,                printf("Comma");
\{                printf("Left brace");
\:\=              printf("Assignment");

```

If more than one regular expression match the same string the one that is defined earlier is used.

Example,

```

%%
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:

```

%%
[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 the 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:

$A \rightarrow Bc$ is written in yacc as `a: b 'c' ;`

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 $L = \{a^n b^n \mid n > 1\}$.

```
/* anbn.l */
%%
a  return (A);
b  return (B);
.  return (yytext[0]);
\n return ('\n');
```

```
/*anbn.y */
%token A B
%%
start:  anbn '\n' {return 0;}
anbn:   A B
        | A anbn B
        ;
%%
#include lex.yy.c"
```

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.

```
/*anbn.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(s)
char *s;
{ printf("%s, it is not anbn\n", s);
}
```

```
$anbn
aabb
  is in anbn
$anbn
acadbefbg
Syntax error, it is not anbn
$
```

Example: printing integers

```

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

```

```

/* printint.y */
%token INTEGER NEWLINE
%%
lines: /* empty */
      | lines NEWLINE
      | lines line NEWLINE {printf("%d\n", $2);}
      | error NEWLINE {yyerror("Reenter:"); yyerrok;}
;
line:  INTEGER {$$ = $1;}
;
%%
#include "lex.yy.c"

```

Positional assignment of values for items.

\$\$: left-hand side

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

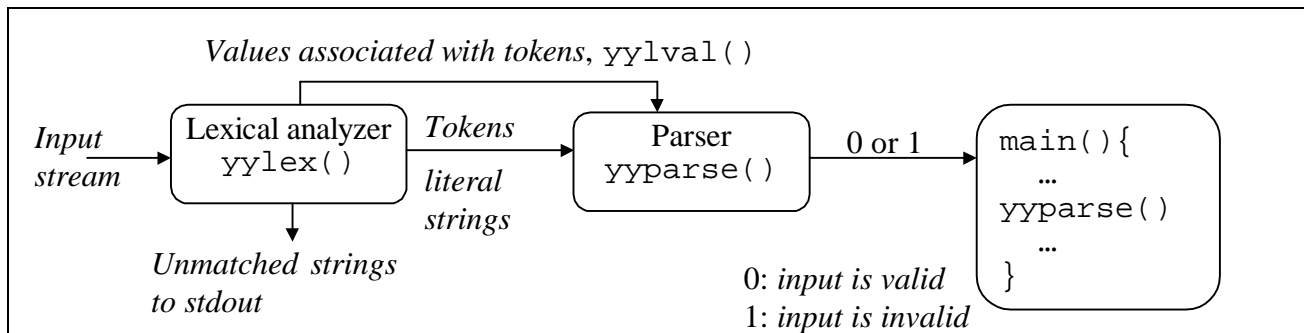
\$n: nth item in the right-hand side

```

$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
```

Types 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 */
alphanumeric [A-Za-z]
digit [0-9]
alphanumeric ({alphanumeric}|{digit})
%%
[+-]?{digit}*{digit}+ {sscanfyytext, "%f", &yylval.real);
                        return REAL;
                        }
{alphanumeric}{alphanumeric}* {strcpyyylval.str, yytext);
                        return IDENTIFIER;
                        }
```

Example: yacc specification of a calculator

In the web page of the class.

Actions between rule elements:

```
/* lex specification */
%%
a return A;
b return B;
\n return NL;
. i
```

input: ab

output: 1452673

input: aa

output: 14 syntax error
526

input: ba

output: 14 syntax error

```
/* yacc specification */
%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");}
  ;
```

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**.

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.

Example:

<pre>%token A B C D E F %% start: x y; x: A B C D; y: A B E F;</pre>	<pre>start: x y; x: A B C D; y: A B E F;</pre>
--	--

after AB

after ABC

No conflict.

Conflict example:

```
%token A
%%
start: x|y;
x: A↑ reduce
y: A↑ reduce
```

After A, there are two pointers. Both rules (x and y) want to reduce at the same time. Such conflicts are called **reduce/reduce** 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 |yR;
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.

Reduce/reduce conflict example:

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

This specification contains a **reduce/reduce** conflict.

On the other hand,

```
%token A B C
%%
start: x B | y C;
x: A;
y: A;                has no conflict.
```

This is because **yacc can look ahead one token** after the point.

However,

```
%token A B C D
%%
start: x B C | y B D;
x: A;
y: A;                reduce/reduce conflict on B
has reduce/reduce conflict; although there is no ambiguity.
```

Empty rules:

```
%token A B
%%
start: empty  $\mathbb{A}$    equivalently   start: {...} A A
      | A B;
empty: ;                shift/reduce conflict on A
```

This specification contains a **shift/reduce** conflict.

Debugging:

```
$yacc -vfilename.y
```

produces a file named `y.output` for debugging purposes.

Contents of a Makefile:

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