What are the main differences ?

1. Token definitions in separate file.
2. Parser just checks if the tokens are legitimate and passes it on to YACC.

```
%{
/* mystuff.y - my Yacc parser */

#include
%}

    /* need to know what tokens to create! */

%token NOUN PRONOUN VERB ADVERB ADJECTIVE PREPOSITION
CONJUNCTION

%%

sentence:  subject VERB object { printf("Sentence is valid.\n");
   ;
subject :  NOUN
  |      PRONOUN
   ;

object :   NOUN
   ;

%%

extern FILE *yyin;

main()
{
   while(!feof(yyin))
      yyparse();
}

yyerror ( char *s )
{
   fprintf(stderr, "%s\n", s);
}
> lex mystuff.l            // makes "lex.yy.c"
> yacc -d mystuff.y        // makes "y.tab.c" and "y.tab.h"
> cc -c lex.yy.c y.tab.c   // compile C files
> cc lex.yy.o y.tab.o -ll  // link object and library files
```

```
short input_state;

short add_word     ( short type, char *word );
short lookup_word  ( char *word );
%}

%%

\n       { state = LOOKUP; }

^defnode  { state = NODE; }
^definput { state = INPUT; }
^deffinal { state = FINAL; }
^defstart { state = START; }

[a-zA-Z]+ {
   if (state != LOOKUP)
      add_word(state,yytext);
   else
   {
      switch(lookup_word(yytext))
      {
        /* read in name and if OK pass to Yacc */

        case NODE:
          return NODE;
          break;
        case INPUT :
          return INPUT;
          break;
        .
        .
        .
        /* what if it's mispelt or undeclared? */

        default : printf ("%s: Unknown token!", yytext");

  }
}

\.\n    { input_state = LOOKUP; return 0; }  /* new rule ! */

.    ;    /* anything else - just ignore, action is null */

%%
```

```
      switch(lookup_word(yytext))
 {
        /* read in name and do something */

        case NODE: ...
        case INPUT : ...
        .
        .
        .
        /* what if it's mispelt or undeclared? */

        default : printf ("%s: Unknown token!", yytext);
      }
   }
}

.    ;    /* anything else - just ignore, action is null */

%%

        /* down here we would have our "main()" and    */
        /* the functions "add_word" and "lookup_word"  */
        /* that would maintain lists of words and      */
        /* and search through them. As long as they do */
        /* their job it doesn't matter what they look  */
        /* like. You can do that stuff right?          */
```

So this means that you could have an input file like :

```
defnode alpha beta gamma
definput able baker
defnode zeta ...
```

The same routine if you want to run with YACC.

```
%{

#include "y.tab.h"    /* we need to know what the tokens are */

#define LOOKUP   0    /* this is OK, Yacc reserves 0, not Lex */
```

```
%%

int cc = 0, wc = 0, lc = 0;

{nonws}+    cc += yyleng; ++wc;
{ws}+       cc += yyleng;
\n          ++lc; ++cc;
<<EOF>>  {
    printf( "%8d %8d %8d\n", lc, wc, cc );
    yyterminate();
    }
```

<u>Building Symbol Tables</u>

```
%{

enum {
   LOOKUP = 0,
   NODE,
   INPUT,
   FINAL,
   START
};short input_state;

short add_word     ( short type, char *word );
short lookup_word  ( char *word );
%}

%%

    /* rules section */

\n        { state = LOOKUP; }

^defnode  { state = NODE; }
^definput { state = INPUT; }
^deffinal { state = FINAL; }
^defstart { state = START; }

[a-zA-Z]+ {
    if (state != LOOKUP)
       add_word(state,yytext);
    else
    {
```

2. In case of a reduce/reduce conflict, reduce by earlier grammar rule.

Types
Default the values returned by actions and the lexical analyzer are integers.

Yacc can also support values of other types, including structures.Yacc keeps track of the types, and inserts appropriate union member names so that the resulting parser will be strictly type checked.

Declare the Yacc value stack as a union.
associates union member names to each token and nonterminal symbol having a value.

Check out commands like "lint".
Approach

```
%union  {
            body of union ...
            }
```

This will cause the external variables yyval and yylval to have type equal to this union.
yacc -d the union declaration is copied onto the y.tab.h file.
OR

```
typedef union {
            body of union ...
            } YYSTYPE;
```

in a separate header file and include this in the declaration section , using ______________.
Ans: %{ and %}.

How do you reference the type and attach it to a particular token ?
< name > is used to indicate a union member name.
If this follows one of the keywords %token, %left, %right, and %nonassoc, the union member name is associated with the tokens listed.

```
%left  <optype>  '+'  '-'
%type  <nodetype>  expr  stat
```

<u>**A Lex Example**</u>

```
%{
/* a sample bit of code */
%}

ws    [ \t]
nonws [^ \t\n]
```

A precedence and associativity is associated with each grammar rule; it is the precedence and associativity of the last token or literal in the body of the rule.

%prec	**Used to override precedence**

Eg:

```
%left  '+'  '-'
     %left  '*'  '/'

     %%

     expr   :     expr  '+'  expr
          |     expr  '-'  expr
          |     expr  '*'  expr
          |     expr  '/'  expr
          |     '-'  expr     %prec  '*'
          |     NAME
;
```

Why do you need this when the disambiguating rules are implicit ?
____.

left associative implies reduce,
right associative implies shift,
and nonassociating implies error.

Yacc

YYPARSE	**Function created by YACC.**
y.tab.c	**C file created by YACC.**
Diagnostics	
extern variables.	
yychar	**Contains the lookahead token number.**
yydebug	**If set to 1, prints out a verbose description of its errors.**

YACC ENCOURAGES LEFT-LINEAR GRAMMAR . WHY ?
____.

*** Yacc is a shift reduce parser.**
*** Anything that is not on the left-hand side of a production rule is assumed to be a terminal symbol. It is a token or a character.**

*** Resolving ambiguity. Yacc uses two rules:**

1. In case of a shift/reduce conflict, do the shift.

NOTE:
The token numbers should be unique.
0 is for EOF. All lexical analyzers should be prepared to return a 0 upon reaching the EOF.

How does the PARSER work ?
FSA stack.
Look ahead token.
Shift/Reduce

Avoiding ambiguity in rules
1. Rewrite
2. Use precedence / associativity of YACC.

exp: exp '+' number | exp '-' number | exp '*' number | exp '/' number | number;

exp: exp '+' mulexp | exp '-' mulexp;
mulexp: mulexp '*' primary | mulexp '/' primary | primary;
primary :'(' exp ')' | '-' primary | number;

Yacc to the rescue
Precedence and associativity.
Left / Right associativity / Non-associativity
expr : expr OP expr
expr : UNARY expr

%left OP
%nonassoc UNARY

The first line gets the least priority.

```
    %right  '='
    %left  '+'  '-'
    %left  '*'  '/'

    %%

    expr    :      expr  '='  expr
            |      expr  '+'  expr
            |      expr  '-'  expr
            |      expr  '*'  expr
            |      expr  '/'  expr
            |      NAME
a  =  b  =  c*d  -  e  -  f*g
```

How will this be intepreted by the above rule ?
a = (b = (((c*d)-e) - (f*g)))

lexical rules to be named ``better'' the UNIX command sequence can just be:

```
yacc good
lex better
cc y.tab.c -ly -ll
```

yylex	**Gets the token**
yylval	**Contains the token value.**

Want to see the actual code ?

```
 yylex(){
        extern int yylval;
        int c;
        . . .
        c = getchar();
        . . .
        switch( c ) {
            . . .
        case '0':
        case '1':
         . . .
        case '9':
            yylval = c-'0';
            return( DIGIT );
            . . .
            }
```

Yacc chooses the token numbers associated with the Token , by DEFAULT. How do you over ride it ?

```
% token 100
```

How does Lex know about the tokens ?

```
/* y.tab.h - file of token defns */

#define NODE    101
#define INPUT   102
#define FINAL   103
#define START   104
```

So, include this file containing the information about the tokens in the Lex file !

```
%{

#include "y.tab.h"
%}
```

ACTIONS

```
A      :      '('  B  ')'
                          {      hello( 1, "abc" );  }
```

C<u>ommunication between action and the parser</u>

```
$$ $1 ....
expr    :       '('  expr  ')'        {  $$ = $2 ;  }
A       :       B    ;
```

frequently need not have an explicit action.Why ?

Because the parser assumes that the value of the rule is the value of the first element.

Shall we shift gears ? What if you need to get control before a rule is fully parsed.

```
A      :      B
                          {  $$ = 1;  }
                    C
                          {   x = $2;   y = $3;  }
             ;
```

This how the above rule/action is handled.

```
$ACT    :      /* empty */
                          {  $$ = 1;  }
             ;

      A      :      B  $ACT  C
                          {   x = $2;   y = $3;  }
             ;
```

Want to build a parse tree ?

```
expr    :      expr  '+'  expr
                    {  $$ = node( '+', $1, $3 );  }
```

<u>Variable declaration</u>

in the declaration section   %{   int variable = 0;   %}.

Is this global / local ? If it global, can Lex "see" it ?

Global and Lex can "see" it ?

Lex and Yacc Communication

Normally, the default main program on the Lex library calls this routine, but if Yacc is loaded, and its main program is used, Yacc will call yylex(). In this case each Lex rule should
end with

```
return(token);
```

where the appropriate token value is returned. An easy way to get access to Yacc's names for tokens is to compile the Lex output file as part of the Yacc output file by placing the line

```
# include "lex.yy.c"
```

in the last section of Yacc input. Supposing the grammar to be named ``good'' and the

BODY represents a sequence zero or more names and literals.
A literal consists of a character enclosed in single quotes.
the backslash ``\'' is an escape character

Eg:	**'\n'**	**newline**
	'\r'	**return**

```
A     :     B  C  D  ;
A     :     E  F  ;
A     :     G  ;

A     :     B  C  D
          |     E  F
          |     G
          ;
```

Why use Yacc , when Lex does something similar ?

Aug 15, 1947

Lex and Yacc together:
date : month_name '/' day '/' year ;

Yacc:
```
month_name  :  'J' 'a' 'n'   ;
month_name  :  'F' 'e' 'b'   ;

month_name  :  'D' 'e' 'c'   ;
```

Such low-level rules tend to waste time and space, and may complicate the specification beyond Yacc's ability to deal with it.
Specify the syntactic constructions in the grammar rules. A grammar yylex() - supplies Yacc with the tokens.
It returns an integer.
Updates yylval.

But how do you represent a token ?
%token name1 name2 . . . in the declaration section.
%start symbol
What if you do not explicitly declare the start symbol ?

Endmarker Signals the end of the input.

```
                case 'a': printf("first"); break;
                case 'b': printf("second"); break;
                case 'c': printf("third"); break;
                default: ECHO; break;
                }
                }
```

I*s there another way ?*
Yes. Use START.
%Start   name1 name2 ...
BEGIN name1;
<name1>expression

```
 %START AA BB CC
              %%
              ^a          {ECHO; BEGIN AA;}
              ^b          {ECHO; BEGIN BB;}
              ^c          {ECHO; BEGIN CC;}
              \n          {ECHO; BEGIN 0;}
              <AA>magic        printf("first");
              <BB>magic        printf("second");
              <CC>magic        printf("third");
```

<u>Yacc - Parser generator.</u>

Parsing is a way of imposing structure on input (think of it as constructing a tree...)

*** It looks at the structure of an input sequence, which is usually in the form of tokens returned by some lexical analyzer (like lex).**

Yacc source looks like:

```
                              declarations
                %%
                rules
                %%
                programs
```

the smallest legal Yacc specification is

```
%%
     rules
```

Comments as in /* ....*/
A typical rule
A : BODY ;

A   represents a nonterminal name

preferred.

Consider the rules
integer keyword action 1...;
[a-z]+ identifier action 2 ...;
Input is "Integers" . What is the action which will be performed ?
Action 2.
If the Input is "Integer" ?
Action 1.

Do we have time ?

```
she   s++;
he    h++;
\n    |
.     ;
```

Input - she. What are the values of s and h , if initially they are zero ?
s=1, h=0.
Why does this happen ?
What if you have to override this choice ?

REJECT it. How ?

```
                    she   {s++; REJECT;}
    he    {h++; REJECT;}
    \n    |
    .     ;
```

<u>Compilation</u>
lex source cc lex.yy.c -ll
note that what Lex writes is a program named yylex(), the name required by Yacc for its analyzer.
<u>Context Sensitivity</u>
Consider the following problem: copy the input to the output, changing the word magic to first on every line which began with the letter a, changing magic to second on every line which began with the letter b, and changing magic to third on every line which began with the letter c. All other words and all other lines are left unchanged.

```
        int flag;
               %%
               ^a     {flag = 'a'; ECHO;}
               ^b     {flag = 'b'; ECHO;}
               ^c     {flag = 'c'; ECHO;}
               \n     {flag =  0 ; ECHO;}
 magic   {
                      switch (flag)
                      {
```

Lex also permits access to the I/O routines it uses. They are:

1) input()	**which returns the next input character;**
2) output(c)	**which writes the character c on the output; and**
3) unput(c)	**pushes the character c back onto the input stream to be read later by input().**

<u>If C code :</u>
proceeds the first %% line, it is placed at the beginning of the output program.
If it follows the first %% line but precedes the second %% line, it is placed at the beginning of the YYLEX function itself. (One might, for instance, insert declarations of local variables in this way).
if there is a second %% line, anything that follows it (whether it begins in column 1 or not) is assumed to be a programmer-supplied declaration and is inserted at the end of the output program, again as a global declaration.

YYWRAP().
executed when the lexical analyzer reaches the end of the input file.
lex program generates a default yywrap() function which can be overridden.
Used to open additional input files. The yywrap() function should return 0 if it has arranged for additional input, 1 if the end of the input has been reached.
EOF returns a 0.

```
letter       [A-Za-z]
digit        [0-9]
sign         [-+]
        int sum = 0;
%%
        int amount;
{letter}+                   { printf("%s\n", yytext); }
{sign}?{digit}+   { sscanf(yytext, "%d", &amount);
                              sum += amount;
                              printf("%s\n", yytext); }
.|\n                         { ; }
%%
yywrap() {
  printf("\nTotal of integer values = %d\n", sum);
  return 1;
}
```

<u>Resolution of Ambiguity</u>

- **The longest match is preferred.**
- **Among rules which matched the same number of characters, the rule given first is**

Example:

```
\"[^"]*   {
                    if (yytext[yyleng-1] == '\\')
                        yymore();
                    else
                        ... normal user processing
                    }
```

How does this recognize "abc\"def" ? abc"def

Example:

```
 =-[a-zA-Z]   {
                     printf("Op (=-) ambiguous\n");
                     yyless(yyleng-1);
                     ... action for =- ...
                     }
```

Treats "=- a" as the operator being "=-".
How do we make it treat as "=   -a" ?

```
                                   yyless(yyleng-2);
```

Is this the only way to do it ?
No. There are simpler ways.
=-/[A-Za-z]
=/-[A-Za-z].

YYLESS(n): **To indicate that not all the characters matched by the currently successful expression are wanted right now.**
n **The number of characters in yytext to be retained.**

Where does the other characters go ?
BACK to the input.

What is the string matched by ab?c ____________________.

*****	**Repeated expression a***
	Any number of "a" characters including 0.
+	**One or more characters.**
	[A-Za-z][A-Za-z0-9]*

_______________________________________.
This is a typical expression for recognizing identifiers in High-level language.

|	**Alternation and grouping.**

(ab|cd) . In this case parenthesis not needed for grouping. But in (ab|cd+)?(ef)* , it is necessary.
Does the above expression generate abefef, efefef, cdef, cddd, abc, abcdef ?
y,y,y,y,n,n.

<u>Context sensitivity</u>

^, $,/	**used to achieve this.**
^	**matches only at the beginning of a line.**
$	**at the end of the line**
/	**Trailing context, implements "followed by"**
	ab/cd

are ab$ and ab/\n equivalent ?

<>	**Start condition**
{ }	**Repetitions and definitions.**
	a{1,5} looks for 1 to 5 occurrence of a
	{digit} looks for a predefined string named digit and inserts
%	**At any point in the Lex input file, one can insert C code, either my identing it (so that it does not begin in column 1) or by preceding it with a line containing only the character pair %{ and %}.**

<u>Lex Actions.</u>

Default **Copies unmatched input to the output.**
[\t\n] ;
causes the three spacing characters (blank, tab, and newline) to be ignored.

YYTEXT:	**External variable that stores the matched string.**
YYLENG:	**Integer variable that stores the length of the string matched.**
[a-z]+	**printf("%s", yytext); Will print the matched string.**

ECHO does the same.

[a-z]+	**ECHO;**
[a-zA-Z]+	***{words++;chars+=yyleng;}*_____________________________**

YYMORE(): **Used to indicate that the next input expression recognized is to be tacked on to the end of this input.**

LEX

The absolute minimum Lex program %%.
Rules:
A table.
Left column - Regular expression
Right column - Action

Eg:

integer	**printf("found keyword INT");**
colour	**printf("color");**
mechanise	**printf("mechanize");**
petrol	**printf("gas");**

What if you had "Petroleum" ?

Lex Regular Expression

Letters and Digits -	**Match only themselves**
Operators-	**" \ [ ] ^ - ? . * + | ( ) $ / { } % < >.**

How do you use operators as a text ?

Use **a) quotations. xy"++" -> xy++.**
b) \ backslash. xy\+\+

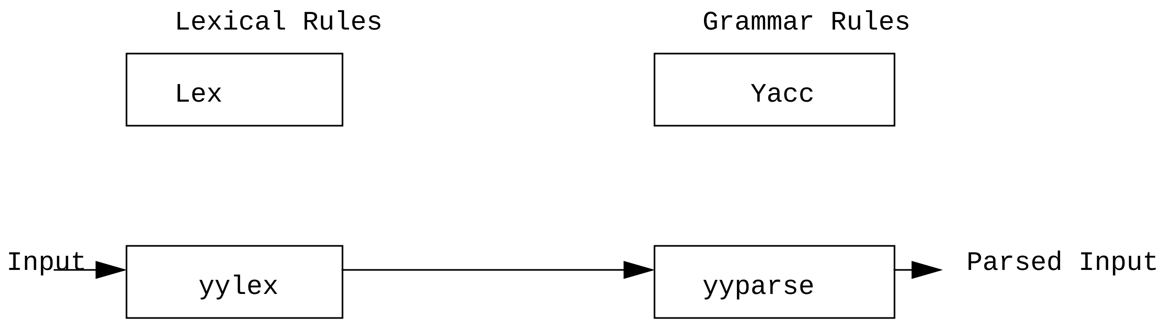
Is "xy++" identical to xy"++" ?

Normally, blanks or tab characters end a rule. How do you include blanks then in your rule?
Character classes: [ ]
Only 3 operators are special.
\ - and ^ .
- **Range [a-z0-9<>]**
What if [0-a] ? ______________________________________.
W i you need to include - in the character set ?
Ans: ***[-+0-9]***

^ **- Complements**
Always the first character after the left bracket.
Eg: **[^abc].**
[^a-zA-Z] ? ____

\ **Escape sequence.**

. **To match any operator, except a newline.**
? **Optional expression. Character preceding it is optional.**



General Format

```
            {definitions}
%%
{rules                    Actions}
%%
{user subroutines}
```

1. What is Lex and why do you need them ?
2. What is Yacc and why do you need them ?
3. How do you write a Lex program ?
- **Syntax**
- **Examples**
- **Compile**

4. How do you write a YACC program ?
- **Syntax**
- **Examples**
- **Compile**

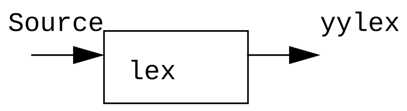
5. Lex and Yacc interaction

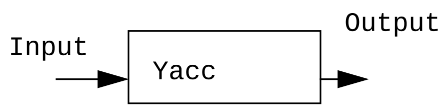
They are <u>PROGRAM GENERATORS.</u>

Lex - Lexical Analyzer.
YACC - Yet Another Compiler Compiler.
It examines the input for a pattern.
- **Tokens.**

Has
- **Rules / Grammars.**
- **Regular expressions specified by the user.**
- **Actions.**

How do they do it ?





Source:	**User specified rules.**
Input:	**Usually, a user abstract function.**
Output:	**Actions.**