

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 |
| | "Abcd" | STRING_CONST |
| | * | MULT_OP |
| | , | COMMA |
| | : | SEMICOLUMN |
| | (| LEFT_PAREN |

Lex is a tool for writing lexical analyzers.

2) Syntactic Analysis (Parsing):

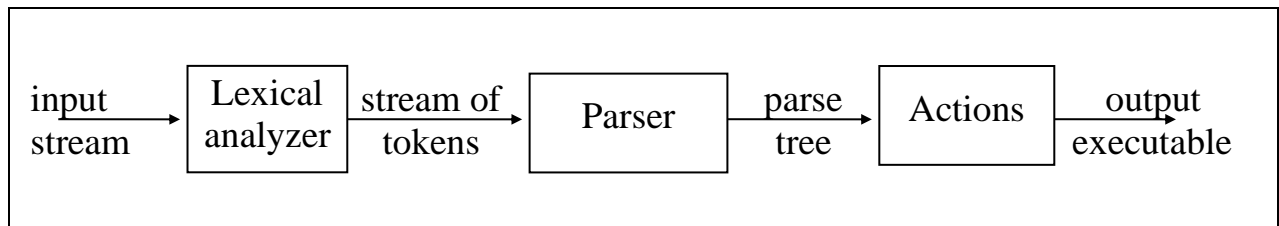
Parser: reads tokens and assembles them into language constructs using the grammar rules of the language.

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:

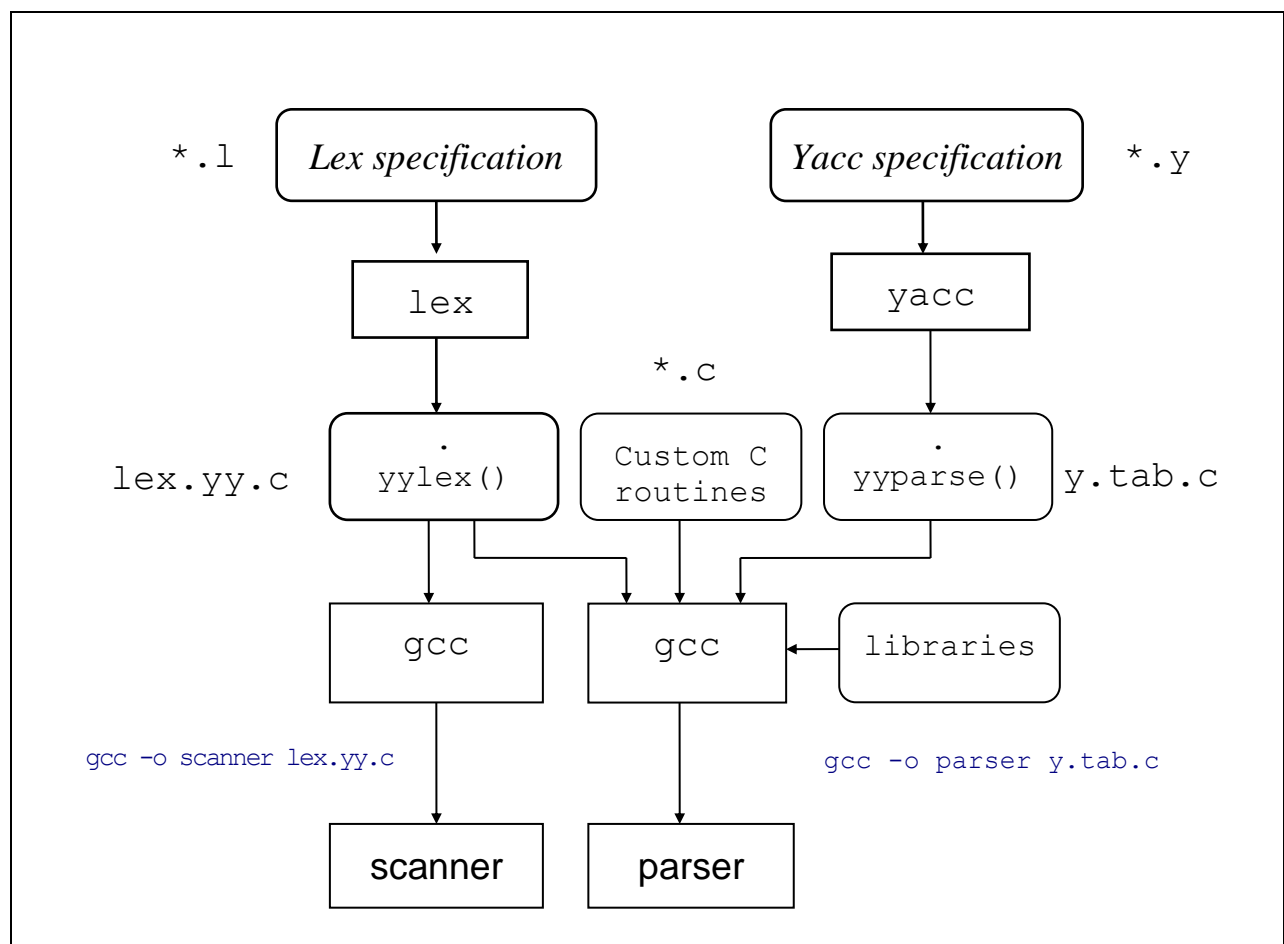


Lex: reads a specification file containing regular expressions and generates a C routine that performs lexical analysis.

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 dot 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
%%
funny printf("I recognized FUNNY");
$lex ex1.1
$ls
ex1.1 lex.yy.c
$gcc -o ex1 lex.yy.c
$ls
ex1 ex1.1 lex.yy.c
$emacs test
$cat test
fun
funny
ali is funny
and the course is fun

$cat test | ./ex1                    or $./ex1 < test
fun
I recognized FUNNY
Ali is I recognized FUNNY
this course is fun
```

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

```
$cat ex2.1
%option main
%%
fun    printf("FUN");
funny  printf("FUNNY");
```


| <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.l */
%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`), 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 | VELI → X. |
| HAMI BEY a | | HAMI BEY |

Definitions can be used in definitions

```
/* def-in-def.l */
%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 all of the rules that match the same number of characters, the rule given first in the file will be chosen.

Example,

```
/* rule-order.l */
%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

Note that we get a warning from lex, about this problem!

Important Lex Rules:

- 1) At any point in the input stream, the rule that matches the longest string is used.
- 2) If two or more rules match the same input string, the one given the earliest in the specification file is used

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> → 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"
int main() {
    return yyparse();
}
int yyerror( char *s ) { fprintf(stderr, "%s\n", s); }
```

If the input stream cannot be derived from the `start` variable, the default message of "syntax error" is printed and the 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"
void yyerror(char *s) { printf("%s, it is not in anbn\n", s); }
int 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"
void yyerror(char *s) { printf("%s, it is not in anbn_0\n", s); }
int 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: Simple adder

```

/* add.1 */
digit [0-9]
%%
{digit}+ {sscanf(yytext, "%d", &yylval);
          return(INT);
        }
\+      return(PLUS);
\n      return(NL);
.       ;
%%
int yywrap() { return 1; }

```

```

/* add.y */
/* L = {INT PLUS INT NL} */
%token INT PLUS NL
%%
add: INT PLUS INT NL { printf(" = %d\n", $1 + $3); }
%%
#include "lex.yy.c"
void yyerror(char *s) { printf("%s\n", s); }
int main() {
    return yyparse();
}

```

```

$ ./add
003 + 05
= 8
1+2
syntax error

```

Example: printing integers in a loop

```

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

```

```

/* print-int.y */
%token INTEGER NEWLINE
%%
lines: /* empty */
      | lines NEWLINE
      | lines value NEWLINE {printf(" =%d\n", $2);}
      | error NEWLINE {yyerror("! Reenter: "); yyerrok;}
      ;
value: INTEGER {$$ = $1;}
      ;
%%
#include "lex.yy.c"
void yyerror(char *s) { printf("%s", s); }
int main() {
    return yyparse();
}

```

Execution:

```

$./print-int
7
=7
007
=7
zippy
syntax error! Reenter: 0007
=7
^D

```

Keeping track of line numbers in the source:

```

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

```

```

/* print-int-wln.y */
/* prints integers with line numbers */
%token INTEGER NEWLINE
%%
lines: /* empty */
      | lines NEWLINE
      | lines line NEWLINE {printf("%d) %d\n", lineno, $2);}
      | error NEWLINE { printf(" in line %d!\nReenter: ", lineno);
                        yyerrok;
                      }
;
line: INTEGER {$$ = $1;}
%%
#include "lex.yy.c"
int lineno=0;
void yyerror(char *s) { printf("%s", s); }
int main() {
    return yyparse();
}

```

Execution:

```

$./print-int-wln
007
1) 7
jhg
syntax error in line 2!
Reenter: 66
3) 66
—

```

`yylval` can carry a value even when returning a `char`.

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.integer: 12
yytext = "+1.70", type of yylval: double, value of yylval.real: 1.7

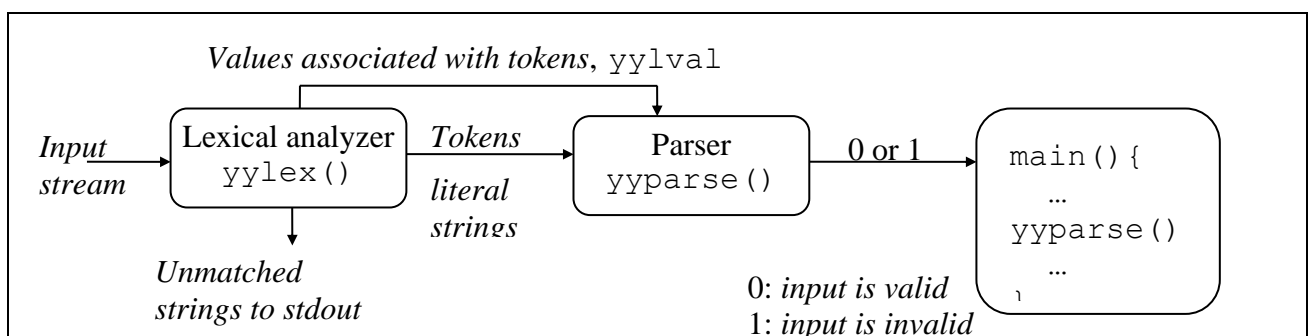
```

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

```

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

```



To return values, associated with tokens, from a lexical analyzer:

```
/* types.l */
alphabetic    [A-Za-z]
digit         [0-9]
alphanumeric  ({alphabetic}|{digit})
%%
[+-]?{digit}*{\.}?{digit}+      {sscanf(yytext, "%lf", &yyval.real);
                                return REAL;
                                }
{alphabetic}{alphanumeric}*      {strcpy(yyval.str, yytext);
                                return IDENTIFIER;
                                }
\<\-                               return ASSIGNOP;
\n                               return NL;
%%
int yywrap() { return 1; }
```

Type of variables can be defined by %type as

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

```
/* types.y */
%union{
    double  real;    /* real value */
    int      integer; /* integer value */
    char     str[30]; /* string value */
}
%token <real> REAL
%token <str> IDENTIFIER
%token ASSIGNOP NL
%type <real> assignment_stmt
%%
assignment_stmt: IDENTIFIER ASSIGNOP REAL NL {
                                $$ = $3;
                                printf("%s is assigned to %g\n", $1, $$);
                                }
%%
#include "lex.yy.c"
void yyerror(char *s) { printf("%s, it is not an assignment!\n", s); }
int main() {
    return yyparse();
}
```

```
[guvenir@dijkstra types]$ ./types
total <- -01.57
total is assigned to -1.57
^D
```

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

Actions between rule elements:

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

```
/* actions.y */
%{
#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(); }
```

```
actions: 14ab
          52673
actions  14aa
          526syntax error
actions  14ba
          syntax error
actions  14xyzafghbnm
          52673
```

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 where 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 pointers 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 to **merge** in a common subrule.

Example:

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

Initially there are two pointers, one in x, the other in y rules. After reading tokens 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;
```

No conflicts

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↑;
```

reduce/reduce conflict on C.

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.

Debugging:

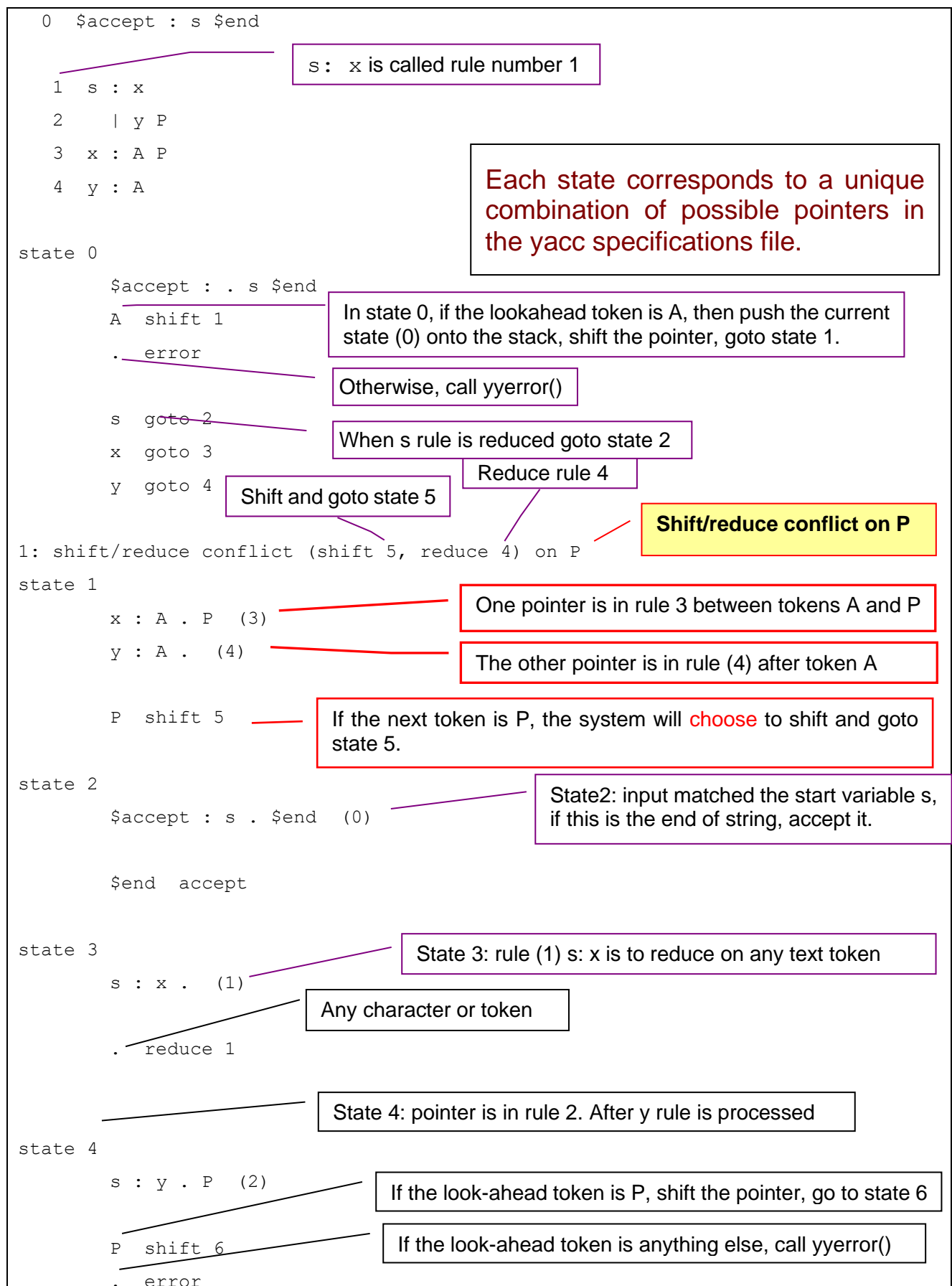
```
$yacc -v filename.y
```

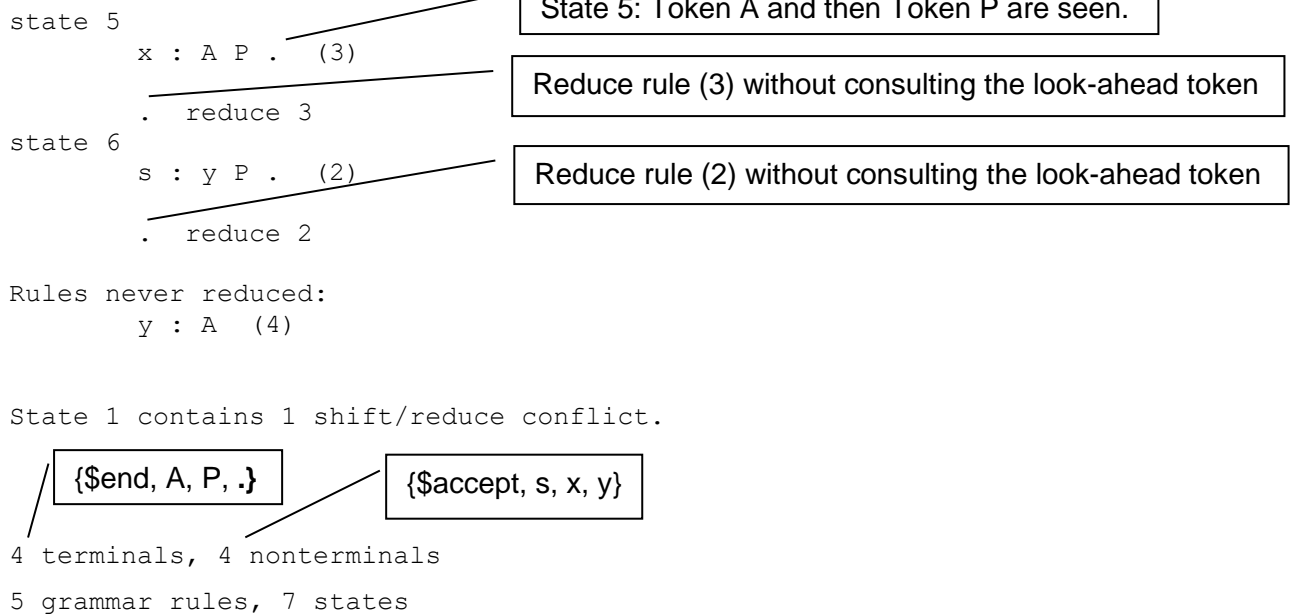
produces a file named y.output for debugging purposes.

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:





Recursive Rules:

Consider the following grammar:

```
/* recursive.y */
%token A
%%
s: A                // L={A, AAA, AAAAA, ...}, Not ambiguous !
  | A s A
;
```

y.output file:

```
0  $accept : s $end

1  s : A
2    | A s A
^L
state 0
    $accept : . s $end (0)

    A shift 1
    . error

    s goto 2          if the state machine pops back to this state,
                     the lookahead symbol is s, the parser will go to state 2

1: shift/reduce conflict (shift 1, reduce 1) on A
state 1
    s : A . (1)          reduce rule (1)
    s : A . s A (2)      shift in rule (2)

    A shift 1          if A, shift to state 1, that is, stay in the same state
    $end reduce 1      if $end, reduce rule 1

    s goto 3

...
```

However, the same language can also be represented by the following grammar, which **does not have any conflict**.

```
/* recursive.y */
%token A
%%
s: A                      // L = {A, AAA, AAAAA, ...}, Not ambiguous !
  | s A A
;
```

Actions on a Rule:

Actions can appear anywhere in the RHS of a rule.

However, for technical reasons, it is convenient for yacc to transform the grammar so that actions always appear at the very end.

For this reason, yacc introduces new variables, called *marker variables* (non-terminals), so that all actions are at the end of the rules.

Example,

Rule

```
a: {action1} b {action2} c {action3};
```

is replaced by

```
a: $S1 b $S2 c {action3};
$S1: {action1}; // Empty rules
$S2: {action2};
```

Example:

```
%token A B NL
%%
start: x | y;
x: A A NL ;
y: A B NL ;
```

Internally:

```
0 $accept : start $end
  1 start : x
  2       | y
  3 x : A A NL
  4 y : A B NL
```

No Conflict.

However, the equivalent following grammar

```
%token A B NL
%%
start: x | y;
x: {printf("using x");} A A NL ;
y: {printf("using y");} A B NL ;
```

Converted into:

```
0 $accept : start $end
1 start : x
2       | y
3 $$1 :
4 x : $$1 A A NL
5 $$2 :
6 y : $$2 A B NL
```

Conflict:

reduce/reduce conflict (reduce 3, reduce 5) on A

Make utility

Using the make utility on linux systems:

Contents of the file named Makefile:

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

On the command prompt, 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.

Bibliography

Saumya Debray “A Quick Introduction to Handling Conflicts in Yacc Parsers”
<https://www2.cs.arizona.edu/~debray/Teaching/CSc453/DOCS/conflicts.pdf>

Tom Niemann, “LEX & YACC TUTORIAL”,
<https://www.epaperpress.com/lexandyacc/>