

---

# **An Introduction to LEX and YACC**

# **Contents**

<b>1</b>	<b>General Structure</b>	<b>3</b>
<b>2</b>	<b>Lex - A lexical analyzer</b>	<b>4</b>
<b>3</b>	<b>Yacc - Yet another compiler compiler</b>	<b>10</b>
<b>4</b>	<b>Main Program</b>	<b>17</b>

# 1 General Structure

Program will consist of three parts:

1. lexical analyzer: `scan.l`
2. parser: `gram.y`
3. “Everything else”: `main.c`

## 2 Lex - A lexical analyzer

```
%{  
/* C includes */  
}%  
/* Definitions */  
%%  
/* Rules */  
%%  
/* user subroutines */
```

Figure 1: Lex program structure

## **Lex Definitions**

- Table with two columns:

1. regular expressions
2. actions

- ie:

```
integer    printf( "found keyword INT" );
```

- If action has more than one statement, enclose it within { }

## Regular Expressions

- text characters: a - z, 0 - 9, *space*...

`\n` : newline.

`\t` : tab.

- operators: " \ [ ] ^ - ? . \* + | ( ) \$ / { } % < >

`"..."` : treat '...' as text characters (useful for spaces).

`\` : treat next character as text character.

`.` : match anything.

- operators (cont):

[ . . . ] : match anything within [ ]

? : match zero or one time, eg:  $ab?c \rightarrow ac, abc$

\* : match zero or more times, eg:  $ab*c \rightarrow ac, abc, abbc\dots$

+ : match one or more times, eg:  $ab+c \rightarrow abc, abbc\dots$

( . . . ) : group ..., eg:  $(ab)^+ \rightarrow ab, abab\dots$

| : alternation, eg  $ab|cd \rightarrow ab, cd$

{ *n* , *m* } : repetition, eg  $a\{1, 3\} \rightarrow a, aa, aaa$

{ *defn* } : substitute *defn* (from first section).

## Actions

- `;` → Null action.
- `ECHO;` → `printf("%s", yytext);`
- `{...}` → Multi-statement action.
- `return yytext;` → send contents of `yytext` to the parser.

**yytext** : C-String of matched characters (Make a copy if necessary!)

**yylen** : Length of the matched characters.



Figure 2: LEX Template

```
%{ /* -*- c -*- */
#include <stdio.h>
#include "gram.tab.h"
%}
extern YYSTYPE yylval;
%%
[0-9]  {
        yylval.anInt = atoi((char *)&yyltext[0]);
        return INTEGER;
}

.      return *yyltext;
%%
```

## 3 Yacc - Yet another compiler compiler

```
%{  
/* C includes */  
}%  
/* Other Declarations */  
%%  
/* Rules */  
%%  
/* user subroutines */
```

Figure 3: Yacc program structure

## YACC Rules

- A grammar rule has the following form:

A : BODY ;

- A is a non-terminal name (LHS).
- BODY consists of names, literals, and actions. (RHS)
- *literals* are enclosed in quotes, eg: ' + '
- '\n' → newline.
- '\'' → single quote.

- The rules:

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

- can be specified as:

A	:	B
		C D
		E F G
		;

- Names representing *tokens* must be declared; this is most simply done by writing

```
%token    name1   name2   .   .   .
```

- Define name1, name2,... in the declarations section.
- Every name not defined in the declarations section is assumed to represent a nonterminal symbol.
- Every nonterminal symbol must appear on the left side of at least one rule.

## Actions

- the user may associate actions to be performed each time the rule is recognized in the input process, eg:

```
XXX : YYZ ZZZ
    { printf("a message\n"); }
    ;
```

- \$ is special!
  - $\$n$  → psuedo-variables which refer to the values returned by the components of the right hand side of the rules.
  - $\$\$$  → The value returned by the left-hand side of a rule.

```
expr : '(' expr ')' { $$ = $2 ; }
```

- Default return type is integer.

## Declarations

**%token** : declares ALL terminals which are not literals.

**%type** : declares return value type for non-terminals.

**%union** : declares other return types.

- the type

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

is generated and must be included into the lex source so that types can be associated with tokens.

Figure 4: YACC Template

```
%{  
#include <stdio.h>  
%}  
%token <anInt>    INTEGER  
%type <anInt>    S E  
%union {  
    int anInt;  
}  
%%  
S        : E  
          { printf( "Result is %d\n", $1 ); }  
          ;  
%%  
yyerror( char * s ) { fprintf( stderr, "%s\n", s ); }
```



## 4 Main Program

Figure 5: Main template

```
#include <stdio.h>
#include <stdlib.h>
extern int yyerror(), yylex();

#define YYDEBUG 1
#include "gram.tab.c"
#include "lex.yy.c"

main()
{
/*      yydebug = 1; */
    yyparse();
}
```

Figure 6: Running the example

```
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

C:\DJGPP\etc\SYSC-3~1>bison -d gram.y

C:\DJGPP\etc\SYSC-3~1>flex scan.l

C:\DJGPP\etc\SYSC-3~1>gcc main.c

C:\DJGPP\etc\SYSC-3~1>echo 1 + 2 | a.exe
Result is 3

C:\DJGPP\etc\SYSC-3~1>
```