



Table of contents

Example program for the lex and yacc programs



This section contains example programs for the **lex** and **yacc** commands.

Together, these example programs create a simple, desk-calculator program that performs addition, subtraction, multiplication, and division operations. This calculator program also allows you to assign values to variables (each designated by a single, lowercase letter) and then use the variables in calculations. The files that contain the example **lex** and **yacc** programs are as follows:

File	Content
calc.lex	Specifies the lex command specification file that defines the lexical analysis rules.
calc.yacc	Specifies the yacc command grammar file that defines the parsing rules, and calls the yylex subroutine created by the lex command to provide input.

The following descriptions assume that the **calc.lex** and **calc.yacc** example programs are located in your current directory.

Compiling the example program

To create the desk calculator example program, do the following:

1. Process the **yacc** grammar file using the **-d** optional flag (which informs the **yacc** command to create a file that defines the tokens used in addition to the C language source code):

```
yacc -d calc.yacc
```

[Copy to clipboard](#)

2. Use the **ls** command to verify that the following files were created:

y.tab.c

The C language source file that the **yacc** command created for the parser

y.tab.h

A header file containing define statements for the tokens used by the parser

3. Process the **lex** specification file:

```
lex calc.lex
```

[Copy to clipboard](#)

4. Use the **ls** command to verify that the following file was created:

lex.yy.c

The C language source file that the **lex** command created for the lexical analyzer

5. Compile and link the two C language source files:

```
cc y.tab.c lex.yy.c
```

Copy to clipboard

6. Use the **ls** command to verify that the following files were created:

y.tab.o

The object file for the **y.tab.c** source file

lex.yy.o

The object file for the **lex.yy.c** source file

a.out

The executable program file

Table of contents

run the program directly from the **a.out** file, type:

a.out

Copy to clipboard

OR

To move the program to a file with a more descriptive name, as in the following example, and run it, type:

```
$ mv a.out calculate
```

```
$ calculate
```

Copy to clipboard

In either case, after you start the program, the cursor moves to the line below the \$ (command prompt). Then, enter numbers and operators as you would on a calculator. When you press the Enter key, the program displays the result of the operation. After you assign a value to a variable, as follows, the cursor moves to the next line.

```
m=4 <enter>
```

```
-
```

Copy to clipboard

When you use the variable in subsequent calculations, it will have the assigned value:

```
m+5 <enter>
```

```
9
```

```
-
```

Copy to clipboard

Parser source code

The following example shows the contents of the **calc.yacc** file. This file has entries in all three sections of a **yacc** grammar file: declarations, rules, and programs.

```
%{
#include<stdio.h>
```

```
int regs[26];
int base;
```

```
%start list
```

```
%union { int a; }
```



Table of contents

```
token DIGIT LETTER
```

```
left '|'
```

```
left '&'
```

```
left '+' '-'
```

```
left '*' '/' '%'
```

```
left UMINUS /*supplies precedence for unary minus */
```

```
%% /* beginning of rules section */
```

```
list: /*empty */
```

```
|
```

```
list stat '\n'
```

```
|
```

```
list error '\n'
```

```
{
```

```
yyerrok;
```

```
}
```

```
;
```

```
stat: expr
```

```
{
```

```
printf("%d\n", $1);
```

```
}
```

```
|
```

```
LETTER '=' expr
```

```
{
```

```
regs[$1.a] = $3.a;
```

```
}
```

```
;
```

```
expr: '(' expr ')'
```

```
{
```

```
$$ = $2;
```

```
}
```

```
|
```

```
expr '*' expr
```

```
{
```

```
$$ .a = $1 .a * $3 .a;
```

```
}
```

```
|
```

```
expr '/' expr
```

```
{
```

```
$$ .a = $1 .a / $3 .a;
```

```

    $$$.a = $1.a / $3.a;
}
|
expr '%' expr
{
    $$$.a = $1.a % $3.a;
}
|
expr '+' expr
{
    $$$.a = $1.a + $3.a;
}
|
expr '-' expr
{
    $$$.a = $1.a - $3.a;
}
|
expr '&' expr
{
    $$$.a = $1.a & $3.a;
}
|
expr '|' expr
{
    $$$.a = $1.a | $3.a;
}
|

'-' expr %prec UMINUS
{
    $$$.a = -$2.a;
}
|
LETTER
{
    $$$.a = regs[$1.a];
}
|
number
;

```

```

number: DIGIT
{
    $$ = $1;
    base = ($1.a==0) ? 8 : 10;
}
|
number DIGIT
{
    $$$.a = base * $1.a + $2.a;
}

```

Table of contents

>

```

    }
    ;

%%
main()
{
    return(yyparse());
}

yyerror(s)
char *s;
{
    fprintf(stderr, "%s\n",s);
}

yywrap()
{
    return(1);
}

```

Copy to clipboard

The file contains the following sections:

- **Declarations section.** This section contains entries that:
 - Include standard I/O header file
 - Define global variables
 - Define the `list` rule as the place to start processing
 - Define the tokens used by the parser
 - Define the operators and their precedence
- **Rules section.** The rules section defines the rules that parse the input stream.
 - **%start** - Specifies that the whole input should match **stat**.
 - **%union** - By default, the values returned by actions and the lexical analyzer are integers. **yacc** can also support values of other types, including structures. In addition, **yacc** keeps track of the types, and inserts appropriate union member names so that the resulting parser will be strictly type checked. The **yacc** value stack is declared to be a union of the various types of values desired. The user declares the union, and associates union member names to each token and nonterminal symbol having a value. When the value is referenced through a `$$` or `$n` construction, **yacc** will automatically insert the appropriate union name, so that no unwanted conversions will take place.
 - **%type** - Makes use of the members of the **%union** declaration and gives an individual type for the values associated with each part of the grammar.
 - **%tokens** - Lists the tokens which come from lex tool with their type.
- **Programs section.** The programs section contains the following subroutines. Because these subroutines are included in this file, you do not need to use the **yacc** library when processing this file.

Subroutine	Description
main	The required main program that calls the yyparse subroutine to start the program.
yyerror(s)	This error-handling subroutine only prints a syntax error message.
yywrap	The wrap-up subroutine that returns a value of 1 when the end of input occurs.

Lexical analyzer source code

This file contains include statements for standard input and output, as well as for the **y.tab.h** file. If you use the **-d** flag with the **yacc** command, the **yacc** program generates that file from the **yacc** grammar file information. The **y.tab.h** file contains definitions for the tokens that the parser program uses. In addition, the **calc.lex** file contains the rules to generate these tokens from the input stream.

The following are the contents of the **calc.lex** file.

```
%
%
#include <stdio.h>
#include "y.tab.h"

int c;

%%

" "      ;

[a-z]    {
    c = yytext[0];
    yylval.a = c - 'a';
    return(LETTER);
}

[0-9]    {
    c = yytext[0];
    yylval.a = c - '0';
    return(DIGIT);
}

[^a-z0-9\b] {
    c = yytext[0];
    return(c);
}


%%
```

Copy to clipboard

Please note that DISQUS operates this forum. When you sign in to comment, IBM will provide your email, first name and last name to DISQUS. That information, along with your comments, will be governed by [DISQUS' privacy policy](#). By commenting, you are accepting the [IBM commenting guidelines](#) and the [DISQUS terms of service](#).




Sign In

0 Comments IBM Knowledge Center

 Recommend  Share

Sort by Best ▾

Nothing in this discussion yet.

 Subscribe  Add Disqus to your siteAdd DisqusAdd  Privacy

[Contact](#) [Privacy](#) [Terms of use](#) [Accessibility](#) [Feedback](#)

English ▾

Table of contents >