# UCS1602 - COMPILER DESIGN

YACC Tool

# Session Meta Data

| | |
|---|---|
| **Author** | **Dr. D. Thenmozhi** |
| Reviewer | |
| Version Number | 1.2 |
| Release Date | 26 February, 2021 |

# Session Objectives

- Understanding YACC tool: parser generator

# Session Outcomes

- At the end of this session, participants will be able to
  - Write programs using YACC tool
  - Parse using YACC tool

# Agenda

- YACC Tool
  - Format
  - Rules
  - Example



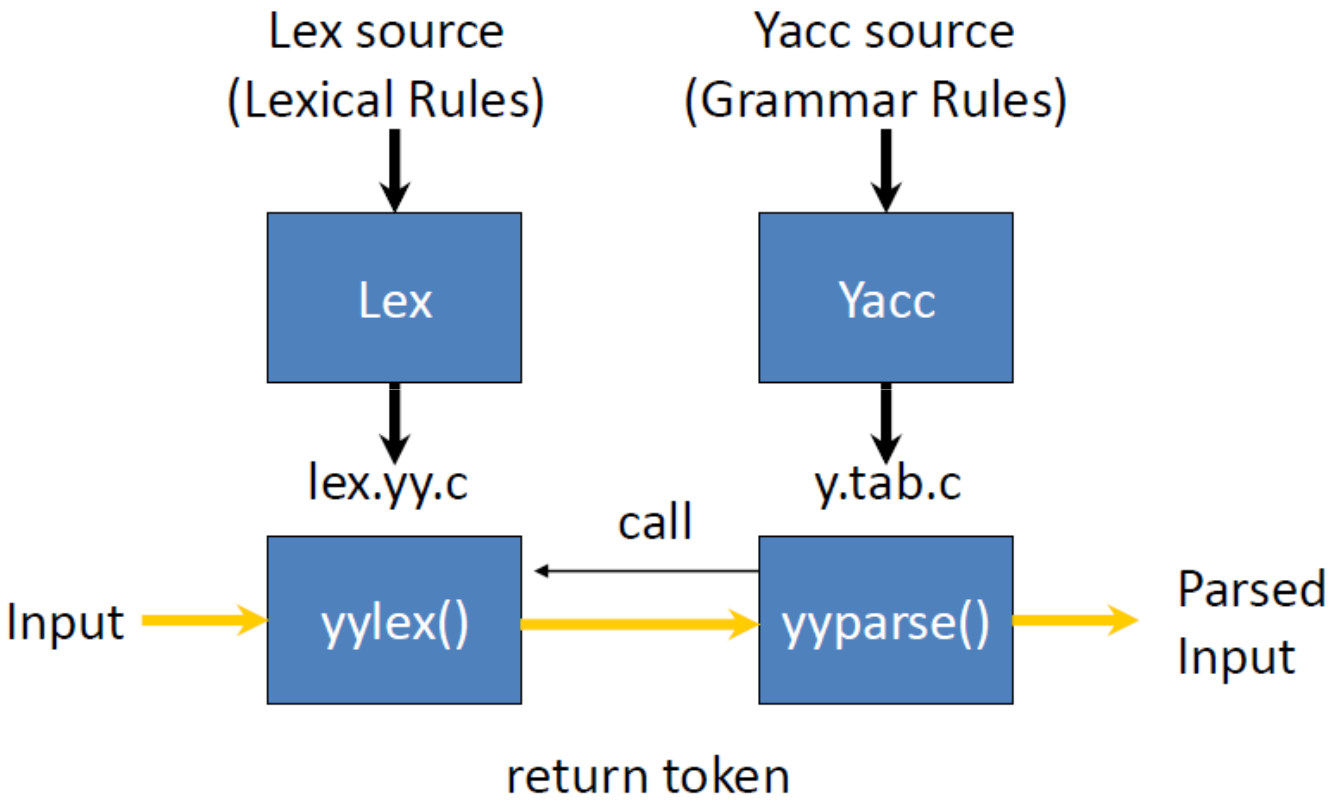
# Introduction

- What is **YACC ?**

  –**Tool which will produce a parser for a given grammar.**

  –YACC (Yet Another Compiler Compiler) is a program designed to compile a LALR(1) grammar and to produce the source code of the syntactic analyzer of the language produced by this grammar.

# Lex with Yacc



# YACC file format

```
%{
        C declarations
%}
        yacc declarations
%%
        Grammar rules
%%
        Additional C code
```

– **Comments enclosed in /* ... */ may appear in any of the sections.**

# Definition Section

```
%{
#include <stdio.h>
#include <stdlib.h>
%}
%token ID NUM
%start expr
```

It is a terminal



# Start Symbol

- The first non-terminal specified in the grammar specification section.
- To overwrite it with %start declaration.

```
%start  non-terminal
```



# Rule section

- This section defines grammar
- Example

```
expr : expr '+' term | term;
term : term '*' factor | factor;
factor : '(' expr ')' | ID | NUM;
```



# Rule section

- Normally written like this
- Example:

```
expr    : expr '+' term
        | term
        ;
term    : term '*' factor
        | factor
        ;
factor  : '(' expr ')'
        | ID
        | NUM
        ;
```



# The position of rules

```
expr : expr '+' term     { $$ = $1 + $3; }
     | term              { $$ = $1; }
     ;
term : term '*' factor   { $$ = $1 * $3; }
     | factor            { $$ = $1; }
     ;
factor : '(' expr ')'    { $$ = $2; }
       | ID
       | NUM
       ;
```

# YACC declaration summary

**`%start'**

Specify the grammar's start symbol

**`%union'**

Declare the collection of data types that semantic values may have

**`%token'**

Declare a terminal symbol (token type name) with no precedence or associativity specified

**`%type'**

Declare the type of semantic values for a nonterminal symbol



# YACC declaration summary

**`%right'**

Declare a terminal symbol (token type name) that is right-associative

**`%left'**

Declare a terminal symbol (token type name) that is left-associative

**`%nonassoc'**

Declare a terminal symbol (token type name) that is nonassociative (using it in a way that would be associative is a syntax error, ex: x *op*. y *op*. z is syntax error)

# Example – Calc.y

```
%{
#include <stdlib.h>
#include <stdio.h>
int yylex(void);
#include "y.tab.h"
%}

%token INTEGER

%%
program:
 line program
 | line

line:
 expr '\n' { printf("%d\n",$1); }

expr:
 expr '+' mulex { $$ = $1 + $3; }
 | expr '-' mulex { $$ = $1 - $3; }
 | mulex { $$ = $1; }

mulex:
 mulex '*' term { $$ = $1 * $3; }
 | mulex '/' term { $$ = $1 / $3; }
 | term { $$ = $1; }
```

```
term:
'(' expr ')' { $$ = $2; }
| INTEGER { $$ = $1; }

%%
void yyerror(char *s)
{
fprintf(stderr,"%s\n",s);
return;
}
yywrap()
{
  return(1);
}
```

```
int main(void)
{
yyparse();
return 0;
}
```

# Lexx.l

```
%{
#include <stdlib.h>
#include <stdio.h>
#include "y.tab.h"
void yyerror(char*)
extern int yylval;
%}

%%
[ \t]+ ;
[0-9]+ {yylval = atoi(yytext);
 return INTEGER;}
[-+*/] {return *yytext;}
"(" {return *yytext;}
")" {return *yytext;}
\n {return *yytext;}
. {char msg[25];
 sprintf(msg,"%s <%s>","invalid character",yytext);
 yyerror(msg);}
```

**Output :-**

```
lex lexx.l
yacc -d calc.y
gcc y.tab.c lex.yy.c
./a.out
```

# Summary

- YACC tool
  - Yet Another Compiler Compiler) to produce the source code of the syntactic analyzer of the language produced by this grammar.
- Structure of YACC program
  - Definition section
  - Rule section

# Check your understanding

- Write a YACC program for desk calculator